Week 4 final

Shah Jainam EE21B122

March 1, 2023

[54]:
```
# Import neccesray libraries

import networkx as nx

#define some important functions
''''''''''''''''''''''''''''''''''''''''''''''''''''''''
# Make a function to alphabetically sort the dictionary
def sort_d(dictionary):
    sorted_keys = sorted(dictionary.keys())
    sorted_dict = {key: dictionary[key] for key in sorted_keys}
    return sorted_dict
''''''''''''''''''''''''''''''''''''''''''''''''''''''''
# Make a loading function
def loader(raw,i):
    value = {}
    for node in raw[0].split():
        '''
        set the key of dic as node name and get node name from the first␣
 ↪element of the
        set the input 1,0 as values, first split the other element the get the␣
 ↪coresponding
        by getting the index of the node name
        '''
        value.update({node:raw[i].split()[raw[0].split().index(node)]})
    return value
''''''''''''''''''''''''''''''''''''''''''''''''''''''''
# Find input and output pins or nodes from the net list itself

# A function that returns intersection of two sets
def node_finder(a,b):
    aset = set(a)
    bset = set(b)

    return aset.intersection(bset)
'''
 This function returns two value first the input pins or nodes and second the␣
 ↪output pins
 mixed with some in between nodes.
'''

''''''''''''''''''''''''''''''''''''''''''''''''''''''
def pin_namer(data):
    i_pins_f = []
    o_pins_f = []
    for i in range(len(data)):
        line = data[i].split()
        i_pins_f.append(line[2])
        i_pins_f.append(line[3])
        o_pins_f.append(line[4])

    i_pins_f = set(i_pins_f)

    nodes = node_finder(i_pins_f,o_pins_f)
    #o_pins = set(o_pins_f).symmetric_difference(list(nodes))
    '''
    uncomment the above line to only get output pins without middle nodes
    Make sure to use o_pins insted o_pins_f in return value
    '''
    i_pins = i_pins_f.symmetric_difference(list(nodes))
    return list(o_pins_f),list(i_pins)
pin_namer(raw_circuit)[1]
''''''''''''''''''''''''''''''''''''''''''''''''''''''

# This function returns a dictionary that contains the keys and values that␣
 ↪have been changed
def changer(dict1, dict2):
    result_dict = {}
    for key in dict1:
        if key in dict2 and dict1[key] != dict2[key]:
            result_dict[key] = dict2[key]
    return result_dict
```

[55]:
```
# Take nodes from .net file
filename = 'c17.net'
with open(filename,'r') as f:
    rawcircuit = f.read().splitlines()

# Take inputs from .input file
file = 'c17.inputs'
with open(file,'r') as f:
    raw_inputs = f.read().splitlines()
####################################################
#for not gate converting, not to nands
raw_circuit = []
```

```
for line in rawcircuit:
    line = line.split()
#     print(line[1])
    if line[1] == 'inv':
#         line[1] = 'nand2'
        line.insert(3,line[2])
    if line[1] == 'buf':
        line.insert(3,line[2])
    line = ' '.join(line)
    raw_circuit.append(line)

# Handling a corrupted netlist that contains missing or excess input nodes
if sorted(raw_inputs[0].split()) != sorted(pin_namer(raw_circuit)[1]):
       print('Wrong Inputs')
```

[56]:
```
# get edges in a list name edge
# edge means inputs and outputs
n = len(raw_circuit)
edge = []
for i in range(n):
    buf = []
    line = raw_circuit[i].split()
    buf.append((line[2],line[4]))
    buf.append((line[3],line[4]))
    edge.extend(buf)
```

[57]:
```
# Make a dictionary to whose key the node name and value is the gate type
namegate = {}

'''
Get inputs as the are names 'pi' not any particular gate type

Output nodes are fine as they can be valued as their gates and hence
they are similar to other nodes
'''
input_pins = pin_namer(raw_circuit)[1]
output_pins = pin_namer(raw_circuit)[0]

# line[1] has gate type in netlist
for i in range(n):
    line = raw_circuit[i].split()
    namegate.update({output_pins[i]:line[1]})

# Give input node as 'pi' as values
for pin in input_pins:
    namegate.update({pin:'pi'})
```

```
[58]: # Define the gate functions
'''
I have not made NOT and BUF gates as they can be said to be NAND and AND with␣
 ↪same inputs

'''
def AND (n1, n2):
    if int(n1) == 1 and int(n2) == 1:
        return '1'
    else:
        return '0'

def NAND (n1, n2):

    if int(n1) == 1 and int(n2) == 1:
        return '0'
    else:
        return '1'

def OR(n1, n2):
    if int(n1) == 1 or int(n2) == 1:
        return '1'
    else:
        return '0'

def NOR(n1, n2):
    if int(n1) == 1 or int(n2) == 1:
        return '0'
    else:
        return '1'

def XOR (n1, n2):
    if int(n1) != int(n2):
        return '1'
    else:
        return '0'

def XNOR (n1, n2):
    if int(n1) != int(n2):
        return '0'
    else:
        return '1'
```

