Probabilistic Graphical Models CS2950P HW1

Sam Boosalis (with Ryan Lester)

March 2013

1 Iterative Sum Product on Any Factor Graph

1.a

(code in sumproduct.py and factorgraph.py and main.py)

1.b

(code in sumproduct.py)

1.c

(test by running ./sumproduct.py).

i first test on small graphs by explicitly passing messages. i then test both sum-product and brute-force on small factor trees by checking that the brute-force marginals equal manual solutions and by checking the algorithms against each other (equality is transitive). i test that the product of marginals of brute-force equal the product of right factors. i use variables of different dimensions, factors of different orders, and factors with different probabilities. i use a chain; a tree; a deeper binary tree; several variables and one factor; several factors and one variable; and a square with cycles. all the acyclic factor graphs converge within 10 iterations, and they all are exact (including the small cyclic graphs). the experimental evidence is to run *sumproduct.py* and see the tests pass, which do the above.

The Sum-Product Algorithm

```
#!/usr/bin/python
from __future__ import division

from numpy import *
from matplotlib.pyplot import *
import numpy.random as sample
import scipy.stats as pdf
import networkx as nx
import itertools
from copy import deepcopy

from factorgraph import *
from sam.sam import *

"""
[1]

(A)
Implement the sum-product algorithm. Your code should support an arbitrary factor graph
linking a collection of discrete random variables. Use a parallel message update schedule,
in which all factor-to-variable messages are updated given the current variable-to-factor
messages, and then all variable-to-factor messages given the current factor-to-variable
messages. Initialize by setting the variable-to-factor messages to equal 1 for all states.
Be careful to normalize messages to avoid numerical under flow.

(B) Write code which explicitly computes a table containing the probabilities of all joint confi
urations of the variables in a factor graph. Also write code which sums these probabilities
to compute the marginal distribution of each variable. Such "brute force" inference code
is of course inefficient, and will only be computationally tractable for small models.

(C) Create a small, tree-structured factor graph linking four variable nodes. Use this model to
verify that the algorithms implemented in parts (a) and (b) are consistent with each other,
and compute correct marginal distributions. Design your factor graph to validate many
aspects of your code, including variables with different numbers of states, and factors of
varying orders. Clearly describe the experimental evidence you use to verify that your
implementations are correct


"""

cartesian = itertools.product

#####################################
# A

def make_messengers(G):

    #: factor , variable => [num]
    # fac to var
    Mu = {}
    for f in G.facs():
        for v in G.N(f):
            Mu[f,v] = array([1 for _ in G.vals(v)])
```

```
    #: variable , factor => [num]
    # var to fac
    Nu = {}
    for f in G.facs():
        for v in G.N(f):
            Nu[v,f] = array([1 for _ in G.vals(v)])

    normalize_messenger(Mu)
    normalize_messenger(Nu)

    return Mu, Nu


def normalize_messenger(X):
    for x in X: X[x] = pd(X[x])



def fac2var(_Mu,Nu, G, f,v):
    """
    eg
    VAR has { val ... }
    N(F) - X = { Y Z }
    forall x in X,  mu[F,X] x = sum[y,z] f x y z * nu[Y, F] y * nu[Z, F] z

    prob. must preserve order of vars (and their vals) for factor and for consistency
    soln. { var => [vals] }

    ndarray.flatten()
    ==
    ndarray.resize(ndarray.size)
    ==
    ndarray.shape = (ndarray.size,)

    """
    #print
    #print "fac '%s' \t=>\t var '%s'" % (f,v)
    assert G.type(f)=='fac' and G.type(v)=='var'

    vars = G.N(f) # for order
    ii = { x:i for i,x in enumerate(vars) } # inverted index

    for val in G.vals(v): # forall val in var

        # "pin down msg var to one val"
        # eg
        # var = 'b'
        # val = 2
        # vars = ['a','b','c']
        # space = {0..1} x {2} x {0..3}
        space = cartesian( *[(G.vals(_v) if _v != v else [val])  for _v in vars] )

        # get _val of _var
        #  _vals[ii[_v]] = _v:str => ii:str=>inx => _vals:inx=>val => Nu[_,_]:val=>num
        #  discrete randvar -> values are indices
        # sum of prod
```

```
        msg = sum(  G(f, *_vals)  *  product([ Nu[_v, f][_vals[ii[_v]]]  for _v in G.N(f)
if _v != v  ])
                    for _vals in space )

        _Mu[f,v][val] = msg

    """

    # sum (fac * prod nus)
    fac = G.node[f]['pmf']
    nus = [ (i, _v, Nu[_v, f])  for i,_v in enumerate(G.N(f))  if _v != v  ]

    msg = fac
    for i,_v,nu in nus:
        # sans broadcast
        shape = [1 for _ in msg.shape]
        shape[i] = G.node[_v]['d']
        nu = resize(nu, tuple(shape))
        nu = resize(nu, msg.shape)
        msg = msg * nu
        # [diff] msg = msg * resize(nu, msg.shape)
        # [diff] msg = resize(nu, msg.shape) * msg

    others = tuple([ i  for i,_v in enumerate(G.N(f))  if _v != v ])
    msg = sum(msg, axis=others) # marginalize every other var
    Mu[f,v] = msg

    """

    #print
    #print 'Mu =', Mu


def var2fac(Mu,_Nu, G, v,f):
    #print
    #print "var '%s' \t=>\t fac '%s'" % (v,f)
    assert G.type(v)=='var' and G.type(f)=='fac'

    """
    for val in G.vals(v):
        msg = product([ Mu[_f, v][val]  for _f in G.N(v)  if _f != f ])
        Nu[v,f][val] = msg
    """
    msg = [ product([ Mu[_f, v][val]  for _f in G.N(v)  if _f != f ])
            for val in G.vals(v) ]

    _Nu[v,f] = msg

    #print
    #print 'Nu =', Nu


def msg(M,N,G, x,y):
    if G.type(x)=='var':
        var2fac(M,N, G, x,y)
    else:
```

```
        fac2var(M,N, G, x,y)



def marginal(Mu, G, v):
    # forall f in G.N(v),   p = Nu[v,f] * Mu[f,v]
    return pd([ product([ Mu[f,v][val] for f in G.N(v) ])  for val in G.vals(v) ])

def marginals(Mu,G, vars):
    return { v : marginal(Mu,G, v) for v in vars }



def marginalize_sumprod(G,
                        M=1, N=500, P=2, eps=1e-6,
                        vars=None, verbose=True):
    """
    : any factor graph => marginals
    : iterative algorithm
     exact and fast on factor trees
     approximate and slow on cyclic factor graphs

    factor graph
    randvars : discrete
    message update schedule : parallel

    [message passing protocol]
    distribute message to some neighbor node
    <->
    have collected message from every other neighbor node

    [message passing protocol]
    : parallel
    each iteration, collect from i-1 => distribute to i

    [iteration]
    set next factor-to-variable msgs  from curr variable-to-factor msgs
    =>
    update variable-to-factor msgs  given factor-to-variable msgs
    =>
    normalize msgs -> avoid underflow numeric

    factor graph ~ undirected graph
    factor graph : tree  <->  factor graph as undirected graph : tree
    -> sumprod on factgraphs iterates approximately (not recurs exactly)


    xs = x        ->  marginalize for this var    => [m]
