Department of
Computer Science and Engineering

Title: Genetic Algorithms

Artificial Intelligence Lab
CSE 404

Green University of Bangladesh

1 Objective(s)

- To understand how to work with Genetic Algorithms.
- To understand how Genetic Algorithms works.
- To understand how to use Genetic Algorithms to solve different problems.

2 Problem analysis

Genetic Algorithms(GAs) are adaptive heuristic search algorithms that belong to the larger part of evolutionary algorithms. Genetic algorithms are based on the ideas of natural selection and genetics. These are intelligent exploitation of random search provided with historical data to direct the search into the region of better performance in solution space. They are commonly used to generate high-quality solutions for optimization problems and search problems.

Genetic algorithms simulate the process of natural selection which means those species who can adapt to changes in their environment are able to survive and reproduce and go to next generation. In simple words, they simulate "survival of the fittest" among individual of consecutive generation for solving a problem. Each generation consist of a population of individuals and each individual represents a point in search space and possible solution. Each individual is represented as a string of character/integer/float/bits. This string is analogous to the Chromosome.

Foundation of Genetic Algorithms

a. Initial population

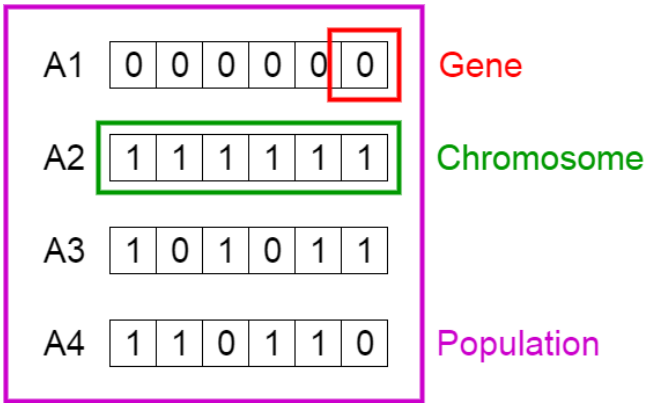
b. Fitness function

c. Selection

d. Crossover

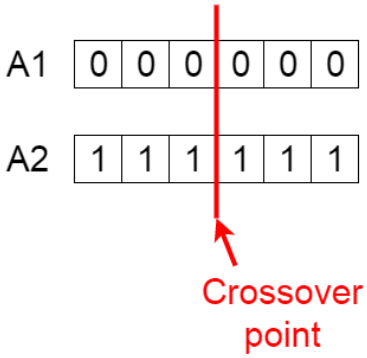
e. Mutation

Initial Population: The process begins with a set of individuals which is called a Population. Each individual is a solution to the problem you want to solve. An individual is characterized by a set of parameters (variables) known as Genes. Genes are joined into a string to form a Chromosome (solution). In a genetic algorithm, the set of genes of an individual is represented using a string, in terms of an alphabet. Usually, binary values are used (string of 1s and 0s). We say that we encode the genes in a chromosome.

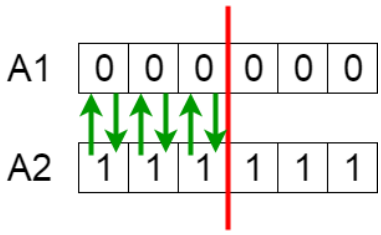


Fitness Function: The fitness function determines how fit an individual is (the ability of an individual to compete with other individuals). It gives a fitness score to each individual. The probability that an individual will be selected for reproduction is based on its fitness score.

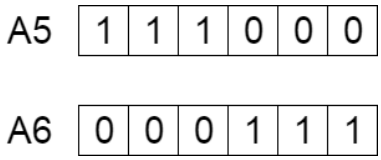
Crossover: Crossover is the most significant phase in a genetic algorithm. For each pair of parents to be mated, a crossover point is chosen at random from within the genes. For example, consider the crossover point to be 3 as shown below.



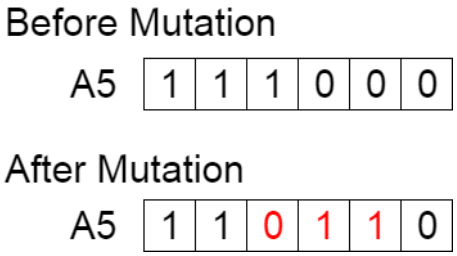
Offspring are created by exchanging the genes of parents among themselves until the crossover point is reached.



The new offspring are added to the population.



Mutation: In certain new offspring formed, some of their genes can be subjected to a mutation with a low random probability. This implies that some of the bits in the bit string can be flipped.



Mutation occurs to maintain diversity within the population and prevent premature convergence.

Termination: The algorithm terminates if the population has converged (does not produce offspring which are significantly different from the previous generation). Then it is said that the genetic algorithm has provided a set of solutions to our problem.

