Jessica Blasch
ECE 428 – fall 2015
Dr. Jeske
Assignment: KL output for large files

Background:

This assignment was optional for undergrad students. By this point, students were to have completed programs to run the KL algorithm on files containing information on small graphs for the class project. Undergrad students then had the option to run their programs on 3 of the benchmark files provided for this assignment.

Benchmarks:

The benchmarks could be downloaded with the following URLs:

add20 - http://staffweb.cms.gre.ac.uk/~wc06/partition/archive/add20/add20.graph uk - http://staffweb.cms.gre.ac.uk/~wc06/partition/archive/uk/uk.graph
data - http://staffweb.cms.gre.ac.uk/~wc06/partition/archive/data/data.graph

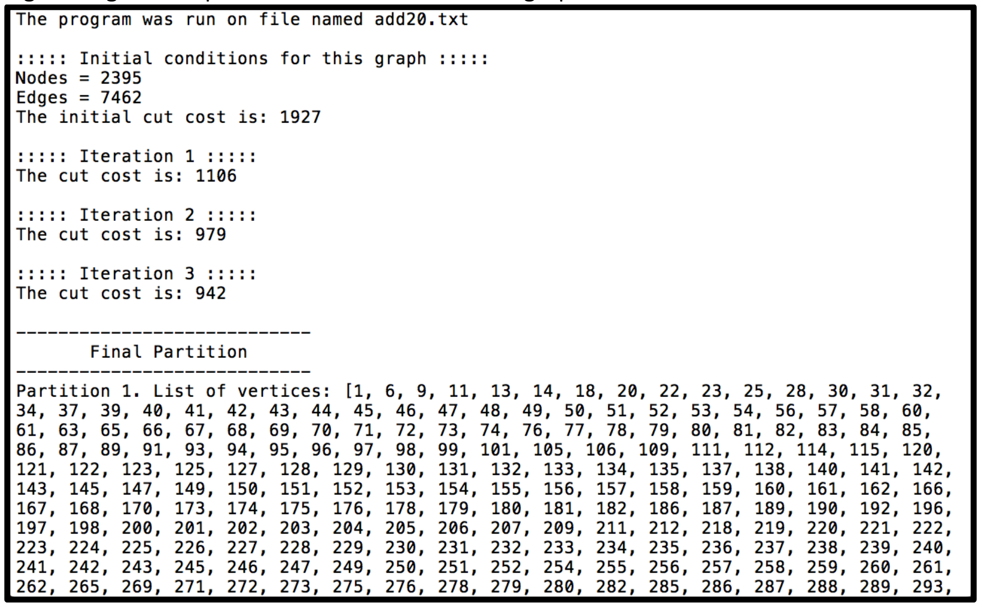
The fourth benchmark had a much higher edge count, with many high-degree nodes:

bcsstk33 - http://staffweb.cms.gre.ac.uk/~wc06/partition/archive/bcsstk33/bcsstk33.graph

Procedure:

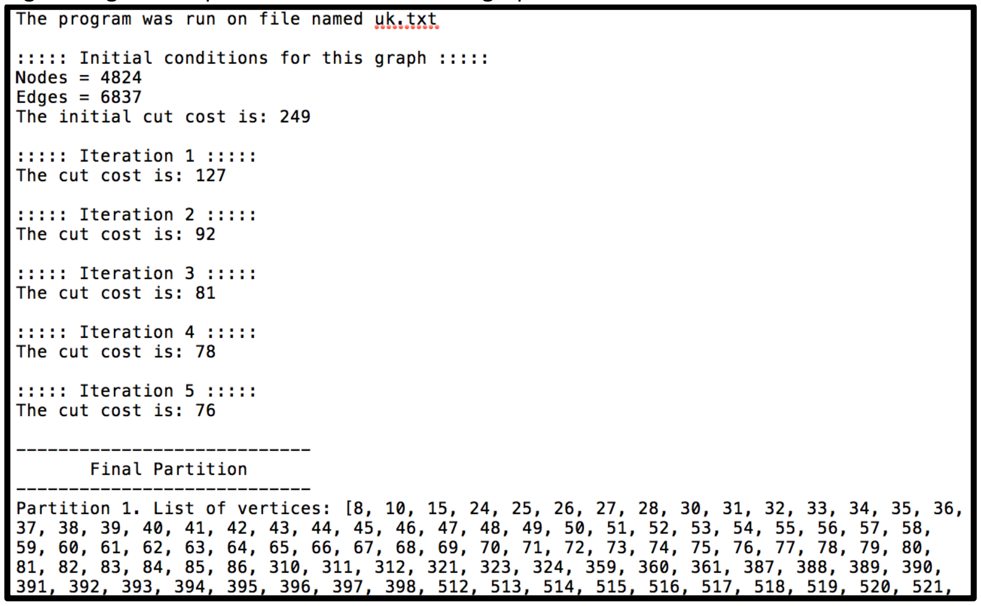
Since the benchmark files were much larger than any files run for the class project, the original code for submittal was reworked in an effort to optimize it. The first thing done was to remove as many functions as possible from within the iterative loop in the program. This necessitated creating a new function, which was then called within the loop. The new function necessitated altering variable names to avoid the issue of global shadowing. This was because the code in the new function was pulled from within the main body of the program. The reworked code was then assessed to determine what calculated items were not necessary to be returned to the loop in the main section of the program. The benchmark files were extremely large with respect to the files run for the project, so it was unrealistic to attempt hand-calculated analysis before running the program on the files.

