Midterm Assignment
Advanced Algorithmics (CPT_S 515)

Coral Jain
WSU ID: 11632780

3. For EFp, following pseudo code is given below, in the EFp, in any one path the condition p will be satisfied for only one node. E, here signifies 'there exists a path' and F signifies 'future' which will justify that there will a node in a path in future which will satisfy the condition p.

[EFp]

H_0 := [p]

k := 0

Repeat

k ++

H_k := the set of all states in any a path such that $u \epsilon H_{k-1}$ in a path OR for all v with R(u, v), $v \epsilon H_{k-1}$

Until $H_k = H_{k-1;}$

Return H_k as [EFp]

For EGp, following pseudo code is given below, in the EGp, in any one path the condition p will be satisfied for all the nodes in a path. E, here signifies 'there exists a path' and G signifies all the nodes of graph G which will justify the condition p.

[EGp]

H_0 := [p]

k := 0

Repeat

k ++

H_k := the set of all states $u \epsilon$ [p] in a path AND for all v with R(u, v), $v \epsilon H_{k-1}$.

Until $H_k = H_{k-1;}$

Return H_k as [EGp]

4. Program:

```
from pyeda.inter import *
%load_ext gvmagic
import itertools

#Defining BDD variables
x0 = bddvar('x',0)
x1 = bddvar('x',1)
x2 = bddvar('x',2)
x3 = bddvar('x',3)
x4 = bddvar('x',4)
x5 = bddvar('x',5)
x6 = bddvar('x',6)
x7 = bddvar('x',7)
x8 = bddvar('x',8)
x9 = bddvar('x',9)
x10 = bddvar('x',10)
x11 = bddvar('x',11)
x12 = bddvar('x',12)
x13 = bddvar('x',13)
x14 = bddvar('x',14)
x15 = bddvar('x',15)
x16 = bddvar('x',16)
x17 = bddvar('x',17)
x18 = bddvar('x',18)
x19 = bddvar('x',19)
x20 = bddvar('x',20)
x21 = bddvar('x',21)
x22 = bddvar('x',22)
x23 = bddvar('x',23)
x24 = bddvar('x',24)
x25 = bddvar('x',25)
x26 = bddvar('x',26)
x27 = bddvar('x',27)
x28 = bddvar('x',28)
x29 = bddvar('x',29)
x30 = bddvar('x',30)
x31 = bddvar('x',31)

#Defining array
arr = [x0,x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12,x13,x14,x15,x16,x17,x18,x
19,x20,x21,x22,x23,x24,x25,x26,x27,x28,x29,x30,x31]

#Printing array
print(arr)

#Logic for connection of nodes
iarr = []
jarr = []
for i in range(0,31):
    for j in range(3,31):
        if (i + 3)%32 == j%32 or (i + 8)%32 == j%32:
            print(arr[i], arr[j])
```

```
            jarr.append(arr[j])
            iarr.append(arr[i])

#Printing i nodes
print(iarr)


#Printing j nodes
print(jarr)


#Building function for BDD construction
f = iarr[0] & jarr[0]|iarr[1] & jarr[1]| iarr[2]& jarr [2]| iarr[3] & jarr
[3] | iarr[4] & jarr[4] | iarr[5] & jarr[5] | iarr[6] & jarr[6] | iarr[7]
& jarr[7]| iarr[8] & jarr[8] | iarr[9] & jarr[9] | iarr[10] & jarr[10] | i
arr[11] & jarr[11] | iarr[12] & jarr[12] |iarr[13] & jarr[13] |iarr[14] &
jarr[14] |iarr[15] & jarr[15] |iarr[16] & jarr[16] |iarr[17] & jarr[17] |i
arr[18] & jarr[18] |iarr[19] & jarr[19] |iarr[20] & jarr[20] |iarr[21] & j
arr[21] |iarr[22] & jarr[22] |iarr[23] & jarr[23] |iarr[24] & jarr[24] |ia
rr[25] & jarr[25] |iarr[26] & jarr[26] |iarr[27] & jarr[27] |iarr[28] & ja
rr[28] |iarr[29] & jarr[29] |iarr[30] & jarr[30] |iarr[31] & jarr[31]


#Plotting BDD diagram using GraphViz
%dotobj f


#Logic for even set of nodes
even = []
for i in arr:
    ind = arr.index(i)
    if ind%2 == 0:
        even.append(i)
print(even)


#Logic for prime set of nodes
prime = []

for i in arr:
    ind = arr.index(i)
    k = 0
    if ind >= 3:
        for z in range(2,ind//2+1):
            if (ind%z == 0):
                k = k + 1
        if k <= 0:
            prime.append(i)

print(prime)


#EFp CTL logic
def EFp_logic(condition):
    condition = prime
```

```
    H[k] = condition
    k = 0
    while H[k] != H[k-1]:
        k++
        H[k] = condition
    return H[k]


#EGp CTL logic
def EGp_logic(condition):
    condition = even and EFp_logic(prime)
    H[k] = condition
    k = 0
    while H[k] != H[k-1]:
        k++
        H[k] = condition
    return H[k]


#Smoothing of nodes
f = smoothing(x0,x2,x4,x6,x8,x10,x12,x14,x16,x18,x20,x22,x24,x26,x28,x30)


#Composing functions
for i in arr:
    rr = compose({x0:x1,x2:x3,x4:5,x6:x7,x8:x11,x10:x13,x12:x17,x14:x19,x1
6:23,x18:29,x20:31})
```

