The University of West Florida
Department of Computer Science
Data Structures and Algorithms II
John W. Coffey

Background
A very famous graph/permutation problem is the Traveling Salesman Problem: given n cities with different distances among them and a designated first city, what is the least cost to visit all cities and return to the first one? Unfortunately, the brute-force method of solving this problem requires that we assess (n - 1)! permutations of the cities. How feasible is this?

However, not all is lost. For computationally "hard" problems such as this one, we may employ any of several means to find good if not strictly optimal solutions - say within 5-10% of optimal. (Note that "computationally hard" problems are formalized as "np-complete" or "np-hard."). One way to address this problem might be based upon a simplified approach to the permutation form of genetic algorithms. As always, Wikipedia has some good information on genetic algorithms.

Here is a basic approach (based upon an algorithm presented at Wikipedia) to solve our problem:

1. Choose the initial population of tours (generate a set of permutations of the orderings of the cities)
2. Evaluate the fitness of each individual tour
3. Repeat
   Select best-ranking individuals to reproduce or as elites (see below)
   Create a new generation through crossover, mutation or both (genetic operations) and give birth to offspring
   Evaluate the individual fitness of each offspring
   Replace worst ranked part of the current population with new permutations
   Until termination criteria are met

Subsequent Generations *might* be made up of:

Mutations:
Make a minor change (for instance, randomly swap two, three, or four cities) to make a slightly different version of a good order.

Crossovers:
Splice the end of one ordering to the end of another. This is not simple to realize with permutation problems (why?)

Elites
Do not cross over or mutate the few best individuals, but rather, keep them as-is.
Or, make copies of them and mutate the copies.

Termination of the Algorithm:
You will repeat the process of creating a generation of tours, checking for the best, determining elites and creating mutations, until the program reaches a terminating condition. Typical ways to terminate include:

- A solution is found that satisfies minimum criteria (under some maximum cost)
- A fixed number of generations is reached
- More iterations do not produce better results

We will use the fixed number of generations approach.

Program Description
Your task in this assignment will be to write *two solutions* to the TSP problem, both of which will run in one program:

- a brute-force permutation program that systematically tries all possible permutations, ultimately computing the optimal answer.
- a genetic algorithm-inspired solution.

Program Inputs

The program will utilize an input file named `cityWeights.txt` which contains the weights of the edges between cities. Your program will read `cityWeights.txt`. For simplicity, we will deal with a problem in which all cities connect to all other - in other words, with a complete graph. We will set up a single graph that can be used for 20 cities, more than we will need for this project.

`cityWeights.txt` will contain the appropriate number of distances between cities (380), one per line. Read them as doubles.

When your program runs, it will interactively get the following information from the user:

- the number of cities to run
- the number of individual tours are in a given generation
- how many generations to run
- what percentage of a generation should be comprised of mutations

Your program will always keep *exactly two* elites. The program reads the `cityWeights.txt` file, gets the input from the user and then the two algorithms will then grind away until output is produced.

Getting tours to try.

Project3TSPGraphic

You can generate permutations using a recursive algorithm as follows:

```
void perm(int n)
{
  int m, k, p , q, i;
  printS();
  for(i = 1; i < nfact; i++)
  {
    m = n - 2;
    while(s[m]>s[m+1])
```

```
        m = m - 1;
      k = n-1;
      while(s[m] > s[k])
        k = k - 1;
      swap(m,k);
      p = m + 1;
      q = n-1;
      while( p < q)
      {
        swap(p,q);
        p++;
        q--;
      }
      printS();
    }
  }
```

Experiments:
You will experiment with various values for these parameters to see which provides the best results for a given number of generations. You will incorporate timer capabilities into the programs so that you know how long runs with various parameters took. You will compare performance of the two approaches for different numbers of cities. Start at 10 cities and go up by 1 from there until you see a runtime of greater than 5 minutes on the brute force solution. You will not have to go far to get there. Create a table that compares the time results from the approximation compared to the brute force from 10 cities up until the brute force method requires more than 5 minutes. Show what percentage of the optimal solution (eg: 120% of optimal) your approximation solution provides at each run.

Output:
When the progam runs, it will produce:

- The number of cities run
- optimal cost from brute force
- time the brute force algorithm took
- cost from the ga
- time the ga took to run
- percent of optimal (eg: 120%) that the ga produced

Deliverables
You will submit the following for this project:
1. The functional decomposition for your program.
2. A User's manual that describes how to set up and run your program.
3. Your source code in C.
4. Results table disk file: The comparison of results between the approximation program and the brute force implementation (for 10, 11, 12, etc cities). This table is a summary of the data for all the individual outputs from 10 cities ...
5. A makefile for the project

Submission Requirements:

Note: you will lose 10% if you do not follow these instructions *exactly*. The reason for this policy is that eLearning puts zip files inside zip files and it is a time-consuming, tedious, and error-prone policy to unzip them all one-by-one. I have a utility I wrote that nicely unzips all these into a usable folder structure. If you do not follow these instructions, your files will get all mashed up together with others, often with name collisions and it is an unusable mess.

1. Compile and run your program one last time before submitting it. Your program must run with gcc in the Linux lab.
2. Place every file in your submission in a SINGLE DIRECTORY named <last name>-<first initial>-p<number>. For instance, I would create directory:
   coffey-j-p3 for project 3.
3. zip that FOLDER into a .zip file with the SAME NAME. This means that inside your zip file, you will have exactly one folder (from the example: coffey-j-p3) showing at the top level. Inside that folder will be ALL the files in your project.
4. *DO NOT* make separate folders for documentation and source files (or anything else) inside the main folder. Having such a setup simply necessitates more navigation time to get where we need to go to grade.
5. Any needed input files should be in the top-level folder along with the source code.
6. MacOS users - remove the _MacOS_ utility folder before you zip up the file. If you cannot, delete it from the archive once the archive is created. It just takes up space and is not needed for anything we do with your submissions.
7. Login to UWF's eLearning system at http://elearning.uwf.edu/. Select our course.
9. Select the Dropbox option. Then select the appropriate project folder.
10. Upload your zip file to that folder. Check the dropbox to insure that the file was uploaded. ALWAYS give yourself enough time. If you are trying to submit at 11:57pm on your machine, the clock might be off and the dropbox might already be closed.

Please review the policy on ACADEMIC MISCONDUCT. This is an *individual assignment.*