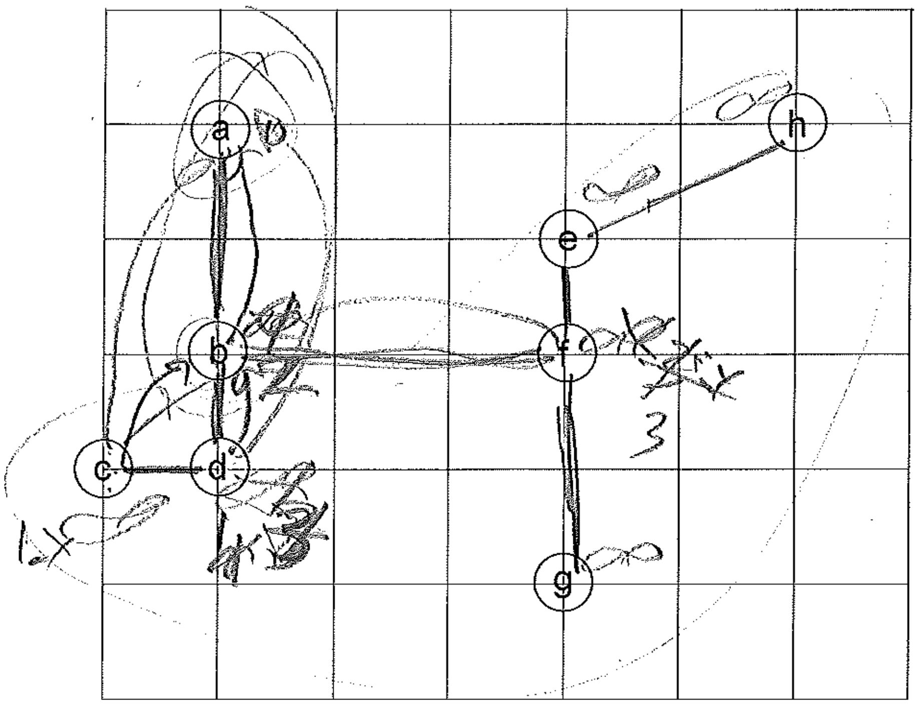
Data Structures (CS 1520) Lecture 27 Name: Mark F.

1. Suppose you had a map of settlements on the planet X
   (Assume edges could connecting all vertices with their Euclidean distances as their costs)



We want to build roads that allow us to travel between any pair of cities. Because resources are scarce, we want the total length of all roads build to be minimal. Since all cities will be connected anyway, it does not matter where we start, but assume we start at "a".

a) Assuming we start at city "a" which city would you connect first? Why this city? b

because it is closest to a

b) What city would you connect next to expand your partial road network? d because it is closest to partial road network.

c) What would be some characteristics of the resulting "graph" after all the cities are connected?

No cycles → "tree"
Connects all vertices → "spanning tree"

d) Does your algorithm come up with the overall best (globally optimal) result?

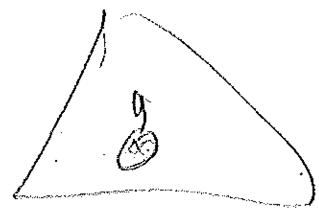
"Minimum spanning tree", MST

2. Prim's algorithm for determining the minimum-spanning tree (MST) of a graph is another example of a *greedy algorithm*. Unlike divide-and-conquer and dynamic programming algorithms, greedy algorithms DO NOT divide a problem into smaller subproblems. Instead a greedy algorithm builds a solution by making a sequence of choices that look best ("locally" optimal) at the moment without regard for past or future choices (no backtracking to fix bad choices).

a) What greedy criteria does Prim's algorithm use to select the next vertex and edge to the partial minimum spanning tree? add next vertex that's closest to partial MST

b) Consider the textbook's Prim's Algorithm code (Listing 7.12 p. 346) which is incorrect.

```
def prim(G,start):
    pq = PriorityQueue()   BinHeap()
    for v in G:                          PriorityQueueEntry(v.getDistance(), v)
        v.setDistance(sys.maxsize)
        v.setPred(None)        ∞           create list
    start.setDistance(0)
    pq.buildHeap([(v.getDistance(),v) for v in G])
    while not pq.isEmpty():
(0,start) currentVert = pq.delMin().getValue()
        for nextVert in currentVert.getConnections():
          newCost = currentVert.getWeight(nextVert) \
PriorityQueueEntry(newCost, nextVertex)  + currentVert.getDistance()
          if nextVert in pq and newCost<nextVert.getDistance():
              nextVert.setPred(currentVert)
              nextVert.setDistance(newCost)
              pq.decreaseKey(nextVert,newCost)
```



c) What is wrong with the code? (Fix the above code.)

