INFO6205_03 GeneticAlgorithmTSP

Tianqi Zheng/ Ruofan Lyu

Problem Statement

This application uses genetic algorithm to solve the Traveling Salesman Problem. The problem is to find out the shortest possible route for a salesman who travelled between each city and back to his origin. This classic problem can be the prototype of a lot of practical tasks which has great research value.

Basic Algorithm Formula

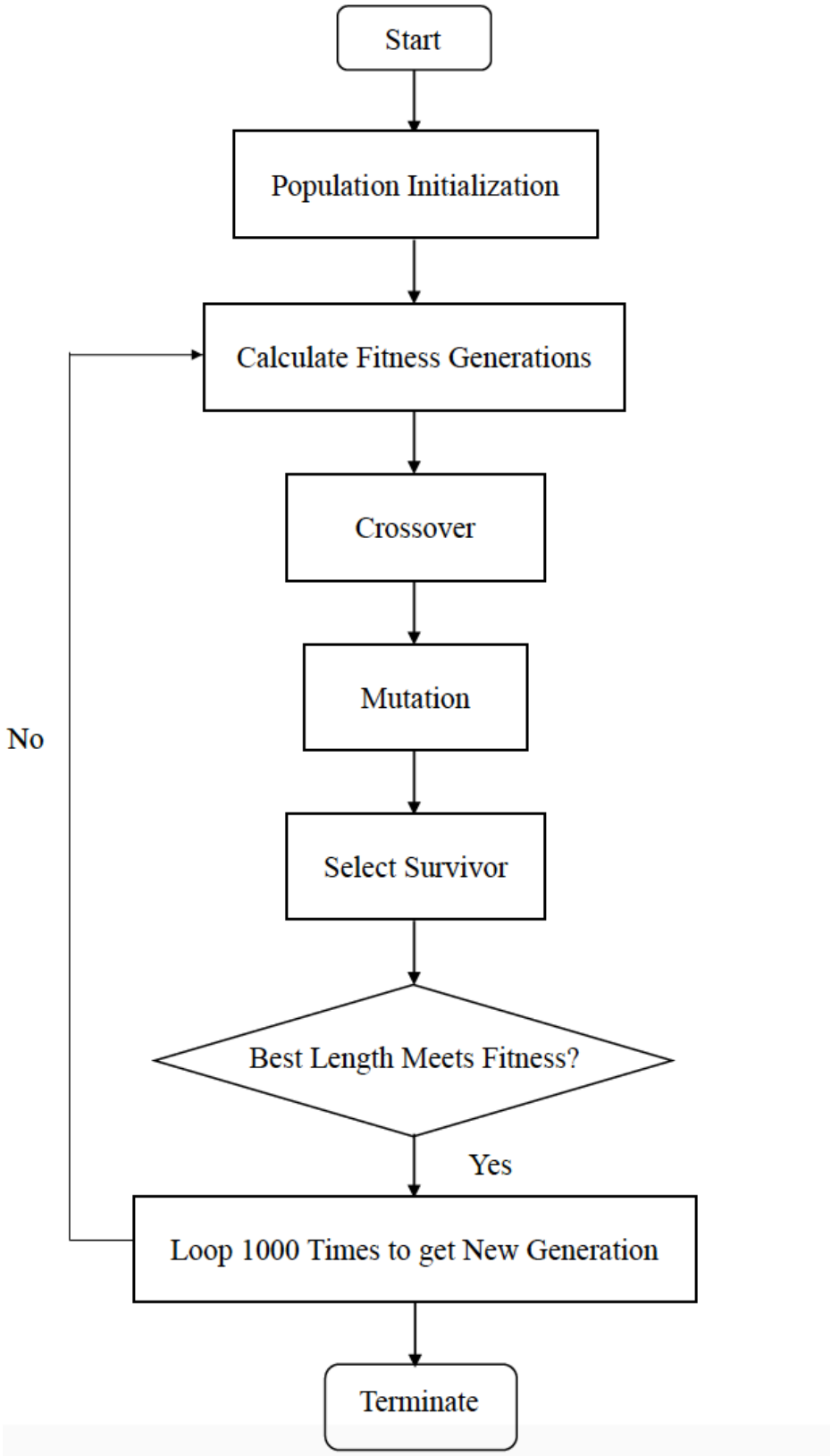
Genetic Algorithms (GAs) are adaptive heuristic search algorithm based on the evolutionary ideas of natural selection and genetics, specially those follow the principles first laid down by Charles Darwin of "survival of the fittest.".

