COMP 4680/8650: Advanced topics in Statistical Machine Learning

Assignment II: Programming Assignment

Manab Chetia, u5492350,
Vadim Solviev, u4573325

1. Procedure

1.1 Brute Force

Chain

1. Using `itertools` we have generated all of the 2^n possible states sequences and stored them in a `numpy` array called configurations. E.g. For n=3, `configurations=[[000][001][010][011][100][101][110][111]]`
2. We assume that each element of the `numpy` array `configurations` represents a possible sequence in the state space.
3. Let's consider `[010]` to be one of the possible sequences of states. We denote the first element of the chain as x_1 i.e. 0 here. Hence $\boldsymbol{P}(x_1) = 0.05$ based on the starting probabilities that we are given. We check the current element and the previous element and calculate the $\boldsymbol{P}(x_n|x_{n-1})$ based on the transition probabilities that we are given.
4. Than we multiply all the probabilities of step 3 to get the joint probability of that configuration as $\boldsymbol{P}(x_1, \dots, x_n) = \boldsymbol{P}(x_1)\boldsymbol{P}(x_2|x_1) \dots \boldsymbol{P}(x_n|x_{n-1})$

   Code is:

```
def getJointProbablity(configuration):
    x1 = configuration[0]
    if(x1) == 0:
        p_x1 = 0.05
    else:
        p_x1 = 0.95
    pii   = 1.0
    n = configuration.size
    for j in range(1, n):
        i   = configuration[j]
        i_1 = configuration[j-1]
        pii *= getConditionalProb(i, i_1)
    prob = p_x1 * pii
    return prob
```

5. To calculate $\boldsymbol{P}(x_5 = 1 | x_1 = 1)$, we sum over all the joint probabilities from step 3 for configurations that have (1 in position 5 and 1 in position 1) of the array and also separately sum over the all the joint probabilities that have (1 in position 1) of the array and then divide the former by the later to get the normalized joint probability of the event.

   Code is:

```
prob2     = 0.0
probx1_1  = 0.0

for configuration in configurations:
        if (configuration[4] == 1 and configuration[0] == 1):
            prob2 += getJointProbablity(configuration)
        if (configuration[0] == 1):
            probx1_1 += getJointProbablity(configuration)

print("p( x5=1 | x1=1 ) : {}".format(prob2 / probx1_1))
```

Tree

1. If L represents the number of layers, then number of nodes $n = 2^L - 1$.
2. Using `itertools` we generate all the 2^n possible states and stored them in a `numpy` array called `configurations`. E.g. For `n=3`, `configurations = [[000][001][010][011][100][101][110][111]]`
3. We created a `dictionary` where the `keys` of the dictionary represent the indices of the node in a tree and `values` represent the value of that node i.e. 0/1. This is an implementation detail introduces to simplify the calculations.

   When we create the dictionary for the first time, we initialize all the values to 0.

   Code is:

```
def createTree(n):
    tree = {x+1: 0 for x in xrange(n)}
    return tree
```

4. As I loop through all the `configurations` created in step 2, I assign each element of a `configuration` as values of the dictionary.

   Code is:

```
def assignValuesToTree(grid, configuration):
    for i in tree.iterkeys():
        grid[i]=configuration[i-1]
    return tree
```

5. Joint probability of a configuration calculated using the formula

$$p(x_1, \dots, x_n) = p(x_1) \prod_{i=2}^{n} p\left(x_i | x_{i/2}\right)$$

   The code is:

```
def getJointProbablity(tree, configuration):
    jointProbability = 1.0
    for index in tree.iterkeys():
        if   (  index == 1  ):
            if (tree[index] == 1):
                jointProbability *= 0.95
            else:
                jointProbability *= 0.05
        else:
            jointProbability *= getTransitionProb(tree[index], tree[index/2])
    return jointProbability
```

6. To calculate $\boldsymbol{P}(x_5 = 1 | x_1 = 1)$, we sum over all the joint probabilities that we get in step 5 for configurations that have (1 in position 5 and 1 in position 1) of the array and also separately sum over the all the joint probabilities that has (1 in position 1) of the array and then divide the former by the later to normalize.

