제 26 회 한국정보처리학회 추계학술발표대회 논문집 제 13 권 제 2 호 (2006. 11)

Sudoku 퍼즐의 구속조건만족문제 해법

이승원, 최호진
한국정보통신대학교 공학부
대전광역시 유성구 문지동 103-6
E-mail: {citizen10, hjchoi}@icu.ac.kr

Solving Sudoku as Constraint Satisfaction Problem

Seungwon Lee and Ho-Jin Choi
School of Engineering, Information and Communications University (ICU)
Yuseong P.O Box 77, Daejeon, 305-600, Korea
E-mail: {citizen10, hjchoi}@icu.ac.kr

Abstract

This paper presents solving the Sudoku puzzle as a constraint satisfaction problem (CSP). After introducing the rules and characteristics of the puzzle, we formulate the puzzle as a CSP and develop various methods of solving the problem. Blind search, minimum remaining value (MRV) heuristic, and some advanced methods are investigated, and their algorithms are implemented in this undergraduate project. The performance comparisons of these methods are discussed in the paper.

1. Introduction

Sudoku [2] is a numeric puzzle with 9 × 9 square. A Sudoku problem starts from a partially filled square (as illustrated in Figure 1). The aim of this puzzle is to fill the whole 81 cells. The rule is that for each row, column, and 3 × 3 box (indicated by thick boundary lines in the figures), each number of 1 to 9 must appear exactly once (as illustrated in Figure 2).

	6		1		4		5	
		8	3		5	6		
2								1
8			4		7			6
		6				3		
7			9		1			4
5								2
		7	2		6	9		
	4		5		8		7	

(Figure 1) Example of a Sudoku problem

The example shown in Figure 1 is a Sudoku puzzle that has exactly one solution. However, there exist puzzles that have no solution, or ones with multiple solutions.

There is no solution for a Sudoku puzzle if one of the two situations occurs: (1) Any number of 1~9 cannot fill a specific blank, or (2) A specific number cannot be held by one of rows/columns/boxes.

9
1
2
8
4
7
5
3
6

9	6	3	1	7	4	2	5	8

9	6	3
1	7	8
2	5	4

(Figure 2) A column, a row, and a 3 × 3 box

The aim of this project is to solve the Sudoku puzzle as a constraint satisfaction problem (CSP). In this paper we will formulate the puzzle as a CSP and develop various methods of solving the problem. Blind search, minimum remaining value (MRV) heuristic, and some other methods will be investigated, and their performance will be compared.

2. Sudoku as a Constraint Satisfaction Problem

A CSP consists of a set of 'variables' and a set of 'constraints'. All variables have their own possible values that do not violate constraints. The aim of a CSP is to find a solution which satisfies all the constraints [1].

A Sudoku problem is a kind of CSP. The variables of a problem are blanks in the 9 × 9 square. The constraint is that the same two numbers cannot exist on the same row, column, or in the same box.

When finding a solution of a CSP, the method called 'backtracking search' is used. This method chooses one value for one variable at a time, and backtracks if there is a variable that does not have any legal value. For a CSP, three heuristics can be used in the backtracking method [1].

1) Minimum remaining values (MRV) heuristic ("most constrained valuable" "fail-first"): it chooses the variable with the fewest possible values. In a Sudoku problem, this algorithm chooses the blank that has the least number of available digits.

2) Degree heuristic ("most constraining variable"): it chooses the variable making the largest number of constraints on other unassigned variables. In a Sudoku problem, this algorithm chooses the blank that removes the largest number of available digits in other blanks.

3) Least-constraining-value heuristic: it chooses a value that removes the fewest possible values for the remaining variables. This heuristic tries to leave flexibility as much as possible. In a Sudoku problem, when filling a blank, this algorithm chooses the digit that makes the least restrictions for available digits of the other blanks.

The Sudoku-solving program in this project will use the MRV heuristic, because the constraint (availability of digits that fill the blank) for each individual blank is easy to recognize.

3. Solving the Sudoku Puzzle

3.1. Using blind backtracking search

The program searches a blank which is not filled. After finding a blank, the program saves the current state first, and it fills the blank with one of the legal values (digits).

When the program meets a dead end (Any number cannot fill one blank, or a specific digit cannot be held by one row/column/box), it goes back to the most recent state before filling a blank. The digit originally filled the blank is now disabled, and another legal digit fills the blank. If there is no remaining available digit, then the program goes back again to the most recent state.

(Algorithm 1)
```
while a solution is not fount yet do
  save the current state
  if there is a blank that can hold a number then
    for each n 1...9
      if the blank can hold n
        then fill the blank with n
    count = count + 1
  if the program meets a dead end then
    while there is no remaining digit for the blank do
      if count is 0
        then the problem has no solution
      else count = count - 1
        backtrack to the most recent state
        change the number put in the blank
```

3.2. Using minimum-remaining value (MRV) heuristic

The backtracking search method using MRV is similar to the blind backtracking search, but MRV heuristic chooses the blank that has the fewest legal digits.

