Untitled18

July 8, 2023

[6]:
```
from main import main
%load_ext memory_profiler
%mprun -f  main main()
```

```
Time taken is 0.0008583068847656250
The memory_profiler extension is already loaded. To reload it, use:
  %reload_ext memory_profiler
Time taken is 0.3983745574951172


Filename: /home/btech/ee21b015/main.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
     6     76.5 MiB     76.5 MiB           1   def main():
     7     76.5 MiB      0.0 MiB           1       g = nx.DiGraph()
     8     76.5 MiB      0.0 MiB           1       with open("Circuit File.txt")␣
 ↪as fo : #To collect data from the circuit netlist given
     9     76.5 MiB      0.0 MiB           1           lines = fo.readlines()
    10
    11     76.5 MiB      0.0 MiB           1       inputs = set() #A set to hold␣
 ↪all the input nodes (all but Z in this case)
    12     76.5 MiB      0.0 MiB           1       ip = []
    13     76.5 MiB      0.0 MiB           1       outputs = {} #dictionary to␣
 ↪hold all the input output pairs
    14
    15     76.5 MiB      0.0 MiB           1       with open("Fault.txt") as ff :
 ↪ #Reading the type of fault from the file given
    16     76.5 MiB      0.0 MiB           1           fault = ff.readlines()
    17     76.5 MiB      0.0 MiB           1       node_fault=fault[0].
 ↪split()[2] #Storing the fault node
    18     76.5 MiB      0.0 MiB           1       type_fault=fault[1].
 ↪split()[2][2] #Storing the fault type
    19
    20
    21     76.5 MiB      0.0 MiB           5       for indline in lines :
 ↪#Generating a DAG, Input set, Output Dictionary
    22     76.5 MiB      0.0 MiB           4           line = indline.split()
  23     76.5 MiB      0.0 MiB           4           if len(line) == 4 and␣
↪line[2]=='~': #Handling the not gate separately due to different input style
  24     76.5 MiB      0.0 MiB           1               i=line[-1]
  25     76.5 MiB      0.0 MiB           1               o=line[0]
  26     76.5 MiB      0.0 MiB           1               inputs.add(i)
  27     76.5 MiB      0.0 MiB           1               outputs[o] = line[2]
  28     76.5 MiB      0.0 MiB           1               g.add_edge(i,o)
  29                                                 else: #Handling all other␣
↪gates except the NOT gate
  30     76.5 MiB      0.0 MiB           3               i1= line[2]
  31     76.5 MiB      0.0 MiB           3               i2= line[4]
  32     76.5 MiB      0.0 MiB           3               o= line[0]
  33     76.5 MiB      0.0 MiB           3               inputs.add(i1)
  34     76.5 MiB      0.0 MiB           3               inputs.add(i2)
  35     76.5 MiB      0.0 MiB           3               outputs[o] = line[3]
  36     76.5 MiB      0.0 MiB           3               g.add_edge(i1,o)
  37     76.5 MiB      0.0 MiB           3               g.add_edge(i2,o)
  38
  39     76.5 MiB      0.0 MiB           8       for inp in inputs :
  40     76.5 MiB      0.0 MiB           7           if inp in outputs.keys() :
  41     76.5 MiB      0.0 MiB           3               continue
  42                                             else :
  43     76.5 MiB      0.0 MiB           4               ip.append(inp) #ip is␣
↪a list that holds just the primary inputs (A,B,C,D)
  44
  45     76.5 MiB      0.0 MiB           5       for inp in ip : #We set the␣
↪gatetypes of the A,B,C,D nodes to Primaryinput type
  46     76.5 MiB      0.0 MiB           4           g.nodes[inp]["gateType"]␣
↪= "PrimaryInput"
  47     76.5 MiB      0.0 MiB           5       for out in outputs:#For the␣
↪other nodes except the first 4, We set the gatetypes to the corresponding␣
↪Output nodes
  48     76.5 MiB      0.0 MiB           4           g.nodes[out]["gateType"]␣
↪= outputs[out]
  49     76.5 MiB      0.0 MiB           1       n1 = list(nx.
