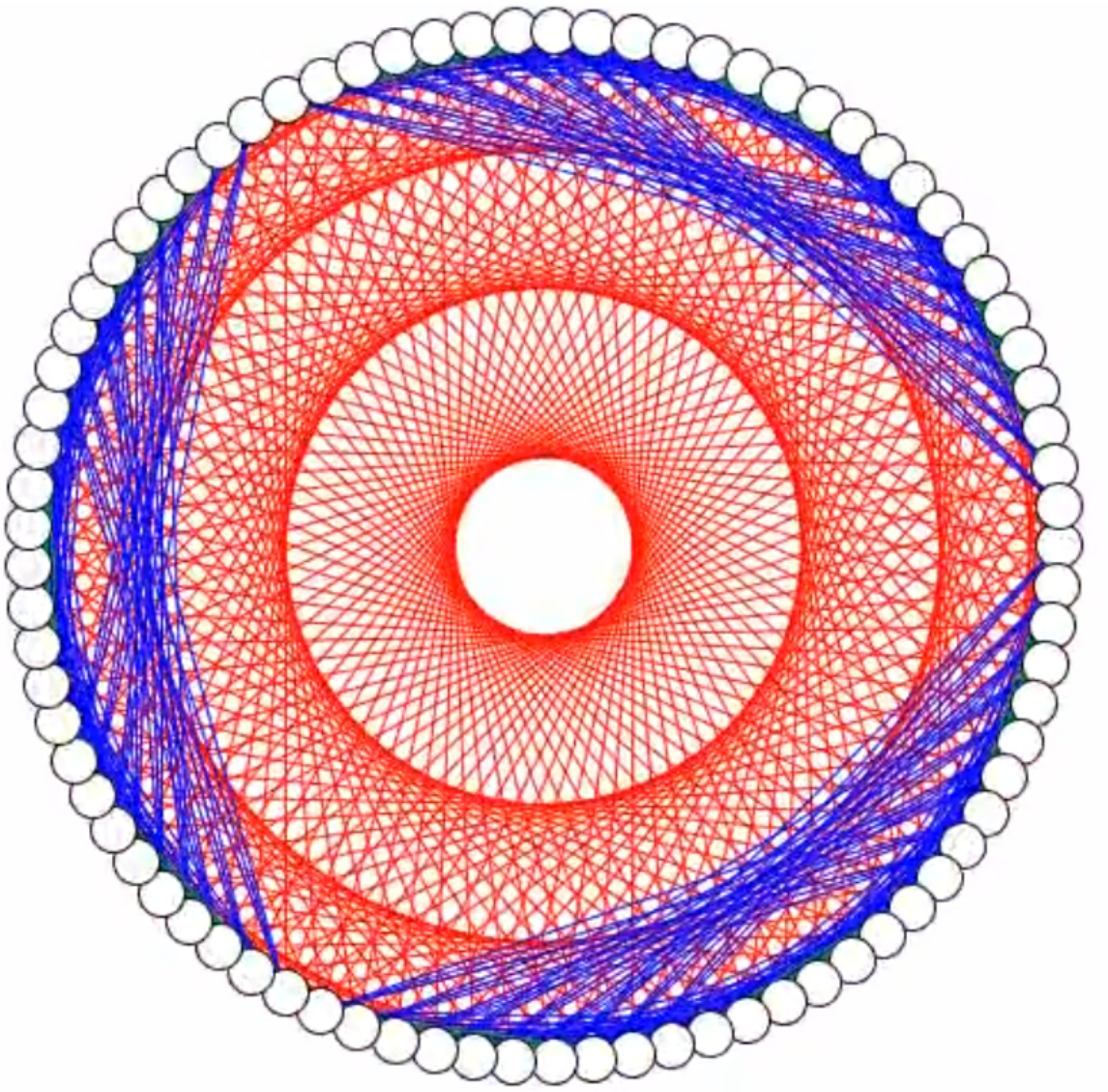
Sudoku



Authors:

Abraham Btesh (777539958)

Muhammad Abu-Ali (208887687)

Issa Abu-Kalbien (207712472)

Introduction

This project attempted to solve the sudoku puzzle, on boards of various sizes, using two different approaches; backtracking and local-search. First, we solve 9 by 9 boards using naive backtracking, backtracking without heuristics. This works fairly well and we continue to boards of larger sizes 16 by 16 and 25 by 25. Backtracking works although, on 16 by 16, it takes several minutes. So we introduced heuristics in order to cut down the runtimes. Theses heuristics are also very useful in solving 25 by 25 boards, which are otherwise not solvable using naive backtracking in any reasonable amount of time.

We proceed to try to solve Sudoku puzzles using two different local search methods; simulated annealing and stochastic beam search. These are both highly unsuccessful approaches for reasons we attempt to analyze later on. Due to their lack of success on small boards, we did not scale them to larger boards. We proceed through the various algorithms in the order we developed them in the hope that this will lend some coherence and story to the report.

Problem Description

Sudoku is a popular game, it is usually played on a 9 by 9 board so that each row and column contains nine squares. Some of the cells are already assigned with a number that can not be changed. The player adds numbers from 1 to 9 to the empty cells and the puzzle is considered solved when each number between 1 and 9 appears once in each column, once in each row, and once in each 3 by 3 sub-grid. This is the goal state. The same can principle can be extended, as we have in this project, to boards of larger sizes like 16 by 16 and 25 by 25.

We attempted to solve this problem by using the following methods:

- Backtracking
- Backtracking with heuristics (Minimum Remaining Value and Least Constrained Value). We refer to this agent throughout the report by the name Csp agent
- Simulated Annealing
- K - Beam Search

Each of the different methods will be described progressively as we reach them.

Constraint Satisfaction Problems

Constraint Satisfaction Problems are defined as a set of variables $\{x_1, ..., x_n\}$ each of which has an associated domain, $\{D_1, \ldots, D_n\}$. However, the CSP's defining characteristic is the set of constraints, C, which describes relationships between particular variables and particular values, for example, $x_1 \neq x_2$.

We can think of a Sudoku problem as taking this general form. Each square in the grid, which, when we start the problem, is unassigned, is a variable which can take the values $1, \ldots, 9$. In other words each unassigned variable has the domain $\{1, \ldots, 9\}$. The set of constraints C is defined in such a way that the same number can not appear in the same row, column or sub-grid. Csp's have a variety of approaches but the basic tool is backtracking.

BackTracking

Description and Implementation

The backtracking algorithm attempts to fill a cell with a value that doesn't violate any constraints. Then backtracking goes to the next cell, the cell in the next column or the next row if it has reached the end of the row, and attempts to fill it with a valid value, if there is no valid value we backtrack, meaning the algorithm goes back to a previous cell, and tries to choose another valid value and then returns to the cell which did not have a valid value and attempts to fill it with a valid value. This is a technique that works fairly well on Sudoku puzzles which are 9 by 9. It is also extremely fast. However, when we tried to solve 16 by 16 boards the runtimes go from several seconds to several minutes. So we introduced some heuristics which cut down on the runtimes significantly.

Heuristics:

- Minimum Remaining Values: we used MRV as a heuristic for choosing variables. According to MRV our backtracking agent picks a variable (a cell on the board) which has minimum number of legal values for available to it considering the current state of the board. (all other places have bigger/same number of legal values).
- Least Constrained Value: we used LCV as a heuristic for choosing values. According to LCV our backtracking agent assigns a legal value - to an already selected variable - which is least constrained by the legal assignments of the other variables (all other values constrains bigger/same number of legal assignments of the other variables).

