**Falah Alhashmi**

**ISTA 450**

**Tyler Millhouse**

**December 12th 2023**

**Final Project – 3x3 Number Puzzle and Sudoku Solver**

The premise of the project was to develop a simple of utilizing cs to solve certain cs problems. Initially, what was utilized was a simple 3x3 number puzzle, where numbers between 0 to 8 were utilized, with 0 signifying an empty block. The cs that were deemed the best for such a model were the backtracking and forward checking, and initially constraint propagation was considered, but then the arc3 was utilized instead. The purpose was to see which of the three algorithms would work best, and success was focused on first, and then time efficiency. Upon the realization that might have been a little challenging initially, with certain code taking very long to render as I attempted to use multiple iterations and generations of 3x3 "matrix" to be solved and have their performance be judged, the "matrix" was eventually changed to be a 2x2, which heavily assisted with debugging efforts through decreasing of time needed for tests. How the code would work is that it would first generate a random matrix to be unscrambled into the goal state, which would be [[1,2,3],[4,5,6],[7,8,0]]. If for any reason that matrix took too much time to solve, it would generate the next puzzle, and so on. If a puzzle were to be solved, it would count as a success, and if not then a failure, and the success percentage would be determined by percent = successes / total. Skipping was intended for the purposes of not having a code that was stuck, but more importantly, was for focusing on solving as many puzzles as possible given a specific time frame.

Backtracking works through two definitions in the code. backtracking\_search\_util recursively explores possible assignments, and backtracking when constraints are encountered, while accounting for explored sets to decrease visitations, and if the goal state is achieved, a solution is returned. backtracking\_search determines how many maximum attempts to be made so as to not waste too much time on a generated matrix that would simply be too difficult, and backtracking\_search\_util is called iteratively until a solution is achieved or the limit is reached, and then the puzzle is checked to see if the goal state and the puzzle state match. Forward checking also has two definitions, one is a recursive function that does forward checking, and one that does forward checking to find a solution given an initial state. Arc-consistency AC-3 utilizes a function implementing the algorithm, and a function to initialize a queue of arcs, another function to enforce arc consistency by eliminating inconsistent values, a function to return neighboring nodes of a given node, and a function that gives the position of an empty cell. The algorithms are tested through one last function test\_algorithm and then their efficiency can be compared. As the script seems to have done a good job at solving simpler puzzles, that code baseline was utilized for a project more complex, which was a sudoku solver. While the method of comparing the three CS methods, will remain the same, more complexity is added through the grid now being a 9x9 instead of a 3x3. Functions are added to best deal with timeouts, and also generating sudoku puzzles.

def backtracking\_search\_util(code, goal\_state, explored\_states):

    if is\_goal\_state(code, goal\_state):

        return code

    zero\_position = find\_zero\_position(code)

    for value in set(goal\_state[zero\_position[0]]) - set(code[zero\_position[0]]):

        new\_code = [row[:] for row in code]

        new\_code[zero\_position[0]][zero\_position[1]] = value

        if tuple(map(tuple, new\_code)) not in explored\_states:

            explored\_states.add(tuple(map(tuple, new\_code)))

            result = backtracking\_search\_util(new\_code, goal\_state, explored\_states)

            if result is not None:

                return result

def forward\_checking\_util(code, goal\_state, remaining\_values):

    if is\_valid\_assignment(code) and is\_goal\_state(code, goal\_state):

        return code

    zero\_position = find\_zero\_position(code)

    for value in remaining\_values:

        new\_code = [row[:] for row in code]

        new\_code[zero\_position[0]][zero\_position[1]] = value

        updated\_remaining\_values = remaining\_values - {value}

        result = forward\_checking\_util(new\_code, goal\_state, updated\_remaining\_values)

        if result is not None:

            return result

    return None

def arc\_consistency\_ac3(code, goal\_state):

    queue = initialize\_queue(code)

    while queue:

        x, y = queue.pop(0)

        if revise(code, goal\_state, x, y):

            if not code[x[0]][x[1]]:

                return None  # Inconsistent assignment

            neighbors = get\_neighbors(x)

            for neighbor in neighbors:

                if neighbor != y:

                    queue.append((neighbor, x))

    if is\_goal\_state(code, goal\_state):

        print("Arc-Consistency (AC-3): Solution found -", code)

        return code

    return None

The results for the 3x3 number solver were actually rather close to each other. For the backtracking algorithm, finding the solutions for 100 of the puzzles required 0.00599 seconds, which was rather rapid, and a success rate of 100%. For the forward checking, a longer time was required which was 0.00299 seconds, and a success rate of 100%. The arc-consistency had a success rate of 100% as well, with a total run time of 0.00503 seconds.

