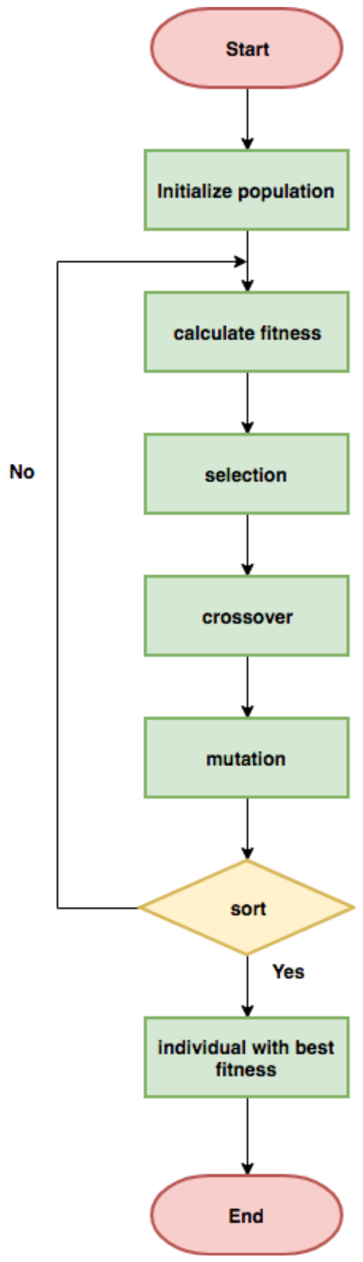
Info6205 Final Report

Changrong Chen 001276880
Tiancheng Lin 001839357

We are using Genetic Algorithm to solve TSP(Traveling Salesman Problem).

Work flow



Implementation

Classes we have:

tspga: class of genetic algorithm, work flow.

Chromosome: class of population, species node, contains gene, length of route and fitness.

