week4_solution

Aman Kumar EE21B013 <ee21b013@smail.iitm.ac.in>

March 1, 2023

1 About Code and How to run:

- This is a simple **.ipynb** file just run it in **jupyter notebook** -I have divided the program in different function so the previous block of code is might be required for getting the correct output form the current block.
- So to avoid some unusual output run the code serially.
- The function are well defind and **comments** have been given where ever required.
- As the code is itself too long and i think it is self explanatory because i tried to use as much similar name of variable to what they are getting used for inside the code ,so why i have not given very large explanation on working of function
- One can easily understand the codes if he/she goes **serially and manually**.
- I have stored output of Both the **topological_sort and EVENT_DRIVEN** seperately in two output files named :
- for **EVENT_DRIVEN**=> file name is **-EVENT_DRIVEN_OUTPUT.txt**
- for **TOPOLOGICAL_SORT**=> file name is - **topological_Sort.txt**
- If you have any difficulty in reading or understanding function anywhere just ask me to explain .

[329]:
```
def pri(A):
    for i in A:
        print(i)
```

1.1 Below Functions are defined for performing logical operation to get output

[330]:
```
#Functions defined for performing logical operation
def andFunc(a,b):
    if(a==0 or b==0):
        return 0
    else:
        return 1

def orFunc(a,b):
    if(a==1 or b==1):
        return 1
    else:
        return 0

def invFunc(a):
```

```python
    if(a==1):
        return 0
    else:
        return 1

def norFunc(a,b):
    if(a==1 or b==1):
        return 0
    else:
        return 1

def nandFunc(a,b):
    if(a==0 or b==0):
        return 1
    else:
        return 0

def xorFunc(a,b):
    if ((a==1 and b==0) or (a==0 and b==1)):
        return 1
    else:
        return 0

def xnorFunc(a,b):
    if ((a==1 and b==0) or (a==0 and b==1)):
        return 0
    else:
        return 1
```

1.2 The below code is for Reading the input file

[331]:
```python
# Function for Reading of gate and circuit information
def read_netlist(filename):
    with open(filename,'r') as mainfile:
        file_storage=mainfile.readlines()
        return file_storage
```

[332]:
```python
# Fucntion for reading of inputs
def read_input(filename):
    with open(filename,'r') as mainfile:
        file_storage=mainfile.readlines()
        return file_storage
```

[333]:
```python
ckt_netlist=read_netlist('/Users/amankumar/Documents/semester4/Applied␣
 ↪Programming Lab(APL)/Assignment4/benchmarks/c17.net')
ckt_detail=[]
for line in ckt_netlist:
```

```
    ckt_detail.append(line)

ckt_input=read_input('/Users/amankumar/Documents/semester4/Applied Programming␣
↪Lab(APL)/Assignment4/benchmarks/c17.inputs')
inputckt_detail=[]
for line in ckt_input:
    inputckt_detail.append(line.rstrip('\n').split())
print('primary_input at different time instant : ')
pri(inputckt_detail)
```

```
primary_input at different time instant :
['N1', 'N2', 'N3', 'N6', 'N7']
['0', '1', '0', '0', '0']
['0', '0', '1', '0', '0']
['1', '0', '0', '0', '0']
['0', '0', '1', '1', '1']
['1', '1', '1', '1', '1']
['1', '1', '1', '0', '0']
['1', '1', '1', '1', '0']
['1', '1', '0', '0', '0']
['0', '1', '1', '0', '1']
['0', '0', '1', '1', '0']
```

