PA11 Traveling Salesperson

Due: Wednesday 4/27 by 11:30PM

Submission:

- DGraph.java - Class representing a directed graph.
- PA11Main.java - Class to perform different algorithms to solve the traveling salesman problem.
- README.md - Results of timing experiments that compare your improved algorithm to the required algorithms for the input file big11.mtx and describes why the algorithms have relative performance differences. Note we have already provided this blank file in your eclipse project. You just need to edit it.

Overview

This assignment has multiple goals:

- To practice new algorithmic patterns: recursive backtracking and heuristic
- To work with a new data structure: graphs
- Decomposition: using multiple interacting classes to achieve a larger goal
- Continue the discussion of performance analysis that we have started this semester

The Traveling Salesperson problem asks what is the shortest trip through a sequence of locations and back to the beginning while only visiting each location once. The distances between locations are the input to the problem. These distances can be represented as weights on edges in a graph.

The Traveling Salesperson problem is a famous problem in computer science. It is an example of an NP-Complete problem. There are a significant number of NP-Complete problems that can all be converted to each other (see here). Some examples: vehicle routing, circuit design, and robot navigation.

Assignment

You will be solving the traveling salesperson problem using three different ways: recursive backtracking, heuristic, and your own approach.

To use the program, someone will need to put two command line arguments into your program:

```
PathTo/infile.mtx [HEURISTIC, BACKTRACK, MINE, TIME]
```

The input files are in the same format as the matrix market format used by the SuiteSparse Matrix Collection. More importantly, see this link for a description of the .mtx format. You **can** write code similar to this link to read in the .mtx format. Here is an example input file:

```
%%MatrixMarket matrix coordinate real general
3 3 6
1 2 1.0
2 1 2.0
1 3 3.0
3 1 4.0
2 3 5.0
3 2 6.0
```

Example output to standard out is provided in the PublicTestCases/ for HEURISTIC and BACKTRACK. Here is the output for the above input file and the HEURISTIC command:

```
cost = 10.0, visitOrder = [1, 2, 3]
```

You will be running your own algorithm with the MINE command. The output for time is shown later in this spec.

This assignment is broken down into parts. You could use iterative development to tackle each part individually: finish reading in the .mtx file into a graph data structure, the heuristic part of the assignment, and the recursive backtracking, mine, and time commands.

There is pseudocode in this spec. As with the last assignment, it is **crucial** that you understand how the pseudocode works, what it is doing, and why it is doing what it is doing before actually writing the code for it, or else you will be lost. Not to mention it will be very difficult to improve upon an algorithm that you do not understand well.

Part One: DGraph

Before you can run algorithms on a graph, you need a graph. The first part of this assignment will be reading in a .mtx file and converting the file into a code representation of a graph. You will need to fill in certain methods and fields in the DGraph.java file. This file exports a class called DGraph. This class will store all necessary information about the graph. See the DGraph.java file in the starter code for a list of methods to fill in. **You should extensively test this class before moving on.** Lingering bugs in the DGraph class will make debugging the later part of the assignment nearly impossible.

With your ~~DGraph class working~~, it is time to write code to read in a .mtx file and store it properly into your DGraph class. This should be done in PA11Main.java. File format descriptions are listed above in the 'Assignment' section. Additionally, we have provided testcases and their corresponding .dot files. These files will be a great testing tool. There are many graph visualizers on the web that accept a .dot file and show you a visualization of the graph. For instance, check out the example.mtx file and then use an online .dot visualizer to view the example.mtx.dot file as a visual graph to understand this file format. The starter code we gave you has methods that will build the dot file as a string for you. I just used this site which happened to be the first google result for me.

~~Part Two: Heuristic~~

Below is the pseudocode we expect you to follow for the HEURISTIC command. If you have ideas on how to improve it, great, save those for the MINE command. Note now would be a good time to review Trip.java which was given as part of the starter code.

```
create a trip
choose city 1 first, call it the current city

for k=2 to numNodes inclusive
    for each neighbor of the current city
        find the neighbor that is available AND the closest to the current city
    choose the closest city that is available for the trip
    call that closest city the current city
```

Part Three: Recursive Backtracking

We will make the decisions starting with node 1 and continuing through the nodes in order. Everytime a node is chosen, that choice will be checked before recursing to do some pruning. We cannot stop at the first node we find, because it is possible that paths through other nodes will cost less.

```
create a trip
choose city 1 first
call backtrackingFunction on the trip

backtrackingFunction( graph data structure, current trip so far,
                                                min trip previously found )

    if all nodes are in trip then
        process the current trip:
            does it have less cost than min trip?
            if so then modify min trip previously found (hint: copyOtherIntoSelf())
        return

    if trip so far has less cost than the min trip previously found
        for each city x of the cities left
            choose x next
            backtrackingFunction( graph data structure, updated trip,
                                    min trip previously found )
            unchoose x
```

Part Four: Your own approach

You will also implement the command MINE that executes your own faster algorithm for performing the traveling salesperson problem. The code you submit should be able to execute the MINE command. For your own approach you can choose to do one of the following:

- improve upon the heuristic approach while not resorting to a trivial solution like just listing all of the nodes in order
- improve upon the recursive backtracking approach by putting in more pruning and then show this is faster than the suggested recursive backtracking approach

Part Five: Timing all of the approaches

Using code similar to the following, you will have a TIME command that times all of the algorithms.

```
long startTime = System.nanoTime();
trip = heuristic(graph);
long endTime = System.nanoTime();
long duration = (endTime - startTime) / 1000000;
System.out.println("heuristic: cost = " + trip.tripCost(graph) + ", " + duration +
                                            " milliseconds");
```

The output for the TIME command will look as follows:

```
heuristic: cost = 935.3299999999999, 0 milliseconds
mine: cost = 935.3299999999999, 0 milliseconds
backtrack: cost = 835.8799999999999, 5 milliseconds
```

None of the grading test cases call TIME, because each time you run the program even on the same input and on the same machine there will be some time variance. Because of that you will probably want to run the timings about 5 times before drawing any conclusions.

A good test is to run the TIME command on your machine for big11.mtx. It takes around a minute for the recursive backtracking algorithm on a 3 year old Mac.

Error Handling

All of the inputs will be correctly formed for this assignment.

Hints

- The node IDs go from 1 to the number of nodes inclusive because the sparse matrix market file format (.mtx) stores node IDs starting at 1 instead of 0.
- We are providing a Trip class that keeps track of a Trip. Read its usage instructions.
- Start early!
- Test the various pieces of your program very well. If you do not ensure that each piece works perfectly, you will find very difficult to detect issues with the whole. Use the debugger and/or various print statements.

Grading Criteria

We are providing testcases for this PA. We will also have our own private grading testcases.

We encourage you to write your own JUnit testcases to ensure your classes work properly, but we will not be collecting or grading these test files.

Your grade will consist of similar style and code clarity points we have looked for in earlier assignments.

Write your own code. We will be using a tool that finds overly similar code. Do not look at other students' code. Do not let other students look at your code or talk in detail about how to solve this programming project. Do not use online resources that have solved the same or similar problems. It is okay to look up, ”How do I do X in Java”, where X is indexing into an array, splitting a string, or something like that. It is **not** okay to look up, ”How do I solve {a programming assignment from CSc210}” and copy someone else's hard work in coming up with a working algorithm.