Greining Reiknirita

TÖL403G

Skilaverkefni 3

Verkefnishöfundar:
Guðmundur Már Gunnarsson
Skarphéðinn Þórðarsson
Sigurður Skúli Sigurgeirsson

Kennari:
Páll Melsted

6. apríl 2014

1 Keyrsla og virkni

1.1 Keyrsla

Öll forritun fór fram í Python og notast við Python version 2.7 fyrir Windows við keyrslu þess. Forritið er aðgengilegt á https://github.com/gudmundurmar/SKSKG/V3

Keyrsla: `$ Python verk3.py < 10.in | diff -w 10.out -`

Til þess að mæla timann var notast við time í Python til þess að mæla tímann

```
start\_time = time.time()
   """Forrit keyrt"""
print time.time() - start\_time, "seconds"
```

1.2 Hugmynd fyrir N-1 MST

Þegar við losum okkur við einn legg í MST þá endum við í raunin með tvo ótengda samhengisþætti. Til þess að finna MST fyrir N-1 trén verðum við því að fara í gegnum notSpan sem er raðaður listi af leggjum sem eru ekki í spantrénu og finnum minnsta legginn sem tengir saman samhengisþættina saman.

2 Keyrslutími

2.1 Tími til að finna næstminnsta spantré fyrir hvern legg

Aðferðin doubleBFS skoðar í mesta lagi alla hnúta í netinu einu sinni og tekur því O(V). Það að finna rétta hnútinn sem tengir saman samhengisþættina tekur O(V*(V-E)). Það þarf að keyra í mesta lagi V-E sinnum í gegnum forlykkju því það er lengdin á notSpan og lengsta mögulega lengdin á context (samhengisþættinum) er V/2 og af því að við gerum "if notSpan[cur][1] in context" sem tekur O(V). Lykkjuna til að finna N-1 spantrén keyrum við O(V) sinnum svo að tímaflækjan fyrir næstminnsta spantré fyrir hvern legg er O(V*(V+V*(E-V))). Þar sem að þetta er mesta flækjustig forritsins verður þetta einnig timaflækja forritsins

Heildartímaflækja forrits: O(V*(V+V*(E-V)))

Inntak	Keyrslutími
10.in	0.00100016593933 sek
100.in	0.0019998550415 sek
1k.in	0.942054033279 sek
10k	117.254707098 sek
100k	Óstaðfestur (keyboard interrupt eftir 40 mín)

2.2 Meginklasinn - Verk3

```
# -*- coding: cp1252 -*-
from priority_dict import priority_dict
from bisect import bisect_left
import time
import sys
```

```
class verk3:

    """
    FG:
    notSpan er listi sem inniheldur þá leggi sem eru ekki í minnsta spantrénu
    Span er listi inniheldur þá leggi sem eru í minnsta span trénu
    wholeNet er listi sem inniheldur alla leggi í netinu
    tree er listi sem inniheldur alla hnúta sem eru í minnsta span trénu
    """


    def __init__(self):
        self.notSpan = []
        self.Span = []
        self.wholeNet = []
        self.tree = []

    """
    Fyrir:  Ekkert
    Eftir:  Búið er að prenta út vigtina á minnsta spantrénu og næstminnsta
            spantré fyrir hvern legg
    """
    def inputToDict(self):
        ...


    """
    Fyrir: G er net hnúta og r er upphafshnútur í netinu
    Eftir: Búið er að finna minnsta spantré fyrir G og vigt þess
    """
    def MSTPRIM(self,G,r):
        ...


    """
    Fyrir: G er net hnúta og w er vigt minnsta spantrés
    Eftir: Búið er að finna og prenta út N-1 minnstu spantrén.
    """
    def NotMinPRIM(self,G,w):

        ...

"""
Fyrir: G er net hnúta, u er hnútur og v er foreldi hnútsins í minnsta spantrénu
Eftir: Búið er að skipta minnsta  spantrénu upp í tvo samhengisþætti og finna minni samhengisþáttinn
"""

def doubleBFS(G,u,v):
    ...


def binary_search(a, x, lo=0, hi=None):   # can't use a to specify default for hi
```

```
    hi = hi if hi is not None else len(a) # hi defaults to len(a)
    pos = bisect_left(a,x,lo,hi)          # find insertion position
    return (pos if pos != hi and a[pos] == x else -1) # don't walk off the end


if __name__ == '__main__':
    V3 = verk3()
    V3.inputToDict()
```

