EC 704 - VLSI Design Automation
Assignment 4

-Shruti Masand, 181EC245

Problem Statement:

Using the netlist, generate the adjacency matrix and do the partitioning using KL method.

Solution:

Approach:

STEP 1 : Firstly, we need to convert the given netlist into a graph. To do that, I have tried to read the Netlist File line by line (using the readline() function) and tried to interpret the meaning of each line. As we have to read the gates, we take all the equations that start with "G".
Then, we can simply read LHS and RHS separately. The LHS part of that particular line will tell us about the output and the RHS part will tell us about the inputs. Hence, we create 2 lists by the name of LHS and RHS that will be used to store the values of the outputs and the inputs respectively.

These (input, output) pairs will then be appended to a list that is given as input to the Networkx library functions.
The commands written using this library called Networkx, very conveniently plot these functions into graphs with gates as nodes and directed connections shown as Edges.
For an even enriching visual aid, we can change the size and color of these edges and vertices as well. The root code for the given library ws thoroughly understood and is also given as one of the links in the references.

STEP 2 : Next step is to use that graph to form the adjacency list/matrix. The adjacency Matrix gives us the information about the nodes that are in the neighbourhood of a particular node. This can be accomplished by the use of the networkx and the numpy library. This is accomplished using a single command that is shown in the code.

STEP 3 : Now we need to apply the Kernighan-Lin Algorithm for performing efficient partitioning. This can be done by coding in accordance with the algorithms discussed in the class.

Code, implemented in Python:

```
## STEP 1 - Plotting a graph from the Netlist

import networkx as nx
import matplotlib.pyplot as plt
from math import *

import numpy as np

def read_gates(l,index):
    t = ''
    t = t + l[index]
    j = index + 1
    while ((l[j] != ' ') and (l[j] != ',') and (l[j] != ')')):
        t = t + l[j]
        j = j +1
    return(t)

# Implementation of Kernighan-Lin graph partitioning algorithm

class Vertex:
    # id, edges, partition_label
    def __init__(self, id):
        self.id = id
        self.edges = []

    def get_D_value(self):
        D_value = 0 # D = E - I

        for edge in self.edges:
            if edge.left_id == self.id:
                other_v = edge.right_v
            elif edge.right_id == self.id:
                other_v = edge.left_v

            if other_v.partition_label != self.partition_label:
                D_value += 1 # external cost
            else:
                D_value -= 1 # internal cost

        return D_value

    def add_edge(self, edge):
```

```
        # undirected graph, ignore reverse direction
        for present_edge in self.edges:
            if present_edge.left_id == edge.right_id and present_edge.right_id == edge.left_id:
                return

        self.edges.append(edge)

class Edge:
    # left_id, right_id, left_v, right_v
    def __init__(self, left_id, right_id):
        self.left_id = left_id
        self.right_id = right_id

class Graph:
    # vertices, edges
    def __init__(self, vertices, edges):
        self.vertices = vertices
        self.edges = edges

        # connect vertices and edges
        vertex_dict = {v.id: v for v in self.vertices}

        for edge in self.edges:
            edge.left_v = vertex_dict[edge.left_id]
            vertex_dict[edge.left_id].add_edge(edge)

            edge.right_v = vertex_dict[edge.right_id]
            vertex_dict[edge.right_id].add_edge(edge)

    def get_partition_cost(self):
        cost = 0

        for edge in self.edges:
            if edge.left_v.partition_label != edge.right_v.partition_label:
                cost += 1

        return cost

##STEP 3: Implementation of Kernighan-Lin graph partitioning algorithm
class KernighanLin():
    def __init__(self, graph):
        self.graph = graph

    def partition(self):
```

```
# initial partition: first half is group A, second half is B
for i in range(int(len(self.graph.vertices)/2)):
    self.graph.vertices[i].partition_label = "A"
for i in range(int(len(self.graph.vertices)/2), len(self.graph.vertices)):
    self.graph.vertices[i].partition_label = "B"

print ("Initial partition cost: " + str(self.graph.get_partition_cost()))
p = 0 # pass
total_gain = 0

# repeat until g_max <= 0
while True:
    group_a = []
    group_b = []

    for i in range(len(self.graph.vertices)):
        if self.graph.vertices[i].partition_label == "A":
            group_a.append(self.graph.vertices[i])
        elif self.graph.vertices[i].partition_label == "B":
            group_b.append(self.graph.vertices[i])

    D_values = {v.id: v.get_D_value() for v in self.graph.vertices}
    gains = [] # [ ([a, b], gain), ... ]

    # while there are unvisited vertices
    for _ in range(int(len(self.graph.vertices)/2)):

        # choose a pair that maximizes gain
        max_gain = -1 * float("inf") # -infinity
        pair = []

        for a in group_a:
            for b in group_b:
                c_ab = len(set(a.edges).intersection(b.edges))
                gain = D_values[a.id] + D_values[b.id] - (2 * c_ab)

                if gain > max_gain:
                    max_gain = gain
                    pair = [a, b]

        # mark that pair as visited
        a = pair[0]
        b = pair[1]
        group_a.remove(a)
```

```
                group_b.remove(b)
                gains.append([[a, b], max_gain])

                # update D_values of other unvisited nodes connected to a and b, as if a and b are swapped
                for x in group_a:
                    c_xa = len(set(x.edges).intersection(a.edges))
                    c_xb = len(set(x.edges).intersection(b.edges))
                    D_values[x.id] += 2 * (c_xa) - 2 * (c_xb)

                for y in group_b:
                    c_yb = len(set(y.edges).intersection(b.edges))
                    c_ya = len(set(y.edges).intersection(a.edges))
                    D_values[y.id] += 2 * (c_yb) - 2 * (c_ya)

            # find j that maximizes the sum g_max
            g_max = -1 * float("inf")
            jmax = 0
            for j in range(1, len(gains) + 1):
                g_sum = 0
                for i in range(j):
                    g_sum += gains[i][1]

                if g_sum > g_max:
                    g_max = g_sum
                    jmax = j

            if g_max > 0:
                # swap in graph
                for i in range(jmax):
                    # find vertices and change their partition_label
                    for v in self.graph.vertices:
                        if v.id == gains[i][0][0].id:
                            v.partition_label = "B"
                        elif v.id == gains[i][0][1].id:
                            v.partition_label = "A"

                p += 1
                total_gain += g_max
                print ("Pass: " + str(p) + "\t\t\tGain: " + str(g_max))
            else: break

        print ("Total passes: " + str(p) + "\t\tTotal gain: " + str(total_gain) + "\t\tFinal partition cost: " + str(self.graph.get_partition_cost()) )
```

