Benjamin Heinze, Braxton McCormack

Ben.c.heinze@gmail.com, braxton.j.mccormack@gmail.com

Group #8

6 September 2023

Project 1 – Sudoku

System Requirements

The primary goal of Project 1 is to give us a firsthand introduction to artificial intelligence. By using five variations of searching algorithms, we aim to solve the popular game Sudoku. Artificial intelligence is optimal for solving sudoku due to how there is only one correct solution per puzzle, and using brute force to solve such a puzzle is time consuming, costly, and inefficient. Thus, we will develop a flexible, high-level structure to easily test individual algorithms and puzzle-difficulty. The solver must handle easy, medium, hard, and evil Sudoku puzzles. The following requirements have been provided below:

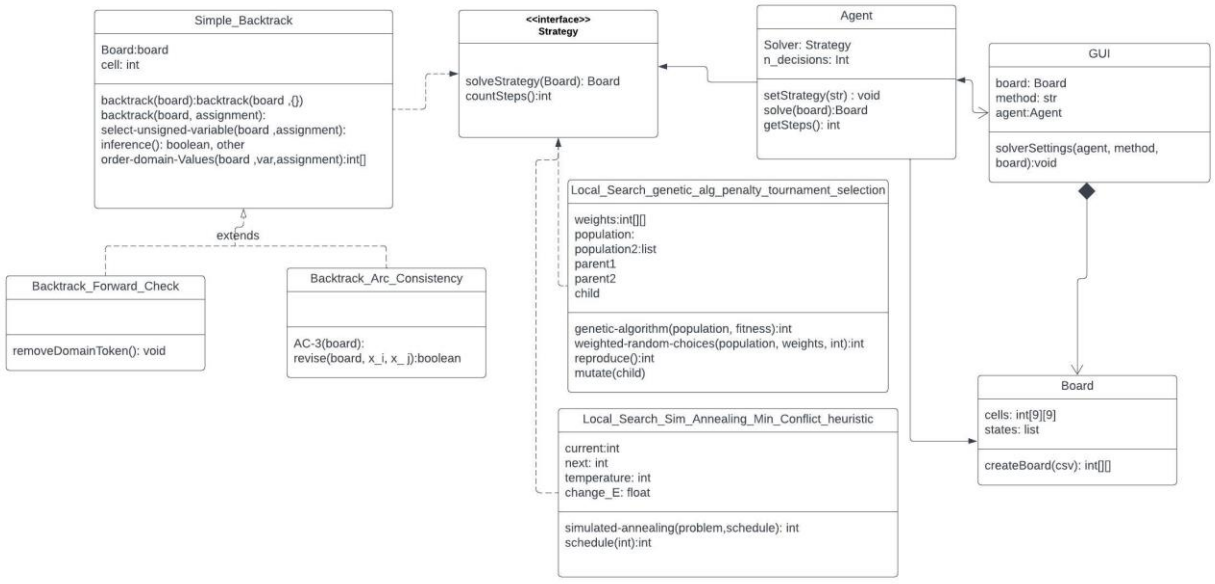
1. Implement a puzzle importer using a standard puzzle format.
    a. Understanding: We will develop a puzzle importer that accepts puzzles in a specified format.
    b. Implementation: Our driver will import the puzzles into a specified puzzle class.
2. Create a high-level architecture schematic that can effortlessly switch puzzles and strategies.
    a. Understanding: We will use object-oriented design with a clear separation between puzzle handling and solving algorithms.
    b. Implementation: We will have separate classes for our board, agent, driver, and algorithms.
3. Implement three separate iterations of the backtracking search: simple backtracking, backtracking with forward checking, and backtracking with arc consistency.
    a. Understanding: We will implement three backtracking algorithms.
    b. Implementation: These algorithms will be implemented with backtracking and then improved upon with forward checking and arc consistency, respectively.
4. Implement two search algorithms: simulated annealing with the minimum conflict heuristic and a genetic algorithm with a penalty function and tournament selection.
    a. Understanding: We will implement two algorithms incorporating simulated annealing with a minimum conflict heuristic and we will implement a genetic algorithm with penalty functions and tournament features.
    b. Implementation: Each algorithm will be designed to optimize Sudoku puzzle solutions using their respective techniques.
5. Run experiments and track the number of each algorithm's decisions to measure algorithm performance.
    a. Understanding: We will design experiments to test our algorithms and record various metrics such as the number of decisions made.
    b. Implementation: Our algorithms will log performance data by incrementing a counter for each epoch counting the number of decisions for a solution.
6. Write a paper summarizing the results of your experiments.
    a. Understanding: We will document the experiment setup, results, observations, and conclusions.

   b. Implementation: The paper will include sections on our methodology, results, observations, and conclusion.
7. Create a video demonstrating the functioning of the code.
   a. Understanding: We will create a video that showcases how our code works with examples of each algorithm running with various puzzles.
   b. Implementation: The video will include a walkthrough, explanations, inputs, outputs, and demonstrations.

