EE3980 Algorithms

Homework 11. Sudoku

By 105061212 王家駿

1. Introduction

In this homework, we solve the Sudoku puzzle by the backtracking algorithm, and find all feasible solution. Then, we repeat the solving for 100 times, and record the average CPU time used. Furthermore, we estimate the complexity of the algorithm and compare the time between different input data set.

2. Implementation

For the input data, there is no more than $9 \times 9 = 81$ blank spaces to be filled. Thus, we could try all the possibility for the problem. Since we can fill a blank space with nine different numbers, there are at most $9^{81} \cong 10^{77}$ different possible solutions for the problem.

However, the restriction of Sudoku is strict, so we could expect that there are only few feasible solutions. Thus, traversing all possible solutions seems not effective, so we use the backtracking method, continue solving the next blank space only when the stage so far fits the restriction, to reduce the CPU running time.

2.1. Data Structure

At first, we read the Sudoku matrix and convert into a two-dimensional array, where the blank spaces are replaced by 0's, so we could get an array with only numbers 0~9 in it.

In addition, to record the places of the blank spaces, we construct an array storing the row and column index of each blank space. Since there are at most 81 blanks in a Sudoku, we allocate 81 spaces to record the information, and also the size of the array (the number of blank spaces). By this step, we could easily know the place of the next blank when traversing the matrix.

2.2. placeable

```
1.  Algorithm placeable(A, r, c, value)
2.  {
3.      // check the row
4.      for i := 1 to 9 do
5.          if (A[r][i] = value) then return false;
6.
7.      // check the column
8.      for i := 1 to 9 do
9.          if (A[i][c] = value) then return false;
10.
11.     // calculating the starting point of the 3x3 submatrix
12.     row_start := ((r - 1) / 3) * 3 + 1;
13.     col_start := ((c - 1) / 3) * 3 + 1;
14.
15.     // check the submatrix
16.     for i := row_start to row_start + 2 do
17.         for j := col_start to col_start + 2 do
18.             if (A[i][j] = value) then return false;
19.
```

```
20.     // placeable
21.     return true;
22. }
```

Every time when try to place a number to a blank space, we have to check if the number placed meets the criterion of Sudoku. Thus, in this function, we check if the number tried could lead to a feasible solution.

The constraints of Sudoku are:

(1) Each row has no duplicated numbers.

(2) Each column has no duplicated numbers.

(3) Each 3 by 3 submatrix has no duplicated numbers.

So, we check if the value meets the three constraints above. First, we iterate through the row to check if there is the same value placed before. Then, we iterate through the column to check the same thing. Next, we find the submatrix where the block belongs to. It could be calculated by the integer operations in C language, and then iterate through the submatrix to check if there is the same value. After these steps, if we didn’t find a block with the same value, we could say that the value in the block is placeable.

In this function, we would check 27 blocks, 9 for the row, 9 for the column, and 9 for the submatrix. Thus, the block visited is independent to the input array, so the function takes a constant time to find if the value is placeable.

2.3. Backtracking

```
1.  Algorithm backtracking(void)
2.  {
3.      // initialize
4.      k := 1;
5.      r := kth row_data;
6.      c := kth col_data;
7.
8.      while (k > 0) do {                      // if there is still blank space
9.          A[r][c] := A[r][c] + 1;             // try new value
10.         while (A[r][c] <= 9) do {           // can find value
11.             if (placeable(A, r, c, A[r][c]) then {
12.                 // one solution found
13.                 if (k is the last blank) then print out result;
14.                 else {
15.                     k := k + 1;             // keep finding next element
16.                     r := kth row_data;
17.                     c := kth row_data;
18.                     A[r][c] := 0;           // initialize to 0
19.                 }
20.             }
21.             A[r][c] := A[r][c] + 1;         // try new value
22.         }
23.         k := k - 1;                         // backtrack the previous
24.         r := kth row_data;
25.         c := kth col_data;
26.     }
27. }
```

We use the backtracking algorithm to solve all feasible solution of Sudoku. That is, in every step, we continue to find for the next blank only when the current number is placeable. If all possibility is performed in the form of tree branches, the traversing steps would act like depth first search (DFS) algorithm.

At first, we start from the first blank space, then keep trying a new value (from 1 to 9) of the blank. If the value is placeable, we make the index k plus 1 in line 15 to check the next blank. If we tired all value and no one is placeable, we make k minus by 1 in line 23 to backtrack to the previous blank. We would keep repeating the steps above until k $\leqq$ 0, which means that all possibility had tried. If we iterate to the last blank and the value is feasible, the whole array is the solution of the Sudoku because the former values must be feasible so that we could reach the last element.

The time complexity depends on the placeable function in line 11. The worst case is that we must traversing all possibility. With 9 possible numbers and n blank spaces, the worst case leads to change the value of 9^n times. And since we know that the time complexity of placeable function is constant, the worst-case time complexity of this problem is $O(9^n)$.

Since we use the backtracking algorithm, we could eliminate some non-feasible solution to some extent. Suppose that one-third blocks had filled and distributed

evenly with numbers in the beginning, we could eliminate one-third possibilities for the first blank space, and so for the rest blanks. Thus, we could reduce the time complexity to at most $O(6^n)$. The more numbers we fill during the solving, the more constraints for the current value. So, the possibilities would be bound further for the last few blanks. Therefore, we could effectively reduce the actual running time by the backtracking algorithm.

3. Executing results

We run the program for 100 times, and then record the results of the solutions of Sudoku and average CPU time used.

No.	Number of blanks	Number of solutions	Average CPU time used
1	45	1	0.092ms
2	45	1	0.111ms
3	49	1	0.376ms
4	53	1	3.482ms
5	57	1	7.917ms
6	57	1	31.11ms
7	59	1	508.7ms
8	57	1	9.584ms
9	57	1	68.08ms

4. Result analysis and conclusion

From the list, we could observe that the CPU time used increases for a larger number of blanks. It's reasonable since we have more possible solutions that increase the depth of backtracking.

However, for the case 5, 6, 8, and 9, they share the same input data size 57, while the time used are ranging from about 8 to 68 milliseconds, which is a large different. There must be something related to the time cost besides the input blank numbers.

As we mentioned above, the number of possibilities depends on the placeable function. So, the CPU time used is related to the structure of Sudoku. If we apply the backtracking from the blank space which has the most constraints, we could eliminate more possibilities at the beginning. Then for the next step, we could also pick the rest blank with the most constraints. By this kind of operation, we could reduce the possibility at first as more as we can.

The blank picking method is like the method when a human solves the Sudoku. We often fill the blank whose value can be sure first. That is, the constraint of this block is as more to reduce the possibility to only one number. Yet, when we implement this method in computer programming, we have to check through the whole matrix to find the block with most constraints in every step, which would lead

to additional time complexity. So, it might not cost less time than the former algorithm.

If we want to use branch and bound algorithm to solve the problem, we have to estimate the bound in each step, and decide if the next step is feasible by the bound. In the lecture, the knapsack problem and the traveling salesperson problem are solving for its extremum, and the value could be estimated by the rest data. However, for the Sudoku problem, it’s hard to define and quantify the bound of the problem, so does to estimating the bound by the rest data. Thus, it’s hard to solve the problem by branch and bound algorithm. If we want to use it, we have to convert the problem to some mathematical models in advance.