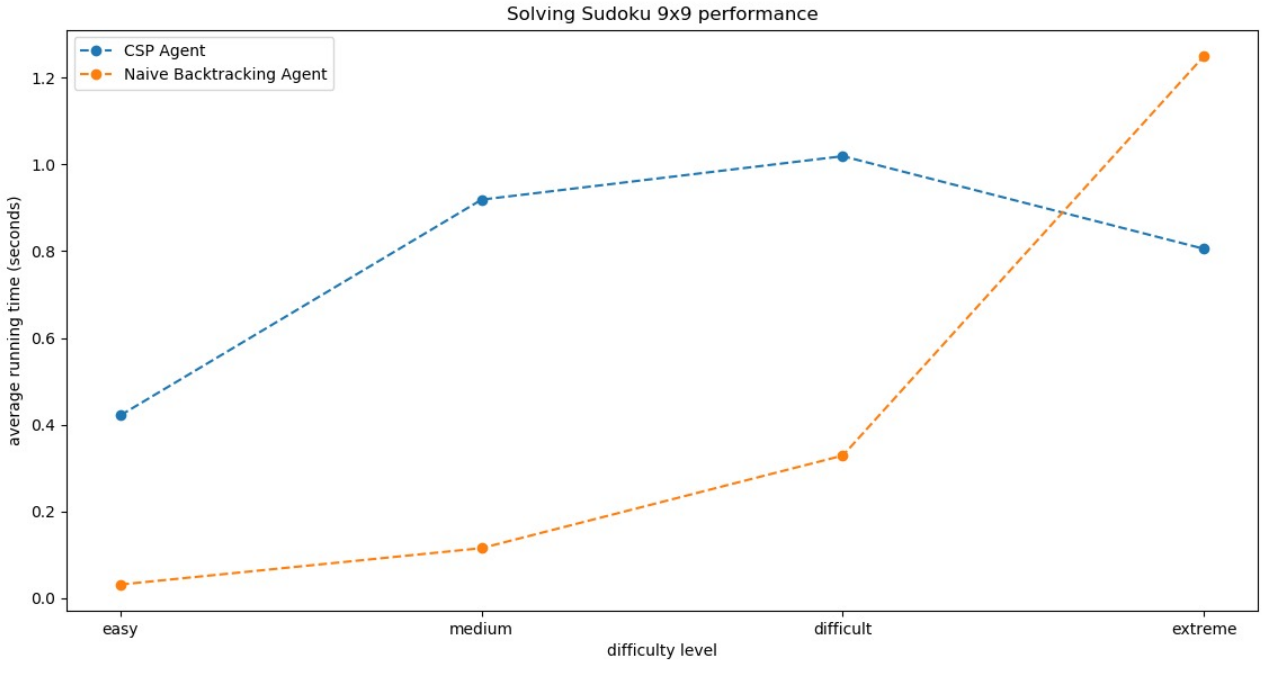
Sudoku: 9 by 9

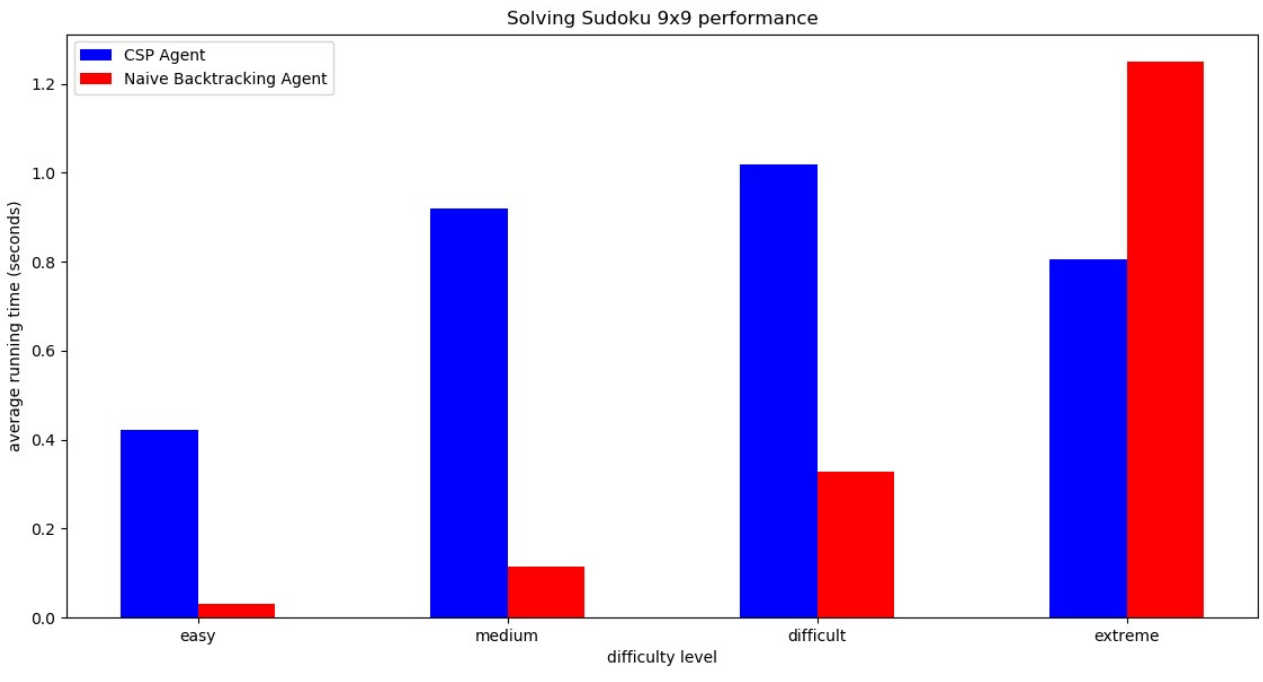
We tested our code for 9 by 9 Sudoku boards using collections of boards of different levels of difficulty. Easy, medium, difficult and extreme (extreme is a Sudoku board which is very difficult and may have more than one possible solution). As a rule most Sudoku boards, those you would find in a newspaper or puzzle book, are well defined, meaning

We tested our code on boards from http://www.sudoku-download.net/index.php. Which is where we acquired the ranking of the collections of boards. Each of the 9 by 9 collections holds sixty boards. Which allowed us to run fairly large tests to establish the behavior of these algorithms on the 9 by 9 Sudoku board.

In order to understand the performance of the algorithm we tracked runtimes and the number of backtracking calls that were made as the algorithm progressed. Each of these factors is presented in the graphs below.

Average Runtime



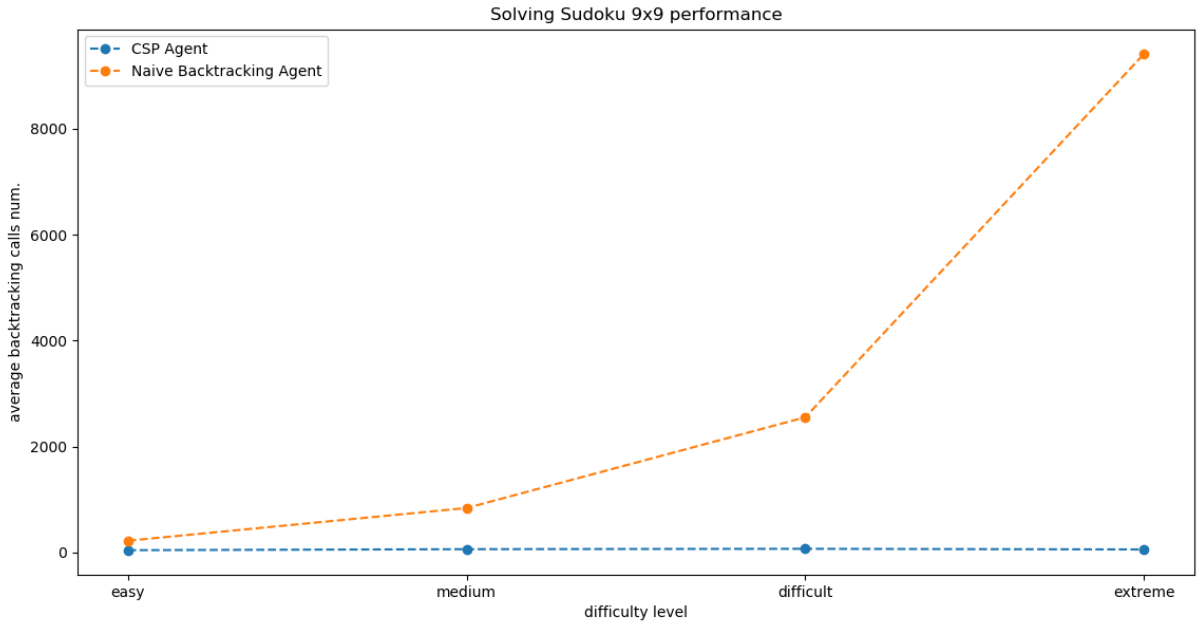


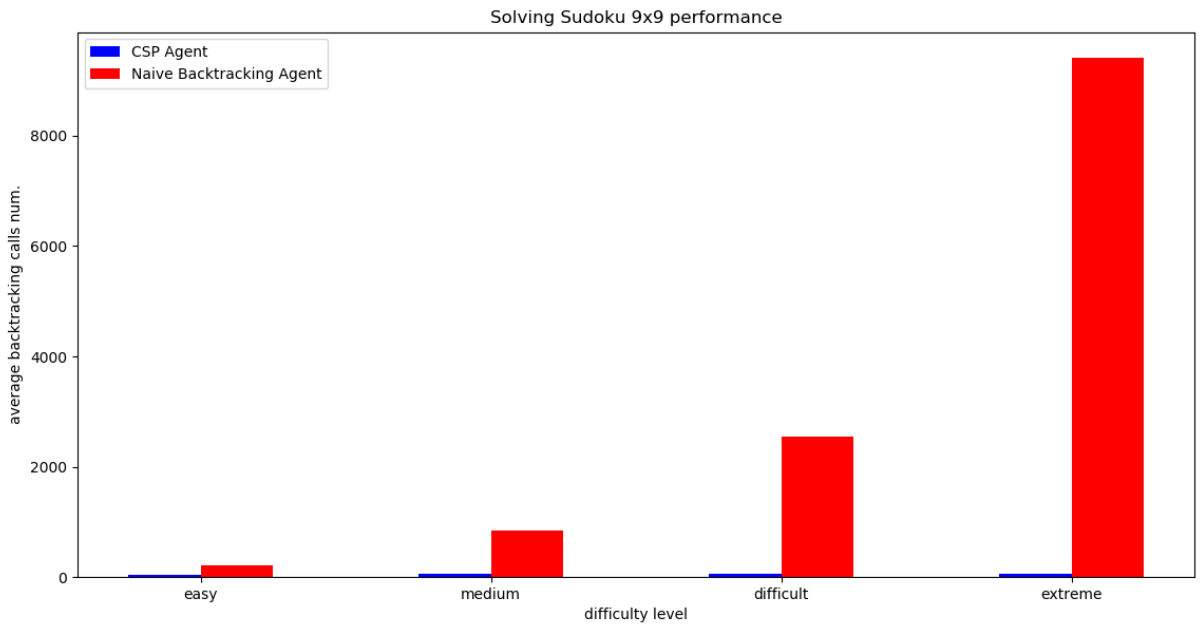
Observations:

There is a half-second added when attempting to solve the easy Sudoku boards using heuristics. We attribute this to the processing time required for the heuristics. There is another half-second (or so) change as we solve medium-level Sudoku boards which seems equivalent to the change in difficulty from one level to the other. However, there is no vast difference between the medium level puzzles and the difficult ones. This is rather odd, at first glance because one would assume, the change in difficulty would correspond to a change in the runtime. However, the runtimes remain rather steady and we do not see such a large change in the runtimes. This seems to be because, once we are using heuristics, solving difficult and medium level boards is not all that different to the algorithm

This is followed by the even more interesting observation that extreme level boards take less time than both medium level boards and difficult ones. So we are forced to ask why this ask happens? This seems to be a consequence of the ease of having several solutions. Using the MRV and LCV heuristics gives a significant advantage. Simply, it means that any of a number of choices are acceptable so there will be less need to go backward. This is a result we will see very clearly in the graph showing number of backtracking calls: below.