1.3 SECTION : Making of graph using inbuilt Function by adding edges

[334]:
```
import networkx as nx
g=nx.DiGraph()
connection={}  # For storing connection like storing of connections based on␣
↪output (i.e. key is output and the value is connected elements)
for i in range(len(ckt_detail)):
    if(ckt_detail[i].split()[1]=='inv' or ckt_detail[i].split()[1]=='buf'):
        connection[ckt_detail[i].split()[3]]=ckt_detail[i].split()[1:
↪len(ckt_detail[i])-2]
    else:
        connection[ckt_detail[i].split()[4]]=ckt_detail[i].split()[1:
↪len(ckt_detail[i])-2]
print(connection)
for ele in connection:
    if(connection[ele][0]=='inv' or connection[ele][0]=='buf'):
        g.add_edges_from([(connection[ele][1],connection[ele][2])])
        # nx.set_node_attributes(g,{connection[ele][2]:ele})
    else:
        g.
↪add_edges_from([(connection[ele][1],connection[ele][3]),(connection[ele][2],connection[ele]
        # nx.set_node_attributes(g,{connection[ele][3]:ele})
```

```
print(g)
```

```
{'N22': ['nand2', 'n_3', 'n_0', 'N22'], 'N23': ['nand2', 'n_3', 'n_2', 'N23'],
'n_3': ['nand2', 'n_1', 'N2', 'n_3'], 'n_2': ['nand2', 'n_1', 'N7', 'n_2'],
'n_0': ['nand2', 'N1', 'N3', 'n_0'], 'n_1': ['nand2', 'N3', 'N6', 'n_1']}
DiGraph with 11 nodes and 12 edges
```

2 Topological sort for ordering of output and input of gate

2.0.1 Error handling

[335]:
```
error=False
try:
    nl =  list(nx.topological_sort(g))
    print('Nodes in topological order', nl)
except:
    error=True
    print("!!!  ERRROR :   CYCLIC GRAPH")
```

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

[358]:
```
primary_input={}
print(inputckt_detail[0])
for i in range(len(inputckt_detail[0])):
    primary_input.update({inputckt_detail[0][i]:[]})
for i in range(1,len(inputckt_detail)):
    for j in range(len(inputckt_detail[0])):
        primary_input[inputckt_detail[0][j]].append(int(inputckt_detail[i][j]))
try:
    for ele in primary_input:
        for value in primary_input[ele] :
            if value==0 or value ==1:
                continue
            else:
                raise ValueError
except ValueError:
    print('!!! ERROR WRONG INPUT FORMAT')
print('Primary Inputs are : ')
print(primary_input)

print()
print('connetion are means in dictionary connection output is key and other␣
 ↪things related to output are its value : ')
print(connection)
```

```
['N1', 'N2', 'N3', 'N6', 'N7']
Primary Inputs are :
```

```
{'N1': [0, 0, 1, 0, 1, 1, 1, 1, 0, 0], 'N2': [1, 0, 0, 0, 1, 1, 1, 1, 1, 0],
'N3': [0, 1, 0, 1, 1, 1, 1, 0, 1, 1], 'N6': [0, 0, 0, 1, 1, 0, 1, 0, 0, 1],
'N7': [0, 0, 0, 1, 1, 0, 0, 0, 1, 0]}

connetion are means in dictionary connection output is key and other things
related to output are its value :
{'N22': ['nand2', 'n_3', 'n_0', 'N22'], 'N23': ['nand2', 'n_3', 'n_2', 'N23'],
'n_3': ['nand2', 'n_1', 'N2', 'n_3'], 'n_2': ['nand2', 'n_1', 'N7', 'n_2'],
'n_0': ['nand2', 'N1', 'N3', 'n_0'], 'n_1': ['nand2', 'N3', 'N6', 'n_1']}
```

3 EXPLANATION OF BELOW CODE

- FUNCTION topo_sort_fun : takes 3 input (connection ,input_detail,primary_input)
- Based on above topological sorting I am just traversing the topological order and calculaing the value of output accordingly.
- after calculating all the output of all the nodes i am storing the output in final output
- after that there is a file writing process that will write the output correspondingly in a file named topological_Sort.txt.