Output: The outputs as obtained are given below:

Printing Array:

```
[x[0], x[1], x[2], x[3], x[4], x[5], x[6], x[7], x[8], x[9], x[10], x[11],
x[12], x[13], x[14], x[15], x[16], x[17], x[18], x[19], x[20], x[21], x[22
], x[23], x[24], x[25], x[26], x[27], x[28], x[29], x[30], x[31]]
```

Logic for connection of nodes:

```
x[0] x[3]
x[0] x[8]
x[1] x[4]
x[1] x[9]
x[2] x[5]
x[2] x[10]
x[3] x[6]
x[3] x[11]
x[4] x[7]
x[4] x[12]
x[5] x[8]
x[5] x[13]
x[6] x[9]
x[6] x[14]
x[7] x[10]
x[7] x[15]
x[8] x[11]
x[8] x[16]
x[9] x[12]
x[9] x[17]
x[10] x[13]
x[10] x[18]
x[11] x[14]
x[11] x[19]
x[12] x[15]
x[12] x[20]
x[13] x[16]
x[13] x[21]
x[14] x[17]
x[14] x[22]
x[15] x[18]
x[15] x[23]
x[16] x[19]
x[16] x[24]
x[17] x[20]
x[17] x[25]
x[18] x[21]
x[18] x[26]
x[19] x[22]
x[19] x[27]
```

```
x[20] x[23]
x[20] x[28]
x[21] x[24]
x[21] x[29]
x[22] x[25]
x[22] x[30]
x[23] x[26]
x[24] x[27]
x[25] x[28]
x[26] x[29]
x[27] x[3]
x[27] x[30]
x[28] x[4]
x[28] x[31]
x[29] x[5]
x[30] x[6]
x[31] x[7]
```

Printing i nodes:

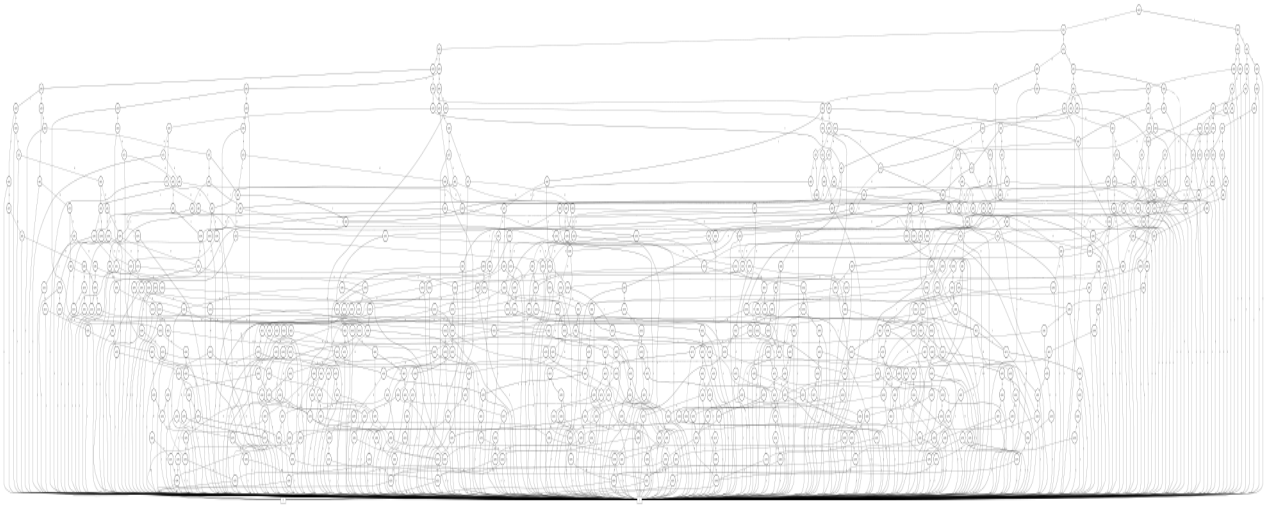
```
[x[0], x[0], x[1], x[1], x[2], x[2], x[3], x[3], x[4], x[4], x[5], x[5], x
[6], x[6], x[7], x[7], x[8], x[8], x[9], x[9], x[10], x[10], x[11], x[11],
x[12], x[12], x[13], x[13], x[14], x[14], x[15], x[15], x[16], x[16], x[17
], x[17], x[18], x[18], x[19], x[19], x[20], x[20], x[21], x[21], x[22], x
[22], x[23], x[24], x[25], x[26], x[27], x[27], x[28], x[29], x[30]]
```

