week4_EE21B043

D.Somesh Reddy <ee21b043@smail.iitm.ac.in>

March 2, 2023

1 Sorting in Topological Order

[16]:
```python
import networkx as nx

# Create an empty directed acyclic graph (DAG)
g = nx.DiGraph()

# Read the netfile and add nodes and edges to the graph
filename = "c17.net"
try:
    with open(filename) as f:
        for line in f:
            parts = line.strip().split()
            if len(parts) < 4:
                raise ValueError("Invalid line: " + line)
            gate_id, gate_type, *inputs, output = parts
            g.add_node(output, gate_type=gate_type)
            for input_node in inputs:
                g.add_edge(input_node, output)
except FileNotFoundError:
    print("Error: File not found:", filename)
    exit(1)
except ValueError as e:
    print("Error:", e)
    exit(1)

# Get the topological order of the nodes
try:
    topological_order = list(nx.topological_sort(g))
    print(topological_order)
except nx.exception.NetworkXUnfeasible:
    print("Error: Graph contains a cycle, so the circuit can't be evaluated.")
```

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

This Python code uses the NetworkX library to read in a netlist file that describes a digital circuit, and then constructs a directed acyclic graph (DAG) representing the circuit. The DAG is used to determine the topological order of the nodes, which can be used to evaluate the circuit in a way that respects the dependencies between gates. The topological order is computed using the topological_sort() function from the NetworkX library. If the graph contains a cycle, the circuit can't be evaluated and an error message is printed.

2 Using topological sort and multiple rounds of circuit evaluations.

[17]:
```python
def evaluate_gate(gate_type, input_states_list):
    if gate_type == "and2":
        return int(all(input_states_list))
    elif gate_type == "nand2":
        return int(not all(input_states_list))
    elif gate_type == "or2":
        return int(any(input_states_list))
    elif gate_type == "nor2":
        return int(not any(input_states_list))
    elif gate_type == "xor2":
        return int(sum(input_states_list) % 2 == 1)
    elif gate_type == "xnor2":
        return int(sum(input_states_list) % 2 == 0)
    elif gate_type == "inv":
        return int(not input_states_list)
    elif gate_type == "buf":
        return int(input_states_list[0])
    else:
        raise ValueError("Invalid gate type")


count=1
# Read the list of input vectors
inputs = []
with open("c17.inputs") as f:

        inputs=f.readlines()
        l=len(inputs)-1
        prim=inputs[0].split()

# Evaluate the circuit and find the state of all nets
for j in range (1,l+1):
    input_vector=inputs[j].split()
    # Set the input states of the primary inputs
    states = {}
    for i, input_id in enumerate(prim):
        if g.in_degree(input_id) == 0:
            states[input_id] = int(input_vector[i])
    # Evaluate the circuit in topological order
    for gate_id in topological_order:
        if g.in_degree(gate_id) > 0:
            input_gate_id=list(g.predecessors(gate_id))
            input_state_list=[states[n] for n in input_gate_id]
            #input_states_list = [states[input_gate_id] for input_gate_id in␣
↪list(g.predecessors(gate_id))]
            gate = g.nodes[gate_id]['gate_type']
            states[gate_id] = evaluate_gate(gate, input_state_list)

    # Print the state of all nets



    print(f"List of states for input vector {count}")
    for node_id in sorted(g.nodes):
        print(states[node_id],end=" ")
    count+=1
    print("\n")
```

```
List of states for input vector 1
0 1 1 1 0 0 0 1 1 1 0

List of states for input vector 2
0 0 0 0 1 0 0 1 1 1 1

List of states for input vector 3
1 0 0 0 0 0 0 1 1 1 1

List of states for input vector 4
0 0 0 0 1 1 1 1 0 1 1

List of states for input vector 5
1 1 1 0 1 1 1 0 0 1 1

List of states for input vector 6
1 1 1 1 1 0 0 0 1 1 0

List of states for input vector 7
1 1 1 0 1 1 0 0 0 1 1

List of states for input vector 8
1 1 1 1 0 0 0 1 1 1 0

List of states for input vector 9
0 1 1 1 1 0 1 1 1 0 0

List of states for input vector 10
```

```
0 0 0 0 1 1 0 1 0 1 1
```

This Python code reads a list of input vectors for a digital circuit and evaluates the circuit using the input vectors. The circuit is represented as a directed acyclic graph (DAG) and is evaluated in topological order. The gate types and input states are used to determine the output state of each gate in the circuit. The state of each net in the circuit is then printed for each input vector.

