Name: Ahmad Farhan

Roll no.i211366

Section: A

Assignment 2

Question 1: *Heuristic Backtracking Solution to Sudoku Puzzle*

Algorithm Parameters

1. Initial Board (n): Randomly Generated Sudoku Board.
2. Variable Selection Heuristic: MRV, Degree or a Combination of both
3. Value Selection Heuristic: Default Ordering or LCS ordering
4. Number of Empty Cells: Number of empty cells in randomly generated board

Algorithm Heuristics

Minimum Remaining Value(MRV) Heuristic:

MRV value of a cell is the size of its remaining domain.
Pick Variable with minimum MRV Value

Degree Heuristic:

Degree value is the number of empty peers that a cell has.
Pick Variable with minimum Degree Value

Least Constraining Score(LCS) Heuristic:

LCS of a value is the number of peer cell domains it belongs to.
Apply Values in increasing order of LCS Value

Arc Consistency (AC3):

The domain of all consistently applicable values for each cell is updated throughout the exploration process which ensures arc consistency is maintained implicitly. The domains are updated based on a hash which defines all values used and usable and within any row, column or subgrid.

Algorithm Steps

Board Generation

1. Generate a 9x9 empty grid.
2. Sequentially select each empty cell.
3. Randomize the order of values being applied to each cell.
4. Use *Constraint Satisfaction* and simple *Backtracking* to fill entire board.
5. Randomly remove values from *n_zero* number of cells.

Board Solution

1. Calculate *hash* of each row, column and block using input board.
2. Calculate *domains* of all cells and number of filled cells in board.
3. Optimize solution by recursively assigning values to all cells with *domain size one*.
4. Use *Variable Selection Heuristic* to select an empty cell.
5. Apply *Value Selection Heuristic* to get the order of domain values of cell.
6. Apply the next value to cell, update hash and domains of all peer cells accordingly.
7. Use Backtracking to explore arrangements of board given the value assigned to current cell.
8. If exploration leads to an empty cell having empty domain, then backtrack.
9. For backtracking, remove value from cell and update domains and hash of peer cells.
10. Repeat 4-9 to recursively fill entire board using *Constraint Satisfaction* and *Backtracking*.

Algorithm Results

Parameters:

Value Selection Heuristic: LCS
Number of Empty Cells: 40

```
Initial Board:
[4, 5, 0, 0, 0, 6, 0, 8, 1]
[1, 0, 0, 4, 5, 0, 2, 0, 6]
[7, 0, 6, 0, 1, 0, 0, 3, 5]
[0, 0, 0, 0, 0, 0, 0, 2, 9]
[2, 0, 0, 6, 0, 0, 0, 0, 8]
[0, 0, 0, 3, 2, 0, 6, 0, 7]
[3, 7, 1, 0, 8, 0, 0, 0, 4]
[8, 0, 0, 0, 0, 0, 0, 7, 3]
[5, 6, 4, 7, 9, 3, 8, 1, 2]
```

Output:

MRV Heuristic

```
Solution Board:
[4, 5, 2, 9, 3, 6, 7, 8, 1]
[1, 3, 8, 4, 5, 7, 2, 9, 6]
[7, 9, 6, 8, 1, 2, 4, 3, 5]
[6, 1, 7, 5, 4, 8, 3, 2, 9]
[2, 4, 3, 6, 7, 9, 1, 5, 8]
[9, 8, 5, 3, 2, 1, 6, 4, 7]
[3, 7, 1, 2, 8, 5, 9, 6, 4]
[8, 2, 9, 1, 6, 4, 5, 7, 3]
[5, 6, 4, 7, 9, 3, 8, 1, 2]

Verifying Consistency : True
Execution Time: 2.97 ms
```