Fig 1: Program output on benchmark 1 – add20.graph



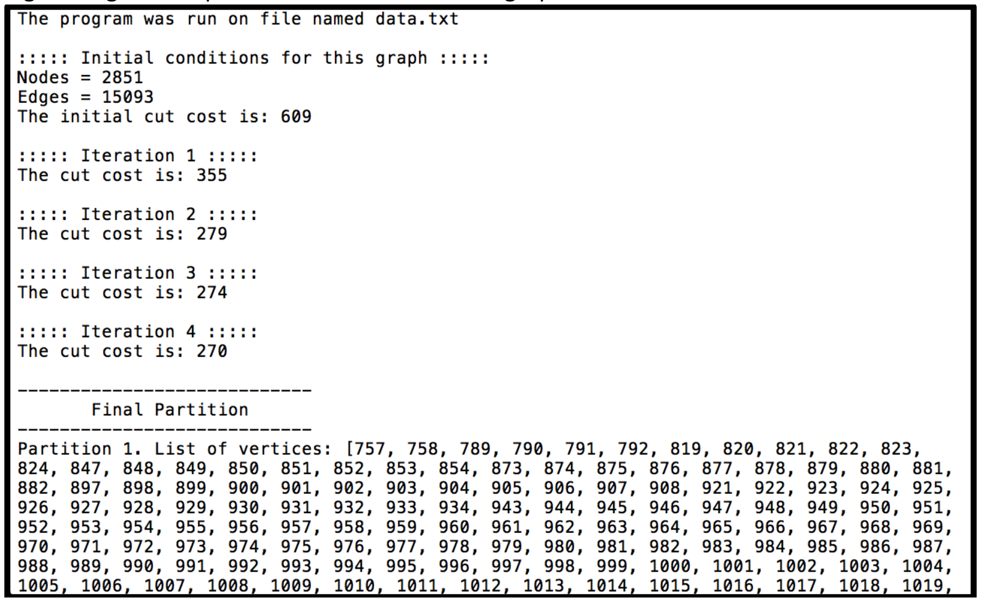
```
The program was run on file named add20.txt

::::: Initial conditions for this graph :::::
Nodes = 2395
Edges = 7462
The initial cut cost is: 1927

::::: Iteration 1 :::::
The cut cost is: 1106

::::: Iteration 2 :::::
The cut cost is: 979

::::: Iteration 3 :::::
The cut cost is: 942

----------------------------
       Final Partition
----------------------------
Partition 1. List of vertices: [1, 6, 9, 11, 13, 14, 18, 20, 22, 23, 25, 28, 30, 31, 32,
34, 37, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 56, 57, 58, 60,
61, 63, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85,
86, 87, 89, 91, 93, 94, 95, 96, 97, 98, 99, 101, 105, 106, 109, 111, 112, 114, 115, 120,
121, 122, 123, 125, 127, 128, 129, 130, 131, 132, 133, 134, 135, 137, 138, 140, 141, 142,
143, 145, 147, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 166,
167, 168, 170, 173, 174, 175, 176, 178, 179, 180, 181, 182, 186, 187, 189, 190, 192, 196,
197, 198, 200, 201, 202, 203, 204, 205, 206, 207, 209, 211, 212, 218, 219, 220, 221, 222,
223, 224, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240,
241, 242, 243, 245, 246, 247, 249, 250, 251, 252, 254, 255, 256, 257, 258, 259, 260, 261,
262, 265, 269, 271, 272, 273, 275, 276, 278, 279, 280, 282, 285, 286, 287, 288, 289, 293,
```

Fig 2: Program output on benchmark 2 – uk.graph



```
The program was run on file named uk.txt

::::: Initial conditions for this graph :::::
Nodes = 4824
Edges = 6837
The initial cut cost is: 249

::::: Iteration 1 :::::
The cut cost is: 127

::::: Iteration 2 :::::
The cut cost is: 92

::::: Iteration 3 :::::
The cut cost is: 81

::::: Iteration 4 :::::
The cut cost is: 78

::::: Iteration 5 :::::
The cut cost is: 76

----------------------------
       Final Partition
----------------------------
Partition 1. List of vertices: [8, 10, 15, 24, 25, 26, 27, 28, 30, 31, 32, 33, 34, 35, 36,
37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58,
59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80,
81, 82, 83, 84, 85, 86, 310, 311, 312, 321, 323, 324, 359, 360, 361, 387, 388, 389, 390,
391, 392, 393, 394, 395, 396, 397, 398, 512, 513, 514, 515, 516, 517, 518, 519, 520, 521,
```

Fig 3: Program output on benchmark 2 – data.graph



```
The program was run on file named data.txt

::::: Initial conditions for this graph :::::
Nodes = 2851
Edges = 15093
The initial cut cost is: 609

::::: Iteration 1 :::::
The cut cost is: 355

::::: Iteration 2 :::::
The cut cost is: 279

::::: Iteration 3 :::::
The cut cost is: 274

::::: Iteration 4 :::::
The cut cost is: 270

----------------------------
       Final Partition
----------------------------
Partition 1. List of vertices: [757, 758, 789, 790, 791, 792, 819, 820, 821, 822, 823,
824, 847, 848, 849, 850, 851, 852, 853, 854, 873, 874, 875, 876, 877, 878, 879, 880, 881,
882, 897, 898, 899, 900, 901, 902, 903, 904, 905, 906, 907, 908, 921, 922, 923, 924, 925,
926, 927, 928, 929, 930, 931, 932, 933, 934, 943, 944, 945, 946, 947, 948, 949, 950, 951,
952, 953, 954, 955, 956, 957, 958, 959, 960, 961, 962, 963, 964, 965, 966, 967, 968, 969,
970, 971, 972, 973, 974, 975, 976, 977, 978, 979, 980, 981, 982, 983, 984, 985, 986, 987,
988, 989, 990, 991, 992, 993, 994, 995, 996, 997, 998, 999, 1000, 1001, 1002, 1003, 1004,
1005, 1006, 1007, 1008, 1009, 1010, 1011, 1012, 1013, 1014, 1015, 1016, 1017, 1018, 1019,
```

Results:

This student ran into issues getting the large files to run. In Canopy, running the files was extremely slow. It took over 24 hours to run benchmark 1 (add20). After 4 days run time on benchmark 2 (data) the program had only put out that it was working on the 4th iteration and had to be stopped in order to run a program for a project in another class. It was noted that between Canopy and the python.exe, a little over 800 mB of memory was being used while processing the files and frequently the activity monitor said python was "not responding". This is what initially incited this student to rework the program.

After reworking the program, memory usage was diminished to under 500 mB, but the files were not processing any faster on this student's laptop. The first three benchmarks then attempted to be run on the school's server, walle. After 2 days, only blank output files were produced. This result was worse than what was experienced running the benchmark files on a personal laptop, so other alternatives were looked at. It was decided to use a program, downloadable on the Internet, called pypy. Within 24 hours, all of the first three benchmarks were run and output files produced. The results are as follows:

Table 1: Benchmark Output Results

Attribute	Benchmark 1 (add20)	Benchmark 2 (uk)	Benchmark 3 (data)
Nodes	2395	4824	2851
Edges	7462	6837	15093
Initial Cut Cost	1927	249	609
Iterations	3	5	4
Final Cut Cost	942	76	270

