Solving Sudoku Problem Using Genetic Algorithm

Group Members:
Zhenhao Li
Runjie Li
Qianjin Wu

Catalogue

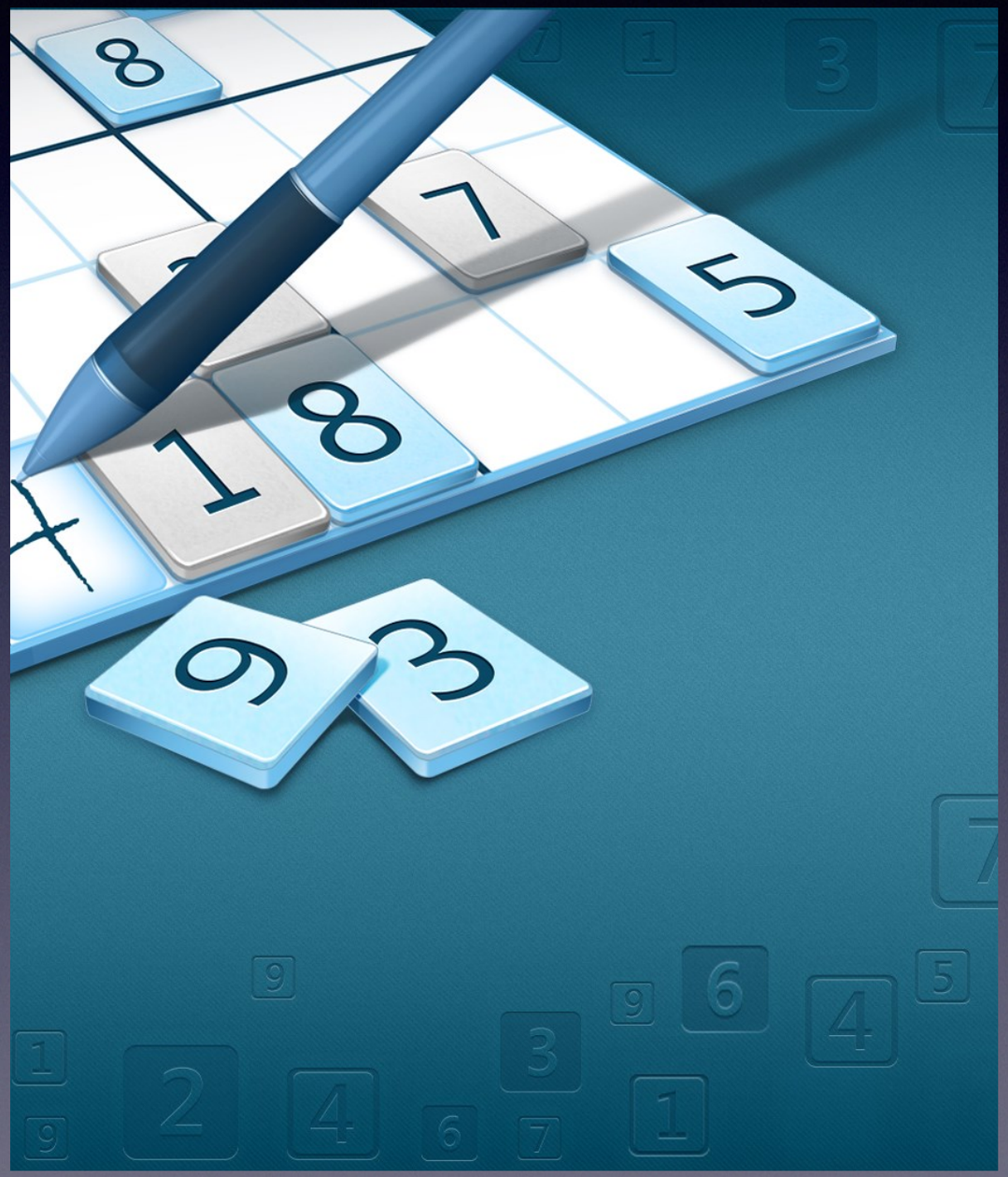
- Problem Description
- Code Description
- Results
- Conclusion

Problems Description

- Our objective is to find the perfect solution for Sudoku puzzles with Genetic Algorithm.