3 Flowchart

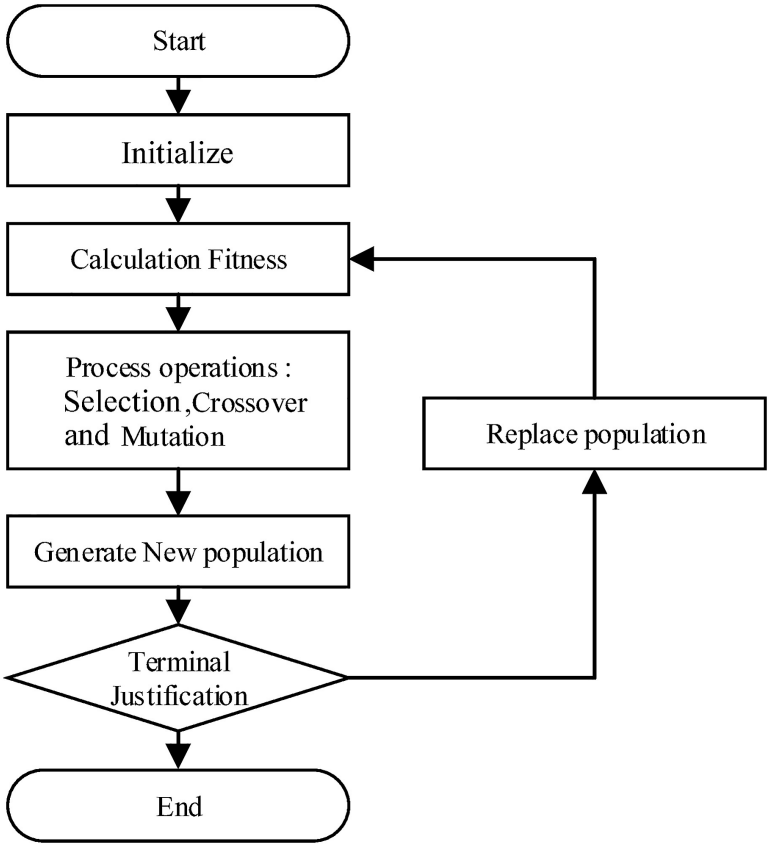


Figure 1: Flowchart

4 Algorithm

Algorithm 1: Genetic Algorithms

Output: Solution

`/* Genetic Algorithms */`

1 Step 1: Randomly initialize populations p
2 Step 2: Determine fitness of population
3 Step 3: Until convergence **repeat** Step 4 to 7:
4 Step 4: Select parents from population
5 Step 5: Crossover and generate new population
6 Step 6:Perform mutation on new population
7 Step 7: Calculate fitness for new population