GAs are based on an analogy with the genetic structure and behavior of chromosomes within a population of individuals using the following foundations:

- Individuals in a population compete for resources and mates.
- Those individuals most successful in each 'competition' will produce more offspring than those individuals that perform poorly.
- Genes from `good' individuals propagate throughout the population so that two good parents will sometimes produce offspring that are better than either parent.
- Thus each successive generation will become more suited to their environment.

The flow of GA we're using shows as the following:

Algorithm flow:



Program Snippets

GeneticAlgorithm.java

The main algorithm class of this project. Implementation of the methods of genetic algorithm including initialization, selection, evolution, etc.

```
public void init(String filename) throws IOException {
    // Read data
    int[] x;
    int[] y;
    String strbuff;
    BufferedReader data = new BufferedReader(new InputStreamReader(new FileInputStream(filename)));
    distance = new int[cityNum][cityNum];
    x = new int[cityNum];
    y = new int[cityNum];
    for (int i = 0; i < cityNum; i++) {

        // Read one line, parameter
        // city number, coordinate of x, coordinate of y
        strbuff = data.readLine();

        // Split a line of data
        String[] strcol = strbuff.split(" ");

        // coordinator of x
        x[i] = Integer.valueOf(strcol[1]);

        // coordinate of y
        y[i] = Integer.valueOf(strcol[2]);
    }
```

To initialize the first generation, we selected a sample data set and use input stream in Java to extract the data and put into the distance array.

Compute the distance:

```
for (int i = 0; i < cityNum - 1; i++) {
    distance[i][i] = 0;
    for (int j = i + 1; j < cityNum; j++) {
        double rij = Math.sqrt(((x[i] - x[j]) * (x[i] - x[j]) + (y[i] - y[j]) * (y[i] - y[j])) / 10.0);
        int tij = (int) Math.round(rij);
        if (tij < rij) {
            distance[i][j] = tij + 1;
            distance[j][i] = distance[i][j];
        } else {
            distance[i][j] = tij;
            distance[j][i] = distance[i][j];
        }
    }
}
```

Then find the best child gene by compare with the known fitness:

```
private void selectBestGene() {
    int k, i, maxid;
    int maxevaluation;

    maxid = 0;
    maxevaluation = fitness[0];
    for (k = 1; k < scale; k++) {
        if (maxevaluation > fitness[k]) {
            maxevaluation = fitness[k];
            maxid = k;
        }
    }

    if (bestLength > maxevaluation) {
        bestLength = maxevaluation;
        BestGeneration = t;
        for (i = 0; i < cityNum; i++) {
            BestPath[i] = InitialGeneration[maxid][i];
        }
    }

    // Copy that gene
    copyGene(0, maxid);
}
```

Randomly select gene within scale, copy gene:

```
private void select() {
    int k, i, selectId;
    float ran1;
    for (k = 1; k < scale; k++) {
        ran1 = (float) (random.nextInt(65535) % 1000 / 1000.0);
        for (i = 0; i < scale; i++) {
            if (ran1 <= IntegrationPossibility[i]) {
                break;
            }
        }
        selectId = i;
        copyGene(k, selectId);
    }
}
```

GeneralEvolution() is the main evolution method in this class:

```
 private void GeneralEvolution() {
     int k;

     selectBestGene();
     select();

     float r;


// Crossing, the group will evolute to next generation
for (k = 0; k < scale; k = k + 2) {

    // Crossing possibility of k and k + 1
    r = random.nextFloat();
    if (r < CrossingPossibility) {
        Crossing.SameGeneOXCross(k, k + 1, cityNum, NewGeneration, random);
    } else {

        // Mutation possibility of this gene (k)
        r = random.nextFloat();

        // Mutation
        if (r < MutationPossibility) {
            Crossing.OnCVariation(k, cityNum, NewGeneration, random);
        }

        // Mutation possibility (k + 1)
        r = random.nextFloat();

        // Mutation
        if (r < MutationPossibility) {
            Crossing.OnCVariation(k + 1, cityNum, NewGeneration, random);
        }
    }

}
```

Select different crossover methods according to crossing possibility and mutation possibility and get new generation by passing in the parameters to these methods.

