1 Design

For the overall design of for the solver experiment code can be split into, the solving algorithms, and the sudoku puzzles representation. There will be two different versions of the solver, which have common features, can use object oriented principle of inheritance which will also help at the later on stages of the experiment code, using polymorphism. The collection of puzzles(population) will also be an attribute in the solver class.

1.1 Requirements

Load sudoku puzzles from a file For the Hybrid repair algorithm:

- Solve a 4x4 puzzle
- Solve a 9x9 puzzle

For the Multi-objective algorithm:

- Solve 4x4 puzzle
- Solve 9x9 puzzle

1.2 Representation of Sudoku Puzzle

Representing the puzzle within the program was a key issues within its creation. The representation used within this program is rather simple, for any space that has a integer value, it uses that integer, e.g. for a 9x9 puzzle it could be any number between 1 and 9. However the issues of blank spaces when trying to solve the program still remains, to deal with this I have extended the range to 0-9, where 0 represents a blank space and 1-9 represents 1-9. The integers for the puzzle are stored in a 2D array, looking very similar to an actual sudoku puzzle, as seen in figure 2. Due to the simplicity of the representation, it also comes with some notable issues. Namely the size of the search space that this creates for the algorithm, especially as the size of the puzzle increases.

Figure 1: Representation for encoding of sudoku puzzle

1.3 Puzzle class

The puzzle class, needs to store the state of the puzzle being solved and the fitness value of the puzzle. The state of the puzzle as previously detailed will be stored in a 2D array of int, the fitness value(s) should be stored in an array of variable size, this allows the puzzle to be set for either type of solver allowing for either single or multi-objective fitness functions. There are two different constructor methods for the puzzle class, a normal constructor and a constructor for copying puzzles. The normal constructor needs the size of the puzzle, the initial puzzle state and number of objectives the fitness function, used to define of the size of the fitness array.

The most important methods outside of the setters and getters, are the methods used to find rule violations and empty spaces. The violation finder method, is important for adding rules, where it is used in the repair method as a way to apply the rules of the game to solver and also used as a part of the fitness function for multi-objective. The method is split into several smaller steps, finding each violation for the 3 different rule violations, counting the number of constraint violations each space in the puzzle has and sorting the spaces with violations by number of violations.

Finding the constraint violations for the row and column, loops through each space in the row or column and comparing the value to each other space on the board. Counting the grid violations is a similar process, however it has to go through the individual grid, which uses a messier loop. The issues with this way of counting violations, is that it counts each violation twice. The empty spaces algorithm, should return an array of all locations within the puzzle space, that have a 0 value, which is useful for the fitness functions and seeing if the puzzle is complete.

1.4 Solver Class

The Solver class is the parent class for the two different types of EA being used in the experiment, so stores their common features, the attributes in the class are, the population which is an array of puzzle objects, and the initial state of the puzzle which is an array that holds the initial state of the puzzle. The important methods that are used by both solvers, are generating the population, mutation of the individual puzzles, checks for whether the puzzle is complete, a space is part of the initial puzzle and finally a method for loading the initial puzzle from a file.

Generating the intial puzzle, is a simple algorithm, that takes a puzzle, creates the initial state from that puzzle, find the fitness value for the puzzle, then adds mutations of the initial puzzle with updated fitness values, to the population, until the defined size of the population is reached.

Mutating a puzzle, displays a different behaviour based on the state of the puzzle. If the puzzle still has empty spaces, it adds a random number between 1 and the puzzle size, to a random empty space in the puzzle. The alternative behaviour is for where all the spaces in the puzzle have been filled, but the puzzle is not yet complete, which only happens to the multi-objective solver, as the repair method for the hybrid solver will always remove values that are violation before mutation happens again. For the alterative behaviour, the mutation changes a random space, that is not in the initial configuration, to a random value between 1 and the puzzle size.

The method that checks puzzles completion, looks at every puzzle in the population, if for that puzzle, there are no constraint violations and no empty spaces, meaning the puzzle is full and none of the rules of sudoku have been violated. If a puzzle that meets these conditions is found, the method prints the puzzles state and returns true, if no completed puzzle is found the method returns false.

Loading a puzzle from a file is needed in the experiment to load different puzzles from the

1.5 Hybrid method

selection details mutation details fitness function details repair method details sort details

1.6 Multi-objective method

selection details mutation details fitness details comparison function

Figure 2: Class diagram for the program

1.7 Testing program

Filewriter current puzzle loader store generations store times averages calculation