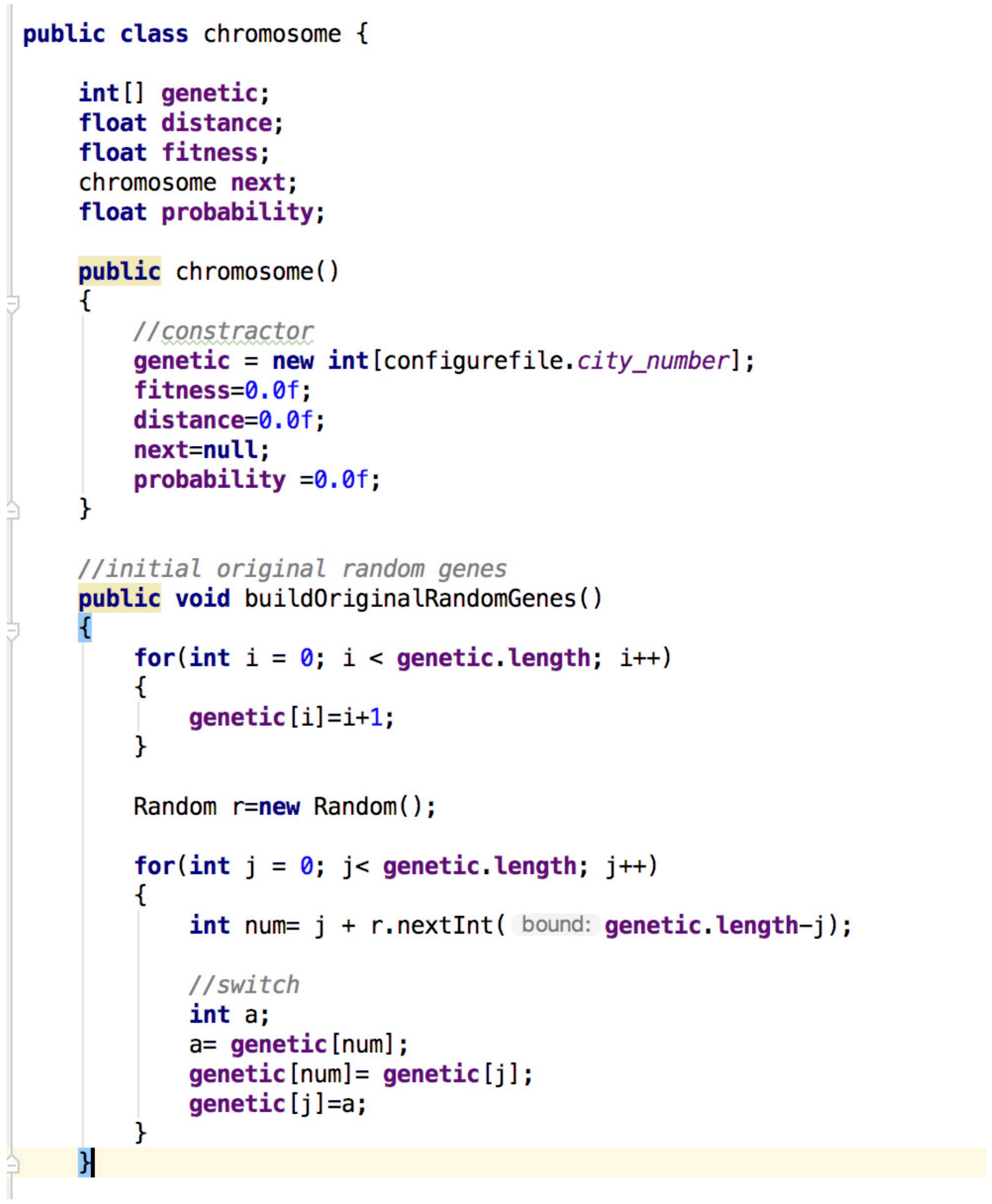
Species: list of species

Configurefile: configuration class, including map, number of population and evolution, crossover and mutation rate.

Main: main class

Initialization-Gnenetic

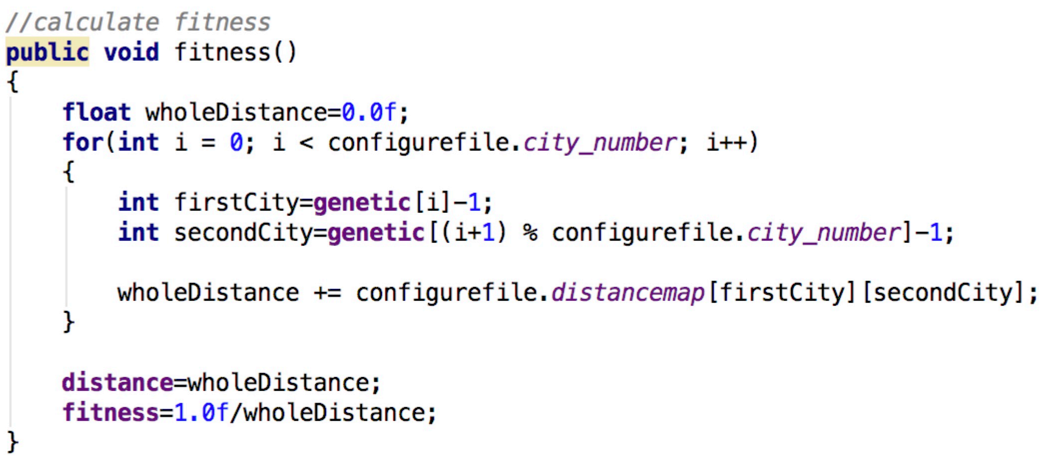
At the beginning, we need to create our original species in the class Chromosome:



```
public class chromosome {

    int[] genetic;
    float distance;
    float fitness;
    chromosome next;
    float probability;

    public chromosome()
    {
        //constractor
        genetic = new int[configurefile.city_number];
        fitness=0.0f;
        distance=0.0f;
        next=null;
        probability =0.0f;
    }

    //initial original random genes
    public void buildOriginalRandomGenes()
    {
        for(int i = 0; i < genetic.length; i++)
        {
            genetic[i]=i+1;
        }

        Random r=new Random();

        for(int j = 0; j< genetic.length; j++)
        {
            int num= j + r.nextInt( bound: genetic.length-j);

            //switch
            int a;
            a= genetic[num];
            genetic[num]= genetic[j];
            genetic[j]=a;
        }
    }
```