Code is:

```
prob2     = 0.0
probx1_1  = 0.0

for configuration in configurations:
      treeNew = assignValuesToGrid(tree, configuration)
      if (treeNew[5] == 1 and treeNew[1] == 1):
          prob2 += getJointProbablity(configuration)
      if (treeNew[1] == 1):
          probx1_1 += getJointProbablity(configuration)

print("p( x5=1 | x1=1 ) : {}".format(prob2 / probx1_1))
```

Grid

1. If L represents the number of layers then number of nodes $n = L * L$.
2. Using `itertools` we generate all 2^n possible states and stored them in a `numpy` array called configurations. E.g. For `n=3`, `configurations = [[000][001][010][011][100][101][110][111]]`.
3. We have create `dictionary` where the `keys` of the dictionary represent the indices of a node in the grid and `values` represent the value of a node i.e. `0/1`. When we create the dictionary it is initialized with zeros.

   Code is:

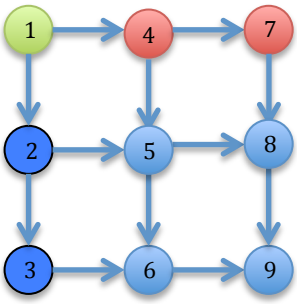
```
def createGrid(n):
    grid = {x+1: 0 for x in xrange(n*n)}
    return grid
```

4. We iterate through all `configurations` created in step 2 and assign each element of a `configuration` as a value in the dictionary.

   Code is:

```
def assignValuesToGrid(grid, configuration):
    for i in grid.iterkeys():
        grid[i]=configuration[i-1]
  return grid
```

5. Now considering a grid with $L = 3$



- Node 1 has no parents, therefore $\boldsymbol{P}(x_1)$ can be calculated taken from the information provided to us.
- Looking at the first row nodes (4, 7) and first column nodes (2, 3) of the grid, we see that only the 1 neighbor is pointing to the next one.

  In case of first column $1 \rightarrow 2, 2 \rightarrow 3$, we see all indices are $\leq L$ i.e.

$$\text{Node}(index \leq L) \rightarrow \text{Node}(index)$$

  In case of first row $1 \rightarrow 4, 4 \rightarrow 7$, the difference between indices (4, 7) with respect to index 1 is a multiple of 3 i.e. a multiple of number of Layers L i.e.

$$\text{Node}(\ (index - 1)\%L == 0\ ) \ \rightarrow \ \text{Node}(index).$$

The rest of the node, such as (5, 6, 7, 8), have 2 arrows pointing to them. E.g. considering index 5, we see $4 \rightarrow 5, 2 \rightarrow 5$, therefore we can say the nodes that pointing to it correspond to the pattern

$$\text{Node}(index - 1), \text{Node}(index - L) \rightarrow \text{Node}(index).$$

6. Following the patterns in 5, we can calculate the joint probability of a configuration using the probabilities provided to us using the following code.

```
def getJointProbability(L, grid):
    jointProbability = 1.0
    for position in grid.iterkeys():
        if   (  index == 1  ):
            jointProbability *= 0.5
        elif ( (index != 1)  and  (  index <= L)  ):
            jointProbability *= getCondtionalProb( grid[index], grid[index-1]
)
        elif ( (index!= 1)  and  ( (index-1)%L == 0 ) ):
            jointProbability *= getCondtionalProb( grid[index], grid[index-L]
)
        else:
            jointProbability  *=  getCondtionalProb(  grid[index],  grid[index-
1], grid[index-L] )
    return jointProbability
```

7. To calculate $\boldsymbol{P}(x_5 = 1 | x_1 = 1)$, we sum over all the joint probabilities that we get in step 6 for the configurations that has 1 in the 5th node of the array.

