Week4final

Shreya .S. Ramanujam EE21B126

March 1, 2023

The zip folder contains the original jupyter notebook (.ipynb), which can be executed either on local jupyter or on this server. It also contains the exported LaTeX version of the notebook, and the data files. I have made use of numpy, cmath and sys libraries in this notebook. I have also made use of the deque class.

0.1 Imports

[1]:
```
import numpy as np
import cmath
import sys
from collections import deque
```

1 c8

1.1 Reading input file

[2]:
```
f = open("c8.netlist", "r")
input0 = f.readlines() # reading the netlist line by line
f.close()
```

1.2 Constructing the graph and checking for cycles

[3]:
```
import networkx as nx
# create a DAG
g = nx.DiGraph()
inputdict = {}
gateprop = {}

for line in input0:
    tokens = line.split() # split into list of individual inputs
    if (tokens[1] != 'inv') and (tokens[1] != 'buf'):
        if tokens[2] not in gateprop: # gateprop is a dictionary that has a␣
↪node as key and has a list of all its successors as value
            gateprop[tokens[2]] = []
        if tokens[3] not in gateprop:
            gateprop[tokens[3]] = []
        gateprop[tokens[2]].append(tokens[4])
```

```
        gateprop[tokens[3]].append(tokens[4])
        g.add_edges_from([(f"{tokens[2]}", f"{tokens[4]}"), (f"{tokens[3]}", f"{tokens[4]}")]) # creating edges of the directed acyclic graph
        inputdict[f"{tokens[4]}"] = [f"{tokens[2]}", f"{tokens[3]}", f"{tokens[1]}"] # creating an input ddictionary which has node name as key and a list of its inputs and gate type as value
    else:
        if tokens[2] not in gateprop:
            gateprop[tokens[2]] = []
        gateprop[tokens[2]].append(tokens[3])
        g.add_edges_from([(f"{tokens[2]}", f"{tokens[3]}")])
        inputdict[f"{tokens[3]}"] = [f"{tokens[2]}", f"{tokens[1]}"] # creating input dictionary for not and buffer gates

# print(gateprop)
if not nx.is_directed_acyclic_graph(g): # checking if graph has cycle; if cyclic, exit, since evaluation not possible
    print("Cycle in graph!")
    sys.exit()
nl = list(nx.topological_sort(g)) # sorting the nodes in topological order
alpha = sorted(nl) # nodes in alphabetical order, used for future file writing
```

2 Topological sort evaluation

First, we read the input from the file and initialize `outputdict` to an empty dictionary for storing steady state outputs

[4]:

```
outputdict = {} # output dictionary which has keys as nodes and their final steady state values as values.

f = open("c8.inputs", "r")
inp = f.readlines()
f.close()

nodeorder = inp[0].split() # order of input nodes in the input file
```

Next, we define `topoeval` which sorts the given nodes in topological order, evaluates the nodes in this order and stores the outputs in `outputdict`. Finally, it writes these steady state values to the output file, `topooutput.txt`

[5]:

```
def topoeval(inp):
    nl = list(nx.topological_sort(g)) # sorting the nodes in topological order
    ft = open("topooutput8.txt", "w") # file to write output to
    for node in alpha:
        ft.write(f"{node} ") # first row of file has alphabetically ordered node names
    ft.write("\n")
```

```python
    for line in inp:
        tok = line.split()
        if tok == nodeorder: # skipping over the first line since it just gives column names
            continue
        for i in range(len(tok)):
            outputdict[f"{nodeorder[i]}"] = int(tok[i]) # initializing primary inputs from the given inputs file
        for i in range(len(nl)):
            if nl[i] not in inputdict: # if it is a primary input, then continue; final value is already in outputdict, nothing to evaluate
                continue
            if inputdict[nl[i]][1] == 'inv': # checking the gate value of each input and calculating the steady state output accordingly
                outputdict[nl[i]] = int(not(outputdict[(inputdict[nl[i]][0])]))
            elif inputdict[nl[i]][1] == 'buf':
                outputdict[nl[i]] = int((outputdict[(inputdict[nl[i]][0])]))
            else:
                gate = inputdict[nl[i]][2]
                a = outputdict[inputdict[nl[i]][0]]
                b = outputdict[inputdict[nl[i]][1]]
                if gate == 'nand2':
                    outputdict[nl[i]] = int(not(a and b))
                elif gate == 'and2':
                    outputdict[nl[i]] = int(a and b)
                elif gate == 'or2':
                    outputdict[nl[i]] = int(a or b)
                elif gate == 'nor2':
                    outputdict[nl[i]] = int(not(a or b))
                elif gate == 'xor2':
                    outputdict[nl[i]] = int((a and (not b)) or (b and (not a)))
                elif gate == 'xnor2':
                    outputdict[nl[i]] = int(not((a and (not b)) or (b and (not a))))
        for node in alpha:
            ft.write(f"{outputdict[node]} ")
        ft.write("\n")
    ft.close()
    # return(outputdict)
```