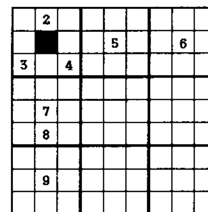
In addition, MRV searches blanks with just one available digit first. When filling these blanks, the process of saving current state can be omitted, because contents these blanks cannot be changed.

3.3. Advanced solving mechanisms of Sudoku problems

When people solve a Sudoku problem, he or she does not use the MRV heuristic only. He/she uses various methods to fill blanks in the problem. There are a lot of methods to fill blanks in a Sudoku problem, but they can be categorized by 5 different mechanisms. Case A and B are the mechanisms of filling blanks with given constraints. Case C, D and E are the mechanisms of deriving another constraint from given constraints.

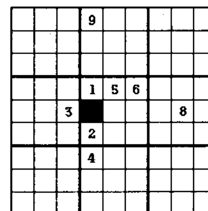
If the program uses these various methods before backtracking, then it will be able to reduce the number of backtracking times.

3.3.1 Case A: when only one specific number can fill the blank



(Figure 3) Example of the case A [3]

In this example, 2, 3, 4, 5, 6, 7, 8, and 9 cannot fill the colored blank. (2, 3, and 4 are in the same box; 5 and 6 are on the same row; 7, 8, and 9 are on the same column). Therefore, the red blank should be filled with 1.



(Figure 4) Another example of the case A

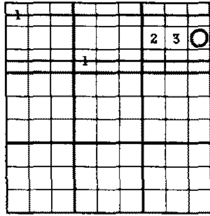
In this example, 1, 2, 3, 4, 5, 6, 8, and 9 cannot fill the colored blank. (1, 2, 5 and 6 are in the same box; 3 and 8 are on the same row; 4 and 9 are on the same column) Therefore, the digit that fills the red blank is 7.

In the case A, the program must find a blank which has only one available digit. Then the program fills the blank with that available digit.

(Algorithm 2)
```
for each row 1...9
  for each column 1...9
    if the cell (row, column) is not filled yet then
      if there is only one available digit for the blank
        then fill the blank with that available digit
```

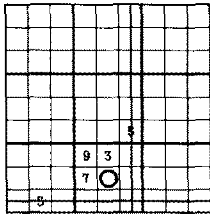
Case A is also covered in the MRV heuristic, because MRV fills the blanks with just one legal digit first.

3.3.2 Case B: when only one blank can hold a specific number



(Figure 5) Example of the case B [3]

In this example, one of the seven blanks in the upper-rightmost box must hold '1'. However, only the circled blank can hold the digit 1. Therefore, the circled blank must be filled with 1.



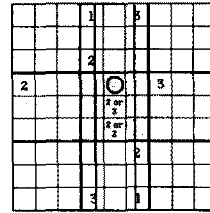
(Figure 6) Another example of the case B

In this example, six colored blanks are in the same box. One of these blanks should hold '5', but only the circled blank can hold 5. That blank must hold 5.

In the case B, the program checks 'necessary digits' of the rows, columns, and boxes. If there is only one blank that can hold the necessary digit, then that blank is filled with the digit.

(Algorithm 3)
```
for each x 1...9
  for each n 1...9
    if the xth row does not contain the digit 'n' yet then
      if there is only 1 blank that can hold n in the row
        then fill the blank with 'n'
    if the xth col does not contain the digit 'n' yet then
      if there is only 1 blank that can hold n in the col
        then fill the blank with 'n'
    if the xth box does not contain the digit 'n' yet then
      if there is only 1 blank that can hold n in the box
        then fill the blank with 'n'
```

3.3.3 Case C: when several blanks can hold a specific number, but these blanks (except one) must hold other numbers



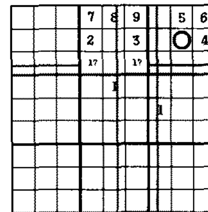
(Figure 7) Example of the case C [3]

In this example, in the central box, 3 blanks can hold '1'. However, two of these blanks are the only that can hold '2' and '3' in the box. These two blanks must hold 2 and 3, so the remaining blank (circled blank) should hold 1.

In the case C, the program checks if there are blanks that must hold two specific digits. Then the program makes the other numbers not available to those blanks.

(Algorithm 4)
```
for each row 1...9
  for each n1 1...9, n2 1...9
    if only two blanks on the row can hold n1, n2
      then the other numbers cannot fill these 2 blanks
for each column 1...9
  for each n1 1...9, n2 1...9
    if only two blanks on the column can hold n1, n2
      then the other numbers cannot fill these 2 blanks
for each box 1...9
  for each n1 1...9, n2 1...9
    if only two blanks on the box can hold n1, n2
      then the other numbers cannot fill these 2 blanks
```

3.3.4 Case D: when several blanks can hold a specific number, but these blanks (except one) cannot hold that number because of other blanks



(Figure 8) Example of the case D

In this example, three blanks can contain 1 in the upper rightmost box. Nevertheless, the two yellow blanks cannot be filled with 1. Why?