Using the function buildOriginalRandomGenes(), we initialized our gene code, and then by the function fitness() we could mapping the distance we could calculate the fitness of the species

Fitness:



```
//calculate fitness
public void fitness()
{
    float wholeDistance=0.0f;
    for(int i = 0; i < configurefile.city_number; i++)
    {
        int firstCity=genetic[i]-1;
        int secondCity=genetic[(i+1) % configurefile.city_number]-1;

        wholeDistance += configurefile.distancemap[firstCity][secondCity];
    }

    distance=wholeDistance;
    fitness=1.0f/wholeDistance;
}
```

After getting the fitness of our species, we could start our work flow, using the class TSPGA to doing the selection, crossover and mutation things.

Selection

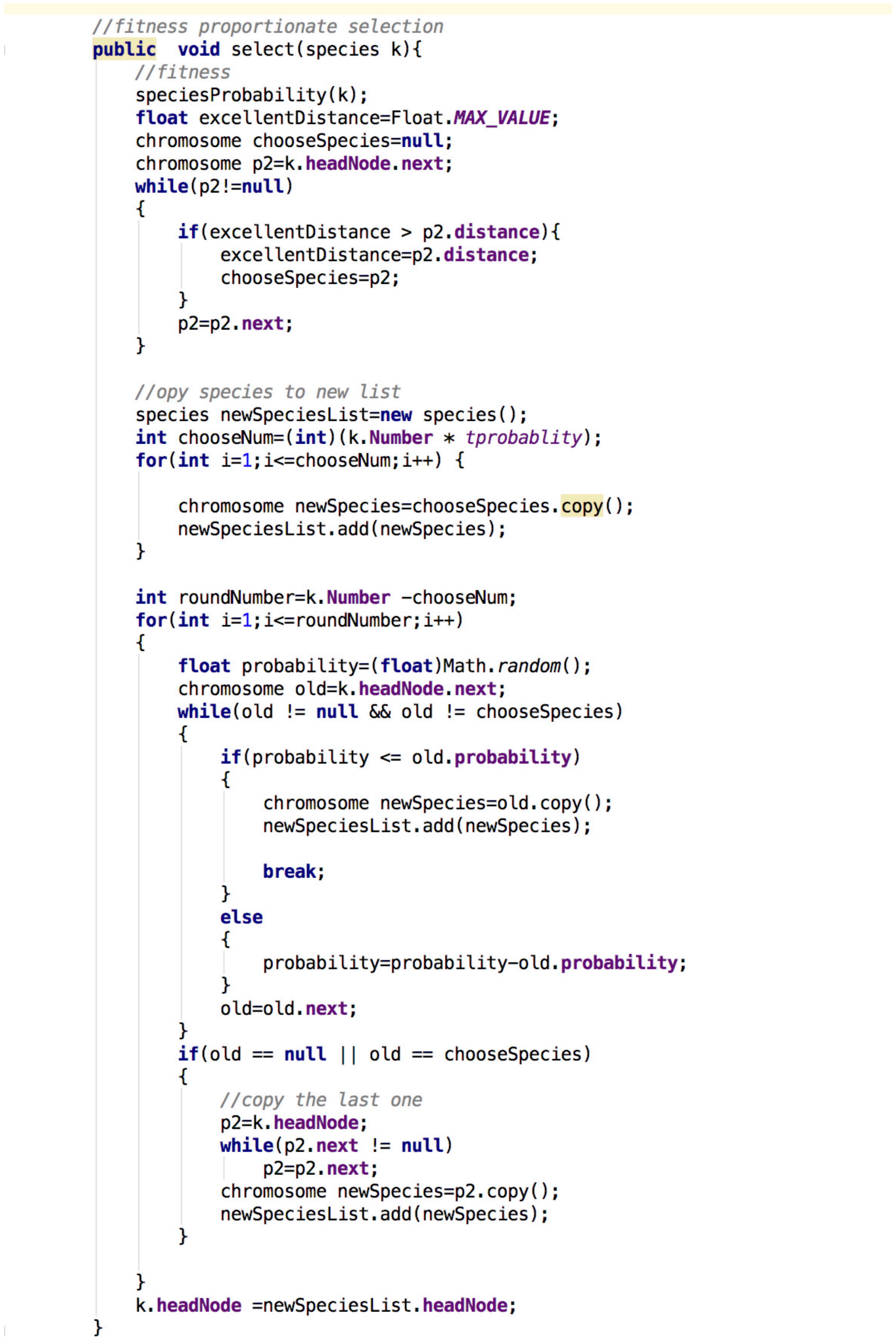
Since we have the fitness of our species f(si) (si is our species , I = 1, 2, 3, … n), we want to calculate the possibility of being selected or being inherited, that is p(si)

