**Graph algorithms - minimum cost path**

Algorithms for finding the minimum cost path between two given vertices.

Example with positive costs:  
A diagram of a network

Description automatically generated

Example with some negative costs:  
A diagram of a diagram

Description automatically generated

Issues:

1. Execute Dijkstra's algorithm on the two examples above;
2. Implement Dijkstra's algorithm (start from BFS from last time and replace the FIFO queue with a priority queue);
3. Analyse its behavior on the two examples above;

Sample partial implementations: [parse.py](https://www.cs.ubbcluj.ro/~rlupsa/edu/grafe/samples/graph.py), [dijkstra.py](https://www.cs.ubbcluj.ro/~rlupsa/edu/grafe/samples/dijkstra.py) and [bellman.py](https://www.cs.ubbcluj.ro/~rlupsa/edu/grafe/samples/bellman.py)

Parse.py:

import random

class MatrGraph:

"""A directed graph, represented by adjacency matrix.

Vertices are numbers from 0 to n-1"""

def \_\_init\_\_(self, n):

"""Creates a graph with n vertices (numbered from 0 to n-1)

and no edges"""

self.\_matr = []

for i in range(n):

self.\_matr.append([])

for j in range(n):

self.\_matr[i].append(False)

def parseX(self):

"""Returns an iterable containing all the vertices"""

nrOfVertices = len(self.\_matr)

return range(nrOfVertices)

def parseNout(self, x):

"""Returns an iterable containing the outbound neighbours of x"""

list =[]

for i in range(len(self.\_matr[x])):

if self.\_matr[x][i] :

list.append(i)

return list

def parseNin(self,x):

"""Returns an iterable containing the inbound neighbours of x"""

list =[]

for i in range(len(self.\_matr)):

if self.\_matr[i][x] :

list.append(i)

return list

def isEdge(self,x,y):

"""Returns True if there is an edge from x to y, False otherwise"""

return self.\_matr[x][y]

def addEdge(self,x,y):

"""Adds an edge from x to y.

Precondition: there is no edge from x to y"""

self.\_matr[x][y] = True

class DictGraph:

"""A directed graph, represented as a map from each vertex to

the set of outbound neighbours"""

def \_\_init\_\_(self,n):

"""Creates a graph with n vertices (numbered from 0 to n-1)

and no edges"""

self.\_dict={}

for i in range(n):

self.\_dict[i]=[]

def parseX(self):

"""Returns an iterable containing all the vertices"""

return self.\_dict.keys()

def parseNout(self,x):

"""Returns an iterable containing the outbound neighbours of x"""

return self.\_dict[x]

def parseNin(self,x):

"""Returns an iterable containing the inbound neighbours of x"""

list=[]

for i in self.\_dict.keys():

if x in self.\_dict[i]:

list.append(i)

return list

def isEdge(self,x,y):

"""Returns True if there is an edge from x to y, False otherwise"""

return y in self.\_dict[x]

def addEdge(self,x,y):

"""Adds an edge from x to y.

Precondition: there is no edge from x to y"""

self.\_dict[x].append(y)

class DoubleDictGraph:

"""A directed graph, represented as two maps,

one from each vertex to the set of outbound neighbours,

the other from each vertex to the set of inbound neighbours"""

def \_\_init\_\_(self,n):

"""Creates a graph with n vertices (numbered from 0 to n-1)

and no edges"""

self.\_dictOut={}

self.\_dictIn = {}

for i in range(n):

self.\_dictOut[i]=[]

self.\_dictIn[i] = []

def parseX(self):

"""Returns an iterable containing all the vertices"""

return self.\_dictOut.keys()

def parseNout(self,x):

"""Returns an iterable containing the outbound neighbours of x"""

return self.\_dictOut[x]

def parseNin(self,x):

"""Returns an iterable containing the inbound neighbours of x"""

return self.\_dictIn[x]

def isEdge(self,x,y):

"""Returns True if there is an edge from x to y, False otherwise"""

return y in self.\_dictOut[x]

def addEdge(self,x,y):

"""Adds an edge from x to y.