Understanding of Requirements

To meet these requirements, we will create a Sudoku solver that reads puzzle files, applies various search algorithms, and records the decision-making processes. Since there are different difficulties for the sudoku puzzles, we will need to judge how each algorithm performs the various difficulties. Expanding on the video portion of the requirements, we will spend time on production quality while maintaining the constraints given. The constraints include restricting the ability to fast forward when showcasing the source code. We must make it as transparent as possible by showing every input and output as well. If we are unable to reach results for specific algorithms, we need to explain where we went wrong and how we could have avoided this. The summary paper will properly resemble a research paper.

System Architecture



When designing the system requirements, it is essential for each code portion to be localized within individual files. Having high-level architecture become a requirement requires legible code, plus it will pay off when we attempt to debug our initial implementation. The system architecture divides into three arbitrary segments: setup, agent, and strategy.

The setup consists of the Board and GUI. The main driver will pass the filename for each sudoku template into the Board class which will be used to construct a 9x9 integer array for the Board object. The Board object will also hold a list of possible states the board can be in. The GUI class eases code from the driver by creating a simple solverSettings() method that has inputs to choose which sudoku template the user would like to solve, which strategic method the agent should use to solve it, and which

agent the user would like to use. We will only need one agent for this project. This contributes to the requirement that we make a high-level architecture design.

The agent subdivision only consists of the agent class, and it holds solver and n_decisions fields. Solver will hold which strategy the agent will use to solve the puzzle. n_decisions will get passed on from the strategy algorithms. Solver n_decisions will count the number of decisions the agent makes when solving each puzzle, which will be used as metric to measure the efficiency of each of the five strategies we will implement. setStrategy(str) takes a string that will choose which solving method to implement. The options will be BT, BTFC, BTAC, SA, GA for backtracking, backtracking forward checking, backtracking arc-consistency, simulated annealing, and genetic algorithm, respectively. solve(board) will initiate the agent to solve the puzzle. getSteps() is a getter function that receives the number of decisions made from the chosen strategy.

Finally, the strategy subdivision is where the AI algorithms live. By using an interface, we can guarantee that every algorithmic class will have the method solveStrategy(board) that returns a board (that is hopefully completed) and countSteps(), which is the function that tracks the number of epochs the algorithm performs. This also contributes to the requirement of high-level, flexible architecture.

The first solving method we will dissect is Simple_Backtrack, as it extends to the forward checking and arc-consistency algorithms. Simple backtracking is a recursive DFS tree that is the least optimal solver out of the five we are testing. The Simple_Backtrack class takes the puzzle board and the empty cell as fields. There are two backtrack methods, one with a board and a list parameter, and another with a board and an assignment parameter. The first backtrack method's only purpose is to begin the recursive sequence of the second backtrack method, as the first method is public, and the second method is private. The next method select-unsigned-variable(board, assignment) whose purpose is to pick an available number in the domain to fill the empty cell with. Since simple backtrack leaves inconsistent inputs in the domain, we will implement static ordering to choose the unsigned variable instead of choosing a random number. Next is the inference() function and its job is to provide further insight to the simple backtracking. This will be important when we get to the arc-consistency class. Finally, there is the order-domain-values(board, int, assignment) which is how we get the possible domain values for a given cell.

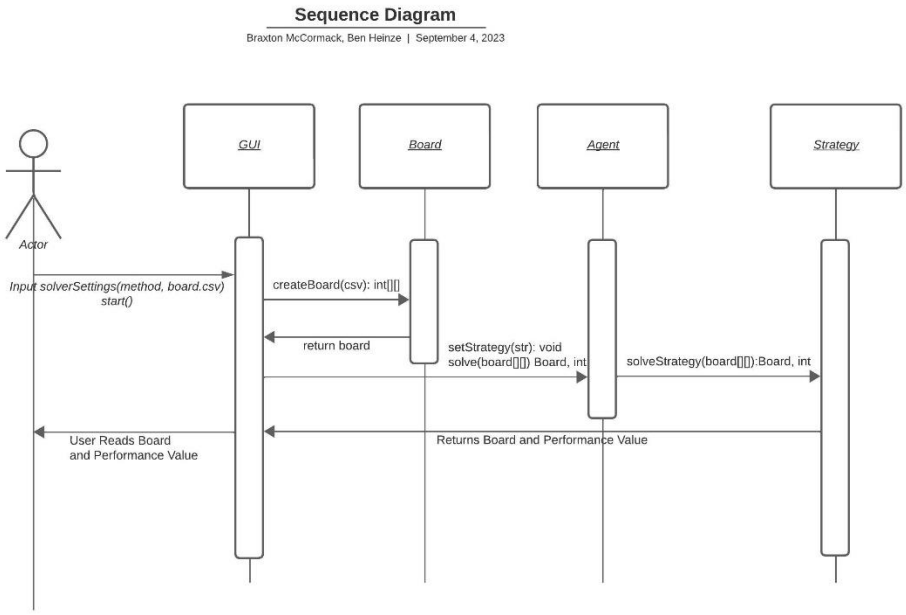
Backtracking_Forward_Check (BTFC) and Backtracking_Arc_Consistency (BTAC) both extend from the Simple_Backtracking. Unlike simple backtracking, BTFC will remove values from the domain that are inconsistent with the given restraints. In other words, if a cell in a row already holds the value of two, two will no longer exist in the domain for that row. The removeDomainToken() function will remove the inconsistent values. On the other hand, BTAC will check if two binary variables are consistent with one another and remove values from the domain for any inconsistencies. The AC-3(board) function kicks off the algorithm and revise(board, x_i, x_j) will take the board, and two variables, and will create new constraints after it revises the board. The difference between BTAC and BTFC is that BTAC will propagate new constraints as a method to maintain consistency over time when future changes are made.