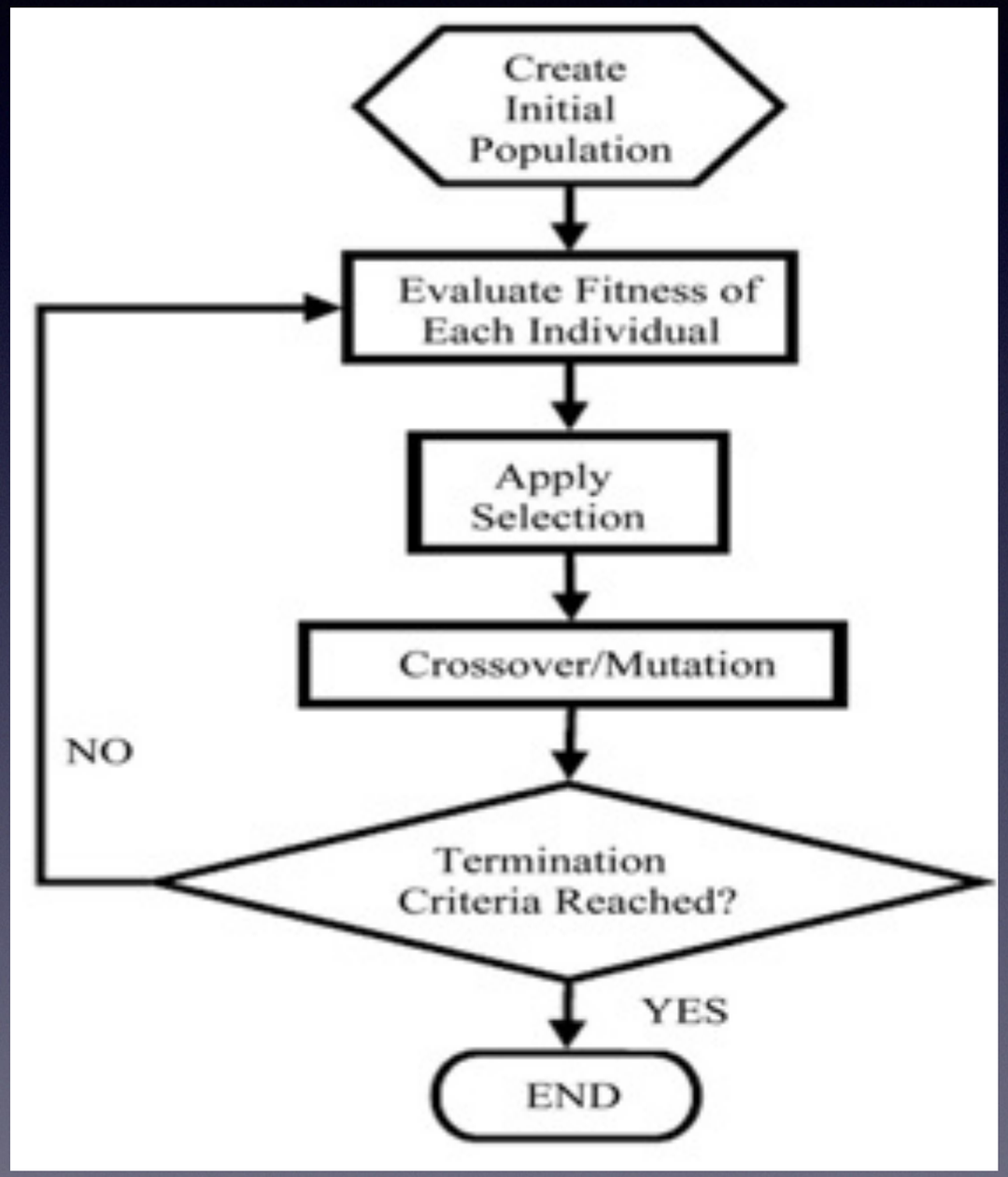
- The initial puzzle is set in main().

- Sudoku puzzle is to make each number only appear once in a row, column, block.



The Flow in GA

- Initial population: 1000
- Max iteration: 500000
- Phenotype mutation rate: 0.1



Expressions of Gene

- In this project, each individual has chromosome(gene) which is a int[] type of number from 0 to 9
- We also have phenotype gene which will be presented but not inherited.

```java
import java.util.Random;

public class Gene {
    public int length;
    public int[] gene;


    public int[] phenotype;
    public double score;

    public int[] getPhenotype() {
        return phenotype;
    }

    public void setPhenotype(int[] phenotype) {
        this.phenotype = phenotype;
    }

    public int getlength() {
        return length;
    }
    public void setlength(int length) {
        this.length=length;
    }
    public int[] getgene() {
        return gene;
    }
    public void setgene(int gene[]) {
        this.gene=gene;
    }
    public double getscore() {
        return score;
    }
    public void setscore(double score) {
        this.score=score;
    }
    public Gene(int length){
        this.length = length;
        gene = new int[length];
        phenotype = new int[length];
        Random rand = new Random();
        for(int i=0;i<length;i++){
            gene[i] = rand.nextInt( bound: 2);
        }
        for(int i=0; i<gene.length; i++) {
            phenotype[i] = gene[i];
        }
        this.score=0;
    }
}
```