   Code is:

```
prob1    = 0.0
probx1_1 = 0.0
for configuration in configurations:
    gridNew = assignValuesToGrid(grid, configuration)
    if (gridNew[5] == 1 and gridNew[1] == 1):
          prob1 += getJointProbablity(configuration)
    if (gridNew[1] == 1):
          probx1_1 += getJointProbablity(configuration)

print("p( x5=1 | x1=1 ) : {}".format(prob1 / probx1_1))
```

1.2 Belief Propagation/Sum Product[1]

This is an exact inference algorithm for finding marginals in a tree like graph and therefore it cannot be used on Grids.

Chain

Considering the chain below and we want to calculate $\boldsymbol{P}(x_3)$:

$\boldsymbol{P}(x_1)$ $[0.7\ 0.3]$ $\quad$ $\boldsymbol{P}(x_2|x_1)$ $\begin{bmatrix}0.4 & 0.6\\ 0.5 & 0.5\end{bmatrix}$ $\quad$ $\boldsymbol{P}(x_3|x_2)$ $\begin{bmatrix}0.8 & 0.2\\ 0.2 & 0.8\end{bmatrix}$ $\quad$ $\boldsymbol{P}(x_4|x_3)$ $\begin{bmatrix}0.5 & 0.5\\ 0.7 & 0.3\end{bmatrix}$

$$\boldsymbol{P}(x_3) = \sum_{x_2}\left(\sum_{x_1}\left(P(x_1)P(x_2|x_1)\right)P(x_3|x_2)\right)\sum_{x_4}P(x_4|x_3)$$

$$\sum_{x_1}\left(P(x_1)P(x_2|x_1)\right) = \begin{matrix}P(x_1=1)P(x_2=1|x_1=1) & P(x_1=1)P(x_2=0|x_1=1)\\ P(x_1=0)P(x_2=1|x_1=0) & P(x_1=0)P(x_2=0|x_1=0)\end{matrix}$$

$$= 0.7\times0.4 + 0.3\times0.5 \quad 0.7\times0.6 + 0.3\times0.5 = 0.43 \quad 0.57$$

This can be also computed as:

$$\sum_{x_1}\left(P(x_1)P(x_2|x_1)\right) = [0.7 \quad 0.3]\begin{bmatrix}0.4 & 0.6\\ 0.5 & 0.5\end{bmatrix} = [0.43 \quad 0.57]$$

Now,

$$\sum_{x_2}\left(\sum_{x_1}\left(P(x_1)P(x_2|x_1)\right)P(x_3|x_2)\right) = [0.7 \quad 0.3]\begin{bmatrix}0.4 & 0.6\\ 0.5 & 0.5\end{bmatrix}\begin{bmatrix}0.8 & 0.2\\ 0.2 & 0.8\end{bmatrix}$$

$$= [0.458 \quad 0.542]$$

Again,

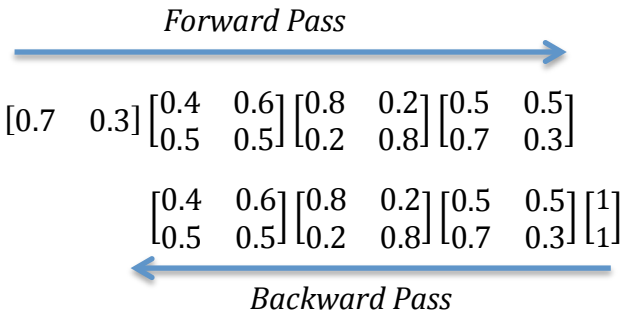
$$\sum_{x_4}P(x_4|x_3) = \begin{bmatrix}0.5 & 0.5\\ 0.7 & 0.3\end{bmatrix}\begin{bmatrix}1\\ 1\end{bmatrix} = \begin{bmatrix}1\\ 1\end{bmatrix}$$

Therefore by combining forward and backward messages we get,

[1] Notes from Advanced Computer Vision by Michael Collins

$$\boldsymbol{P}(x_3) = [0.458(1) \quad 0.542(1)] = [0.458 \quad 0.542] = [P(x_3 = 1) \quad P(x_3 = 0)]$$

The way we have implemented *Belief Propagation* is using one cascaded forward pass and one cascaded backward pass along the graph, and then combining results and normalizing at each node.