[347]:
```
import copy
# Function topo sort find the solution based on topological sorting ordering
def topo_sort_func(connection,inputc8_detail,primary_input):
    if(error):
        print('!!! ERRROR CYCLIC GRAPH ')
        return
    final_output=[]
    for i in range(1,len(inputc8_detail)):
        output_detail={}  #storing dictionary for output
        for ele in nl:
            if(ele in primary_input.keys()):
                output_detail.update({ele:primary_input[ele][i-1]})
            else:
                if(connection[ele][0]=='and2'):
                    output_detail.update({ele:
↪andFunc(output_detail[connection[ele][1]],output_detail[connection[ele][2]])})
                if(connection[ele][0]=='or2'):
                    output_detail.update({ele:
↪orFunc(output_detail[connection[ele][1]],output_detail[connection[ele][2]])})
                if(connection[ele][0]=='inv'):
                    output_detail.update({ele:
↪invFunc(output_detail[connection[ele][1]])})
                if(connection[ele][0]=='nand2'):
                    output_detail.update({ele:
↪nandFunc(output_detail[connection[ele][1]],output_detail[connection[ele][2]])})
                if(connection[ele][0]=='nor2'):
                    output_detail.update({ele:
↪norFunc(output_detail[connection[ele][1]],output_detail[connection[ele][2]])})
```

```
                if(connection[ele][0]=='xor2'):
                    output_detail.update({ele:
↪xorFunc(output_detail[connection[ele][1]],output_detail[connection[ele][2]])})
                if(connection[ele][0]=='xnor2'):
                    output_detail.update({ele:
↪xnorFunc(output_detail[connection[ele][1]],output_detail[connection[ele][2]])})
                if(connection[ele][0]=='buf'):
                    output_detail.update({ele:
↪output_detail[connection[ele][1]]})
        # print(f"for input {i} values of all noes are :") # Uncomment this and␣
↪the below line to view the output at differnt input in file
        # print(output_detail)
        final_output.append(copy.deepcopy(output_detail))
    return final_output

topo_sort_Output=topo_sort_func(connection,inputckt_detail,primary_input)

# print('topo_sort_Output : ')
# print(topo_sort_Output)

# File writing for the solution
with open('topological_Sort.txt','w') as f:
    for nodes in sorted(nl):
        f.write(f'{nodes:<10}')
    f.write("\n")
    for element in topo_sort_Output:
        for ele in sorted(element.keys()):
            f.write(f'{element[ele]:<10}')
        f.write('\n')
```

4 EVENT DRIVEN SOLUTION

[342]:
```
# connections
# print(connection) #storage dictionary according to output key is output and␣
↪other things are stored in lsit
# print(inputc8_detail) # storage after reading of input file
# print(primary_input) # store primary input values
```

5 Explantaiton of below function

- Function is taking three argument-> connection (stores information about circuit), input_detail(details of input),primaryinput(stores primary input and its values at different time instant) all these three are defined above and it is passed in the parameter just after the function end when calling the function.
- Inside the function I am using the queue to sotre and do the operation needed for evnet driven

approach.

- Inside the function iterating over the different instant of input and checking the input when the input changes i am just appending that element in the queue and then iterating over the queue till the queue is not empty and assigning the updated output of different nodes.
- I have used input_connection to store the output that will be affected by changing the that particular input.
- final_output is storing the final output .
-