↪topological_sort(g)) #n1 contains all the nodes in the order in which they␣
↪appear in the graph
  50
  51
  52     76.5 MiB      0.0 MiB           9       for node in n1:
  53     76.5 MiB      0.0 MiB           8           g.nodes[node]['value'] =␣
↪0 #We initialise all nodes with a value of 0
  54     76.5 MiB      0.0 MiB           1       f=g.copy() #We create a deep␣
↪copy of this graph for the second DAG evaluation
  55
  56
  57     76.5 MiB      0.0 MiB           1       l1= ["A","B","C","D"]  #l1␣
↪holds the primary inputs
  58     76.5 MiB      0.0 MiB           1       l2=␣
↪list(set(permutations([0,0,0,0,1,1,1,1], 4))) #We are testing all 16 possible␣
↪inputs, and hence we create a permutation such that all 16 are created
  59     76.5 MiB      0.0 MiB           1       l3=[]
  60
  61                                             #Below we have defined the␣
↪logical Operations encountered and the outputs for alll cases
  62     76.5 MiB      0.0 MiB          29       def AND(a, b):
  63     76.5 MiB      0.0 MiB          28           return a*b
  64     76.5 MiB      0.0 MiB          41       def OR(a,b) :
  65     76.5 MiB      0.0 MiB          40           if a+b == 0 :
  66     76.5 MiB      0.0 MiB          10               return 0
  67                                                 else :
  68     76.5 MiB      0.0 MiB          30               return 1
  69     76.5 MiB      0.0 MiB          41       def XOR(a,b) :
  70     76.5 MiB      0.0 MiB          40           if (a==1 and b==0) or␣
↪(a==0 and b==1):
  71     76.5 MiB      0.0 MiB          18               return 1
  72                                                 else :
  73     76.5 MiB      0.0 MiB          22               return 0
  74     76.5 MiB      0.0 MiB          41       def NOT(a):
  75     76.5 MiB      0.0 MiB          40           if a == 1:
  76     76.5 MiB      0.0 MiB          30               return 0
  77                                                 else :
  78     76.5 MiB      0.0 MiB          10               return 1
  79
  80                                             #This update code is for the␣
↪first DAG evaluation. This takes a node and checks the predecessor and␣
↪gatetype to change the value in the node in graph g
  81     76.5 MiB      0.0 MiB         113       def update(node):
  82     76.5 MiB      0.0 MiB         112           ip = list(g.
↪predecessors(node))
  83     76.5 MiB      0.0 MiB         112           ips = []
  84     76.5 MiB      0.0 MiB         308           for i in ip:
  85     76.5 MiB      0.0 MiB         196               ips.append(g.
↪nodes[i]['value'])
  86     76.5 MiB      0.0 MiB         112           if g.
↪nodes[node]['gateType'] == "&":
  87     76.5 MiB      0.0 MiB          28               g.
↪nodes[node]['value'] =  AND(ips[0], ips[1])
  88     76.5 MiB      0.0 MiB         112           if g.
↪nodes[node]['gateType'] == "|" :
  89     76.5 MiB      0.0 MiB          28               g.
↪nodes[node]['value'] =  OR(ips[0], ips[1])
```

```
  90     76.5 MiB      0.0 MiB         112            if g.
↪nodes[node]['gateType'] == "^":
  91     76.5 MiB      0.0 MiB          28                g.
↪nodes[node]['value'] =  XOR(ips[0], ips[1])
  92     76.5 MiB      0.0 MiB         112            if g.
↪nodes[node]['gateType'] == "~":
  93     76.5 MiB      0.0 MiB          28                g.