[6]:
```python
topoeval(inp)
%timeit topoeval(inp)
```

```
1.42 ms ± 74.2 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

3 Gate driven

Since we have already read the input file and initialized the output dictionary, we directly start by defining our function, `gatedriven`.

This function first reads in the input line by line, and for each new line, we check each input against its previous value. If an input has changed from its previous value, we update it in `outputdict` and add all its immediate successors to the processing queue (since they may change due to the change in this node).

We then process this queue until it is empty. We evaluate the element at the top of the queue, and if it changes from its previously known output, then add its successors to the processing queue. Finally, we write the values of `outputdict` to the output file, `gateoutput.txt`

[7]:
```
def gatedriven(inp1):
    f1 = open("gateoutput8.txt", "w") # file to write output to
    for node in alpha:
        f1.write(node + " ") # first line is node names in alphabetical order
    f1.write("\n")
    previnp = ["x" for i in range(len(alpha))] # initialize previous input to␣
↪garbage values initially

    task = deque()

    for line in inp1:
        vals = line.split()
        for i in range(len(vals)):
            if vals[i] != previnp[i]: # if any primary input changes..
                outputdict[nodeorder[i]] = int(vals[i]) # update the new value␣
↪of the input in the output dictionary, and...
                for ele in gateprop[nodeorder[i]]: # add all its immediately␣
↪connected outputs to the processing queue
                    task.append(ele)
                    # print(task.get())
        previnp = vals # now, current input becomes previous input for the next␣
↪iteration, so update it
        while(bool(task)): # while the processing queue is not empty
            currnode = task[0] # processing first element of the queue
            try:
                while(task[0] == currnode): # if multiple of the same node are␣
↪added consecutively to the queue, pop them out; these are redundant, and␣
↪will result in the same output
                    task.popleft()
            except:
                pass
            # print(currnode)
            prevoutput = outputdict[currnode] # storing the value of the node␣
↪before evaluating it for the changed inputs
```

```
            if inputdict[currnode][1] == 'inv': # check the gate type and␣
↪update steady state dictionary accordingly
                outputdict[currnode] =␣
↪int(not(outputdict[(inputdict[currnode][0])]))
            elif inputdict[currnode][1] == 'buf':
                outputdict[currnode] = int(outputdict[(inputdict[currnode][0])])
            else:
                gate = inputdict[currnode][2]
                a = int(outputdict[inputdict[currnode][0]])
                b = int(outputdict[inputdict[currnode][1]])
                if gate == 'nand2':
                    outputdict[currnode] = int(not(a and b))
                elif gate == 'and2':
                    outputdict[currnode] = int(a and b)
                elif gate == 'or2':
                    outputdict[currnode] = int(a or b)
                elif gate == 'nor2':
                    outputdict[currnode] = int(not(a or b))
                elif gate == 'xor2':
                    outputdict[currnode] = int((a and (not b)) or (b and (not␣
↪a)))
                elif gate == 'xnor2':
                    outputdict[currnode] = int(not((a and (not b)) or (b and␣
↪(not a))))
            if prevoutput != outputdict[currnode]: # only add successors to the␣
↪queue if the output has changed after processing this node again (from the␣
↪queue)
                try:
                    for ele in gateprop[currnode]:
                        task.append(ele)
                except:
                    pass
        for node in alpha: # writing outputs for the current input to the file
            f1.write(f"{outputdict[node]} ")
        f1.write("\n")

    f1.close()
```

[8]:
```
input1 = inp[1:]
gatedriven(input1)
%timeit gatedriven(input1)
```

```
1.71 ms ± 93.6 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

4 c17

4.1 Reading input file

[9]:
```python
f = open("c17.netlist", "r")
input0 = f.readlines() # reading the netlist line by line
f.close()
```

4.2 Constructing the graph and checking for cycles

[10]:
```python
import networkx as nx
# create a DAG
g = nx.DiGraph()
inputdict = {}
gateprop = {}

for line in input0:
    tokens = line.split() # split into list of individual inputs
    if (tokens[1] != 'inv') and (tokens[1] != 'buf'):
        if tokens[2] not in gateprop: # gateprop is a dictionary that has a node as key and has a list of all its successors as value
            gateprop[tokens[2]] = []
        if tokens[3] not in gateprop:
            gateprop[tokens[3]] = []
        gateprop[tokens[2]].append(tokens[4])
        gateprop[tokens[3]].append(tokens[4])
        g.add_edges_from([(f"{tokens[2]}", f"{tokens[4]}"), (f"{tokens[3]}", f"{tokens[4]}")]) # creating edges of the directed acyclic graph
        inputdict[f"{tokens[4]}"] = [f"{tokens[2]}", f"{tokens[3]}", f"{tokens[1]}"] # creating an input ddictionary which has node name as key and a list of its inputs and gate type as value
    else:
        if tokens[2] not in gateprop:
            gateprop[tokens[2]] = []
        gateprop[tokens[2]].append(tokens[3])
        g.add_edges_from([(f"{tokens[2]}", f"{tokens[3]}")])
        inputdict[f"{tokens[3]}"] = [f"{tokens[2]}", f"{tokens[1]}"] # creating input dictionary for not and buffer gates

# print(gateprop)
if not nx.is_directed_acyclic_graph(g): # checking if graph has cycle; if cyclic, exit, since evaluation not possible
    print("Cycle in graph!")
    sys.exit()
nl = list(nx.topological_sort(g)) # sorting the nodes in topological order
alpha = sorted(nl) # nodes in alphabetical order, used for future file writing
```

5 Topological sort evaluation

First, we read the input from the file and initialize `outputdict` to an empty dictionary for storing steady state outputs

[11]:
```
outputdict = {} # output dictionary which has keys as nodes and their final␣
 ↪steady state values as values.

f = open("c17.inputs", "r")
inp = f.readlines()
f.close()

nodeorder = inp[0].split() # order of input nodes in the input file
```

Next, we define `topoeval` which sorts the given nodes in topological order, evaluates the nodes in this order and stores the outputs in `outputdict`. Finally, it writes these steady state values to the output file, `topooutput.txt`