Fitness

- The first for loop is to add 1 in fitness when duplicate found in each row or column

- The second for loop is to add 1 in fitness when duplicate found in each block

```java
public int fitness() {
    double prob = Math.random();

    if (prob<0.1) {
        this.mutatePheno();
    }
    return fitness(this.phenotype);
}

public static int fitness(int[] gene) {
    int ft = 0;

    int[][] newG = oneTotwo(gene);
    int[][] newInitialGene = oneTotwo(initialGene);
    for (int i = 0; i < newG.length; i++) {
        boolean[] rowFlag = new boolean[newG.length + 1];
        boolean[] colFlag = new boolean[newG.length + 1];
        for (int j = 0; j < newG.length; j++) {
            if (rowFlag[newG[i][j]])
                ft++;
            if (colFlag[newG[j][i]])
                ft++;
            if ((newInitialGene[i][j] != 0 && newInitialGene[i][j] != newG[i][j]) || newG[i][j] == 0)
                ft += 10;

            rowFlag[newG[i][j]] = true;
            colFlag[newG[j][i]] = true;
        }
    }
    int blockSize = (int) Math.sqrt(newG.length);
    for (int i = 0; i < newG.length; i += blockSize) {
        for (int j = 0; j < newG.length; j += blockSize) {
            boolean[] blockFlag = new boolean[newG.length + 1];

            for (int k = 0; k < blockSize; k++) {
                for (int l = 0; l < blockSize; l++) {
                    if (blockFlag[newG[i + k][j + l]])
                        ft++;
                    blockFlag[newG[i + k][j + l]] = true;
                }
            }
        }
    }
    return ft;
}
```

Mutation

```java
public static int[] mutation(int[] gene) {
    int dim = (int) Math.sqrt(gene.length);
    int[] muta = mutation(gene, new Random().nextInt(gene.length),  value: new Random().nextInt(dim) + 1);
        return muta;
    }

public static int[] mutation(int[] gene, int index, int value) {
    gene[index] = value;

    return gene;
}

public static int[] mutatePheno(int[] gene) {
    int dim = (int) Math.sqrt(gene.length);
    int[] muta = mutatePheno(gene, new Random().nextInt(gene.length),  value: new Random().nextInt(dim) + 1);
    return muta;
}

public static int[] mutatePheno(int[] gene, int index, int value) {
    gene[index] = value;

    return gene;
}
```

- Mutation: arbitrary change one gene in chromosome
- MutatePheno: arbitrary change one phenotype

Crossover

```java
public static int[][] crossover(int[] g1, int[] g2) {
    int start = new Random().nextInt(g1.length);
    int end = new Random().nextInt( bound: g1.length - start) + start;

    return crossover(g1, g2, start, end);
}

public static int[][] crossover(int[] g1, int[] g2, boolean sp) {
    int start = (sp) ? 0 : new Random().nextInt(g1.length);
    int end = new Random().nextInt( bound: g1.length - start) + start;

    return crossover(g1, g2, start, end);
}

public static int[][] crossover(int[] g1, int[] g2, int start, int end) {
    int[][] newG = new int[2][g1.length];
    for (int i = 0; i < g1.length; i++) {
        newG[0][i] = (i >= start && i <= end) ? g2[i] : g1[i];
        newG[1][i] = (i >= start && i <= end) ? g1[i] : g2[i];
    }
    return newG;
}
```

- Input 2 genes from parents
- Arbitrary start & end point for chromosome crossover

Selection Algorithm

- BestSelection Algorithm

```java
public static Sudoku bestSelection(Vector<Sudoku> sudokus) {
    Sudoku min = sudokus.firstElement();
    for (Sudoku sudoku : sudokus)
        if (sudoku.fitnessValue < min.fitnessValue)
            min = sudoku;

    return min;
}
```