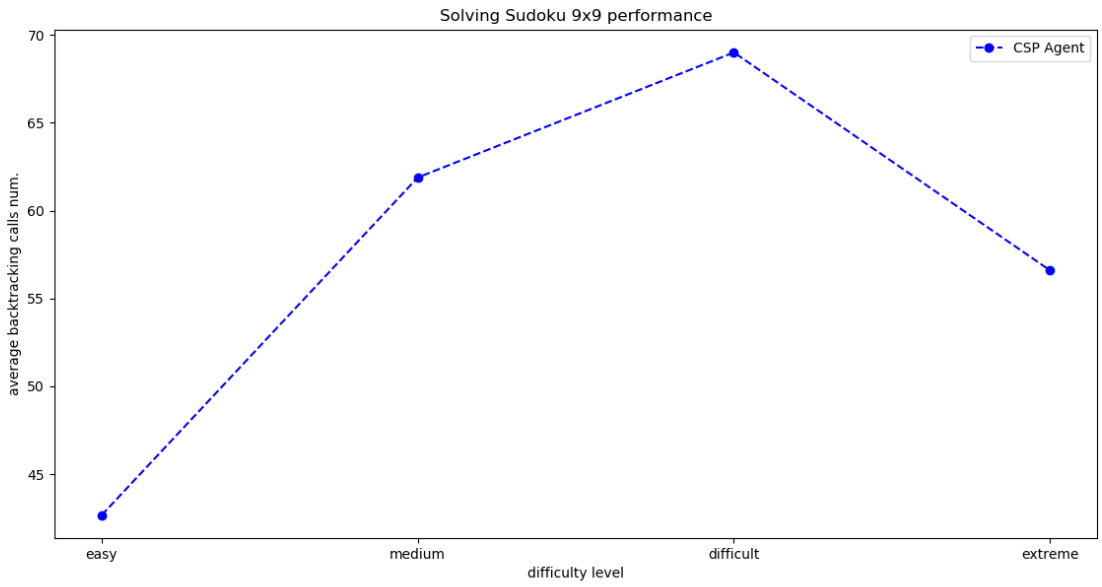
Average Number of backtracking calls





Observations

1. Using the heuristics the number of backtracking calls is extremely small and stays flat even as the Sudoku boards get harder and harder.
2. The number of backtracking calls required for the extreme level boards is smaller than the number of backtracking calls for the medium level and difficult boards. Here we see the advantage provided by the existence of many solutions when using heuristics which we noticed in the previous graph. We can see the difference rather dramatically represented in the graph below.

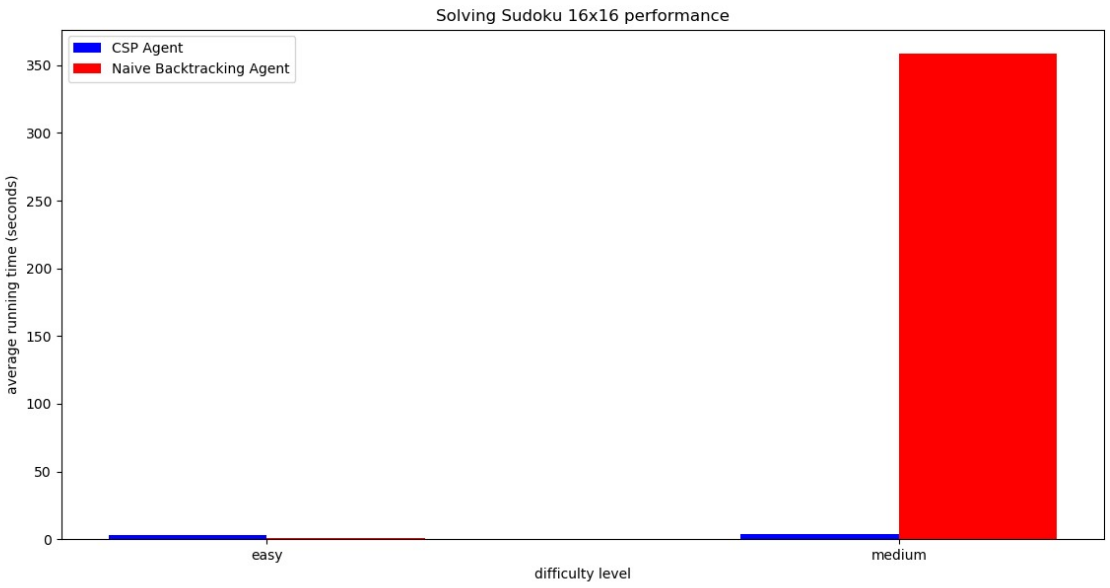


Sudoku boards: 16 by 16

As seen above, backtracking works very well and very quickly with 9 by 9 boards and we wanted to see if we could extend our methods and algorithms to larger sizes and observe the results. We tested our methods on boards of two different levels of difficulty, easy and medium, to see how our algorithms performed when faced with a much larger challenge. The results are presented below, as above, with runtime and number of backtracking calls.

We had a sample of 24 easy-level boards and 23 medium-level boards on which to test our code.

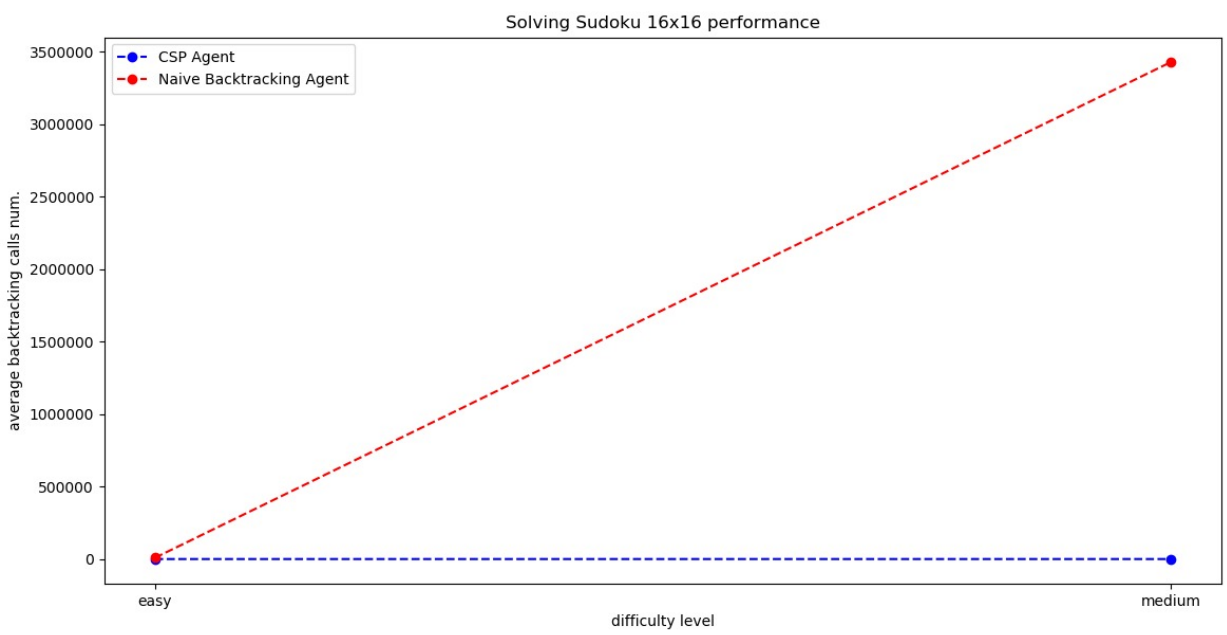
Average Runtime



Observations:

1. There is a difference of nearly six minutes between backtracking without heuristics than with heuristics.
2. With heuristics, the runtimes for the easy boards goes up, however, we attribute this to the extra time required to run the heuristics.
3. There is one second difference between easy level boards and medium level boards with backtracking and using heuristics. This is an enormous difference from the almost six minute difference backtracking without heuristics and shows us the massive gain in performance made when backtracking and using these heuristics.

Average Number of Backtracking Calls



Observations:

With and without heuristics backtracking makes almost the same number of calls on the easy-level boards. However, the difference between the number of backtracking calls on the medium level boards is marked. This shows us just how successful these heuristics for this problem, by keeping the number of backtracking calls almost flat.

Sudoku boards: 25 by 25

In an attempt to push our code and methods to their limit we decided to test our code on 25 by 25 boards. We took a set of twelve, 25 by 25 boards which are ranked as easy boards. (Found in the file boards/sudoku_25×25_easy.txt) With heuristics it takes an average of one minute and 51 seconds to complete each of the twelve boards. However, without heuristics the computer we ran this on crashed after running for nearly 24 hours without solving even the first board. For this reason we don't have a measurement of how long naive backtracking takes on 25 by 25 boards. This is not surprising because the space of possible solutions is extremely large. We can get a gross upper bound by looking at the following the number of cells on the board, total, raised to the power of all possible assignment.

$$(number\ \ of\ \ cells)^{(possible\ \ \ assignments)} = 25^{2*25} = 78886090522101180541172856528278622967320643510902300477027893066406 25$$

This is clearly a gross overestimation as it does not take into account the constraints placed on a sudoku board. However, it gives us an idea of what we are dealing with. We can get a more accurate estimation of why it takes so long to solve the easy boards, without heuristics, by calculating the average number of cells which are not initially assigned, in a set of 25 by 25 easy boards, and raising them to the power of their assigned domain.

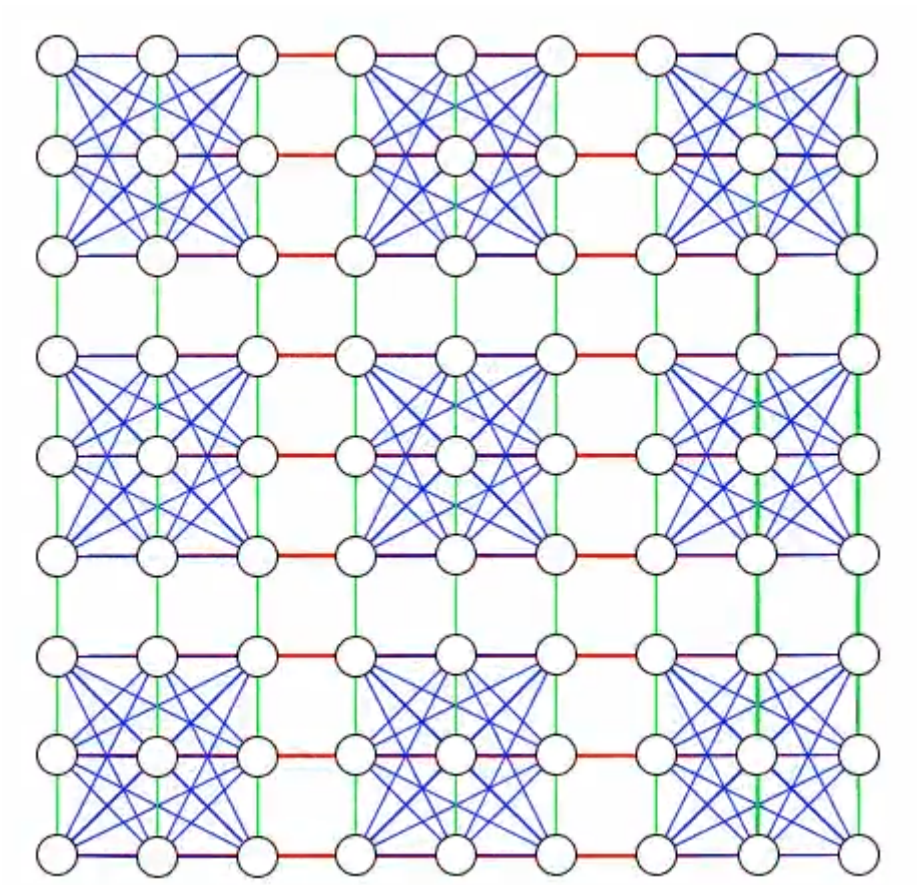
$$(average\ \ number\ \ of\ \ unassigned\ \ cells)^{(possible\ \ \ assignmets)} \approx 276^{25} = 10537244362908691113461693809166669858907622272081351332069376$$

This gives a slightly more accurate estimation of the space of possible solutions backtracking must explore in order to find the correct solution. So we are not to be surprised at its failure to do so. Notably, our heuristics failed to solve the board in the file boards/sudoku_25×25.txt. This is a more difficult sudoku board because it has 372 empty cells initially. Clearly the number of possibilities is monstrous and even using heuristics our computers crashed after running the algorithm for twenty four hours without finding a solution.