↪nodes[node]['value'] =  NOT(ips[0])
  94                                              #This update code is for the␣
↪second DAG evaluation. This takes a node and checks the predecessor and␣
↪gatetype to change the value in the node in graph f
  95     76.5 MiB      0.0 MiB          37        def fault_update(node):
  96     76.5 MiB      0.0 MiB          36            ip = list(f.
↪predecessors(node))
  97     76.5 MiB      0.0 MiB          36            ips = []
  98     76.5 MiB      0.0 MiB          96            for i in ip:
  99     76.5 MiB      0.0 MiB          60                ips.append(f.
↪nodes[i]['value'])
 100     76.5 MiB      0.0 MiB          36            if f.
↪nodes[node]['gateType'] == "&":
 101                                                      f.
↪nodes[node]['value'] =  AND(ips[0], ips[1])
 102     76.5 MiB      0.0 MiB          36            if f.
↪nodes[node]['gateType'] == "|" :
 103     76.5 MiB      0.0 MiB          12                f.
↪nodes[node]['value'] =  OR(ips[0], ips[1])
 104     76.5 MiB      0.0 MiB          36            if f.
↪nodes[node]['gateType'] == "^":
 105     76.5 MiB      0.0 MiB          12                f.
↪nodes[node]['value'] =  XOR(ips[0], ips[1])
 106     76.5 MiB      0.0 MiB          36            if f.
↪nodes[node]['gateType'] == "~":
 107     76.5 MiB      0.0 MiB          12                f.
↪nodes[node]['value'] =  NOT(ips[0])
 108
 109                                              #This DAG function sends the␣
↪node to DAG evaluation, provided it is not a Primary input
 110     76.5 MiB      0.0 MiB          29        def DAG():
 111     76.5 MiB      0.0 MiB         252            for node in n1 :
 112     76.5 MiB      0.0 MiB         224                if g.
↪nodes[node]['gateType'] == "PrimaryInput" :
 113     76.5 MiB      0.0 MiB         112                    continue
 114                                                      else :
 115     76.5 MiB      0.0 MiB         112                    update(node)
```

```
116                                             #This DAG function checks if the node is not the predecessor of the fault node, and that it is not primary, and sends the node to DAG evaluation
117      76.5 MiB      0.0 MiB          13       def fault_DAG():
118      76.5 MiB      0.0 MiB         108           for node in n1 :
119      76.5 MiB      0.0 MiB          96               if node not in pred:
120
121      76.5 MiB      0.0 MiB          72                   if f.nodes[node]['gateType'] == "PrimaryInput":
122      76.5 MiB      0.0 MiB          24                       continue
123      76.5 MiB      0.0 MiB          48                   elif node==node_fault:
124      76.5 MiB      0.0 MiB          12                       f.nodes[node]['value'] = int(type_fault)
125                                                         else:
126
127      76.5 MiB      0.0 MiB          36                       fault_update(node)
128
129
130                                             #Actual solver main code for the first evaluation
131      76.5 MiB      0.0 MiB           2       def solveDAG(g, l1, l2, n1):
132
133      76.5 MiB      0.0 MiB           1           yyy=[]
134      76.5 MiB      0.0 MiB          17           while len(l2) !=0 :
135
136      76.5 MiB      0.0 MiB          16               l21 = l2[0]
137      76.5 MiB      0.0 MiB          80               for ele in l1:
138      76.5 MiB      0.0 MiB          64                   g.nodes[ele]['value'] = int(l21[l1.index(ele)]) #We get different values of A,B,C,D from l2 and change it in the graph g
139      76.5 MiB      0.0 MiB          16               DAG() #We change the values of other output nodes
140      76.5 MiB      0.0 MiB          16               u=l2.pop(0) #We move to the next set of input values
141      76.5 MiB      0.0 MiB          16               if g.nodes[node_fault]['value']==1^int(type_fault): #l3 will have all A,B,C,D values such that the value at the node is opposite to the stuck at given
142      76.5 MiB      0.0 MiB         108                   for node in n1:
143      76.5 MiB      0.0 MiB          96                       l3.append(u) #This list is created to store values for the next round of faulty evaluation
144                                                 global l4; #We save l4 as the set of values in l3 to ensure uniqueness
145
146      76.5 MiB      0.0 MiB           1           l4=list(set(l3))
147      76.5 MiB      0.0 MiB           1           l5=list(l4)
```

```
 148                                                 #The below code is to␣
↪make sure that the order of elements returned by this evaluation and the␣
↪faulty evaluation are in the same order as ;4
 149
 150     76.5 MiB      0.0 MiB          13           while len(l5) !=0 :
 151     76.5 MiB      0.0 MiB          12               s=''
 152     76.5 MiB      0.0 MiB          12               l23 = l5[0]
 153     76.5 MiB      0.0 MiB          60               for ele in l1:
 154     76.5 MiB      0.0 MiB          48                   g.
