EE2703 - Week 4

Hardik Gagrani, EE21B047

March 1, 2023

[1]:
```
def read_netlist(filename):

    with open(filename, 'r') as f:
        lines = f.readlines()

    components = []

    for line in lines:
        if line.strip() != "":
            line = line.strip().split()

            if str(line[0][0]) == 'g' and line[1] != 'inv' and line[1] != 'buf':
                components.append(('g', (line[1]), (line[2]), (line[3]),␣
 ↪(line[4])))
            else:
                components.append(('g', (line[1]), (line[2]), (line[3])))

    return components
```

This function `read_netlist` takes in the net file input and makes the data given in it user friendly so that it would be easy to use them while coding later on. In this I am appending the `gate_ID`, `parent_gates` and `output_gate` in a list of tuples which will make it easy to use later on.

[2]:
```
import networkx as nx

def topological_order(components):
    g = nx.DiGraph()
    Edges = []

    for component in components:
        if component[0] == 'g' and component[1] != 'inv' and component[1] !=␣
 ↪'buf':
            Edges.append(((component[2]), (component[4])))
            Edges.append(((component[3]), (component[4])))
        else:
            Edges.append(((component[2]), (component[3])))

    g.add_edges_from(Edges)
```

```
    if(nx.is_directed_acyclic_graph(g)==True):
        pass
    else:
        print("Error!")
        print("The given gate connections form a cycle.")
        print("Terminating.")
        return

    nl = list(nx.topological_sort(g))

    return nl
```

This function creates a `topological_order` of all the nodes which are present in the logic gate circuit. It will place them in the order of them occuring; Starting with primary input and ending with the final output states who don't have any successors (who don't give any further output).

[3]:
```
def successors(components, A):
    g = nx.DiGraph()
    Edges = []

    for component in components:
        if component[0] == 'g' and component[1] != 'inv' and component[1] !=␣
↪'buf':
            Edges.append(((component[2]), (component[4])))
            Edges.append(((component[3]), (component[4])))
        else:
            Edges.append(((component[2]), (component[3])))

    g.add_edges_from(Edges)
    suc = list(g.successors(A))

    return suc
```

This block of code will find the `successors` of a given node in logic gate circuit and return them in the form of list.

[4]:
```
def nandgate(a, b):
    return int(not(a and b))

def orgate(a, b):
    return int(a or b)

def andgate(a, b):
    return int(a and b)

def norgate(a, b):
    return int(not(a or b))
```

```
def xorgate(a, b):
    return int(a ^ b)

def xnorgate(a, b):
    return int(not(a ^ b))

def invgate(a):
    return int(not a)
```

I have defined simple logic gate functions here in this block of code which are used to calculate output later on when needed

1 TOPOLOGICAL METHOD:

[5]:
```
def topological_solution(filename_inputs, filename_net):
    cp = read_netlist(filename_net)
    xp = topological_order(cp)

    x_dict = {xp[i]: i for i in range(len(xp))}

    with open(filename_inputs, 'r') as f:
        lines = f.readlines()

    allinputs= []

    for line in lines:
        input_layer = []
        dict_ = {}

        for line_ in lines:
            line_ = line_.strip().split()
            for j in range(len(line_)):
                dict_.update({line_[j]: line_[j]})
                input_layer.append(line_[j])
            break

        if line[0] != "N" and line[0] != "a":
            line = line.strip().split()
            for j in range(len(line)):
                dict_[input_layer[j]] = line[j]

            allinputs.append(dict_)

    for i in allinputs:
        while(len(i) != len(x_dict)):
            for comp in cp:
```

```
                if comp[2] in list(i):
                    if comp[1] == 'inv':
                        e = i[comp[2]]
                        i.update({comp[3]: invgate(int(e))})
                    elif comp[1] == 'buf':
                        m = i[comp[2]]
                        i.update({comp[3]: int(m)})
                    elif comp[3] in list(i.keys()):
                        p = i[comp[2]]
                        q = i[comp[3]]
                        if comp[1] == 'nand2':
                            i.update({comp[4]: nandgate(int(p), int(q))})
                        elif comp[1] == 'and2':
                            i.update({comp[4]: andgate(int(p), int(q))})
                        elif comp[1] == 'or2':
                            i.update({comp[4]: orgate(int(p), int(q))})
                        elif comp[1] == 'nor2':
                            i.update({comp[4]: norgate(int(p), int(q))})
                        elif comp[1] == 'xor2':
                            i.update({comp[4]: xorgate(int(p), int(q))})
                        elif comp[1] == 'xnor2':
                            i.update({comp[4]: xnorgate(int(p), int(q))})

    return allinputs
```