Shuffling

Backtracking takes a very systematic approach to choosing which cell to proceed to next. It fills the proceeding cell in the row and at the end the row, descends to the row below. As the project progressed we tried shuffling the cells, i.e. we made the order in which we would proceed to the next cell random and found that the runtimes increased substantially. To the point that solving boards which would otherwise take seconds took minutes or hours.

We can think of Sudoku as a coloring problem on a graph. This means that, in some sense, the effectiveness of the backtracking depends on the structure of the graph. Below we see a graph of the problem which helps us to understand its structure a little better.

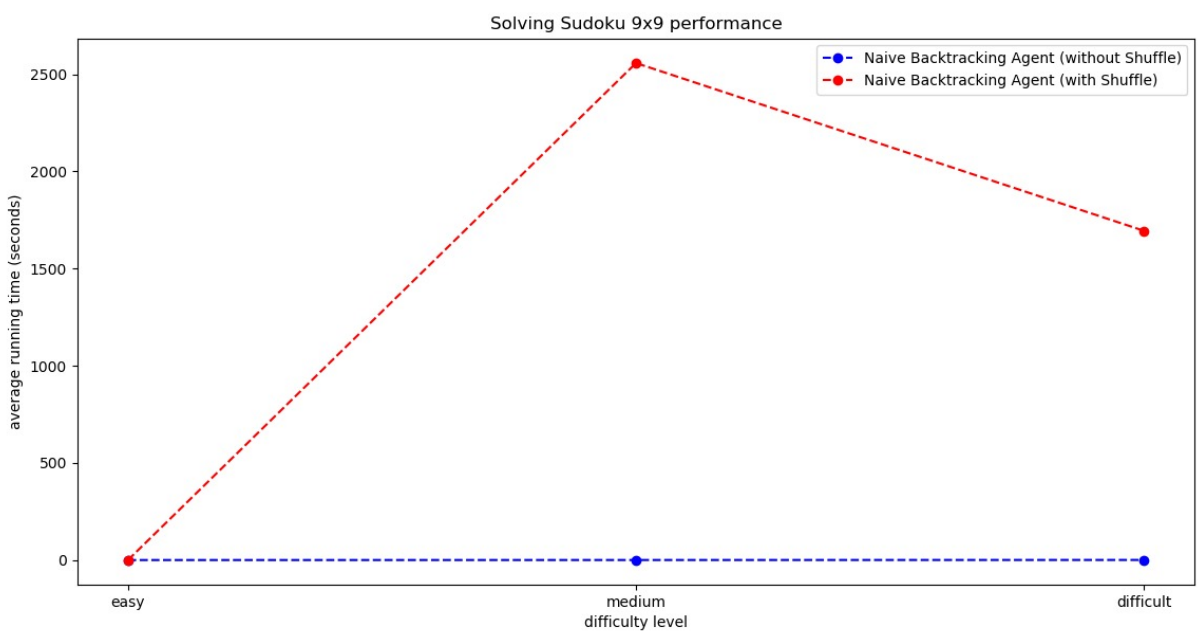


This internal structure tells us why it takes so long for backtracking to solve a Sudoku board by randomly moving through the graph. What the randmoness does is force backtracking to move away from areas on the graph with high connectivity before they are completely filled in and move to another area. This makes it more likely that we will need to backtrack because options that would have been unfeasible if rows, columns or sub-grids, were dealt with first and more systematically are suddenly explored. Backtracking is forced to move back and forth far more than it otherwise would. In short, randomly moving through this graph and trying to solve it is very inefficient because of the graphs internal structure.

Backtracking and Shuffle (No Heuristics)

Below we present the results of backtracking without using heuristics. As we will see shuffling drastically increases the runtimes. The averages are far greater than they were when solving Sudoku boards with the usual backtracking order. For this reason these measurements were taken only on five boards rather than all twenty. We were also unable to test it on the extreme level. The runtimes would become enormous, in the order of five or six hours and each time we tried the computer we were using reliably died.

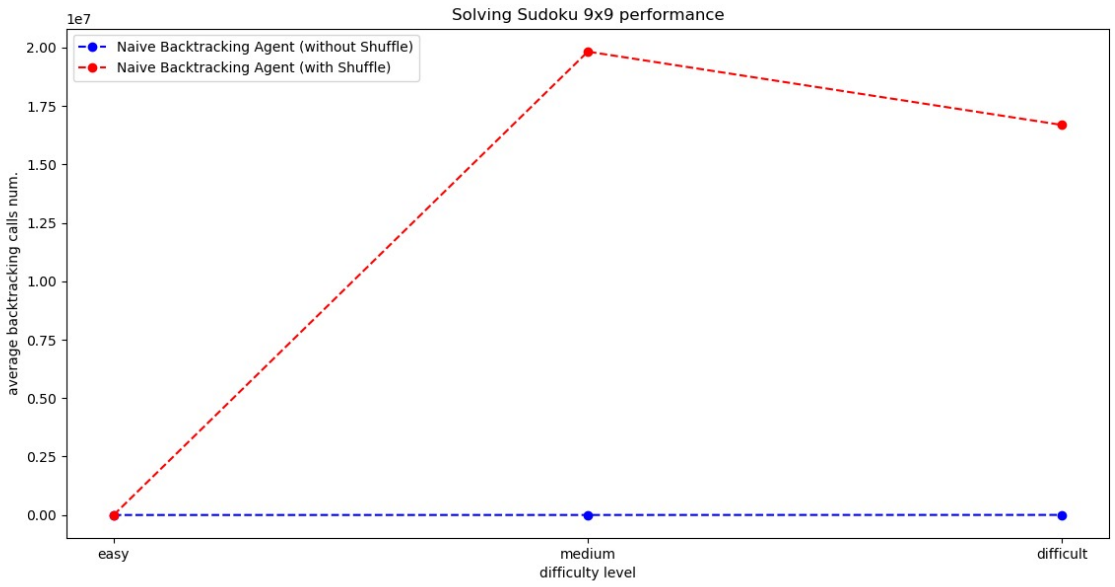
Average runtime



Observations:

1. The run times for easy boards are almost identical. This seems to be because of the level. Despite the fact that an impediment has been placed in the way of the backtracking algorithm because the puzzles are easier it does not make all that much of a difference.
2. The inconsistency in results. Some boards took hours and hours others took only minutes. Results that are difficult to represent in an average. However, this is a result that shows the strain placed on the algorithm by choosing cells randomly.

Average Number of Backtracking Calls



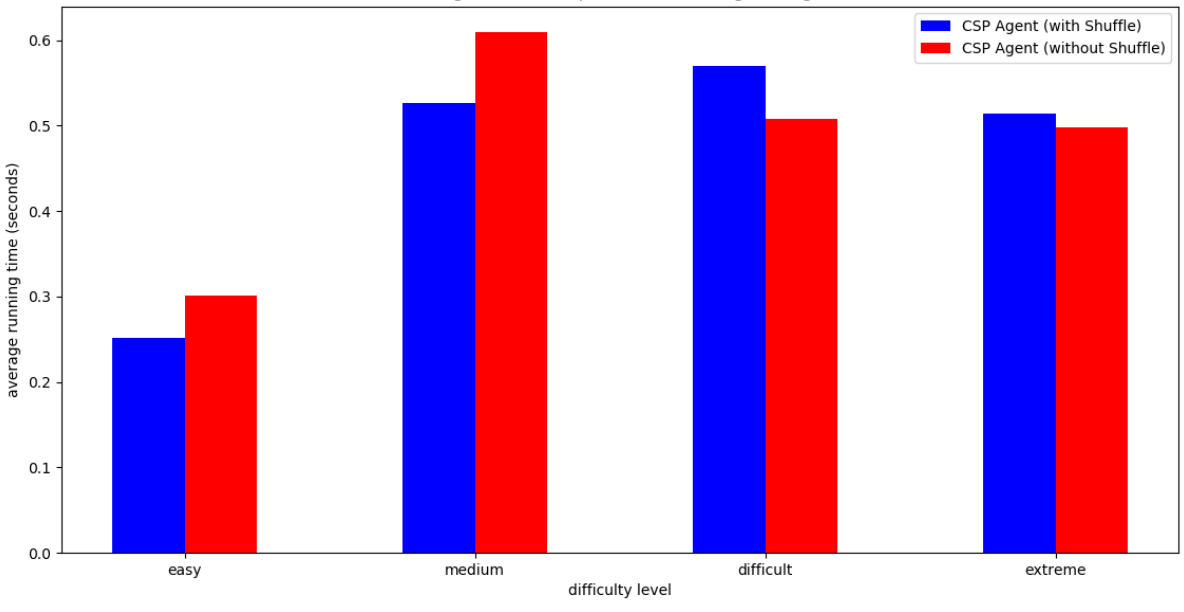
Observations:

Almost exactly the same results can be seen in the data for average number of backtracking calls as we saw in the run time data.

Backtracking with heuristics and shuffle

Below we present the results of shuffling while using heuristics.