Forward Pass	[0.7 0.3]	[0.43 0.57]	[0.458 0.542]	[0.6084 0.3916]
Backward Pass	[1 1]	[1 1]	[1 1]	[1 1]
Combined + Normalized	[0.7 0.3]	[0.43 0.57]	[0.458 0.542]	[0.6084 0.3916]

Therefore storing (`combined + normalized`) in an array we can get marginal probabilities of any node.

E.g. We want to know $\boldsymbol{P}(x_3=1)$:

Let,

`marginalProbabilities = [[0.7 0.3] [0.43 0.57] [0.458 0.542] [0.6084 0.3916]]`

Then,

$\boldsymbol{P}(x_3=1)$ = `marginalProbabilities[2][0]`

In our case, the backward pass is not needed as we are summing over rows and this always produces [1 1]. Consequently forward pass can give us the results directly.

Tree

In case of tree, we have done one forward pass from the root node (x_1) to the all nodes *(leaf nodes)* in the last layer, then we have done one backward pass from all nodes in the last layer *(leave nodes)* to the root node (x_1) and then combined messages from all passes for every node just like we shown above *(in Chain)* to get marginal probabilities of each node.

In our case, the backward pass is not needed as we are summing over rows and always produces $[1 \quad 1]$. Consequently forward pass can give us the results directly.

The same method applies for both Smaller Tree and Larger Tree.

Grid

This method is not applicable to grids.

1.3 Directed Sampling

To obtain samples from a joint distribution that is represented by a Bayesian Network, we make a pass from parent to child nodes and in the process sampling from the conditional distribution $\boldsymbol{P}(x|pa_i)$. After one pass through the graph we obtain one sample.

Chain

We are given that $\boldsymbol{P}(x_1 = 1) = 0.95$ that means there is a 95% chance of getting 1 and 5% chance of getting 0. So, we generate a random number, which is uniformly distributed between 0 and 1 and if that number is ≤ 0.95, we return 1 and else we return 0.

Hence, we have a sample from node x_1. Now, we use this sample to obtain a sample from node x_2 based on the transition probabilities we are given.

The code is:

```
# Probabilities
p1     = np.array(   [0.95, 0.05]                 )
pn_n_1 = np.array( [ [0.95, 0.05], [0.95, 0.05] ] )

def sampleFrom(p, sampleX = None):
    '''
    This function generates a sample based on the transition probabilities at
each of the chain
    @param p       : transition probabilities
    @param sampleX : sample generated at previous step of a chain
    @return        : a new sample using the previous sample and transition
probabilities
    '''
    sampleUniform = np.random.uniform( 0.0, 1.0, size = 1 )
    if sampleX == None:
        if sampleUniform < p[0]:
            return 1
        else:
            return 0
    else:
        if sampleUniform < p[0]:
            return sampleX
        else:
            return 1-sampleX
```

We do this, until we reach the last node of the chain. The *samples* we obtained at each step of the graph are saved in a `numpy` array called `configuration`. In this way we obtain 10^5 `configurations`.