This function takes two file as inputs: one for input values and other is the netlist. It uses the `read_netlist` function to read netlist and then uses `topological_order` function to determine the order in which the gates should be evaluated.

Later it reads input value file and create dictionaries for each line whose keys are the respective primary input gate and key is also the same. I have done like this now because afterwards I am appending them with the respective input given because there is not a single input that's why it was good to generalise them first. These all dictionaries after appending inputs are then stored them in a list called `allinputs`.

Afterwards each dictionary I iterate each dictionary through a loop of all components present in netlist wherein the dictionary would be updated untill and unless all states are present in the dictionary. This loop evaluates the gates in the netlist by checking whether the inputs for each gate are in the dictionary. If they are, the function uses the input values to perform the appropriate gate operation and updates the dictionary with the output value.

The loop condition that is `len(i) != len(x_dict)` will make the loop run exactly the `number of times = number of layer` present in the topological sequence. This loop is repeated until all gates for the current input values have been evaluated. After processing all input values, the function returns a list of dictionaries, where each dictionary represents the output values for a set of input values.

2 EVENT-DRIVEN APPROACH:

[6]:
```
def inputs(filename_inputs, filename_net):
    cp = read_netlist(filename_net)
    xp = topological_order(cp)

    x_dict = {xp[i]: i for i in range(len(xp))}

    with open(filename_inputs, 'r') as f:
        lines = f.readlines()

    allinputs= []

    for line in lines:
        input_layer = []
        dict_ = {}

        for line_ in lines:
            line_ = line_.strip().split()
            for j in range(len(line_)):
                dict_.update({line_[j]: line_[j]})
                input_layer.append(line_[j])
            break

        if line[0] != "N" and line[0] != "a":
            line = line.strip().split()
            for j in range(len(line)):
                dict_[input_layer[j]] = line[j]

            allinputs.append(dict_)
    return allinputs
```

Similarly as did in `topological_order_function` this function also creates a list of dictionaries of all the inputs which needs to be evaluated.

[7]:
```
def output_demo(filename_inputs, filename_net):
    cp = read_netlist(filename_net)
    xp = topological_order(cp)

    allinputs = inputs(filename_inputs, filename_net)
    alloutputs = []
    for inputs_ in allinputs:
        outputdict = inputs_
        for i in xp:
            if i not in outputdict:
                outputdict.update({i:'x'})

        alloutputs.append(outputdict)
```

```
    return alloutputs
```

This function called `output_demo` in same way as we created list of dictionaries for inputs it will create list of dictionaries for outputs where in the values of primary inputs are specified as given in the output list but for other states simply `x` is given as it's value as we are using queue method here, it will be updated later on in next block of code