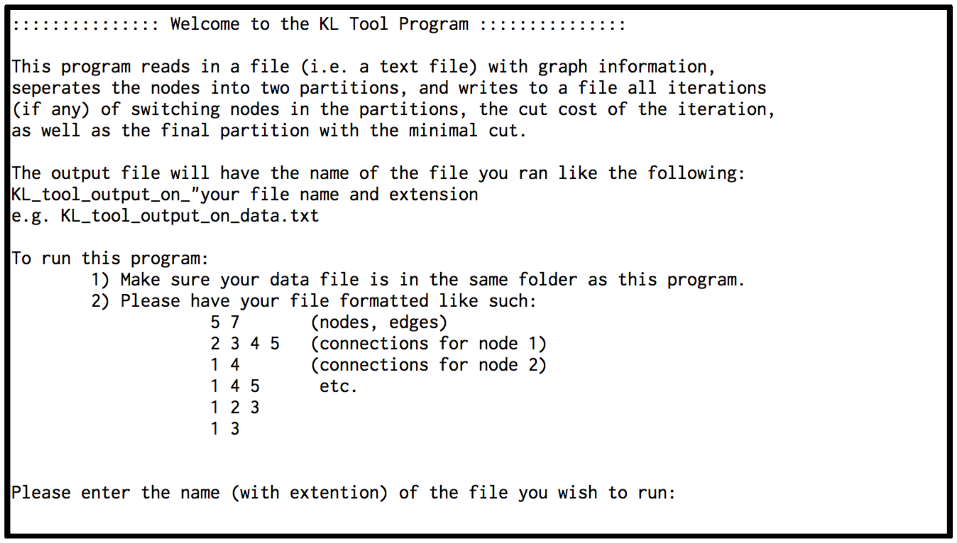
READ ME:

This section is dedicated to explaining how to use/run the program which was written for this project. It is assumed the file that will be used has been formatted appropriately, but there is instruction in the heading of the file as well. Python 2.7 was used to write the program for the project. It is not known if the program will work with other versions.

1) The source file and test files must be in the same folder for the program to run. This student used Canopy to code and run the program. However, a program like Anaconda would likely work also (launch the Spyder app). PLEASE NOTE: For large files, it is strongly encouraged to use a program like pypy, which is downloadable from the Internet. Otherwise, run time may be excessive.

2) When you launch the program, it will ask you for the input file name.

   *** Make sure you enter the **name & extension** ***



```
:::::::::::::: Welcome to the KL Tool Program ::::::::::::::

This program reads in a file (i.e. a text file) with graph information,
seperates the nodes into two partitions, and writes to a file all iterations
(if any) of switching nodes in the partitions, the cut cost of the iteration,
as well as the final partition with the minimal cut.

The output file will have the name of the file you ran like the following:
KL_tool_output_on_"your file name and extension
e.g. KL_tool_output_on_data.txt

To run this program:
        1) Make sure your data file is in the same folder as this program.
        2) Please have your file formatted like such:
                    5 7       (nodes, edges)
                    2 3 4 5   (connections for node 1)
                    1 4       (connections for node 2)
                    1 4 5      etc.
                    1 2 3
                    1 3


Please enter the name (with extention) of the file you wish to run:
```

3) Once the program is complete, it will write a file to the same folder, which contains the final information on iterations needed, cut cost, and partition for the graph with the name:

   KL_tool_output_on_(Your file's name here). (Your file's extension here)

ECE 428/528 – The future

This section of the paper is in response to future classes and a possible restriction of only allowing C++ to be used for programming. I will be speaking first person here and offer a possible solution.

There are two reasons this is an important topic to me.

One: It would have meant starting from ground zero just to be able to code, had I needed to use a program such as C++, and I would not have been able to complete the coding assignments, on time.

Two: Future students may be deterred from taking this class if there is no flexibility with the coding language. The information/concepts presented in this course are often complex, so being able to use a coding language that is comfortable to the student will help minimize difficulty when applying concepts to projects.

Problem cited:

1. Students not submitting proper instructions on how to run their code and other complications running student's programs.

Other potential issues:

1. Using an integrated IDE to run Python code slows run time down. Python is an interpreted language and so, even though student may submit proper instructions on how to run their code, the run time may be prohibitive with respect to determining if a student's program runs on large files.

2. Python has add-on modules available that students with a higher level of coding experience may use. This adds complexity for the individual running the program, if they do not already have the needed module.

Possible solution(s):

1. Restrict students from using add-on modules in programs. (e.g. "numpy", "cython", etc)
   - This will ensure that running students' programs is not complicating the instructor's job by having to go search for and procure a module that they may not have.

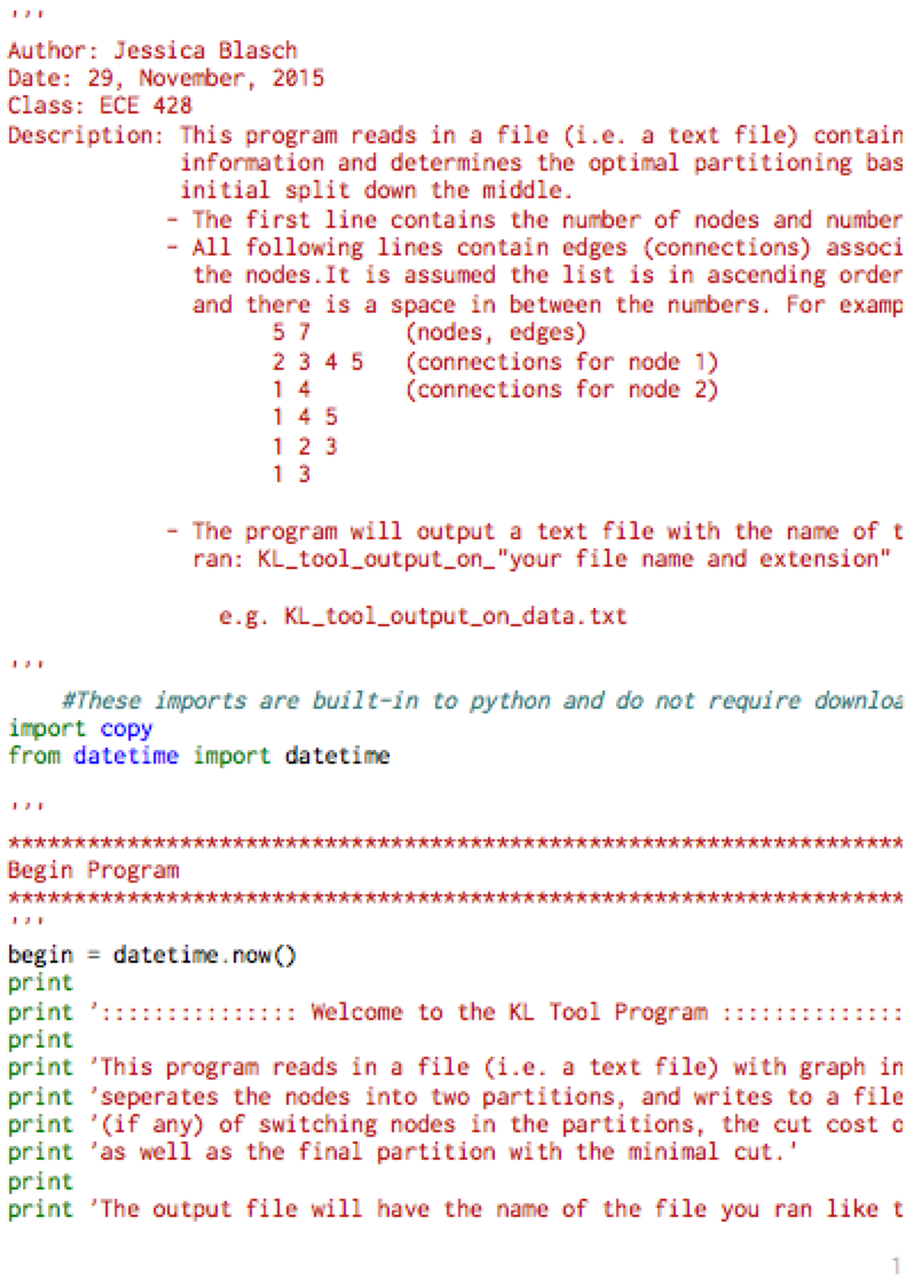
2. Use a program like pypy to run students' programs.
   - If you already have a program installed for running python on your computer (ie: Canopy), you may have to uninstall it, if you use a Mac. I ran into problems getting pypy to run on my Mac, which I suspect is because it associates Canopy with Python programs. However, I did not want to uninstall Canopy on my computer because I use it to code. However, I was able to use pypy with ease on another computing system in order to complete running the benchmarks.
   - The interface is very much like the terminal interface for shh.
   - Run time is much faster than using programs like Canopy or Anaconda.
   - For more information: http://pypy.org/

