CSE104 Coursework 2

(Due date: 5pm 06/15/2020 China Time, online ICE submission ONLY)

Learning Outcomes

On successful completion of this assignment, students are expected to:

- understand and be able to apply a variety of data structures, together with their internal representation and algorithms;
- be able to make informed choices between alternative ways of implementation, justifying choices on grounds such as time and space complexity;
- be able to select, with justification, appropriate data structures to ensure efficient implementation of an algorithm.

Background information

This Coursework 2 is a continuation of Coursework 1, where you are required to further develop algorithms for the travelling salesman problem (TSP). In case that some of you didn't work out the first coursework, this coursework will be based on the solution of Coursework 1, which has been uploaded on the ICE page.

In this coursework, you are required to develop a more advanced algorithm for the TSP that is able to improve upon an existing TSP routine continuously. You will be guided step by step to complete this algorithm. Through this special journey, you will learn how more complicated algorithms work and will be given chances to write your own ones.

Current structure of the project

Open the project zip file for Coursework 2, it should contain three files:

1. City.java — Represents a city in TSP and stores its information.
2. TSPSolver — This is the file that we worked on in coursework 1. At the current stage, it contains several functions:
    a. readFile() — loads the text files of TSP problems.
    b. solveProblem() — generates an initial routine by continuously choosing the closest city to visit.
    c. printSolution() — prints a routine and returns the total distance of that routine.
    d. evaluateSolution() — calculates then returns the total distance of a routine, it is a simplified version of printSolution(). It will be needed in this coursework.
    e. swapCity(), evalSwap(), evalSwap_SLOW_() and swapFirstImprove() — they will be explained in the next section.
    f. improveRoutine() — the entry point of the "more complicated algorithm" that you need to implement.
3. Main.java —This file contains the necessary code to make use of TSPSolver.

The overall work flow in Main is described below:

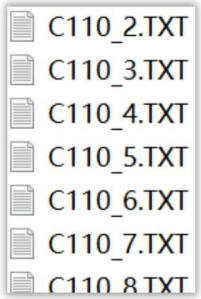
```
public static void main(String[] args) {
    ArrayList<City> cities = TSPSolver.readFile(args[0]);
    cities = TSPSolver.solveProblem(cities);
    System.out.printf("Distances: %f\n", TSPSolver.printSolution(cities));
    cities = TSPSolver.improveRoutine(cities);
    System.out.printf("After improving: %f\n", TSPSolver.printSolution(cities));
}
```

We first read in a text file using the file path provided in the first argument of our java program (which is args[0]). The list of cities read by the function is then passed to another function "solveProblem()" and a routine for this problem is generated. The routine is printed out to allow us to check how much distance has the salesman travelled. So far, these are the functions that already exist in the coursework 1.

The next function is quite important: improveRoutine(). It takes in the routine we generated previously and aims to further reduce its total distance travelled.

Background information

To be able to understand what we are going to do in this coursework, you need some background information about algorithms for TSP-like problems. You have already tested your algorithm against several sample files like below in the coursework 1:



Each file contains a long list of cities and you need to visit. These files are often referred to as **problem instances**. A valid routine that visits all cities of a problem instance exactly once is often called a **solution** of the problem instance. As you already know, one problem instance of TSP can have many different solutions. Let's look at two different solutions for a simple TSP:

Solution 1: 0->1->2->3->4->5->0
Total distance: 565

Solution 2: 0->1->3->4->2->5->0
Total distance: 555

The difference between these two solutions is that the city 2 is moved from the slot between 1 and 3 to the slot between 4 and 5. After changing the order of visiting these cities, the total distance is reduced by 10.

Based on this observation, we can infer that, after trying other different insertion slots in this routine, it is possible to find other better solutions that has less total distance. Calculating the total distance of a routine is very tiresome for human beings but is much easier for computers. That is an advantage that we must make full use of when designing algorithms. Apart from moving cities, there are other strategies like swapping cities in the routine, or reverse the visit sequence of a part of the routine etc..