[12]:
```
def topoeval(inp):
    nl = list(nx.topological_sort(g)) # sorting the nodes in topological order
    ft = open("topooutput17.txt", "w") # file to write output to
    for node in alpha:
        ft.write(f"{node} ") # first row of file has alphabetically ordered␣
 ↪node names
    ft.write("\n")
    for line in inp:
        tok = line.split()
        if tok == nodeorder: # skipping over the first line since it just gives␣
 ↪column names
            continue
        for i in range(len(tok)):
            outputdict[f"{nodeorder[i]}"] = int(tok[i]) # initializing primary␣
 ↪inputs from the given inputs file
        for i in range(len(nl)):
            if nl[i] not in inputdict: # if it is a primary input, then␣
 ↪continue; final value is already in outputdict, nothing to evaluate
                continue
            if inputdict[nl[i]][1] == 'inv': # checking the gate value of each␣
 ↪input and calculating the steady state output accordingly
                outputdict[nl[i]] = int(not(outputdict[(inputdict[nl[i]][0])]))
            elif inputdict[nl[i]][1] == 'buf':
                outputdict[nl[i]] = int((outputdict[(inputdict[nl[i]][0])]))
            else:
                gate = inputdict[nl[i]][2]
                a = outputdict[inputdict[nl[i]][0]]
                b = outputdict[inputdict[nl[i]][1]]
                if gate == 'nand2':
                    outputdict[nl[i]] = int(not(a and b))
```

```
                elif gate == 'and2':
                    outputdict[nl[i]] = int(a and b)
                elif gate == 'or2':
                    outputdict[nl[i]] = int(a or b)
                elif gate == 'nor2':
                    outputdict[nl[i]] = int(not(a or b))
                elif gate == 'xor2':
                    outputdict[nl[i]] = int((a and (not b)) or (b and (not a)))
                elif gate == 'xnor2':
                    outputdict[nl[i]] = int(not((a and (not b)) or (b and (not␣
↪a))))
        for node in alpha:
            ft.write(f"{outputdict[node]} ")
        ft.write("\n")
    ft.close()
    # return(outputdict)
```

[13]:

```
topoeval(inp)
%timeit topoeval(inp)
```

```
867 µs ± 17.7 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

6 Gate driven

Since we have already read the input file and initialized the output dictionary, we directly start by defining our function, `gatedriven`.

This function first reads in the input line by line, and for each new line, we check each input against its previous value. If an input has changed from its previous value, we update it in `outputdict` and add all its immediate successors to the processing queue (since they may change due to the change in this node).

We then process this queue until it is empty. We evaluate the element at the top of the queue, and if it changes from its previously known output, then add its successors to the processing queue. Finally, we write the values of `outputdict` to the output file, `gateoutput.txt`

[14]:

```
def gatedriven(inp1):
    f1 = open("gateoutput17.txt", "w") # file to write output to
    for node in alpha:
        f1.write(node + " ") # first line is node names in alphabetical order
    f1.write("\n")
    previnp = ["x" for i in range(len(alpha))] # initialize previous input to␣
↪garbage values initially

    task = deque()

    for line in inp1:
        vals = line.split()
```

```
        for i in range(len(vals)):
            if vals[i] != previnp[i]: # if any primary input changes..
                outputdict[nodeorder[i]] = int(vals[i]) # update the new value of the input in the output dictionary, and...
                for ele in gateprop[nodeorder[i]]: # add all its immediately connected outputs to the processing queue
                    task.append(ele)
                    # print(task.get())
        previnp = vals # now, current input becomes previous input for the next iteration, so update it
        while(bool(task)): # while the processing queue is not empty
            currnode = task[0] # processing first element of the queue
            try:
                while(task[0] == currnode): # if multiple of the same node are added consecutively to the queue, pop them out; these are redundant, and will result in the same output
                    task.popleft()
            except:
                pass
            # print(currnode)
            prevoutput = outputdict[currnode] # storing the value of the node before evaluating it for the changed inputs
            if inputdict[currnode][1] == 'inv': # check the gate type and update steady state dictionary accordingly
                outputdict[currnode] = int(not(outputdict[(inputdict[currnode][0])]))
            elif inputdict[currnode][1] == 'buf':
                outputdict[currnode] = int(outputdict[(inputdict[currnode][0])])
            else:
                gate = inputdict[currnode][2]
                a = int(outputdict[inputdict[currnode][0]])
                b = int(outputdict[inputdict[currnode][1]])
                if gate == 'nand2':
                    outputdict[currnode] = int(not(a and b))
                elif gate == 'and2':
                    outputdict[currnode] = int(a and b)
                elif gate == 'or2':
                    outputdict[currnode] = int(a or b)
                elif gate == 'nor2':
                    outputdict[currnode] = int(not(a or b))
                elif gate == 'xor2':
                    outputdict[currnode] = int((a and (not b)) or (b and (not a)))
                elif gate == 'xnor2':
                    outputdict[currnode] = int(not((a and (not b)) or (b and (not a))))
```

```
            if prevoutput != outputdict[currnode]: # only add successors to the␣
↪queue if the output has changed after processing this node again (from the␣
↪queue)
                try:
                    for ele in gateprop[currnode]:
                        task.append(ele)
                except:
                    pass
        for node in alpha: # writing outputs for the current input to the file
            f1.write(f"{outputdict[node]} ")
        f1.write("\n")

    f1.close()
```

[15]:
```
input1 = inp[1:]
gatedriven(input1)
%timeit gatedriven(input1)
```

```
919 µs ± 30 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

7 c17_1

7.1 Reading input file

[16]:
```
f = open("c17_1.netlist", "r")
input0 = f.readlines() # reading the netlist line by line
f.close()
```