MRV with Degree Tiebreaker Heuristic

```
Solution Board:
[4, 5, 2, 9, 3, 6, 7, 8, 1]
[1, 3, 8, 4, 5, 7, 2, 9, 6]
[7, 9, 6, 2, 1, 8, 4, 3, 5]
[6, 1, 7, 8, 4, 5, 3, 2, 9]
[2, 4, 3, 6, 7, 9, 1, 5, 8]
[9, 8, 5, 3, 2, 1, 6, 4, 7]
[3, 7, 1, 5, 8, 2, 9, 6, 4]
[8, 2, 9, 1, 6, 4, 5, 7, 3]
[5, 6, 4, 7, 9, 3, 8, 1, 2]
Verifying Consistency : True
Execution Time: 3.77 ms
```

Degree Heuristic

```
Solution Board:
[4, 5, 2, 9, 3, 6, 7, 8, 1]
[1, 3, 8, 4, 5, 7, 2, 9, 6]
[7, 9, 6, 8, 1, 2, 4, 3, 5]
[6, 8, 7, 5, 4, 1, 3, 2, 9]
[2, 4, 3, 6, 7, 9, 1, 5, 8]
[9, 1, 5, 3, 2, 8, 6, 4, 7]
[3, 7, 1, 2, 8, 5, 9, 6, 4]
[8, 2, 9, 1, 6, 4, 5, 7, 3]
[5, 6, 4, 7, 9, 3, 8, 1, 2]

Verifying Consistency : True
Execution Time: 7.04 ms
```

Question 2: *Genetic Algorithm Solution to 3x3 Magic Square Puzzle*

Algorithm Parameters

1. Population size (n): Number of individuals in population.
2. Elite size: Number of individuals with highest fitness scores to be transferred to the next generation automatically.
3. Mutation Probability: Chances of Mutation occurrence within individual
4. Crossover Functions: Both Are single point crossover functions.
    a. Partially Mapped Crossover
    b. Inversion Sequence Crossover (https://user.ceng.metu.edu.tr/~ucoluk/research/publications/tspnew.pdf))

Algorithm Definitions

Individual Representation:
Each individual is represented as a simple array with the square in row major order i.e., [1,2,3,4,5,6,7,8,9]

Fitness Function:
1. *Magic Number*(M) is calculated using the formula (n(n^2 + 1)/2)
2. *Sum*(S) of each row, column and diagonals is calculated.
3. *Absolute Difference*(D) of each S is taken with M i.e. |S-M|
4. *Sum* of all D of a square gives fitness of individual

Mating Pool Selection:
1. Individuals are selected using *tournament selection*.
2. Mating pool size is set to half of population size.
3. Two individuals with different fitness values are randomly selected.
4. The individual with better fitness is added to the pool.
5. This process is repeated till the required pool size.

Elitism Selection:
A total of *elite_size* individuals with highest fitness value are selected and passed onto the next generation automatically.

Crossover Function:
Both Crossover options are based on single point crossover and retain consistency of magic square problem i.e., do not allow duplicate genes within an individual. Cross over point k is picked randomly.

1. Partially Mapper Crossover:
   k values of the two parents, suppose s and t, are swapped. t_i is substituted into s by swapping the t_i value and the s_i value of s. e.g. Let s = [5,7,1,3,6,4,2], t = [4,6,2,7,3,1,5], the first swap would leave s as [4,7,1,3,6,5,2]

2. Inversion Sequence Crossover:
   This crossover method was proposed in this research paper
   For each parent an inversion sequence array is calculated, which is a reversible representation of the same individual. Being reversible means that we can recalculate the original permutation from the inversion sequence. The inversion sequences are swapped up to k values. Then permutations are recalculated to complete crossover. This results in a crossover without generation of duplicate genes.

Mutation Function:
When probability condition is satisfied, two randomly selected genes of an individual are swapped within the same individual.

Algorithm Steps

1. Start with a randomly generated initial population.
2. Calculate Fitness of each individual in population.
3. Sort Population based on fitness values.
4. Select k individuals for mating pool.
5. Pass *elite_size* individuals to next generation as elites.
6. Randomly select two individuals with different fitness values.
7. Perform crossover and mutation to generate offspring.
8. This process is repeated till the target found or max iterations hit.