Task 1: Finishing the first local search operator

There's a class of algorithms called local search (LS). LS explores many solutions of a problem instance by trying different possibilities of a single solution mutation strategy, such as moving cities, swapping cities etc.. Just talking about how an algorithm works is not intuitive, below shows one LS I have partially implemented, it is available in the TSPSolver.java:

```
public static void swapCity(ArrayList<City> routine, int index1, int index2) {
    // your implementation goes here
}

public static double evalSwap(ArrayList<City> routine, int index1, int index2) {
    double oldDistance = evaluateRoutine(routine);
    swapCity(routine, index1, index2);
    double newDistance = evaluateRoutine(routine);
    swapCity(routine, index1, index2);
    return oldDistance - newDistance;
}

public static boolean swapFirstImprove(ArrayList<City> routine) {
    for (int i = 1; i < routine.size() - 1; i++) {
        for (int j = i + 1; j < routine.size() - 1; j++) {
            double diff = evalSwap(routine, i, j);
            if (diff - 0.00001 > 0) { // I really mean diff > 0 here
                swapCity(routine, i, j);
                return true;
            }
        }
    }
    return false;
}
```

In the code example above, "swapCity()" should simply swap the two cities indicated by index1 and index2 in the routine and then exit. An example of a swap operation is shown below:

Routine: 0->A->B->C->D->E->0

After calling "swapCity(routine, 2, 5)", we will get:

Routine': 0->A->E->C->D->B->0

This function does not need to return anything as the changes made to the "ArrayList<City> routine" can be seen outside of this function anyway.

Requirement 1: Implement the "swapCity()" function. This is a warm-up task for you.

The second function "evalSwap()" is designed to evaluate the effect of swapping two cities. The parameters of this function is the same as "swapCity()". Its output is "oldDistance – newDistance", which indicates how much travelling distance will be reduced if such a swap operation is applied. A positive value means that the operation does reduce the total distance of the current routine, while a negative value means a worse solution is obtained. We will call "oldDistance – newDistance" **delta** from now on.

In its current implementation, "evalSwap()" actually applies the swap so that the new total distance can be obtained. Then it swaps these two cities back to their original position in the routine. The whole routine is checked twice in this process. This is very inefficient if the routine has many cities.

Requirement 2: Re-implement the "evalSwap()" function, so that it calculates the delta by only examining the modified parts of the routine.

The third function "swapFirstImprove()" is fully implemented. It iterates through all possibilities of city swap operations for this routine. If it finds a swap operation that leads to shorter total distance, it will apply the swap and return true. If, after trying all possibilities, there is no beneficial swap operation found, it will return false. If you observe the code carefully, you can see that both for-loops in this function start with index 1 instead of 0, and end at "routine.size() – 1". That's because the routine of TSP must start and end with city 0, changing the order of city 0 will break the routine. Make sure you avoid such mistakes when doing the Task 2.

Task 2: Creating the second local search operator

In this task, you are required to create another local search operator, just like the one in task 1.

```
/*
    Moves the city at index "from" to index "to" inside the routine
 */
private static void moveCity(ArrayList<City> routine, int from, int to) {
    // provide your code here.
}

public static double evalMove(ArrayList<City> routine, int from, int to) {
    // your implementation goes here
}

public static boolean moveFirstImprove(ArrayList<City> routine) {
    // your implementation goes here
}
```

Requirement 3: Implement the "moveCity()" function, which moves the city at index "from" to the position right before index "to". The index here refers to the index in the ArrayList. Here are three examples of move operations:

Routine: 0->A->B->C->D->E->0

After calling "moveCity(routine, 2, 5)", we will get:
Routine': 0->A->C->D->B->E->0

After calling "moveCity(routine, 2, 1)", we will get:
Routine': 0->B ->A ->C->D->E->0

After calling "moveCity(routine, 2, 3)", the solution stays the same. Because the city at index 3 is 'C', 'B' is put right before 'C'. Thus, the position of 'B' is unchanged.

Requirement 4: Implement the "evalMove()" function, which evaluates the effect of moving a city from one index to another. It returns the delta value of the total distances, as explained earlier. Designs that lead to less computational time yet still maintains the correctness will get higher marks.

Requirement 5: Implement the "moveFirstImprove()" function, which iterates through all possibilities of city move operations for this routine. If it finds a move operation that leads to shorter total distance, it will apply the move and return true. If, after trying all possibilities, there is no beneficial move operation found, it will return false.

Task 3: Putting them together!