7.2 Constructing the graph and checking for cycles

[17]:
```
import networkx as nx
# create a DAG
g = nx.DiGraph()
inputdict = {}
gateprop = {}

for line in input0:
    tokens = line.split() # split into list of individual inputs
    if (tokens[1] != 'inv') and (tokens[1] != 'buf'):
        if tokens[2] not in gateprop: # gateprop is a dictionary that has a␣
↪node as key and has a list of all its successors as value
            gateprop[tokens[2]] = []
        if tokens[3] not in gateprop:
            gateprop[tokens[3]] = []
        gateprop[tokens[2]].append(tokens[4])
        gateprop[tokens[3]].append(tokens[4])
```

```
        g.add_edges_from([(f"{tokens[2]}", f"{tokens[4]}"), (f"{tokens[3]}",␣
↪f"{tokens[4]}")]) # creating edges of the directed acyclic graph
        inputdict[f"{tokens[4]}"] = [f"{tokens[2]}", f"{tokens[3]}",␣
↪f"{tokens[1]}"] # creating an input ddictionary which has node name as key␣
↪and a list of its inputs and gate type as value
    else:
        if tokens[2] not in gateprop:
            gateprop[tokens[2]] = []
        gateprop[tokens[2]].append(tokens[3])
        g.add_edges_from([(f"{tokens[2]}", f"{tokens[3]}")])
        inputdict[f"{tokens[3]}"] = [f"{tokens[2]}", f"{tokens[1]}"] # creating␣
↪input dictionary for not and buffer gates

# print(gateprop)
if not nx.is_directed_acyclic_graph(g): # checking if graph has cycle; if␣
↪cyclic, exit, since evaluation not possible
    print("Cycle in graph!")
    sys.exit()
nl = list(nx.topological_sort(g)) # sorting the nodes in topological order
alpha = sorted(nl) # nodes in alphabetical order, used for future file writing
```

```
Cycle in graph!
```

```
An exception has occurred, use %tb to see the full traceback.

SystemExit
```

```
/usr/local/lib/python3.9/dist-packages/IPython/core/interactiveshell.py:3450:
UserWarning: To exit: use 'exit', 'quit', or Ctrl-D.
  warn("To exit: use 'exit', 'quit', or Ctrl-D.", stacklevel=1)
```

As we can see, there is a cycle in the graph, so not possible to evaluate.

8 c432

8.1 Reading input file

[18]:
```
f = open("c432.netlist", "r")
input0 = f.readlines() # reading the netlist line by line
f.close()
```

8.2 Constructing the graph and checking for cycles

[19]:
```
import networkx as nx
# create a DAG
g = nx.DiGraph()
inputdict = {}
gateprop = {}

for line in input0:
    tokens = line.split() # split into list of individual inputs
    if (tokens[1] != 'inv') and (tokens[1] != 'buf'):
        if tokens[2] not in gateprop: # gateprop is a dictionary that has a␣
↪node as key and has a list of all its successors as value
            gateprop[tokens[2]] = []
        if tokens[3] not in gateprop:
            gateprop[tokens[3]] = []
        gateprop[tokens[2]].append(tokens[4])
        gateprop[tokens[3]].append(tokens[4])
        g.add_edges_from([(f"{tokens[2]}", f"{tokens[4]}"), (f"{tokens[3]}",␣
↪f"{tokens[4]}")]) # creating edges of the directed acyclic graph
        inputdict[f"{tokens[4]}"] = [f"{tokens[2]}", f"{tokens[3]}",␣
↪f"{tokens[1]}"] # creating an input ddictionary which has node name as key␣
↪and a list of its inputs and gate type as value
    else:
        if tokens[2] not in gateprop:
            gateprop[tokens[2]] = []
        gateprop[tokens[2]].append(tokens[3])
        g.add_edges_from([(f"{tokens[2]}", f"{tokens[3]}")])
        inputdict[f"{tokens[3]}"] = [f"{tokens[2]}", f"{tokens[1]}"] # creating␣
↪input dictionary for not and buffer gates

# print(gateprop)
if not nx.is_directed_acyclic_graph(g): # checking if graph has cycle; if␣
↪cyclic, exit, since evaluation not possible
    print("Cycle in graph!")
    sys.exit()
nl = list(nx.topological_sort(g)) # sorting the nodes in topological order
alpha = sorted(nl) # nodes in alphabetical order, used for future file writing
```

9 Topological sort evaluation

First, we read the input from the file and initialize `outputdict` to an empty dictionary for storing steady state outputs

[20]:
```
outputdict = {} # output dictionary which has keys as nodes and their final␣
↪steady state values as values.
```

```python
f = open("c432.inputs", "r")
inp = f.readlines()
f.close()

nodeorder = inp[0].split() # order of input nodes in the input file
```

Next, we define `topoeval` which sorts the given nodes in topological order, evaluates the nodes in this order and stores the outputs in `outputdict`. Finally, it writes these steady state values to the output file, `topooutput.txt`

