By Mouhannad Al-khalkhali
s3749653

Advanced Solver

The approach taken to implement an advanced solver for the Killer Sudoku grid involved; building upon the foundation of the backtracking algorithm through the addition of some smart heuristics. The backtracking algorithm contained various inefficient methods that could be altered to increase the efficiency.

The backtracking algorithm looped over the grid systematically. The process consisted of the algorithm progressing row by row from left to right

The primary issue with this process was the validation for the cage. A cage could only be verified once all the cells of the cage had been filled, or if the total of the cells already within exceeded the total required.

For instance, a cage consists of two cells, the first cell in the grid (top left) and the one under it. That cage's validation cannot be confirmed until the loop reaches the second cell. Thus, in a 9X9 grid, it had to verify 10 cells before it could check the cage constraint, whilst each time calling the validator method for each potential value. Meaning if no value satisfied the 10th cell, it had to backtrack all the way to first cell, and then the cycle started anew. In contrast the advanced solver retrieved a collection of cages, then looped over each cage respectively. This worked much more efficiently, as provided an option of early termination should a grid have not been filled.

Since each cage is filled from the same collection as the validator; when the validator checks if the cell is there, it can terminate the loop if it is not. This is due to the validator knowing that this constraint can't be checked as it had not been filled yet. This changed the time complexity of the validation of an empty grid from O(n) to O(1) and increased the overall efficiency for all other verification.

The validation methods of the backtracking algorithm involved checking if the whole grid was valid. This resulted in a worst-case time complexity of O(n) for each constraint. Since each cell is checked individually, there is no requirement to check the whole grid each time a cell is allocated a value. Hence, the advanced solvers validation method was refactored to take coordinates of the cell as a parameter as well as a cage. With these coordinates, the row constraint only checks the row of the cell and ensures it is valid, the column only checks the column it's in, and the box constraint does the same. This results in a worst-case time complexity of each validation to go from O(n) to $O(n^{1/2})$ for the row, column and box constraint. Additionally, a similar addition was added to the to the cage constraint. Instead of checking every cage and making sure they are all valid, it only checks the cage the cell belongs to. The efficency increase of this is more subject to the number of cages in the sudoku. ultimately the new validation methods greatly increasing the efficiency of the algorithm.

Furthermore, a similar princinple to that of the early termination in bubble sort was implemented. The advanced solver always kept track of the current total of a box with given values, which allowed it to check if adding the input makes the total greater than the

required, if it did, it terminates that loop. This avoids checking values unneccesarily which would have lead to the function being called recursively as well ass the validation loop, reducing the efficiency of the algorithm

The order of checking constraints as well as the data structures used also played a large role in the solve speed of the algorithm. After rearranging the constraint checks it was found checking the box followed by row, then column and lastly cage constraint was the fastest order considerably hence the implementation.

Order	Avg time with killer_99_cs.in. (seconds)
Box -> column -> row -> cage	54.3
Box -> row -> column -> cage	62.3
Box -> column -> cage -> row	72.5
Box -> cage ->column -> row	84.3
cage -> column -> row -> box	59.2
cage -> column -> box -> row	75.3
cage -> box -> row -> column	87.3
cage -> box -> column -> row	73.2

Note: Starting with the row or column was considerably slower so it was neglected int the results.

The use of a stack instead of an arraylist for storing values already in the grid(when calling the validator) resulted In better run times for the algorithm. This was as the stack had an insertion and deletion of O(1). In contrast the arraylists time complexity of O(n) was considerably slower making it ultimately less efficient then the stack.

The advanced solver built upon the fundamentals of the backtracking algorithm with the addition of constraints and principles to solve the grid at a great speed. Through avoiding redundant cell constraint validation, placing values cage by cage and avoiding unnecessary testing of values and early termination, it was able to solve the killer_99_cs.in test case in 53.4 seconds, and the kill_99 in 0.16 seconds compared to the backtracking algorithm that did it in 2640 seconds and 2.3 seconds respectively.