GeneralSolve() method generates next generation with gene crossing:

```
// Initial the group
initGroup();

// Initial group fitness
for (k = 0; k < scale; k++) {
    fitness[k] = Evaluation.evaluate(InitialGeneration[k], cityNum, distance);
}

// Compute initial generation's integration possibility
Evaluation.countRate(scale, IntegrationPossibility, fitness);
```

This method returns the best length of the path.

```
for (t = 0; t < GenerationTimes; t++) {

    GeneralEvolution();

    // Copy, prepare to generate
    for (k = 0; k < scale; k++) {
        for (i = 0; i < cityNum; i++) {
            InitialGeneration[k][i] = NewGeneration[k][i];
        }
    }

    // Count fitness
    for (k = 0; k < scale; k++) {
        fitness[k] = Evaluation.evaluate(InitialGeneration[k], cityNum, distance);
    }

    // Compute integration possibility
    Evaluation.countRate(scale, IntegrationPossibility, fitness);
}
```

In addition, we've implemented non-crossing evolution methods, which is a sequential crossover method rather than the general evolution:

```
for (k = 1; k + 1 < scale / 2; k = k + 2) {

    // Crossing possibility of k and k + 1
    r = random.nextFloat();
    if (r < CrossingPossibility) {
        Crossing.SameGeneOXCross(k, k + 1, cityNum, NewGeneration, random);
    } else {

        r = random.nextFloat();
        if (r < MutationPossibility) {
            Crossing.OnCVariation(k, cityNum, NewGeneration, random);
        }

        r = random.nextFloat();
        if (r < MutationPossibility) {
            Crossing.OnCVariation(k + 1, cityNum, NewGeneration, random);
        }
    }
}

if (k == scale / 2 - 1) {
    r = random.nextFloat();
    if (r < MutationPossibility) {
        Crossing.OnCVariation(k, cityNum, NewGeneration, random);
    }
}
```

Analyse.java

Implementation of two measurement methods: getStandardDevition and getAvg, which returns the standard deviation using GA and the average length. The value return is used to test the feasibility of the algorithm.

Crossing.java

This class contains plenty of crossover operators, since there're many ways to do the crossover and get offspring, we selected order crossover.

OXCross() is an order crossover operator. It is a fairly simple permutation crossover. Swath of consecutive alleles from parent 1 drops down, and remaining values are placed in the child in the order which they appear in parent 2.

```
ran1 = random.nextInt(65535) % cityNum;
ran2 = random.nextInt(65535) % cityNum;

while (ran1 == ran2) {
    ran2 = random.nextInt(65535) % cityNum;
}

if (ran1 > ran2) {
    temp = ran1;
    ran1 = ran2;
    ran2 = temp;
}
```

Select a random swath of consecutive alleles from parent 1

```
for (i = 0, j = ran1; i < flag; i++, j++) {
    Gh1[i] = NewGeneration[k2][j];
    Gh2[i] = NewGeneration[k1][j];
}

for (k = 0, j = flag; j < cityNum;) {
    Gh1[j] = NewGeneration[k1][k++];
    for (i = 0; i < flag; i++) {
        if (Gh1[i] == Gh1[j]) {
            break;
        }
    }
    if (i == flag) {
        j++;
    }
}

for (k = 0, j = flag; j < cityNum;) {
    Gh2[j] = NewGeneration[k2][k++];
    for (i = 0; i < flag; i++) {
        if (Gh2[i] == Gh2[j]) {
            break;
        }
    }
    if (i == flag) {
        j++;
    }
}

for (i = 0; i < cityNum; i++) {
    NewGeneration[k1][i] = Gh1[i];
    NewGeneration[k2][i] = Gh2[i];
}
```

Drop the swath down to Child 1 and mark out these alleles in Parent 2. Starting on the right side of the swath, grab alleles from parent 2 and insert them in Child 1 at the right edge of the swath.

