INFO6205 37198

Program Structure & Algorithms SEC 05 - Spring 2018

Team Project - Genetic Algorithms

Team member name	Zhiyong Zhang
Team member name	Yixuan LI
NUID	001833676
NUID	001822594
Team Number	#516

Github link: https://github.com/redbeanlyx/INFO6250_516.git

CONTENTS

1. Requirement

1.1 basic components

a) a genetic code (or use the four bases of DNA for simplicity) and a random generator/mutator of such codes;

b) gene expression: how do individual genes code for particular traits--a symbol table using a hash function?

c) a fitness function--this is essentially a measure of how good a candidate (organism) solution is for the problem you have chosen to solve;

d) a sort function (priority queue is best) -- to order the organisms by their fitness function;

e) an evolution mechanism--this takes care of the seeding of generation 0, and the births and deaths between generation N and N+1;

f) a logging function to keep track of the progress of the evolution, including the best candidate from the final generation;

g) a set of unit tests which ensure that the various components are operating properly;

h) (optional) a parallel computation mechanism so that you can divide your population up into colonies (sub-populations) and create the next generations for each colony in parallel;

i) (optional) a user interface to show the progress of the evolution.

1.2 basic concepts

1.2.1 modes of reproduction

Asexual: from a single parent

Sexual: from two parents

1.2.2 source of randomization

Crossover: some of the genes of the offspring come from the mother and some from the father. It is the main source of randomization in sexual reproduction.

Mutation: It is relatively rare. Nevertheless, mutation is the only mechanism available to asexual organisms.

1.2.3 genotype and phenotype

Genotype: The set of the genes in an organism is called the genotype. In nature, genes are coded in DNA。

Phenotype: Phenotype has fitness in an environment. It is made up of a set of individual traits, each of which corresponds to a single gene. The phenotype is expressed from the genotype.

2. Project Objectives

The goal of the project is to develop a genetic algorithm and to use it to find a good solution to a highly complex problem.

The basic idea behind GAs is to model the search for a solution in the same way that organisms in nature are adapted to their environment and lifestyle. You start with a random population (generation 0) and you "breed" successive generations from that. In each generation you "cull" those organisms which are unfit for the environment and you give birth to new organisms which are based, genetically, on the organisms which are fit.

3. Problem Description

The knapsack problem or rucksack problem is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.

The problem often arises in resource allocation where there are financial constraints and is studied in fields such as combinatorics, computer science, complexity theory, cryptography, applied mathematics, and daily fantasy sports.

The knapsack problem has been studied for more than a century, with early works dating as far back as 1897. The name "knapsack problem" dates back to the early works of mathematician Tobias Dantzig (1884–1956), and refers to the commonplace problem of packing the most valuable or useful items without overloading the luggage.

4. Runtime Environment

Operating System	Mac OS
Model Name	MacBook Pro
Processor Name	Intel Core i5
Processor Speed	3.1 GHz
Number of Processors	1
Total Number of Cores	2
Memory	8 GB
IDE	Intellij