[343]:
```python
import copy
# print(primary_input)
def Event_Driven_solution(connection,input_detail,primary_input):

    if(error):
        print('!!! ERRROR CYCLIC GRAPH ')
        return

    final_output=[]
    final_output.clear()
    updated_output={}  # dictionary for storing the output at different␣
 ↪changing output
    updated_output.clear()
    queue=[]

    # print(connection)
    # storing the connection like whcih are the outut corresponding to a␣
 ↪particular input inside the input_connection
    intput_connnection={}
    for ele in connection:
        if(connection[ele][0]!='inv' and connection[ele][0]!='buf'):
            intput_connnection.update({connection[ele][1]:[]})
            intput_connnection.update({connection[ele][2]:[]})
        else:
            intput_connnection.update({connection[ele][1]:[]})

    # Populating the input_connection dictonary
    for ele in connection:
        # print(ele)
        if(connection[ele][0]!='inv' and connection[ele][0]!='buf'):
            current_ele1=intput_connnection[connection[ele][1]]
            current_ele1=current_ele1+[ele]

            current_ele2=intput_connnection[connection[ele][2]]
            current_ele2=current_ele2+[ele]

            intput_connnection.update({connection[ele][1]:current_ele1})
            intput_connnection.update({connection[ele][2]:current_ele2})
```

```
        else:
            current_ele1=intput_connnection[connection[ele][1]]
            current_ele1=current_ele1+[ele]
            intput_connnection.update({connection[ele][1]:current_ele1})

    # print('input connection : ')   #Uncomment this and below three lines it␣
↪will show the input connection means storing what are
    # print(intput_connnection)   #different output connected to a given input
    # print(primary_input)
    queue.clear()
    # print(len(input_detail))
    for i in range(len(input_detail)-1):
        for ele in primary_input:
            if(i==0):
                queue.append(ele)
                updated_output.update({ele:primary_input[ele][i]})
            else:
                if(primary_input[ele][i]!=primary_input[ele][i-1]):
                    queue.append(ele)
                    updated_output.update({ele:primary_input[ele][i]})
                else:
                    updated_output.update({ele:primary_input[ele][i]})

        # Do the operation while queue is not empty
        while(len(queue)>0):

            curr_vertex=queue.pop(0)
            if curr_vertex in primary_input.keys():
                if curr_vertex in intput_connnection.keys():
                    for all_output in intput_connnection[curr_vertex]:
                        queue.append(all_output)
            else:
                if connection[curr_vertex][0]!='inv' and␣
↪connection[curr_vertex][0]!='buf':
                    inp1=connection[curr_vertex][1]
                    inp2=connection[curr_vertex][2]
                    if inp1 in updated_output.keys() and inp2 in updated_output.
↪keys():
                        if connection[curr_vertex][0]=='and2':
                            updated_output.update({curr_vertex:
↪andFunc(updated_output[inp1],updated_output[inp2])})
                        elif connection[curr_vertex][0]=='or2':
                            updated_output.update({curr_vertex:
↪orFunc(updated_output[inp1],updated_output[inp2])})
                        elif connection[curr_vertex][0]=='nand2':
                            updated_output.update({curr_vertex:
↪nandFunc(updated_output[inp1],updated_output[inp2])})
```

```
                        elif connection[curr_vertex][0]=='nor2':
                            updated_output.update({curr_vertex:
 ↪norFunc(updated_output[inp1],updated_output[inp2])})
                        elif connection[curr_vertex][0]=='xor2':
                            updated_output.update({curr_vertex:
 ↪xorFunc(updated_output[inp1],updated_output[inp2])})
                        elif connection[curr_vertex][0]=='xnor2':
                            updated_output.update({curr_vertex:
 ↪xnorFunc(updated_output[inp1],updated_output[inp2])})
                    else:
                        queue.append(curr_vertex)
                        continue
                    if curr_vertex in intput_connnection.keys():
                        for all_output in intput_connnection[curr_vertex]:
                            queue.append(all_output)

                else:
                    inp1=connection[curr_vertex][1]
                    if(inp1 in updated_output.keys()):
                        if connection[curr_vertex][0]=='inv':
                            updated_output.update({curr_vertex:
 ↪invFunc(updated_output[inp1])})
                        elif connection[curr_vertex][0]=='buf':
                            updated_output.update({curr_vertex:
 ↪updated_output[inp1]})
                    else:
                        queue.append(curr_vertex)
                        continue
                    if curr_vertex in intput_connnection.keys():
                        for all_output in intput_connnection[curr_vertex]:
                            queue.append(all_output)
        # print(f'Different Nodes value for input at {i} are : ')# Uncomment␣
 ↪these two lines for showing output at different instance of time
        # print(updated_output)
        final_output.append(copy.deepcopy(updated_output))
    return final_output
event_driven_Output=Event_Driven_solution(connection,inputckt_detail,primary_input)␣
 ↪ #Calling of Event_Driven function for getting output

# Writing the Output in file EVENT_DRIVEN_OUTPUT.txt
with open('EVENT_DRIVEN_OUTPUT.txt','w') as f:
    for nodes in sorted(nl):
        f.write(f'{nodes:<10}')
    f.write("\n")
    for element in event_driven_Output:
        for ele in sorted(element.keys()):
```

```python
            f.write(f'{element[ele]:<10}')
        f.write('\n')
```

[345]:

```python
print('The time taken by Event_driven_solution : ')
%timeit Event_Driven_solution(connection,inputckt_detail,primary_input)
print('the time taken by topo_sort_function : ')
%timeit topo_sort_func(connection,inputc8_detail,primary_input)
```

```
The time taken by Event_driven_solution :
146 µs ± 300 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
the time taken by topo_sort_function :
94.2 µs ± 618 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

In some cases the time taken by topological sort may be less than the time taken by Event_driven apporach But in large no of cases we will observe that the time taken by event driven approach is less because if the input changes in small no than the event driven approach is a way bettr than the topological sort because we need to update only the corresponding output but in topological sort we need to iterate the whole vertex each time to update the output at each terminal respectively.