1 Topological Order

[59]:
```
# Object generator
g = nx.DiGraph()

# Add edges using list edge made earlier
g.add_edges_from(edge)
# Set nodes using dictionary namegate made earlier
nx.set_node_attributes(g,namegate,name="gateType")

# print(g.nodes(data=True))
# print()
try:
    nl = list(nx.topological_sort(g))
    # Print in topological order
    print('Nodes in topological order',nl)
except nx.NetworkXUnfeasible:
    print('Cannot Evaluate a sequential cicuit')
```

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

2 Topological Evaluation

2.1 This evaluated using Method-1

- Method-1 was to repetadly loading the inputs and get the coressponding outputs

- g.nodes[node] gives me a dictionary that has key gateType whose value is gate type
- The inputs to the gate can be found using g.predeccesors and we can be safe in assuming that its list length will be 2 except in BUF and NOT gates with length being 1
- Thow the predecessor nodes in a dictionary as keys, the value of the keys is the state of the node
- Update the value dict by adding the state of nodes step by step in topological order
- We can be sure that there is no KeyError because we are iterating in order of topological sort so unknow states are calculated from its predeccesor nodes

[60]:
```
# Solve the combinational circuit now
# Total number of inputs in the file
n = len(raw_inputs)

value = {}
outputs = []


for i in range(1,n):
    value.update(loader(raw_inputs,i))
    for node in nl:
        if g.nodes[node]['gateType'] == 'pi':
            pass
    #         value_t[node] = value[node]
        elif g.nodes[node]['gateType'] == 'nand2':
            nands = NAND(value[list(g.predecessors(node))[0]],value[list(g.
 ↪predecessors(node))[1]])
            value.update({node:nands})
        elif g.nodes[node]['gateType'] == 'and2':
            ands = AND(value[list(g.predecessors(node))[0]],value[list(g.
 ↪predecessors(node))[1]])
            value.update({node:ands})
        elif g.nodes[node]['gateType'] == 'or2':
            ors = OR(value[list(g.predecessors(node))[0]],value[list(g.
 ↪predecessors(node))[1]])
            value.update({node:ors})
        elif g.nodes[node]['gateType'] == 'nor2':
            nors = NOR(value[list(g.predecessors(node))[0]],value[list(g.
 ↪predecessors(node))[1]])
            value.update({node:nors})
        elif g.nodes[node]['gateType'] == 'xor2':
            xors = XOR(value[list(g.predecessors(node))[0]],value[list(g.
 ↪predecessors(node))[1]])
            value.update({node:xors})
        elif g.nodes[node]['gateType'] == 'xnor2':
            xnors = XNOR(value[list(g.predecessors(node))[0]],value[list(g.
 ↪predecessors(node))[1]])
            value.update({node:xnors})
        elif g.nodes[node]['gateType'] == 'inv':
            nots = NAND(value[list(g.predecessors(node))[0]],value[list(g.
 ↪predecessors(node))[0]])
            value.update({node:nots})
        elif g.nodes[node]['gateType'] == 'buf':
            buffs = AND(value[list(g.predecessors(node))[0]],value[list(g.
 ↪predecessors(node))[0]])
            value.update({node:buffs})
#   sort the dictionary alphabetically
    value = sort_d(value)
    outputs.append(list(value.values()))
```

[61]:
```
print(list(value.keys()))
outputs
```

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

[61]:
```
[['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']]
```

[62]:
```python
# Write the solution in a text file
file_answers = 'topo_eval_ee21b122_17.txt'
with open(file_answers,'w') as f:
    f.write(' '.join(list(value.keys()))+'\n')
    for out in outputs:
        f.write(' '.join(out) +'\n')
```

3 Event-driven simulation

- This computed using event driven evaluation
- In event driven evalution I pop a node from top and push its sucsessor from the bottom
- For simulation I push the nodes whose inputs are changed and run the same algorithm

