week4prob

GVV Praneeth Reddy <EE21B048>

March 3, 2023

[1]:
```
import networkx as nx
```

1 Using Topological Order

[2]:
```
netfile = input("Enter the name of netlist file: ")
inputfile = input("Enter the name of input file: ")
```

```
Enter the name of netlist file:  c17.net
Enter the name of input file:  c17.inputs
```

Taking the names of the net and inputs files as input from the user

[3]:
```
edges=[]
nets=[]
node_attr={}
with open(netfile, "r") as file:
    data=file.readlines()
    for l in data:
        l=l.split()
        if l[1]=="inv" or l[1]=="buf":
            edges.append([l[2], l[3]])
            node_attr[l[3]] = l[1]
            nets.append(l[3])
        else:
            edges.append([l[2], l[4]])
            edges.append([l[3], l[4]])
            node_attr[l[4]] = l[1]
            nets.append(l[4])
with open(inputfile, "r") as file:
    data=file.readlines()
    niv=len(data)-1
    node_val={}
    l=data[0].split()
    for i in l:
        node_attr[i]='PI'
        nets.append(i)
    for i in range(1,len(data)):
        a=data[i].split()
```

```
        dic = {}
        for j in range(len(l)):
            dic[l[j]]=int(a[j])
        node_val[i]=dic
```

Reading the .net and .inputs file and creating the necessary dictionaries and lists

[4]:
```
g=nx.DiGraph()
print(edges, "\n")
g.add_edges_from(edges)
nx.set_node_attributes(g, node_attr, name="gateType")
print(g.nodes(data=True))
```

```
[['n_3', 'N22'], ['n_0', 'N22'], ['n_3', 'N23'], ['n_2', 'N23'], ['n_1', 'n_3'],
['N2', 'n_3'], ['n_1', 'n_2'], ['N7', 'n_2'], ['N1', 'n_0'], ['N3', 'n_0'],
['N3', 'n_1'], ['N6', 'n_1']]

[('n_3', {'gateType': 'nand2'}), ('N22', {'gateType': 'nand2'}), ('n_0',
{'gateType': 'nand2'}), ('N23', {'gateType': 'nand2'}), ('n_2', {'gateType':
'nand2'}), ('n_1', {'gateType': 'nand2'}), ('N2', {'gateType': 'PI'}), ('N7',
{'gateType': 'PI'}), ('N1', {'gateType': 'PI'}), ('N3', {'gateType': 'PI'}),
('N6', {'gateType': 'PI'})]
```

Creating the graph

[5]:
```
nl=list(nx.topological_sort(g))
print('Nodes in topological order',nl)
```

```
Nodes in topological order ['N2', 'N7', 'N1', 'N3', 'N6', 'n_0', 'n_1', 'n_3',
'n_2', 'N22', 'N23']
```

Sorting the nets in topological order

[6]:
```
def INV(x, node):
    y=list(g.predecessors(node))
    node_val[x][node] = int(not(node_val[x][y[0]]))

def BUF(x, node):
    y=list(g.predecessors(node))
    node_val[x][node]=int(node_val[x][y[0]])

def OR(x, node):
    y=list(g.predecessors(node))
    node_val[x][node]=int((node_val[x][y[0]]) or (node_val[x][y[1]]))

def AND(x, node):
    y=list(g.predecessors(node))
    node_val[x][node]=int((node_val[x][y[0]]) and (node_val[x][y[1]]))
```

```
def XOR(x, node):
    y=list(g.predecessors(node))
    node_val[x][node]=int(((node_val[x][y[0]])and(not(node_val[x][y[1]]))) or␣
 ↪((node_val[x][y[1]])and(not(node_val[x][y[0]]))))

def NOR(x, node):
    y=list(g.predecessors(node))
    node_val[x][node] = int(not((node_val[x][y[0]]) or (node_val[x][y[1]])))

def NAND(x, node):
    y=list(g.predecessors(node))
    node_val[x][node] = int(not(node_val[x][y[0]] and node_val[x][y[1]]))
```

Defining some functions to get the outputs at different types of gates

[7]:
```
def solve_top(x):
    for i in nl:
        if node_attr[i]=='inv':
            INV(x, i)
        elif node_attr[i]=='buf':
            BUF(x, i)
        elif node_attr[i]=='or2':
            OR(x, i)
        elif node_attr[i]=='and2':
            AND(x, i)
        elif node_attr[i]=='xor2':
            XOR(x, i)
        elif node_attr[i]=='nand2':
            NAND(x, i)
        elif node_attr[i]=='nor2':
            NOR(x, i)
```

Defining a function to get all the states of the nets.

[8]:
```
nets.sort()
print(nets)
for i in range(1, niv+1):
    solve_top(i)
    net_state=[]
    for j in nets:
        net_state.append(node_val[i][j])
    print(net_state)
```

```
['N1', 'N2', 'N22', 'N23', 'N3', 'N6', 'N7', 'n_0', 'n_1', 'n_2', 'n_3']
[0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0]
[0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1]
[1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1]
[0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1]
[1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1]
```

```
[1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0]
[1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1]
[1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0]
[0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0]
[0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1]
```

printing states of all the nets for each of the input vectors

2 Using Event driven evaluation

[9]:
```
netfile = input("Enter the name of netlist file: ")
inputfile = input("Enter the name of input file: ")
```

```
Enter the name of netlist file:  c17.net
Enter the name of input file:  c17.inputs
```