Average runtime

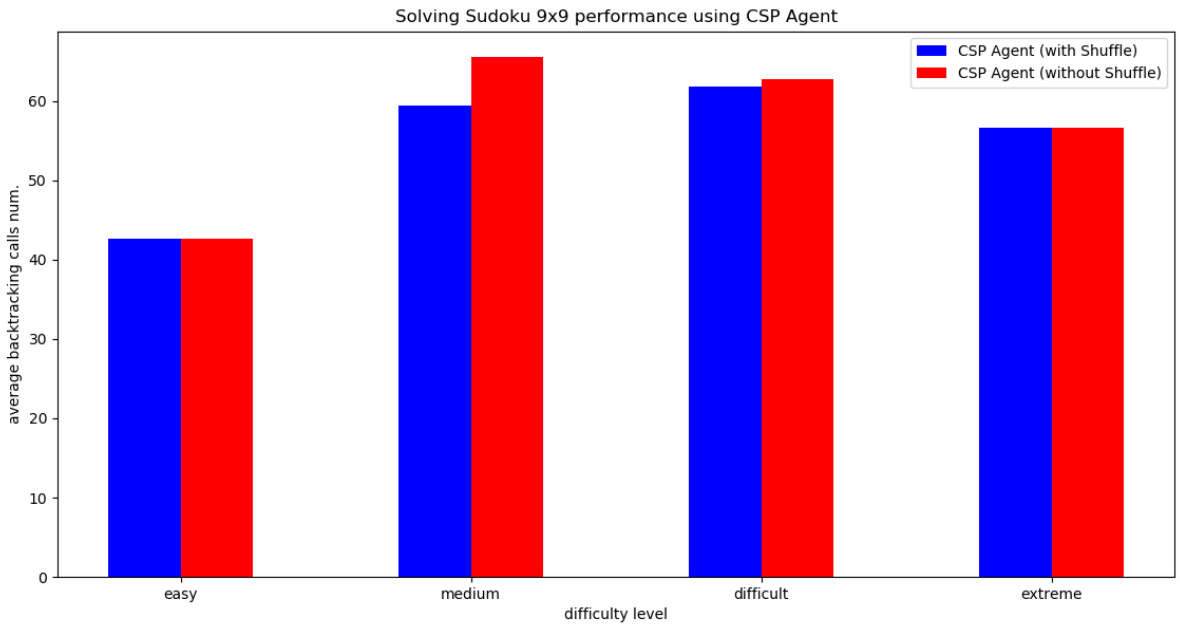


Observations:

Unsurprisingly, it does not take very long to run backtracking, with heuristics, on 9 by 9 Sudoku boards after the order of the cells has been shuffled. There is only a seconds difference between them. This is because the heuristics help

backtracking choose which variable and which value to choose next. Therefore their predefined order is less relevant.

Average Number of Backtracking Calls



Observations:

We notice the same phenomena that we noticed with the runtimes. The numbers are more or less the same because the heuristics help us choose variables and values. This helps prevent unnecessary backtracking and means that the number of times the algorithm backtracks is not significantly different.

Backtracking Summary:

Backtracking is a fast and very efficient technique for solving 9 by 9 boards. Once it tries to solve 16 by 16 Sudoku boards, it becomes slower, as we would expect. However, with heuristics backtracking's performance improves immensely and the 16 by 16 boards can be solved very quickly, in seconds, rather than minutes. As the size of the boards gets larger the heuristics prove themselves even more useful. A runtime which we were unable to even calculate becomes tractable and finishes very quickly.

Finally, our observation that shuffling the variables has marked effect on the performance of backtracking stood out to us particularly. It showed us how sensitive backtracking is not only to the size of the space of possible solutions which it must explore but to the inherent structure of the graph. The change in runtimes is truly remarkable going from seconds to hours, often even longer.

Local Search

Local search is an approach often indicated for optimization problems. Above we considered the Sudoku puzzle as a CSP and a graph coloring problem. However, we thought we could take a different approach, based somewhat on local searches relative success with the eight queens problem. Although it very rarely succeeds it comes close often enough.

How can we turn the Sudoku problem into an optimization problem? Our solution was to take the initial board and populate it randomly with all the elements needed in order to solve the board. So a provided Sudoku board would be filled in with a random and incorrect solution. The local search algorithm would then try to optimize the solution by swapping the values in two different cells.

The general idea is to take the state of a problem, generate all its successors and make the best possible choice, based on some criteria. Local search's biggest drawback is that it often gets stuck in local maxima. Meaning it arrives at a state and every successor is worse than the current one, however, the current state is not the best solution to the problem. For this reason we tried simulated annealing, which should be able to avoid this problem.

Simulated Annealing

Description:

Simulated Annealing conceives of an optimization problem as a heated material. At the beginning, when the system is *hot* if none of the available moves are significantly better, we permit with a high probability, based on the exponent of the change divided by the temperature, a suboptimal choice. This means that as the temperature gets lower suboptimal moves will be accepted with lower and lower probabilities. Allowing suboptimal moves at an early stage means simulated annealing explores the solution space more thoroughly making it less likely that the program will get caught in a local maxima. It also has the added benefit that at each iteration it only produces one random successor, rather than every possible successor.

Implementation:

We start with an initial sudoku board. This board is then populated with all the numbers needed in order to solve the board in randomly chosen cells on the board, i.e. in a solved 9 by 9 Sudoku board the number 2 appears nine times once in each of the nine rows so if the number 2 was only found 3 times in the initial board, the algorithm populates the board with the number 3 another six times in random locations. The same thing is done for every missing value. We place everything that is missing from the board in some random location so that Simulated Annealing can try to find the optimal placement for each cell. Optimality is defined by the number of collisions, where a collision is having the same number appear more than once in the same row or column or sub-grid. In this way we can turn simulated annealing into an optimization problem by trying to get it optimize/minimize the number of collisions on the board after it has been populated.

Simulated annealing's most interesting feature, as compared to general local search, is that it does not produce every successor. Rather it produces one at random. Our implementation does this by randomly choosing two cells on the board. Unless the cells are fixed, meaning they were initially a part of the board we are trying to solve, they can be swapped. We then compare the number of collisions in the current board as compared to the number of collisions in the board with the swap. If the swap has less collisions than the current board,

$$\Delta E := current\ \ number\ \ collisions\ \ - number\ \ of\ \ collisions\ \ after\ \ swap > 0,$$

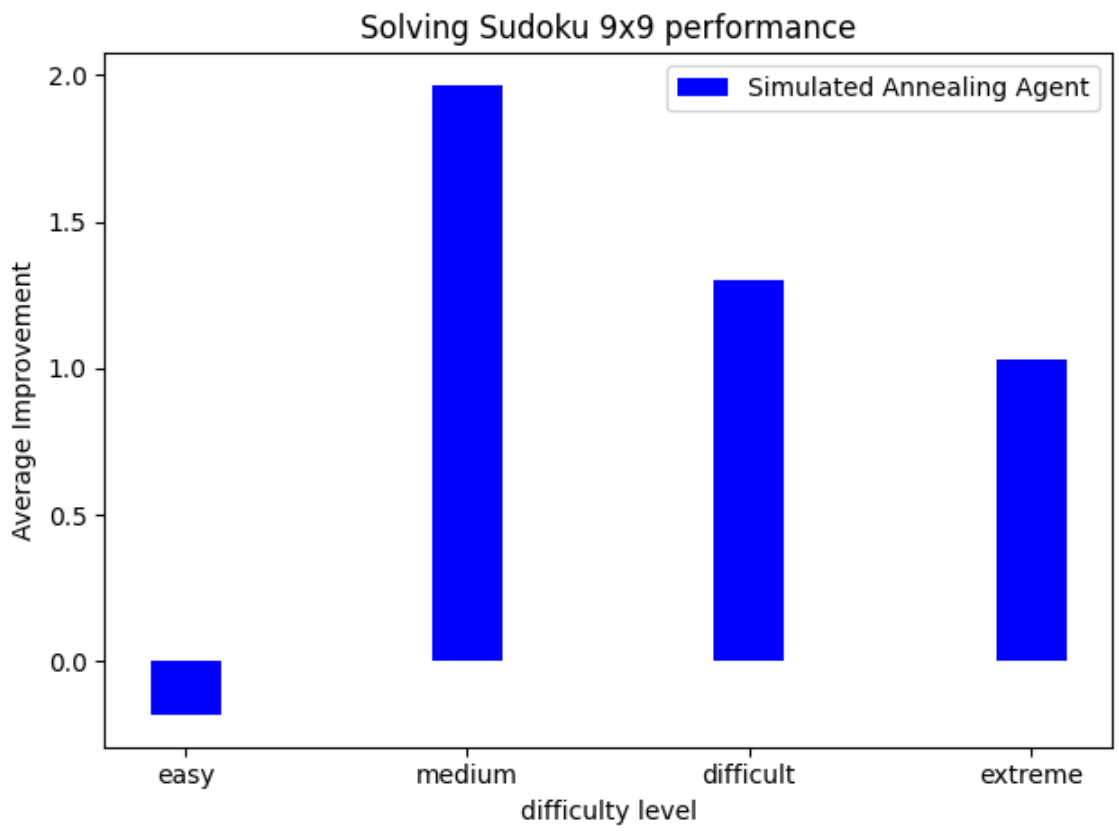
we do the swap. Otherwise the swap is suboptimal and does not improve the state of the board. However, simulated annealing is also characterized by its willingness to explore suboptimal moves at higher temperatures. Therefore, with a probability of $e^{\Delta E/Temperature}$ the suboptimal swap is accepted. As the system cools, the temperature goes down and the probability that we will choose suboptimal moves goes down.

After each iteration the temperature is decreased by some rate of decay, which corresponds to how slowly our system cools. A slower rate of decay means that the system will explore suboptimal moves for longer periods. We allow the user to control these variables from the command line.

If the temperature is high enough Simulated Annealing will always manage to find a global maximum. As we will see, despite its speed, simulated annealing needs a great deal of time to make significant improvements and we were never able to find a combination of parameters that would allow it to successfully solve a Sudoku board. Although often, like its performance in the n-queens problem, it usually made reasonable improvements and did so in a reasonable amount of time.

The Simulated Annealing algorithm defaults to a temperature of 10,000 and 0.99. These parameters allow for very quick runtimes although as the graph below shows the temperature is not high enough to make consistent or particularly good improvements to the initial number of collisions. Improvement in performance is measured by the average difference in the starting number of collisions and the final number of collisions. The graph below shows Simulated Annealing's performance with these parameters on all levels of 9 by 9 Sudoku boards.