↪nodes[ele]['value'] = int(l23[l1.index(ele)])
 155     76.5 MiB      0.0 MiB          12               DAG()
 156     76.5 MiB      0.0 MiB          12               u=l5.pop(0)
 157
 158
 159     76.5 MiB      0.0 MiB          12               if g.
↪nodes[node_fault]['value']==1^int(type_fault):
 160     76.5 MiB      0.0 MiB         108                   for node in n1:
 161     76.5 MiB      0.0 MiB          96                       s+=str((g.
↪nodes[node]['value']))
 162
 163     76.5 MiB      0.0 MiB          12               yyy.append(s)
 164     76.5 MiB      0.0 MiB           1           return yyy
 165
 166
 167     76.5 MiB      0.0 MiB           1       pred=(list(g.
↪predecessors(node_fault))) #This saves the predecessors values
 168
 169
 170     76.5 MiB      0.0 MiB           1       l=[]
 171     76.5 MiB      0.0 MiB           9       for i in n1:
 172     76.5 MiB      0.0 MiB           8           if i not in pred:
 173     76.5 MiB      0.0 MiB           6               l.append(i)
 174                                             #Actual solver for primary␣
↪faults
 175     76.5 MiB      0.0 MiB           2       def␣
↪solve_with_faults_primary(f, l1, l2, n1,uu):
 176     76.5 MiB      0.0 MiB           1           yyy=[]
 177     76.5 MiB      0.0 MiB          13           while len(l2) !=0 :
 178     76.5 MiB      0.0 MiB          12               s=''
 179     76.5 MiB      0.0 MiB          12               l21 = l2[0]
 180     76.5 MiB      0.0 MiB          60               for ele in l1:
 181     76.5 MiB      0.0 MiB          48                   if ele!
↪=node_fault and ele not in pred:
 182     76.5 MiB      0.0 MiB          24                       f.
↪nodes[ele]['value'] = int(l21[l1.index(ele)])
 183     76.5 MiB      0.0 MiB          24                   elif␣
↪ele==node_fault and ele not in pred :
 184                                                             f.
↪nodes[ele]['value'] = int(type_fault)
 185     76.5 MiB      0.0 MiB          12               fault_DAG() #We use a␣
↪different solver with a different update function
 186
 187     76.5 MiB      0.0 MiB          12               rr=l2.pop(0)
 188
 189     76.5 MiB      0.0 MiB         108               for node in n1:
 190     76.5 MiB      0.0 MiB          96                   if node in l:
 191
 192     76.5 MiB      0.0 MiB          72                       s+=str(f.
↪nodes[node]['value'])
 193                                                         else:
 194     76.5 MiB      0.0 MiB          24                      ␣
↪s+=str(rr[uu[node]]) #Here when the node is C or D, we instruct the function␣
↪to take the original value itsself without taking the eval one
 195     76.5 MiB      0.0 MiB          12               yyy.append(s)
 196     76.5 MiB      0.0 MiB           1           return(yyy)
 197
 198                                             #Actual solver for secondary␣
↪or more faults
 199     76.5 MiB      0.0 MiB           1       def␣
↪solve_with_faults_secondary(f, l1, l2, n1,uu):
 200                                                 yyy=[]
 201                                                 yp=0
 202                                                 while len(l2) !=0 :
 203
 204                                                     s=''
 205                                                     l21 = l2[0]
 206
 207                                                     for ele in l1:
 208                                                         f.