p(si) = f(si)/(1+2+3+…n)f(si)

And then, we need to do selection, the way we use is fitness proportionate selection or roulette wheel selection. In each round of selection, we set the size of the population the same as the previous round, keeping the total population as a constant, that is, the size of the initial population.

Before fitness proportionate selection, we select the species with highest fitness, and then copy a number of them, let them be part of the next generation, then we will have a new population. This can avoid we ignore some of the best species due to probabilistic reasons. but it is also easy to fall into a local optimum. Therefore, the number of elites needs to be continuously adjusted, here we copy 1/4 of them.

Selection code:

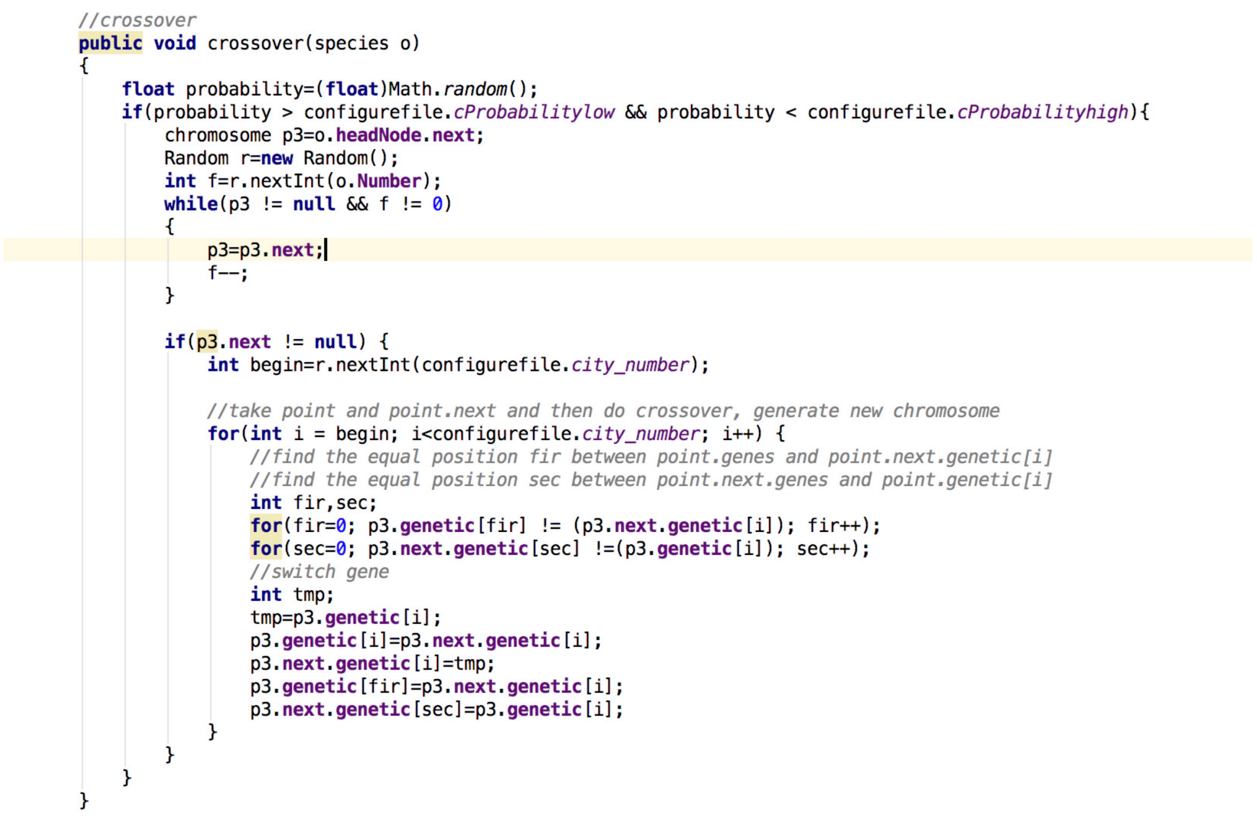


```
//fitness proportionate selection
public  void select(species k){
    //fitness
    speciesProbability(k);
    float excellentDistance=Float.MAX_VALUE;
    chromosome chooseSpecies=null;
    chromosome p2=k.headNode.next;
    while(p2!=null)
    {
        if(excellentDistance > p2.distance){
            excellentDistance=p2.distance;
            chooseSpecies=p2;
        }
        p2=p2.next;
    }

    //opy species to new list
    species newSpeciesList=new species();
    int chooseNum=(int)(k.Number * tprobablity);
    for(int i=1;i<=chooseNum;i++) {

        chromosome newSpecies=chooseSpecies.copy();
        newSpeciesList.add(newSpecies);
    }

    int roundNumber=k.Number -chooseNum;
    for(int i=1;i<=roundNumber;i++)
    {
        float probability=(float)Math.random();
        chromosome old=k.headNode.next;
        while(old != null && old != chooseSpecies)
        {
            if(probability <= old.probability)
            {
                chromosome newSpecies=old.copy();
                newSpeciesList.add(newSpecies);

                break;
            }
            else
            {
                probability=probability-old.probability;
            }
            old=old.next;
        }
        if(old == null || old == chooseSpecies)
        {
            //copy the last one
            p2=k.headNode;
            while(p2.next != null)
                p2=p2.next;
            chromosome newSpecies=p2.copy();
            newSpeciesList.add(newSpecies);
        }

    }
    k.headNode =newSpeciesList.headNode;
}
```

Crossover:

Because we use city as our gene code, when doing crossover, we randomly choose a position in the gene code as beginning, and then we switch the rest gene of two species.
For example we have two species, their gene code is 123456789 and 98765432 we take the last six numbers to crossover, and then we will have two new species 12365432 and 987456789.

Code:

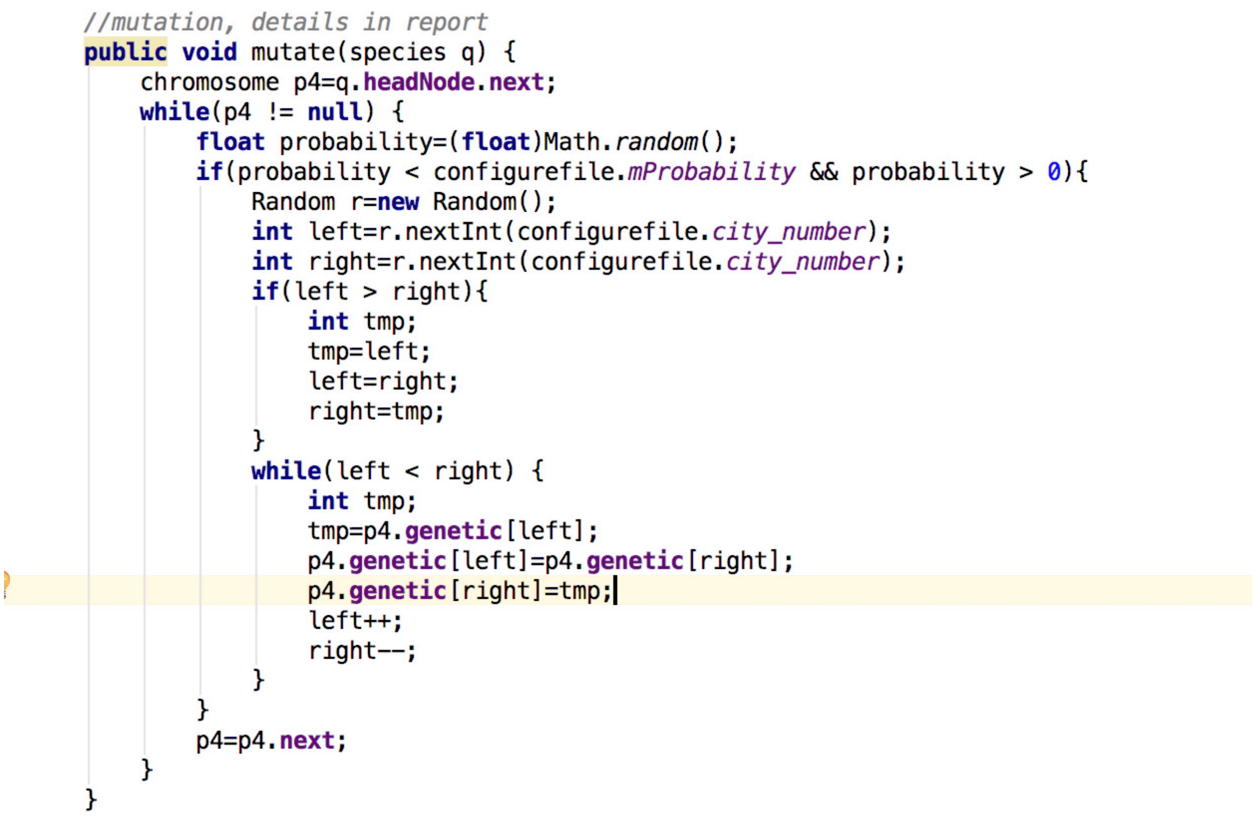


```java
//crossover
public void crossover(species o)
{
    float probability=(float)Math.random();
    if(probability > configurefile.cProbabilitylow && probability < configurefile.cProbabilityhigh){
        chromosome p3=o.headNode.next;
        Random r=new Random();
        int f=r.nextInt(o.Number);
        while(p3 != null && f != 0)
        {
            p3=p3.next;
            f--;
        }

        if(p3.next != null) {
            int begin=r.nextInt(configurefile.city_number);

            //take point and point.next and then do crossover, generate new chromosome
            for(int i = begin; i<configurefile.city_number; i++) {
                //find the equal position fir between point.genes and point.next.genetic[i]
                //find the equal position sec between point.next.genes and point.genetic[i]
                int fir,sec;
                for(fir=0; p3.genetic[fir] != (p3.next.genetic[i]); fir++);
                for(sec=0; p3.next.genetic[sec] !=(p3.genetic[i]); sec++);
                //switch gene
                int tmp;
                tmp=p3.genetic[i];
                p3.genetic[i]=p3.next.genetic[i];
                p3.next.genetic[i]=tmp;
                p3.genetic[fir]=p3.next.genetic[i];
                p3.next.genetic[sec]=p3.genetic[i];
            }
        }
    }
}
```

Mutation

We use mutation to avoid local optimum. The way we doing mutation is, find k and m in the gene, and then reverse the CITY_NUM between k and m. For example we have two species
A(n1, n2, n3…nk, nk+1…nm-1, nm …nn)
After mutation, we have
B(n1, n2, n3…nm, nm-1 … nk+1, nk …nn)