Precondition: there is no edge from x to y"""

self.\_dictOut[x].append(y)

self.\_dictIn[y].append(x)

def accessible(g, s):

"""Returns the set of vertices of the graph g that are accessible

from the vertex s"""

acc = set()

acc.add(s)

list = [s]

while len(list) > 0:

x = list[0]

list = list[1 : ]

for y in g.parseNout(x):

if y not in acc:

acc.add(y)

list.append(y)

return acc

class GoatStatus:

def \_\_init\_\_(self, i):

self.\_status = i

def \_\_str\_\_(self):

return self.strX(~self.\_status) + "/" + self.strX(self.\_status)

def \_\_eq\_\_(self, other):

if isinstance(other, self.\_\_class\_\_):

return self.\_\_dict\_\_ == other.\_\_dict\_\_

else:

return False

def \_\_ne\_\_(self, other):

return not self.\_\_eq\_\_(other)

def \_\_hash\_\_(self):

return self.\_status

def isValid(self):

"""True if nobody eats nobody in this state"""

return self.isValidBank(self.\_status) and self.isValidBank(~self.\_status)

def parseN(self):

ret = []

for i in range(4):

if (self.\_status & 1) == ((self.\_status >> i) & 1):

ns = self.\_status ^ ((1 << i) | 1);

s = GoatStatus(ns)

if s.isValid():

ret.append(s)

return ret

def isValidBank(self, i):

return (i&4) == 0 or (i&1) == 1 or ((i&2) == 0 and (i&8) == 0)

def strX(self, i):

ret = "("

for j in range(4):

if (i & (1<<j)) != 0:

ret = ret + " " + self.names[j]

return ret + ")"

names = ("boat", "cabbage", "goat", "wolf")

class GoatGraph:

def parseX(self):

ret = []

for i in range(16):

s = GoatStatus(i)

if s.isValid():

ret.append(s)

return ret

def parseNout(self, s):

return s.parseN()

def parseNin(self, s):

return s.parseN()

def initMyGraph(ctor):

"""Constructs and returns a hard-coded sample graph.

ctor must be a callable that gets the number of vertices and

creates a graph with the given number of vertces and with no edges"""

g = ctor(5)

g.addEdge(0,1)

g.addEdge(1,0)

g.addEdge(1,1)

g.addEdge(1,2)

g.addEdge(4,0)

g.addEdge(4,2)

return g

def initRandomGraph(ctor,n,m):

"""Constructs and returns a randomly generated graph

with n vertices and m edges.

ctor must be a callable that gets the number of vertices and

creates a graph with the given number of vertces and with no edges"""

g=ctor(n)

addedEdges=0

while addedEdges < m:

x=random.randrange(0,n)

y=random.randrange(0,n)

if not g.isEdge(x,y):

g.addEdge(x,y)

addedEdges+=1

return g

def run():

g = initMyGraph(DoubleDictGraph)

for x in g.parseX():

print ("%s:" % x)

for y in g.parseNin(x):

print ("%s <- %s" % (x, y))

run()

g = initMyGraph(MatrGraph)

s = 0

print accessible(g, s)

Dijkstra.py:  
import graph

class PriorityQueue:

def \_\_init\_\_(self):

self.\_\_values = {}

def isEmpty(self):

return len(self.\_\_values) == 0

def pop(self):

topPriority = None

topObject = None

for obj in self.\_\_values:

objPriority = self.\_\_values[obj]

if topPriority is None or topPriority>objPriority:

topPriority = objPriority

topObject = obj

del self.\_\_values[topObject]

return topObject

def add(self, obj, priority):

self.\_\_values[obj] = priority

def contains(self, val):

return val in self.\_\_values

def getChildren(x,prev):

list=[]

for i in prev:

if prev[i]==x:

list.append(i)

return list

def printDijkstraTree(s, q, d, prev,indent):

if q.contains(s):

star = ''

else:

star = '\*'

print "%s%s [%s]%s" % (indent,s, d[s], star)

for x in getChildren(s,prev):

printDijkstraTree(x,q,d,prev,indent+' ')

def printDijkstraStep(s, x, q, d, prev):

print '----'

if x is not None:

print 'x=%s [%s]' % (x, d[x])

printDijkstraTree(s,q,d,prev,'')

def dijkstra(g, s):

prev = {}

q = PriorityQueue()

q.add(s, 0)

d = {}

d[s] = 0

visited = set()

visited.add(s)