Backtracking Search: Solution found - [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

Backtracking Search: Solution found - [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

Backtracking Search: Solution found - [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

Backtracking Search:

Success Rate: 100.00%

Failure Rate: 0.00%

Total Time: 0.04713 seconds

Forward Checking:

Forward Checking: Solution found - [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

Forward Checking: Solution found - [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

Forward Checking:

Success Rate: 100.00%

Failure Rate: 0.00%

Total Time: 0.02022 seconds

Arc-Consistency (AC-3):

Arc-Consistency (AC-3): Solution found - [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

Arc-Consistency (AC-3): Solution found - [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

Arc-Consistency (AC-3):

Success Rate: 100.00%

Failure Rate: 0.00%

Total Time: 0.03606 seconds

No progress made, generating a new initial state. Attempts left: 10

No progress made, generating a new initial state. Attempts left: 9

No progress made, generating a new initial state. Attempts left: 8

No progress made, generating a new initial state. Attempts left: 7

No progress made, generating a new initial state. Attempts left: 6

No progress made, generating a new initial state. Attempts left: 5

No progress made, generating a new initial state. Attempts left: 4

No progress made, generating a new initial state. Attempts left: 3

No progress made, generating a new initial state. Attempts left: 2

No progress made, generating a new initial state. Attempts left: 1

No progress made, generating a new initial state. Attempts left: 0

Backtracking Search (Sudoku): No solution found after multiple attempts.

Success Rate: 0.00%

Failure Rate: 100.00%

Backtracking Search (Sudoku):

Success Rate: 100.00%

Failure Rate: 0.00%

Total Time: 0.00000 seconds

Forward Checking (Sudoku): No solution found.

Forward Checking (Sudoku): No solution found.

Forward Checking (Sudoku): No solution found.

Forward Checking (Sudoku): No solution found.

Forward Checking (Sudoku): No solution found.

Forward Checking (Sudoku): No solution found.

Forward Checking (Sudoku): No solution found.

Forward Checking (Sudoku): No solution found.

Forward Checking (Sudoku):

Success Rate: 100.00%

Failure Rate: 0.00%

Total Time: 0.00000 seconds

Arc-Consistency (AC-3) (Sudoku): No solution found.

Arc-Consistency (AC-3) (Sudoku): No solution found.

Arc-Consistency (AC-3) (Sudoku):

Success Rate: 100.00%

Failure Rate: 0.00%

Total Time: 0.00000 seconds

To analyze these results, we need to understand what these cs algorithms actually do. Backtracking is a search algorithm that is frequently used to find a solution to problems that could be solved through systematic exploration of all possible configurations. This algorithm was chosen due to it matching what our type of problems are. The benefit of this algorithm is that pruning the search space makes it substantially quicker to find a solution, as unnecessary branches that would not have yielded a solution would no longer be considered. This algorithm also is thorough, ensuring that a solution is guaranteed to be found, given that it exists. Forward checking works by eliminating inconsistent values from the domains of variables as soon as a value is assigned to a variable which greatly reduces the search space. The constraints applied are accounted for to remove variables. When potential failure is detected, which can appear after an assignment of a variable, the domain of any variable becomes empty, this tells the algorithm that the current assignment cannot possibly have a solution, which triggers backtracking. The previous point is backtracked to, and alternative paths are taken until a solution is found. A benefit of such an algorithm is the operation of backtracking being more efficient. The reduction of domain sizes early better results are shown, and with adaptive processing, the constraints and domain information are constantly being updated.

