CS 242 Project 2 Spring 2014

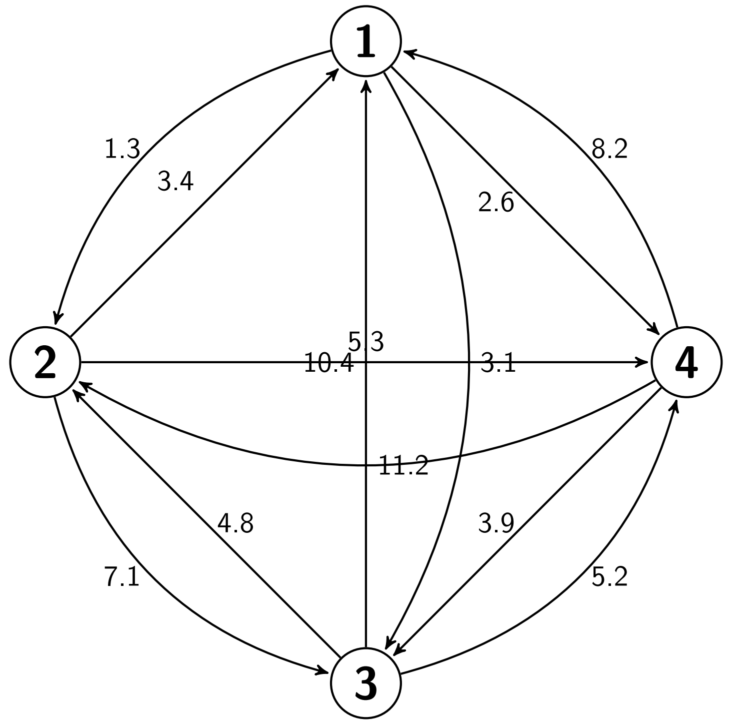
Due: April 14, 2014

The goal of this programming project is to explore some problems that involves generating all the permutations of a set.

Shown below is a weighted, directed graph. Think of each circle (known as node or vertex) as representing a city and a directed arrow (known as directed edge or arc) from one city to another as a highway. The number associated with the edge represents the cost of traveling. (It could be the air-fare or cost of gas or toll or time taken or any other cost.)

A Traveling Salesperson Problem (TSP) deals with a traveling salesperson who visits all the cities exactly once and returns to the starting city and the goal of the problem is to minimize the total cost of the tour. A TSP tour is a sequence of nodes that goes through all the vertices of the graph (exactly once) and returns to the starting vertex. For example, [2, 4, 1, 3, 2] is a TSP tour. Each tour is associated with a permutation of the cities, by omitting the last node (which is the repeat of the first vertex). Cost of a tour is defined as the sum of the weights on the edges included in the tour. Thus, the cost of tour [2, 4, 1, 3] is 21.4.

1. Enumerate all the TSP tours in the graph shown in the figure below that starts and ends at vertex 1. For each tour, calculate the cost. What is the optimal tour in this graph?



2. Let the weights of edges of a graph be represented as a matrix known as *adjacency matrix.* Assume that the vertices are labeled 0, 1, ..., $n-1$ for some n. The edge weights on the graph will be defined by a matrix M as follows: $M[i][j]$ = the weight of the edge from i to j if such an edge exists, otherwise this number will be -1. We will assume that the

only missing edges are the self-loops. For example, the adjacency matrix of the graph shown in the figure below is:

$$\begin{bmatrix} -1 & 1.3 & 5.3 & 2.6 \\ 3.4 & -1 & 7.1 & 3.1 \\ 5.3 & 4.8 & -1 & 5.2 \\ 8.2 & 11.2 & 3.9 & -1 \end{bmatrix}$$

Write a function *FindCost* that takes as input a permutation and calculates the cost of the tour. Thus if the input is the permutation [2 4 1 3], the output will be 21.4

3. Write a program *PermGen* that takes as input a positive integer n, and outputs a two-dimensional array of order $m \times n$ where $m = n!$ such that each row of the matrix represents a permutation of $[01...n-1]$. You are to use the following recursive algorithm to solve this problem: If $n = 1$, there is only one permutation, and so the output will be simply [0]. For $n > 1$, we will use the function to recursively generate all the permutations of $[012...n-2]$. Make n copies of this matrix, and in the first copy, create a new column containing $n-1$, and add this column as the first column. In the second copy, create the new column containing $n-1$ and add the column it as the second column and so on. Then, stack the matrices into a single matrix. Python code implementing this algorithm (named *perm*) is shown below, with some statements omitted. (Also omitted is a function *insert* that is used by perm.)

```
def perm(n):
    if n == 1:
        return [[1]]
    lst = ________________     (recursive call to perm with input n-1
    flst =  ______________      (empty list)
    for i in range(1, n+1):
        for j in lst:
            next = insert(n,i,j)
            flst = flst + [next]
    return flst
```

4. Write a function *OptTSPTour* that takes as input the adjacency matrix of a graph G and outputs the minimal TSP tour cost as well as the permutation that has the minimal cost. (Use the previous two functions. Generate all the permutations, then consider permutations one by one and keep track of the best solution found so far.)

5. The solution described above generates all the permutations before computing the cost associated with each. When n, the number of vertices, in the graph is large, your computer may not have memory to store all the permutations. Here we consider an approach that generates permutations one at a time, and evaluate its cost, and moves to the next one. This requires an algorithm that generates permutations one at a time based only on the previous permutation. One such algorithm is called *Steinhaus-Johnson-Trotter algorithm*, presented below as pseudo-code:

ALGORITHM 1: *Permutation List*

```
begin
    for i ← 1 to n + 1 do
        $\pi[i] \leftarrow i$
        $\pi^{-1}[i] \leftarrow i$
        $d[i] \leftarrow -1$
    $\pi[0] \leftarrow n + 1$
    $A \leftarrow \{2, \dots, n\}$

      Done ← false
      while not Done do
          Print($\pi$)
          if $A \neq \varnothing$ then
              $m \leftarrow \max\{i : i \in A\}$
              $j \leftarrow \pi^{-1}[m]$
              $\pi[j] \leftarrow \pi[j + d[m]]$
              $\pi[j + d[m]] \leftarrow m$
              $\pi^{-1}[m] \leftarrow \pi^{-1}[m] + d[m]$
              $\pi^{-1}[\pi[j]] \leftarrow j$
              if $m < \pi[j + 2 \cdot d[m]]$ then
                  $d[m] \leftarrow -d[m]$
                  $A \leftarrow A - \{m\}$
              $A \leftarrow A \cup \{m + 1, \dots, n\}$
          else
              Done ← true
  end.
```

Write a function OptTSPTour2 that generates permutations one at a time, determine its cost and keep track of the least cost tour and finally output the optimal tour.

6. (extra-credit) A major problem with the approach hinted at in the above exercises is that it takes enormous amount of time as well as storage to solve TSP problem even on graphs with only 10 cities. This exercise presents a significant improvement to the previous solution. The idea is as follows: Suppose V is the set of all vertices of a graph G, and $S \subset V$ is a subset of vertices. Let $v \in V\ S$, i.e., it is a vertex in V but not in S. Define the optimal path for the pair $< S, v >$ as the *minimal cost* path that starts in some vertex $u \in S$, and goes through all vertices in S exactly once and then finally ends in v. For example, ... The new algorithm computes the optimal path for $< S, v >$ for every S and for every $v \in V\ S$. From this, the optimal TSP tour in G can be computed. We will discuss this algorithm in class.

Details of submission will be discussed in lecture (and will also be included in the Moodle submission page.)