Code is:

```
# Probabilities
p1     = np.array(   [0.95, 0.05]                 )
pn_n_1 = np.array( [ [0.95, 0.05], [0.95, 0.05] ] )

def genOneSample(n):
```

```
    '''Generate ONE sample and the CONFIGURATION (that led to that sample) from
the graph
    @param  n : number of nodes in graph
    @return x : returns a sample which is coming out of the graph
    @return   : returns the configuration of various nodes that resulted in x
    '''
    configuration = np.empty(n)
    for i in xrange(n):
        if i==0:
            x = sampleFrom(p1, None)
        else:
            x = sampleFrom(pn_n_1[x], x)
        configuration[i] = x
    return x, configuration

def genXSamples(nSamples, n):
    '''Generates user desired Number of Samples and the configuration that
generated those Samples
    @param nSamples : no of Samples to be generated
    @param n        : Chain Length
    @return         : desired number of Samples
    @return         : configurations that  created those Samples
    '''
    configurations = []         # Storage for configurations
    samples = np.empty(nSamples)
    for i in xrange(nSamples):
        samples[i], configuration = genOneSample(n)
        configurations.append(configuration)
    return samples, np.array(configurations)
```

E.g. we want to know $\boldsymbol{P}(x_5 = 1|x_1 = 1)$, we count the number of configurations that has (1 in 5th position and 1 in 1st position) and separately count the number of configurations that has (1 in 1st position) and then divide the former by the later to normalize. The code is:

```
def printProbabilities(configurations, nSamples):
    '''This function calculates the required probabilities as asked
    @param configurations : list of configurations that resulted from Sampling
    @nSamples             : number of Samples generated
    '''
    countSamples1    = 0.0
    countSamplesx1_1 = 0.0

    for configuration in configurations:
        if ( configuration[4] == 1 and configuration[0] == 1 ):
            countSamples1    += 1
        if ( configuration[0] == 1 ):
            countSamplesx1_1 += 1

    print( "p( x5=1 | x1=1 )  : {}".format(countSamples2/countSamplesx1_1) )
```

Tree *(Both Small and Large Tree)*

To simulate a tree, we create a similar dictionary as in Brute Force method where keys represent the indices of the node and values represent that value that node is taking. Initially, dictionary is initialized with zeros.

The code is:

```
def createTree(n):
    '''This function creates a dictionary where keys represent the indices and
    values represent the value at that index i.e. 0/1
    @param n : node of nodes in tree
    @return  : {node1: 0, node2: 0, node3: 0, ..., nodeN:0}
    '''
    tree = {x+1: 0 for x in xrange(n)}
    return tree
```

We are given $\boldsymbol{P}(x_1 = 1) = 0.95$ that means there is a 95% chance of getting 1 and 5% chance of getting 0. So, we generate a random number, which is uniformly distributed between 0 and 1 and if that number is ≤ 0.95, we return 1 and else we return 0.

Hence, we have a sample from node x_1. Now, we use this sample to obtain a sample from node x_2 based on the transition probabilities we are given.

The Code is:

```
# Probabilities
p1   = np.array(   [0.95, 0.05]                 )
pn_2 = np.array( [ [0.95, 0.05], [0.95, 0.05] ] )

def sampleFrom(p, sampleX = None):
    '''
    This function generates a sample based on the transition probabilities at
each of the chain
    @param p       : transition probabilities
    @param sampleX : sample generated at previous step of a chain
    @return        : a new sample using the previous sample and transition
probabilities
    '''
    sampleUniform = np.random.uniform( 0.0, 1.0, size = 1 )
    if sampleX == None:
        if sampleUniform < p[0]:
            return 1
        else:
            return 0
    else:
        if sampleUniform < p[0]:
            return sampleX
        else:
            return 1-sampleX   # Taking Complement
```

We do this, until we reach the last node of the tree. The *samples* we obtained at each step of the graph, we save it in a `numpy` array called `configuration`. We obtain 10^5 `configurations` this way.