Onto the two independent algorithms, Local_Search_Simulated_Annealing_Min_Conflict_Heuristic (SA) and Local_Search_Genetic_Alg_Penalty_Tournament_Selection. The goal of SA is to combine low-cost helpful choices with random choices. In the context of sudoku, each number will have a certain probability for each cell. The higher the temperature, the more likely a number with a lower probability will be chosen. As time passes, the temperature lowers and numbers with higher probability will be chosen more. The probability exponentially lowers with how "bad" the choice was. The SA class has four fields. Current will hold the current integer choice for cell, next is the next possible option for that number in the given cell, temperature, which is how long the algorithm will take to "cool," and change_E, which

is the amount the probability decreases based on the "badness" of the decision. Simulated_annealing() starts the simulation and schedule() determines the value of the temperature as a function of time.

Lastly, we have the genetic algorithm, attempts to assign every empty cell at once, then attempts to converge on a solution through crossbreeding and mutations. The fields consist of the population (every empty cell on the board), the weights (survival capability, aka how high their value is), the parents, which is where we will splice the board to create two segments, and child, which is a spliced combination of the two parents. We may choose to increase the number of parents. The functions genetic_algorithm(population, fitness) kicks off the four step-process. Weighted-random-choices() is how the GA ranks each choice for fitness which is how it will choose the parents. reproduce() will activate the combination of the two parents to create a child with the best fitness values and mutate() will randomly change some arbitrarily chosen numbers on the board. The idea of GA came from the theory of how creature's DNA morphs over time. We will need to incorporate a penalty/tournament condition into our GA in order to meet system requirements. We plan to do this by rewarding useful generations and setting a penalty threshold that will kill worse generations. This meets the requirement of incorporating a genetic algorithm.

System Flow



In the above UML sequence diagram:

1. The Actor initiates the interaction by inputting the file name of a puzzle into the GUI. The Actor then select a solving method through the GUI.
2. The GUI sends a request to the Board to create a game board with the provided puzzle file. The Board generates a game board and returns it to the GUI.
3. The GUI passes the generated game board to the Agent. The Agent receives the board alongside the selected solving method from the GUI.
4. The Agent initiates communication with the Strategy component, signaling it to start solving the puzzle. The Strategy component performs the provided problem-solving method from the Agent and tracks the performance measure (value of decisions made in the algorithmic strategies).

5. After solving the game board, the Strategy component returns the solved game board to the GUI alongside the performance measure.
6. The GUI returns the solved board and performance measure back to the Actor. The Actor can then evaluate performance value to determine how efficient the selected solving method was.

Test Strategy

Functional testing

Our testing strategy will focus on functional testing to ensure each project requirement is met. We will perform the following tests:

1. Import and solve easy Sudoku puzzles using simple backtracking.
2. Import and solve medium Sudoku puzzles using backtracking with forward checking.
3. Import and solve hard Sudoku puzzles using backtracking with arc consistency.
4. Perform local search using simulated annealing on medium Sudoku puzzles.
5. Perform local search using a genetic algorithm on hard Sudoku puzzles.
6. Measure and compare the performance of all algorithms in terms of decision counts.

Reporting and Documentation

We will document the test case, expected results, and actual outcomes in detail. Any deviations from expected results will be analyzed, and debugging will be performed when necessary. The results of these tests will be included in the final project report. We plan to use ANOVA for statistical analysis. It is important to note that not every solving-strategy will guarantee a solved puzzle, so we will need to work out testing strategies that consider how the strategies work instead of whether they guarantee the solution.

Task Assignments and Schedule

Format is as follows: Task (initials, internal due date)

Workspace Setup (bm, 9/8)

Implementation of Sudoku Puzzle Class (board class) and GUI/Driver class. (bh, 9/8)

Implementation of Simple Backtracking Algorithm and Backtracking with forward Checking Algorithm. (bm, 9/13)

Implementation of Simulated Annealing Algorithm, Arc Consistency Algorithm and Genetic Algorithm. (bh, 9/13)

Experimentation (bm bh, 9/14)

Video Creation (bh, 9/18)

Paper Authoring (bh bm, 9/18)

References

Levine, John. Constraint Satisfaction: The AC-3 Algorithm. YouTube, 14 Dec. 2019, https://youtu.be/4cCS8rrYT14?si=MHB8XkCbzfEuxBrw. Accessed 1 Sept. 2023.

Russell, Stuart J., et al. "Chapter 4 Search in Complex Environments, Chapter 6 Constraint Satisfaction Problems." Artificial Intelligence: A Modern Approach, Fourth Edition, Fourth ed., Pearson Education, Harlow, 2022.