Bayesian Probabilistic Graphical Models

April 6, 2019

1 Submission by Varun Nagaraj.

```
In [68]: print("Reference: https://ublearns.buffalo.edu/bbcswebdav/pid-4944430-dt-content-rid-2
         print("UBIT number: 50290761")
         print("UB name: varunnag")
```

```
Reference: https://ublearns.buffalo.edu/bbcswebdav/pid-4944430-dt-content-rid-22724271_1/course
UBIT number: 50290761
UB name: varunnag
```

2 Imports

```
In [2]: import bif_parser
        import prettytable as pt
        import pydotplus
        from IPython.core.display import display, Image
        from bayesian.bbn import *
```

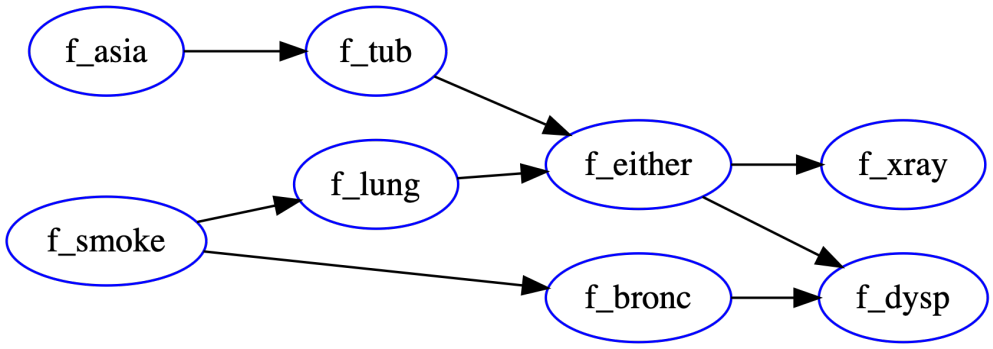
3 Create the model using the asia bif file in the path.

```
In [5]: module_name = bif_parser.parse('asia')
        module = __import__(module_name)
        bg = module.create_bbn()
```

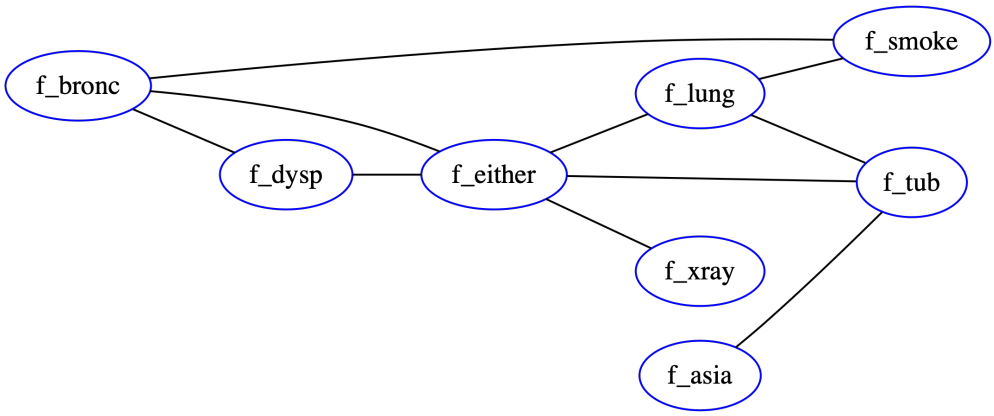
4 Input image and plot using graphviz

```
In [11]: s0 = bg.get_graphviz_source()
         inp = pydotplus.graph_from_dot_data(s0)
         inp.write_png('inp.png')
```

```
Out[11]: True
```



Input Tree



Moralized Tree

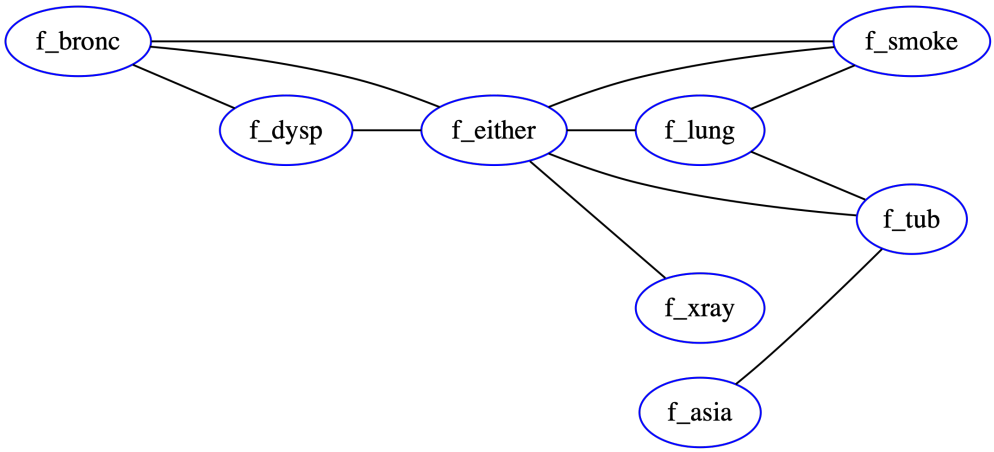
5 Create a moralized graph by taking the undirected copy of the graph and creating a path between parents of a node if there are 2 or more.

```
In [13]: gu = make_undirected_copy(bg)
         m1 = make_moralized_copy(gu, bg)
         s1 = m1.get_graphviz_source()
         graph1 = pydotplus.graph_from_dot_data(s1)
         graph1.write_png('temp1.png')
```

```
Out[13]: True
```

6 After moralizing the graph, now we can triangulate the graph by adding an edge to adjacent vertices in cycles of length greater than or equal to 4.

```
In [14]: cliques, elimination_ordering = triangulate(m1, priority_func)
         s2 = m1.get_graphviz_source()
         graph2 = pydotplus.graph_from_dot_data(s2)
         graph2.write_png('temp2.png')
```



Triangulated Tree

```
Out[14]: True
```

7 We can now build the junction tree, which creates cliques according to the preceding graph and creates the sepsets which are the intersection points between every pair of cliques.

```
In [15]: jt = bg.build_join_tree()
         s3 = jt.get_graphviz_source()
         graph2 = pydotplus.graph_from_dot_data(s3)
         graph2.write_png('temp3.png')
```