Simulated Annealing: Average Improvement with default parameters



Observations:

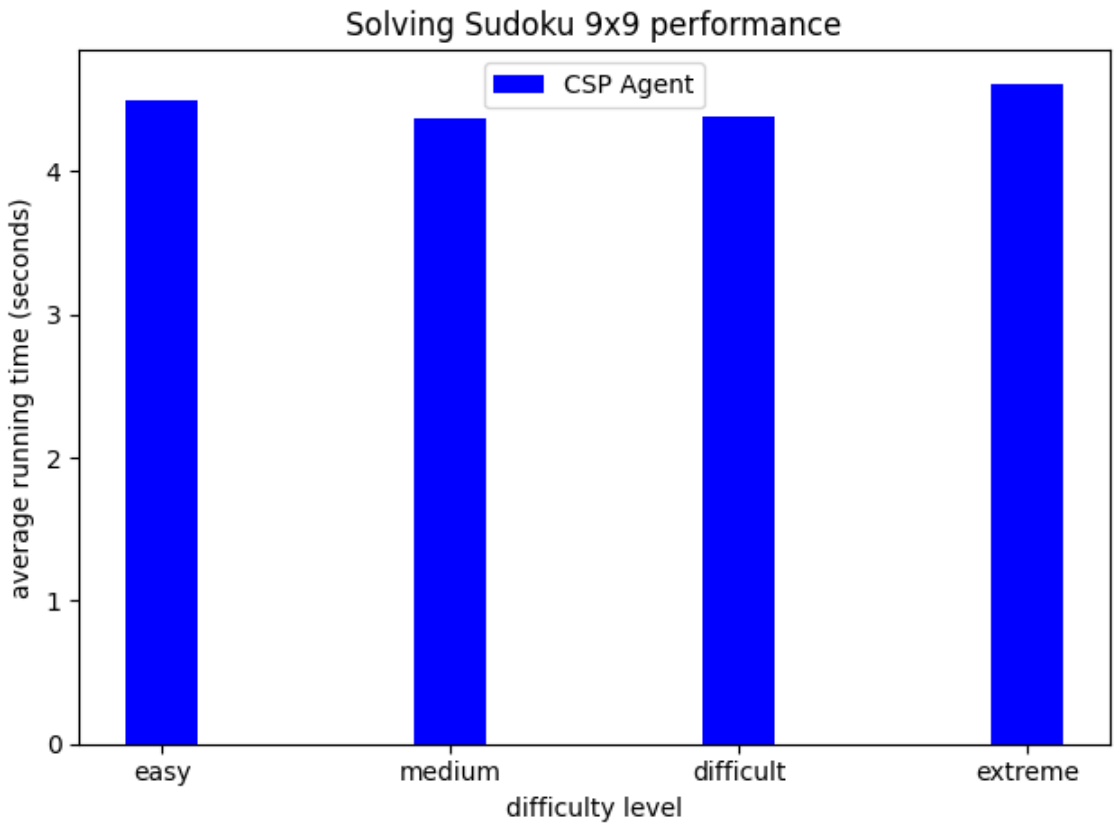
1. The fairly low temperature and the relatively high rate of decay made it difficult to improve the average improvement in the algorithm's performance. The algorithm remained bounded to a maximum average improvement of 2 collisions per board.
2. On the easy-level boards the algorithm makes things worse, it increases the number of collisions on the board. This seems to be because there is not enough time for the algorithm to recover from its initial bad moves.
3. Simulated annealing manages some consistent level of improvement with medium-level boards. However, as the level of difficulty increases its performance gets worse and worse. Which is to be expected. This does however, make its performance on the easy level boards rather hard to explain although we will attempt to in the following section, populating.

Populating and Randomness's Effect on Measurements:

In order to turn the Sudoku puzzle into an n-queens style optimization problem we populated the board with all the numbers required to complete the puzzle, placing them randomly on the board. This introduces a level of randomness because it means that the starting position, even in a difficult board is not consistent. A random placement can be favorable and be easier to solve than another and there is no way to guarantee the initial conditions. This makes our measurements rather tentative and we take them not as a strict representative but as a general idea of the algorithms performance. i.e. randomly populating a board of a particularly difficulty level may make it far more difficult to solve than another random configuration.

We suspect that this is the cause of the negative result simulated annealing gives on the set of easy boards, as opposed to the boards of greater difficulty. Our suspicion is that some property of the easy boards and randomly populating them makes it particularly difficult for Simulated Annealing to solve in the very short amount of time provided by the fairly low temperature of the default settings. i.e. because the temperature is so low and we allow suboptimal moves these properties allow Simulated Annealing to make moves which it does not have time to correct.

Simulated Annealing: Runtimes for default parameters

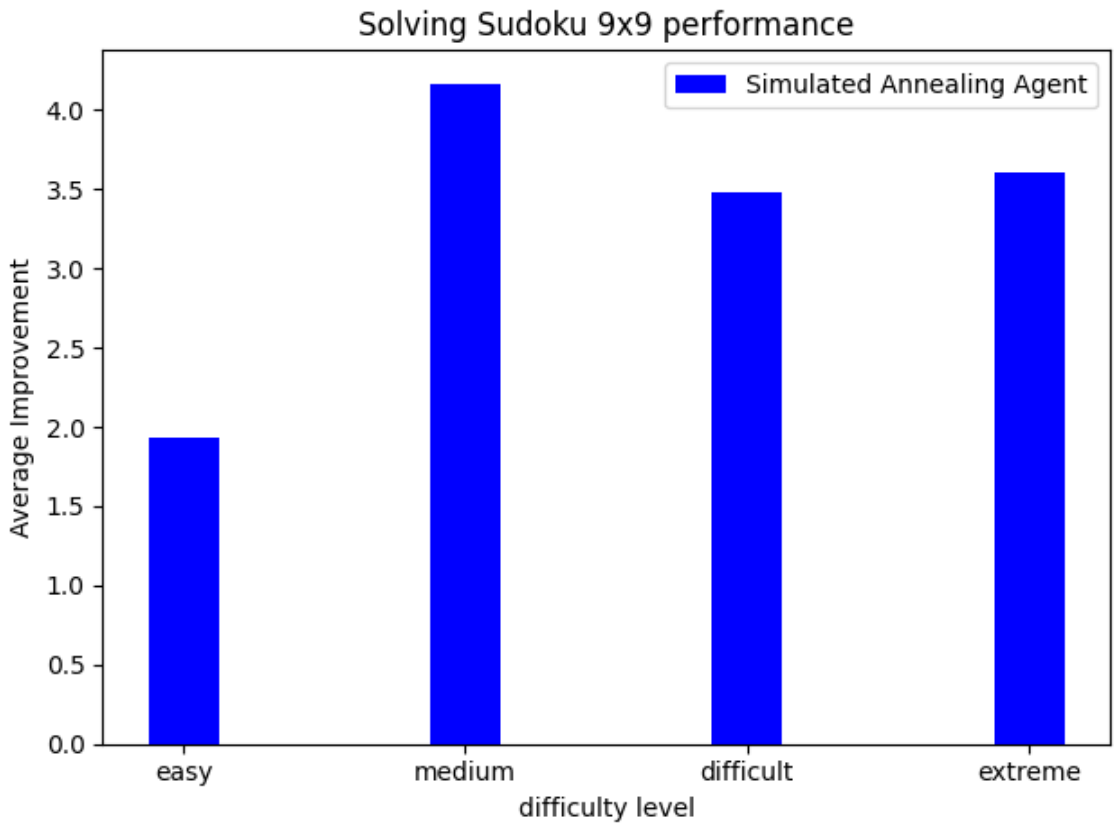


Comments:

We note that the runtimes stay flat across difficulty levels. This is not surprising as Simulated Annealing decays at the same rate across difficulty levels making this result rather expected.

Simulated Annealing - Different Parameters: Average Improvement

The following is a graph of Simulated Annealing's performance with the temperature set to 90000000 and the rate of decay set to 0.999.



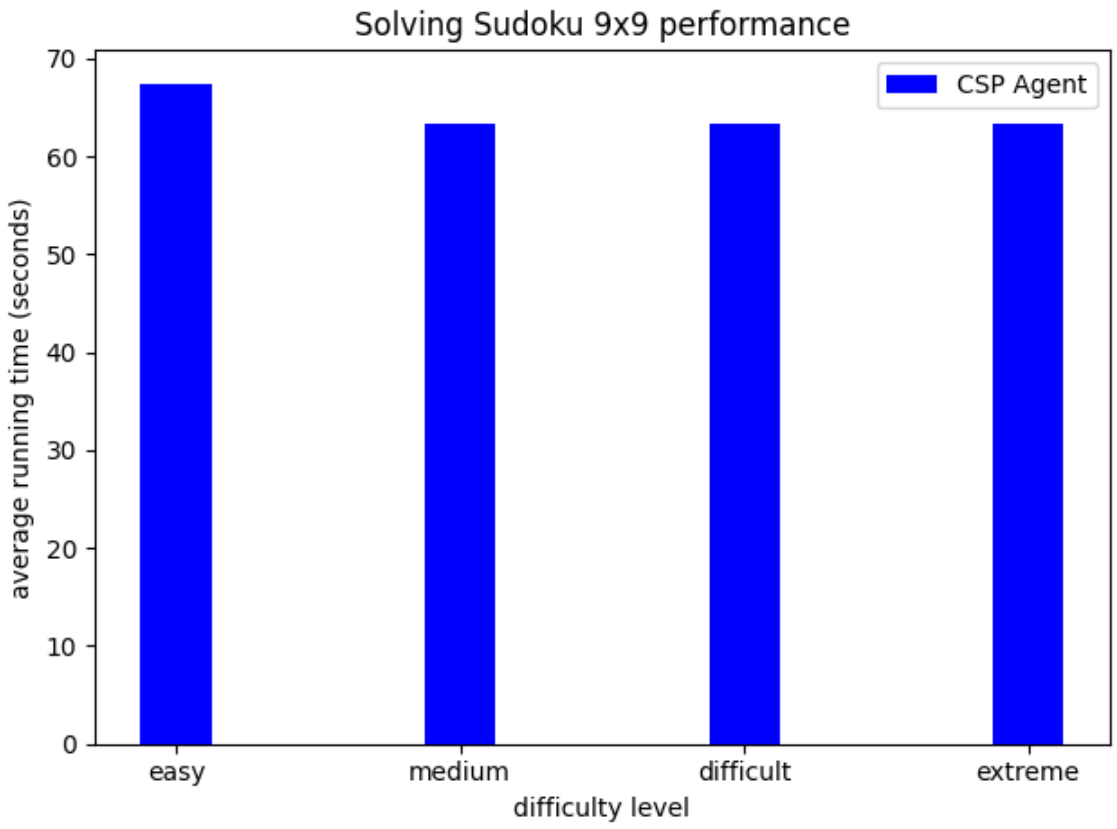
Comments:

1. We note that with a higher temperature and a slower rate of decay Simulated Annealing performs much better on the set of easy boards than it did with a lower temperature, where it did in fact make things worse.