Algorithm Results

Parameters:

Crossover Function: Inverse Sequence Permutation
Mutation Probability: 0.5
Population Size: 9
Elite Size: 1

Output:

Test 1:
Generation: 0

```
([4, 2, 9, 6, 1, 3, 5, 7, 8]), 22)
([2, 8, 6, 7, 4, 3, 9, 5, 1]), 24)
([3, 5, 6, 9, 7, 4, 2, 8, 1]), 24)
([8, 4, 6, 1, 9, 3, 2, 5, 7]), 25)
([4, 9, 6, 1, 8, 5, 2, 7, 3]), 27)
([1, 2, 3, 4, 6, 7, 9, 5, 8]), 27)
([1, 5, 8, 7, 4, 9, 2, 6, 3]), 28)
([8, 6, 9, 4, 2, 7, 1, 5, 3]), 29)
([4, 5, 3, 8, 7, 9, 1, 2, 6]), 30)

Result Generation : [9]
Result : (2, 7, 6, 9, 5, 1, 4, 3, 8)
```

Test 2:
Generation: 0

```
([7, 3, 9, 6, 5, 2, 1, 8, 4], 11)
([7, 3, 4, 8, 6, 2, 5, 9, 1], 19)
([4, 1, 9, 5, 3, 2, 6, 8, 7], 22)
([7, 5, 2, 1, 8, 6, 3, 9, 4], 22)
([8, 1, 3, 2, 6, 5, 4, 7, 9], 24)
([4, 6, 3, 7, 8, 1, 9, 5, 2], 28)
([5, 7, 1, 9, 2, 8, 6, 4, 3], 29)
```

```
([3, 4, 1, 7, 9, 5, 6, 8, 2], 30)
([5, 6, 7, 2, 9, 8, 1, 4, 3], 32)
```

Generation: 30

```
([6, 2, 9, 7, 5, 3, 1, 8, 4], 6)
([6, 2, 9, 7, 5, 3, 1, 8, 4], 6)
([6, 2, 7, 9, 5, 3, 1, 8, 4], 8)
([6, 2, 7, 9, 5, 3, 1, 8, 4], 8)
([5, 6, 8, 7, 2, 3, 1, 9, 4], 20)
([5, 6, 9, 7, 2, 3, 1, 8, 4], 21)
([6, 3, 7, 9, 8, 5, 1, 2, 4], 24)
([6, 2, 7, 9, 5, 8, 1, 3, 4], 26)
([6, 5, 2, 9, 8, 3, 1, 7, 4], 29)

Result Generation : [37]
Result : (6, 1, 8, 7, 5, 3, 2, 9, 4)
```

Parameters:

Crossover Function: Partially Mapped
Mutation Probability: 0.5
Population Size: 9
Elite Size: 1

Output:

Test 1:
Generation: 0

```
(array([2, 9, 3, 4, 8, 5, 6, 1, 7]), 14)
(array([9, 5, 2, 1, 8, 7, 3, 4, 6]), 18)
(array([6, 3, 5, 9, 8, 2, 1, 7, 4]), 20)
(array([3, 7, 2, 1, 8, 9, 4, 6, 5]), 22)
(array([7, 6, 4, 3, 9, 1, 8, 2, 5]), 26)
(array([7, 4, 2, 1, 8, 3, 9, 5, 6]), 28)
(array([5, 7, 9, 8, 3, 4, 6, 2, 1]), 29)
(array([6, 1, 3, 8, 7, 9, 5, 2, 4]), 30)
(array([1, 8, 4, 3, 5, 2, 7, 9, 6]), 32)
```