The code is:

```
# Probabilities
p1   = np.array( [0.95, 0.05]                  )
pn_2 = np.array( [ [0.95, 0.05], [0.95, 0.05] ] )


def genOneSample(n, tree):
    '''This function Generates a sample from the TREE
    @param  n : no of Nodes in tree. e.g.n=15
    @return x : returns a sample that is represented by the joint Distribution
    @return   : returns the configuration (or samples) of various nodes that
resulted in x
    '''
    configuration = np.empty(n)
    for i in xrange(1, n+1):
        if i==1:
            x = sampleFrom( p1, None )
        else:
            x = sampleFrom( pn_2[x], tree[i/2] )
        tree[i]            = x
        configuration[i-1] = x
    return x, configuration

def genXSamples(nSamples, n, tree):
    '''This function Generates user desired Number of Samples and the
configuration that generated those Samples
    @param nSamples : no of Samples to be generated
    @param n        : no of Nodes
    @return         : desired number of Samples
    @return         : configurations that  created those Samples'''
    configurations = []            # Storage for configurations
    samples = np.empty(nSamples)
    for i in xrange(nSamples):
        samples[i], configuration = genOneSample(n, tree)
        configurations.append(configuration)
    return samples, np.array(configurations)
```

E.g. we want to know $\boldsymbol{P}(x_5 = 1|x_1 = 1)$, we count the number of configurations that has (1 in 5th position and 1 in 1st position) and separately count the number of configurations that has (1 in 1st position)then divide the former by the later.

The code is:

```
def printProbabilities(configurations, nSamples):
    '''This function calculates the required probabilities as asked
    @param configurations : list of configurations that resulted from Sampling
    @nSamples             : number of Samples generated
    '''
    countSamples1    = 0.0
    countSamplesx1_1 = 0.0

    for configuration in configurations:
        if ( configuration[4] == 1 and configuration[0] == 1 ):
            countSamples1    += 1
        if ( configuration[0] == 1 ):
            countSamplesx1_1 += 1
print( "p( x5=1 | x1=1 )  : {}".format(countSamples2/countSamplesx1_1) )
```

Grid *(Both Small and Large Grid)*

To simulate a grid, we apply a similar pattern and create a dictionary where the keys represent the indices of a node and values represent that value that node is taking. Initially, when creating the dictionary, we initialize values to 0.

The code is:

```
def createGrid(n):
    '''This function creates a dictionary where keys represent the indices and
    values represent the value at that index i.e. 0/1
    @param n : node of nodes in tree
    @return  : {node1: 0, node2: 0, node3: 0, ..., nodeN:0}
    '''
    grid = {x+1: 0 for x in xrange(n)}
    return grid
```

We are given that $\boldsymbol{P}(x_1 = 1) = 0.5$ that means there is a 50% chance of getting 1 and 50% chance of getting 0. So, we generate a random number, which is uniformly distributed between 0 and 1 and if that number is ≤ 0.50, we return 1 and else we return 0.

Hence, we have a sample from node x_1. Now, we use this sample to obtain a sample from node x_2 based on the transition probabilities we are given.

The Code is:

```
# Probabilities
p1   = np.array(   [0.5,  0.5 ]                                          )
pn_2 = np.array( [ [0.95, 0.05], [0.95, 0.05]                           ] )
pn_3 = np.array( [ [0.01, 0.99], [0.5,  0.5 ], [0.5, 0.5], [0.99, 0.01] ] )

def sampleFrom(p, sampleX = None):
    '''
    This function generates a sample based on the transition probabilities at
each of the chain
    @param p       : transition probabilities
    @param sampleX : sample generated at previous step of a chain
    @return        : a new sample using the previous sample and transition
probabilities
    '''
    sampleUniform = np.random.uniform( 0.0, 1.0, size = 1 )
    if sampleX == None:
        if sampleUniform < p[0]:
            return 1
        else:
            return 0
    else:
        if sampleUniform < p[0]:
            return sampleX
        else:
            return 1-sampleX   # Taking Complement
```

We do this, until we reach the last node of the grid. The *samples* we obtained at each step of the graph are saved in a `numpy` array called `configuration`. We obtain 10^5 `configurations` this way.