If you have any questions/concerns, please let me know.

Thank you for your time.

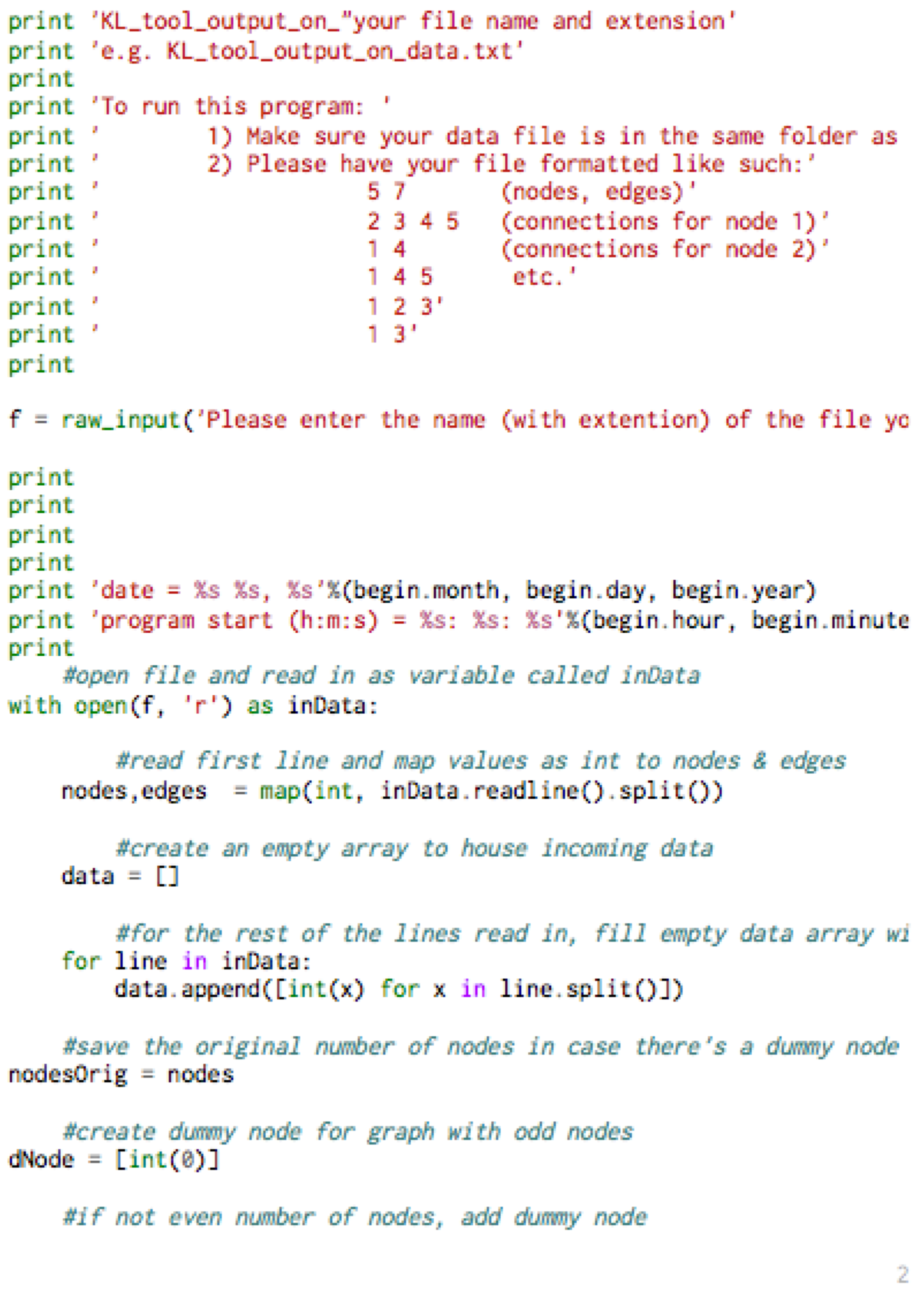
~Jessica

Appendix



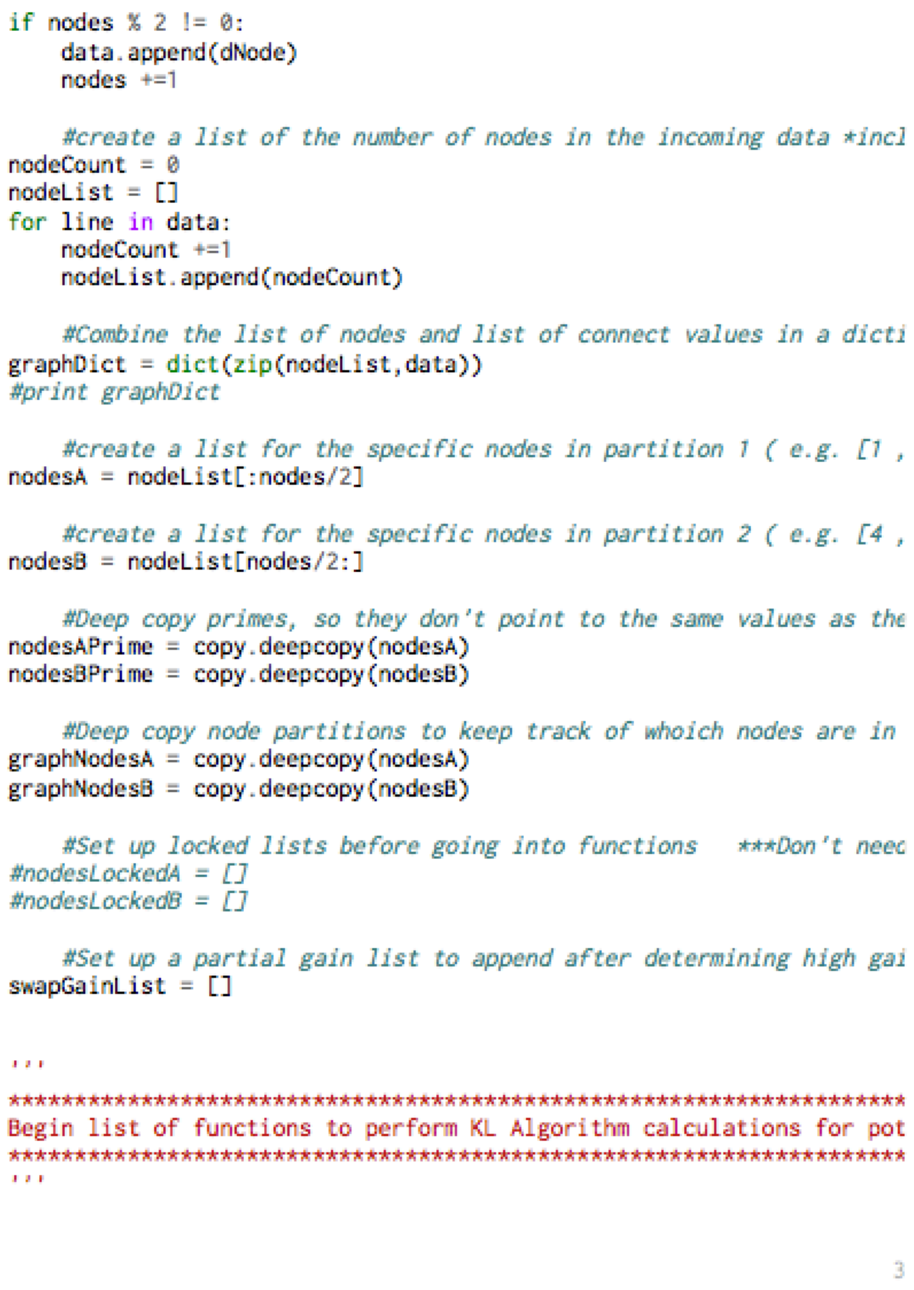
```
'''
Author: Jessica Blasch
Date: 29, November, 2015
Class: ECE 428
Description: This program reads in a file (i.e. a text file) contain
             information and determines the optimal partitioning bas
             initial split down the middle.
            - The first line contains the number of nodes and number
            - All following lines contain edges (connections) associ
              the nodes.It is assumed the list is in ascending order
              and there is a space in between the numbers. For examp
                    5 7        (nodes, edges)
                    2 3 4 5    (connections for node 1)
                    1 4        (connections for node 2)
                    1 4 5
                    1 2 3
                    1 3

            - The program will output a text file with the name of t
              ran: KL_tool_output_on_"your file name and extension"

                e.g. KL_tool_output_on_data.txt

'''
    #These imports are built-in to python and do not require downloa
import copy
from datetime import datetime

'''
*********************************************************************
Begin Program
*********************************************************************
'''

begin = datetime.now()
print
print '::::::::::::::: Welcome to the KL Tool Program :::::::::::::
print
print 'This program reads in a file (i.e. a text file) with graph in
print 'seperates the nodes into two partitions, and writes to a file
print '(if any) of switching nodes in the partitions, the cut cost o
print 'as well as the final partition with the minimal cut.'
print
print 'The output file will have the name of the file you ran like t
```

1



```
print 'KL_tool_output_on_"your file name and extension'
print 'e.g. KL_tool_output_on_data.txt'
print
print 'To run this program: '
print '        1) Make sure your data file is in the same folder as
print '        2) Please have your file formatted like such:'
print '                    5 7         (nodes, edges)'
print '                    2 3 4 5     (connections for node 1)'
print '                    1 4         (connections for node 2)'
print '                    1 4 5        etc.'
print '                    1 2 3'
print '                    1 3'
print

f = raw_input('Please enter the name (with extention) of the file yo

print
print
print
print
print 'date = %s %s, %s'%(begin.month, begin.day, begin.year)
print 'program start (h:m:s) = %s: %s: %s'%(begin.hour, begin.minute
print
    #open file and read in as variable called inData
with open(f, 'r') as inData:

        #read first line and map values as int to nodes & edges
    nodes,edges  = map(int, inData.readline().split())

        #create an empty array to house incoming data
    data = []

        #for the rest of the lines read in, fill empty data array wi
    for line in inData:
        data.append([int(x) for x in line.split()])

    #save the original number of nodes in case there's a dummy node
nodesOrig = nodes

    #create dummy node for graph with odd nodes
dNode = [int(0)]

    #if not even number of nodes, add dummy node
```

2



```
if nodes % 2 != 0:
    data.append(dNode)
    nodes +=1

    #create a list of the number of nodes in the incoming data *incl
nodeCount = 0
nodeList = []
for line in data:
    nodeCount +=1
    nodeList.append(nodeCount)

    #Combine the list of nodes and list of connect values in a dicti
graphDict = dict(zip(nodeList,data))
#print graphDict

    #create a list for the specific nodes in partition 1 ( e.g. [1 ,
nodesA = nodeList[:nodes/2]

    #create a list for the specific nodes in partition 2 ( e.g. [4 ,
nodesB = nodeList[nodes/2:]

    #Deep copy primes, so they don't point to the same values as the
nodesAPrime = copy.deepcopy(nodesA)
nodesBPrime = copy.deepcopy(nodesB)

    #Deep copy node partitions to keep track of whoich nodes are in
graphNodesA = copy.deepcopy(nodesA)
graphNodesB = copy.deepcopy(nodesB)

    #Set up locked lists before going into functions   ***Don't neec
#nodesLockedA = []
#nodesLockedB = []

    #Set up a partial gain list to append after determining high gai
swapGainList = []


'''
**********************************************************************
Begin list of functions to perform KL Algorithm calculations for pot
**********************************************************************
'''
```

3

```
'''
    Function to compare the list of nodes to get the cut cost
'''
def graph_cut_cost(gNodeA, gNodeB, gDict):
    total = 0

    for key in gDict.keys():
        for x in gNodeB:
            if x == key:
                for y in gNodeA:
                    if y in gDict[key]:
                        total +=1

    return total


'''
    Function to calculate internal and external cut costs
'''
def cut_cost(nPrime, gNodeList, gDict):
    connectsList = []
    costList = []

        #Make a list of connect values for nList2
    for node in gNodeList:
        for key in gDict.keys():
            if node == key:
                connectsList.append(gDict[key])

        #Compare nList1 with connect values list for nList2
    for node in nPrime:
        total = 0
        for line in connectsList:
            for x in line:
                if node == x:
                    total +=1
        costList.append(total)

    return costList


'''
    Function to calculate D-Values
```

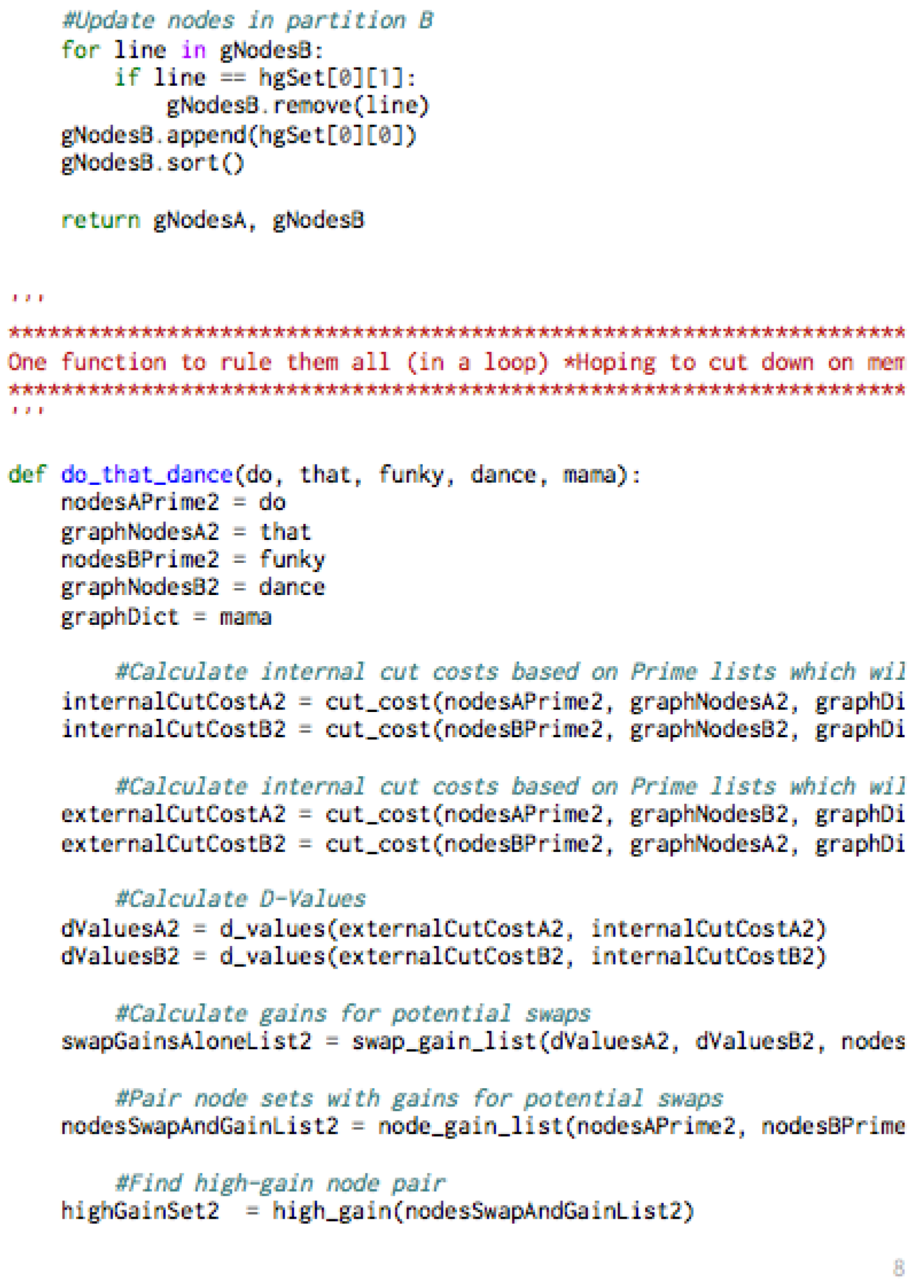
```
'''
def d_values(extCutCostList, intCutCostList):
    Lmerge = [x - y for x, y in zip(extCutCostList, intCutCostList)]

    return Lmerge


'''
    Function to caluclate gains -no node pairs attached yet
'''
def swap_gain_list(dValA, dValB, nListA, nListB, gDict):
    connectsList = []
    addDABList = []
    cABList = []

        #Make a list of connect values for nListB
    for node in nListB:
        for key in gDict.keys():
            if node == key:
                connectsList.append(gDict[key])

        #Make a list of (Da + Db) values
    for dA in dValA:
        for dB in dValB:
            dAplusB = dA + dB
            addDABList.append(dAplusB)

        #Calculate cAB connections
    for node in nListA:
        for line in connectsList:
            total = 0
            for x in line:
                if node == x:
                    total += 1
            cAB = 2*total
            cABList.append(cAB)

        #Calculate gains gab = Da + Db - 2cab
    Lmerge = [x - y for x, y in zip(addDABList, cABList)]

    return Lmerge
```

5

```
'''
    Function to pair potential node swap pairs with associated gain
'''
def node_gain_list(nListA, nListB, gList):
    nAnB = []
    nAnBList = []

    for x in nListA:
        for y in nListB:
            nAnB = [x] + [y]
            nAnBList.append(nAnB)

    Lmerge = zip(nAnBList, gList)

    return Lmerge


'''
    Function to determine the highest gain and node pair of valid no
'''
def high_gain(Lmerge):
    gain = -10000
    hiGain = -10000
    hiGainSet = []

    for line in Lmerge:
        if line[0] not in nodesBPrime:
            nAB = line[0]
        gain = line[1]

        if gain > hiGain:
            hiGain = gain
            hinAB = nAB
            hiGainSet = [hinAB] + [hiGain]

    return  hiGainSet


'''
    Function to create altered Prime lists and get Locked nodes
'''

def new_node_lists(nListA, nListB, hgSet):
```