[8]:
```
import queue

def queuing(filename_inputs, filename_net):
    comp = read_netlist(filename_net)
    topo = topological_order(comp)
    alloutputs = output_demo(filename_inputs, filename_net)

    for output in alloutputs:
        compare_dict = alloutputs[0]
        q = queue.Queue()
        for i in output:
            q.put(i)

        while not q.empty():
            item = q.get()
            for c in comp:
                if c[1] == 'inv':
                    if c[3] == item:
                        initial = compare_dict[item]
                        a = output[c[2]]
                        try:
                            a == int(a)
                            final = final = invgate(int(a))
                            if initial != final:
                                output[item] = final
                                suc = successors(comp, item)
                                for i in suc:
                                    q.put(i)
                            else:
                                output[item] = initial
                        except:
                            pass

                elif c[1] == 'buf':
                    if c[3] == item:
                        initial = compare_dict[item]
                        a = output[c[2]]
                        try:
                            a == int(a)
```

```
                        final = final = int(a)
                        if initial != final:
                            output[item] = final
                            suc = successors(comp, item)
                            for i in suc:
                                q.put(i)
                        else:
                            output[item] = initial
                    except:
                        pass

            elif c[4] == item:
                initial = compare_dict[item]
                a = output[c[2]]
                b = output[c[3]]
                try:
                    a == int(a) and b == int(b)
                    if c[1] == 'nand2':
                        final = (nandgate(int(a), int(b)))
                    elif c[1] == 'and2':
                        final = (andgate(int(a), int(b)))
                    elif c[1] == 'xor2':
                        final = (xorgate(int(a), int(b)))
                    elif c[1] == 'xnor2':
                        final = (xnorgate(int(a), int(b)))
                    elif c[1] == 'xnor2':
                        final = (xnorgate(int(a), int(b)))
                    elif c[1] == 'or2':
                        final = (orgate(int(a), int(b)))
                    elif c[1] == 'nor2':
                        final = (norgate(int(a), int(b)))

                    if final != initial:
                        output[item] = final
                        suc = successors(comp, item)
                        for i in suc:
                            q.put(i)

                    else:
                        output[item] = initial

                except:
                    pass
    return alloutputs
```

This function will operated the `queuing method` (event_driven approach) to solve for all the states of logic gate circuit. A loop is iterated for each set of output values we get from `output_demo` function. It starts by creating a queue for the first set of output values and uses the topological

order and primary inputs to determine the order of gates to be evaluated.

Inside the loop I have implemented normal queuing algorithm. Fr each of those gates, it checks the gate type to determine how to evaluate the gate. If the gate is an inverter or buffer, it simply applies the appropriate operation to the input value and updates the output value for that gate in the output dictionary. If the gate is a 2-input logic gate, it applies the appropriate logic operation to the input values and updates the output value for that gate in the output dictionary.

The queuing algorithm which I used here is after popping a node from the queue, it will determine it's initial value and the final value depeding on it's predessors and gate type or it can be a primary input and after that if the inital value is equal to final value it directly jumps to another element in the queue.

But if initial value is not equal to final value, it will add the successor's of the input because since the input is not same anymore the gate also won't be same anymore and so there's a need of it to be added to the list again. After this the final value is appended in the output dictionary for the particular input of that particular state.

This all process goes on untill the queue becomes empty that is there are no more successor's pending to be added in that queue. Once all this is dont the output dictionary for the current set of input values is addede to a list called `alloutputs`

Finally, after all sets of input values have been evaluated, the function returns the `alloutputs` list containing a dictionary of output values for each set of input values.

[9]:
```
def filetransfer_topological(filename_inputs, filename_net, filename_create):
    components = read_netlist(filename_net)
    topo_order = topological_order(components)

    topo_order.sort()

    topological = topological_solution(filename_inputs, filename_net)

    with open(filename_create, 'w') as f:
        for i in topo_order:
            f.write(f"{i} ")
        f.write("\n")
        for i in topological:
            list_output = dict(sorted(i.items()))
            list_output = list(list_output.values())
            for i in list_output:
                f.write(f"{i} ")
            f.write("\n")

        f.write("\n")

    return
```

This block of code writes the output generated by `topological_solution` method in the format which is asked. It first sorts the topological order and the output dictionary order in the alphabetical order and depending on the input vector the corresponding output vector is written in the file.

