**Graph algorithms - trees**

Construct the Minimum Spanning Tree for the following graph

A diagram of a triangle with numbers and lines

Description automatically generated

Code samples: [mintree.py](https://www.cs.ubbcluj.ro/~rlupsa/edu/grafe/samples/mintree.py)

import copy

from heapq import heappop, heappush

class DictGraph:

"""A directed graph"""

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

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

and no edges"""

self.\_dict={}

self.\_cost={}

for i in vertices:

self.\_dict[i]=[]

def parseX(self):

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

return self.\_dict.keys()

def parseN(self,x):

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

return self.\_dict[x]

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 cost(self, x, y):

return self.\_cost[(x,y)]

def addEdge(self,x,y,c):

"""Adds an edge from x to y.

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

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

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

self.\_cost[(x,y)] = c

self.\_cost[(y,x)] = c

def genGraph():

g = DictGraph(range(1,7))

g.addEdge(1,2,3)

g.addEdge(1,3,2)

g.addEdge(1,4,4)

g.addEdge(2,3,2)

g.addEdge(2,6,1)

g.addEdge(3,4,4)

g.addEdge(3,5,3)

g.addEdge(3,6,2)

g.addEdge(4,5,5)

g.addEdge(5,6,5)

return g

def prim2(g):

for x in g.parseX():

s = x

break

prev = {} # prev[x] = (if x outside of the tree) the vertex of the tree that is closest to x

q = []

d = {} # d[x] = (if x is outside of the tree) cost of the min cost edge from any vertex of the tree to x

d[s] = 0

treeVertices = set()

treeVertices.add(s)

treeEdges = []

for x in g.parseN(s):

heappush(q, (g.cost(s,x), x))

d[x] = g.cost(s,x)

prev[x] = s

treeCost = 0

print(s, q, d, prev)

while len(q)>0:

c,x = heappop(q)

if x in treeVertices:

print("Skip ", x)

continue

for y in g.parseN(x):

if y not in treeVertices and (y not in d.keys() or d[y] > g.cost(x, y)):

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

prev[y] = x

heappush(q, (d[y], y))

print(x, q, d, prev)

treeVertices.add(x)

treeEdges.append((x, prev[x]))

treeCost = treeCost + c

return (treeEdges, treeCost)

def prim(g):

for x in g.parseX():

s = x

break

q = []

for x in g.parseN(s):

heappush(q, (g.cost(s,x), s, x,))

print("New candidate edge: (%x,%x) of cost %s" % (s,x,g.cost(s,x)))

treeVertices = set([s])

treeEdges = []

treeCost = 0

while len(q) > 0:

cost, x, y = heappop(q)

print("Processing: (%x,%x) of cost %s" % (x,y,cost))

if y in treeVertices:

print("Skip")

continue

treeVertices.add(y)

treeEdges.append((x,y))

treeCost = treeCost + cost

for z in g.parseN(y):

if z not in treeVertices:

heappush(q, (g.cost(y,z), y, z))

print("New candidate edge: (%x,%x) of cost %s" % (y,z,g.cost(y,z)))

return (treeEdges, treeCost)

class DisjointSet:

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

self.parent = {}

self.height = {}

for x in vertices:

self.parent[x] = None

self.height[x] = 0

def checkAndMerge(self, x, y):

'''Checks if x and y are verices belonging to the same connected component or not.

If they are in the same component, the function returns False

If they are in distinct components, it merges the two components and returns True

'''

rx = self.getRoot(x)

ry = self.getRoot(y)

print("root(%s)=%s, root(%s)=%s"% ( x, rx, y, ry))

if rx == ry:

return False

hx = self.height[rx]

hy = self.height[ry]

if hx < hy:

self.parent[rx] = ry

del self.height[rx]

print("%s->%s" % (rx, ry))

elif hx > hy:

self.parent[ry] = rx

del self.height[ry]

print("%s->%s" % (ry, rx))

else:

self.parent[ry] = rx

del self.height[ry]

self.height[rx] = self.height[rx] + 1

print("%s->%s" % (ry, rx))

return True

def getRoot(self, x):

while self.parent[x] is not None:

x = self.parent[x]

return x

def kruskal(g):

edges = []

for x in g.parseX():

for y in g.parseN(x):

if x < y:

edges.append((g.cost(x,y), x, y))

edges.sort()

components = DisjointSet(g.parseX())

tree = []

treeCost = 0

for cost, x, y in edges:

if components.checkAndMerge(x,y):

tree.append((x,y))

treeCost = treeCost + cost

return (tree, treeCost)

g = genGraph()

tree,totalCost = prim2(g)

print(tree,totalCost)