6

```
    nAPrime = []
    nBPrime = []
    #nLockedA = [] #As it turns out, we don't really need to keep tr
    #nLockedB = [] #The reason being that the prime lists delete loc
                  #they are not used in calculations.

    a2bSwap = hgSet[0][0] #node from A moving to B
    b2aSwap = hgSet[0][1] #node from B moving to A

    #Aprime
    for node in nListA:
        if node != a2bSwap:
            nAPrime.append(node)
    nAPrime.sort()

    #A-Locked
    #nLockedA.append(b2aSwap)

    #BPrime
    for node in nListB:
        if node != b2aSwap:
            nBPrime.append(node)
    nBPrime.sort()

    #B-Locked
    #nLockedB.append(a2bSwap)

    return nAPrime, nBPrime#, nLockedA, nLockedB


'''
    Function to update which nodes are in each partition
'''
def update_partition(hgSet, gNodesA, gNodesB ):

    #Update nodes in partition A
    for line in gNodesA:
        if line == hgSet[0][0]:
            gNodesA.remove(line)

    gNodesA.append(hgSet[0][1])
    gNodesA.sort()
```



```
    #Update nodes in partition B
    for line in gNodesB:
        if line == hgSet[0][1]:
            gNodesB.remove(line)
    gNodesB.append(hgSet[0][0])
    gNodesB.sort()

    return gNodesA, gNodesB


'''
**********************************************************************
One function to rule them all (in a loop) *Hoping to cut down on mem
**********************************************************************
'''

def do_that_dance(do, that, funky, dance, mama):
    nodesAPrime2 = do
    graphNodesA2 = that
    nodesBPrime2 = funky
    graphNodesB2 = dance
    graphDict = mama

        #Calculate internal cut costs based on Prime lists which wil
    internalCutCostA2 = cut_cost(nodesAPrime2, graphNodesA2, graphDi
    internalCutCostB2 = cut_cost(nodesBPrime2, graphNodesB2, graphDi

        #Calculate internal cut costs based on Prime lists which wil
    externalCutCostA2 = cut_cost(nodesAPrime2, graphNodesB2, graphDi
    externalCutCostB2 = cut_cost(nodesBPrime2, graphNodesA2, graphDi

        #Calculate D-Values
    dValuesA2 = d_values(externalCutCostA2, internalCutCostA2)
    dValuesB2 = d_values(externalCutCostB2, internalCutCostB2)

        #Calculate gains for potential swaps
    swapGainsAloneList2 = swap_gain_list(dValuesA2, dValuesB2, nodes

        #Pair node sets with gains for potential swaps
    nodesSwapAndGainList2 = node_gain_list(nodesAPrime2, nodesBPrime

        #Find high-gain node pair
    highGainSet2  = high_gain(nodesSwapAndGainList2)
```

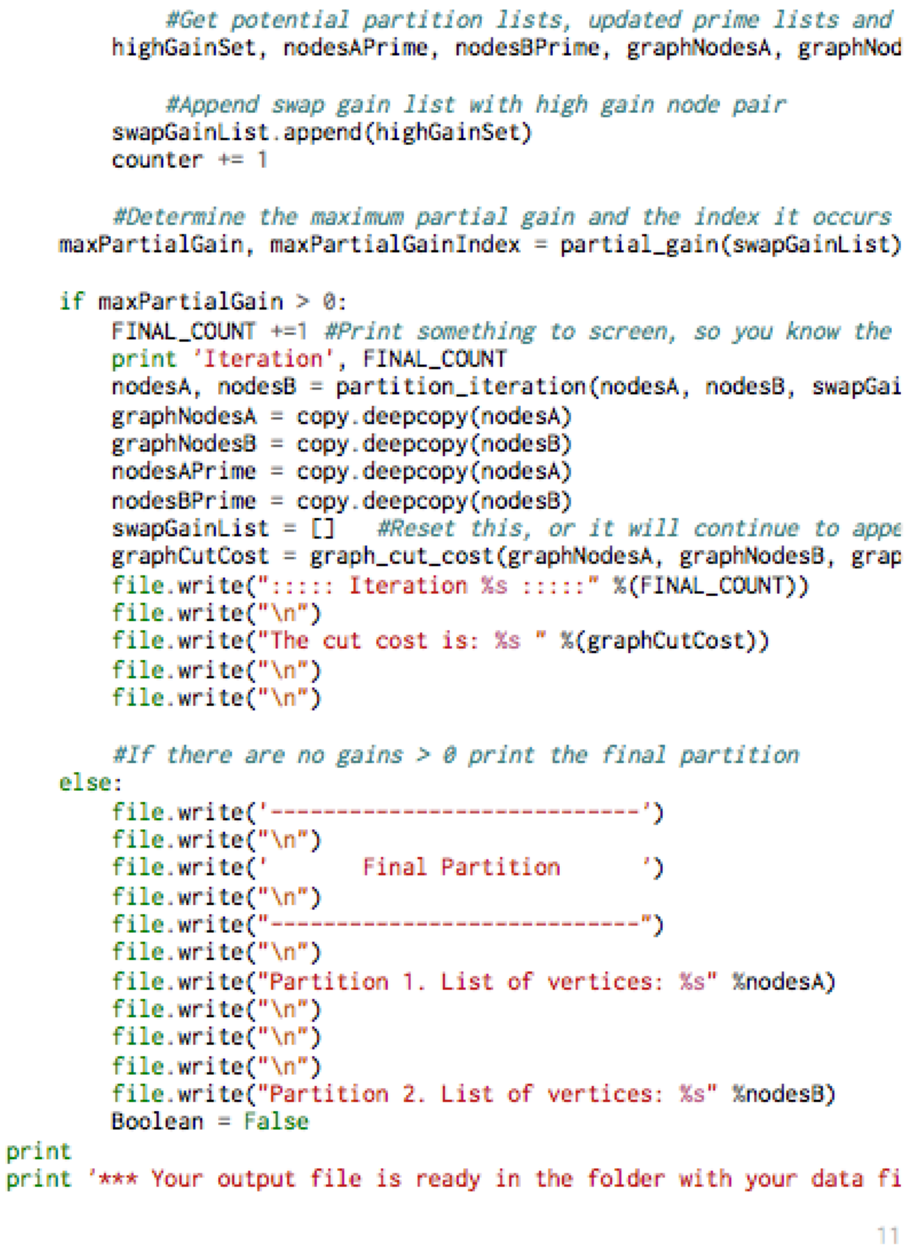
8

```
        #Delete high gain node pair from respective prime list
    nodesAPrime2, nodesBPrime2 = new_node_lists(nodesAPrime2, nodesB

        #Update nodes in partition A and B
    graphNodesA2, graphNodesB2 = update_partition(highGainSet2, grap

        #Only return what is needed, so the program doesn't use exce
    return highGainSet2, nodesAPrime2, nodesBPrime2, graphNodesA2, g


'''
********************************************************************
Begin list of functions to determine KL Algorithm real swaps and prc
********************************************************************
'''

        #Function to take the list of potential gains and calculate
def partial_gain(sGList):
    maxPGain = -10000
    pgTotal = 0
    maxPGSwapIndex = 0

    c=0
    for line in swapGainList:
        pgTotal+=sGList[c][1]
        if pgTotal> maxPGain:
            maxPGain=pgTotal #max partial gain in list
            maxPGSwapIndex=c  #index at which max partial gain occur
        c+=1

    return maxPGain, maxPGSwapIndex


    #Function that determines the new partition after real swap
def partition_iteration(nListA, nListB, sgList, sgIndex):
    count = 0

    for n in range(0,sgIndex+1):
            #Update the partitions for a real swap
        A,B=update_partition(sgList[count], nListA, nListB)
        nListA=copy.deepcopy(A)
        nListB=copy.deepcopy(B)
```