2. While the previous graph showed Simulated Annealing had a worse performance for the extreme-level boards we note that with more time the algorithm manages to improve its performance on these boards. This may be due to luck, meaning that by chance there were boards which were initially populated with favorable initial configurations, however, this may also be due to the fact that the extreme level boards have more than one acceptable configuration. The answer on these boards is not unique, there is more than one goal state. This may allow Simulated Annealing to find a better configuration because there are more configurations that are acceptable to one of its several acceptable answers.

Simulated Annealing - Different Parameters: Runtimes

The following is a graph of the average runtimes for simulated annealing with the parameters above.

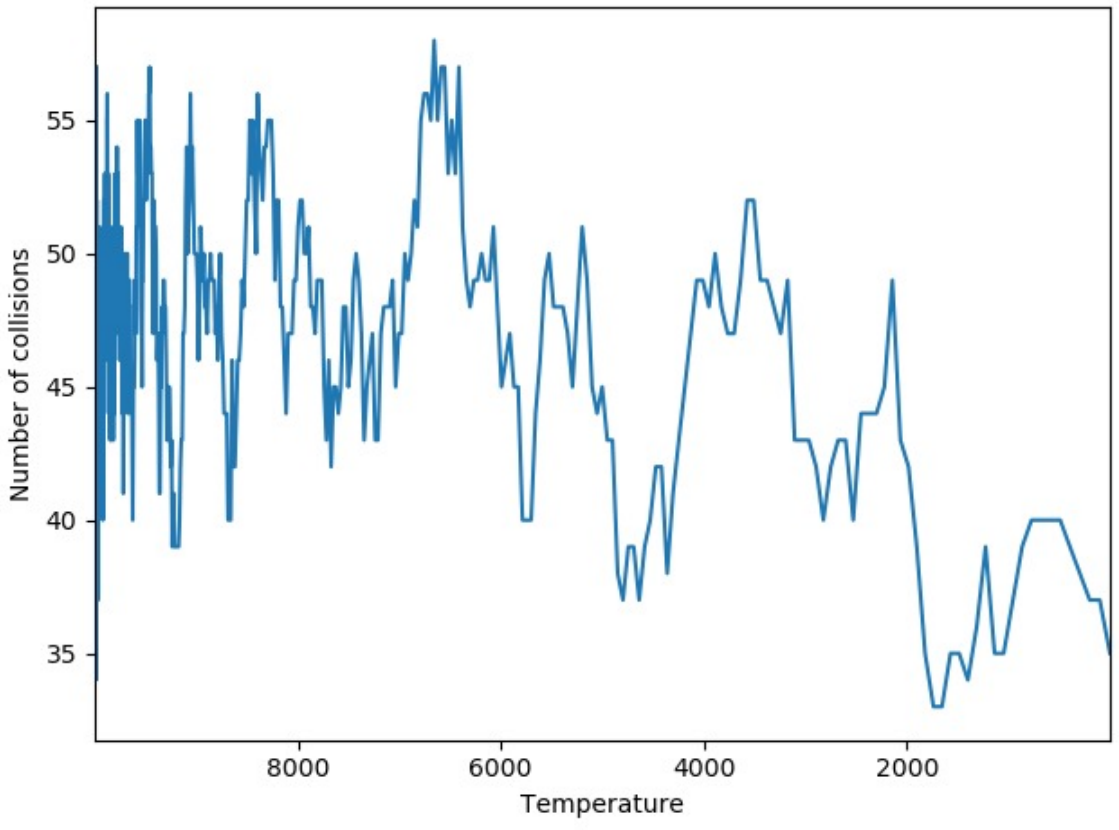


Comments:

With the increased temperature and smaller decay parameters the algorithm takes about a minute to finish for each board. Unsurprisingly, as run times do not depend on the difficulty of the board the run times remain very consistent, as noted above.

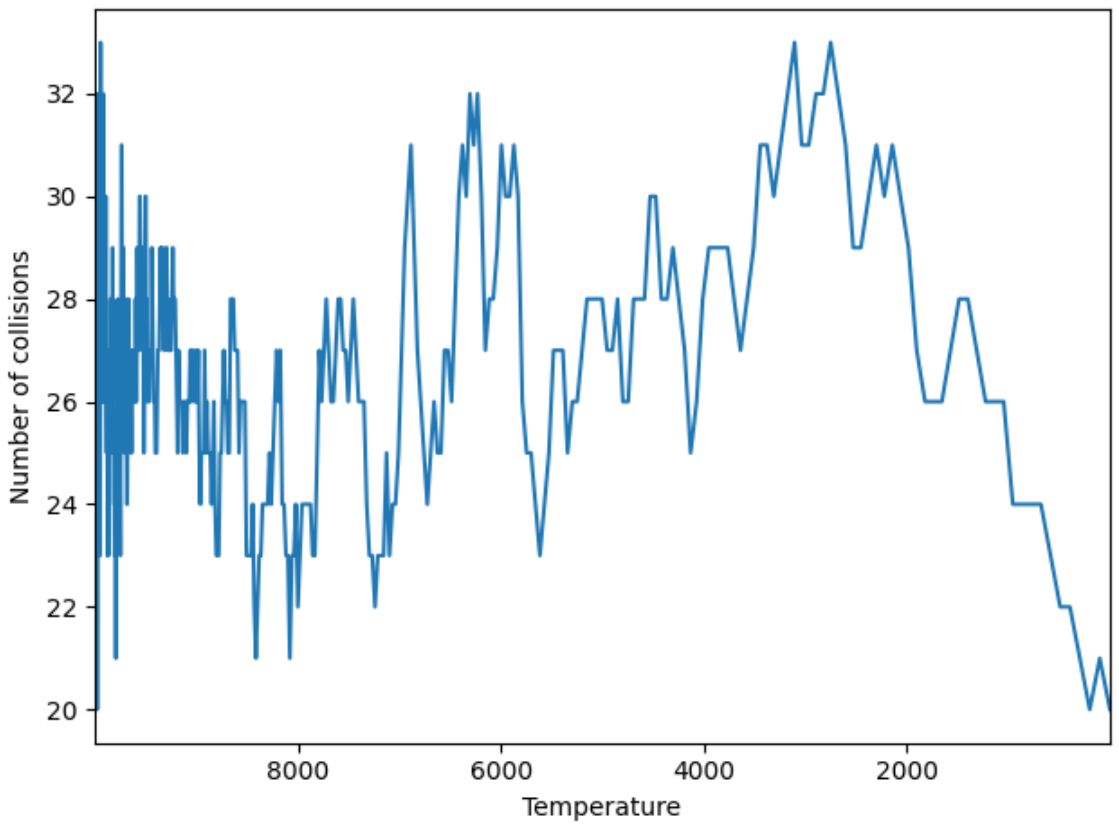
Simulated Annealing: Examples of Typical Behavior

The following are graphs of what simulated annealing's behavior looks, with the default parameters, as the temperature decreases.



Observations:

Initially the system starts with 52 collisions. At the initial temperature of 10,000 huge amounts of exploration and suboptimal choices are allowed. As the temperature cools fewer and fewer suboptimal choices are made and Simulated Annealing begins to track downward. This is a rather successful example of a run. It terminates with a final number of 36 collisions, which seems like a reasonable improvement. Often this is not so successful and the number of collisions is not significantly affected. As in the following example.



Observations:

This is a particularly unsuccessful attempt by Simulated Annealing. The initial number of collisions is 20. At termination, Simulated Annealing has increased the number of collisions to 24.

Explanation of Simulated Annealing's Performance:

Why is Simulated Annealing such an inefficient algorithm for this problem? We attribute this to how Simulated Annealing chooses its next move. It does not move to the best possible move of all possible successors. It produces a successor randomly and only accepts that successor if that move betters the situation. The space of possible successors is very large. It is bounded by the size of the set of possible swaps, which allows us to calculate the upper bound for the set of successors. We can find the size of the space of swaps by counting how many pairs of swaps can be created. As there is

no particular difference to order, swapping cell x with cell y is the same as swapping cell y with cell x, we choose 2 cells from 81 cells. Giving us a total of $\binom{81}{2} = 3240$ possible swaps. This is naturally an upper bound as not all the cells can be changed and so a swap with them is not permitted. However, such large numbers help us explain why simulated annealing struggles, as it does, to solve Sudoku puzzles. It is also true that a particular swap, while it may reduce the number of overall collisions, does not necessarily improve the situation. I.e. the odds of a particularly successful swap being randomly produced are not very high. Very often Simulated Annealing seems to play itself into a corner, i.e. while the board has not been solved there is no good swap, a minimum has been found, and without more time to make a suboptimal move away from this configuration Simulated Annealing fails.

Due to Simulated Annealing's failure to solve any 9 by 9 board, we decided not to run it on boards of other sizes. In order to acquire meaningful results the runtimes would have to have been enormous and with the available time and equipment restraints it seemed unnecessary to proceed to larger boards.

Stochastic Beam Search

Description:

A k - beam search tracks some finite set of randomly generated states and at each iteration produces all the successors of each states and attempts to choose the best of successor of each state. However, what characterizes the beam search is that information is shared between the various beams and unsuccessful beams are abandoned in favor of states that seem more promising. While this is promising it can produce a similar problem as local search in so much as the separate beams quickly congregate together and remain together, rather than explore the space separately. This is similar to the way local search gets easily stuck in local maxima or minima. Everything clumps together and the space is not very fully explored. For this reason we attempted a variation of k - beam search called stochastic beam search.