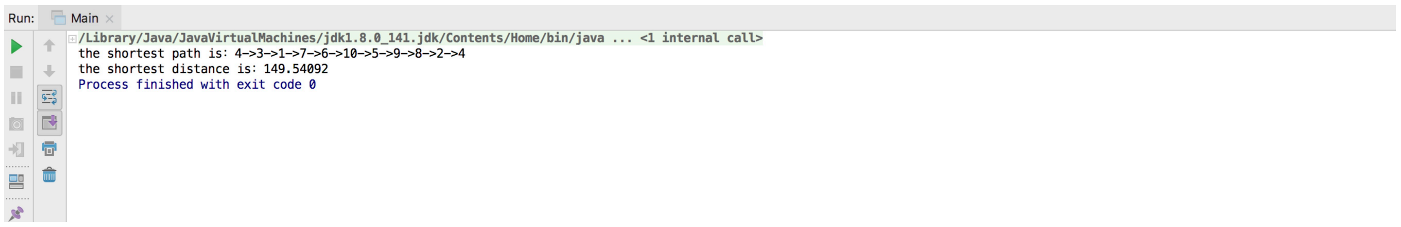
Code:



```java
//mutation, details in report
public void mutate(species q) {
    chromosome p4=q.headNode.next;
    while(p4 != null) {
        float probability=(float)Math.random();
        if(probability < configurefile.mProbability && probability > 0){
            Random r=new Random();
            int left=r.nextInt(configurefile.city_number);
            int right=r.nextInt(configurefile.city_number);
            if(left > right){
                int tmp;
                tmp=left;
                left=right;
                right=tmp;
            }
            while(left < right) {
                int tmp;
                tmp=p4.genetic[left];
                p4.genetic[left]=p4.genetic[right];
                p4.genetic[right]=tmp;
                left++;
                right--;
            }
        }
        p4=p4.next;
    }
}
```

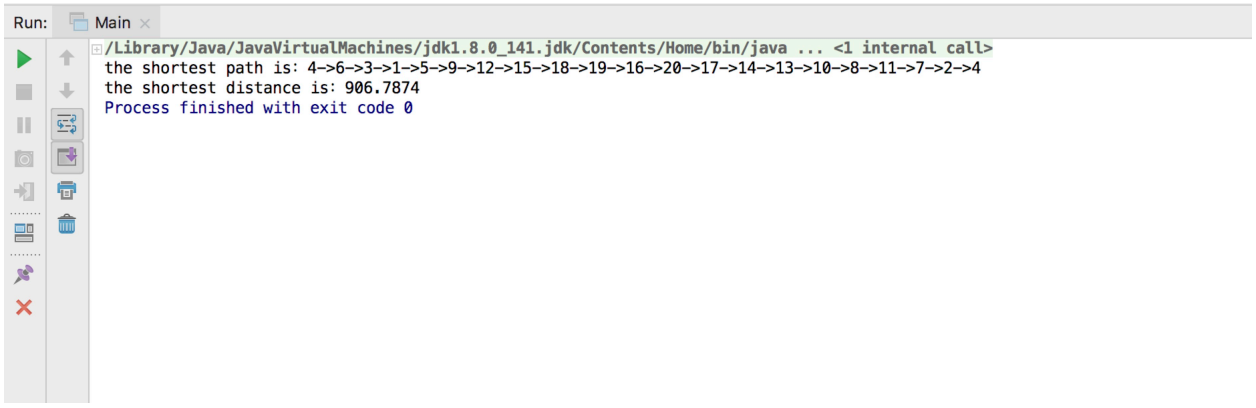
Outputs

When there are 10 cities: (0,0), (12,32), (5,25), (8,45), (33,17), (25,7), (15,15), (15,25), (25,15), (41,12)