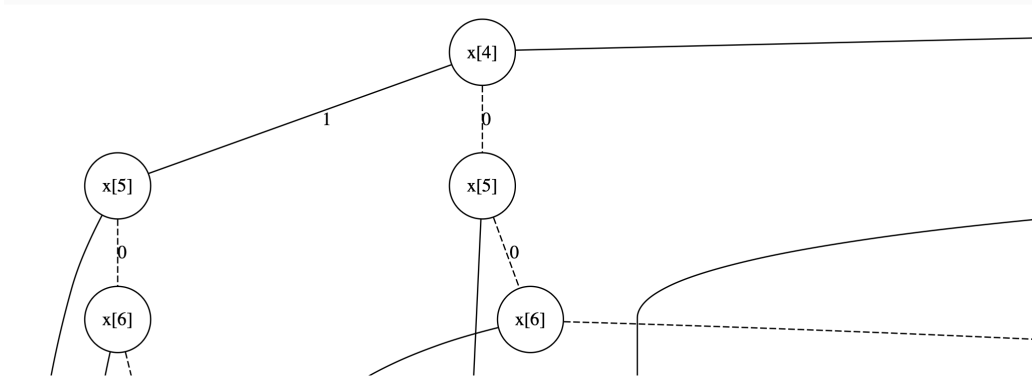
Printing j nodes:

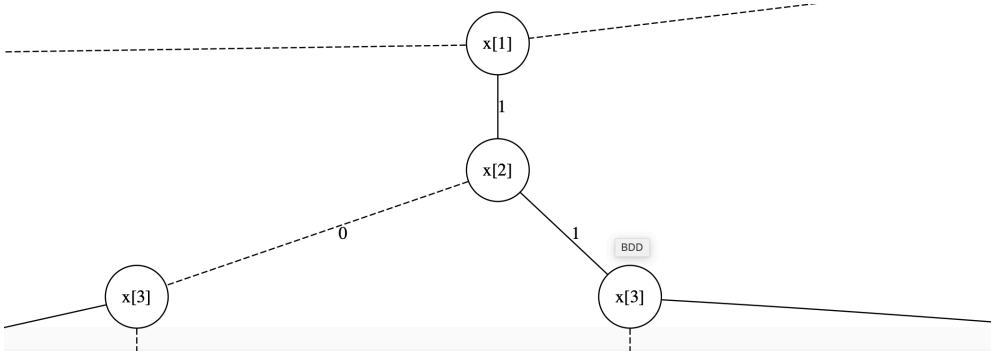
```
[x[3], x[8], x[4], x[9], x[5], x[10], x[6], x[11], x[7], x[12], x[8], x[13
], x[9], x[14], x[10], x[15], x[11], x[16], x[12], x[17], x[13], x[18], x[
14], x[19], x[15], x[20], x[16], x[21], x[17], x[22], x[18], x[23], x[19],
x[24], x[20], x[25], x[21], x[26], x[22], x[27], x[23], x[28], x[24], x[29
], x[25], x[30], x[26], x[27], x[28], x[29], x[3], x[30], x[4], x[5], x[6]
]
```

BDD construction:

Note: The plot of graph was very difficult to visual. I have posted the first picture which shows the complete diagram. The diagrams following it show few images to give an idea as bits and pieces about how the diagram is plotted.







Logic for even set of nodes:

```
[x[0], x[2], x[4], x[6], x[8], x[10], x[12], x[14], x[16], x[18], x[20], x
[22], x[24], x[26], x[28], x[30]]
```

Logic for prime set of nodes:

```
[x[3], x[5], x[7], x[11], x[13], x[17], x[19], x[23], x[29], x[31]]
```

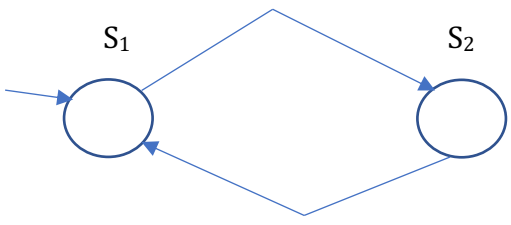
5. <u>Mini paper on Finding a metric for comparison of two DFA modes for a given set.</u>

For this we are given a finite set L of DNA strands with the help of Deterministic Finite Automation which will accept or reject a string of data given to it as input. It is a 5-tuple set defined by elements: $(Q, \Sigma, \delta, q_0, F)$. In our case, we are given that a finite set L of DNA strands where every strand is made of A, T, C, G symbols. When DFA is applied to this, one by one sequentially A, T, C, G is collected i.e. one symbol at a time all the symbols of the set {A, T, C, G} will be collected which will form a strand w.

