Prof. Dionne Aleman
dionne.aleman@utoronto.ca

Project 5: Traveling salesman problem v3

Contents

New concepts: Inheritance

1 Overview

This project builds upon the previous project, where a program was built to load and store many graphs and test solutions to the traveling salesman problem (TSP) using a simple nearest-neighbor heuristic. TSP is defined as follows. Say we have a set of cities that a traveling salesman needs to visit, and it takes a certain amount of time to travel between each city; what is the cheapest route to visit all cities exactly once?

The remainder of this section provides an overview of TSP from the previous project.

1.1 TSP as a graph

TSP is represented using a graph, where nodes are cities and undirected arcs between them indicate if it is possible to travel between two cities directly, and if so, the cost (distance). Each city (node) will contain the following information:

- Name
- latitude $\in [-90, 90]$
- longitude $\in [-180, 180]$

The cost to travel from node i to node j is the distance between the two cities' coordinates, calculated using the Haversine formula:

$$\begin{aligned} a &= \sin^2(\Delta x/2) + \cos(x_i) \times (x_j) \times \sin^2(\Delta y/2) \\ b &= 2 \times \text{atan2}\left(\sqrt{a}, \sqrt{1-a}\right) \\ c_{ij} &= R \times b \end{aligned} \qquad \text{(Haversine Formula)}$$

where

- x_i and x_j are the latitudes of nodes i and j, respectively, in radians
- Δx is the latitude difference in radians
- Δy is the longitude difference in radians
- $R = 6371$, the diameter of Earth in km
- atan2 is the `Math.atan2()` function

Costs are not input by the user; instead, they are calculated automatically by the program. Costs c_{ij} are stored in **cost matrix C**, where $c_{ij} = 0$ if nodes i and j are not connected.

An **adjacency matrix** is a common method to store arcs in a graph. An adjacency matrix **A** has size $n \times n$ (where n is the number of nodes), and each element a_{ij} is 1 if nodes i and j are connected, and 0 otherwise. Similarly, **C** is the same as the adjacency matrix, but with costs c_{ij} instead of ones. Note that because arcs are undirected, **A** and **C** will always be symmetric.

1.2 TSP solution paths

A TSP solution path (formally called a Hamiltonian cycle) is a sequence of $n + 1$ nodes, satisfying the following requirements:

1. Start and end cities match
2. Sequential cities are connected by an arc
3. Cities (other than the home city, the first city in the path) are visited exactly once
4. All cities are visited

1.3 Loading graphs

Graphs are read from text files, formatted as described in previous projects. Files may contain multiple graphs, and multiple files may be loaded by the user.

2 Program description

In this project, you will compare the performance of three different variations of the nearest neighbor heuristic to find TSP solutions:

- Nearest neighbor (NN), exactly the heuristic you previously wrote
- Nearest neighbor first-last (NN-FL)
- Node insertion (NI)

Your program will allow the user to load graphs, run all of the heuristics on all of the graphs, and then compare the performance of the heuristics to try to determine which algorithm performs best. Algorithm performance will be compared using three metrics:

1. Average cost of solution paths found

2. Average computation time
3. Success rate

The performance comparison will not only be useful to indicate which algorithm you might want to use if you have to solve TSP in the future, but it will also illustrate how sensitive greedy heuristics can be, since the only difference between the three algorithms is where the nearest neighbor is inserted into the TSP path.

3 Algorithm descriptions

As previously stated, each of the three algorithms is a variation of the same basic idea: Add the nearest neighbor to the existing path until the path is complete. After the last node is selected, the starting node is added to the end of the path. If at any time there is no valid node to add, or if there is no arc between the last node and the starting node, then the heuristic failed to find a TSP path.

Since the only difference between the algorithms is where the nearest neighbor is put into the path, it makes sense to create a single basic algorithm class, and then have each algorithm inherit from that algorithm class, only changing the next node selection and insertion behavior.

3.1 Nearest neighbor (NN) heuristic

NN is a greedy algorithm that constructs a path by starting at node 1 (the home city) and adding the nearest neighbor to the last node in the path.

3.2 Nearest neighbor first-last (NN-FL) heuristic

NN-FL builds on NN by adding cheapest nearest neighbors to the beginning *or* end of the path. In each iteration, NN-FL checks the first node's nearest neighbor (f) and the last node's nearest neighbor (ℓ), and adds whichever is cheaper. If arc $(1, f)$ has a cheaper cost than arc (n, ℓ), then node f is added to the front of the path. Otherwise, node ℓ is added to the end of the path.

3.3 Node insertion (NI) heuristic

NI builds on NN-FL by adding the cheapest nearest neighbor to *any* node in the path, not just the first or last node. There are three possible scenarios:

- The first node has the cheapest nearest neighbor: Add the new node to the front of the path.
- The last node has the cheapest nearest neighbor: Add the new node to the end of the path.
- An internal node i has the cheapest nearest neighbor: Insert the new node after node i in the path, but only if an arc exists between the new node and the node that comes after node i.

