Problem 1 (0.7 points)

The famous traveling salesmen problem is defined as follows:

Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?

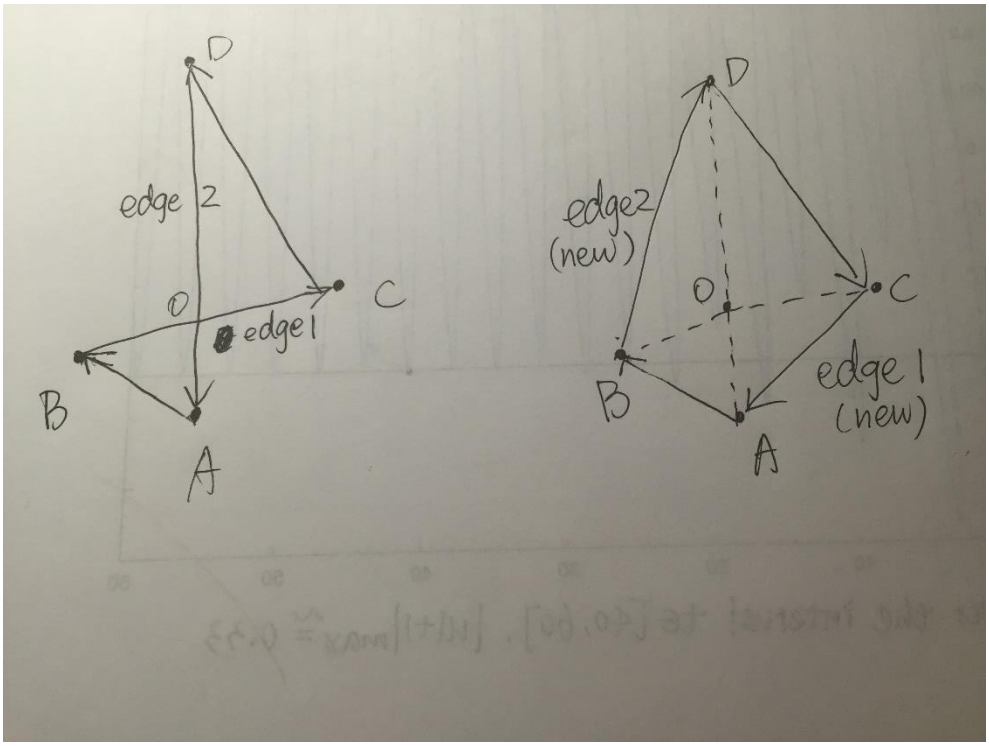
1) The most direct way to solve this problem is using BRUTAL FORCE SEARCH, which is to find all the possible routes through every city and find the shortest one. However, it is an almost impossible way to solve the problem. Explain why this is not a good idea.
2) If not using the direct way, it is also very hard to find a way to implement the algorithm. So one of the laziest guys came up with an algorithm called NEAREST NEIGHBOR ALGORITHM. The algorithm is just like its name and defined as follows:

   i) start on an arbitrary vertex as current vertex.
   ii) find out the shortest edge connecting current vertex and an unvisited vertex V.
   iii) set current vertex to V.
   iv) mark V as visited.
   v) if all the vertices in domain are visited, then terminate.
   vi) Go to step 2.

   With this algorithm, the runtime can be compressed down to O(n), where n is the number of cities needed to be visited. This algorithm was proven to be non-efficient but a possible quick way to find a path. Explain why this algorithm cannot always find the shortest path.

Solution

1) The runtime of this algorithm can be O(n!). When n goes larger than 100, the runtime will be extremely long to find the shortest path.



2) As shown on the graph, the first graph is the shortest path come up with the nearest neighbor algorithm. The second graph is the actual shortest path. The difference between these two shortest paths is edge 1 & 2. However, by using triangular inequality, we can easily figure out that the length of the addition of edge 1 & 2 is shorter than the length before. So the nearest neighbor algorithm cannot always find the shortest path on a TSP problem.

Problem 2 (0.7 points)

Please read the following algorithm and answer the two questions:

1. What is the goal of the algorithm?
2. If we want to recover the array after the function is executed, what should we do?

There is a function that accepts an array(unsign int * array). Let n be the length of the array and assume that each integer in the array has a value between 0 and n-1

1. Iterate though input array *arr[]*, for every element *arr[i]*, increment *arr[arr[i]%k]* by *k*
2. Find the maximum value in the modified array and return the corresponding index.

Solution

1. The algorithm finds the mode of the array, i.e., the most frequent element
2. Iterate through the array and set arr[i] = arr[i] % n

Problem 3 (0.6 points)

A node in a binary tree is an only-child if it has a parent node but no sibling node. The "loneliness-ratio" of a given binary tree T is defined as the following ratio: LR(t) = (the number of nodes in T that are only children)/(the number of nodes in T).
Prove that for any nonempty AVL tree T we have that LR(t)<=1/2.

Solution

Every lonely child has a unique parent that has a sibling, and hence, the number of lonely children is equal to the number of their parents (who have siblings); any other node in the AVL tree is not a lonely child. It follows that the number of lonely children is at most half the number of nodes in the AVL tree.