5. System Design

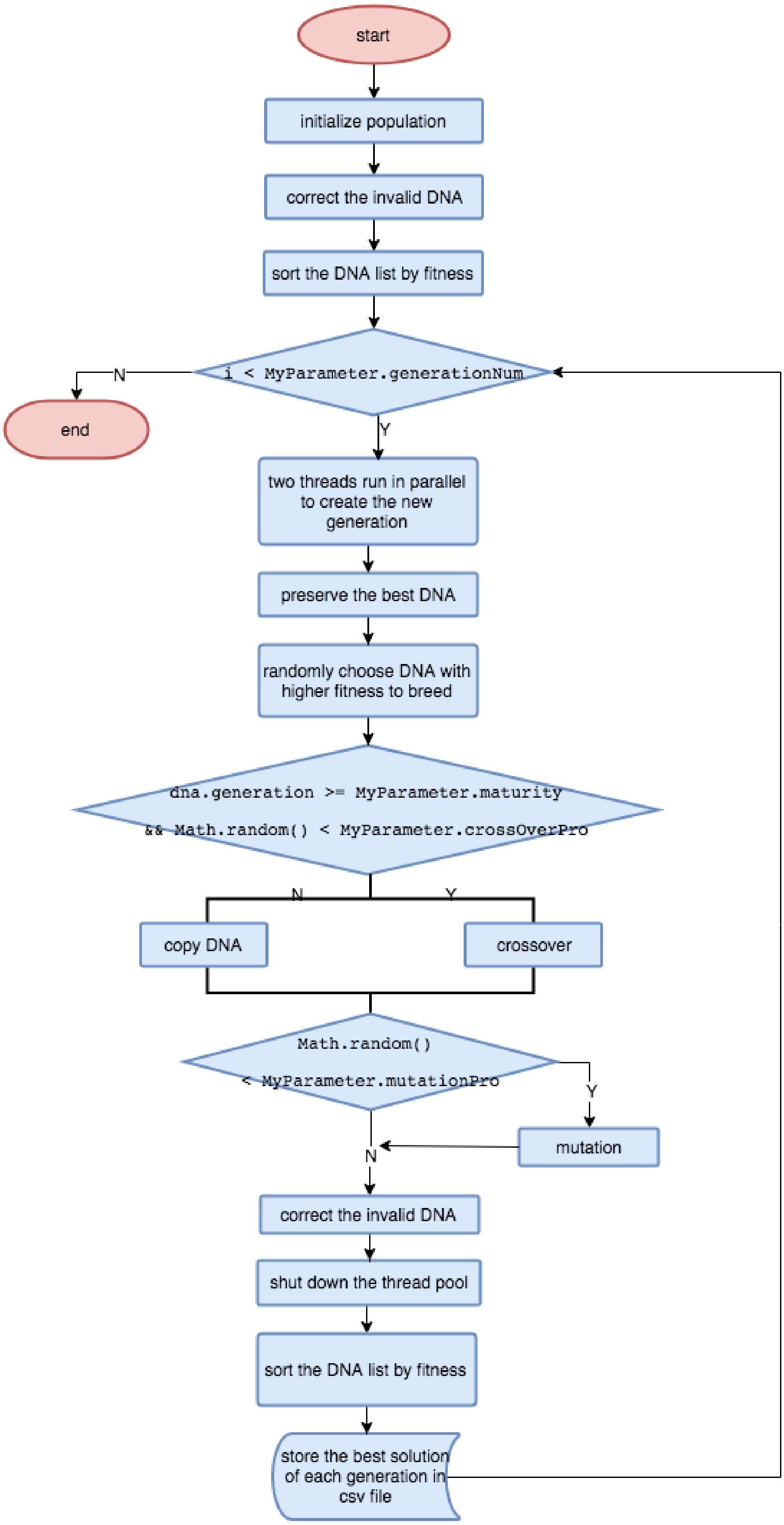
5.1 constants

variable	value	description
itemNum	2000	Number of items
weightMax	10000	Maximum weight of the bag
coreNum	2	Total number of cores

5.2 parameters

variable	value	description
populationNum	1000	the nunber of DNAs in one population
survivePro	0.5f	The proportion of DNA that survive and breed
fecundity	2	The number of offspring of a pair of parents
maturity	1	Generations to reproductive maturity
generationNum	10000	Maximum number of generations
crossOverPro	0.5f	The probability of crossover
mutationPro	0.01f	The probability of mutation

6. Control Flow



7. Code Analysis

7.1 data structures

7.1.1 DNA

DNA
+ sequence: boolean[]
+ weight: int
+ value: int
+ generation: int

7.1.2 Item

Item
+ weight: int
+ value: int

7.2 functions

a) fitness function

this is essentially a measure of how good a candidate (organism) solution is for the problem

calculate the sum of weight and value of the DNAs

```
public void calc() {
    int weightResult = 0;
    int valueResult = 0;
    for (int i = 0; i < sequence.length; i++) {
        if (sequence[i]) {
            weightResult = weightResult + MyConstant.items[i].weight;
            valueResult = valueResult + MyConstant.items[i].value;
        }
    }
    weight = weightResult;
    value = valueResult;
}
```

b) correction function

after generating the new DNA, if the sum of the weight larger the maximum weight of the bag, randomly choose an index of the DNA sequence, if the element is true, then change it to false, loop until the sum of the weight smaller than the maximum weight of the bag.

```java
public void correctionDNA(DNA dna) {
    dna.calc();
    while (dna.weight > MyConstant.weightMax) {
        int index = MyRandom.randomInt(0, MyConstant.itemNum - 1);
        if (dna.sequence[index]) {
            dna.sequence[index] = false;
        }
        dna.calc();
    }
}
```

c) crossover function

pass the two parent DNAs as the parameter of the method

Randomly choose a cross point

To breed child 1: Combine the left side of the parent 1 and the right side of the parent 2

