CS182 Assignment 3

Charles Liu
cliu02@g.harvard.edu
10/16/15

5: Analyzing performance of solvers

Solver	Time (s)	Explored
CSP	15.96	56734
CSP + Forward	5.99	7095
CSP + Constrained	0.39	787
CSP + Constrained + Forward	0.65	734

The ordering of the search states according to the minimum constrained heuristic brought about the largest improvement in terms of speed. Adding the forward checking component decreased the number of expanded states, but the additional computational overhead of calculating the factor constraints caused the algorithm to be slower. A way to circumvent this issue would be to have the ability to pass the factor domains hash in the constructor for a new Sudoku object. Similarly, because the structure of the code was in such a way that a lot of it was shared among these methods, the basic CSP algorithm had the same updateAllFactors() call that the subsequent solvers had. This isn't necessary as only the row/column/box for the variable in question needs to be updated at any iteration, and when that was removed the speed improved to 7.65 seconds (the number of search states remains the same as this is only reducing computational overhead).

8: Stochastic Descent

My initial local solver was not returning any solutions. The initial number of conflicts ranged from 45-55, with the solver reducing the conflicts to 10-15 (on rare occasions dipping under 10). After implementing the moving to a higher conflict state with probability 0.001, the solver started to return solutions but more regularly it would hit 2 conflicts (the lowest possible) and then either stagnate or jump because of the 0.001 probability of moving to a less optimal state. Increasing the number of iterations to 500000 almost always returned a solution whereas the initial solver would still remain stagnant. The reason for this is perhaps in the first case the board is stalling in a local minima and unable to move out unless you first accept a worse state first. Take for example the following board:

1	3	2
1	2	3
2	1	3

In this example let's just concern ourselves with the row/column constraints. Currently there are 2 conflicts, with repeated 1's and 3's in the first and third columns respectively. On the one hand the simplest solution would be to rotate (0,0) and (0,1) then (2,1) and (2,2). This wouldn't increase the conflicts at any point. However, another solution would be to rotate (1,0) and (1,1), (2,1) and (2,2), and (2,0) and (2,1). This actually increases the number of conflicts to 3 in the first step, but ultimately ends in a consistent state. Adding this variablility opens up the possibility to these other paths to solutions.

9: Genetic Algorithms

The chromosomes would represent a board similar to our gradient descent - where each row is filled in completely and is consistent. We create N of these boards and rank them by the number of violations (fitness function) - $F_1, F_2, ..., F_N$. We then randomly choose with replacement N grids based on the inverse probabilities, so for chromosome x it's probability of being chosen is $\frac{\frac{1}{F_x}}{\sum_{i=1}^{N} \frac{1}{F_i}}$. The crossover stage is choosing a random number between 0 and 7. That number, let's say x, would mean the first chromosome would keep it's rows from 0 to x, and have rows x+1 to 8 from it's paired chromosome. The paired chromosome would have it's own rows for indices x+1 to 8 and the first chromosome's 0 to x rows. Mutation would be if a random.random() returns $\leq .001$, then we perform the random swap that the gradient descent.