3 Event driven evaluation

[18]:
```
import networkx as nx
from queue import Queue
filename = "c17.net"
try:
    with open(filename) as f:
        # Create an empty directed acyclic graph (DAG)
        g = nx.DiGraph()
        for line in f:
            parts = line.strip().split()
            if len(parts) < 4:
                raise ValueError("Invalid line: " + line)
            gate_id, gate_type, *inputs, output = parts
            g.add_node(output, gate_type=gate_type)
            for input_node in inputs:
                g.add_edge(input_node, output)
except FileNotFoundError:
    print("Error: File not found:", filename)
    exit(1)
except ValueError as e:
    print("Error:", e)
    exit(1)


# Function to evaluate the logic of a gate
def evaluate_gate(gate_type, input_states_list):
    if gate_type == "and2":
        return int(all(input_states_list))
    elif gate_type == "nand2":
        return int(not all(input_states_list))
    elif gate_type == "or2":
        return int(any(input_states_list))
    elif gate_type == "nor2":
        return int(not any(input_states_list))
    elif gate_type == "xor2":
        return int(sum(input_states_list) % 2 == 1)
    elif gate_type == "xnor2":
        return int(sum(input_states_list) % 2 == 0)
```

```python
    elif gate_type == "inv":
        return int(not input_states_list)
    elif gate_type == "buf":
        return int(input_states_list[0])
    else:
        raise ValueError("Invalid gate type")




# Read the input values for the primary inputs
with open("c17.inputs") as f:
    inputs = f.readlines()
    l = len(inputs) - 1
    primary_inputs = inputs[0].split()


input_vector=inputs[1].split()

queue = []
    # Initialize the queue with primary inputs
for node_id in g.nodes():
    queue.append(node_id)


    # Initialize the states of all gates to None
states = {node_id: None for node_id in g.nodes()}

for node_id in sorted(g.nodes):
    print(f"{node_id}",end=" ")

print("\n")

for i, input_id in enumerate(primary_inputs):
    states[input_id] = int(input_vector[i])

    #print(queue)
# Process the nodes in the queue
while queue:
        # Get the next node from the queue
    node_id = queue.pop(0)

        # Check if all input states are available
    if(states[node_id]==1 or states[node_id]==0):
        continue
    input_states_list = [states[input_gate_id] for input_gate_id in g.
↪predecessors(node_id)]
    if None in input_states_list:
            # Some input states are not available, so put this node back in the␣
↪queue
        queue.append(node_id)
    else:
            # Evaluate the output state of the gate
        gate = g.nodes[node_id]['gate_type']
        new_state = evaluate_gate(gate, input_states_list)

            # Update the state of the gate
        states[node_id] = new_state


print("List of states for input vector 1")
for node_id in sorted(g.nodes):
    print(states[node_id],end=" ")

print("\n")

count=2
for j in range(2,l+1):
    input_vector=inputs[j].split()

    for i, input_id in enumerate(primary_inputs):

        if(states[input_id]!=int(input_vector[i])):
            states[input_id]=int(input_vector[i])
            for succ_id in g.successors(input_id):
                states[succ_id]=None
                queue.append(succ_id)

    while queue:

        # Get the next node from the queue
        node_id = queue.pop(0)

        # Check if all input states are available
        if(states[node_id]==1 or states[node_id]==0):
            continue

        input_states_list = [states[input_gate_id] for input_gate_id in g.
↪predecessors(node_id)]
        if None in input_states_list:
            # Some input states are not available, so put this node back in the␣
↪queue
```

```
                queue.append(node_id)
            else:
                # Evaluate the output state of the gate
                gate = g.nodes[node_id]['gate_type']
                new_state = evaluate_gate(gate, input_states_list)

                # Update the state of the gate
                states[node_id] = new_state

                # Add the successors of the gate to the queue
            for succ_id in g.successors(node_id):
                states[succ_id]=None
                queue.append(succ_id)


        print(f"List of states for input vector {count}")
        for node_id in sorted(g.nodes):
            print(states[node_id],end=" ")
        count+=1
        print("\n")
```

```
N1 N2 N22 N23 N3 N6 N7 n_0 n_1 n_2 n_3

List of states for input vector 1
0 1 1 1 0 0 0 1 1 1 0

List of states for input vector 2
0 0 0 0 1 0 0 1 1 1 1

List of states for input vector 3
1 0 0 0 0 0 0 1 1 1 1

List of states for input vector 4
0 0 0 0 1 1 1 1 0 1 1

List of states for input vector 5
1 1 1 0 1 1 1 0 0 1 1

List of states for input vector 6
1 1 1 1 1 0 0 0 1 1 0

List of states for input vector 7
1 1 1 0 1 1 0 0 0 1 1

List of states for input vector 8
1 1 1 1 0 0 0 1 1 1 0

List of states for input vector 9
```

```
0 1 1 1 1 0 1 1 1 0 0

List of states for input vector 10
0 0 0 0 1 1 0 1 0 1 1
```

This code reads in a netlist file describing a digital logic circuit and an input file containing input values for the primary inputs of the circuit. It constructs a directed acyclic graph (DAG) representing the circuit and then simulates the circuit's behavior using a breadth-first search algorithm to evaluate the logic of each gate based on the states of its input signals. The code then prints the resulting output states for each input vector in the input file. The program evaluates each input vector by updating the states of each primary input, re-evaluating each gate's state, and then updating the states of the subsequent gates in the circuit based on the changes.

4 Difference in Time

The event-driven simulation approach generally takes less time than the topological sort approach, especially for large circuits with many inputs and gates.

In the topological sort approach, the circuit is evaluated in multiple rounds, where each round evaluates the outputs of gates whose inputs are available. This requires repeatedly iterating over the gates and inputs of the circuit, which can be time-consuming for large circuits. Additionally, there may be multiple valid orderings of gates in a given round, which can complicate the implementation of the topological sort algorithm.

In contrast, the event-driven simulation approach only evaluates gates when their inputs change, using a priority queue to keep track of the next gate to evaluate. This approach avoids the need for multiple rounds of evaluation and can be more efficient for large circuits with many gates and inputs, especially if many of the inputs do not change frequently.

However, the event-driven simulation approach may require more initial setup time to build the priority queue and initialize the state of the circuit, compared to the topological sort approach. Additionally, the event-driven simulation approach may be more complex to implement, especially for circuits with complex gate interactions and feedback loops.