To breed child 2: Combine the right side of the parent 1 and the left side of the parent 2

Keep loop until the number of children reach to the fecundity of mating

If the fecundity is an odd number, breed one more child 1

```java
public DNA[] crossOver(DNA dna1, DNA dna2) {
    DNA[] result = new DNA[MyParameter.fecundity];
    for (int i = 0; i < MyParameter.fecundity - 1; i = i + 2) {
        result[i] = new DNA();
        result[i + 1] = new DNA();
        int random = MyRandom.randomInt(1, MyConstant.itemNum - 1);
        for (int j = 0; j < random; j++) {
            result[i].sequence[j] = dna1.sequence[j];
            result[i + 1].sequence[j] = dna2.sequence[j];
        }
        for (int j = random; j < MyConstant.itemNum; j++) {
            result[i].sequence[j] = dna2.sequence[j];
            result[i + 1].sequence[j] = dna1.sequence[j];
        }
    }
    if (MyParameter.fecundity % 2 == 1) {
        result[result.length - 1] = new DNA();
        int random = MyRandom.randomInt(1, MyConstant.itemNum - 1);
        for (int j = 0; j < random; j++) {
            result[result.length - 1].sequence[j] = dna1.sequence[j];
        }
        for (int j = random; j < MyConstant.itemNum; j++) {
            result[result.length - 1].sequence[j] = dna2.sequence[j];
        }
    }
    return result;
}
```

d) mutation function

select an index of the DNA sequence randomly, if this index point to an element which is true, then change it to false, and vise versa.

```java
public void mutation(DNA dna) {
    int index = MyRandom.randomInt(0, MyConstant.itemNum - 1);
    if (dna.sequence[index]) {
        dna.sequence[index] = false;
    } else {
        dna.sequence[index] = true;
    }
}
```

e) thread implementation

if both of the parent DNAs reach the number of generation to reproductive maturity and hit the probability of crossover, execute crossover method, otherwise copy the parent DNAs.

After crossover, if hit the probability of mutation, execute the mutation method.

As a result, the children were born

At this point, two threads will compete for the same lock, thread that acquires the lock is eligible to put its children into population

```java
@Override
public void run() {
    DNA[] result;
    if (dna1.generation >= MyParameter.maturity
            && dna2.generation >= MyParameter.maturity
            && Math.random() < MyParameter.crossOverPro) {
        result = crossOver(dna1, dna2);
    } else {
        result = new DNA[2];
        result[0] = copyDNA(dna1);
        result[1] = copyDNA(dna2);
    }
    for (int i = 0; i < result.length; i++) {

        if (Math.random() < MyParameter.mutationPro) {
            mutation(result[i]);
        }
        correctionDNA(result[i]);
        result[i].generation++;
    }
    synchronized (myLock) {
        for (int i = 0; i < result.length; i++) {
            if (myLock.index >= MyParameter.populationNum) {
                break;
            }
            DNAList[myLock.index] = result[i];
            myLock.index++;
        }
    }
}
```

f) parallel evolve function

create a new DNA list to store a new population

generate a fixed thread pool, which contains two threads

create an object lock to synchronize the index of the new DNA list

choose the best DNA from the old DNA list, and preserve it to the new population randomly select two DNAs from the survived DNAs with higher fitness

execute the thread and put it into threads pool

when the new population was fully filled, shut down the thread pool and wait it for termination