SameGeneOXCross() is the improvement of OXCross(). It added mutate process into crossover, the same gene are crossing together to mutate new gene. OnCVariation() is a multi-change mutation operator, it fully-random generate the generation.

The reason we use order crossover operator is it is fastest of all crossover operators, it requires virtually no overhead operations. Also, it meets the requirement of TSP. We first select a father generation which is already optimization and make it the first of children. The child gene remain to be generated are random based on the parent. Thus, we need a highly-random crossover operator.

Evaluation.java

This class contains two methods: evaluate() and countRate(). evaluate() takes an array of chromosome, an array of distance from city to city and city number, it calculates and returns an integer of the distance of a given generation which is the fitness we are looking for to get the best solution. countRate() takes the integer scale, array IntegrationPossibility and array fitness, it generate the content of array IntegrationPossibility.

Main.java

The class to run the project with testing outputs. 1000 times of generate new gene to observe the result.

Testing.java

This class contains two testing methods: testGeneral() and testNoCrossing(), which passes in parameters to run the genetic algorithm in GeneticAlgorithm.java.

Program Testing

To test the validity of the project, we've implemented a testing class——**GATest.java**.

We've created several class invariants for this class to go through the tests, and set up 5 tests.

- testInitialDistance(): to test the distance matrix has been successfully initialized or not
- testGeneralSolve(): to test if the general solve methods does work in the GeneticAlgorithm class
- testNoCrossingSolve():to test if the no crossing solve methods does work in the GeneticAlgorithm class
- testGeneralSolveResult(): to test the correctness of result calculated by general genetic algorithm by getting its standard deviation
- testNoCrossingSolveResult():to test the correctness of result calculated by no crossing genetic algorithm by getting its standard deviation

Results

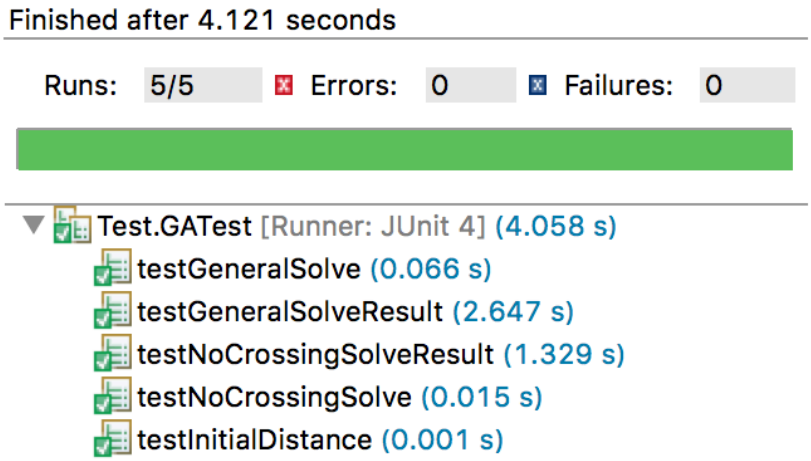
As we run the main class, the result and system logs are showing as the following:

```
信息: Best generation of no crossing evolution: 979
十二月 07, 2017 6:20:16 下午 GeneticAlgorithm.Testing testNoCrossing
信息: Best generation of no crossing evolution: 979
十二月 07, 2017 6:20:16 下午 GeneticAlgorithm.Testing testNoCrossing
信息: Best generation of no crossing evolution: 979
十二月 07, 2017 6:20:16 下午 GeneticAlgorithm.Testing testNoCrossing
信息: Best generation of no crossing evolution: 979
十二月 07, 2017 6:20:16 下午 GeneticAlgorithm.Testing testNoCrossing
信息: Best generation of no crossing evolution: 979
十二月 07, 2017 6:20:16 下午 GeneticAlgorithm.Testing testNoCrossing
信息: Best generation of no crossing evolution: 979
十二月 07, 2017 6:20:16 下午 GeneticAlgorithm.main main
信息: No Crossing evolution finished.
十二月 07, 2017 6:20:16 下午 GeneticAlgorithm.main main
信息: Result of general evolution:
Best generation of General Evolution: 581
Best Path of General Evolution:
19 -> 24 -> 4 -> 32 -> 11 -> 12 -> 22 -> 6 -> 18 -> 27 ->
8 -> 42 -> 43 -> 17 -> 15 -> 28 -> 47 -> 23 -> 44 -> 34 ->
25 -> 3 -> 1 -> 29 -> 46 -> 13 -> 9 -> 41 -> 31 -> 38 ->
21 -> 16 -> 5 -> 14 -> 37 -> 2 -> 20 -> 10 -> 7 -> 40 ->
33 -> 45 -> 30 -> 26 -> 36 -> 35 -> 0 -> 39 -> 0
Average length of General Solving: 28672
Standard Division of General Solving: 18066.538927641897

十二月 07, 2017 6:20:16 下午 GeneticAlgorithm.main main
信息: Result of no crossing evolution:
Best generation of No Crossing Evolution: 979
Best Path of No Crossing Evolution:
3 -> 25 -> 47 -> 4 -> 41 -> 31 -> 38 -> 12 -> 46 -> 20 ->
22 -> 10 -> 39 -> 11 -> 32 -> 19 -> 29 -> 36 -> 5 -> 26 ->
18 -> 16 -> 42 -> 27 -> 17 -> 6 -> 30 -> 14 -> 45 -> 35 ->
43 -> 37 -> 8 -> 7 -> 0 -> 15 -> 21 -> 2 -> 40 -> 33 ->
13 -> 24 -> 28 -> 1 -> 9 -> 23 -> 34 -> 44 -> 0
Average length of No Crossing Solving: 13848
Standard Division of No Crossing Solving: 3241.6373328921295
```

It shows the logs, best generation, best evolution path, average length of the observation and standard deviation.

For the Junit test, the results as following:



As it shown, all tests passed which means the project does work accurately.

Conclusion

Through implementing this project, we found that genetic algorithm is a effective way to generate high-quality solutions to optimization and search problems by relying on bio-inspired operators.

In this particular task, we need to find out the best way for the salesman to travel between cities, GA helps a lot on this problem. Comparing the results, it's not hard to see that the no crossing evolution is more accurate than the general evolution. Due to the indeterminacy of the new offspring generated by the parent of general evolution, the possibility of produce worse gene became higher. Since the no crossing evolution directly crossover two parent gene, the number of crossover has been reduced. In this case, the optimization possibility increased, better generation can be produced.

README.md

This application uses genetic algorithm to solve the Traveling Salesman Problem. The test data comes from TSPLIB, and the best solution is known as 10628.

The test file att48.tsp should be put at the root path of the project.

(1) The city index (Vertex in the graph) is our gene in this solution, and the number of genes is 48.

(2) An array of genes in this solution is our trait (the order in the trait is the path direction in the graph, order [1, 3, 5, 2] means the path in the graph is 1 -> 3 -> 5 -> 2), each trait includes city index from 1 to 48. 30 traits form a group.

(3) The fitness of each trait is calculated by computing the path length of this trait. Path length (Edge in the graph) is stored in a distance matrix when initiating the InitialGeneration group, fitness function will call it to compute.

(4) We will let our group evolve 1,000 times. After each evolution, we find the best fit trait and set that trait to be the first trait in the next generation, then let the other traits to evolve using a function called "Crossing", also there will be mutations.

(5) Each group will have a best trait, this is the current best solution of TSP.

(6) After each evolution, best trait will be logged in the console.

(7) After 1,000 evolution, application will log the best fit result.

(8) We have two different evolution function, the analyze of result will be logged.