The code is:

```
# Probabilities
p1   = np.array(   [0.5,  0.5 ]                                          )
pn_2 = np.array( [ [0.95, 0.05], [0.95, 0.05]                           ] )
pn_3 = np.array( [ [0.01, 0.99], [0.5,  0.5 ], [0.5, 0.5], [0.99, 0.01] ] )

def genOneSample(n, L, grid):
    '''This function Generates ONE sample from the GRID represented by the joint
distribution
    @param  n : no of Nodes
    @param  L : no of Layers
    @return x : returns a sample which is coming out of the GRID
    @return   : returns the configuration of various nodes that resulted in x
    '''
    configuration = np.empty(n)
    for i in xrange(1, n+1):   # i means index
        if i==1:               # First position has nothing pointing to it
            x = sampleFrom( p1, None )
        elif (i <= L):         # Nodes in first column of grid
            x = sampleFrom( pn_2[x], grid[i-1] )
        elif ( (i-1)%L == 0 ): # Nodes in first row of grid
            x = sampleFrom( pn_2[x], grid[i-L] )
        else:                  # Rest all grids
            xj = grid[i-1]     # Ancestor Neighbour Node with index j
            xk = grid[i-L]     # Ancestor Neighbour Node with index k
            if   ( xj == 0 ) and ( xk == 0):
                index = 0
            elif ( xj == 0 ) and ( xk == 1):
                index = 1
            elif ( xj == 1 ) and ( xk == 0):
                index = 2
            else:
                index = 3
            x = sampleFrom(pn_3[index], None)
        grid[i] = x
        configuration[i-1] = x
    return x, configuration


def genXSamples(nSamples, n, L, grid):
    '''Generates user desired Number of Samples and the configuration that
generated those Samples
    @param nSamples : no of Samples to be generated
    @param n        : Chain Length
    @return         : desired number of Samples
    @return         : configurations that  created those Samples'''
    configurations = []   # Storage for configurations
    samples = np.empty(nSamples)
    for i in xrange(nSamples):
        samples[i], configuration = genOneSample(n, L, grid)
        configurations.append(configuration)
    return samples, np.array(configurations)
```

E.g. we want to know $\boldsymbol{P}(x_5 = 1|x_1 = 1)$, we count the number of configurations that have (1 in 5th position and 1 in 1st position) and separately count the number of configurations that have (1 in 1st position) and then divide the former by the later to normalize.

The code is:

```
def printProbabilities(configurations, nSamples):
    '''This function calculates the required probabilities as asked
    @param configurations : list of configurations that resulted from Sampling
    @nSamples             : number of Samples generated
```

```
'''
countSamples1    = 0.0
countSamplesx1_1 = 0.0

for configuration in configurations:
    if ( configuration[4] == 1 and configuration[0] == 1 ):
        countSamples1    += 1
    if ( configuration[0] == 1 ):
        countSamplesx1_1 += 1

print( "p( x5=1 | x1=1 )  : {}".format(countSamples2/countSamplesx1_1) )
```

2. Issues

2.1 Brute Force

Brute Force computation does not scale due to a combinatorial explosion of possible states. This problem occurs when a small increase in the number of elements causes at least an exponential increase in the state space.

Complexity of iterating over all possible states in a fully connected graph (brute force solution) is $\Theta(n!)$. Reducing the number of possible connections can simplify the problem, but not nearly enough to make it tractable in polynomial time algorithm. Because the problem has an exponential complexity in the number of states it becomes intractable as we increase the number of states.