so far, we have finished generating the new population! :)

```
public DNA[] parallelEvolve(DNA[] oldList) {
    DNA[] newList = new DNA[MyParameter.populationNum];
    ExecutorService threadPool = Executors.newFixedThreadPool(MyConstant.coreNum);
    MyLock myLock = new MyLock();
    myLock.index = 1;
    threadPool.execute(() -> {
        newList[0] = copyDNA(oldList[0]);
        newList[0].generation++;
    });
    for (int i = 1; i < MyParameter.populationNum
            && myLock.index < MyParameter.populationNum; i++) {
        int[] twoNums = MyRandom.randomIntNums( start: 0,
                 end: (int) (MyParameter.populationNum * MyParameter.survivePro) - 1,  num: 2);
        threadPool.execute(new Evolve(oldList[twoNums[0]], oldList[twoNums[1]], newList, myLock));
    }
    threadPool.shutdown();
    try {
        threadPool.awaitTermination( timeout: 10, TimeUnit.SECONDS);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return newList;
}
```

g) geneticAlgorithm function

an evolution mechanism--this takes care of the seeding of generation 0, and the births and deaths between generation N and N+1

start with a random population (generation 0) and "breed" successive generations from that.

a logging function to keep track of the progress of the evolution, including the best candidate from the final generation;

this function operates like the main function, the flow chart above has demonstrated the logic clearly

```java
public int geneticAlgorithm() {
    File file = new File( pathname: "result.csv");
    if(!file.exists())
    {
        try {
            file.createNewFile();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    BufferedWriter bw = null;
    try {
        bw = new BufferedWriter(new FileWriter(file));
    } catch (IOException e) {
        e.printStackTrace();
    }
    DNA[] oldDNAList = initPopulation();
    select(oldDNAList);
    for (int i = 0; i < MyParameter.generationNum; i++) {
        DNA[] newDNAList = parallelEvolve(oldDNAList);
        select(newDNAList);
        oldDNAList = newDNAList;
        try {
            bw.write( str: oldDNAList[0].value+"");
            bw.newLine();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    try {
        bw.flush();
        bw.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return oldDNAList[0].value;
}
```

8. Result Analysis

8.1 chart

8.2 analysis

the result is very gratifying!

at the beginning of the generation, it evolves very fast

after about 1000 generations, the speed of the evolution slow down

but it still has a chance to evolve a better solution

the best solution of our algorithm is at the 9821th generation

9. Test

using JUnit to test the functions

```
@Test
public void testRandomInt() {
    for (int i = 0; i < 10000; i++) {
        int start = 1;
        int end = 10;
        int result = MyRandom.randomInt(start, end);
        assertTrue( condition: result >= start);
        assertTrue( condition: result <= end);
    }
}
```

```
@Test
public void testRandomIntNums() {
    for (int i = 0; i < 100; i++) {
        int random = MyRandom.randomInt(1, 10);
        int[] nums = MyRandom.randomIntNums( start: 1, random, random);
        for (int j = 0; j < nums.length; j++) {
            for (int k = j + 1; k < nums.length; k++) {
                assertTrue( condition: nums[j] != nums[k]);
            }
        }
    }
}
```

```
@Test
public void testInitDNA() {
    DNA dna = myAlgorithm.initDNA();
    assertTrue( condition: dna.sequence != null);
}
```

```
@Test
public void testCorrectionDNA() {
    for (int i = 0; i < MyParameter.populationNum; i++) {
        DNA dna = myAlgorithm.initDNA();
        myAlgorithm.correctionDNA(dna);
        assertTrue( condition: dna.weight <= MyConstant.weightMax);
    }
}
```