2.3 Priority dict

```
from heapq import heapify, heappush, heappop

class priority_dict(dict):
    """Dictionary that can be used as a priority queue.

    Keys of the dictionary are items to be put into the queue, and values
    are their respective priorities. All dictionary methods work as expected.
    The advantage over a standard heapq-based priority queue is
    that priorities of items can be efficiently updated (amortized O(1))
    using code as 'thedict[item] = new_priority.'

    The 'smallest' method can be used to return the object with lowest
    priority, and 'pop_smallest' also removes it.

    The 'sorted_iter' method provides a destructive sorted iterator.
    """

    def __init__(self, *args, **kwargs):
        ...

    def _rebuild_heap(self):
        ...
    def smallest(self):
        """Return the item with the lowest priority.

        Raises IndexError if the object is empty.
        """
            ...
    def pop_smallest(self):
        """Return the item with the lowest priority and remove it.

        Raises IndexError if the object is empty.
        """
            ...

    def __setitem__(self, key, val):
        # We are not going to remove the previous value from the heap,
        # since this would have a cost O(n).

        # When the heap grows larger than 2 * len(self), we rebuild it
        # from scratch to avoid wasting too much memory.
            ...
```

```
def update(self, *args, **kwargs):
    # Reimplementing dict.update is tricky -- see e.g.
    # http://mail.python.org/pipermail/python-ideas/2007-May/000744.html
    # We just rebuild the heap from scratch after passing to super.


def sorted_iter(self):
    """Sorted iterator of the priority dictionary items.

    Beware: this will destroy elements as they are returned.
    """
        ...
```