(See attached code)

3. To avoid "massive" changes to the binHeap class, it can store `PriorityQueueEntry` objects:

```
class PriorityQueueEntry:
    def __init__(self,x,y):
        self.key = x    <= priority / distance
        self.val = y    <= vertex

    def getKey(self):
        return self.key

    def getValue(self):
        return self.val

    def setValue(self, newValue):
        self.val = newValue
```

```
    def __lt__(self,other):
        return self.key < other.key

    def __gt__(self,other):
        return self.key > other.key

    def __eq__(self, other):
        return self.val == other.val

    def __hash__(self):
        return self.key
```

a) Update the above Prim's algorithm code to use `PriorityQueueEntry` objects.

b) Why do the `__lt__` and `__gt__` methods compare `key` attributes, but `__eq__` compare `val` attributes?

```
from graph import Graph
from binheap import BinHeap
from binheap import PriorityQueueEntry
import sys

def prim(G,start):
    pq = BinHeap()
    for v in G:
        v.setDistance(sys.maxsize)
        v.setPred(None)
    start.setDistance(0)
    pq.buildHeap([PriorityQueueEntry(v.getDistance(),v) for v in G])
    while not pq.isEmpty():
        currentVert = pq.delMin().getValue()
        if currentVert.getPred() != None:
          print("Prim: edge",currentVert.getPred().getId(),
                "to", currentVert.getId())
        for nextVert in currentVert.getConnections():
          newCost = currentVert.getWeight(nextVert)
          if PriorityQueueEntry(0,nextVert) in pq and \
             newCost<nextVert.getDistance():
              nextVert.setPred(currentVert)
              nextVert.setDistance(newCost)
              pq.decreaseKey(PriorityQueueEntry(newCost, nextVert))


g = Graph()

graphFile = open("cities.txt")
cityList = []
print("Map graph:")
for cityLine in graphFile:
    currentCityList = cityLine.strip().split(':')
    city = currentCityList[0]
    x = eval(currentCityList[1])
    y = eval(currentCityList[2])
    cityList.append((city, x, y))
    print("city", city, "at x =", x, "y =", y)

for c1 in cityList:
    for c2 in cityList:
        if c1[0] != c2[0]:
            distance = ((c1[1]-c2[1])**2 + (c1[2]-c2[2])**2)**0.5
            g.addEdge(c1[0], c2[0], distance)
            g.addEdge(c2[0], c1[0], distance)
for v in g:
    print (v)

prim(g,g.getVertex("a"))
```

```
print("\nMinimum Spanning Tree Edges:")
for v in g:
    if v.getPred() != None:
        print(v.getPred().getId(), "to", v.getId())
```

c) When used for Prim's algorithm what type of objects are the `vals` compared by `__eq__`? Vertex objects

d) What changes to the Graph and Vertex classes need to be made? None

[0, V1, x, V, x]

e) Complete the `__contains__` and `decreaseKey` methods.

Queue
PriorityQueue
Object

```
class BinHeap:
    def __init__(self):
        self.heapList = [0]
        self.currentSize = 0
        foundIndex = 0
    def buildHeap(self,alist):
        i = len(alist) // 2
        self.currentSize = len(alist)
        self.heapList = [0] + alist[:]
        while (i > 0):
            self.percDown(i)
            i = i - 1

    def percDown(self,i):
        while (i * 2) <= self.currentSize:
            mc = self.minChild(i)
            if self.heapList[i] > self.heapList[mc]:
                tmp = self.heapList[i]
                self.heapList[i] = self.heapList[mc]
                self.heapList[mc] = tmp
            i = mc

    def minChild(self,i):
        if i * 2 + 1 > self.currentSize:
            return i * 2
        else:
            if self.heapList[i*2] < self.heapList[i*2+1]:
                return i * 2
            else:
                return i * 2 + 1

    def percUp(self,i):
        while i // 2 > 0:
            if self.heapList[i] < self.heapList[i//2]:
                tmp = self.heapList[i // 2]
                self.heapList[i // 2] = self.heapList[i]
                self.heapList[i] = tmp
            i = i // 2

    def insert(self,k):
        self.heapList.append(k)
        self.currentSize = self.currentSize + 1
        self.percUp(self.currentSize)

    def delMin(self):
        retval = self.heapList[1]
        self.heapList[1] = self.heapList[self.currentSize]
        self.currentSize = self.currentSize - 1
        self.heapList.pop()
        self.percDown(1)
        return retval

    def isEmpty(self):
        return self.currentSize == 0

    def size(self):
        return self.currentSize

    def __str__(self):
        return str(self.heapList[1:])
```

```
def __contains__(self, ~~value~~ targetItem):
    for index in range(1, len(self.heapList)):
        if targetItem == self.heapList[index]:
            self.foundIndex = index
            return True
    return False

def decreaseKey(self, decreasedValue):   PriorityQueue Entry
    """Precondition: decreasedValue
       not in heap already"""
    if decreasedValue in self:
        raise(ValueError, "  ")
    self.heapList[foundIndex] = decreasedValue
    self.percUp(foundIndex)
```