printDijkstraStep(s, None, q, d, prev)

while not q.isEmpty():

x = q.pop()

for y in g.parseNout(x):

if y not in visited or d[y] > d[x] + g.cost(x, y):

d[y] = d[x] + g.cost(x, y)

visited.add(y)

q.add(y, d[y])

prev[y] = x

printDijkstraStep(s, x, q, d, prev)

return (d, prev)

def printTree(tree,root):

printTreeAux(tree,root,"")

def printTreeAux(tree,root,indent):

print indent+str(root)

children=tree[root]

newindent=indent+" "

for i in children:

printTreeAux(tree,i,newindent)

def getPath(s, t, prev):

list = []

while t != s:

list.append(t)

t = prev[t]

ret = [s]

for i in range(len(list)):

ret.append(list[len(list) - i - 1])

return ret

def bellman(g, s):

w = [{s : 0}]

path = [{s : (0,)}]

print "k=%s" % 0

print "w=%s" % w[0]

print "path = %s" % path[0]

for k in range(1, len(g.parseX())):

w.append({})

path.append({})

for y in w[k-1]:

for x in g.parseNout(y):

if ((x not in w[k]) or

(w[k][x]>w[k-1][y]+g.cost(x, y) )):

w[k][x] = w[k-1][y]+g.cost(y, x)

path[k][x] = path[k-1][y]+ (x,)

print "k=%s" % k

print "w=%s" % w[k]

print "path = %s" % path[k]

g = graph.initSecondGraph(graph.DoubleDictGraph)

s = 0

t = 4

dijkstra(g, s)

#d, prev = dijkstra(g, s)

#print d

#print prev

Bellman.py:  
import graph

def computeIsWalk(graph, start, maxLen):

initSet = set()

initSet.add(start)

isWalk = [initSet]

for k in range(1, maxLen+1):

prevSet = isWalk[k-1]

currSet = set()

for y in prevSet:

for x in graph.parseNout(y):

currSet.add(x)

isWalk.append(currSet)

return isWalk

def getExistingWalk(graph, isWalk, start, target, length):

walk = []

currVertex = target

currLength = length

while currLength > 0:

walk.insert(0, currVertex)

for prevVertex in graph.parseNin(currVertex):

if prevVertex in isWalk[currLength-1]:

currVertex = prevVertex

break

currLength = currLength - 1

walk.insert(0, start)

return walk

def loverProblemSolution(graph, start1, start2):

nrVertices = len(graph.parseX())

maxLen = nrVertices\*(nrVertices-1)

isWalk1 = computeIsWalk(graph, start1, maxLen)

isWalk2 = computeIsWalk(graph, start2, maxLen)

for i in range(maxLen):

for x in isWalk1[i]:

if x in isWalk2[i]:

return (i, getExistingWalk(graph, isWalk1, start1, x, i),

getExistingWalk(graph, isWalk2, start2, x, i) )

return None

def bellman(graph, start, maxLen):

initMap = {start : 0}

dist = [initMap]

for k in range(1, maxLen+1):

prevMap = dist[k-1]

currMap = {}

for y in prevMap:

for x in graph.parseNout(y):

if x not in currMap or currMap[x]>prevMap[y]+graph.cost(y,x) :

currMap[x] = prevMap[y]+graph.cost(y,x)

dist.append(currMap)

return dist

def getMinCostWalk(graph, dist, start, target, length):

walk = []

currVertex = target

currLength = length

while currLength > 0:

walk.insert(0, currVertex)

for prevVertex in graph.parseNin(currVertex):

if prevVertex in dist[currLength-1] and (

dist[currLength-1][prevVertex]

+ graph.cost(prevVertex, currVertex)

== dist[currLength][currVertex]):

currVertex = prevVertex

break

currLength = currLength - 1

walk.insert(0, start)

return walk

def run():

g = graph.initLoverProblemGraph(graph.DoubleDictGraph)

print loverProblemSolution(g, 0, 2)

def run1():

g = graph.initSecondGraph(graph.DoubleDictGraph)

s = 0

t = 5

maxLen = 3\*len(g.parseX())

dist = bellman(g, s, maxLen)

for i in range(maxLen+1):

if t in dist[i]:

print (getMinCostWalk(g, dist, s, t, i), dist[i][t])

run1()