[63]:
```python
# Intializing the que
input_pins = raw_inputs[0].split()

# Intializing variables
value = {}
prev={}
new = {}
outpus = []

# Simulation
for i in range(1,n):
#   load the i'th input
    value.update(loader(raw_inputs,i))

#   Is the circuit running for first time? Yes the proceed or go to if block
    if i!=1:
        '''
        Load the previous input and use the changer() that returns a
        dictionary that keys and values which are changed
        now add this keys to the que and clear the dictionary
        '''
        prev.update(loader(raw_inputs,i-1))
        new.update(changer(prev,value))
        input_pins.extend(list(new.keys()))
        new = {}

#   Run a infinte loop till the queue is empty
    while True:
        '''
        Using try-except block because when the circuit runs for the first time␣
 ↪the
        dictionary values only contains the inputs and no other nodes,and thus␣
 ↪this
        gives a key error in below code
        '''
        try:
            '''Pop the first node and push the succsesor'''
            node = input_pins.pop(0)
            input_pins.extend(list(g.successors(node)))

            if len(input_pins)!=0:
                if g.nodes[node]['gateType'] == 'pi':
                    pass
                elif g.nodes[node]['gateType'] == 'nand2':
                    nands = NAND(value[list(g.
 ↪predecessors(node))[0]],value[list(g.predecessors(node))[1]])
                    value.update({node:nands})
                elif g.nodes[node]['gateType'] == 'and2':
                    ands = AND(value[list(g.
 ↪predecessors(node))[0]],value[list(g.predecessors(node))[1]])
                    value.update({node:ands})
                elif g.nodes[node]['gateType'] == 'or2':
                    ors = OR(value[list(g.predecessors(node))[0]],value[list(g.
 ↪predecessors(node))[1]])
                    value.update({node:ors})
                elif g.nodes[node]['gateType'] == 'nor2':
                    nors = NOR(value[list(g.
 ↪predecessors(node))[0]],value[list(g.predecessors(node))[1]])
                    value.update({node:nors})
                elif g.nodes[node]['gateType'] == 'xor2':
                    xors = XOR(value[list(g.
 ↪predecessors(node))[0]],value[list(g.predecessors(node))[1]])
                    value.update({node:xors})
                elif g.nodes[node]['gateType'] == 'xnor2':
                    xnors = XNOR(value[list(g.
 ↪predecessors(node))[0]],value[list(g.predecessors(node))[1]])
                    value.update({node:xnors})
                elif g.nodes[node]['gateType'] == 'inv':
                    nots = NAND(value[list(g.
↪predecessors(node))[0]],value[list(g.predecessors(node))[0]])
                    value.update({node:nots})
                elif g.nodes[node]['gateType'] == 'buff':
                    buffs = AND(value[list(g.
↪predecessors(node))[0]],value[list(g.predecessors(node))[0]])
                    value.update({node:buffs})

            else:
                break
        except KeyError:
            pass
#   Sort the dictionary alphabetically
    value = sort_d(value)
    outpus.append(list(value.values()))
```

[64]:
```
print(list(value.keys()))
outpus
```

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

[64]:
```
[['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']]
```

[65]:
```
# Write the solution in a text file
file_answers = 'event_simula_ee21b122_17.txt'
with open(file_answers,'w') as f:
    f.write(' '.join(list(value.keys()))+'\n')
    for out in outputs:
        f.write(' '.join(out) +'\n')
```

4 Conclusion

- Topological Evaluation can be used when there are a lot nodes provided but number of input states provided are less.
- Event-driven evaluation can be used when the number of nodes are less but number of input

states provided are a lot

- This is because topological evaluation works on toplogical sort whose lenght is finite i.e number of nodes, but event driven evaluation works on principle of queue and queue lenght depends on number of nodes, hence if queue is quite large evaluation can take up more time than topological evaluation
- Hence,
  - If number of nodes are **lot** and number of input states provided are **less**: Topological Evaluation
    - As a lot of nodes means a lot of time in event driven simulation
  - If number of nodes are **less** and number of input states provided are **lot**: Event Driven Simulation
    - less states means event driven simulation is quiete fast and as input states provided are quiet large topological evaluation will be slow as compared
  - If number of nodes are **lot** and number of input states provided are **lot**: Topological Evaluation
    - lots of nodes so toplogical evaluation is faster
  - If number of nodes are **less** and number of input states provided are **less**: Event Driven Simulation
    - As less nodes so event driven simulation is faster

tl;dr Topological sort speed depends on number of times input node states are changed and Event driven simulation speed depends on number of nodes in the circuit

input states means how many times the input nodes state changes