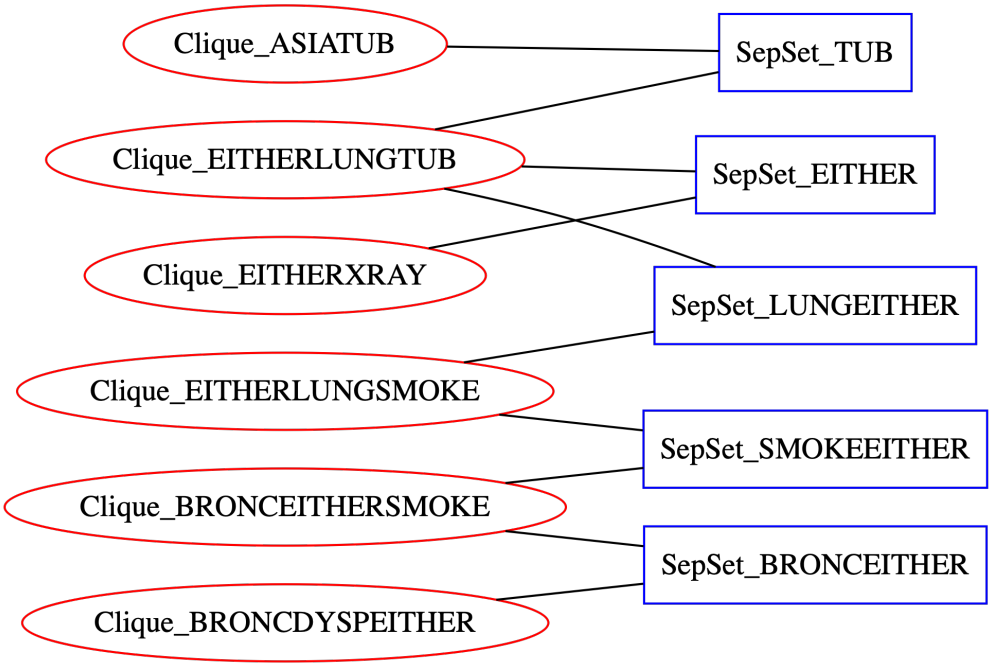
```
Out[15]: True
```

8 Running Intersection Property

This property states that if a node appears in two cliques, it should also appear in all the nodes on the path that connect the two cliques. In the above junction tree this property can be validated. We can see that if there is a node appearing in 2 or more clusters say bronc, then there will be a node on the path connecting the two clusters i.e "bronceither". This proves the running intersection property.

9 Initialize the potentials for the cliques.

```
In [67]: assignments = jt.assign_clusters(bg)
         jt.initialize_potentials(assignments, bg)
```



Junction Tree

10 There are two messages sent between each pair of cliques: one in a forward and another in a reverse pass. There are three actors in a message pass: the source cluster, the intervening sepset, and the destination cluster.

```
In [24]: jt.propagate()

In [27]: for i in jt.clique_nodes:
             print i.variable_names

['asia', 'tub']
['bronc', 'dysp', 'either']
['bronc', 'either', 'smoke']
['either', 'lung', 'tub']
['either', 'lung', 'smoke']
['either', 'xray']
```

11 Once all the message passing is done, we are left with a tree that has consistent beliefs in all its clusters. On querying for a particular variable (for example, bronc), we just have to find a cluster (or a sepset) that has bronc in its scope and marginalize the other variables.

```
In [31]: bronc_clust = [i for i in jt.clique_nodes for v in i.variable_names if v =='bronc']
         pot = bronc_clust[0].potential_tt
         pot2 = bronc_clust[1].potential_tt
```

12 Marginal for Bronc found by marginalizing Dysp and Either potentials

```
In [29]: pot
```

```
Out[29]: {(('bronc', 'no'), ('dysp', 'no'), ('either', 'no')): 0.46892195999999986,
          (('bronc', 'no'), ('dysp', 'no'), ('either', 'yes')): 0.008692680000000001,
          (('bronc', 'no'), ('dysp', 'yes'), ('either', 'no')): 0.052102439999999986,
          (('bronc', 'no'), ('dysp', 'yes'), ('either', 'yes')): 0.020282920000000003,
          (('bronc', 'yes'), ('dysp', 'no'), ('either', 'no')): 0.08282951999999998,
          (('bronc', 'yes'), ('dysp', 'no'), ('either', 'yes')): 0.0035852400000000017,
          (('bronc', 'yes'), ('dysp', 'yes'), ('either', 'no')): 0.3313180799999999,
          (('bronc', 'yes'), ('dysp', 'yes'), ('either', 'yes')): 0.03226716000000002}
```

13 Marginal for Bronc found by marginalizing Smoke and Either potentials

```
In [32]: pot2
```

```
Out[32]: {(('bronc', 'no'), ('either', 'no'), ('smoke', 'no')): 0.3428963999999999,
          (('bronc', 'no'), ('either', 'no'), ('smoke', 'yes')): 0.17812799999999998,
          (('bronc', 'no'), ('either', 'yes'), ('smoke', 'no')): 0.0071036,
          (('bronc', 'no'), ('either', 'yes'), ('smoke', 'yes')): 0.021872000000000006,
          (('bronc', 'yes'), ('either', 'no'), ('smoke', 'no')): 0.14695559999999996,
          (('bronc', 'yes'), ('either', 'no'), ('smoke', 'yes')): 0.26719199999999993,
          (('bronc', 'yes'), ('either', 'yes'), ('smoke', 'no')): 0.003044400000000001,
          (('bronc', 'yes'), ('either', 'yes'), ('smoke', 'yes')): 0.03280800000000002}
```

14 We claim that the cluster will have the same marginals. To prove this consider the cluster (bronc, dysp, either) and (bronc, smoke, either). When we check the values of bronc we can see that the marginals of bronc are the same across both the clusters. This indicated that the message passing has worked correctly and the belief for all the variables is consistent across all the clusters in the graph.

```
In [62]: sum_assignments =lambda imap, tup: sum([v for k, v in imap.iteritems() for i in k if i
         yes, no =[sum_assignments(pot, ('bronc', i)) for i in ['yes', 'no']]
         print 'bronc: True ', yes/float(yes+no), " False ", no/float(yes+no)
         yes, no =[sum_assignments(pot2, ('bronc', i)) for i in ['yes', 'no']]
         print 'bronc: True ', yes/float(yes+no), " False ", no/float(yes+no)

         xray_clust = [i for i in jt.clique_nodes for v in i.variable_names if v =='xray']
         xray_pot = xray_clust[0].potential_tt
         yes, no =[sum_assignments(xray_pot, ('xray', i)) for i in ['yes', 'no']]
         print '\nxray: True ', yes/float(yes+no), " False ", no/float(yes+no)

         lung_clust = [i for i in jt.clique_nodes for v in i.variable_names if v =='lung']
         lung_pot =lung_clust[0].potential_tt
         yes, no =[sum_assignments(lung_pot, ('lung', i)) for i in ['yes', 'no']]
         print '\nlung: True ', yes/float(yes+no), " False ", no/float(yes+no)

         tub_clust = [i for i in jt.clique_nodes for v in i.variable_names if v =='tub']
         tub_pot =tub_clust[0].potential_tt
         yes, no =[sum_assignments(tub_pot, ('tub', i)) for i in ['yes', 'no']]
         print '\ntub: True ', yes/float(yes+no), " False ", no/float(yes+no)
```

```
bronc: True  0.45  False  0.55
bronc: True  0.45  False  0.55

xray: True  0.11029004  False  0.88970996

lung: True  0.055  False  0.945

tub: True  0.0104  False  0.9896
```

15 The above steps can be repeated to the other nodes in the network. This will again prove the belief propogation is consistent across all the nodes in the network.

```
In [56]: lung_pot
```

```
Out[56]: {(('either', 'no'), ('lung', 'no'), ('tub', 'no')): 0.9351719999999998,
          (('either', 'no'), ('lung', 'no'), ('tub', 'yes')): 0,
```

```
          (('either', 'no'), ('lung', 'yes'), ('tub', 'no')): 0,
          (('either', 'no'), ('lung', 'yes'), ('tub', 'yes')): 0,
          (('either', 'yes'), ('lung', 'no'), ('tub', 'no')): 0,
          (('either', 'yes'), ('lung', 'no'), ('tub', 'yes')): 0.009828000000000003,
          (('either', 'yes'), ('lung', 'yes'), ('tub', 'no')): 0.05442800000000001,
          (('either', 'yes'), ('lung', 'yes'), ('tub', 'yes')): 0.0005720000000000001}

In [57]: xray_pot

Out[57]: {(('either', 'no'), ('xray', 'no')): 0.8884133999999998,
          (('either', 'no'), ('xray', 'yes')): 0.0467586,
          (('either', 'yes'), ('xray', 'no')): 0.0012965600000000002,
          (('either', 'yes'), ('xray', 'yes')): 0.06353144000000001}

In [58]: tub_pot

Out[58]: {(('asia', 'no'), ('tub', 'no')): 0.9800999999999999,
          (('asia', 'no'), ('tub', 'yes')): 0.009900000000000003,
          (('asia', 'yes'), ('tub', 'no')): 0.009499999999999998,
          (('asia', 'yes'), ('tub', 'yes')): 0.0005000000000000001}
```