5 Implementation in Java

```
1
2  package simpledemoga;
3
4  import java.util.Random;
5
6  public class SimpleDemoGA {
7
8      Population population = new Population();
9      Individual fittest;
```

```
10      Individual secondFittest;
11      int generationCount = 0;
12
13      public static void main(String[] args) {
14
15          Random rn = new Random();
16
17          SimpleDemoGA demo = new SimpleDemoGA();
18
19          //Initialize population
20          demo.population.initializePopulation(10);
21
22          //Calculate fitness of each individual
23          demo.population.calculateFitness();
24
25          System.out.println("Generation: " + demo.generationCount + " Fittest: "
                + demo.population.fittest);
26
27          //While population gets an individual with maximum fitness
28          while (demo.population.fittest < 5) {
29              ++demo.generationCount;
30
31              //Do selection
32              demo.selection();
33
34              //Do crossover
35              demo.crossover();
36
37              //Do mutation under a random probability
38              if (rn.nextInt()%7 < 5) {
39                  demo.mutation();
40              }
41
42              //Add fittest offspring to population
43              demo.addFittestOffspring();
44
45              //Calculate new fitness value
46              demo.population.calculateFitness();
47
48              System.out.println("Generation: " + demo.generationCount + " Fittest
                    : " + demo.population.fittest);
49          }
50
51          System.out.println("\nSolution found in generation " + demo.
                generationCount);
52          System.out.println("Fitness: "+demo.population.getFittest().fitness);
53          System.out.print("Genes: ");
54          for (int i = 0; i < 5; i++) {
55              System.out.print(demo.population.getFittest().genes[i]);
56          }
57
58          System.out.println("");
59
60      }
61
62      //Selection
63      void selection() {
64
```

```
65          //Select the most fittest individual
66          fittest = population.getFittest();
67
68          //Select the second most fittest individual
69          secondFittest = population.getSecondFittest();
70      }
71
72      //Crossover
73      void crossover() {
74          Random rn = new Random();
75
76          //Select a random crossover point
77          int crossOverPoint = rn.nextInt(population.individuals[0].geneLength);
78
79          //Swap values among parents
80          for (int i = 0; i < crossOverPoint; i++) {
81              int temp = fittest.genes[i];
82              fittest.genes[i] = secondFittest.genes[i];
83              secondFittest.genes[i] = temp;
84
85          }
86
87      }
88
89      //Mutation
90      void mutation() {
91          Random rn = new Random();
92
93          //Select a random mutation point
94          int mutationPoint = rn.nextInt(population.individuals[0].geneLength);
95
96          //Flip values at the mutation point
97          if (fittest.genes[mutationPoint] == 0) {
98              fittest.genes[mutationPoint] = 1;
99          } else {
100             fittest.genes[mutationPoint] = 0;
101         }
102
103         mutationPoint = rn.nextInt(population.individuals[0].geneLength);
104
105         if (secondFittest.genes[mutationPoint] == 0) {
106             secondFittest.genes[mutationPoint] = 1;
107         } else {
108             secondFittest.genes[mutationPoint] = 0;
109         }
110     }
111
112     //Get fittest offspring
113     Individual getFittestOffspring() {
114         if (fittest.fitness > secondFittest.fitness) {
115             return fittest;
116         }
117         return secondFittest;
118     }
119
120
121     //Replace least fittest individual from most fittest offspring
122     void addFittestOffspring() {
```

```
123
124         //Update fitness values of offspring
125         fittest.calcFitness();
126         secondFittest.calcFitness();
127
128         //Get index of least fit individual
129         int leastFittestIndex = population.getLeastFittestIndex();
130
131         //Replace least fittest individual from most fittest offspring
132         population.individuals[leastFittestIndex] = getFittestOffspring();
133     }
134
135 }
136
137
138 //Individual class
139 class Individual {
140
141     int fitness = 0;
142     int[] genes = new int[5];
143     int geneLength = 5;
144
145     public Individual() {
146         Random rn = new Random();
147
148         //Set genes randomly for each individual
149         for (int i = 0; i < genes.length; i++) {
150             genes[i] = Math.abs(rn.nextInt() % 2);
151         }
152
153         fitness = 0;
154     }
155
156     //Calculate fitness
157     public void calcFitness() {
158
159         fitness = 0;
160         for (int i = 0; i < 5; i++) {
161             if (genes[i] == 1) {
162                 ++fitness;
163             }
164         }
165     }
166
167 }
168
169 //Population class
170 class Population {
171
172     int popSize = 10;
173     Individual[] individuals = new Individual[10];
174     int fittest = 0;
175
176     //Initialize population
177     public void initializePopulation(int size) {
178         for (int i = 0; i < individuals.length; i++) {
179             individuals[i] = new Individual();
180         }
```

```
181     }
182
183     //Get the fittest individual
184     public Individual getFittest() {
185         int maxFit = Integer.MIN_VALUE;
186         int maxFitIndex = 0;
187         for (int i = 0; i < individuals.length; i++) {
188             if (maxFit <= individuals[i].fitness) {
189                 maxFit = individuals[i].fitness;
190                 maxFitIndex = i;
191             }
192         }
193         fittest = individuals[maxFitIndex].fitness;
194         return individuals[maxFitIndex];
195     }
196
197     //Get the second most fittest individual
198     public Individual getSecondFittest() {
199         int maxFit1 = 0;
200         int maxFit2 = 0;
201         for (int i = 0; i < individuals.length; i++) {
202             if (individuals[i].fitness > individuals[maxFit1].fitness) {
203                 maxFit2 = maxFit1;
204                 maxFit1 = i;
205             } else if (individuals[i].fitness > individuals[maxFit2].fitness) {
206                 maxFit2 = i;
207             }
208         }
209         return individuals[maxFit2];
210     }
211
212     //Get index of least fittest individual
213     public int getLeastFittestIndex() {
214         int minFitVal = Integer.MAX_VALUE;
215         int minFitIndex = 0;
216         for (int i = 0; i < individuals.length; i++) {
217             if (minFitVal >= individuals[i].fitness) {
218                 minFitVal = individuals[i].fitness;
219                 minFitIndex = i;
220             }
221         }
222         return minFitIndex;
223     }
224
225     //Calculate fitness of each individual
226     public void calculateFitness() {
227
228         for (int i = 0; i < individuals.length; i++) {
229             individuals[i].calcFitness();
230         }
231         getFittest();
232     }
233
234 }
```

6 Sample Input/Output (Compilation, Debugging & Testing)

Generation: 0 Fittest: 4
Generation: 1 Fittest: 4
Generation: 2 Fittest: 3
Generation: 3 Fittest: 3
Generation: 4 Fittest: 4
Generation: 5 Fittest: 4
Generation: 6 Fittest: 3
Generation: 7 Fittest: 3
Generation: 8 Fittest: 3
Generation: 9 Fittest: 3
Generation: 10 Fittest: 3
Generation: 11 Fittest: 5

Solution found in generation 11
Fitness: 5
Genes: 11111

7 Lab Task (Please implement yourself and show the output to the instructor)

1. Modify the program to take input as text file from computer.

8 Lab Exercise (Submit as a report)

- Write a program to perform N-Queen problem using Genetic Algorithms.

9 Policy

Copying from internet, classmate, seniors, or from any other source is strongly prohibited. 100% marks will be *deducted* if any such copying is detected.