CS 325: Project 4

Robert Erick, Jacob Mastel, Cera Olson

7 June 2015

Contents

1 The Algorithm

T he algorithm used by our group to solve this problem was Prim's Algorithm. Prim's Algorithm is a greedy algorithm that finds the shortest distance between a list of nodes on an undirected, weighted graph. The is a common and effective algorithm used to solve the Traveling Salesman Problem (TSP). See Appendix 1 for the actual code we used to find our results.

O ur Algorithm is programmed in Python. We started by defining 2 global functions: aCity and getCities. aCity takes list of strings and converts them to integers, the cost of movement to the city. getCities takes the values returned in aCity and creates a list of these items. getCities is used in the different classes to find the neighboring cities and to determine the cost to their connections.

1.1 Node Class

The node class represents a single city. This class contains multiple functions that define the individual values of each node in a tree - distance, cost, neighbors, and location in the spanning tree.

1.2 MST Class

1.3 TSP Class

2 The Tours

2.1 Tour 1

2.2 Tour 2

2.3 Tour 3

2.4 Competition Tours

3 Appendix

3.1 Appendix 1: Code

```
#Robert Erick & project 29 group
#project #4
#traveling salesperson
#6/5/2015
#this follows the methodology outlined on page 1112 of "introduction to algorith
#this also follows the methodology on page 634, ibid, for Prim's algorithm
import time

#controls the search
MAX_COMPARE=20
MAX_SEARCH_LATEST=40
#controls the lookup cache
MAX_CACHE=20000
MIN_CACHE=5000
MAX_CACHE_DIST=10000
#controls other
SORT_NOTCONNECTED=True
MAX_FILE=4
DEBUG=False

def aCity(l):#this takes a 3 list of strings and converts to integers
    tmp=l.rstrip()
    tmp=l.split()
    tmp=list(tmp)
    tmp=[int(x) for x in tmp]
    return tmp

def getCities(pth):#this returns a list of lists, not yet node objects, from fil
    fobj=open(pth,'r')
    cities=list(fobj)
    fobj.close()
    cities=[aCity(c) for c in cities]
    return cities

class node(object):#this node object represents a single city
    def __init__(self,lst,memo):
        self.no,self.x,self.y=lst
```

```
##          self.diag=(self.x**2+self.y**2)**.5
        self.memo=memo
        self.parent=None
        self.children=[]

    def distance(self,other):
        d=(other.x-self.x)**2+(other.y-self.y)**2
        d=d**.5
        return int(round(d,0))

    def minEdge(self,notConnected):
        me=None
        theMax=min(len(notConnected),MAX_COMPARE)
        for other in notConnected[:theMax]:# limited
            if other==self:continue
            oid=id(other)
            k=[id(self),oid]
            k.sort()
            k=tuple(k)

            d=None
            if self.memo.has_key(k):# using memo
                d=self.memo[k][0]
            else:
                d=self.distance(other)
                if d<MAX_CACHE_DIST:
                    last=self.memo['last']+1
                    self.memo['last']=last
                    self.memo[k]=(d,last)
            if me==None or d<me[0]:# make the tuple
                me=(d,self,other)
            self.adjustMemo()# adjust the memo if too big
        return me

    def preorder(self):
        tmp=[self]
        for c in self.children:
            tmp.extend(c.preorder())
        return tmp

    def adjustMemo(self):
```

```
        if len(self.memo)<MAX_CACHE:return
        if DEBUG:print 'adjusting memo'
        threshold=self.memo['last']-MIN_CACHE
        keys=self.memo.keys()
        for k in keys:
            if k=='last':continue
            v=self.memo[k]
            try:
                i=v[1]
                if i<threshold:self.memo.pop(k)
            except:
                print 'problem',k,v

    def __eq__(self,other):return ((self.x,self.y)==(other.x,other.y))
    def __ne__(self,other):return not self==other
    def __gt__(self,other):return ((self.x,self.y)>(other.x,other.y))
    def __lt__(self,other):return ((self.x,self.y)<(other.x,other.y))
    def __ge__(self, other):return (self>other) or (self==other)
    def __le__(self, other):return (self<other) or (self==other)


    def printTree(self,lvl=0):#just for debug
        padding=' '*lvl
        print '%s Level: %s <Node>: no=%i,x=%i,y=%s'%(padding,lvl,self.no,self.x
        for c in self.children:
            c.printTree(lvl+1)

    def __str__(self):#just for debug
        return '<Node>: no=%i,x=%i,y=%s'%(self.no,self.x,self.y)

class mst(object):
    #lst is a list of lists
    #each sublist is of form [id number,x coordinate,y coordinate]
    def __init__(self,lst):
        self.memo={'last':0,}
        self.notConnected=[node(x,self.memo) for x in lst]
        self.connected=[]
        self.root=None
        self.getMst()

    def getMst(self):
```

```
        if SORT_NOTCONNECTED: self.notConnected.sort()
        n=self.notConnected.pop(0) #should this just be 0? or better choice?
        self.connected.append(n)
        self.root=n

        i=0
        denom=len(self.notConnected)
        modval=int(denom*.05)
        if denom>5000:modval=int(denom*.01)
        denom=float(denom)
        while(self.notConnected):#while notConnect is not empty
            if i%modval==0:
                num=len(self.connected)
                frac=num*100/denom
                print '%.0f%%'%(frac)
            self.extractMin()#extract the minimum and join it
            i+=1
        print

    def extractMin(self):#part of prim's algorithm
        me=None
        for i,node in enumerate(self.connected[-MAX_SEARCH_LATEST:]):
            anme=node.minEdge(self.notConnected)
            if anme!=None:
                d=anme[0]
                if me==None or d<me[0]:
                    me=anme
        if me:#now join the minimum edge into the existing graph
            frm=me[1]
            to=me[2]
            self.connected.append(to)
            self.notConnected.remove(to)
            to.parent=frm
            frm.children.append(to)  ###probably could save distance too

    def printAll(self):#just for debug
        for node in self.allNodes:
            print node

class tsp(mst):#extends mst and adds a preorder and presentation
    def __init__(self,lst):
```

```
        mst.__init__(self,lst)
        self.preorder=[]
        self.total=0
        self.presentation=[]

        self.getPreorder()
        self.getTotal()
        self.getPresentation()

    def getPreorder(self):
        self.preorder=self.root.preorder()
    def getTotal(self):
        for i in range(len(self.preorder)-1):
            o1=self.preorder[i]
            o2=self.preorder[i+1]
            d=o1.distance(o2)
            self.total+=d
    def getPresentation(self):
        self.presentation=[self.total]
        self.presentation.extend([x.no for x in self.preorder])
    def fileLines(self):#can call this for fileobj.writelines(tsp.fileLines())
        return ['%s\n'%x for x in self.presentation]


if __name__=='__main__':
    for i in range(1,MAX_FILE):#(1,4)!
        start=time.asctime()
        pth='tsp_example_%i.txt'%i
        cities=getCities(pth)
        atsp=tsp(cities)
        p=atsp.presentation
        print 'total is: %i'%atsp.total
        out='tsp_example_%i_test.txt'%i
        print 'writing to %s'%out
        fobj=open(out,'w')
        fobj.writelines(atsp.fileLines())
        fobj.close()
        stop=time.asctime()
        print 'start %s, stop %s'%(start,stop)
        print 'Done.'
        for j in range(5):
```

```
print
```

3.2 Appendix 2: Resources Used

- Prim's Algorithm Description from "Introductions to Algorithm" by Corman, et al.