Traveling Salesman Problem

The TSP can be mathematically described as follows: Let a network G = [N, A, C] be defined with N the set of nodes, A the set of branches, and C = [c_{ij}] the matrix of costs. That is, c_{ij} is the cost of moving from node i to node j. Of course, other metrics such as time and distance can also be considered. The TSP requires finding a Hamiltonian cycle in G with minimal total; that is, it requires the determination of a minimal cost cycle that passes through each node in the relevant graph exactly once. If costs are symmetric, i.e., the cost of traveling between two locations does not depend on the direction of travel, the TSP is called symmetric or undirected; otherwise, asymmetric or directed. A feasible solution to a symmetric TSP has two arcs incident to each node, whereas for an asymmetric TSP there is one arc into and one arc out of every node.[1]

Implementations

Nearest Neighbor

The Nearest Neighbor Heuristic, which attempts to construct Hamiltonian cycles based on connections to near neighbors. The nearest neighbor procedure for a standard TSP runs in time O(n^2). This implementation always starts with the origin of (0,0). It finds the next unvisited node closest to the last node added to the path. Once it's found it adds that node to the path. This step is repeated until all the nodes are contained in the path. Then, it joins the first and last node. This approach is quick but not the best to find the optimal tour. Pseudocode of the nearest neighbor can be found below.

NearestNeighbor(P)

Pick and visit an initial point p0 from P

p = p0

i = 0

While there are still unvisited points

i = i + 1

Select pi to be the closest unvisited point to pi−1

Visit pi

Return to p0 from pn−1

[1] New Heuristic Algorithms for Solving Single-Vehicle and Multi-Vehicle Generalized Traveling Salesman Problems

Exhaustive

Exhaustive Search is a brute-force algorithm that systematically enumerates all possible solutions to a problem and checks each one to see if it is a valid solution. This algorithm is typically used for problems that have a small and well-defined search space, where it is feasible to check all possible solutions. This algorithm searches all O(n-1)! possible paths starting at 1, and keeps the best one. There is a great deal of parallelism, because after k recursive call to Search, there are (n-1)*(n-2)*...*(n-k) independent subtrees to search, which can be farmed out to as many processors. Since subtrees are all equally large, the load balance is perfect.[2]

OptimalTSP(P)

d = ∞

For each of the n! permutations Pi of point set P

If (cost(Pi) ≤ d) then d = cost(Pi) and Pmin = Pi

Return Pmin

Worst-case Time Complexity

To compute running time of recursive functions, an upper bound on the running time will be the number of times the recursive function is called multiplied by the runtime of the function in a single call.

Nearest Neighbor

The time complexity of the nearest neighbor algorithm is O(n^2). The number of computations required will not grow faster than (n^2). No constant worst-case performance guarantee can be given. In fact, it can be shown that for arbitrarily large n there exists TSP instances on n nodes such that the nearest neighbor solution is Θ(logn) times as long as an optimal Hamiltonian cycle.

Exhaustive

To see that the worst-case runtime is O((n-1)!), consider how many paths are in a fully connected graph - you can visit any node directly from any node. Another way of phrasing this is that you can visit the nodes in any order, save the starting state. This is the same as the number of

[2] From the edited volume of Theory of Complexity article How to Solve the Traveling Salesman Problem

permutations of (n-1) elements. This is going to be O(n!), since we are iterating over all edges which takes O(n) for each state on the path (n×(n-1)!).

Random Algorithm

Nearest Neighbor		0.000369611	Exhaustive Search		3.56349E-07
n	avg runtime(ns)	c1 * O(n^2)	n	avg runtime(s)	C2*O((n+1)!)
300	33.265	33.265	6	0.001796	5040
600	139.601	133.06	8	0.043421	362880
900	278.0412	299.385	10	3.461548	39916800
1200	499.562	532.24	12	70.97236	6227020800

For the nearest neighbor heuristic, it's logical to make n greater than exhaustive search since nearest neighbor has a fast approach, however it's not guaranteed to find the optimal tour. I choose to increment n by 300 so it can still run and make the 10 second limit. The exhaustive search has a slower approach which is why I choose to increment n by two; n starts at six since the original input for this assignment already has 4 points. When a TSP instance is large, the number of possible solutions in the solution space is so large as to forbid an exhaustive search for the optimal solutions.

Theory and Practice

The traveling salesman problem asks 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? For the nearest neighbor algorithm, the number of operations the algorithm does as a function of its input size defines the algorithm complexity. As n moves towards infinity, this ratio converges to a positive constant. In this solution for an input N = 300 has to do 900 things and for the input of N = 600 the work increased from 900 to 1200, etc.

In this solution, every time we increase the input by one, we have to do a significant number of additional steps and is clear that is not a 1 to 1 relation between the input and the algorithm. We can say that the amount of work had to do N squared $O(n^2) = (300^2, 600^2, 900^2, \ldots)$ or quadratic runtime. The same theory follows for the exhaustive algorithm, however, because n is less than nearest neighbor the increment is lower.

