Sudoku Solver using a Generic Algorithm

Fitness Function:

Before describing which fitness function was picked, it is worth going over what exactly the fitness function is trying to measure. Basically, the point of Sudoku is to finish laying out the symbols that someone already started for you. An empty Sudoku grid has a number of solutions, but it does not qualify as a puzzle. Sudoku puzzles start with some symbols already in use, which serve to limit the alphabet that can be used and restrict placement so that when every single symbol is put into the puzzle, they form the intended configuration. In a valid puzzle, there is only one way to fill every single gap, and it just so happens that is the correct solution. The first thing this solver does is go through every gap and checks how many values that gap can be filled with, based on the row, column and box that the gap is in. If it turns out there is only one, the solution is trivial and automatically applied as part of the pre-processing. The challenge lies with gaps that can be filled with several values. Only one of them is the correct choice and picking the wrong one will lead to a dead end. The only way to know for sure which choice is the correct one is to pick one and try to solve the rest of the puzzle. This of course leads to a ridiculous amount of computation, which of course is what Sudoku's NP-completeness is all about. The point is, each gap can be filled with several values and these values are distinct for most gaps. This means that the first impulse to use the gaps as a population is probably a bad idea, given that each one essentially has its own alphabet. After some thought, it occurred that since there is exactly one way to fill every gap, there must be some specific order that the gaps can be filled in that will result in the solution. This became what the population represented. Each unit of each population represents the order that the gap at that position gets filled in. For example, if the second unit of the first population is 14, it means that gap #2 will be filled after 13 other gaps are filled. Because a population does not directly represent anything that a meaningful score can be computed from, the gaps have to be filled in that population's order to get an actual score. The gap filling function uses a map to see which symbols have been used in which rows, columns and boxes, so when a hole needs to be filled but no more available symbols remain for it, it gets filled with a 0.

The fitness function measures the amount of non-zero values that resulted from applying a *population* to the gaps in the given puzzle, divided by the population length (which is the number of gaps in the current puzzle). The higher the value, the closer the puzzle is to being complete. Values used for selection are normalized using 2^n and values reported as average and max are raw.

Recombination operator:

Because each population string represents a set of positions, it cannot be chopped up into pieces during the mating process. In fact, the only operator that makes sense is the swap() function, because what matters in a population is not the value but the position of each element. During recombination, both individuals are split into MATING_CROSS_POINTS pieces. In a zig-zag pattern, the odd pieces remain in place and even pieces are changed to look like the other individual. This is accomplished by doing basically a selection sort on just that section of the individual. When an item at position n of the section being re-arranged is found to not match the item in the parent, the child is

scanned from beginning to end, looking for that item. When it is found, it is swapped with whatever the current item is. After this is done for the whole segment, the child's segment looks like the parent's segment and the values that were swapped are now elsewhere. This didn't seem like the perfect way of doing things, since the original recombination only copies sections into the child and doesn't change anything outside of those, but that was the only way I could think of doing it.

Mutation approach:

At first, mutation was implemented in a way very similar to recombination. For each element of each of the two individuals, if a random value was above the treshold, that element was made to be the same element as in the position of the other individual. It seemed to work good enough, but would be very weak after a lot of iterations, providing barely any changes. After some thought, it was changed to simply swap a random value with another random value within the same individual. This way seemed to introduce much more consistency to variation, it is now present in the same amount regardless of the number of iterations.

Values of parameters:

Population size: 500. This seemed like a good middle ground, at least with 9x9 grids. Fewer than 400 resulted with just one or two individuals per population being the best. More than 500 resulted in slower performance and no noticeable gains.

Mating bias: 75%. Too far below, the population converged much more slowly. Too far above, the population oscillated between bad and worse. At 75%, it converged to the upped 90s fairly quick.

Crossover points: 10. An even number of crossover points seemed to result in faster score improvements, but only barely. When the value was set to 4, there were three instances when the hardest problems were solved within 300 iterations, but that is a very small amount out of the hundreds of runs that were done. A high number of crossover points may slow down initial convergence rate, but not by a lot. While crossover points did not considerably change the performance of valid sudokus, the blank one converged much faster with 10 as a crossover point value.

Mutation rate: 20%. Because a lot of the gaps are independent of each other, the mutation rate had to be set pretty high for it to matter. Even very high mutation rates did not seem to negatively affect the best score per population, but they did make the average score oscillate much more.

Final thoughts:

The problem with this implementation is that for very hard problems, it gets to a very high maximum value and just sits there. Easy problems gets solved very fast and hard problems are done within a few hundred iterations. However, if it gets past about 500 iterations, it will probably sit at the very high max value forever. I ran a very hard hexadoku overnight and it actually came down to the lower 70s by the morning. I need to figure out why it never gets done if it keeps going too far.