Once you have finished Task 1 and Task 2. You are ready to tackle the final part of this project:

```
public static ArrayList<City> improveRoutine(ArrayList<City> routine) {
    // Can you improve this simple algorithm a bit?
    swapFirstImprove(routine);
    moveFirstImprove(routine);
    return routine;
}
```

The "improveRoutine()" function takes in, as its parameter, the routine generated by the algorithm of coursework 1 and returns an improved routine if possible. In its current state, this function just called the two "xxxFirstImprove()" functions once. You can certainly improve upon it.

Requirement 6: Re-implement the "improveRoutine()" function so that it can explore more solutions of TSP problem instances. Better solutions will result in higher marks for this part. You do not have to use xxxFirstImprove() functions.

Since in Task 3, you can make any decisions, it is impossible for me to investigate the code of everybody and point out what could be improved. Your algorithm will instead be compared with each other and whoever is ranked at the first will get the full mark for benchmarking (20%). You are allowed to add your own functions or inner classes in TSPSolver.java or City.java.

I will use bash scripts to read the output of your programs. To ease my marking, your program should not output anything when running. I will call the "printSolution()" function in my own version of the main function. Thus, make sure that you delete any calls to "System.out.print()" in TSPSolver and City classes, so that your diagnostic outputs do not interfere with my bash script (except for the "printSolution()" function). Failing to do so will disqualify yourself from the benchmarking and you will only get the base mark for Task 3 (up to 10%), given that the algorithm works correctly. That is, the only function that is allowed to call "System.out.print()" or other console output functions is the "printSolution()" function from Coursework 1.

About the benchmarking

This is the hardware that I will use for the benchmarking:

Dual Intel® Xeon® Processor E5-2680 v2 (25M Cache, 2.80 GHz) – 10 cores each, 20 cores in total
64 GB of RAM
A 64-bit Linux with OpenJDK 11

Once I have collected all of your submissions, I will compile your programs and run against the problem instances I have (Not just those files like C110_1.txt on ICE, but also other instances that you have not been given). Your program will be given 5 minutes to solve each instance using a single core of the aforementioned CPU. Once 5 minutes have reached, I will terminate your program no matter whether it is still running or has finished. I will use my own Main.java to call your "improveRoutine()" function exactly like below:

```
ArrayList<City> cities = TSPSolver.readFile(args[0]);
cities = TSPSolver.solveProblem(cities);
cities = TSPSolver.improveRoutine(cities);
```

Then, the routine is printed and the total distance calculated by my own function will be stored in a text file. (this is to prevent someone cheating on the results)

Once all total distances are collected, I will use the following method to calculate individual marks. Assume that your total distance is x, the shortest total distance found in CSE104 is L, and an upper bound U (decided by me):

$$\frac{U-x}{U-L} = \frac{Your\ Mark}{20}$$

Currently, I plan to use the total distance of "solveProblem()" as the upper bound U. But I reserve the right to change it.

Task 4: Alternative data structure discussion

This is the final task for you. Currently, ArrayLists are used to represent the routine of TSP. ArrayList uses array as its internal data structure. What will happen if the routines are

represented using LinkedList? Will the original code in task 3 run faster or slower? You need to provide your justifications in the report.

Coursework Submission

In this coursework, you will be given three template files: "Main.java", "City.java" and "TSPSolver.java". You should add code into these files according to this coursework specification. You are free to change Main.java in order to test your code. But Main.java will NOT be examined. I will simply replace it with my version when marking the coursework. All of your implementations should be placed into the City class and the TSPSolver class.

You should submit your Java program code files together with your report to the entry I provided on ICE. The submitted program solution should be well commented and include three files: "City.java", "TSPSolver.java" and a 4-page report with a file name "Report.pdf" or "Report.doc" (or docx).

The report should not exceed 4 pages, without counting your .java files submitted separately from this report, with the module title and your name/student number & signed declaration for non-plagiarism shown on the title page. Your report should explain your data structure(s), what data structures/algorithms (rendered in pseudo code) you have designed and how they are tested and used in your programs. You should also analyse the space complexity (i.e. memory costs) of your data structures and the complexity of your city-visit major algorithms in TSPSolver (i.e. cost of running time asymptotically).

Mark distribution

The mark distribution (100% total) is arranged as follows:

1. The task 1 worth up to 15%.
2. The task 2 worth up to 30%.
3. The task 3 worth up to 30%: 10% for attempting this task (with visible efforts) and 20% comes from the benchmark.
4. Your report worth up to 25%.

This assignment: 50% of the overall marks.