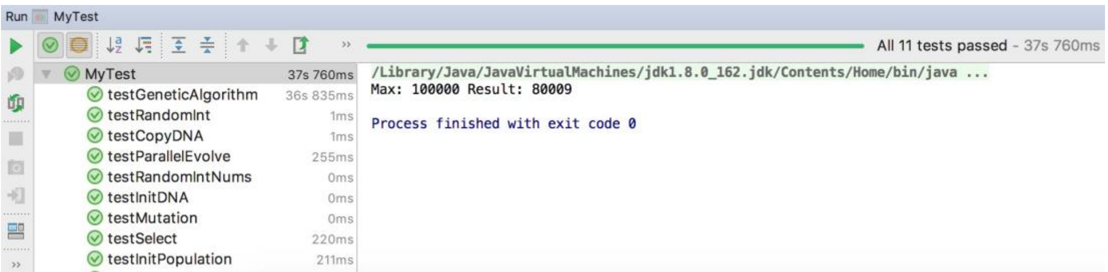
```java
@Test
public void testCopyDNA() {
    DNA oldDNA = myAlgorithm.initDNA();
    DNA newDNA = myAlgorithm.copyDNA(oldDNA);
    for (int i = 0; i < MyConstant.itemNum; i++) {
        assertEquals(oldDNA.sequence[i], newDNA.sequence[i]);
    }
    assertEquals(oldDNA.generation, newDNA.generation);
    assertEquals(oldDNA.weight, newDNA.weight);
    assertEquals(oldDNA.value, newDNA.value);
    newDNA.generation++;
    assertTrue( condition: oldDNA.generation != newDNA.generation);
}

@Test
public void testInitPopulation() {
    DNA[] DNAList = myAlgorithm.initPopulation();
    for (int i = 0; i < MyParameter.populationNum; i++) {
        assertTrue( condition: DNAList[i].weight <= MyConstant.weightMax);
    }
}


@Test
public void testSelect() {
    DNA[] DNAList = myAlgorithm.initPopulation();
    myAlgorithm.select(DNAList);
    for (int i = 0; i < MyParameter.populationNum - 1; i++) {
        assertTrue( condition: DNAList[i].value >= DNAList[i + 1].value);
    }
}


@Test
public void testCrossOver() {
    DNA dna1 = myAlgorithm.initDNA();
    DNA dna2 = myAlgorithm.initDNA();
    DNA[] result = myAlgorithm.crossOver(dna1, dna2);
    for (int k = 0; k < result.length - 1; k = k + 2) {
        for (int i = 0; i < MyConstant.itemNum; i++) {
            if (dna1.sequence[i] != result[k].sequence[i]) {
                for (int j = 0; j < i; j++) {
                    assertEquals(dna1.sequence[j], result[k].sequence[j]);
                    assertEquals(dna2.sequence[j], result[k + 1].sequence[j]);
                }
                for (int j = i; j < MyConstant.itemNum; j++) {
                    assertEquals(dna2.sequence[j], result[k].sequence[j]);
                    assertEquals(dna1.sequence[j], result[k + 1].sequence[j]);
                }
                break;
            }
        }
    }
    if (MyParameter.fecundity % 2 == 1) {
        for (int i = 0; i < MyConstant.itemNum; i++) {
            if (dna1.sequence[i] != result[result.length - 1].sequence[i]) {
                for (int j = 0; j < i; j++) {
                    assertEquals(dna1.sequence[j], result[result.length - 1].sequence[j]);
                }
                for (int j = i; j < MyConstant.itemNum; j++) {
                    assertEquals(dna2.sequence[j], result[result.length - 1].sequence[j]);
                }
                break;
            }
        }
    }
}
```

```java
@Test
public void testMutation() {
    DNA dna = myAlgorithm.initDNA();
    DNA oldDNA = myAlgorithm.copyDNA(dna);
    myAlgorithm.mutation(dna);
    int count = 0;
    for (int i = 0; i < MyConstant.itemNum; i++) {
        if (dna.sequence[i] != oldDNA.sequence[i]) {
            count++;
        }
    }
    assertEquals( expected: 1, count);
}
```

```java
@Test
public void testParallelEvolve() {
    DNA[] oldList = myAlgorithm.initPopulation();
    myAlgorithm.select(oldList);
    DNA[] newList = myAlgorithm.parallelEvolve(oldList);
    for (int i = 0; i < MyParameter.populationNum; i++) {
        assertTrue( condition: newList[i] != null);
    }
}
```

```java
@Test
public void testGeneticAlgorithm() {
    int result = myAlgorithm.geneticAlgorithm();
    assertTrue( condition: result <= MyConstant.weightMax * 10);
    System.out.println("Max: " + MyConstant.weightMax * 10 + " Result: " + result);
}
```

there are 11 test cases, and they all pass!



10. Reference

http://www.cs.cmu.edu/~02317/slides/lec_8.pdf

https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3

http://www.joinville.udesc.br/portal/professores/parpinelli/materiais/IntroductionGA.pdf

11. harvest

We collaborated with each other and finished the project.

During the project, we got more familiar with the concepts of genetic algorithm and used it to solve the problem.

We want to accumulate more experience in the future! :)