9

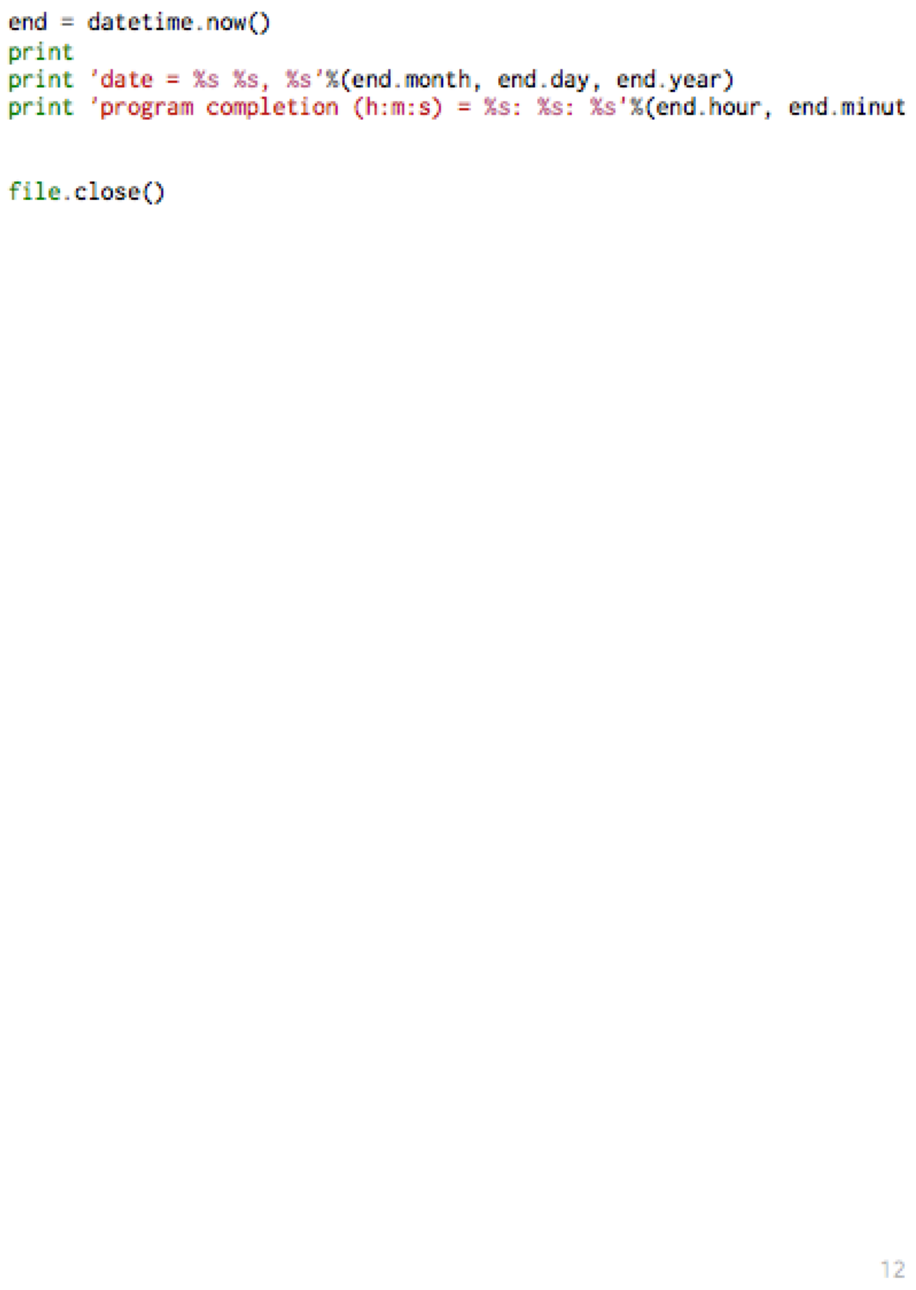
```
        count+=1

    return nListA, nListB

'''
************************************************************
Begin KL Algorithm printout to screen and file out
************************************************************
'''


    #Begin printing to file
file = open("KL_tool_output_on_%s" %f, "w")
file.write("The program was run on file named %s " %f)
file.write("\n")
file.write("\n")
file.write("::::: Initial conditions for this graph :::::")
file.write("\n")
file.write("Nodes = %s" %(nodesOrig))
file.write("\n")
file.write("Edges = %s" %(edges))
file.write("\n")

    #Get the initial graph cut cost
graphCutCost = graph_cut_cost(graphNodesA, graphNodesB, graphDict)

file.write("The initial cut cost is: %s " %graphCutCost)
file.write("\n")
file.write("\n")

FINAL_COUNT = 0

Boolean = True

    #Perform loop until partial gains < 0
while Boolean:

    lenListInitial = len(nodesAPrime)
    counter = 1

    #Perform all the initial calulations for all potential node swa
    while (counter <= lenListInitial):
```

10



```
            #Get potential partition lists, updated prime lists and
        highGainSet, nodesAPrime, nodesBPrime, graphNodesA, graphNod

            #Append swap gain list with high gain node pair
        swapGainList.append(highGainSet)
        counter += 1

        #Determine the maximum partial gain and the index it occurs
    maxPartialGain, maxPartialGainIndex = partial_gain(swapGainList)

    if maxPartialGain > 0:
        FINAL_COUNT +=1 #Print something to screen, so you know the
        print 'Iteration', FINAL_COUNT
        nodesA, nodesB = partition_iteration(nodesA, nodesB, swapGai
        graphNodesA = copy.deepcopy(nodesA)
        graphNodesB = copy.deepcopy(nodesB)
        nodesAPrime = copy.deepcopy(nodesA)
        nodesBPrime = copy.deepcopy(nodesB)
        swapGainList = []   #Reset this, or it will continue to appe
        graphCutCost = graph_cut_cost(graphNodesA, graphNodesB, grap
        file.write("::::: Iteration %s :::::" %(FINAL_COUNT))
        file.write("\n")
        file.write("The cut cost is: %s " %(graphCutCost))
        file.write("\n")
        file.write("\n")

        #If there are no gains > 0 print the final partition
    else:
        file.write('----------------------------')
        file.write("\n")
        file.write('       Final Partition      ')
        file.write("\n")
        file.write("----------------------------")
        file.write("\n")
        file.write("Partition 1. List of vertices: %s" %nodesA)
        file.write("\n")
        file.write("\n")
        file.write("\n")
        file.write("Partition 2. List of vertices: %s" %nodesB)
        Boolean = False
print
print '*** Your output file is ready in the folder with your data fi
```

11



```
end = datetime.now()
print
print 'date = %s %s, %s'%(end.month, end.day, end.year)
print 'program completion (h:m:s) = %s: %s: %s'%(end.hour, end.minut


file.close()
```

12