[10]:
```
edges=[]
queue=[]
nets=[]
node_attr={}
with open(netfile, "r") as file:
    data=file.readlines()
    for l in data:
        l=l.split()
        if l[1]=="inv" or l[1]=="buf":
            edges.append([l[2], l[3]])
            node_attr[l[3]] = l[1]
            queue.append(l[3])
            nets.append(l[3])
        else:
            edges.append([l[2], l[4]])
            edges.append([l[3], l[4]])
            node_attr[l[4]] = l[1]
            queue.append(l[4])
            nets.append(l[4])
with open(inputfile, "r") as file:
    data=file.readlines()
    niv=len(data)-1
    node_val={}
    l=data[0].split()
    for i in l:
        node_attr[i]='PI'
        queue.append(i)
        nets.append(i)
    for i in range(1,len(data)):
        a=data[i].split()
        dic = {}
        for j in range(len(l)):
```

```
            dic[l[j]]=int(a[j])
        node_val[i]=dic
    for j in queue:
        if node_attr[j]!='PI':
            node_val[1][j]=None
```

[11]:
```
g=nx.DiGraph()
print(edges, "\n")
g.add_edges_from(edges)
nx.set_node_attributes(g, node_attr, name="gateType")
print(g.nodes(data=True))
```

```
[['n_3', 'N22'], ['n_0', 'N22'], ['n_3', 'N23'], ['n_2', 'N23'], ['n_1', 'n_3'],
['N2', 'n_3'], ['n_1', 'n_2'], ['N7', 'n_2'], ['N1', 'n_0'], ['N3', 'n_0'],
['N3', 'n_1'], ['N6', 'n_1']]

[('n_3', {'gateType': 'nand2'}), ('N22', {'gateType': 'nand2'}), ('n_0',
{'gateType': 'nand2'}), ('N23', {'gateType': 'nand2'}), ('n_2', {'gateType':
'nand2'}), ('n_1', {'gateType': 'nand2'}), ('N2', {'gateType': 'PI'}), ('N7',
{'gateType': 'PI'}), ('N1', {'gateType': 'PI'}), ('N3', {'gateType': 'PI'}),
('N6', {'gateType': 'PI'})]
```

[12]:
```
def INV(x, node):
    y=list(g.predecessors(node))
    node_val[x][node] = int(not(node_val[x][y[0]]))

def BUF(x, node):
    y=list(g.predecessors(node))
    node_val[x][node]=int(node_val[x][y[0]])

def OR(x, node):
    y=list(g.predecessors(node))
    node_val[x][node]=int((node_val[x][y[0]]) or (node_val[x][y[1]]))

def AND(x, node):
    y=list(g.predecessors(node))
    node_val[x][node]=int((node_val[x][y[0]]) and (node_val[x][y[1]]))

def XOR(x, node):
    y=list(g.predecessors(node))
    node_val[x][node]=int(((node_val[x][y[0]])and(not(node_val[x][y[1]]))) or␣
 ↪((node_val[x][y[1]])and(not(node_val[x][y[0]]))))

def NOR(x, node):
    y=list(g.predecessors(node))
    node_val[x][node] = int(not((node_val[x][y[0]]) or (node_val[x][y[1]])))
```

```
def NAND(x, node):
    y=list(g.predecessors(node))
    node_val[x][node] = int(not(node_val[x][y[0]] and node_val[x][y[1]]))
```

[13]:
```
def solve_gate(x,node):
    if node_attr[node]=='inv':
        INV(x, node)
    elif node_attr[node]=='buf':
        BUF(x, node)
    elif node_attr[node]=='or2':
        OR(x, node)
    elif node_attr[node]=='and2':
        AND(x, node)
    elif node_attr[node]=='xor2':
        XOR(x, node)
    elif node_attr[node]=='nand2':
        NAND(x, node)
    elif node_attr[node]=='nor2':
        NOR(x, node)
```

[14]:
```
x=1
while queue:
    node_t1 = queue.pop(0)
    if node_val[x][node_t1]==1 or node_val[x][node_t1]==0:
        continue
    y=list(g.predecessors(node_t1))
    if node_val[x][y[0]]==None or node_val[x][y[1]]==None:
        queue.append(node_t1)
    else:
        solve_gate(x, node_t1)

for x in range(2,niv+1):
    queue=[]
    for i in nets:
        if (node_attr[i]=='PI') and (node_val[x][i]!=node_val[x-1][i]):
            queue.append(i)
            for n in queue:
                l=list(g.successors(n))
                queue.extend(l)
    a=list(set(queue))
    for i in nets:
        if (i not in a):
            node_val[x][i]=node_val[x-1][i]
        if (i in a) and (node_attr[i]!='PI'):
            node_val[x][i]=None
    while a:
```

```
        node_t2 = a.pop(0)
        if node_val[x][node_t2]==1 or node_val[x][node_t2]==0:
            continue
        z=list(g.predecessors(node_t2))
        if node_val[x][z[0]]==None or node_val[x][z[1]]==None:
            a.append(node_t2)
        else:
            solve_gate(x, node_t2)
```

[15]:
```
nets.sort()
print(nets)
for i in range(1, niv+1):
    net_state=[]
    for j in nets:
        net_state.append(node_val[i][j])
    print(net_state)
```

```
['N1', 'N2', 'N22', 'N23', 'N3', 'N6', 'N7', 'n_0', 'n_1', 'n_2', 'n_3']
[0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0]
[0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1]
[1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1]
[0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1]
[1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1]
[1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0]
[1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1]
[1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0]
[0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0]
[0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1]
```

- In the topological order method it does multiple rounds of circuit evaluation where all the nets are evaluated again but in the event driven evaluation only the nets which change when some of the primary inputs are changed are evaluated. So it appears topological method takes more time but if the number of primary inputs and their successors that change are high poping and appending again and again in event driven method takes more time. but if very few of the inputs and successors changes then topological may take more time