It is already given to us that there can be a number of ways by which M can be defined. One way amongst which is that only one state can be used to symbols can be collection in a single state. Second way could be that M can be modelled in a way in which we can have more than one state collecting the given symbols. There can be a number of ways in which in which we can determine the kind of models M that can be used to perform the experiment that is required for us to perform.

One of the ways to compare the two models is to convert both the models M_1 and M_2 into their regular expressions. By converting the models into regular expressions and thus determining how efficient a model behaves in the environment, we can derive a precision Q (M, L), which will tell us which model is best fit for our use. Arden's theorem is a good example for this. This theorem can be used to convert any RA into regular expression.

However, as much as this choice appear to be correct, we must realize that a finite automation can be converted into multiple regular expressions. This can be shown by the example below:



In the FSM diagram shown above, the regular expression for it can be given in a number of ways:

i. $(S_1S_2)^*$

ii. $(S_1S_2)^* (S_1S_2)^*$

iii. $(S_1S_2)^* (S_1S_2)^* (S_1S_2)^*$

In the above ways, there can be a number of ways by which a regular expression can be written. Hence, for different regular expressions, there can be different values of metrics. Hence, this method does not give us results as expected.

Another way to check the precision of the given two models i.e. Q (M, L) is to take the idea of equivalence into the consideration. For example, if we are given two models M_1 and M_2 and we need to determine their equivalence, we must first check the two things:

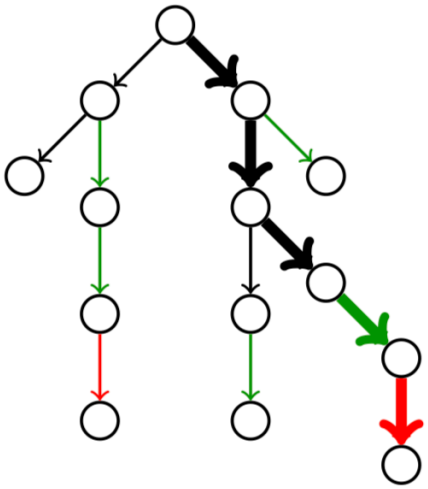
- If we start traversing from start state of M_1 and reach via a path to accepting state of M_1 labelled M_{11}, M_{12}, …. M_{1k} then there is a path from start state of M_2 and reach via a path to accepting state of M_2 labelled M_{21}, M_{22}, …. M_{2k}.
- Similarly, if we start traversing from start state of M_1 and reach via a path to accepting state of M_2 labelled M_{21}, M_{22}, …. M_{2j} then there is a path from start state of M_2 and reach via a path to accepting state of M_2 labelled M_{11}, M_{12}, …. M_{1j}.

There are a number of algorithms that can carry out equivalence testing. One such method is Symmetric Difference in which it is determined for two modes (say M_1 and M_2 in our case) which will tell us whether there are some symbols that are accepted by some state and not by others. This justifies the amount of precision Q in our case. This is because can be a number of states of our machine which cannot give us the required result by not providing us all the symbols or nucleotides.

Another such efficient algorithm is Table-filling algorithm. In this algorithm two models are treated as a single model and kept on top of each other and to check which states show different behavior. The distinguished states will determine the precision of the two models.

However, it is to be noted that measuring the equivalence will not justify our problem since we require a metric Q which will determine whether two models taken into consideration can be compared based on it.

One such solution is to use Log-based metric, a solution given by Petra van den Bos in his journal/text called, “Enhancing Automata Learning by Log-based Metrics”. In the text, the writers have used a logarithmic based metric which is deployed in a machine to check whether there is an error in the output of the machine. With the help of the percentage error the precision Q (M, L) in the form of a real value can be obtained. This is as shown below:



In the above figure as shown below, there are a number of states for an M. Each state is capable of collecting all the symbols that make up a strand. Here in this we also define a weight function which can solve our problem

This function is given by $w(log, \rho) = 2^{-|\rho|}$.

In the above figure, we see that black line is the log input, green in input with same output and red with input with different output. With the help of weight function, the metric Q can be defined.

References:

I. Algorithms for testing equivalence of finite automata, with a grading tool for JFLAP, by Daphne Norton, Rochester Institute of Technology, Thesis, 2009.
II. Enhancing Automata Learning by Log -Based Metrics by Peter van den Bos, Rick Smetsers, Frits Vaandrager, Radboud University Nijmegen.