Sudoku with Hill Climbing

Kanstantsin Downar

CTU–FIT

downakan@fit.cvut.cz

May 12, 2024

1 Topic

This documentation will describe the Sudoku solution using the "Hill Climbing" algorithm. The main idea is to gradually improve the current solution, moving towards the optimal solution, by changing the current state. In our context, each attempt to solve a puzzle can be considered as a state, and the state of the optimal solution can be considered as a target state. The goal is to change the current solution in such a way as to get closer to the target state (the perfect solution to the puzzle).

2 Input data

Our program should be able to work with a generalized version of Sudoku (not only with the classic 9x9 version). Therefore, 16x16 Sudoku variants were also prepared. The entrance to our program is a partially solved Sudoku of a certain size. The number of blank fields varies from 10 to 35 in our case.

3 Methods/procedures/algorithms

Step-by-step description of the solution algorithm:

1. **Determination of the evaluation function**: Each state (solution) is evaluated using a function that counts the number of collisions in rows, columns, and subgrids. The lower the value of the function, the better the solution. Zero value is a perfectly solved Sudoku.

2. **Generating the first solution**: Empty fields are filled in with random valid numbers.

3. **An iterative improvement process**: Until the optimal state (target solution) is reached or until the number of iterations of the algorithm has exceeded the limit, new solutions will be generated by small (local) changes to the current solution. If the new state is better (closer to the target state), accept it and continue to search for improvements. Otherwise, this iteration will be considered unsuccessful. If the algorithm exceeds the limit of consecutive unsuccessful steps, the solutions will be considered as unpromising and a new starting state (the first solution) will be generated.

4. **Stop**: The program will stop if the iteration limit is reached or if an ideal solution is found (the value of the evaluation function is 0).

4 Generating a new solution

During the work on this project, relatively many ways were tried to generate a new solution. This report will describe those who showed the best result.

1. **A random approach**: The number on a random position is changed to a random valid number.

2. **Changes in the number that violates the rules of the game**: A number is selected in the grid that exactly violates the rules of the game, specifically the neighboring number is the same (not taking into account the diagonal neighbors). The selected number is changed to a random valid number.

3. **Changing the number that violates the most rules of the game**: The algorithm calculates the value of the evaluation function for this solution (score). For each number in the grid, the value of the evaluation function (ignoreScore) is calculated, which ignores all collisions that occur due to the number in this position. The algorithm then selects the position for which the value of the expression

$$score - ignoreScore$$

is the maximum. The number in this position is replaced by a random valid number.

4. **Changing a random number to the one that brings the solution closest to the ideal one**: The algorithm selects a random number in the grid and replaces it with the one that will most improve the value of the evaluation function (bring it closer to zero).

It is worth emphasizing that these methods work only with those numbers that are located in cells that were originally empty.

5 Results

The results of each described method are clearly shown in the table below. The first two columns describe the task to be solved (Size - the size of the field, Emptiness - the number of empty cells). The time values are indicated for how long, on average, this method solves this problem.

Size	Emptiness	RandomSolver	NeighborSolver	ChangeWorstSolver	MinimizeSolver
9x9	10	245ms	31ms	49ms	11ms
16x16	10	43ms	405ms	124ms	39ms
9x9	15	255ms	1s 993ms	5s 381ms	11ms
16x16	15	766ms	307ms	17s 831ms	76ms
9x9	25	40s 530ms	9s 79ms	>10m	2s 167ms
16x16	25	234ms	5s 220ms	14s 177ms	151ms
9x9	35	4m 42s 803ms	2m 17s 155ms	>10m	1m 59s 564ms
16x16	35	5s 848ms	18s 590ms	>10m	16s 913ms

Based on the table, we can conclude that the best algorithm in our case is the one that uses the last mentioned method of generating a new solution by replacing a random number with one that brings the solution as close as possible to the ideal one.