↪nodes[ele]['value'] = int(l21[l1.index(ele)])
 209
 210                                                     fault_DAG()
 211                                                     rr=l2.pop(0)
 212
 213                                                     for node in n1:
 214                                                         if node not in uu:
 215
 216                                                             s+=str(f.
↪nodes[node]['value'])
 217                                                         else:
 218                                                            ␣
↪s+=var1[yp][uu[node]] #Here when the node is one of the predecessors, we␣
↪instruct the function to take the original value itsself without taking the␣
↪eval one
```

```
 219
 220                                                     yyy.append(s)
 221                                                     yp+=1
 222                                                 return(yyy)
 223
 224     76.5 MiB      0.0 MiB           1       begin = time.time()
 225     76.5 MiB      0.0 MiB           1       var1=solveDAG(g, l1, l2, n1)
 226                                             #print(var1)
 227     76.5 MiB      0.0 MiB           1       f.remove_nodes_from(pred)␣
↪#This removes the nodes of the predecessors as well
 228
 229     76.5 MiB      0.0 MiB           1       F=0 #Flag
 230     76.5 MiB      0.0 MiB           1       if node_fault in l1 or␣
↪pred[0] in l1: #if the fault node is at Primary level, or at one of the inputs␣
↪A,B,C,D
 231     76.5 MiB      0.0 MiB           1           uu={}
 232
 233     76.5 MiB      0.0 MiB           3           for i in pred:
 234     76.5 MiB      0.0 MiB           2               uu[i]=(l1.index(i))
 235     76.5 MiB      0.0 MiB           1           ␣
↪var2=solve_with_faults_primary(f, l1, l4, n1,uu)
 236
 237                                             elif node_fault not in l1 or␣
↪pred[0] not in l1:#if the fault node is at Secondary level
 238                                                 uu={}
 239
 240                                                 for i in pred:
 241                                                     uu[i]=(list(inputs).
↪index(i))
 242                                                 ␣
↪var2=solve_with_faults_secondary(f, l1, l4,n1,uu)
 243                                             #print(var2)
 244     76.5 MiB      0.0 MiB           1       fobj=open("Sample Outp.
↪txt",'a+') #Writing to the output file
 245
 246     76.5 MiB      0.0 MiB          13       for i in range(len(var1)):
 247     76.5 MiB      0.0 MiB          12           if var1[i][-1]!
↪=var2[i][-1]: #if the Z value differs, we include it as a valid test input␣
↪vector
 248     76.5 MiB      0.0 MiB          12               a_s=var2[i][:4]
 249     76.5 MiB      0.0 MiB          12               a_l=[]
 250     76.5 MiB      0.0 MiB          12               F=1
 251     76.5 MiB      0.0 MiB          60               for ii in a_s:
 252     76.5 MiB      0.0 MiB          48                   a_l.
↪append(int(ii))
 253                                                     #print("[A, B, C, D]␣
↪=",a_l,", Z = ",int(var2[i][-1]))
```

```
   254     76.5 MiB      0.0 MiB          12               fobj.write("[A, B, C, D] ="+str(a_l)+", Z = "+(var2[i][-1])+"\n")
   255                                             #print(n1)
   256     76.5 MiB      0.0 MiB           1       if F==0:
   257                                                 fobj.write("NO INPUT TEST VECTOR CAN HELP US IDENTIFY THIS STUCK-AT-FAULT")
   258
   259
   260     76.5 MiB      0.0 MiB           1       end = time.time()
   261                                             #fobj.close()
   262     76.5 MiB      0.0 MiB           1       print("Time taken is "+str(end-begin))
```

[ ]: