Project report from Team 13: Nan Deng & Yuan Li

Problem:

In our project, we are trying to solve the Travelling Salesman Problem (TSP) using Genetic Algorithm. When given a set of cities and distance between every pair of cities, the goal is to find the shortest possible route that visits every city exactly once and returns to the starting point.

Dataset:

The datasets we used come from the TSPLIB[1], which includes the FRI26 (a set of 26 cities and the minimal tour has length 937), GR17 (a set of 17 cities and the minimal tour has length 2085) and the ATT48 (a set of 48 US state capitals and the minimal tour has length 10628).

Implementation:

- **Gene expression**: in our model, each chromosome stands for a path of all places (cities) and each gene stands for a place (city). So isolate the **Phenotype** and the **Genotype**, we adopted the following strategy: we give each city a total order by which they are indexed; depending on the data size, the index can be mapped to its binary form which composes its gene type. Here is an example:
    - If we have 10 countries: AF, BR, CA, CN, FR, IN, JP, RS, UK, US which are the **phenotype**
    - They are indexed as: AF -> 1, BR -> 2, CA -> 3, …., US -> 10
    - The index is mapped to binary form: 1 -> 0001, 2 -> 0010, 3 -> 0011, …., 10 -> 1010
    - So the original country list AF -> BR -> CA -> CN -> FR -> IN -> JP -> RS -> UK -> US can be mapped into: 0001 -> 0010 -> 0011 -> 0100 -> 0101 -> 0110 -> 0111 -> 1000 -> 1001 -> 1010
    - The **genotype** are: 000100100011010001010110011110001001101 0

  The actual decode (bytes to String code) implementation is as followed:

```
// Decode the string
public final String decodeChromo() {

    // Create a buffer
    decodeChromo.setLength(0);

    // Loop throught the chromo
    for (int x=0;x<chromo.length();x+=4) {
        // Get the
        int idx = Integer.parseInt(chromo.substring(x,x+4), 2);
        if (idx < nodes.length) decodeChromo.append(nodes[idx]);
    }

    // Return the string
    return decodeChromo.toString();
}
```

  In order to translate between genotype and phenotype, we use HashMap<String, Integer> to represent their mapping relationship.

- **Crossover**: when two chromosomes meet each other, there is a chance they will reproduce their children. The children will inherit genes from parents by swapping part of the list. This was done by randomly choosing a position and swap the left half and right half among two chromosomes. Let's see this example

    - We have two chromosomes: 0010, 0011, 0100 and 0101, 1010, 1011
    - Then we use Random.nextInt() to generate a random number between 0 and 12 as the cutting point. Let's say it is 6. So we cut the two chromosomes by the middle point 001000110100 / 010110101011 and then swap the bits: 001000101011 / 010110110100.

  Below is the real implementation:

```
// Crossover bits
public final void crossover(Chromosome other) {
    int len = this.nodes.length;

    // Should we cross over?
    if (rand.nextDouble() > crossRate) return;

    // Generate a random position
    int pos = rand.nextInt(len);
    //System.out.println(pos);

    Set<String> secondHalf = new HashSet<>();
    for (int x = pos; x < len; x++) {
        secondHalf.add(this.nodes[x]);
    }

    String[] temp = new String[len];

    // copy 2nd half to temp
    int right = pos; int left = 0;
    for (int i = 0; i < len; i++) {
        if (secondHalf.contains(other.nodes[(i+pos)%len])) {
            temp[right++] = other.nodes[(i+pos)%len];
        } else {
            temp[left++] = other.nodes[(i+pos)%len];
        }
```

- **Mutation**: there is a possibility that a gene can mutate – swapping the genes in itself. This is done by randomly choosing two position and exchanging their corresponding gene. If we choose 1 & 3 from 0010, 0011, 0100 and swap them then the chromosome becomes 0100, 0011, 0010.

```
// Mutation - swap two elements' position
public final void mutate() {
    // shall we mutate?
    if (rand.nextDouble() <= mutRate) {
        //System.out.println("We have a mutation");
        int swapsNo = rand.nextInt(nodes.length/2);
        for (int i = swapsNo; i > 0; i--) {
            int first = rand.nextInt(nodes.length);
            int second = rand.nextInt(nodes.length);
            String tmp = nodes[first];
            nodes[first] = nodes[second];
            nodes[second] = tmp;
        }
    }

}
```

- **Elimination**: in one colony as the food and resource are limited, only a certain amount of chromosomes can survive. So we only keep the first N members who have the highest score.

```
// eliminate
for (int i = list.size(); i >= originalSize; i--) {
    list.remove(list.size()-1);
}
```

- **Fitness function**: in our TSP problem, the fitness function is expressed by adding the distance between each adjacent places up and return the total. The distance is maintained in a 2D matrix. Let's see an example:

	UK	US	CN	IN	RS
UK	0	1	100	100	100
US	100	0	1	100	100
CN	100	100	0	1	100
IN	100	100	100	100	1
RS	1	100	100	100	0

Given a path from UK - > US -> CN -> IN -> RS -> UK, we can find the total distance is 5. For this test case, this is the optimal solution.
Code structure looks like this:

```
// Add up the contents of the decoded chromo
public final int calcDistance() {
    int tot = 0;
    for (int i = 1; i < nodes.length; i++) {
        tot += distances[placeToIndex.get(nodes[i-1])][placeToIndex.get(nodes[i])];
    }
    return tot + distances[placeToIndex.get(nodes[nodes.length-1])][placeToIndex.get(nodes[0])];
}
```

- **Sorting function**: when we have multiple chromosomes (paths), they are sorted by their distance. To achieve this we utilized the Collection.sort() and passed in a new Comparator<>()

```
// sort
Collections.sort(list, new Comparator<Chromosome>(){
    @Override
    public int compare(Chromosome o1, Chromosome o2) {
    return o1.calcDistance() - o2.calcDistance();
    }
});
```

- **Evolution Mechanism**: to evolve, we follow the following pattern. At each generation, the sequence is mutate -> crossover -> sort -> best score -> eliminate -> gen++

```
while (true) {
    // mutate
    for (Chromosome ch: list) { ch.mutate(); }

    // crossover
    for (int i = 0; i < originalSize/2; i++) { swap (first, second)}

    // sort
    Collections.sort(list);

    // update best score
    best = list.get(0).calcDistance() > best ? ;

    // eliminate
    for (int i = list.size(); i >= originalSize; i--) { list.remove(last);}

    // increment generation
    gen++;
```

- **Logging function**: during the running, we kept track of two kinds of information in the console – the best score and the number of generations. To ratelimit the number of logs, the generation log will only be printed when gen is a module of 10000.

```
// print out the best score
if (cur < score || gen % 10000 == 0) {
    score = cur;  logger.info("The minimum score is: " + score);
    String path = "";
    for (String node: list.get(0).nodes) {  path += node + " -> ";  }
    logger.info("The path is: " + path);
}
```

```
// print the number of generations
gen++;
if (gen % 1000 == 0) {
    logger.info("This is the " + gen + "th generation");
}
```

- **Unit test**: while developing this project, we kept validating functions we wrote by using the JUnit test framework. To name a few:

```
@Test
public void testConstructor() {
    String[] places = {"US", "UK", "CN", "IN", "RS"};
    Chromosome cs1 = new Chromosome(5, 0.1, 0.5, places, null);
    //for (String place: cs1.nodes) System.out.println(place);
    assertTrue(places.length == cs1.nodes.length);
}

@Test
public void testMutate() {
    String[] places = {"US", "UK", "CN", "IN", "RS"};
    Chromosome cs2 = new Chromosome(5, 1.0, 0.5, places, null);
    places = new String[5];
    for (int i = 0; i < 5; i++) { places[i] = cs2.nodes[i]; }
    cs2.mutate();
    for (String place: places) System.out.print(place + " ");
    System.out.println();
    for (String place: cs2.nodes) System.out.print(place + " ");
    System.out.println();
    String[] firstReverse = {"","",};
    String[] secondReverse = {"", ""};

@Test
public void testCrossover() {
    String[] places = {"US", "UK", "CN", "IN", "RS"};

    Chromosome cs1 = new Chromosome(5, 0.1, 1.0, places, null);
    Chromosome cs2 = new Chromosome(5, 0.1, 1.0, places, null);

    for (int i = 0; i < places.length; i++) {
        cs1.nodes[i] = places[i];
        cs2.nodes[i] = places[4-i];
    }

    int pos = cs1.testCrossOver(cs2);

@Test
public void testSortDistance() {
    String[] input = {"US", "UK", "CN", "IN", "RS"};
    int[][] dist = {{0, 1, 1000, 1000, 1000},
                    {1000, 0, 1, 1000, 1000},
                    {1000, 1000, 0, 1, 1000},
                    {1000, 1000, 1000, 0, 1},
                    {1, 1000, 1000, 1000, 0}
    };

    List<Chromosome> list = new ArrayList<>();
    // create 40 chromosome
    for (int i = 0; i < 5; i++) {
        list.add(new Chromosome(5, 0.05, 1.0, input, dist));
        //System.out.println(list.get(i).calcDistance());
    }
```

Results:

The **GR17 dataset** includes 17 nodes and the shortest distance is 2085. We achieved this goal between 10,000th generation and 11,000th generation. The path was printed out and was consistent with the proven answer.

```
INFO: This is the 10000th generation
Dec 11, 2017 9:50:35 AM Runner main
INFO: The minimum score is: 2085
Dec 11, 2017 9:50:35 AM Runner main
INFO: The path is: 7 -> 8 -> 6 -> 17 -> 14 -> 15 -> 3 -> 11 -> 10 -> 2 -> 5 -> 9 -> 12 -> 16 -> 1 -> 4 -> 13 ->
Dec 11, 2017 9:50:35 AM Runner main
INFO: This is the 11000th generation
```

The **FRI26 dataset** includes 26 nodes and the shortest distance is 937. We have achieved 1015 (108.3%):

```
Dec 11, 2017 10:05:04 AM Runner main
INFO: The minimum score is: 1015
Dec 11, 2017 10:05:04 AM Runner main
INFO: The path is: 1 -> 15 -> 13 -> 11 -> 12 -> 21 -> 22 -> 25 -> 24 -> 23 -> 26 -> 17 -> 18 -> 20 -> 19 -> 16 -> 8 -> 7 -> 9 -> 10 -> 14
Dec 11, 2017 10:05:04 AM Runner main
INFO: This is the 821000th generation
```

and 955 (101.9%):

```
INFO: The minimum score is: 955
Dec 11, 2017 12:32:56 PM Runner main
INFO: The path is: 13 -> 15 -> 14 -> 3 -> 5 -> 6 -> 4 -> 2 -> 1 -> 25 -> 24 -> 23 -> 26 -> 22 -> 21 -> 17 -> 18 -> 20 -> 19 -> 16 -> 8 -
Dec 11, 2017 12:32:57 PM Runner main
INFO: This is the 4000th generation
```

and 1003 (107.0%)

```
INFO: This is the 10000th generation
Dec 11, 2017 12:36:32 PM Runner main
INFO: The minimum score is: 1003
Dec 11, 2017 12:36:32 PM Runner main
INFO: The path is: 1 -> 2 -> 3 -> 4 -> 6 -> 5 -> 7 -> 8 -> 19 -> 20 -> 18 -> 26 -> 23 -> 24 -> 25 -> 22 -> 21 -> 17 -> 16 -> 9
Dec 11, 2017 12:36:33 PM Runner main
INFO: This is the 11000th generation
```

those are not optimal but very good result (quite close to the optimal solution).

The ATT48 dataset includes 48 nodes and the shortest distance is 10628. So far the best result we can get is 34434.

```
Dec 10, 2017 11:39:50 PM Runner main
INFO: This is the 5840000th generation
Dec 10, 2017 11:39:50 PM Runner main
INFO: The minimum score is: 34434
Dec 10, 2017 11:39:50 PM Runner main
INFO: This is the 5841000th generation
```

It is still very distant from the optimal. Based on the description one possible improvement is to add more cut points when conducting crossover operation.

That's the end of the report, thanks for reading.

Sincerely, Nan and Yuan,