[10]:
```
def filetransfer_queue(filename_inputs, filename_net, filename_create):
    components = read_netlist(filename_net)
    topo_order = topological_order(components)

    topo_order.sort()

    queue = queuing(filename_inputs, filename_net)

    with open(filename_create, 'w') as f:
        for i in topo_order:
            f.write(f"{i} ")
        f.write("\n")
        for i in queue:
            list_output = dict(sorted(i.items()))
            list_output = list(list_output.values())
            for i in list_output:
                f.write(f"{i} ")
            f.write("\n")

    return
```

This is the same code as above just the change present here is the output is taken from `queuing` function.

3 C17_1.net

[11]:
```
comp = read_netlist("c17_1.net")
topo = topological_order(comp)
```

```
Error!
The given gate connections form a cycle.
Terminating.
```

[12]:
```
filetransfer_topological("c17.inputs", "c17.net", "c17_topo.outputs")
%timeit filetransfer_topological("c17.inputs", "c17.net", "c17_topo.outputs")
filetransfer_topological("parity.inputs", "parity.net", "parity_topo.outputs")
%timeit filetransfer_topological("parity.inputs", "parity.net", "parity_topo.
 ↪outputs")
filetransfer_topological("c8.inputs", "c8.net", "c8_topo.outputs")
%timeit filetransfer_topological("c8.inputs", "c8.net", "c8_topo.outputs")
filetransfer_topological("c432.inputs", "c432.net", "c432_topo.outputs")
%timeit filetransfer_topological("c432.inputs", "c432.net", "c432_topo.outputs")
```

```
923 µs ± 65.9 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
1.47 ms ± 64.7 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
16.4 ms ± 592 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
1.74 s ± 18 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[13]:
```
filetransfer_queue("c17.inputs", "c17.net", "c17_queue.outputs")
%timeit filetransfer_queue("c17.inputs", "c17.net", "c17_queue.outputs")
filetransfer_queue("parity.inputs", "parity.net", "parity_queue.outputs")
%timeit filetransfer_queue("parity.inputs", "parity.net", "parity_queue.outputs")
filetransfer_queue("c8.inputs", "c8.net", "c8_queue.outputs")
%timeit filetransfer_queue("c8.inputs", "c8.net", "c8_queue.outputs")
filetransfer_queue("c432.inputs", "c432.net", "c432_queue.outputs")
%timeit filetransfer_queue("c432.inputs", "c432.net", "c432_queue.outputs")
```

```
2.56 ms ± 68.3 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
5.67 ms ± 667 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
124 ms ± 4.96 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
18.8 s ± 240 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

4 CONCLUSION

As seen by the time it operations it can be seen as the topological method is faster than the queuing method in terms of solving and taking down the values for all the states which are present in the logic gate circuit. It can be seen from the inputs that topological sort with multiple rounds of circuit evaluations may be more efficient for small and simple circuits, while event-driven simulation using queues may be more efficient for larger and more complex circuits.

As can be seen, topological sort requires an initial pass to determine the order of the nodes, followed by multiple rounds of circuit evaluations to determine the output values of the gates. The number of rounds required is determined by the depth of circuit that is the number of levels of states which are present in the circuit. Topological sort can be very efficient for small circuits with few gates, but for larger circuits, the number of rounds can become a heck as it will start calculating output for start as each new level comes into play. As just imagine we are calculating output for level 100, it will start from level 1 come to level 99 and then it will evaluate results for level 100 which is a lot of time taking process.

Event-driven simulation with queues, on the other hand, can handle circuits of any size and complexity, but requires a more neat implementation. Each gate is represented as an event that is added to a queue when its inputs change. When an event is processed, the gate's output value is calculated, and events for its successor gates are added to the queue when there is a change in event because the logic is if input gate changes there is a high possibility that the output might chage. Because it only processes events that are affected by input changes rather than computing all gates in each round, this approach can be very efficient for large circuits with many gates and high nodes.

In summary, both approaches have advantages and disadvantages, and the choice depends on the specific problem and the characteristics of the input data.