Computationele Intelligentie
Practicum 2: Local Search

Wouter Bink, 4105524

Iterated Local Search

Implementation

A sudoku puzzle is represented by an $n*n$ two-dimensional array of integers, where n is the size of the sudoku. Each sudoku is divided in $\sqrt{n}$ **blocks**, where a **block** is an $\sqrt{n}*\sqrt{n}$ square itself. These **blocks** are not implemented as their own data structure, but instead as a layer of abstraction on the larger sudoku. A pair of Getters and Setters allows the rest of the code to access each **block** separately. Additionally, during the computation of a Swap, the **block** currently being worked on is flattened to a one-dimensional array.

The current value of the Evaluation Function for each cell is kept in another two-dimensional array of integers.

There are two parameters relevant to the program. *S*, as given in the exercises, defines the amount of times the state is perturbed upon reaching a maximum. *N* defines the maximum amount of sideways moves allowed before a perturbation occurs. This was needed to prevent the program from getting stuck on plateaus for a possibly infinite amount of time. Additionally, the Time Limit was set to 10 minutes, which allows the results to be directly comparable to those of the last exercise.

Initial testing revealed that the optimal values for both *S* and *N* varied greatly with different sudokus, and different seeds for the PRNG. Because of this, "4" was chosen by fair dice roll to be the seed in every run.

This still left *S* and *N*. A pattern did emerge where *S* would usually stay between 1 and 3, while *N* could grow up to 8. But one static optimal value did not appear. A possible relationship with the State Evaluation Function was briefly looked into, but discarded due to lack of time.

The solution that was eventually settled upon was to run multiple searches in parallel, with each having different values for *S* and *N*. This had the added advantage of

allowing the program to scale with multiple cores, something that would otherwise be hard to do when solving sudokus (It was tried in Prac 1. This was not a success.)

The combination of parameters that most often saw success was *S*=1 and *N*=5.

```
ILS(Sudoku)
counter = 0

while NOT TimeOut AND EvaluateState > 0
        if counter > N then:
                counter = 0
                repeat S:
                        SearchOp(Sudoku, Random.NextBlock(), true)
        else:
                result = LocalSearch(sudoku)
                if result = 0 then counter++
return Sudoku
```

```
LocalSearch(Sudoku)
blacklist = {}

do:
        Block = Random.NextBlock(blacklist)
        result = SearchOp(Sudoku, block, false)
        if result < 0 then blacklist.Add(block)
        if result > 0 then blacklist.Clear()
while result IS NOT 0 AND NOT blacklist.Full

return result
```

```
SearchOp(Sudoku, Block, Perturb)
flat = Block.Flatten

if Perturb then:
        Swap(flat[Random.Next()], flat[Random.Next()])
        return
else:
        possibleswaps = SELECT (x, y), val = EvaluateSwap(x, y)
                        FROM Pairs*
                        ORDER BY val DESC;
        bestswap = possibleswaps.First
        if bestswap.val ≥ 0 then:
                Swap(bestswap.x, bestswap.y)
        return bestswap.val
```

*a precomputed set of possible pairs.

The following methods/functions are important, but were not included in the pseudocode:

- EvaluateSwap(Sudoku, cell1, cell2): Returns the change to the State Evaluation as a result of this Swap, without actually performing it. Only checks the rows and columns relevant to the swap.
- EvaluateState(Sudoku): Returns the total value of the current state. Checks the entire sudoku.
- ReEvaluateCells(Sudoku, cell1, cell2): Updates the values in global evaluation table for the rows and columns relevant to the cells that changed. Called as part of the Swap.

Table 1: ILS compared to Backtracking and Backtracking with Forward Checking

Puzzel		**Multithreaded ILS**	**Forward Checking**	**Fastest BT**
Standaard puzzel	Runtime: Recursion:	27 ms S=1, N=7	519 ms 39,223	1,220 ms 518,275
Project Euler 01	Runtime: Recursion:	Time limit	8 ms 50	5 ms 201
Project Euler 02	Runtime: Recursion:	Time limit	10 ms 56	8 ms 295
Project Euler 29	Runtime: Recursion:	Time limit	14 ms 376	24 ms 5,246
Project Euler 36	Runtime: Recursion:	Time limit	8 ms 50	4 ms 84
Project Euler 50	Runtime: Recursion:	Time limit	10 ms 156	16 ms 4,585
Sfsudoku Line 01	Runtime: Recursion:	22 S=1, N=5	30 ms 1,589	494 ms 204,901
Sfsudoku Line 02	Runtime: Recursion:	195 S=3, N=7	99 ms 5,203	374 ms 130,495
Sfsudoku Line 03	Runtime: Recursion:	37ms S=1, N=5	491 ms 38,927	6,227 ms 2,768,022
Sfsudoku Line 499	Runtime: Recursion:	47ms S=1, N=3	70 ms 5,109	146 ms 46,182
Sfsudoku Line 500	Runtime: Recursion:	33 ms S=1, N=5	88 ms 6,393	643 ms 235,915
16x16_1	Runtime: Recursion:	Time limit	30 ms 346	146 ms 42,435
16x16_2	Runtime: Recursion:	Time limit	Time limit	Time limit
16x16_3	Runtime: Recursion:	Time limit	Time limit	Time limit

Conclusion

This implementation of ILS appears at first glance to be extremely unreliable. Half the time, it keeps up with the other algorithms and is often faster. The rest of the time, it times out. Something very strange is happening here. Every sudoku from projecteuler.net ran out of time, while the ones from sfsudoku.com ran just fine. Additional sudokus were tested and although they were not included in the table, they show the exact same pattern.

The exercise description for Prac1 indicates that the other sudokus have the fewest possible fixed numbers; while this normally makes a sudoku more difficult, the opposite seems to be the case here. Perhaps the additional free numbers allow the ILS to have more possibilities to swap, and therefore make better decisions. Regardless, if one were to forego testing these sudokus and only test the algorithm with the sudokus from sfsudoku.com, a totally different image of this program would arise!

Not only do the different parameters have a large impact, but the chosen seed for the PRNG also ends up mattering a lot. An alternative to this stochastic process for choosing which block to evaluate would probably largely fix this. A possibility here would be to compute the Evaluation Function for each block, and choose the one with the highest value (and therefore, most missing values).

On the other hand, as it is a Local Search Algorithm, its memory footprint is significantly lower. This was very visible during testing: despite some extremely inefficient memory usage in the programming, it used less memory than the earlier Backtracking algorithm.

Reflection

While the programming itself was not particularly difficult, development was still very troublesome. Communication with my practicum-partner slowed down after the first assignment and he eventually stopped responding altogether. I was personally hesitant to take action and instead set up a rough framework for the program, after which I focussed on other tasks until last week. By then it became clear that this would be a solo project, so I finished the project by myself. To make things clear: my practicum-partner did not contribute anything to this project. Naturally, I should have acted sooner, and the fact that I had to work with a limited time frame was still fully my fault.

Despite this, the current implementation offers something interesting in that it is a multithreaded sudoku solver (although not a very good one). Sudokus are tricky puzzles to solve in parallel and the large amount of variation between parameter values is easily capitalised upon to create a parallel algorithm. Because of this, and the better-than-expected solving speed for most sudokus, I'm relatively happy about the end result.

However, most importantly, I really want to know why the algorithm performs so badly (or more accurately, not at all) with the sudokus from projecteuler.net. If the person reading this has any idea, I'd love to know!