In the box which already contains 2, 3, 7, 8 and 9, one of the two blanks on the third row must be filled with 1. Therefore, the other blanks of the third row cannot hold 1. The two yellow blank is on the third row, so they cannot hold 1. In conclusion, the red blank must be filled with 1.

In the case D, the program checks if there are only two blanks that can hold a specific digit. Also, if the two blanks are on the same row/column/box, the program makes the other blanks on the same row/column/box not hold that digit.

(Algorithm 5)

```
for each box 1...9
  for each n 1...9
    if only two blanks in the box can hold n then
      if those two blanks are on the same row
        then the other blanks on that row cannot hold n
      if those two blanks are on the same col
        then the other blanks on that col cannot hold n
for each row 1...9
  for each n 1...9
    if only two blanks on the row can hold n then
      if those two blanks are on the same col
        then the other blanks on that col cannot hold n
      if those two blanks are in the same box
        then the other blanks in that box cannot hold n
for each column 1...9
  for each n 1...9
    if only two blanks on the column can hold n then
      if those two blanks are on the same row
        then the other blanks on that row cannot hold n
      if those two blanks are in the same box
        then the other blanks on that box cannot hold n
```

3.3.5 Case E: when only two specific numbers are available for two specific blanks

(Figure 9) Example of the case E
(partially solved problem from one in Chosun Ilbo, 06/29/2006 [4])

The two colored blanks must be filled with one of two numbers: 1 or 9. After the two yellow blank is filled with 1 and 9, the other blanks on the same column cannot hold 1 or 9.

The blank above those two blanks can hold one of three numbers: 1, 7, or 9. However, the three blanks are on the same column. After filling the two colored blanks with 1 and 9, we cannot fill the above blank with 1 or 9. Therefore, the content of the above blank is 7.

In the case E, the program detects "just two blanks that have just two possible digits". Then the program makes the other blanks on the same row/column/box not hold that two digits.

(Algorithm 6)

```
for each row 1...9
  if there are just two blanks that can hold n1, n2 only
    then the other blanks on the row cannot hold n1, n2
for each column 1...9
  if there are just two blanks that can hold n1, n2 only
    then the other blanks on the col cannot hold n1, n2
for each box 1...9
  if there are just two blanks that can hold n1, n2 only
    then the other blanks on the box cannot hold n1, n2
```

4. Performance Comparison

The methods described in the previous section have been tested with various Sudoku problems. First, we divide Sudoku problems into three classes: Class 1 indicates those problems which have only one solution; Class 2 those with no solution; Class 3 those having two or more solutions. Then, we have tested five instances of the Sudoku problems using (1) simple backtracking, (2) backtracking with MRV heuristic, and (3) backtracking with MRV heuristic and advanced solving methods developed in this project, and measured the times taken to solve the problems.

(Table 1) Time taken to solve Sudoku problems (msec)

	Simple BT	MRV	MRV+Adv
Class 1	72.8	1.9	1.7
Class 2	43.1	0.8	0.8
Class 3	143.5	4.1	2.3

Table 1 shows the average times taken to solve the problem instances (measured in milli-seconds). Although the number of problem instances generated and tested in this experiment were small, the result shows that the method using the MRV heuristic performs far better than the simple backtracking method without any heuristics. However, the performance gain over the MRV heuristic by the advanced algorithms has turned out to be marginal.

5. Conclusion

Like a shortest path problem and map coloring problem, Sudoku is a kind of problem in which intelligent solving algorithm is important. Normally, Sudoku puzzles are done with 9 × 9 square. Nonetheless, 16 × 16, 25 × 25, or even 100 × 100 Sudoku problems are possible to be made, if we want to make them. If the square used in a Sudoku puzzle becomes larger, it becomes more important to use more intelligent methods to solve the puzzle.

The MRV heuristic is one of heuristics used to solve CSP problems. It improved the speed of the 'Sudoku solver' program by dozens of times. The advanced solving mechanisms reduced the number of backtracking times by filling blanks by other method or deriving additional constraints. Sometimes the advanced methods were not useful, but in most cases, they speeded up the 'Sudoku solver'.

This paper showed us the utility of heuristics and other intelligent methods for finding solutions in complex problems.

References

[1] S. Russell and P. Norvig, Artificial Intelligence: A Modern Approach (second edition), Prentice Hall, Pearson Education, New Jersey, 2003

[2] Sudoku – one of the Puzzles by Pappocom (http://www.sudoku.com)

[3] Kwak Sung-eun's Sudoku (http://my8972.com.ne.kr)

[4] Chosun Ilbo, Readers' Quiz – Sudoku