Note that if the new node already exists in the path, it cannot be inserted into the path.

4 Menu options

- `L - Load graphs from file`
  Read in graphs from a user-specified file.
- `I - Display graph info`
  Display summary info for each graph, and allow user to select graphs to see detailed info.
- `C - Clear all graphs`
  Clear all graphs.
- `R - Run nearest neighbor algorithm`
  Run the nearest neighbor algorithm on all loaded graphs.

- `D - Display algorithm performance`
  Display algorithm performance for each graph, and provide a statistical summary.
- `X - Compare average algorithm performance`
  Compare the average performance of each algorithm variation on the three metrics (cost, computation time, success rate) and identify the best performer in each category and the overall best performer. Ties in best performance are broken in this order: NN, NN-FL, NI.
- `Q - Quit`
  Quit the program.

5 Error messages

- `ERROR: Invalid menu choice!`
  when the user enters an invalid menu choice.
- `ERROR: No graphs have been loaded!`
  when the user attempts to do anything with the graphs before graphs have been loaded.
- `ERROR: Results do not exist for all algorithms!`
  when the user attempts to display algorithm performance when all algorithms have not been run.
- `ERROR: File not found!`
  when the user enters a file name that is not found.
- `ERROR: <name> did not find a TSP route for Graph <i>!`
  when the algorithm `<name>` fails to find a TSP route for Graph i.
- `ERROR: Input must be an integer in [<LB>, <UB>]!`
  where `LB` and `UB` are lower and upper bounds. If UB is equal to the maximum storable value of an `int`, then "`infinity`" is displayed instead of the actual UB value. Similarly, if LB is equal to the negative of the maximum storable value of an `int`, then "`-infinity`" is displayed instead of the actual LB value.
- `ERROR: Input must be a real number in [<LB>, <UB>]!`
  where `LB` and `UB` are lower and upper bounds. If UB is equal to the maximum storable value of a `double`, then "`infinity`" is displayed instead of the actual UB value. Similarly, if LB is equal to the negative of the maximum storable value of a `double`, then "`-infinity`" is displayed instead of the actual LB value.

6 Required elements

Your project must use the each of the elements described in this section. You can have more functions, variables, and objects if you want, but you must at least use the elements described in this section.

Failure to correctly implement any of the elements in this section may result in a zero on the project.

6.1 Variables

- A single `NNSolver` object, as defined in Section 6.2
- A single `NNFLSolver` object, as defined in Section 6.2
- A single `NISolver` object, as defined in Section 6.2
- A single `ArrayList` or array of `Graph` objects, as defined in Section 6.2

- Global `public static BufferedReader` as **the only** `BufferedReader` object in the entire program for reading user input
  - Although your programs will compile and run just fine on your own computer with multiple `BufferedReader` objects, they will not run correctly on a web server with multiple `BufferedReader` objects. Since your programs will be graded on a web server, please don't have more than one `BufferedReader` for reading input from the console.

6.2 Classes

You are required to write the following classes:

1. `Node` class to store node data
2. `Graph` class to store graph data
3. `TSPSolver` class that provides the TSP algorithm template (basically the same as the class used in the previous project)
4. `NNSolver` class that inherits from the `TSPSolver` class and runs the NN variation of the algorithm described in Section 3.1
5. `NNFLSolver` class that inherits from the `TSPSolver` class and runs the NN-FL variation of the algorithm described in Section 3.2
6. `NISolver` class that inherits from the `TSPSolver` class and runs the NI variation of the algorithm described in Section 3.3

You can write more classes, and you can add to these classes, but you must at least implement the classes defined in this section.


```java
public class Node {
  private String name;   // node name
  private double lat;    // latitude coordinate
  private double lon;    // longitude coordinate

  // constructors
  public Node()
  public Node(String name, double lat, double lon)

  // setters
  public void setName(String name)
  public void setLat(double lat)
  public void setLon(double lon)

  // getters
  public String getName()
  public double getLat()
  public double getLon()

  public void userEdit() // get user info and edit node
  public void print() // print node info as a table row
  public static double distance(Node i, Node j) // calc distance btwn two nodes
}
```

