**Lab 4- Traveling Salesperson Problem with Depth First Search**

We will explore algorithms for solving the very famous traveling salesperson problem.   
  
Consider a graph of cities. The cost on an edge represents the cost to travel between two cities. The goal of the salesperson is to visit each city once and only once, and returning to the city he/she originally started at. Such a path is called a "tour" of the cities.   
How many tours are there? If there are N cities, then there can be N! possible tours. If we tell you which city to start at, there are (N-1)! Possible tours. For example, if there are 4 cities (A, B, C, D), and we always start at city A, then there are 3! possible tours:   
  
(A, B, C, D) (A, B, D, C) (A, C, B, D) (A, C, D, B) (A, D, B, C) (A, D, C, B)   
  
It is understood that one travels back to A at the end of the tour.   
  
The traveling salesperson problem consists of finding the tour with the lowest cost. The cost includes the trip back to the starting city. Clearly this is a horrendously difficult problem, since there are potentially (N-1)! possible solutions that need to be examined. We will consider DFS algorithms for finding the solution.   
  
The DFS algorithm is exhaustive - it will attempt to examine all (N-1)! possible solutions. This can be accomplished via a recursive algorithm (call it **recTSP**). This function is passed a "partial tour" (a sequence of M cities (M <= N) which is initially empty) and a “remaining cities” (sequence of N cities). There are clearly M-N cities not in this partial tour. Thus the function**recTSP** will have to call itself recursively M-N times, adding each of the M-N cities to the current partial tour out of the remaining cities. If M=N we have a complete tour.   
  
For example, we start with **recTSP** ({A}). This will have to call **recTSP** ({A, B}), **recTSP** ({A, C}, and **recTSP** ({A, D}). Here is a partial picture of how the sequence of function calls is done. This tree is not something you build explicitly - it arises from your function calls. You traverse this tree in a "depth-first" manner, The numbers tell you the order in which the nodes are processed.

Each leaf node is a complete tour, which you will compute the cost of. Note that each non-leaf node is an incomplete tour, which you can also compute the cost of. If the cost of an incomplete tour is greater than the best complete tour that you have found thus far, you clearly do not have to continue working on that incomplete tour. Thus you can "prune" your search.   
  
Just how hard are these problems? For example, if there are 29 cities, how many possible tours are there? If you can check 1,000,000 tours per second, how many years would it take to check all possible tours? Has the universe been around that long?   
  
Since this program may take too long to complete, be sure to output the tour and its cost when it finds a new best tour.

We have to first develop the distance matrix, also called adjacency matrix. This adjacency matrix is populated using a given data file. You will run your program to find the best tours for [12Preview the documentView in a new window](https://iu.instructure.com/courses/1485363/files/57136085/download?wrap=1), [14Preview the documentView in a new window](https://iu.instructure.com/courses/1485363/files/57136086/download?wrap=1), [16Preview the documentView in a new window](https://iu.instructure.com/courses/1485363/files/57136089/download?wrap=1), [19Preview the documentView in a new window](https://iu.instructure.com/courses/1485363/files/57136088/download?wrap=1), and [29Preview the documentView in a new window](https://iu.instructure.com/courses/1485363/files/57136090/download?wrap=1) cities.

Here is a sample code to populate the distance matrix

 public void populateMatrix(int[][]  adjacency){

int value, i, j;   
for (i = 0; i < CITI && input.hasNext(); i++) { //CITI is a constant    
  for (j = i; j < CITI && input.hasNext(); j++){   
     if (i == j) {   
            adjacency[i][j] = 0;  
     }  
     else {  
            value = input.nextInt();  
            adjacency[i][j] = value;  
            adjacency[j][i] = value;  
     }  
   }  
}

}

Here is an algorithm to compute tour cost

Algorithm computeCost (ArrayList<Integer> tour)

            Set totalcost = 0

            For (all cities in this tour)

                   totalcost += adjacency [tour.get(i)][ tour.get(i+1)]

            EndFor

            If (tour is a complete tour)

                        totalcost += adjacency [tour.get(tour.size()-1)][0]

            EndIf

            return totalcost

 End computeCost

Here is the DFS algorithm

Use ArrayList for “partialTour” and “remainingCities”

/\* requies : partialTour = <0>, remainingCities = <1,2, 3, ….N-2, N-1>

   ensures: partialTour = <0,…..n> where n E <1,2,3, …, N-1> &&

                 Cost(partialTour) is the absolute minimum cost possible.

\*/

Algorithm recDFS (ArrayList<Integer> partialTour, ArrayList<Integer> remainingCities )

            If (remainingCities is empty)

                        Compute tour cost for partialTour

                        If (tour cost is less than best known cost)

                                    Set best known cost with tour cost

                                    Output this tour and its cost

                        EndIf

            Else

                        For (all cities in remainingCities)

                                    Create a newpartialTour with partialTour

                                    Add the i\_th city of remainingCities to newpartialTour

                                    Compute the cost of newpartialTour

                                    If (newpartialTour cost is less than the best known cost) // pruning

                                                Create newRemainingCities with remainingCities

                                                Remove the i\_th city from newRemainingCities

                                                Call recDFS with newpartialTour and newRemainingCities

                                    EndIf

                        EndFor

            EndIf

 End recDFS

The minimal cost path for 12 cities is 821, and the minimal cost path for 29 cities is 1610,

**but 29! = 8841761993739700772720181510144 (!!!!!)**

Turn in your source program and outputs as an attachment of this assignment. You should copy and paste your outputs at the bottom of your source program.