[21]:
```python
def topoeval(inp):
    nl = list(nx.topological_sort(g)) # sorting the nodes in topological order
    ft = open("topooutput432.txt", "w") # file to write output to
    for node in alpha:
        ft.write(f"{node} ") # first row of file has alphabetically ordered␣
 ↪node names
    ft.write("\n")
    for line in inp:
        tok = line.split()
        if tok == nodeorder: # skipping over the first line since it just gives␣
 ↪column names
            continue
        for i in range(len(tok)):
            outputdict[f"{nodeorder[i]}"] = int(tok[i]) # initializing primary␣
 ↪inputs from the given inputs file
        for i in range(len(nl)):
            if nl[i] not in inputdict: # if it is a primary input, then␣
 ↪continue; final value is already in outputdict, nothing to evaluate
                continue
            if inputdict[nl[i]][1] == 'inv': # checking the gate value of each␣
 ↪input and calculating the steady state output accordingly
                outputdict[nl[i]] = int(not(outputdict[(inputdict[nl[i]][0])]))
            elif inputdict[nl[i]][1] == 'buf':
                outputdict[nl[i]] = int((outputdict[(inputdict[nl[i]][0])]))
            else:
                gate = inputdict[nl[i]][2]
                a = outputdict[inputdict[nl[i]][0]]
                b = outputdict[inputdict[nl[i]][1]]
                if gate == 'nand2':
                    outputdict[nl[i]] = int(not(a and b))
                elif gate == 'and2':
                    outputdict[nl[i]] = int(a and b)
                elif gate == 'or2':
                    outputdict[nl[i]] = int(a or b)
                elif gate == 'nor2':
                    outputdict[nl[i]] = int(not(a or b))
                elif gate == 'xor2':
```

```
                    outputdict[nl[i]] = int((a and (not b)) or (b and (not a)))
                elif gate == 'xnor2':
                    outputdict[nl[i]] = int(not((a and (not b)) or (b and (not␣
↪a))))
        for node in alpha:
            ft.write(f"{outputdict[node]} ")
        ft.write("\n")
    ft.close()
    # return(outputdict)
```

[22]:
```
topoeval(inp)
%timeit topoeval(inp)
```

```
18 ms ± 817 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

10 Gate driven

Since we have already read the input file and initialized the output dictionary, we directly start by defining our function, `gatedriven`.

This function first reads in the input line by line, and for each new line, we check each input against its previous value. If an input has changed from its previous value, we update it in `outputdict` and add all its immediate successors to the processing queue (since they may change due to the change in this node).

We then process this queue until it is empty. We evaluate the element at the top of the queue, and if it changes from its previously known output, then add its successors to the processing queue. Finally, we write the values of `outputdict` to the output file, `gateoutput.txt`

[23]:
```
def gatedriven(inp1):
    f1 = open("gateoutput432.txt", "w") # file to write output to
    for node in alpha:
        f1.write(node + " ") # first line is node names in alphabetical order
    f1.write("\n")
    previnp = ["x" for i in range(len(alpha))] # initialize previous input to␣
↪garbage values initially

    task = deque()

    for line in inp1:
        vals = line.split()
        for i in range(len(vals)):
            if vals[i] != previnp[i]: # if any primary input changes..
                outputdict[nodeorder[i]] = int(vals[i]) # update the new value␣
↪of the input in the output dictionary, and...
                for ele in gateprop[nodeorder[i]]: # add all its immediately␣
↪connected outputs to the processing queue
                    task.append(ele)
```

```python
                    # print(task.get())
        previnp = vals # now, current input becomes previous input for the next iteration, so update it
        while(bool(task)): # while the processing queue is not empty
            currnode = task[0] # processing first element of the queue
            try:
                while(task[0] == currnode): # if multiple of the same node are added consecutively to the queue, pop them out; these are redundant, and will result in the same output
                    task.popleft()
            except:
                pass
            # print(currnode)
            prevoutput = outputdict[currnode] # storing the value of the node before evaluating it for the changed inputs
            if inputdict[currnode][1] == 'inv': # check the gate type and update steady state dictionary accordingly
                outputdict[currnode] = int(not(outputdict[(inputdict[currnode][0])]))
            elif inputdict[currnode][1] == 'buf':
                outputdict[currnode] = int(outputdict[(inputdict[currnode][0])])
            else:
                gate = inputdict[currnode][2]
                a = int(outputdict[inputdict[currnode][0]])
                b = int(outputdict[inputdict[currnode][1]])
                if gate == 'nand2':
                    outputdict[currnode] = int(not(a and b))
                elif gate == 'and2':
                    outputdict[currnode] = int(a and b)
                elif gate == 'or2':
                    outputdict[currnode] = int(a or b)
                elif gate == 'nor2':
                    outputdict[currnode] = int(not(a or b))
                elif gate == 'xor2':
                    outputdict[currnode] = int((a and (not b)) or (b and (not a)))
                elif gate == 'xnor2':
                    outputdict[currnode] = int(not((a and (not b)) or (b and (not a))))
            if prevoutput != outputdict[currnode]: # only add successors to the queue if the output has changed after processing this node again (from the queue)
                try:
                    for ele in gateprop[currnode]:
                        task.append(ele)
                except:
```

```
                    pass
            for node in alpha: # writing outputs for the current input to the file
                f1.write(f"{outputdict[node]} ")
            f1.write("\n")

        f1.close()
```

[24]:
```
input1 = inp[1:]
gatedriven(input1)
%timeit gatedriven(input1)
```

26.8 ms ± 736 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

10.1 Reading input file

11 parity

11.1 Reading input file

[25]:
```
f = open("parity.netlist", "r")
input0 = f.readlines() # reading the netlist line by line
f.close()
```

11.2 Constructing the graph and checking for cycles

[26]:
```
import networkx as nx
# create a DAG
g = nx.DiGraph()
inputdict = {}
gateprop = {}

for line in input0:
    tokens = line.split() # split into list of individual inputs
    if (tokens[1] != 'inv') and (tokens[1] != 'buf'):
        if tokens[2] not in gateprop: # gateprop is a dictionary that has a␣
 ↪node as key and has a list of all its successors as value
            gateprop[tokens[2]] = []
        if tokens[3] not in gateprop:
            gateprop[tokens[3]] = []
        gateprop[tokens[2]].append(tokens[4])
        gateprop[tokens[3]].append(tokens[4])
        g.add_edges_from([(f"{tokens[2]}", f"{tokens[4]}"), (f"{tokens[3]}",␣
 ↪f"{tokens[4]}")]) # creating edges of the directed acyclic graph
        inputdict[f"{tokens[4]}"] = [f"{tokens[2]}", f"{tokens[3]}",␣
 ↪f"{tokens[1]}"] # creating an input ddictionary which has node name as key␣
 ↪and a list of its inputs and gate type as value
    else:
```