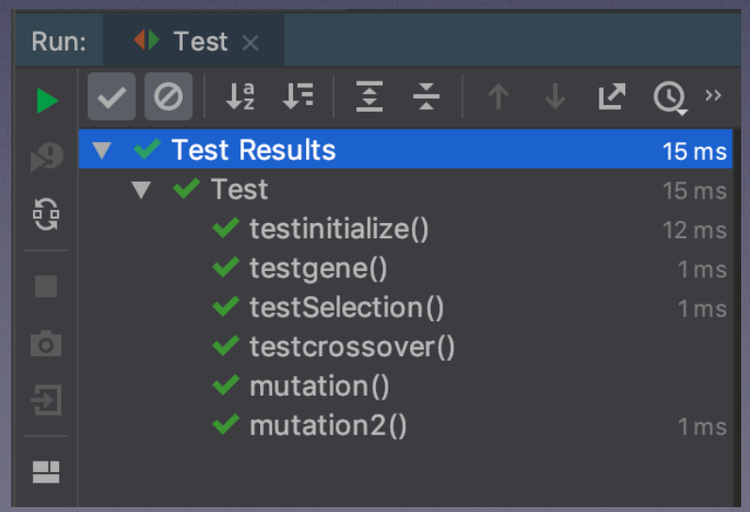
- RouletteSelection Algorithm

```java
public static Sudoku rouletteSelection(Vector<Sudoku> sudokus) {
    int max = 0;
    for (Sudoku sudoku : sudokus)
        if (sudoku.fitnessValue > max) max = sudoku.fitnessValue;


    int sum = 0;
    for (Sudoku sudoku : sudokus)
        sum += max - sudoku.fitnessValue;
    for (Sudoku sudoku : sudokus)
        sudoku.setProbability((max - sudoku.fitnessValue) / (sum * 1.0));

    double random = Math.random() * sum;
    int i;
    for (i = 0; i < sudokus.size() && random > 0; i++) {
        random -= max - sudokus.get(i).fitnessValue;
    }
    return sudokus.get(i - 1);
}
```

Unit Test

- TestInitialize: whether gene initialized correctly
- TestGene: whether GeneticOperator works properly
- TestSelection: whether selection function works
- TestCrossover: whether crossover function works
- Mutation: whether mutation function works
- Mutation2: whether mutatePheno function works

```java
@org.junit.jupiter.api.Test
void testcrossover() {
    GeneticOperators ge = new GeneticOperators();

    int [] curgene = ge.initialize(testarray);
    int[][] crossgene = GeneticOperators.crossover(originalarray,curgene, start: 3, end: 5);
    for(int i=3; i<5; i++){
        assertEquals(crossgene[0][i],curgene[i]);
        assertEquals(crossgene[1][i],originalarray[i]);
    }

}

@org.junit.jupiter.api.Test
void testSelection() {
    GeneticOperators ge = new GeneticOperators();
    Vector<Sudoku> sudokus = new Vector<>();
    Sudoku.setInitialGene(testarray);
    int[] curgene = ge.initialize(testarray);
    Sudoku currenS = new Sudoku(curgene);
//    Sudoku currentSudoku = new Sudoku(ge.initialize(Sudoku.getInitialGene().clone()));
    sudokus.add(new Sudoku(currenS.getGene()));
    int maxfv = Sudoku.fitness(curgene);
    assertEquals(currenS.bestSelection(sudokus).getFitnessValue(),maxfv);
}

@org.junit.jupiter.api.Test
void mutation() {

    GeneticOperators ge = new GeneticOperators();
    int [] curgene = ge.initialize(testarray);
    for(int i=0; i<curgene.length; i++) {
        System.out.println(curgene[i]);
    }

    Sudoku.setInitialGene(testarray);
    Sudoku testmt =new Sudoku(curgene);
    testmt.mutation( index: 2, value: 3);
    Assertions.assertNotEquals(testarray[2], actual: 1);

}
```