3 Forritið í heild

```
# -*- coding: cp1252 -*-
from priority_dict import priority_dict
from bisect import bisect_left
import time
import sys

class verk3:

    """
    FG:
    notSpan er listi sem inniheldur þá leggi sem eru ekki í minnsta spantrénu
    Span er listi inniheldur þá leggi sem eru í minnsta span trénu
    wholeNet er listi sem inniheldur alla leggi í netinu
    tree er listi sem inniheldur alla hnúta sem eru í minnsta span trénu
    """


    def __init__(self):
        self.notSpan = []
        self.Span = []
        self.wholeNet = []
        self.tree = []

    """
    Fyrir:  Ekkert
    Eftir:  Búið er að prenta út vigtina á minnsta spantrénu og næstminnsta
            spantré fyrir hvern legg
    """
    def inputToDict(self):
        dict = {}

        length = int(sys.stdin.readline())
        file = sys.stdin.readlines()
        for i in range(0, length):
            dict[str(i)] = [float("inf"), None, []]
        for line in file:

            split = line.split(" ")
            dict[split[0]].append([split[1], split[2]])
            dict[split[1]].append([split[0], split[2]])
            self.wholeNet.append([int(split[2]),split[0],split[1], False])

        self.wholeNet.sort()
        weight = self.MSTPRIM(dict, dict['0']);
        print weight
        self.NotMinPRIM(dict, weight)



    """
```

```
    Fyrir: G er net hnúta og r er upphafshnútur í netinu
    Eftir: Búið er að finna minnsta spantré fyrir G og vigt þess
    """
    def MSTPRIM(self,G,r):

        r[0] = 0

        w = 0

        Q =  priority_dict(G);
        while Q:
            Q._rebuild_heap()

            u = Q.pop_smallest()

            if not(G[u][1] is None):
                if(int(u) < int(G[u][1])):
                    self.Span.append([int(G[u][0]), u,G[u][1], False])
                else:
                    self.Span.append([int(G[u][0]), G[u][1], u, False])
            w += int(G[u][0])

            self.tree.append(u)

            for v in range(3,len(G[u])):
                ver = G[u][v]
                if ver[0] in Q:
                    if float(ver[1]) < float(Q[ver[0]][0]):
                        Q[ver[0]][1] = u
                        G[ver[0]][1] = u

                        if float(G[u][0]) == float(0):
                            G[u][0] = float(ver[1])

                        Q[ver[0]][0] = float(ver[1])
                        G[ver[0]][0] = float(ver[1])


        for i in range(1,len(self.tree)):
            G[str(G[self.tree[i]][1])][2].append(self.tree[i])

        return w


    """
    Fyrir: G er net hnúta og w er vigt minnsta spantrés
    Eftir: Búið er að finna og prenta út N-1 minnstu spantrén.
    """
    def NotMinPRIM(self,G,w):

        self.Span.sort()
        for e in self.Span:
            res = binary_search(self.wholeNet, e)
```

```
104             if not(res == -1):
105                 self.wholeNet[res][3] = True
106 
107 
108         for e in range(0,len(self.wholeNet)):
109             cur = len(self.wholeNet)-1-e
110             if not(self.wholeNet[cur][3]):
111                 self.notSpan.append(self.wholeNet[cur])
112 
113 
114         cnt = 0
115         nrSecond = 1
116         weight = 0
117         shortest = {}
118 
119 
120         for i in range(1,len(self.tree)):
121 
122             node = self.tree[i]
123 
124             context = doubleBFS(G, node, G[node][1])
125 
126             for j in range(0,len(self.notSpan)):
127                 cur = len(self.notSpan)-1-j
128                 if self.notSpan[cur][1] in context and self.notSpan[cur][2] in context:
129                     continue
130                 elif self.notSpan[cur][1] in context or self.notSpan[cur][2] in context:
131                     weight = w-G[node][0]+self.notSpan[cur][0]
132                     break
133 
134             if(int(node) < int(G[node][1])):
135                 shortest[cnt] = [int(node), int(G[node][1]), int(weight)]
136             else:
137                 shortest[cnt] = [int(G[node][1]), int(node), int(weight)]
138             cnt += 1
139 
140 
141         Q =  priority_dict(shortest);
142 
143         while Q:
144             u = Q.smallest()
145             print str(Q[u][0])+" "+str(Q[u][1])+" "+str(Q[u][2])
146             u = Q.pop_smallest()
147 
148 
149 """
150 Fyrir: G er net hnúta, u er hnútur og v er foreldi hnútsins í minnsta spantrénu
151 Eftir: Búið er að skipta minnsta  spantrénu upp í tvo samhengisþætti og finna minni samhengisþáttinn
152 """
153 
154 def doubleBFS(G,u,v):
155     contextU = [u]
156     contextV = [v]
```

```
157
158     treeU = [u]
159     treeV = [v]
160
161     while treeU and treeV:
162         curU = treeU.pop()
163         for adjU in G[curU][2]:
164             treeU.append(adjU)
165             contextU.append(adjU)
166         curV = treeV.pop()
167         for adjV in G[curV][2]:
168             treeV.append(adjV)
169             contextV.append(adjV)
170
171
172     if len(contextU) < len(contextV):
173         return contextU
174     else:
175         return contextV
176
177
178
179 def binary_search(a, x, lo=0, hi=None):   # can't use a to specify default for hi
180     hi = hi if hi is not None else len(a) # hi defaults to len(a)
181     pos = bisect_left(a,x,lo,hi)          # find insertion position
182     return (pos if pos != hi and a[pos] == x else -1) # don't walk off the end
183
184
185 if __name__ == '__main__':
186     V3 = verk3()
187     V3.inputToDict()
```