##STEP 2: Getting other information about the graph - Number of vertices, edges, connections, Adjacency List.

```
def main():
    graph = load_data('s27.bench')
    kl = KernighanLin(graph)
    kl.partition()

def load_data(filename):

    lhs = []
    rhs = []
    exray = []
    vega = []

    abc = 0

    dict = {}

    file1 = open(filename, 'r')
    Lines = file1.readlines()

    for line in Lines:
        if(len(line)>0):
            if(line[0] == 'G'):
                gate_lhs = read_gates(line, 0)
                if(dict.get(gate_lhs)):
                    gate_lhs_index = dict[gate_lhs]

                else:
                    gate_lhs_index = abc
                    dict[gate_lhs] = abc
                    abc = abc+1


                for i in range(1, len(line)):
                    if (line[i] == "G"):
                        gate_rhs = read_gates(line,i)
                        if(dict.get(gate_rhs)):
                            gate_rhs_index = dict[gate_rhs]

                        else:
                            gate_rhs_index = abc
                            dict[gate_rhs] = abc
```

```
                    abc = abc+1

                lhs.append((gate_rhs_index,gate_lhs_index))

    #To get a list of all Vertices:
                exray.append(gate_rhs)
                exray.append(gate_lhs)

    lhs.sort()

    G = nx.DiGraph()

    G.add_edges_from( lhs )

    pos = nx.spring_layout(G)

    nx.draw_networkx_nodes(G, pos, node_size = 500)
    nx.draw_networkx_edges(G, pos, edgelist = G.edges(), edge_color = 'black')
    nx.draw_networkx_labels(G, pos)
    plt.show()

    edges = []
    vertices = []
    seen_vertex_ids = []

    for elem in lhs:
        #v_list = line.split()
        left_id = int(elem[0])
        right_id = int(elem[1])

        edges.append(Edge(left_id, right_id))

        if left_id not in seen_vertex_ids:
            vertices.append(Vertex(left_id))
            seen_vertex_ids.append(left_id)

        if right_id not in seen_vertex_ids:
            vertices.append(Vertex(right_id))
            seen_vertex_ids.append(right_id)

    return Graph(vertices, edges)

if __name__ == "__main__":
    main()
```

