My programs are developed on python3.4+. They rely on numpy and scipy library.

Problem 1.1. *For brute force version, please refers to "p1_brute_force.py". You can run the code with "python3 p1_brute_force.py" under the project directory. The output result should be straight forward.*
As for sum-product algorithm, please refer to "p1_fg.py". You can run the code with "python3 p1_fg.py" under the project directory. The console output should also be straight forward. For clarification, I am doing a bit extra for this problem. First the parser reads in bayesian network from "p1.cif". Then it moralizes the DAG to a Markov network. With Markov network, we can build a corresponding Beth cluster graph. Finally, the program does inferences with belief propagation algorithm from textbook. You can see the result matches the brute-force version.

Problem 1.2. *You can run the code with "python3 p1.py". Main part of the implementation is in "bn.py". The algorithm basically follows "A Computational Model for Causal and Diagnostic Reasoning in Inference Systems" mentioned in homework. The inference result is:*

$KINKEDTUBE$	$\mathbb{P}(KINKEDTUBE)$
$FALSE$	0.95909208
$TRUE$	0.04090792

Problem 1.3. *We can moralize the graph, and use max-chordality algorithm mentioned in the book to check whether the graph is chordal or not. If max-chordality can finish without adding any fill-in edge, this means the graph is chordal without triangulation. Since for each chordal graph, there exists a corresponding clique tree. We can build a clique tree from the moralized graph. Because the generated graph is a clique tree, there is no loopy belief propagation involved. Belief-propagation will converge.*

Algorithm 1: IsConverge

Input : Node N, graph G and evidence Z
Output: True if belief propagation converge, False otherwise
$G' \leftarrow Morlize(G)$ //Morlize the graph G ;
$G'' \leftarrow IncludeEvidence(G, Z)$ //Include evidences by adjusting Markov network's factors and edges ;
Run max-chordality algorithm to check if G'' is chordal;
Return True is max-chordality algorithm cannot add any fill-in edge;
Return False otherwise;

Problem 2.1. *You can refers to "p2_ic_100_0.py", "p2_ic_100_0_005.py", "p2_ic_100_0_2.py", "p2_ic_1000_0.py", "p2_ic_1000_0_005.py", "p2_ic_1000_0_2.py" for implementation. The code can be executed using command "python3 filename". The suffix for each file stands for different parameters. For example "p2_ic_100_0.py" means the program runs with $n = 100, p = 0$. The console output should be straight forward. Generally speaking, when $n = 100$, because of small sample size, there usually is no edge in learned structures.*

1. *When $n = 1000, p = 0$, there is no edge in learned structure, because $p = 0$ makes $x_k = y_k k = 1, 2, 3, 4, 5$. This probability property makes each x_k and y_k dependent.*
2. *When $n = 1000, p = 0.05$, the learned structure is usually like this:*

```
Directed : x0 x1
Undirected : x0 y1
Directed : y0 x1
Directed : x2 x3
Undirected : x2 y1
Directed : x2 y3
Directed : x3 x4
Directed : y2 x3
Directed : x3 y4
Directed : x5 x4
```

```
Directed : x4 y3
Directed : x4 y5
Directed : y4 x5
Directed : y3 y2
Directed : y3 y4
Directed : y4 y5
```

3. *When $n = 1000, p = 0.2$, the learned structure is usually like this:*

```
Directed : x0 x1
Directed : x0 y1
Directed : x1 x2
Directed : y2 x1
Directed : y1 x2
Directed : x2 y3
Directed : x3 x4
Undirected : x3 y2
Directed : x3 y4
Directed : y3 x4
Directed : x4 y5
Directed : y4 x5
Directed : y0 y1
Directed : y2 y3
Directed : y3 y4
Directed : y4 y5
```

Problem 2.2. *You can refers to "p2_ics_100_0.py", "p2_ics_100_0_005.py", "p2_ics_100_0_2.py", "p2_ics_1000_0.py", "p2_ics_1000_0_005.py", "p2_ics_1000_0_2.py" for implementation. The code can be executed using command "python3 filename". The suffix for each file stands for different parameters. For example "p2_ics_100_0.py" means the program runs with $n = 100, p = 0$. The console output should be straight forward. Generally speaking, when $n = 100$, because of small sample size, there usually is no edge in learned structures.*

1. *When $n = 1000, p = 0$, there is no edge in learned structure, because $p = 0$ makes $x_k = y_k k = 1, 2, 3, 4, 5$. This probability property makes each x_k and y_k dependent. Then, for any pair $x_i, y_j, i \neq j, i, j = 1, 2, 3, 4, 5$, we can always find a d-separation set y_i or x_j. Therefore, the algorithm won't add any edge.*

2. *When $n = 1000, p = 0.05$, the learned structure is usually like this:*

```
DirectedStar : x0 x1
DirectedStar : y1 x0
DirectedStar : x1 x2
DirectedStar : y2 x1
DirectedStar : x2 x3
DirectedStar : y3 x2
DirectedStar : x3 x4
DirectedStar : x3 y2
DirectedStar : y4 x3
DirectedStar : x4 x5
DirectedStar : y3 x4
DirectedStar : x4 y5
DirectedStar : y4 x5
DirectedStar : y1 y0
```

```
DirectedStar : y1 y2
DirectedStar : y4 y3
```

3. *When $n = 1000, p = 0.2$, the learned structure is usually like this:*

```
DirectedStar : x1 x0
Undirected : x0 y1
DirectedStar : x1 x2
DirectedStar : x1 y0
DirectedStar : x1 y2
DirectedStar : x2 x3
DirectedStar : y3 x2
DirectedStar : x3 x4
DirectedStar : x3 y2
DirectedStar : x3 y4
DirectedStar : x4 x5
DirectedStar : y3 x4
DirectedStar : x4 y5
DirectedStar : y4 x5
DirectedStar : y4 y3
DirectedStar : y4 y5
```

Problem 2.3. *The structure is explained in previous problem. The reason for the difference is that the introduction of latent variables e_k, e'_k make x_k, y_k less likely to be dependent. It disrupts the deterministic relations between x_k, y_k. This makes the algorithm*

Problem 2.4. *The structure learned under this setting doesn't have any edge. There are two reasons: First, with $p = 0$, $x_k = y_k = x_{k+1} = y_{k+1}, k = 1, 2, 3, 4$. This is a deterministic relation. Therefore, pair x_k, y_k are dependent. Then, for any pair $x_i, y_j, i \neq j, i, j = 1, 2, 3, 4, 5$, we can always find a d-separation set y_i or x_j. Therefore, the algorithm won't add any edge.Second, because I am using fisher test as independence test, when $n = 100$, which is a small sample size, nodes are more likely to be independent.*