Node_Template.java


```java
public class Graph {
  private int n;                    // number of nodes
  private int m;                    // number of arcs
  private ArrayList<Node> node;     // ArrayList *or* array of nodes
  private boolean[][] A;            // adjacency matrix
  private double[][] C;             // cost matrix

  // constructors
  public Graph()
  public Graph(int n)

  // setters
  public void setN(int n)
  public void setM(int m)
  public void setArc(int i, int j, boolean b)
  public void setCost(int i, int j, double c)

  // getters
  public int getN()
  public int getM()
  public boolean getArc(int i, int j)
  public double getCost(int i, int j)
  public Node getNode(int i)

  public void init(int n) // initialize values and arrays
  public void reset() // reset the graph
  public boolean existsArc(int i, int j) // check if arc exists
  public boolean existsNode(Node t) // check if node exists
  public boolean addArc(int i, int j) // add an arc, return T/F success
  public void removeArc(int k) // remove an arc
  public boolean addNode(Node t) // add a node

  public void print() // print all graph info
  public void printNodes() // print node list
  public void printArcs() // print arc list

  public boolean checkPath(int[] P) // check feasibility of path P
  public double pathCost(int[] P) // calculate cost of path P
}
```

Graph_Template.java

```java
public class TSPSolver {
  private ArrayList<int[]> solnPath; // ArrayList *or* array of solution paths
  private double[] solnCost;         // ArrayList *or* array of solution costs
  private double[] compTime;         // ArrayList *or* array of computation times
  private boolean[] solnFound;       // ArrayList *or* array of T/F solns found
  private boolean resultsExist;      // whether or not results exist
  private String name;               // name of this solver

  // constructors
  public TSPSolver() { }
  public TSPSolver(ArrayList<Graph> G) { }

  // getters
  public int[] getSolnPath(int i)
  public double getSolnCost(int i)
  public double getCompTime(int i)
  public boolean getSolnFound(int i)
```



```
  public boolean hasResults()
  public String getName()

  // setters
  public void setSolnPath(int i, int[] solnPath)
  public void setSolnCost(int i, double solnCost)
  public void setCompTime(int i, double compTime)
  public void setSolnFound(int i, boolean solnFound)
  public void setHasResults(boolean b)
  public void setName(String name)

  public void init(ArrayList<Graph> G) // initialize variables and arrays
  public void reset() // reset variables and arrays

  public void run(ArrayList<Graph> G, int i, boolean suppressOutput) // run
      nearest neighbor on Graph i
  public int[] nextNode(Graph G, ArrayList<Integer> visited) // find next node
      (return 2D array with [0]=next node, [1]=insertion position)
  public int nearestNeighbor(Graph G, ArrayList<Integer> visited, int k) // find
      node k's nearest unvisited neighbor
  public double successRate() // calculate success rate
  public double avgCost() // calculate average cost of successful routes
  public double avgTime() // calculate average comp time of successful routes
  public void printSingleResult(int i, boolean rowOnly) // print results for a
      single graph
  public void printAll() // print results for all graphs
  public void printStats() // print statistics
}
```

TSPSolver_Template.java

```
public class NNSolver extends TSPSolver {
  public NNSolver()

  @Override
  public int[] nextNode(Graph G, ArrayList<Integer> visited)
}
```

NNSolver_Template.java

```
public class NNFLSolver extends TSPSolver {
  public NNFLSolver()

  @Override
  public int[] nextNode(Graph G, ArrayList<Integer> visited)
}
```

NNFLSolver_Template.java

```
public class NISolver extends TSPSolver {
  public NISolver()

  @Override
  public int[] nextNode(Graph G, ArrayList<Integer> visited)

  // check if node k can be inserted at position i
  public boolean canBeInserted(Graph G, ArrayList<Integer> visited, int i, int k)
}
```

NISolver_Template.java

6.3 Functions

Function prototypes and short descriptions are provided below. It is your job to determine exactly what should happen inside each function, though it should be clear from the function and argument names and function descriptions. You can write more functions (and are strongly recommended to), but you must at least implement the functions defined in this section.

- `public static void displayMenu()`
  Display the menu.
- `public static boolean loadFile(ArrayList<Graph> G)`
  Read in graphs from a user-specified file.
- `public static void displayGraphs(ArrayList<Graph> G)`
  Display summary info for each graph, and allow user to select graphs to see detailed info.
- `public static void resetAll(NNSolver NN, NNFLSolver FL, NISolver NI)`
  Reset all solvers.
- `public static void runAll(ArrayList<Graph> G, NNSolver NN, NNFLSolver FL, NISolver NI)`
  Run all solvers on all graphs.
- `public static void printAll(NNSolver NN, NNFLSolver FL, NISolver NI)`
  Print the detailed results and statistics summaries for all algorithms.
- `public static void compare(NNSolver NN, NNFLSolver FL, NISolver NI)`
  Compare the performance of the algorithms and pick winners.
- `public static int getInteger(String prompt, int LB, int UB)`
  Get an integer in the range [LB, UB] from the user. Prompt the user repeatedly until a valid value is entered.
- `public static double getDouble(String prompt, double LB, double UB)`
  Get a real number in the range [LB, UB] from the user. Prompt the user repeatedly until a valid value is entered.

7 Helpful hints

- All the hints from the previous project are still useful.