```python
        if tokens[2] not in gateprop:
            gateprop[tokens[2]] = []
        gateprop[tokens[2]].append(tokens[3])
        g.add_edges_from([(f"{tokens[2]}", f"{tokens[3]}")])
        inputdict[f"{tokens[3]}"] = [f"{tokens[2]}", f"{tokens[1]}"] # creating input dictionary for not and buffer gates

# print(gateprop)
if not nx.is_directed_acyclic_graph(g): # checking if graph has cycle; if cyclic, exit, since evaluation not possible
    print("Cycle in graph!")
    sys.exit()
nl = list(nx.topological_sort(g)) # sorting the nodes in topological order
alpha = sorted(nl) # nodes in alphabetical order, used for future file writing
```

12 Topological sort evaluation

First, we read the input from the file and initialize `outputdict` to an empty dictionary for storing steady state outputs

[27]:
```python
outputdict = {} # output dictionary which has keys as nodes and their final steady state values as values.

f = open("parity.inputs", "r")
inp = f.readlines()
f.close()

nodeorder = inp[0].split() # order of input nodes in the input file
```

Next, we define `topoeval` which sorts the given nodes in topological order, evaluates the nodes in this order and stores the outputs in `outputdict`. Finally, it writes these steady state values to the output file, `topooutput.txt`

[28]:
```python
def topoeval(inp):
    nl = list(nx.topological_sort(g)) # sorting the nodes in topological order
    ft = open("topooutputp.txt", "w") # file to write output to
    for node in alpha:
        ft.write(f"{node} ") # first row of file has alphabetically ordered node names
    ft.write("\n")
    for line in inp:
        tok = line.split()
        if tok == nodeorder: # skipping over the first line since it just gives column names
            continue
        for i in range(len(tok)):
```

```
            outputdict[f"{nodeorder[i]}"] = int(tok[i]) # initializing primary␣
↪inputs from the given inputs file
        for i in range(len(nl)):
            if nl[i] not in inputdict: # if it is a primary input, then␣
↪continue; final value is already in outputdict, nothing to evaluate
                continue
            if inputdict[nl[i]][1] == 'inv': # checking the gate value of each␣
↪input and calculating the steady state output accordingly
                outputdict[nl[i]] = int(not(outputdict[(inputdict[nl[i]][0])]))
            elif inputdict[nl[i]][1] == 'buf':
                outputdict[nl[i]] = int((outputdict[(inputdict[nl[i]][0])]))
            else:
                gate = inputdict[nl[i]][2]
                a = outputdict[inputdict[nl[i]][0]]
                b = outputdict[inputdict[nl[i]][1]]
                if gate == 'nand2':
                    outputdict[nl[i]] = int(not(a and b))
                elif gate == 'and2':
                    outputdict[nl[i]] = int(a and b)
                elif gate == 'or2':
                    outputdict[nl[i]] = int(a or b)
                elif gate == 'nor2':
                    outputdict[nl[i]] = int(not(a or b))
                elif gate == 'xor2':
                    outputdict[nl[i]] = int((a and (not b)) or (b and (not a)))
                elif gate == 'xnor2':
                    outputdict[nl[i]] = int(not((a and (not b)) or (b and (not␣
↪a))))
        for node in alpha:
            ft.write(f"{outputdict[node]} ")
        ft.write("\n")
    ft.close()
    # return(outputdict)
```

[29]:
```
topoeval(inp)
%timeit topoeval(inp)
```

```
920 µs ± 22.4 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

13 Gate driven

Since we have already read the input file and initialized the output dictionary, we directly start by defining our function, `gatedriven`.

This function first reads in the input line by line, and for each new line, we check each input against its previous value. If an input has changed from its previous value, we update it in `outputdict` and add all its immediate successors to the processing queue (since they may change due to the change in this node).

We then process this queue until it is empty. We evaluate the element at the top of the queue, and if it changes from its previously known output, then add its successors to the processing queue. Finally, we write the values of `outputdict` to the output file, `gateoutput.txt`