Appendix A

```
import sys
import timeit

start = timeit.default_timer()

data = sys.argv[1]

#data = "data.txt"

### read data to list
with open(data,"r") as rf:
   rln = rf.readlines()

nline = int(rln[0].strip())

lspts = []
for i in range(nline):
   readdata = [int(j) for j in rln[i+1].strip().split()]
   lspts += [readdata]
   #print (readdata)


### find distance
def distance(p1,p2):
   " find distance from p1 to p2 "
   dist = ( (p1[0] - p2[0])**2 + (p1[1] - p2[1])**2 ) ** 0.5
   return dist


def findclosest(p1,lsp):
   " find closest distance from p1 to list of points lsp "
   distinit = 10e10
   ptfound = []
   distfound = 0
   idxfound  = 0
   iidx = -1
   for p2 in lsp:
```

```
      iidx += 1
      dist = distance(p1,p2)
      if dist < distinit:
         distinit = dist
         ptfound   = p2
         distfound = dist
         idxfound  = iidx

   return ptfound,distfound,idxfound

totdist  = 0
pointseq = [lspts[0]]


for i in range(len(lspts)-1):
   if i==0:
      #print (i)

      lssearch = lspts[1:]
      p1,d,idxf = findclosest(lspts[0],lssearch)
      #print (lspts[0],lssearch,idxf,d,p1)

      totdist  += d
      pointseq += [p1]

      #print ("totdist:",totdist)

   else:
      #print ("\n-----",i)
      del lssearch[idxf]
      #print (p1,lssearch)

      p1,d,idxf = findclosest(p1,lssearch)
      #print ("   found:",idxf,p1,d)

      #print ("tot dist before adding:",totdist)
      totdist  += d
      #print ("totdist:",totdist)
      pointseq += [p1]

### count back to initial index 0
#print ("end:",p1)
d = distance(p1,lspts[0])
totdist  += d
```

```
print ("{:.3f}".format(totdist))
#print (pointseq+[lspts[0]])

#print ("\n\nProcessing time {:.9f} secs\n\n".format(timeit.default_timer()-
start))
```

Appendix B

```
import sys
import timeit

start = timeit.default_timer()

#data = "data.txt"

### read data to list
def readfile(filein):
   with open(filein,"r") as rf:
      rln = rf.readlines()

   nline = int(rln[0].strip())

   lspts = []
   for i in range(nline):
      readdata = [int(j) for j in rln[i+1].strip().split()] # list comprehension
4 di dalem list
      lspts += [readdata]
      #print (readdata)
   return lspts # stores in list set

### find distance
def distance(p1,p2):
   " find distance from p1 to p2 "
   dist = ( (p1[0] - p2[0])**2 + (p1[1] - p2[1])**2 ) ** 0.5
   return dist


def findclosest(p1,lsp):
   " find closest distance from p1 to list of points lsp "
   distinit = 10e10
   ptfound = []
   distfound = 0
   idxfound  = 0
   iidx = -1
```

```
   for p2 in lsp:
      iidx += 1
      dist = distance(p1,p2)
      if dist < distinit:
         distinit = dist
         ptfound   = p2
         distfound = dist
         idxfound  = iidx

   return ptfound,distfound,idxfound


def main_NN(lspts): #nearest neighbor

   totdist  = 0
   pointseq = [lspts[0]]

   for i in range(len(lspts)-1):
      if i==0:
         #print (i)

         lssearch = lspts[1:]
         p1,d,idxf = findclosest(lspts[0],lssearch)
         #print (lspts[0],lssearch,idxf,d,p1)

         totdist  += d
         pointseq += [p1]

         #print ("totdist:",totdist)

      else:
         #print ("\n-----",i)
         del lssearch[idxf]
         #print (p1,lssearch)

         p1,d,idxf = findclosest(p1,lssearch)
         #print ("   found:",idxf,p1,d)

         #print ("tot dist before add:",totdist)
         totdist  += d
         #print ("totdist:",totdist)
         pointseq += [p1]

   ### count back to initial index 0
   #print ("end:",p1)
```

```
   d = distance(p1,lspts[0])
   totdist  += d

   print ("\nTotal distance = {:.3f}".format(totdist))
   print (pointseq+[lspts[0]])

   return totdist,pointseq+[lspts[0]]


def main_Distance(lspts):

   totdist  = 0
   for i in range(len(lspts)-1):
      d = distance(lspts[i],lspts[i+1])
      totdist += d
   ### count back to 0
   d = distance(lspts[i+1],lspts[0])
   totdist += d

   #print (totdist)

   return totdist


# exhaustive search
filein = sys.argv[1]
lsptsr = readfile(filein)

if False:
   main_NN(lsptsr)
if True:
   TotDistMin = 10e10
   seqselect  = ""

   lsidx = [ i for i in range(len(lsptsr))]
   # ambil index aja

   import itertools
   # to use premutation

   permutation = list(itertools.permutations(lsidx[1:],len(lsidx)-1)) # ambil
index yg dari 1 since 0 is fixed then add (0,0) at the end
   for lsidx in permutation:                                          # do not
hardcode use len
      #print ("\n=======================")
```

```
      lsptsi = [lsptsr[0]] + [lsptsr[j] for j in lsidx] # list comprehension
      #print (lsidx , lsptsi )

      totdisti = main_Distance(lsptsi)
      #print ("     ",totdisti)

      if totdisti<TotDistMin: # simpen dulu point that's less that 10e10
         TotDistMin = totdisti
         seqselect  = lsptsi # sequence select

   print("{:.3f}".format(TotDistMin))
   #print ("\nExhaustive --> min distance {:.3f}    seq =
{}".format(TotDistMin,seqselect))

#print ("\n\n  Processing time {:.9f} secs\n\n".format(timeit.default_timer()-
start))
```