See attached

```
# Bradley N. Miller, David L. Ranum
# Introduction to Data Structures and Algorithms in Python
# Copyright 2005
# Modified by Mark Fienup 2016 to include a __contains__ and decreaseKey
method
import unittest

# this heap takes key value pairs, we will assume that the keys are integers
class BinHeap:
    def __init__(self):
        self.heapList = [0]
        self.currentSize = 0
        foundIndex = 0

    def buildHeap(self,alist):
        i = len(alist) // 2
        self.currentSize = len(alist)
        self.heapList = [0] + alist[:]
        while (i > 0):
            self.percDown(i)
            i = i - 1

    def percDown(self,i):
        while (i * 2) <= self.currentSize:
            mc = self.minChild(i)
            if self.heapList[i] > self.heapList[mc]:
                tmp = self.heapList[i]
                self.heapList[i] = self.heapList[mc]
                self.heapList[mc] = tmp
            i = mc

    def minChild(self,i):
        if i * 2 + 1 > self.currentSize:
            return i * 2
        else:
            if self.heapList[i * 2] < self.heapList[i * 2 + 1]:
                return i * 2
            else:
                return i * 2 + 1

    def percUp(self,i):
        while i // 2 > 0:
            if self.heapList[i] < self.heapList[i//2]:
               tmp = self.heapList[i // 2]
               self.heapList[i // 2] = self.heapList[i]
               self.heapList[i] = tmp
            i = i // 2

    def insert(self,k):
        self.heapList.append(k)
        self.currentSize = self.currentSize + 1
```

```
        self.percUp(self.currentSize)

    def delMin(self):
        retval = self.heapList[1]
        self.heapList[1] = self.heapList[self.currentSize]
        self.currentSize = self.currentSize - 1
        self.heapList.pop()
        self.percDown(1)
        return retval

    def isEmpty(self):
        return self.currentSize == 0

    def size(self):
        return self.currentSize

    def __str__(self):
        return str(self.heapList[1:])

    def __contains__(self, value):
        for index in range(len(self.heapList)-1,0,-1):
            if self.heapList[index] == value:
                self.foundIndex = index
                return True
        return False

    def decreaseKey(self, decreasedValue):
        """Precondition:  decreasedValue in heap already"""
        if not decreasedValue in self:
            raise(ValueError, "decreaseKey value must be in heap")

        self.heapList[self.foundIndex] = decreasedValue
        self.percUp(self.foundIndex)



class PriorityQueueEntry:
    def __init__(self,x,y):
        self.key = x
        self.val = y

    def getKey(self):
        return self.key

    def getValue(self):
        return self.val

    def setValue(self, newValue):
        self.val = newValue
```

```
    def __lt__(self,other):
        return self.key < other.key

    def __gt__(self,other):
        return self.key > other.key

    def __eq__(self, other):
        return self.val == other.val

    def __hash__(self):
        return self.key

class TestBinHeap(unittest.TestCase):
    def setUp(self):
        self.theHeap = BinHeap()
        self.theHeap.insert(PriorityQueueEntry(5,'a'))

        self.theHeap.insert(PriorityQueueEntry(9,'d'))
        self.theHeap.insert(PriorityQueueEntry(1,'x'))
        self.theHeap.insert(PriorityQueueEntry(2,'y'))
        self.theHeap.insert(PriorityQueueEntry(3,'z'))

    def testInsert(self):
        assert self.theHeap.currentSize == 5

    def testDelmin(self):
        assert self.theHeap.delMin().getValue() == 'x'
        assert self.theHeap.delMin().getValue()  == 'y'
        assert self.theHeap.delMin().getValue()  == 'z'
        assert self.theHeap.delMin().getValue()  == 'a'

    def testMixed(self):
        myHeap = BinHeap()
        myHeap.insert(9)
        myHeap.insert(1)
        myHeap.insert(5)
        assert myHeap.delMin() == 1
        myHeap.insert(2)
        myHeap.insert(7)
        assert myHeap.delMin() == 2
        assert myHeap.delMin() == 5

    def testDupes(self):
        myHeap = BinHeap()
        myHeap.insert(9)
        myHeap.insert(1)
        myHeap.insert(8)
        myHeap.insert(1)
        assert myHeap.currentSize == 4
        assert myHeap.delMin() == 1
        assert myHeap.delMin() == 1
```

```
        assert myHeap.delMin() == 8

    def testBuildHeap(self):
        myHeap = BinHeap()
        myHeap.buildHeap([9,5,6,2,3])
        f = myHeap.delMin()
        #print("f = ", f)
        assert f == 2
        assert myHeap.delMin() == 3
        assert myHeap.delMin() == 5
        assert myHeap.delMin() == 6
        assert myHeap.delMin() == 9

if __name__ == '__main__':
##    d = {}
##    d[PriorityQueueEntry(1,'z')] = 10
##    unittest.main()
    suite = unittest.makeSuite(TestBinHeap)
    unittest.TextTestRunner().run(suite)
```