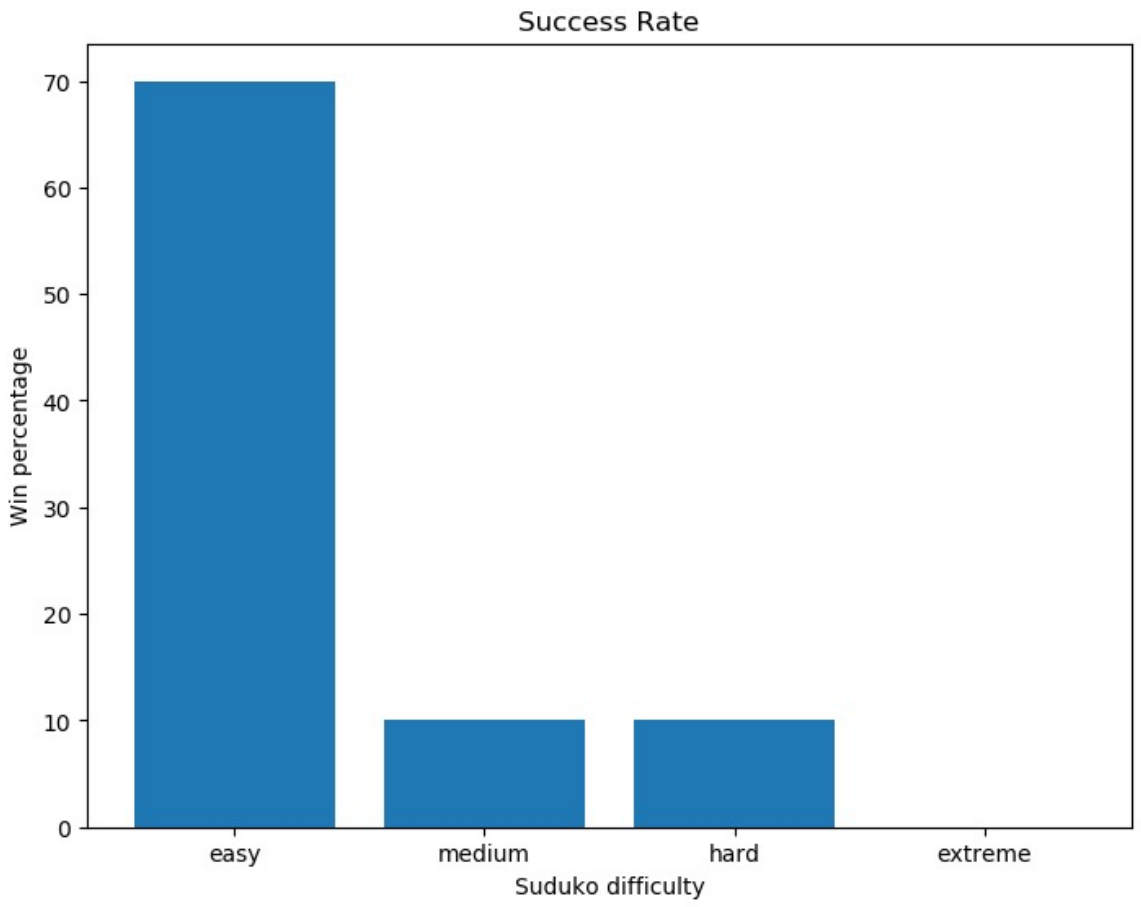
Implementation:

Stochastic beam search begins by taking a sudoku board and producing ten copies and populating each of them separately. At least in part, this allows us to avoid the problem we noticed with simulated annealing, that particular random starting configurations may seriously disadvantage performance. On each iteration we produce ten successors for each board and with a probability of 80% we will accept the best successor and with a probability of 20% we check a random successor.

If the current state was not improved by any of the ten randomly produced successors, with a probability of 20%, we accept a suboptimal successor. This allows us to avoid getting stuck at a local minimum. If we have taken a suboptimal move, we will check whether there is an improvement over our current state, if it is we accept it as the next state. Because improvements are only accepted with an 80% probability this means that an optimal move may have been rejected. So we perform a check at this stage to see if it can be improved. Otherwise, if the successor is not an improvement, with a probability of 20% we restart the board, by creating a new board, repopulating it and replacing the current board with the new one. (The probabilities used were chosen as the result of some tinkering because they seemed to produce the best results. However, due to the runtimes involved we were unable to test the performance of this algorithm with large variations in these parameters.) This process continues for a maximum of 100,000 iterations.

Stochastic Beam Search: Success Rate

Unlike simulated annealing our stochastic beam search algorithm did successfully solve sudoku boards. The following is a graph of the rate of percentage of Sudoku boards solved by level of difficulty.

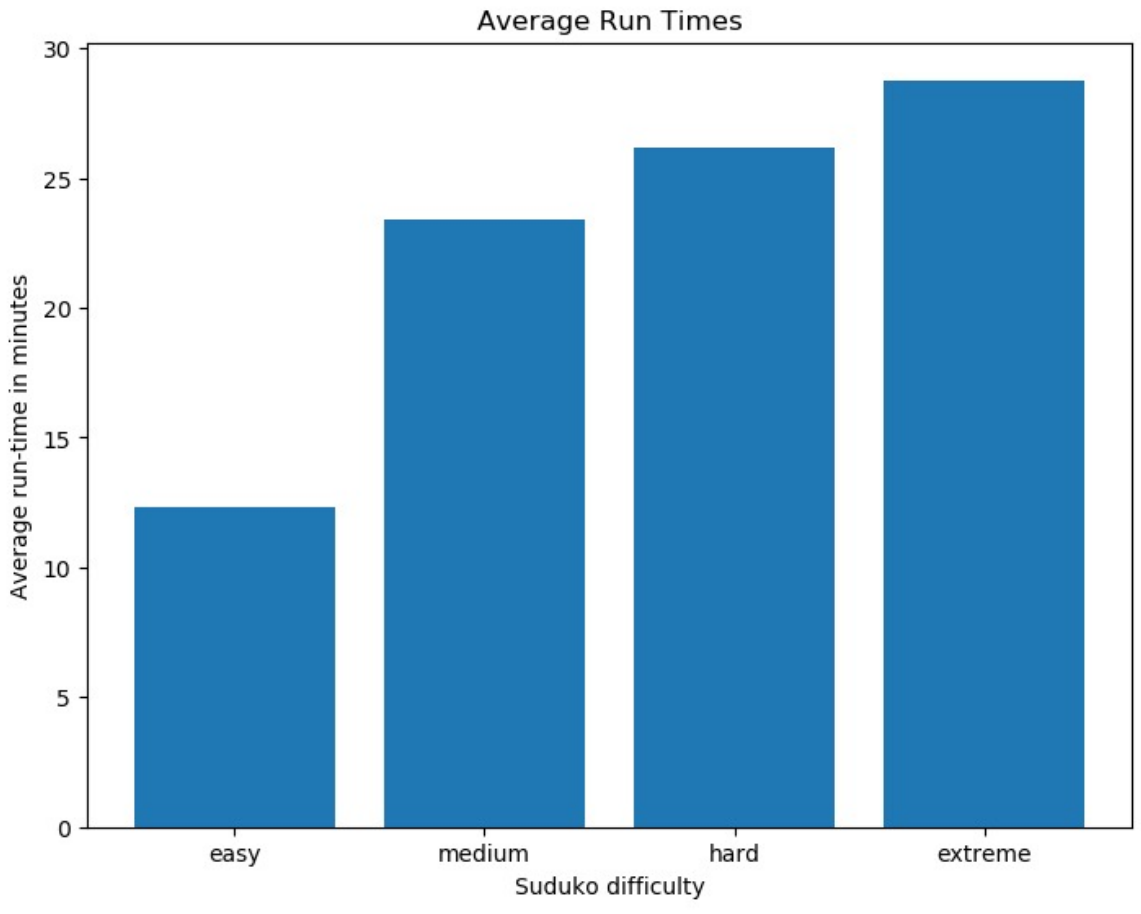


Observations:

1. The algorithm succeeds remarkably well on the easy level boards, which is rather surprising given what we observed with Simulated Annealing's performance with easy level boards.
2. There is a remarkable drop off in performance from the easy level to the medium and difficult levels and the algorithm was unable to solve a single board from the extreme level.

Stochastic Beam Search: Average Run Times

The following is a graph of average runtimes per level of difficulty.



Observations:

1. We note that the run times for easy level boards are significantly lower than the runtimes for the rest of the levels. It is here important to remember that the algorithm is limited to 100,00 iterations. However, because easy success so often it does not have to use all these iterations. The rest of the levels runtimes increase as a function of their success

rate. The less successful they were the more time was required. Because the extreme level boards are never successful, each board uses all of the iterations, and therefore they require the most time.

Summary of Stochastic Beam Search:

For the most part it seems that stochastic beam search works because it allows improvements and when it sees that improvements can not be made one of two things happens, we allow a suboptimal move ensuring that we avoid local minima as much as possible, and we repopulate the board. As we noted previously, one of the difficulties with succeeding and gauging the performance of Simulated Annealing was the randomness of the initial configuration. This randomness introduces a lot of noise and uncertainty into the algorithms behavior. However, when at intervals we create a new random configuration, we can avoid disadvantageous initial configurations. To these two qualities we attribute stochastic beam search's success over simulated Annealing.

Project Summary And Conclusions:

We attempted to solve Sudoku puzzles using backtracking and variations of local search. As we progressed through backtracking we noticed that it was a highly successful and efficient algorithm, even for large boards when it had been outfitted with a variety of heuristics. However, local search algorithms performed very badly generally and by comparison. Although we attempted to increase run times, in simulated annealing, by increasing temperature and lowering the decay, we were still unsuccessful. However, once we had tried to find a way to overcome Simulated Annealing's shortcomings for this particular problem we were able to find some limited success.

We conclude that Sudoku puzzles are to be thought of as a graphing problem and not an optimization problem. The failure of local search methods to even manage a high and consistent success rate shows that using local search amounts to randomly producing an answer and hoping that the produced is close enough to a solution so that a fairly small number of swaps later we can arrive at the solution. In essence, this means we are hoping to luck our way into a single point in a massive solution space, which as we have shown by repeatedly failing to do so, is highly unlikely.

Appendix and Miscellaneous Comments

- Throughout this project we suffered a lot of technical issues with our personal devices. For this reason the runtimes may be different on other computers.
- We relied on the material in the course textbook and the course notes.