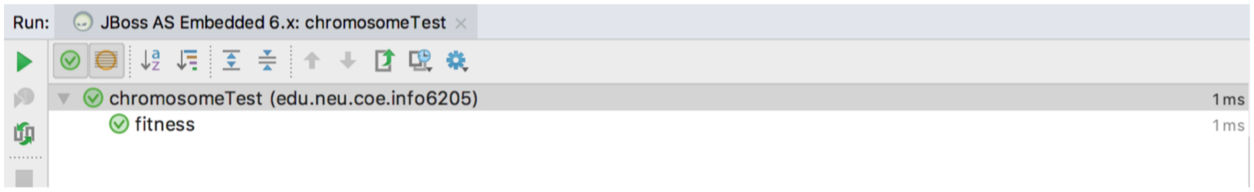
```
/Library/Java/JavaVirtualMachines/jdk1.8.0_141.jdk/Contents/Home/bin/java ... <1 internal call>
the shortest path is: 4->3->1->7->6->10->5->9->8->2->4
the shortest distance is: 149.54092
Process finished with exit code 0
```

When there are 20 cities: (60,200), (180,200), (80,180), (140,180), (20,160), (100,160), (200,160), (140,140), (40,120), (100,120), (180,100), (60,80), (120,80), (180,60), (20,40), (100,40), (200,40), (20,20),(60,20),(160,20)

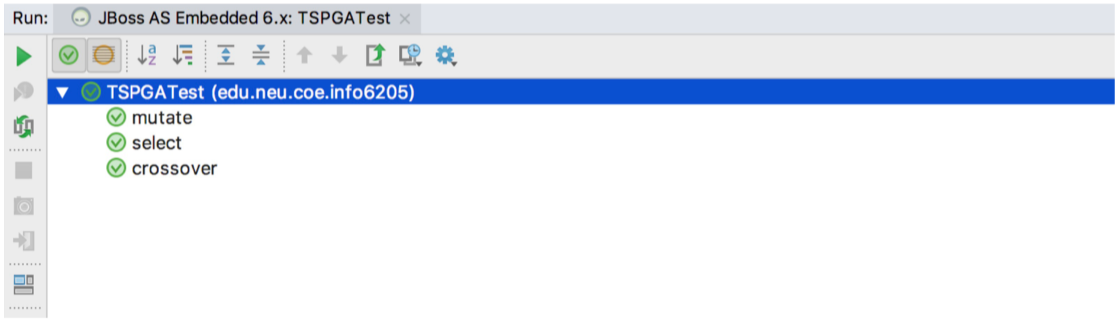


```
/Library/Java/JavaVirtualMachines/jdk1.8.0_141.jdk/Contents/Home/bin/java ... <1 internal call>
the shortest path is: 4->6->3->1->5->9->12->15->18->19->16->20->17->14->13->10->8->11->7->2->4
the shortest distance is: 906.7874
Process finished with exit code 0
```

Unit Test:



```
chromosomeTest (edu.neu.coe.info6205)    1 ms
    fitness                              1 ms
```



```
TSPGATest (edu.neu.coe.info6205)
    mutate
    select
    crossover
```

Conclusion

Number of Cities	Cities	Best Solution Length	Genetic Algorithm Solution Length	Shortest Path in Genetic Algorithm
10	(0,0),(12,32),(5,25),(8,45),(33,17),(25,7),(15,15),(15,25),(25,15),(41,12)	147.34	149.5402	4,3,1,7,6,10,5,9,8,2,4
20	(60,200),(180,200),(80,180),(140,180),(20,160),(100,160),(200,160),(140,140),(40,120),(100,120),(180,100),(60,80),(120,80),(180,60),(20,40), (100,40),(200,40),(20,20),(60,20),(160,20)	870.26	871.1174	4,6,3,1,5,9,12,15,18,19,16,20,17,14,13,10,8,11,7,2,4

After using genetic algorithm, we could find the shortest path and its length, when the number of cities are larger, the possibility to find best path using genetic algorithm are smaller, that is because the influence of the selection, crossover, mutation is larger when the number of cities are larger, so it’s unstable, when the number of cities are larger, we need to run the program more times to find the shortest path.