3. Results

3.1 Chain

Number of nodes : 15

Time Required

Brute Force : 0.7813 secs
Belief Propagation : 0.0017 secs
Directed Sampling : 7.1535 secs

	$P(x_5=1)$	**$P(x_5=1\|x_1=1)$**	**$P(x_5=1\|x_1=1, x_{10}=1)$**	**$P(x_5=1\|x_1=1, x_{10}=1, x_{15}=0)$**
Brute Force	0.7952	0.8280	0.94924	0.9492
Belief Propagation	0.7952	NA	NA	NA
Directed Sampling	0.7963	0.8273	0.9487	0.9499
	0.7940	0.8276	0.9499	0.9502
	0.7955	0.8281	0.9505	0.9554
	0.7961	0.8282	0.9487	0.9499
	0.7955	0.8287	0.9479	0.9480

3.2 Small Tree

Number of Layers : 4
Number of Nodes : 15

Time Required

Brute Force	: 0.7965	secs
Belief Propagation	: 0.0020	secs
Directed Sampling	: 7.4473	secs

	$P(x_8=1)$	**$P(x_8=1\|x_{12}=1)$**	**$P(x_8=1\|x_{12}=1, x_7=1)$**	**$P(x_8=1\|x_{12}=1, x_7=1, x_{15}=0)$**
Brute Force	0.82805	0.85853	0.86227	0.86227
Belief Propagation	0.82805	NA	NA	NA
Directed Sampling	0.82593	0.85944	0.86292	0.86385
	0.82905	0.85864	0.86212	0.86341
	0.82928	0.85952	0.86345	0.86034
	0.82859	0.85913	0.86307	0.86848
	0.82736	0.85869	0.86277	0.87180

3.3 Large Tree

Number of Layers : 6
Number of Nodes : 63

Time Required

Brute Force : ∞ secs
Belief Propagation : 0.0017 secs
Directed Sampling : 29.213 secs

	$P(x_{32}=1)$	**$P(x_{32}=1\|x_{45}=1)$**	**$P(x_{32}=1\|x_{45}=1, x_{31}=1)$**	**$P(x_{32}=1\|x_{45}=1, x_{31}=1, x_{63}=0)$**
Brute Force	X	X	X	X
Belief Propagation	0.7657205	NA	NA	NA
Directed Sampling	0.76717	0.81438	0.81940	0.83187
	0.76573	0.81350	0.81920	0.81504
	0.76732	0.81534	0.82117	0.81608
	0.76573	0.81450	0.82019	0.81668
	0.76199	0.81023	0.81530	0.818181

3.4 Small Grid

Number of Layers : 4
Number of Nodes : 16

Time Required
Brute Force : 6.4712 secs
Directed Sampling : 8.2362 secs

	$P(x_6=1)$	**$P(x_6=1\|x_{16}=0)$**	**$P(x_6=1\|x_{16}=0, x_1=0)$**	**$P(x_6=1\|x_{16}=0, x_1=0, x_{15}=0)$**
Brute Force	0.5	0.10449	0.02383	0.01853
Belief Propagation	NA	NA	NA	NA
Directed Sampling	0.50103	0.10659	0.02413	0.01912
	0.49854	0.10374	0.02325	0.01828
	0.49751	0.10472	0.02429	0.01935
	0.50128	0.10639	0.02452	0.01846
	0.49790	0.10296	0.02425	0.01921

3.5 Large Grid

Number of Layers : 8
Number of Nodes : 64

Time Required

Brute Force : ∞ secs
Directed Sampling : 32.27 secs

	$P(x_6=1)$	**$P(x_6=1\|x_{64}=0)$**	**$P(x_6=1\|x_{64}=0, x_1=0)$**	**$P(x_5=1\|x_{57}=\ldots=x_{64}=0)$**
Brute Force	X	X	X	X
Belief Propagation	NA	NA	NA	NA
Directed Sampling	0.49795	0.25306	0.15864	0.21671
	0.50107	0.25113	0.15594	0.21556
	0.50333	0.25632	0.16133	0.22064
	0.49947	0.25194	0.15942	0.21631
	0.49925	0.25265	0.16094	0.21869