Arc-Consistency 3 or AC-3 is an algorithm used to enforce arc consistency, which means that every value in the domain of a variable is consistent with the domains of its neighboring variables, given the conditions specified. The three types of arcs that have to be maintained between each variable are the Unary Arcs, Binary arcs, and ternarny Arcs. The Unary arcs involve constraints on a single variable, which could be used in specifying that a variable has to have a certain value. Binary arcs revolve around values between two variables and their non-conflict. This could be of a request in a prompt such as for whatever reason, "the number 2 and 3 cannot be next to each other". Ternary arcs act the same way as binary arcs except with three variables instead. The algorithm works by iteratively processing a queue of arcs, looking at each arc and making sure that the values are consistent by removing ones from the domains of the variables analyzed. If a domain is decreased, the queue receives more arcs to be analyzed. This proceeds until a solution is found by exhausting all reductions, and if there are no solutions, the domain of variable will be empty. Obvious benefits of the algorithm is efficient searches, early detection of inconsistencies, domain reduction, and global consistency, as well as its adaptability towards multiple csps, making this a good algorithm to be used.

As such, my wacky theory as to why the forward checking had the best results is that the backtracking was the most simple method, sacrificing pruning capabilities but still being reliable in terms of finding a solution. AC3 on the other hand, was too complex for such a problem, yielding the results seen, and giving it second place in terms of time required. Forward checking had the best results, due to its balance of complexity, pruning and reduction of search space. This finding was genuinely interesting in that it demonstrates the importance of choosing the correct cs for the specific csp.

However, the results of the sudoku solving were not very promising. The direct implementation of the original puzzle solver to a 9x9 proved to be very difficult in producing proper results. Even given an allotted depth of 1000 for backtracking, the failure rate was 100% given multiple generated states. No solutions were able to be found. The results would be similar for forward checking and even AC3. Moreover, there were ample challenges in getting the code to work, with errors appearing frequently. This would demonstrate that more constraints would have to be used to refine the search. As there was a desire to maintain a limit on how many iterations or time passed on a generated state, a different module had to be implemented. Obtaining consistent outputs was not a huge challenge in that generally, the trend was correct of which constrain solving algorithm was the best, but there was large variance in between test runs, which reveals that a different metric, such as the success rate to be more important would be a useful consideration for future projects.

Perhaps more iterations or time allotted for certain generated states would be given such that the model would confidently and accurately determine there to be no solutions, for both debugging and calculation purposes. A balanced approach between combining the two metrics of success percentage with time needed for a specific number of iterations could be incorporated. For issues of computation, parallelization could be used to utilize multiple processors, which would ideally aid with time spent debugging. For issues of code, hidden singles could be used, where candidates in each empty cell of a grid would be remembered. The candidates would be determined by the constraints of the grid which would be the numbers in the same row, column and 3 x 3. The hidden single would appear only once as a possibility in the smaller 3x3 or row or column, because it would be the only valid option for that cell. A naked pair strategy could be utilized as well. A naked pair happens when two cells in a specific row or subgrid or column have the same two candidate numbers, and in that row or subgrid or column, no other cells contain those two. With this, the naked pair’s candidate numbers can be eliminated from the possibilities of other cells in that row or column or subgrid.

More advanced strategies include X wing which is the creation of an x shape by aligning two rows and two columns, allowing for the elimination of candidate numbers in particular positions. This is known as candidate exclusivity where a situation is found where a candidate number is looked for can only appear in two columns in two different rows, or two different columns in two different columns. The candidate would form a rectangular shape. Once the x wing pattern is found, the candidate can be removed from the cells outside the X. This is because the candidate must appear in only two columns, forcing it to appear in the mutual rows. The swordfish technique, which is an even more advanced x-wing focuses on three columns or rows. The candidate has to be in exactly two positions within each of the rows or columns. In each row, cells where a specific candidate can be placed will be marked in each row or column, then the three rows or columns will be checked for a swordfish pattern formation which are that the candidate should appear exactly twice in each of the three rows or columns. Finally, an XYZ wing would be looked at to improve the algorithms, where there are three different candidates and three different cells where the cells must form a specific pattern and the candidates appear exactly once in each row, column or block. It is paramount that there are only two ways that these three cells can be filled with three candidates. When an XYZ wing has been identified, a candidate can be removed from the fourth cell that shares either a subgrid, row, or column with all three cells in the XYZ wing.

These techniques would all be used within the different algorithms and tested to see which ones would work best with the particular algorithm. As it has been demonstrated earlier, with different complexities and algorithms come certain considerations to be made that could either harm or benefit the efficiency of the algorithms.