[30]:
```
def gatedriven(inp1):
    f1 = open("gateoutputp.txt", "w") # file to write output to
    for node in alpha:
        f1.write(node + " ") # first line is node names in alphabetical order
    f1.write("\n")
    previnp = ["x" for i in range(len(alpha))] # initialize previous input to␣
 ↪garbage values initially

    task = deque()

    for line in inp1:
        vals = line.split()
        for i in range(len(vals)):
            if vals[i] != previnp[i]: # if any primary input changes..
                outputdict[nodeorder[i]] = int(vals[i]) # update the new value␣
 ↪of the input in the output dictionary, and...
                for ele in gateprop[nodeorder[i]]: # add all its immediately␣
 ↪connected outputs to the processing queue
                    task.append(ele)
                    # print(task.get())
        previnp = vals # now, current input becomes previous input for the next␣
 ↪iteration, so update it
        while(bool(task)): # while the processing queue is not empty
            currnode = task[0] # processing first element of the queue
            try:
                while(task[0] == currnode): # if multiple of the same node are␣
 ↪added consecutively to the queue, pop them out; these are redundant, and␣
 ↪will result in the same output
                    task.popleft()
            except:
                pass
            # print(currnode)
            prevoutput = outputdict[currnode] # storing the value of the node␣
 ↪before evaluating it for the changed inputs
            if inputdict[currnode][1] == 'inv': # check the gate type and␣
 ↪update steady state dictionary accordingly
                outputdict[currnode] =␣
 ↪int(not(outputdict[(inputdict[currnode][0])]))
            elif inputdict[currnode][1] == 'buf':
                outputdict[currnode] = int(outputdict[(inputdict[currnode][0])])
            else:
                gate = inputdict[currnode][2]
                a = int(outputdict[inputdict[currnode][0]])
```

```
                b = int(outputdict[inputdict[currnode][1]])
                if gate == 'nand2':
                    outputdict[currnode] = int(not(a and b))
                elif gate == 'and2':
                    outputdict[currnode] = int(a and b)
                elif gate == 'or2':
                    outputdict[currnode] = int(a or b)
                elif gate == 'nor2':
                    outputdict[currnode] = int(not(a or b))
                elif gate == 'xor2':
                    outputdict[currnode] = int((a and (not b)) or (b and (not␣
↪a)))
                elif gate == 'xnor2':
                    outputdict[currnode] = int(not((a and (not b)) or (b and␣
↪(not a))))
            if prevoutput != outputdict[currnode]: # only add successors to the␣
↪queue if the output has changed after processing this node again (from the␣
↪queue)
                try:
                    for ele in gateprop[currnode]:
                        task.append(ele)
                except:
                    pass
        for node in alpha: # writing outputs for the current input to the file
            f1.write(f"{outputdict[node]} ")
        f1.write("\n")

    f1.close()
```

[31]:
```
input1 = inp[1:]
gatedriven(input1)
%timeit gatedriven(input1)
```

```
950 µs ± 74.8 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

14 Conclusions:

As we can see, the topological sort is faster in all the cases. This could possibly be because the event driven approach is optimal in cases where the number of changes in the inputs for consecutive time stamps is small.

However, this is not the case for our inputs, since a significant number of inputs change for consecutive time stamps, which would increase redundancy in queueing and evaluation. However, the topo sort evaluates each node only once in any case. This is why event driven is slower than topo sort for our inputs, since we evaluate each node too many times.

One way to solve this problem would be to optimize the event driven inputs by preprocessing them, and arranging the inputs such that consecutive inputs have the maximum similarity, i.e. the least number of input changes. This would reduce the queue size and redundant evaluation, thus significantly speeding up the event driven approach.

Overall, event driven would be faster for more similar inputs (ie less changes in consecutive input lines), whereas for significantly dissimilar inputs, topological sort reduces redundancy, and hence, is optimal.

[32]:
```python
f = open("c17.netlist", "r")
input0 = f.readlines() # reading the netlist line by line
f.close()
```

First, we read the input from the file and initialize `outputdict` to an empty dictionary for storing steady state outputs

[ ]:
```python
outputdict = {} # output dictionary which has keys as nodes and their final␣
↪steady state values as values.

f = open("c17inputs.txt", "r")
inp = f.readlines()
f.close()

nodeorder = inp[0].split() # order of input nodes in the input file
```

Next, we define `topoeval` which sorts the given nodes in topological order, evaluates the nodes in this order and stores the outputs in `outputdict`. Finally, it writes these steady state values to the output file, `topooutput.txt`

[ ]:
```python
def topoeval(inp):
    nl = list(nx.topological_sort(g)) # sorting the nodes in topological order
    ft = open("topooutput.txt", "w") # file to write output to
    for node in alpha:
        ft.write(f"{node} ") # first row of file has alphabetically ordered␣
↪node names
    ft.write("\n")
    for line in inp:
        tok = line.split()
        if tok == nodeorder: # skipping over the first line since it just gives␣
↪column names
            continue
        for i in range(len(tok)):
            outputdict[f"{nodeorder[i]}"] = int(tok[i]) # initializing primary␣
↪inputs from the given inputs file
        for i in range(len(nl)):
            if nl[i] not in inputdict: # if it is a primary input, then␣
↪continue; final value is already in outputdict, nothing to evaluate
                continue
            if inputdict[nl[i]][1] == 'inv': # checking the gate value of each␣
↪input and calculating the steady state output accordingly
                outputdict[nl[i]] = int(not(outputdict[(inputdict[nl[i]][0])]))
```

```
            elif inputdict[nl[i]][1] == 'buf':
                outputdict[nl[i]] = int((outputdict[(inputdict[nl[i]][0])]))
            else:
                gate = inputdict[nl[i]][2]
                a = outputdict[inputdict[nl[i]][0]]
                b = outputdict[inputdict[nl[i]][1]]
                if gate == 'nand2':
                    outputdict[nl[i]] = int(not(a and b))
                elif gate == 'and2':
                    outputdict[nl[i]] = int(a and b)
                elif gate == 'or2':
                    outputdict[nl[i]] = int(a or b)
                elif gate == 'nor2':
                    outputdict[nl[i]] = int(not(a or b))
                elif gate == 'xor2':
                    outputdict[nl[i]] = int((a and (not b)) or (b and (not a)))
                elif gate == 'xnor2':
                    outputdict[nl[i]] = int(not((a and (not b)) or (b and (not␣
↪a))))
        for node in alpha:
            ft.write(f"{outputdict[node]} ")
        ft.write("\n")
    ft.close()
    # return(outputdict)
```

[ ]:
```
topoeval(inp)
%timeit topoeval(inp)
```

Since we have already read the input file and initialized the output dictionary, we directly start by defining our function, `gatedriven`.

This function first reads in the input line by line, and for each new line, we check each input against its previous value. If an input has changed from its previous value, we update it in `outputdict` and add all its immediate successors to the processing queue (since they may change due to the change in this node).

We then process this queue until it is empty. We evaluate the element at the top of the queue, and if it changes from its previously known output, then add its successors to the processing queue. Finally, we write the values of `outputdict` to the output file, `gateoutput.txt`

[ ]:
```
def gatedriven(inp1):
    f1 = open("gateoutput.txt", "w") # file to write output to
    for node in alpha:
        f1.write(node + " ") # first line is node names in alphabetical order
    f1.write("\n")
    previnp = ["x" for i in range(len(alpha))] # initialize previous input to␣
↪garbage values initially
```