Screenshots of the results obtained:

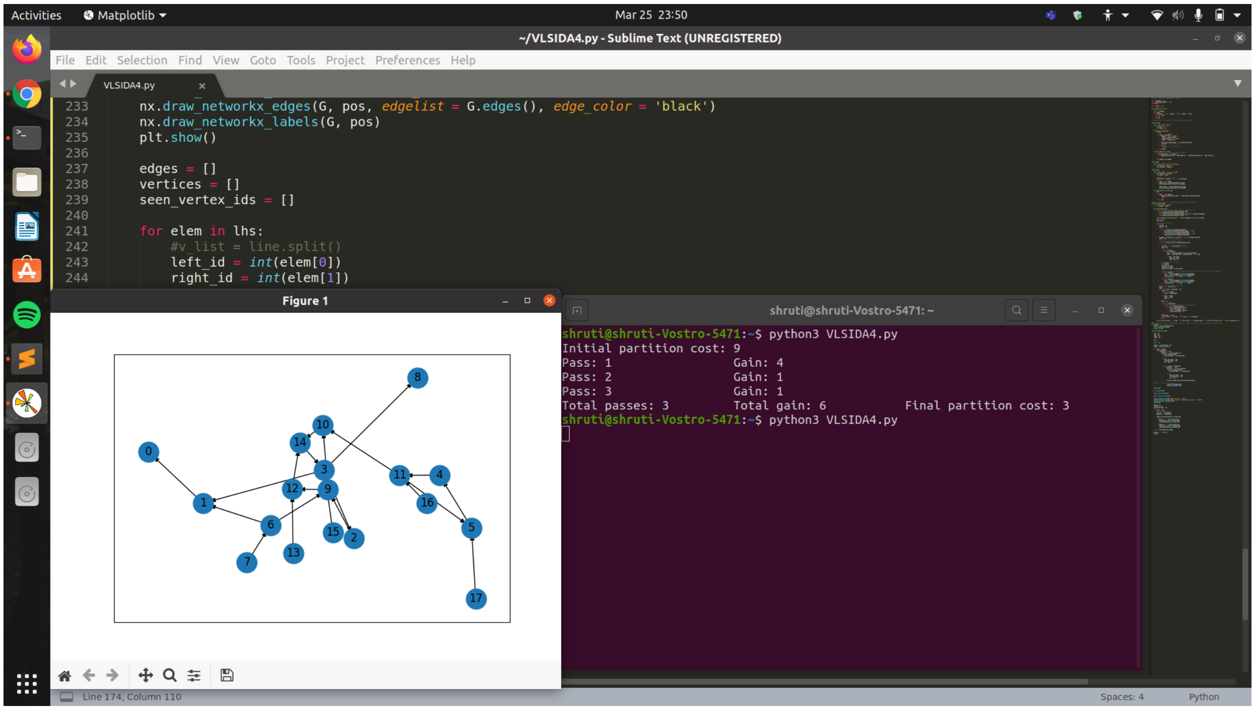

Fig. Overall Implementation of the above code

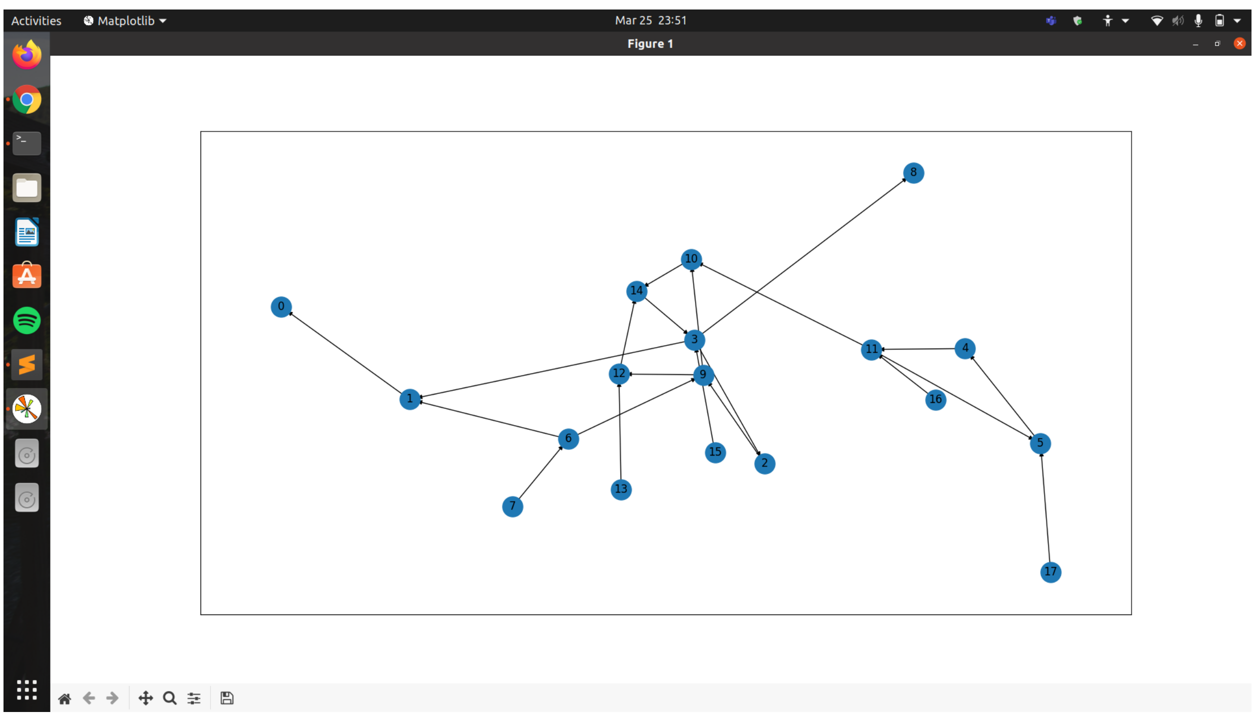

Fig. Directed Graph for s27.bench file

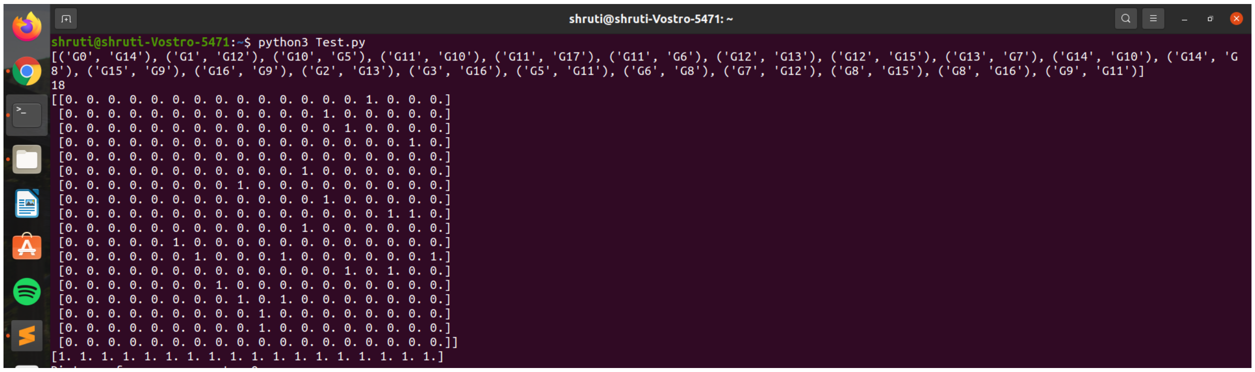

Fig.) The following results display for s27.bench file:

a) A list of all the edge connections with vertices in the order (input, output)

b) Adjacency Matrix for the above shown Directed Graph

c) List of weights(here all have weight = 1)

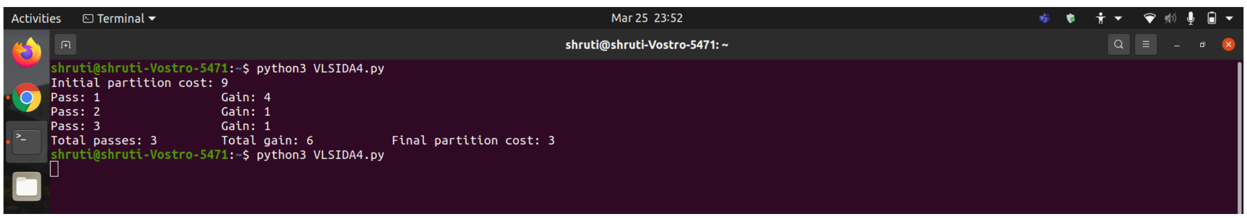

Fig.) Result for Kernighan-Lin Algorithm

Gives Final and Initial Partition Cost after all permutations

It can be seen that the Final Cost is minimised