Generation: 30

```
(array([2, 5, 8, 9, 3, 1, 4, 7, 6]), 8)
(array([3, 5, 9, 8, 2, 1, 4, 7, 6]), 14)
(array([3, 8, 5, 9, 2, 4, 7, 1, 6]), 15)
(array([5, 9, 3, 2, 4, 1, 8, 7, 6]), 26)
(array([9, 8, 5, 2, 3, 7, 4, 1, 6]), 26)
(array([9, 5, 3, 8, 2, 1, 4, 7, 6]), 28)
(array([3, 8, 5, 2, 4, 1, 9, 7, 6]), 29)
(array([5, 9, 3, 2, 7, 1, 4, 8, 6]), 32)
(array([5, 2, 9, 4, 8, 7, 3, 1, 6]), 33)
```

Generation: 60

```
(array([3, 4, 8, 9, 5, 1, 2, 7, 6]), 3)
(array([3, 4, 8, 7, 5, 2, 1, 9, 6]), 12)
(array([5, 3, 8, 9, 4, 2, 7, 1, 6]), 20)
(array([3, 4, 5, 1, 9, 7, 8, 2, 6]), 22)
(array([7, 4, 5, 1, 9, 3, 2, 8, 6]), 24)
(array([9, 4, 8, 1, 3, 5, 2, 7, 6]), 25)
(array([2, 5, 1, 7, 4, 8, 3, 9, 6]), 30)
(array([2, 5, 1, 7, 4, 3, 9, 8, 6]), 30)
(array([2, 5, 1, 7, 4, 3, 8, 9, 6]), 31)

Result Generation : [78]
Result : (4, 3, 8, 9, 5, 1, 2, 7, 6)
```

Test 2:

Generation: 0

```
(array([9, 3, 6, 8, 4, 1, 2, 7, 5]), 20)
(array([9, 2, 5, 3, 8, 6, 7, 4, 1]), 22)
(array([1, 3, 9, 4, 5, 8, 2, 6, 7]), 25)
(array([8, 2, 5, 6, 4, 7, 9, 3, 1]), 25)
(array([5, 7, 2, 3, 6, 1, 9, 4, 8]), 26)
(array([6, 8, 4, 9, 1, 3, 7, 2, 5]), 26)
(array([3, 2, 4, 8, 6, 1, 9, 5, 7]), 27)
(array([4, 9, 1, 6, 8, 5, 3, 7, 2]), 30)
(array([7, 2, 3, 8, 5, 9, 6, 1, 4]), 30)
```

Generation: 30

```
(array([6, 9, 3, 2, 5, 8, 7, 1, 4]), 6)
(array([6, 1, 3, 2, 5, 8, 7, 9, 4]), 10)
(array([1, 6, 3, 2, 9, 8, 7, 5, 4]), 25)
...
(array([7, 6, 5, 2, 1, 9, 3, 8, 4]), 21)
(array([9, 5, 3, 2, 8, 7, 1, 6, 4]), 25)
(array([9, 5, 3, 2, 8, 7, 1, 6, 4]), 25)
(array([3, 5, 9, 2, 8, 7, 1, 6, 4]), 29)
(array([6, 9, 3, 2, 8, 7, 1, 5, 4]), 30)
(array([7, 6, 5, 2, 3, 1, 9, 8, 4]), 31)
```

Generation: 840

```
(array([6, 8, 2, 1, 5, 9, 7, 3, 4]), 5)
(array([5, 8, 2, 1, 6, 9, 7, 3, 4]), 6)
(array([6, 5, 2, 1, 8, 9, 7, 3, 4]), 13)
(array([6, 8, 2, 1, 5, 9, 3, 7, 4]), 17)
(array([8, 9, 1, 2, 3, 6, 7, 5, 4]), 20)
(array([6, 3, 8, 2, 5, 9, 7, 1, 4]), 23)
(array([7, 9, 1, 2, 3, 6, 5, 8, 4]), 25)
(array([3, 9, 1, 2, 5, 6, 7, 8, 4]), 27)
(array([2, 5, 6, 8, 3, 9, 7, 1, 4]), 29)

Result Generation : [851]
Result : (6, 7, 2, 1, 5, 9, 8, 3, 4)
```