```python
    task = deque()

    for line in inp1:
        vals = line.split()
        for i in range(len(vals)):
            if vals[i] != previnp[i]: # if any primary input changes..
                outputdict[nodeorder[i]] = int(vals[i]) # update the new value of the input in the output dictionary, and...
                for ele in gateprop[nodeorder[i]]: # add all its immediately connected outputs to the processing queue
                    task.append(ele)
                    # print(task.get())
        previnp = vals # now, current input becomes previous input for the next iteration, so update it
        while(bool(task)): # while the processing queue is not empty
            currnode = task[0] # processing first element of the queue
            try:
                while(task[0] == currnode): # if multiple of the same node are added consecutively to the queue, pop them out; these are redundant, and will result in the same output
                    task.popleft()
            except:
                pass
            # print(currnode)
            prevoutput = outputdict[currnode] # storing the value of the node before evaluating it for the changed inputs
            if inputdict[currnode][1] == 'inv': # check the gate type and update steady state dictionary accordingly
                outputdict[currnode] = int(not(outputdict[(inputdict[currnode][0])]))
            elif inputdict[currnode][1] == 'buf':
                outputdict[currnode] = int(outputdict[(inputdict[currnode][0])])
            else:
                gate = inputdict[currnode][2]
                a = int(outputdict[inputdict[currnode][0]])
                b = int(outputdict[inputdict[currnode][1]])
                if gate == 'nand2':
                    outputdict[currnode] = int(not(a and b))
                elif gate == 'and2':
                    outputdict[currnode] = int(a and b)
                elif gate == 'or2':
                    outputdict[currnode] = int(a or b)
                elif gate == 'nor2':
                    outputdict[currnode] = int(not(a or b))
                elif gate == 'xor2':
```

```
                    outputdict[currnode] = int((a and (not b)) or (b and (not␣
↪a)))
                elif gate == 'xnor2':
                    outputdict[currnode] = int(not((a and (not b)) or (b and␣
↪(not a))))
            if prevoutput != outputdict[currnode]: # only add successors to the␣
↪queue if the output has changed after processing this node again (from the␣
↪queue)
                try:
                    for ele in gateprop[currnode]:
                        task.append(ele)
                except:
                    pass
        for node in alpha: # writing outputs for the current input to the file
            f1.write(f"{outputdict[node]} ")
        f1.write("\n")

    f1.close()
```

[ ]:
```
input1 = inp[1:]
gatedriven(input1)
%timeit gatedriven(input1)
```

First, we read the input from the file and initialize `outputdict` to an empty dictionary for storing steady state outputs

[ ]:
```
outputdict = {} # output dictionary which has keys as nodes and their final␣
↪steady state values as values.

f = open("c17inputs.txt", "r")
inp = f.readlines()
f.close()

nodeorder = inp[0].split() # order of input nodes in the input file
```

Next, we define `topoeval` which sorts the given nodes in topological order, evaluates the nodes in this order and stores the outputs in `outputdict`. Finally, it writes these steady state values to the output file, `topooutput.txt`

[ ]:
```
def topoeval(inp):
    nl = list(nx.topological_sort(g)) # sorting the nodes in topological order
    ft = open("topooutput.txt", "w") # file to write output to
    for node in alpha:
        ft.write(f"{node} ") # first row of file has alphabetically ordered␣
↪node names
    ft.write("\n")
    for line in inp:
        tok = line.split()
```

```
        if tok == nodeorder: # skipping over the first line since it just gives␣
↪column names
            continue
        for i in range(len(tok)):
            outputdict[f"{nodeorder[i]}"] = int(tok[i]) # initializing primary␣
↪inputs from the given inputs file
        for i in range(len(nl)):
            if nl[i] not in inputdict: # if it is a primary input, then␣
↪continue; final value is already in outputdict, nothing to evaluate
                continue
            if inputdict[nl[i]][1] == 'inv': # checking the gate value of each␣
↪input and calculating the steady state output accordingly
                outputdict[nl[i]] = int(not(outputdict[(inputdict[nl[i]][0])]))
            elif inputdict[nl[i]][1] == 'buf':
                outputdict[nl[i]] = int((outputdict[(inputdict[nl[i]][0])]))
            else:
                gate = inputdict[nl[i]][2]
                a = outputdict[inputdict[nl[i]][0]]
                b = outputdict[inputdict[nl[i]][1]]
                if gate == 'nand2':
                    outputdict[nl[i]] = int(not(a and b))
                elif gate == 'and2':
                    outputdict[nl[i]] = int(a and b)
                elif gate == 'or2':
                    outputdict[nl[i]] = int(a or b)
                elif gate == 'nor2':
                    outputdict[nl[i]] = int(not(a or b))
                elif gate == 'xor2':
                    outputdict[nl[i]] = int((a and (not b)) or (b and (not a)))
                elif gate == 'xnor2':
                    outputdict[nl[i]] = int(not((a and (not b)) or (b and (not␣
↪a))))
        for node in alpha:
            ft.write(f"{outputdict[node]} ")
        ft.write("\n")
    ft.close()
    # return(outputdict)
```

[ ]:
```
topoeval(inp)
%timeit topoeval(inp)
```