    xs = [x,y..]  ->  marginalize for these vars  => [m..]
    xs = None     ->  marginalize for each var    => [m..]

    """

    if not vars: vars = G.vars()

    # "_X" set/write to next/new
```

```
    # "X" get/read from curr/old
    _Mu, _Nu = make_messengers(G)
    Mu, Nu = deepcopy(_Mu), deepcopy(_Nu)

    i = 0
    _diff, diff = +inf, 0
    stuck = 0
    while True:
        if not i < N:
            alert('[sumprod: iterated too many times (N=%d) with (diff=%.9f)]' % (N,diff))
            break

        if M < i:
            if abs(_diff - diff) < eps and stuck > 1: #HACK diff hits zero every other dunno wh
                print '[sumprod: converged (eps=%.0e) in %d iterations]' % (eps, i)
                break

            if stuck > P:
                print '[sumprod: got stuck %d times at diff=%.9f]' % (P, diff)
                break

        i += 1
        if verbose: print; print i

        # parallel update schedule

        # factor-to-variable
        for f in G.facs():
            for v in G.N(f):
                msg(_Mu,Nu,G, f,v)
        normalize_messenger(_Mu)

        # variable-to-factor
        for v in G.vars():
            for f in G.N(v):
                msg(_Mu,_Nu,G, v,f)
        normalize_messenger(_Nu)

        # var('Mu',Mu)
        # var('_Mu',_Mu)
        _diff = max( max( max(abs(_Mu[fv] - Mu[fv])) for fv in Mu ),
                     max( max(abs(_Nu[vf] - Nu[vf])) for vf in Nu ))

        if verbose: var('diff', '%.12f' % abs(_diff - diff))

        stuck = 1+stuck if abs(_diff - diff) < eps else 0

        diff   = _diff
        Mu, Nu = deepcopy(_Mu), deepcopy(_Nu)

    return marginals(Mu,G, vars=vars)



################################################################
# B
```

```
def joint(G, xs=None):
    """

    CASE
    same vars in diff factors
    (in particular, two factors on same vars should just be multiplied and renormalized)
    eg joint p(x,y) q(y,z) => pq(x,y,z) not pq(x,y,y,z)


    """
    vars = G.vars() #: [var]
    facs = { f : G.N(f) for f in G.facs() } #: fac => vars

    dims = [G.node[x]['d'] for x in vars] #: [nat]
    _joint = ones(dims)

    for vals in itertools.product( *(xrange(d) for d in dims) ): # cartesian product
        _vars = dict(zip(vars,vals)) #: var => val
        vals = tuple(vals) # to index
        #print
        #print _vars
        for fac in facs:
            _vals = [_vars[v] for v in facs[fac]] # keep only fac's vars' vals
            #print '%s%s' % (fac, tuple(_vals))
            _joint[vals] *= G(fac, *_vals)

    Z = sum(_joint)

    return pd(_joint), Z


def marginalize_bruteforce(G, vars=None):
    if not vars: vars = G.vars()

    p,_ = joint(G)
    def but(i): return tuple([j for j in range(len(G.vars())) if j!=i])

    marginals = { v : p.sum(axis=but(i))
                  for i,v in enumerate(G.vars())
                  if v in vars }

    return marginals

####################################
# C


def test(G, **kwargs):

    var('G', G.node)

    bf = marginalize_bruteforce(G)
    sp = marginalize_sumprod(G, **kwargs)

    var('bf', bf)
```

```
    var('sp', sp)

    compare(sp,bf)


def compare(ps,qs, fail=True):
    print

    if fail:
        for var,p,q in zip( ps, ps.values(), qs.values() ):
            assert all([near(pi,qi) for pi,qi in zip(p,q)])
        return True

    else:
        return all([ all([near(pi,qi) for pi,qi in zip(p,q)])  for var,p,q in zip(ps, ps.values


if __name__=='__main__':

    div('testing bruteforce marginalize')

    print
    print
    T = factor_tree()
    pT,Z = joint(T)
    assert pT.shape == (1,2,3,4)
    assert near( pT[0,0,0,0] ,
                T('f[ac]',0,0) * T('f[bc]',0,0) * T('f[cd]',0,0) *
                T('f[a]',0) * T('f[b]',0) * T('f[d]',0) *
                (1/Z)
                )
    assert near( pT[0,1,2,3] ,
                T('f[ac]',0,2) * T('f[bc]',1,2) * T('f[cd]',2,3) *
                T('f[a]',0) * T('f[b]',1) * T('f[d]',3) *
                (1/Z)
                )

    print
    print
    L = factor_list()
    pL,Z = joint(L)
    assert pL.shape == (1,2,3,4)
    assert near( pL[0,0,0,0] ,
                L('f[ab]',0,0) * L('f[bc]',0,0) * L('f[cd]',0,0) *
                (1/Z)
                )
    assert near( pL[0,1,2,3] ,
                L('f[ab]',0,1) * L('f[bc]',1,2) * L('f[cd]',2,3) *
                (1/Z)
                )

    print
    print
    C = factor_clique()
    pC,Z = joint(C)
    assert pC.shape == (1,2,3,4)
```

```
assert near( pC[0,0,0,0] , C('f[abcd]', 0,0,0,0) * (1/Z))
assert near( pC[0,1,2,3] , C('f[abcd]', 0,1,2,3) * (1/Z))

div('testing sumprod marginalize')



print
print
print 'testing 3 facs 1 var...'
G = factor_3f1v()
Mu,Nu = make_messengers(G)
m = implicit(Mu,Nu,G)(msg)

m('f1','v')
m('f2','v')
m('f3','v')

m('v','f1')
m('v','f2')
m('v','f3')

sp = {v:marginal(Mu,G, v) for v in G.vars()}
var('sp', sp)

bf = marginalize_bruteforce(G)
var('bf', bf)

compare(sp,bf)



print
print
print 'testing 1 fac 3 var...'
G = factor_1f3v()
Mu,Nu = make_messengers(G)
m = implicit(Mu,Nu,G)(msg)

m('a','f')
m('b','f')
m('c','f')

m('f','a')
m('f','b')
m('f','c')

sp = {v:marginal(Mu,G, v) for v in G.vars()}
var('sp', sp)

bf = marginalize_bruteforce(G)
var('bf', bf)

compare(sp,bf)
```

```
print
print
print 'testing small list...'
G = factor_small()
Mu,Nu = make_messengers(G)
m = implicit(Mu,Nu,G)(msg)

m('f[a]', 'a')
m('a','f[ab]')
m('f[ab]', 'b')

m('b', 'f[ab]')
m('f[ab]','a')
m('a', 'f[a]')

sp = {v:marginal(Mu,G, v) for v in G.vars()}
var('sp', sp)

bf = marginalize_bruteforce(G)
var('bf' , bf)

compare(sp,bf)



print
print
print 'testing list...'
G = factor_list()
Mu,Nu = make_messengers(G)
m = implicit(Mu,Nu,G)(msg)

m('a','f[ab]')
m('f[ab]', 'b')
m('b','f[bc]')
m('f[bc]', 'c')
m('c','f[cd]')
m('f[cd]','d')

m('d','f[cd]')
m('f[cd]','c')
m('c', 'f[bc]')
m('f[bc]','b')
m('b', 'f[ab]')
m('f[ab]','a')

sp = {v:marginal(Mu,G, v) for v in G.vars()}
var('sp', sp)

bf = marginalize_bruteforce(G)
var('bf', bf)

compare(sp,bf)
```

```
 print
 print
 print 'testing tree...'
G = factor_tree()
Mu,Nu = make_messengers(G)
m = implicit(Mu,Nu,G)(msg)
#nx.draw(G); show()

# start at leaves of tree
m('f[a]','a')
m('f[b]','b')
m('f[d]','d')

m('a','f[ac]')
m('b','f[bc]')
m('d','f[cd]')

# goto root of tree
m('f[ac]','c')
m('f[bc]','c')
m('f[cd]','c')

m('c','f[ac]')
m('c','f[bc]')
m('c','f[cd]')

m('f[ac]','a')
m('f[bc]','b')
m('f[cd]','d')

m('a','f[a]')
m('b','f[b]')
m('d','f[d]')

 sp = {v:marginal(Mu,G, v) for v in G.vars()}
 var('sp', sp)

 bf = marginalize_bruteforce(G)
 var('bf', bf)

 compare(sp,bf)



 div('testing general iterative sumprod')

 print;print;print 'testing...'
G = factor_1f3v()
 test(G)

 print;print; alert('testing...')
G = factor_3f1v()
 test(G)

 print;print; alert('testing list...')
```

```
G = factor_list()
test(G)

print;print; alert('testing tree...')
G = factor_tree()
test(G)

print;print; alert('testing square graph with a cycle...', t=0)
G = factor_square()
test(G)

print;print; alert('testing binary tree, whether it converges in |depth| iters...', t=0)
G = factor_btree()
test(G)

print
print
print 'all tests passed!'
```

Port the Matlab to Python

```
#!/usr/bin/python
from __future__ import division
from sam.sam import *
from sam import sam

from numpy import *
import numpy as np
from matplotlib.pyplot import *
import nltk
import numpy.random as sample
import scipy.stats as pdf

from factorgraph import *

zeros = sam.splat(zeros)

# # # # # # # # # # # # # # # # # # # # # # # # # #
# INIT

G = FactorGraph()

def add_var(G, *args, **kwargs):
    G.add_var(*args, **kwargs)

def add_fac(G, *args, **kwargs):
    G.add_fac(*args, **kwargs)

MINVOLSET = 'MINVOLSET'
add_var(G, MINVOLSET, 3)
p = [ 0.05, 0.9, 0.05 ]
add_fac(G, p, [MINVOLSET])

VENTMACH = 'VENTMACH'
add_var(G, VENTMACH, 4)
p = zeros(4,3)
p[:,0] = [ 0.05, 0.93, 0.01, 0.01 ]
p[:,1] = [ 0.05, 0.01, 0.93, 0.01 ]
p[:,2] = [ 0.05, 0.01, 0.01, 0.93 ]
add_fac(G, p, [VENTMACH, MINVOLSET])

DISCONNECT = 'DISCONNECT'
add_var(G, DISCONNECT, 2)
p = [ 0.1, 0.9 ]
add_fac(G, p, [DISCONNECT])

# VENTUBE | VENTMACH, DISCONNECT
VENTTUBE = 'VENTTUBE'
add_var(G, VENTTUBE, 4)
p = zeros(4,4,2)
p[:,0,0] = [ 0.97, 0.01, 0.01, 0.01 ]
p[:,0,1] = [ 0.97, 0.01, 0.01, 0.01 ]
p[:,1,0] = [ 0.97, 0.01, 0.01, 0.01 ]
p[:,1,1] = [ 0.97, 0.01, 0.01, 0.01 ]
p[:,2,0] = [ 0.97, 0.01, 0.01, 0.01 ]
```

```
p[:,2,1] = [ 0.01, 0.97, 0.01, 0.01 ]
p[:,3,0] = [ 0.01, 0.01, 0.97, 0.01 ]
p[:,3,1] = [ 0.01, 0.01, 0.01, 0.97 ]
add_fac(G, p, [VENTTUBE, VENTMACH, DISCONNECT])

PULMEMBOLUS = 'PULMEMBOLUS'
add_var(G, PULMEMBOLUS,2)
p = [ 0.01, 0.99 ]
add_fac(G, p, [PULMEMBOLUS])

INTUBATION = 'INTUBATION'
add_var(G, INTUBATION,3)
p = [ 0.92, 0.03, 0.05 ]
add_fac(G, p, [INTUBATION])

# PAP | PULMEMBOLUS
PAP = 'PAP'
add_var(G, PAP,3)
p = zeros(3,2)
p[:,0] = [ 0.01, 0.19, 0.8 ]
p[:,1] = [ 0.05, 0.9, 0.05 ]
add_fac(G, p, [PAP, PULMEMBOLUS])

# SHUNT | PULMEMBOLUS, INTUBATION
SHUNT = 'SHUNT'
add_var(G, SHUNT,2)
p = zeros(2,2,3)
p[:,0,0] = [ 0.1, 0.9 ]
p[:,0,1] = [ 0.1, 0.9 ]
p[:,0,2] = [ 0.01, 0.99 ]
p[:,1,0] = [ 0.95, 0.05 ]
p[:,1,1] = [ 0.95, 0.05 ]
p[:,1,2] = [ 0.05, 0.95 ]
add_fac(G, p, [SHUNT, PULMEMBOLUS, INTUBATION])

KINKEDTUBE = 'KINKEDTUBE'
add_var(G, KINKEDTUBE,2)
p = [ 0.04, 0.96]
add_fac(G, p, [KINKEDTUBE])

# PRESS | VENTTUBE, KINKEDTUBE, INTUBATION
PRESS = 'PRESS'
add_var(G, PRESS,4)
p = zeros(4,4,2,3)
p[:,0,0,0] = [ 0.97, 0.01, 0.01, 0.01 ]
p[:,0,0,1] = [ 0.01, 0.3, 0.49, 0.2 ]
p[:,0,0,2] = [ 0.01, 0.01, 0.08, 0.9 ]
p[:,0,1,0] = [ 0.01, 0.01, 0.01, 0.97 ]
p[:,0,1,1] = [ 0.97, 0.01, 0.01, 0.01 ]
p[:,0,1,2] = [ 0.1, 0.84, 0.05, 0.01 ]
p[:,1,0,0] = [ 0.05, 0.25, 0.25, 0.45 ]
p[:,1,0,1] = [ 0.01, 0.15, 0.25, 0.59 ]
p[:,1,0,2] = [ 0.97, 0.01, 0.01, 0.01 ]
p[:,1,1,0] = [ 0.01, 0.29, 0.3, 0.4 ]
p[:,1,1,1] = [ 0.01, 0.01, 0.08, 0.9 ]
p[:,1,1,2] = [ 0.01, 0.01, 0.01, 0.97 ]
```

```
p[:,2,0,0] = [ 0.97, 0.01, 0.01, 0.01 ]
p[:,2,0,1] = [ 0.01, 0.97, 0.01, 0.01 ]
p[:,2,0,2] = [ 0.01, 0.01, 0.97, 0.01 ]
p[:,2,1,0] = [ 0.01, 0.01, 0.01, 0.97 ]
p[:,2,1,1] = [ 0.97, 0.01, 0.01, 0.01 ]
p[:,2,1,2] = [ 0.4, 0.58, 0.01, 0.01 ]
p[:,3,0,0] = [ 0.2, 0.75, 0.04, 0.01 ]
p[:,3,0,1] = [ 0.2, 0.7, 0.09, 0.01 ]
p[:,3,0,2] = [ 0.97, 0.01, 0.01, 0.01 ]
p[:,3,1,0] = [ 0.010000001, 0.90000004, 0.080000006, 0.010000001 ]
p[:,3,1,1] = [ 0.01, 0.01, 0.38, 0.6 ]
p[:,3,1,2] = [ 0.01, 0.01, 0.01, 0.97 ]
add_fac(G, p, [PRESS, VENTTUBE, KINKEDTUBE, INTUBATION])

# VENTLUNG | VENTTUBE, KINKEDTUBE, INTUBATION
VENTLUNG = 'VENTLUNG'
add_var(G, VENTLUNG,4)
p = zeros(4,4,2,3)
p[:,0,0,0] = [ 0.97, 0.01, 0.01, 0.01 ]
p[:,0,0,1] = [ 0.95000005, 0.030000001, 0.010000001, 0.010000001 ]
p[:,0,0,2] = [ 0.4, 0.58, 0.01, 0.01 ]
p[:,0,1,0] = [ 0.3, 0.68, 0.01, 0.01 ]
p[:,0,1,1] = [ 0.97, 0.01, 0.01, 0.01 ]
p[:,0,1,2] = [ 0.97, 0.01, 0.01, 0.01 ]
p[:,1,0,0] = [ 0.97, 0.01, 0.01, 0.01 ]
p[:,1,0,1] = [ 0.97, 0.01, 0.01, 0.01 ]
p[:,1,0,2] = [ 0.97, 0.01, 0.01, 0.01 ]
p[:,1,1,0] = [ 0.95000005, 0.030000001, 0.010000001, 0.010000001 ]
p[:,1,1,1] = [ 0.5, 0.48, 0.01, 0.01 ]
p[:,1,1,2] = [ 0.3, 0.68, 0.01, 0.01 ]
p[:,2,0,0] = [ 0.97, 0.01, 0.01, 0.01 ]
p[:,2,0,1] = [ 0.01, 0.97, 0.01, 0.01 ]
p[:,2,0,2] = [ 0.01, 0.01, 0.97, 0.01 ]
p[:,2,1,0] = [ 0.01, 0.01, 0.01, 0.97 ]
p[:,2,1,1] = [ 0.97, 0.01, 0.01, 0.01 ]
p[:,2,1,2] = [ 0.97, 0.01, 0.01, 0.01 ]
p[:,3,0,0] = [ 0.97, 0.01, 0.01, 0.01 ]
p[:,3,0,1] = [ 0.97, 0.01, 0.01, 0.01 ]
p[:,3,0,2] = [ 0.97, 0.01, 0.01, 0.01 ]
p[:,3,1,0] = [ 0.01, 0.97, 0.01, 0.01 ]
p[:,3,1,1] = [ 0.01, 0.01, 0.97, 0.01 ]
p[:,3,1,2] = [ 0.01, 0.01, 0.01, 0.97 ]
add_fac(G, p, [VENTLUNG, VENTTUBE, KINKEDTUBE, INTUBATION])

FIO2 = 'FIO2'
add_var(G, FIO2,2)
p = [ 0.05, 0.95 ]
add_fac(G, p, [FIO2])

# MINVOL | VENTLUNG, INTUBATION
MINVOL = 'MINVOL'
add_var(G, MINVOL,4)
p = zeros(4, 4, 3)
p[:,0,0] = [ 0.97, 0.01, 0.01, 0.01 ]
p[:,0,1] = [ 0.01, 0.97, 0.01, 0.01 ]
p[:,0,2] = [ 0.01, 0.01, 0.97, 0.01 ]
```

```
p[:,1,0] = [ 0.01, 0.01, 0.01, 0.97 ]
p[:,1,1] = [ 0.97, 0.01, 0.01, 0.01 ]
p[:,1,2] = [ 0.6, 0.38, 0.01, 0.01 ]
p[:,2,0] = [ 0.5, 0.48, 0.01, 0.01 ]
p[:,2,1] = [ 0.5, 0.48, 0.01, 0.01 ]
p[:,2,2] = [ 0.97, 0.01, 0.01, 0.01 ]
p[:,3,0] = [ 0.01, 0.97, 0.01, 0.01 ]
p[:,3,1] = [ 0.01, 0.01, 0.97, 0.01 ]
p[:,3,2] = [ 0.01, 0.01, 0.01, 0.97 ]
add_fac(G, p, [MINVOL, VENTLUNG, INTUBATION])

# VENTALV | VENTLUNG, INTUBATION
VENTALV = 'VENTALV'
add_var(G, VENTALV,4)
p = zeros(4,4,3)
p[:,0,0] = [ 0.97, 0.01, 0.01, 0.01 ]
p[:,0,1] = [ 0.01, 0.97, 0.01, 0.01 ]
p[:,0,2] = [ 0.01, 0.01, 0.97, 0.01 ]
p[:,1,0] = [ 0.01, 0.01, 0.01, 0.97 ]
p[:,1,1] = [ 0.97, 0.01, 0.01, 0.01 ]
p[:,1,2] = [ 0.01, 0.97, 0.01, 0.01 ]
p[:,2,0] = [ 0.01, 0.01, 0.97, 0.01 ]
p[:,2,1] = [ 0.01, 0.01, 0.01, 0.97 ]
p[:,2,2] = [ 0.97, 0.01, 0.01, 0.01 ]
p[:,3,0] = [ 0.030000001, 0.95000005, 0.010000001, 0.010000001 ]
p[:,3,1] = [ 0.01, 0.94, 0.04, 0.01 ]
p[:,3,2] = [ 0.01, 0.88, 0.1, 0.01 ]
add_fac(G, p, [VENTALV, VENTLUNG, INTUBATION])

ANAPHYLAXIS = 'ANAPHYLAXIS'
add_var(G, ANAPHYLAXIS,2)
p = [ 0.01, 0.99 ]
add_fac(G, p, [ANAPHYLAXIS])

# PVSAT | VENTALV, FIO2
PVSAT = 'PVSAT'
add_var(G, PVSAT,3)
p = zeros(3,4,2)
p[:,0,0] = [ 1.0, 0.0, 0.0 ]
p[:,0,1] = [ 0.99, 0.01, 0.0 ]
p[:,1,0] = [ 0.95, 0.04, 0.01 ]
p[:,1,1] = [ 0.95, 0.04, 0.01 ]
p[:,2,0] = [ 1.0, 0.0, 0.0 ]
p[:,2,1] = [ 0.95, 0.04, 0.01 ]
p[:,3,0] = [ 0.01, 0.95, 0.04 ]
p[:,3,1] = [ 0.01, 0.01, 0.98 ]
add_fac(G, p, [PVSAT, VENTALV, FIO2])

# ARTCO2 | VENTALV
ARTCO2 = 'ARTCO2'
add_var(G, ARTCO2,3)
p = zeros(3,4)
p[:,0] = [ 0.01, 0.01, 0.98 ]
p[:,1] = [ 0.01, 0.01, 0.98 ]
p[:,2] = [ 0.04, 0.92, 0.04 ]
p[:,3] = [ 0.9, 0.09, 0.01 ]
```

```
add_fac(G, p, [ARTCO2, VENTALV])

# TPR | ANAPHYLAXIS
TPR = 'TPR'
add_var(G, TPR,3)
p = zeros(3,2)
p[:,0] = [ 0.98, 0.01, 0.01 ]
p[:,1] = [ 0.3, 0.4, 0.3 ]
add_fac(G, p, [TPR, ANAPHYLAXIS])

# SAO2 | SHUNT, PVSAT
SAO2 = 'SAO2'
add_var(G, SAO2,3)
p = zeros(3, 2, 3)
p[:,0,0] = [ 0.98, 0.01, 0.01 ]
p[:,0,1] = [ 0.01, 0.98, 0.01 ]
p[:,0,2] = [ 0.01, 0.01, 0.98 ]
p[:,1,0] = [ 0.98, 0.01, 0.01 ]
p[:,1,1] = [ 0.98, 0.01, 0.01 ]
p[:,1,2] = [ 0.69, 0.3, 0.01 ]
add_fac(G, p, [SAO2, SHUNT, PVSAT])

INSUFFANESTH = 'INSUFFANESTH'
add_var(G, INSUFFANESTH,2)
p = [ 0.1, 0.9]
add_fac(G, p, [INSUFFANESTH])

# EXPCO2 | VENTLUNG, ARTCO2
EXPCO2 = 'EXPCO2'
add_var(G, EXPCO2,4)
p = zeros(4,4,3)
p[:,0,0] = [ 0.97, 0.01, 0.01, 0.01 ]
p[:,0,1] = [ 0.01, 0.97, 0.01, 0.01 ]
p[:,0,2] = [ 0.01, 0.97, 0.01, 0.01 ]
p[:,1,0] = [ 0.01, 0.97, 0.01, 0.01 ]
p[:,1,1] = [ 0.97, 0.01, 0.01, 0.01 ]
p[:,1,2] = [ 0.01, 0.01, 0.97, 0.01 ]
p[:,2,0] = [ 0.01, 0.01, 0.97, 0.01 ]
p[:,2,1] = [ 0.01, 0.01, 0.97, 0.01 ]
p[:,2,2] = [ 0.97, 0.01, 0.01, 0.01 ]
p[:,3,0] = [ 0.01, 0.01, 0.01, 0.97 ]
p[:,3,1] = [ 0.01, 0.01, 0.01, 0.97 ]
p[:,3,2] = [ 0.01, 0.01, 0.01, 0.97 ]
add_fac(G, p, [EXPCO2, VENTLUNG, ARTCO2])

LVFAILURE = 'LVFAILURE'
add_var(G, LVFAILURE,2)
p = [ 0.05, 0.95 ]
add_fac(G, p, [LVFAILURE])

HYPOVOLEMIA = 'HYPOVOLEMIA'
add_var(G, HYPOVOLEMIA,2)
p = [ 0.2, 0.8 ]
add_fac(G, p, [HYPOVOLEMIA])

# CATECHOL | TPR, SAO2, INSUFFANESTH, ARTCO2
```

```
CATECHOL = 'CATECHOL'
add_var(G, CATECHOL,2)
p = zeros(2,3,3,2,3)
p[:,0,0,0,0] = [ 0.01, 0.99 ]
p[:,0,0,0,1] = [ 0.01, 0.99 ]
p[:,0,0,0,2] = [ 0.01, 0.99 ]
p[:,0,0,1,0] = [ 0.01, 0.99 ]
p[:,0,0,1,1] = [ 0.01, 0.99 ]
p[:,0,0,1,2] = [ 0.01, 0.99 ]
p[:,0,1,0,0] = [ 0.01, 0.99 ]
p[:,0,1,0,1] = [ 0.01, 0.99 ]
p[:,0,1,0,2] = [ 0.01, 0.99 ]
p[:,0,1,1,0] = [ 0.01, 0.99 ]
p[:,0,1,1,1] = [ 0.01, 0.99 ]
p[:,0,1,1,2] = [ 0.01, 0.99 ]
p[:,0,2,0,0] = [ 0.01, 0.99 ]
p[:,0,2,0,1] = [ 0.01, 0.99 ]
p[:,0,2,0,2] = [ 0.01, 0.99 ]
p[:,0,2,1,0] = [ 0.05, 0.95 ]
p[:,0,2,1,1] = [ 0.05, 0.95 ]
p[:,0,2,1,2] = [ 0.01, 0.99 ]

p[:,1,0,0,0] = [ 0.01, 0.99 ]
p[:,1,0,0,1] = [ 0.01, 0.99 ]
p[:,1,0,0,2] = [ 0.01, 0.99 ]
p[:,1,0,1,0] = [ 0.05, 0.95 ]
p[:,1,0,1,1] = [ 0.05, 0.95 ]
p[:,1,0,1,2] = [ 0.01, 0.99 ]
p[:,1,1,0,0] = [ 0.05, 0.95 ]
p[:,1,1,0,1] = [ 0.05, 0.95 ]
p[:,1,1,0,2] = [ 0.01, 0.99 ]
p[:,1,1,1,0] = [ 0.05, 0.95 ]
p[:,1,1,1,1] = [ 0.05, 0.95 ]
p[:,1,1,1,2] = [ 0.01, 0.99 ]
p[:,1,2,0,0] = [ 0.05, 0.95 ]
p[:,1,2,0,1] = [ 0.05, 0.95 ]
p[:,1,2,0,2] = [ 0.01, 0.99 ]
p[:,1,2,1,0] = [ 0.05, 0.95 ]
p[:,1,2,1,1] = [ 0.05, 0.95 ]
p[:,1,2,1,2] = [ 0.01, 0.99 ]

p[:,2,0,0,0] = [ 0.7, 0.3 ]
p[:,2,0,0,1] = [ 0.7, 0.3 ]
p[:,2,0,0,2] = [ 0.1, 0.9 ]
p[:,2,0,1,0] = [ 0.7, 0.3 ]
p[:,2,0,1,1] = [ 0.7, 0.3 ]
p[:,2,0,1,2] = [ 0.1, 0.9 ]
p[:,2,1,0,0] = [ 0.7, 0.3 ]
p[:,2,1,0,1] = [ 0.7, 0.3 ]
p[:,2,1,0,2] = [ 0.1, 0.9 ]
p[:,2,1,1,0] = [ 0.95, 0.05 ]
p[:,2,1,1,1] = [ 0.99, 0.01 ]
p[:,2,1,1,2] = [ 0.3, 0.7 ]
p[:,2,2,0,0] = [ 0.95, 0.05 ]
p[:,2,2,0,1] = [ 0.99, 0.01 ]
p[:,2,2,0,2] = [ 0.3, 0.7 ]
```

```
p[:,2,2,1,0] = [ 0.95, 0.05 ]
p[:,2,2,1,1] = [ 0.99, 0.01 ]
p[:,2,2,1,2] = [ 0.3, 0.7 ]

add_fac(G, p, [CATECHOL, TPR, SAO2, INSUFFANESTH, ARTCO2])

# HISTORY | LVFAILURE
HISTORY = 'HISTORY'
add_var(G, HISTORY,2)
p = zeros(2,2)
p[:,0] = [ 0.9, 0.1 ]
p[:,1] = [ 0.01, 0.99 ]
add_fac(G, p, [HISTORY, LVFAILURE])

# LVEDVOLUME | LVFAILURE, HYPOVOLEMIA
LVEDVOLUME = 'LVEDVOLUME'
add_var(G, LVEDVOLUME,3)
p = zeros(3,2,2)
p[:,0,0] = [ 0.95, 0.04, 0.01 ]
p[:,0,1] = [ 0.98, 0.01, 0.01 ]
p[:,1,0] = [ 0.01, 0.09, 0.9 ]
p[:,1,1] = [ 0.05, 0.9, 0.05 ]
add_fac(G, p, [LVEDVOLUME, LVFAILURE, HYPOVOLEMIA])

# STROKEVOLUME | LVFAILURE, HYPOVOLEMIA
STROKEVOLUME = 'STROKEVOLUME'
add_var(G, STROKEVOLUME,3)
p = zeros(3,2,2)
p[:,0,0] = [ 0.98, 0.01, 0.01 ]
p[:,0,1] = [ 0.95, 0.04, 0.01 ]
p[:,1,0] = [ 0.5, 0.49, 0.01 ]
p[:,1,1] = [ 0.05, 0.9, 0.05 ]
add_fac(G, p, [STROKEVOLUME, LVFAILURE, HYPOVOLEMIA])

ERRLOWOUTPUT = 'ERRLOWOUTPUT'
add_var(G, ERRLOWOUTPUT,2)
p = [ 0.05, 0.95 ]
add_fac(G, p, [ERRLOWOUTPUT])

# HR | CATECHOL
HR = 'HR'
add_var(G, HR,3)
p = zeros(3, 2)
p[:,0] = [ 0.05, 0.9, 0.05 ]
p[:,1] = [ 0.01, 0.09, 0.9 ]
add_fac(G, p, [HR, CATECHOL])

ERRCAUTER = 'ERRCAUTER'
add_var(G, ERRCAUTER,2)
p = [ 0.1, 0.9 ]
add_fac(G, p, [ERRCAUTER])

# CVP | LVEDVOLUME
CVP = 'CVP'
add_var(G, CVP,3)
p = zeros(3,3)
```

```
p[:,0] = [ 0.95, 0.04, 0.01 ]
p[:,1] = [ 0.04, 0.95, 0.01 ]
p[:,2] = [ 0.01, 0.29, 0.7 ]
add_fac(G, p, [CVP, LVEDVOLUME])

# PCWP | LVEDVOLUME
PCWP = 'PCWP'
add_var(G, PCWP,3)
p = zeros(3,3)
p[:,0] = [ 0.95, 0.04, 0.01 ]
p[:,1] = [ 0.04, 0.95, 0.01 ]
p[:,2] = [ 0.01, 0.04, 0.95 ]
add_fac(G, p, [PCWP, LVEDVOLUME])

# CO | STROKEVOLUME, HR
CO = 'CO'
add_var(G, CO,3)
p = zeros(3,3,3)
p[:,0,0] = [ 0.98, 0.01, 0.01 ]
p[:,0,1] = [ 0.95, 0.04, 0.01 ]
p[:,0,2] = [ 0.8, 0.19, 0.01 ]
p[:,1,0] = [ 0.95, 0.04, 0.01 ]
p[:,1,1] = [ 0.04, 0.95, 0.01 ]
p[:,1,2] = [ 0.01, 0.04, 0.95 ]
p[:,2,0] = [ 0.3, 0.69, 0.01 ]
p[:,2,1] = [ 0.01, 0.3, 0.69 ]
p[:,2,2] = [ 0.01, 0.01, 0.98 ]
add_fac(G, p, [CO, STROKEVOLUME, HR])

# HRBP | HR, ERRLOWOUTPUT
HRBP = 'HRBP'
add_var(G, HRBP,3)
p = zeros(3,3,2)
p[:,0,0] = [ 0.98, 0.01, 0.01 ]
p[:,0,1] = [ 0.4, 0.59, 0.01 ]
p[:,1,0] = [ 0.3, 0.4, 0.3 ]
p[:,1,1] = [ 0.98, 0.01, 0.01 ]
p[:,2,0] = [ 0.01, 0.98, 0.01 ]
p[:,2,1] = [ 0.01, 0.01, 0.98 ]
add_fac(G, p, [HRBP, HR, ERRLOWOUTPUT])

# HREKG | HR, ERRCAUTER
HREKG = 'HREKG'
add_var(G, HREKG,3)
p = zeros(3,3,2)
p[:,0,0] = [ 0.33333334, 0.33333334, 0.33333334 ]
p[:,0,1] = [ 0.33333334, 0.33333334, 0.33333334 ]
p[:,1,0] = [ 0.33333334, 0.33333334, 0.33333334 ]
p[:,1,1] = [ 0.98, 0.01, 0.01 ]
p[:,2,0] = [ 0.01, 0.98, 0.01 ]
p[:,2,1] = [ 0.01, 0.01, 0.98 ]
add_fac(G, p, [HREKG, HR, ERRCAUTER])

# HRSAT | HR, ERRCAUTER
HRSAT = 'HRSAT'
add_var(G, HRSAT,3)
```

```
p = zeros(3,3,2)
p[:,0,0] = [ 0.33333334, 0.33333334, 0.33333334 ]
p[:,0,1] = [ 0.33333334, 0.33333334, 0.33333334 ]
p[:,1,0] = [ 0.33333334, 0.33333334, 0.33333334 ]
p[:,1,1] = [ 0.98, 0.01, 0.01 ]
p[:,2,0] = [ 0.01, 0.98, 0.01 ]
p[:,2,1] = [ 0.01, 0.01, 0.98 ]
add_fac(G, p, [HRSAT, HR, ERRCAUTER])

# BP | TPR, CO
BP = 'BP'
add_var(G, BP,3)
p = zeros(3, 3, 3)
p[:,0,0] = [ 0.98, 0.01, 0.01 ]
p[:,0,1] = [  0.98, 0.01, 0.01 ]
p[:,0,2] = [ 0.9, 0.09, 0.01 ]
p[:,1,0] = [ 0.98, 0.01, 0.01 ]
p[:,1,1] = [ 0.1, 0.85, 0.05 ]
p[:,1,2] = [ 0.05, 0.2, 0.75 ]
p[:,2,0] = [ 0.3, 0.6, 0.1 ]
p[:,2,1] = [ 0.05, 0.4, 0.55 ]
p[:,2,2] = [ 0.01, 0.09, 0.9 ]
add_fac(G, p, [BP, TPR, CO])

zeros = np.zeros

# # # # # # # # # # # # # # # # # # # # # # #
# MAIN

if __name__=='__main__':
    for n in G.node:
        print
        print '%s' % (n)
        print '%s' % (G.node[n])

    print
    print G.vals(HR)

    nx.draw(G);show()
```

Port the Matlab to Python

```
#!/usr/bin/python
from __future__ import division

from numpy import *
from matplotlib.pyplot import *
import nltk
import numpy.random as sample
import scipy.stats as pdf
import networkx as nx
from copy import deepcopy

from sam.sam import *
from sam import sam

"""
init factor graph => add vars => add facs

factor graph : bipartite btwn factors and variables

var
: Maybe Val
has dim
has facts

fact
has potential
has vars
vars = domain potential

repr graph as adjacencies
repr pdfs as numeric potential tables

"""

class FactorGraph(nx.Graph):
    def __init__(self, data=None, **attr):
        """
        factor graph : bipartite btwn factors and variables

        node
        | var
        | fac

        edge
        : from var to fac

        """
        super(FactorGraph, self).__init__(data=data, **attr)

        self.graph['vars'] = [] # in deterministic order
        self.graph['facs'] = [] # in deterministic order


    def subgraph(self, vars, condition={}):
```

```
        """
        must condition or marginalize complement


        """

        H = deepcopy(self)

        facs = list(set(flatten([H.N(v) for v in vars])))

        complement = set([v  for v in H.vars()  if v not in set(vars)])
        if condition:
            assert set(condition) <= set(complement)
            for v in condition:
                assert condition[v] < self.node[v]['d']
        complement = set(complement) - set(condition)

        # condition by slicing factor
        H.condition(**condition)

        # marginalize by elimination
        H.eliminate(*complement)

        return H


    def eliminate(self, *vs):
        """
        vs : [var]

        """

        for v in vs:
            for f in self.N(v):
                fac = self.node[f]

                i = fac['vars'].index(v)
                fac['vars'].remove(v)

                if len(fac['pmf'].shape) > 1:
                    # sum pmf
                    # eg sum at i=1. shape(2,3,4) => shape(2,4)
                    fac['pmf'] = sum( fac['pmf'], axis=i )

                else:
                    # no other var needs this fac
                    self.remove_node(f)
                    self.graph['facs'].remove(f)

            self.remove_node(v)
            self.graph['vars'].remove(v)


    def condition(self, **vxs):
        """
        vxs : {var: val, ...}
```

```
    """

    for v,x in vxs.items():

        for f in self.N(v):
            fac = self.node[f]

            i = fac['vars'].index(v)
            fac['vars'].remove(v)

            if len(fac['pmf'].shape) > 1:
                # slice pmf
                # eg rollaxis. i=1  shape(2,3,4) => shape(3,2,4)
                # eg slice.     x=_  shape(3,2,4) => shape(2,4)
                fac['pmf'] = rollaxis( fac['pmf'], i,0 )[x,:]

            else:
                # no other var needs this fac
                self.remove_node(f)
                self.graph['facs'].remove(f)

        self.remove_node(v)
        self.graph['vars'].remove(v)


def new(self, name):
    n = { 'x': len(self.vars()),
          'f': len(self.facs()),
        }[name]

    name = '%s%d' % (name, n)

    while name in self:
        n = n+1
        name = '%s%d' % (name, n)

    return name


def add_var(self, name=None, d=0, val=None):
    """
    variable has...
    v : Maybe Val
    factors fs : [node]
    dimensionality d : nat

    eg
    val is unknown
    fs = [X Y]
    d = 3
    """

    if val:
        if d:
            assert d == len(val)
```

```
        else:
            d = len(vals)

    if name in self:
        raise ValueError('%s in graph' % name)

    if name is None:
        name = self.new('x')

    self.add_node( name, d=d, type='var' )
    self.graph['vars'].append( name )


def add_fac(self, p, vars, name=None):
    """
    factor has...
    potential p : [real]
    vars = nodes

    eg
    p(a,b,c, ..)  :=  potential when A=a, B=b, C=c, ..
    vars = [A B C ..]


    """
    p = pd(p)

    if name is None or name in self:
        name = self.new('f')

    p = array(p)

    self.add_node( name, pmf=p, vars=vars, type='fac' ) # 'vars' for order
    self.graph['facs'].append( name )

    for x in vars:
        self.add_edge( name, x )


def N(self, x):
    t = self.node[x]['type']

    if t=='var':
        return [x for x in self.edge[x].keys()]

    if t=='fac':
        return self.node[x]['vars']

    raise ValueError('%s must be var or fac' % x)


def type(self, node):
    return self.node[node]['type']


def vars(self, but=None):
```

```
        #return [x for x in self  if self.node[x]['type']=='var']
        return self.graph['vars']

    def facs(self, but=None):
        #return [x for x in self  if self.node[x]['type']=='fac']
        return self.graph['facs']


    def __call__(self, fac, *vals):
        #print '%s%s' % (fac, vals)
        f = self.node[fac]['pmf'] #: table
        return f[vals]

    def val(self, var):
        assert self.type(var) == 'var'
        return G.node[var]['x']

    def vals(self, var):
        """
        discrete random variables
        ->
        all functions on them are arrays
        all their values are indices
        """

        assert self.type(var) == 'var'
        return range(self.node[var]['d'])

    def conditions(self, var, val):
        pass

#############################
# Functions on Factor Graphs


def isTree(G):
    return nx.cycle_basis(G) == []


# Example Factor Graphs


def factor_tree():
    """
    4 vars
    2 facs

    """

    G = FactorGraph()
    a,b,c,d = 'a','b','c','d'
    G.graph['root'] = c

    G.add_var(a, d=1)
    G.add_var(b, d=2)
    G.add_var(c, d=3)
```

```
    G.add_var(d, d=4)

    p1 = pd(magic(1))
    p2 = pd(magic(2))
    p4 = pd(magic(4))

    G.add_fac(p1, [a], name='f[a]')
    G.add_fac(p2, [b], name='f[b]')
    G.add_fac(p4, [d], name='f[d]')

    p13 = pd(magic((1,3)))
    p23 = pd(magic((2,3)))
    p34 = pd(magic((3,4)))

    G.add_fac(p13, [a,c], name='f[ac]')
    G.add_fac(p23, [b,c], name='f[bc]')
    G.add_fac(p34, [c,d], name='f[cd]')

    return G


def factor_small():
    """ small list """

    G = FactorGraph()

    a,b = 'a','b'
    G.add_var(a, d=2)
    G.add_var(b, d=2)

    fa  = pd(magic((2)))
    fab = pd(magic((2,2)))
    G.add_fac(fa,  [a],   name='f[a]')
    G.add_fac(fab, [a,b], name='f[ab]')

    return G


def factor_list():
    """
    4 vars
    3 fac

    """

    G = FactorGraph()

    a,b,c,d = 'a','b','c','d'
    G.add_var(a, d=1)
    G.add_var(b, d=2)
    G.add_var(c, d=3)
    G.add_var(d, d=4)

    p12 = pd(magic((1,2)))
    p23 = pd(magic((2,3)))
    p34 = pd(magic((3,4)))
```

```
    G.add_fac(p12, [a,b], name='f[ab]')
    G.add_fac(p23, [b,c], name='f[bc]')
    G.add_fac(p34, [c,d], name='f[cd]')

    return G


def factor_clique():
    """
    4 vars
    1 fac

    """

    G = FactorGraph()

    a,b,c,d = 'a','b','c','d'
    G.add_var(a, d=1)
    G.add_var(b, d=2)
    G.add_var(c, d=3)
    G.add_var(d, d=4)

    p = pd(magic((1,2,3,4)))
    G.add_fac(p, [a,b,c,d], name='f[abcd]')

    return G


def factor_3f1v():
    G = FactorGraph()
    v = 'v'
    d = 3
    G.add_var(v, d=d)

    p1 = pd(ones(d))
    p2 = pd(magic(d))
    p3 = array([0.8,0.15,0.05])

    G.add_fac(p1, v, name='f1')
    G.add_fac(p2, v, name='f2')
    G.add_fac(p3, v, name='f3')

    return G


def factor_1f3v():
    G = FactorGraph()

    a,b,c = 'a','b','c'
    G.add_var(a, d=2)
    G.add_var(b, d=3)
    G.add_var(c, d=4)

    f = pd(magic((2,3,4)))
    G.add_fac(f, [a, b, c], name='f')
```

```
    return G


def factor_square():
    G = FactorGraph()

    a,b,c,d = 'a','b','c','d'
    G.add_var(a, d=2)
    G.add_var(b, d=3)
    G.add_var(c, d=4)
    G.add_var(d, d=5)

    ab = pd(magic((2,3)))
    bc = pd(magic((3,4)))
    cd = pd(magic((4,5)))
    da = pd(magic((5,2)))
    G.add_fac(ab, [a,b], name='f[ab]')
    G.add_fac(bc, [b,c], name='f[bc]')
    G.add_fac(cd, [c,d], name='f[cd]')
    G.add_fac(da, [d,a], name='f[da]')

    return G

def factor_xy():
    G = FactorGraph()

    G.add_var('x', d=2)
    G.add_var('y', d=2)
    G.add_fac( pd(magic((2,2))), ['x','y'], name='f' )

    return G

def factor_btree():
    G = FactorGraph()

    a = 'a'
    b1,b2 = 'b1','b2'
    c1,c2,c3,c4 = 'c1','c2','c3','c4'
    for v in [a, b1,b2, c1,c2,c3,c4]: G.add_var(v, d=2)

    p = pd(magic((2,2)))
    G.add_fac(p, [a,b1])
    G.add_fac(p, [a,b2])
    G.add_fac(p, [b1,c1])
    G.add_fac(p, [b1,c2])
    G.add_fac(p, [b2,c1])
    G.add_fac(p, [b2,c2])

    return G

if __name__=='__main__':

    # T = factor_tree()
    # L = factor_list()
    # C = factor_clique()
    # S = factor_square()
```

```
    # nx.draw(factor_square())
    # show()


    div('testing FactorGraph.subgraph( [var ...], condition={var:val, ...} )')

    G = factor_square()
    a,b,c,d = G.vars()


    alert('testing conditioning...')
    H = G.subgraph([a,c], condition={d:5-1})

    assert H.vars() == [a,c]

    fcd = H.node['f[cd]']
    fda = H.node['f[da]']

    assert fcd['pmf'].shape == (H.node[c]['d'],)
    assert fda['pmf'].shape == (H.node[a]['d'],)

    var('f cd', fcd)
    var('f da', fda)
    var('H', H.node)


    alert('testing marginalization...')

    var('xy', factor_xy().node)

    G = factor_xy()
    G.eliminate('x')
    var('elim x', G.node)
    assert near( G.node['f']['pmf'] , array([0.4, 0.6]) )

    G = factor_xy()
    G.eliminate('y')
    var('elim y', G.node)
    assert near( G.node['f']['pmf'] , array([0.3, 0.7]) )

    G = factor_xy()
    G.eliminate('x','y')
    var('elim x y', G.node)
    assert G.node == {}

    G = factor_xy()
    G.eliminate('y','x')
    var('elim y x', G.node)
    assert G.node == {}

""" ?
order of FactorGraph.vars() or FactorGraph.facs() matters
"""
```

2 Inference on ALARM

2.a

sumprod was exact
mean PULMEMBOLUS = 1.99
mean INTUBATION = 1.12999999822
mean KINKEDTUBE = 1.96000000177
mean VENTTUBE = 3.9400000013

2.b

sumprod was exact
mean PULMEMBOLUS = 1.98107225552
mean INTUBATION = 2.90293067903
mean KINKEDTUBE = 1.9985228765
mean VENTTUBE = 3.92576957723

2.c

sumprod was approx (VENTLUNG and INTUBATION are off by over 10%)

[bruteforce means]
mean VENTLUNG = 3.42842646489
mean KINKEDTUBE = 1.99787701961
mean INTUBATION = 1.54975162592
mean VENTTUBE = 3.45951755981

[sumprod means]
mean VENTLUNG = 2.66478879532
mean KINKEDTUBE = 1.9931210537
mean INTUBATION = 1.91289992443
mean VENTTUBE = 3.43053893744

2.d

if we think of the factor graphs as just undirected graphs, all three have cycles. but if we only think about the variables communicating via factors, then the (c) subgraph has MINVOL, which makes a cycle between VENTLUNG and INTUBATION that propagates messages. this is consistent in that only VENTLUNG and INTUBATION are significantly different from their exact values.

2.e

sumprod probably computes inexact marginals. the graph has too many variable cycles.

[sumprod: converged (eps=1e-06) in 14 iterations]
mean DISCONNECT = 1.9
mean PULMEMBOLUS = 1.99001429995
mean INSUFFANESTH = 1.90043976042
mean LVFAILURE = 1.95
mean ANAPHYLAXIS = 1.99106463727

2.f

i don't think these estimates coincide with the true means.

[sumprod: converged (eps=1e-06) in 24 iterations]
mean DISCONNECT = 1.90000000337
mean HYPOVOLEMIA = 1.8
mean LVFAILURE = 1.95
mean KINKEDTUBE = 1.9600000015
mean INTUBATION = 1.12999999528
mean INSUFFANESTH = 1.9
mean ANAPHYLAXIS = 1.99
mean PULMEMBOLUS = 1.99

2.g

[sumprod: iterated too many times (N=500) with (diff=0.576824503)]
mean DISCONNECT = 1.9046026259
mean HYPOVOLEMIA = 1.8022572855
mean LVFAILURE = 1.00455640119
mean KINKEDTUBE = 1.95330338292
mean INTUBATION = 1.21543803025
mean INSUFFANESTH = 1.90640901319
mean ANAPHYLAXIS = 1.99181228874
mean PULMEMBOLUS = 1.99059265509

2.h

the only significant difference between the means of (g) and (h) is INTUBATION, differing by over 10%. this makes sense as INTUBATION is in the most factors of any of the diagnostic variables.

the most significant difference is that (h) converges whereas (g) doesn't.

[sumprod: converged (eps=1e-06) in 289 iterations]
mean DISCONNECT = 1.89900334967
mean HYPOVOLEMIA = 1.80280398577
mean LVFAILURE = 1.00519005918
mean KINKEDTUBE = 1.96375990063
mean INTUBATION = 1.46785783435
mean INSUFFANESTH = 1.8986736855
mean ANAPHYLAXIS = 1.98457027707
mean PULMEMBOLUS = 1.9899078908

ALARM

```
#!/usr/bin/python
from __future__ import division
from sam.sam import *

from numpy import *
from matplotlib.pyplot import *
import numpy.random as sample
import scipy.stats as pdf

from sumproduct import *
from main import *

"""
run sumprod until
max |message[t] - message[t-1]| < 1e-6
|iterations| > 500


"""

runA=1
runB=1
runC=1

runE=1
runF=1
runG=1
runH=1

VERBOSE = 0


def expectation(p,f, xs):
    return sum( array([p(x) for x in xs]) * array([f(x) for x in xs]) )

def mean(p, xs):
    return dot(p, xs)

def means(G, marginals):
    for v, p in marginals.items():
        var( 'mean %s' % v,  mean(p, [1+x for x in range(G.node[v]['d'])]), new=False, tab=True


def test(H, fail=False, vars=None):
    if not vars: vars = H.vars()

    marginals = marginalize_sumprod(H, vars=vars, verbose=VERBOSE)
    _marginals = marginalize_bruteforce(H, vars=vars)

    var('[sumprod]', marginals)
    var('[bruteforce]', _marginals)

    same = compare(marginals , _marginals, fail=fail)
```

```
    if same:
        means(H, marginals)
    else:
        alert('[sumprod] != [bruteforce]');print

        alert('[bruteforce means]')
        means(H, _marginals)

        alert('[sumprod means]')
        means(H, marginals)



""" A
condition on VENTMACH=4-1 and DISCONNECT=2-1
 index 2-1 and 4-1 on all their facs
 renormalize
marginalize all others (i.e. in some fac of some var, but not a var)

sumprod => means of causes
cmp to bruteforce

sumprod: converged (eps=1e-06) in 5 iterations
mean PAP = 2.0079
mean VENTLUNG = 2.10221452209
mean SHUNT = 1.10309499984
mean VENTTUBE = 3.9400000013
mean KINKEDTUBE = 1.96000000177
mean INTUBATION = 1.12999999822
mean PRESS = 2.2487738396
mean PULMEMBOLUS = 1.99

"""
div('A')

if runA:
    causes  = [PULMEMBOLUS, INTUBATION, VENTTUBE, KINKEDTUBE]
    effects = [PAP, SHUNT, PRESS, VENTLUNG]
    vars = causes + effects
    H = G.subgraph(vars, condition={VENTMACH: 4-1, DISCONNECT: 2-1})

    test(H, fail=True, vars=causes)




""" B
also condition on SHUNT=2=1 and PRESS=4=3
(python has zero-based indexing, and i index with vals)

sumprod => means of causes
cmp to bruteforce
```

```
# same as enumerate-marginals on approx

# mean VENTTUBE = 3.92576957844
# mean KINKEDTUBE = 1.99852287653
# mean INTUBATION = 2.90293068064
# mean PULMEMBOLUS = 1.98107225563

"""
div('B')

if runB:
    causes  = [PULMEMBOLUS, INTUBATION, VENTTUBE, KINKEDTUBE]
    effects = [PAP, VENTLUNG]
    vars = causes + effects
    H = G.subgraph(vars, condition={ VENTMACH: 4-1, DISCONNECT: 2-1, SHUNT: 2-1, PRESS: 4-1 })

    test(H, vars=causes)




""" C
VENTMACH=4-1 DISCONNECT=2-1 PRESS=4-1 MINVOL=2-1

sumprod => means of unobserved
cmp to bruteforce


[bruteforce] means
mean VENTLUNG = 3.42842646489
mean KINKEDTUBE = 1.99787701961
mean INTUBATION = 1.54975162592
mean VENTTUBE = 3.45951755981

sumprod: converged (eps=1e-06) in 105 iterations
[sumprod] means
mean VENTLUNG = 2.66478879532
mean KINKEDTUBE = 1.9931210537
mean INTUBATION = 1.91289992443
mean VENTTUBE = 3.43053893744

"""
div('C')

if runC:
    unobserved  = [INTUBATION, VENTTUBE, KINKEDTUBE, VENTLUNG]
    H = G.subgraph(unobserved, condition={ VENTMACH: 4-1, DISCONNECT: 2-1, PRESS: 4-1, MINVOL: 2

    test(H, vars=unobserved)

""" D
discuss what caused exact v approx marginals
```

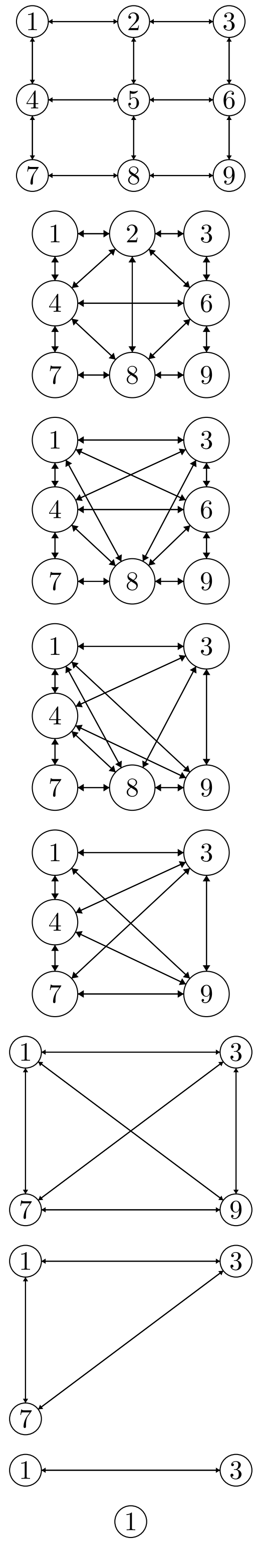
```
"""


""" E

probably inexact. the graph is too big

sumprod: converged (eps=1e-06) in 14 iterations
mean DISCONNECT = 1.9
mean PULMEMBOLUS = 1.99001429995
mean INSUFFANESTH = 1.90043976042
mean LVFAILURE = 1.95
mean ANAPHYLAXIS = 1.99106463727

"""
div("E")

if runE:
    H = deepcopy(G)
    H.condition(HYPOVOLEMIA=0, HR=0, INTUBATION=0, KINKEDTUBE=0, VENTALV=0)
    vars = [LVFAILURE, ANAPHYLAXIS, INSUFFANESTH, PULMEMBOLUS, DISCONNECT]

    marginals = marginalize_sumprod(H, vars=vars, verbose=VERBOSE)
    means(H, marginals)



""" F


[sumprod: converged (eps=1e-06) in 24 iterations]
mean DISCONNECT = 1.90000000337
mean HYPOVOLEMIA = 1.8
mean LVFAILURE = 1.95
mean KINKEDTUBE = 1.9600000015
mean INTUBATION = 1.12999999528
mean INSUFFANESTH = 1.9
mean ANAPHYLAXIS = 1.99
mean PULMEMBOLUS = 1.99

"""
div("F")

if runF:
    H = deepcopy(G)
    vars = [LVFAILURE, HYPOVOLEMIA, ANAPHYLAXIS, INSUFFANESTH, PULMEMBOLUS, INTUBATION, DISCONNI

    marginals = marginalize_sumprod(H, vars=vars, verbose=VERBOSE)
    means(H, marginals)
```

```
""" G

[sumprod: iterated too many times (N=500) with (diff=0.576824503)]
\newline mean DISCONNECT        =       1.9046026259
\newline mean HYPOVOLEMIA       =       1.8022572855
\newline mean LVFAILURE =       1.00455640119
\newline mean KINKEDTUBE        =       1.95330338292
\newline mean INTUBATION        =       1.21543803025
\newline mean INSUFFANESTH      =       1.90640901319
\newline mean ANAPHYLAXIS       =       1.99181228874
\newline mean PULMEMBOLUS       =       1.99059265509

"""
div("G")

if runG:
    H = deepcopy(G)
    H.condition( HISTORY=0, CVP=0, PCWP=0, BP=0, HRBP=0, HREKG=0, HRSAT=0, EXPCO2=0, MINVOL=0 )
    vars = [LVFAILURE, HYPOVOLEMIA, ANAPHYLAXIS, INSUFFANESTH, PULMEMBOLUS, INTUBATION, DISCONN

    marginals = marginalize_sumprod(H, vars=vars, verbose=VERBOSE)
    means(H, marginals)






""" H

[sumprod: converged (eps=1e-06) in 289 iterations]
\newline mean DISCONNECT        =       1.89900334967
\newline mean HYPOVOLEMIA       =       1.80280398577
\newline mean LVFAILURE =       1.00519005918
\newline mean KINKEDTUBE        =       1.96375990063
\newline mean INTUBATION        =       1.46785783435
\newline mean INSUFFANESTH      =       1.8986736855
\newline mean ANAPHYLAXIS       =       1.98457027707
\newline mean PULMEMBOLUS       =       1.9899078908

"""
div("H")

if runH:
    for v in [HRBP, HREKG, HRSAT]:
        var( 'max %s' % v, G.node[v]['d'] )

    H = deepcopy(G)
    H.condition( HISTORY=0, CVP=0, PCWP=0, BP=0,
                 HRBP=3-1, HREKG=3-1, HRSAT=3-1,
                 EXPCO2=0, MINVOL=0 )
    vars = [LVFAILURE, HYPOVOLEMIA, ANAPHYLAXIS, INSUFFANESTH, PULMEMBOLUS, INTUBATION, DISCONN

    marginals = marginalize_sumprod(H, vars=vars, N=2000, verbose=VERBOSE)
    means(H, marginals)
```

3 Elimination on 2D Grids

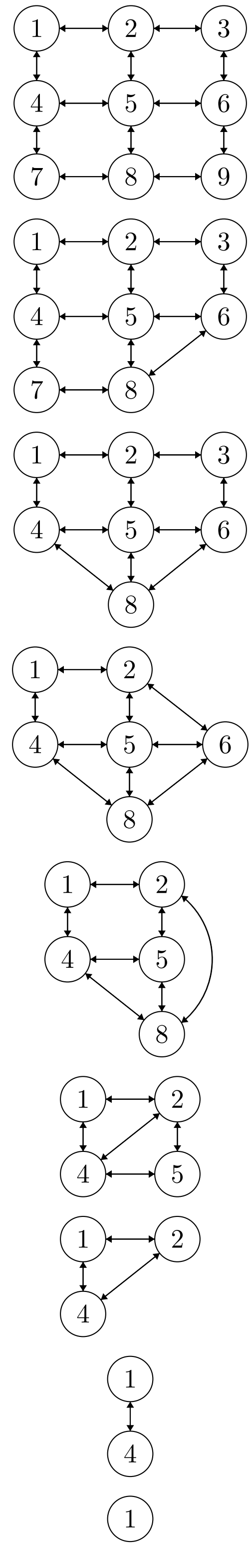
3.a

the maximal elimination clique is 5 once we eliminate (2) (the graph gets connected and stays connected).



3.b

the maximal elimination clique is only 3. this is the more efficient elimination ordering (probably the minimal one).



3.c

n^2 would be trivially true as it about the tree width of an $n \times n$ clique. assuming the elimination ordering of (3b) is optimal or near optimal (which should be the point, as it works in from the lower-degree corners and edges), there are only $3-$cliques. also, the maximal clique of the minimal elimination ordering of a 2×2 grid is 3 (by symmetry every elimination ordering is isomorphic). i played around with a 4×4 grid but i couldn't reduce by some optimal elimination ordering to a 3×3. i also tried to see if i could find an elimination ordering for the 4×4 that preserves a constant 3 clique, by working in from(i.e. eliminating first) the corners and edges; or row-by-row and col-by-col. but i saw $4-$cliques. however, i can't prove either that this was a *minimal* elimination ordering or a *maximal* clique (the "minimax" definition of treewidth). it's probably the minimal elimination ordering, but it might not be the maximal elimination clique. the treewidth could be (unlikely) some $O(n)$ or (even less likely) a const $3 - 1$.

thus, i say the treewidth of an $n \times n$ grid graph is $n - 1$ as i think the maximal elimination clique of the minimal elimination ordering is n.