Results

- Output: Current Gene + Phenotype + Fitness + iteration
- Goal: To solve Sudoku (when fitness = 0)

```
Current Gene: 424768193893415627761392584915837246638249715247156938352974861179683452486521379
4 2 4 7 6 8 1 9 3
8 9 3 4 1 5 6 2 7
7 6 1 3 9 2 5 8 4
9 1 5 8 3 7 2 4 6
6 3 8 2 4 9 7 1 5
2 4 7 1 5 6 9 3 8
3 5 2 9 7 4 8 6 1
1 7 9 6 8 3 4 5 2
4 8 6 5 2 1 3 7 9
PhenoType:424768193893415627761392584915837246638249715247156938352974861179683452486521379
4 2 4 7 6 8 1 9 3
8 9 3 4 1 5 6 2 7
7 6 1 3 9 2 5 8 4
9 1 5 8 3 7 2 4 6
6 3 8 2 4 9 7 1 5
2 4 7 1 5 6 9 3 8
3 5 2 9 7 4 8 6 1
1 7 9 6 8 3 4 5 2
4 8 6 5 2 1 3 7 9
Fitness: 3, iteration: 34474
Solution Gene: 524768193893415627761392584915837246638249715247156938352974861179683452486521379
5 2 4 7 6 8 1 9 3
8 9 3 4 1 5 6 2 7
7 6 1 3 9 2 5 8 4
9 1 5 8 3 7 2 4 6
6 3 8 2 4 9 7 1 5
2 4 7 1 5 6 9 3 8
3 5 2 9 7 4 8 6 1
1 7 9 6 8 3 4 5 2
4 8 6 5 2 1 3 7 9
PhenoType:524768193893415627761392584915837246638249715247156938352974861179683452486521379
5 2 4 7 6 8 1 9 3
8 9 3 4 1 5 6 2 7
7 6 1 3 9 2 5 8 4
9 1 5 8 3 7 2 4 6
6 3 8 2 4 9 7 1 5
2 4 7 1 5 6 9 3 8
3 5 2 9 7 4 8 6 1
1 7 9 6 8 3 4 5 2
4 8 6 5 2 1 3 7 9
Fitness: 0, iteration: 34475

Process finished with exit code 0
```

Results

- You can see fitness is decreasing…
- It means gene is evolving…

```
1 7 9 6 8 3 4 5 2
4 8 6 5 2 1 3 7 9
PhenoType:524768393893415627761396584915837246638249715247156938352674861179683452486521379
5 2 4 7 6 8 3 9 3
8 9 3 4 1 5 6 2 7
7 6 1 3 9 6 5 8 4
9 1 5 8 3 7 2 4 6
6 3 8 2 4 9 7 1 5
2 4 7 1 5 6 9 3 8
3 5 2 6 7 4 8 6 1
1 7 9 6 8 3 4 5 2
4 8 6 5 2 1 3 7 9
Fitness: 29, iteration: 34464
Current Gene: 524768393893415627761392584915837246638249715247156938352674861179683452486521379
5 2 4 7 6 8 3 9 3
8 9 3 4 1 5 6 2 7
7 6 1 3 9 2 5 8 4
9 1 5 8 3 7 2 4 6
6 3 8 2 4 9 7 1 5
2 4 7 1 5 6 9 3 8
3 5 2 6 7 4 8 6 1
1 7 9 6 8 3 4 5 2
4 8 6 5 2 1 3 7 9
PhenoType:524768393893415627761392584915837246638249715247156938352674861179683452486521379
5 2 4 7 6 8 3 9 3
8 9 3 4 1 5 6 2 7
7 6 1 3 9 2 5 8 4
9 1 5 8 3 7 2 4 6
6 3 8 2 4 9 7 1 5
2 4 7 1 5 6 9 3 8
3 5 2 6 7 4 8 6 1
1 7 9 6 8 3 4 5 2
4 8 6 5 2 1 3 7 9
Fitness: 16, iteration: 34465
Current Gene: 524768393893415627761392584915837246638249715247156938352974861179683452486521379
5 2 4 7 6 8 3 9 3
8 9 3 4 1 5 6 2 7
7 6 1 3 9 2 5 8 4
9 1 5 8 3 7 2 4 6
6 3 8 2 4 9 7 1 5
2 4 7 1 5 6 9 3 8
3 5 2 9 7 4 8 6 1
1 7 9 6 8 3 4 5 2
4 8 6 5 2 1 3 7 9
PhenoType:524768393893415627761392584915837246638249715247156938352974861179683452486521379
5 2 4 7 6 8 3 9 3
8 9 3 4 1 5 6 2 7
7 6 1 3 9 2 5 8 4
9 1 5 8 3 7 2 4 6
6 3 8 2 4 9 7 1 5
2 4 7 1 5 6 9 3 8
3 5 2 9 7 4 8 6 1
1 7 9 6 8 3 4 5 2
4 8 6 5 2 1 3 7 9
Fitness: 3, iteration: 34466
```

Conclusion

- By using genetic algorithm, we could select individual who has better fitness(the lower the better in our program) to get the final solution for our Sudoku puzzle.

- After many adjustments we find out our optimal genetic algorithm model(Initial population: 1000, Phenotype mutation rate: 0.1, Max iteration: 500000)