Homework 4

Thibault Doutre, Student ID 26980469

CS281A/STAT241A : Statistical Learning Theory
University of California, Berkeley

1 IPF

1.1 Preliminaries

First, I set the working directory and the global variables, including the three graphs A, B and C. The undirected graphs are uniquely represented by the adjacence matrix which is an upper triangular matrix here. I also store the edges in a list for computational efficiency.

```
# Load Data
setwd("~/Documents/STAT241/hw4")
data=read.table("./hw4data.data")
head(data)

##   V1 V2 V3 V4 V5 V6 V7
## 1  0  0  0  0  0  1  0
## 2  0  1  0  1  0  0  0
## 3  0  0  0  0  0  1  0
## 4  0  1  0  0  1  0  0
## 5  1  1  0  1  1  0  1
## 6  1  1  1  1  1  0  1

# Global variables
d=7
N=500
V=1:d

# Graph A
Ga_adj=matrix(c(0,0,0,0,1,1,0,
                0,0,0,0,1,0,0,
                0,0,0,1,0,1,0,
                0,0,0,0,1,0,1,
                0,0,0,0,0,0,0,
                0,0,0,0,0,0,1,
                0,0,0,0,0,0,0),d,byrow=T)
```

```
Ga_edges=list(c(1,5),c(1,6),c(2,5),c(3,4),c(3,6),c(4,5),c(4,7),c(6,7))
# Graph B
Gb_adj=matrix(c(0,1,1,1,0,0,1,
                0,0,0,1,0,1,0,
                0,0,0,1,1,0,1,
                0,0,0,0,0,1,0,
                0,0,0,0,0,0,1,
                0,0,0,0,0,0,0,
                0,0,0,0,0,0,0),d,byrow=T)
Gb_edges=list(c(1,2),c(1,3),c(1,4),c(1,7),c(2,4),c(2,6),c(3,4),c(3,5),
              c(3,7),c(4,6),c(5,7))
# Graph C
Gc_adj=matrix(c(0,1,1,1,1,1,1,
                0,0,1,1,1,1,1,
                0,0,0,1,1,1,1,
                0,0,0,0,1,1,1,
                0,0,0,0,0,1,1,
                0,0,0,0,0,0,1,
                0,0,0,0,0,0,0),d,byrow=T)
Gc_edges=list(c(1,2),c(1,3),c(1,4),c(1,5),c(1,6),c(1,7),
              c(2,3),c(2,4),c(2,5),c(2,6),c(2,7),
              c(3,4),c(3,5),c(3,6),c(3,7),c(4,5),c(4,6),
              c(4,7),c(5,6),c(5,7),c(6,7))
```

The potentials are initialized by assigning the value 1 to each edge. The potentials are stored in a d*d*2*2 array.

```
# Initialize potentials
init_phi=function(G_adj){
  phi0=array(dim=c(d,d,2,2))
  phi0[,,1,1]=G_adj
  phi0[,,1,2]=G_adj
  phi0[,,2,1]=G_adj
  phi0[,,2,2]=G_adj
  return(phi0)
}
```

I also create a function which displays every possible combination of zero and ones in a vector, given a length n. This function will be usefull to compute all the possible configurations of the observations.

```
# Possibilities for V
comb = function(n){
  if (n==1)
    return(matrix(c(0,1),2,1))
  else{
    old=comb(n-1)
    add=matrix(NA,2^n,1)
    return(rbind(cbind(matrix(1,2^(n-1),1),old),
                 cbind(matrix(0,2^(n-1),1),old)))
  }
}
# Example
comb(2)


##      [,1] [,2]
## [1,]    1    0
## [2,]    1    1
## [3,]    0    0
## [4,]    0    1
```

1.2 Joint distribution

I compute first the non-normalized joint distribution and then, using the comb function I normalize.

```
# Joint distribution, non normalized
p0 = function(v,G_edges,phi){
  # v is a d-dimensional vector
  # corresponding to a configuration
  res=1
  for (edge in G_edges){
    i=edge[1]
    j=edge[2]
    res=res*phi[i,j,v[i]+1,v[j]+1]
  }
  return(res)
}
# Normalizing factor
Z = function(G_edges,phi){
  sum(apply(comb(d),1,function(v) p0(v,G_edges,phi)))
}

# Normalized joint distribution
p = function(v,G_edges,phi){
  p0(v,G_edges,phi)/Z(G_edges,phi)
}
```

1.3 Marginals

Then, I compute the marginals.

```
# Marginal probability
pC = function(G_edges,phi,C){
  # C=NA,1,NA,0,NA,NA,NA..
  a=which(!is.na(C))[1]
  b=which(!is.na(C))[2]
  m=comb(d)[comb(d)[,a]==C[a], ]
  M=m[m[,b]==C[b], ]
  sum(apply(M,1,function(x) p(x,G_edges,phi)))
}

# Marginal counts
m = function(v,data){
  # v is a d-dimensional vector with NAs
  # indicating which variables to sum
  # in order to have the marginal
  sum(apply(data,1,function(l) all(l==v,na.rm=T)))
}
```

1.4 IPF Algorithm

With these functions, I can now compute the IPF Algorithm. I use a L1 norm over the d*d*2*2 phi array for the stopping criteria. The function returns the updated array phi, the difference between the last two iterations, accoring to the L1 norm and the number of iterations. In the while loop, I print the L1 norm of the difference at each iteration in order to see the convergence of the algorithm.

```
# IPF
IPF = function(phi0,data,G_edges,eps){
  phi=phi0
  old_phi=phi+2*eps
  it=0
  while (max(abs((old_phi-phi))) > eps){
    old_phi=phi
    for (edge in G_edges){
      a=edge[1]
      b=edge[2]
      for (c in list(c(0,0),c(1,0),c(0,1),c(1,1))){
        v1=c[1]+1
        v2=c[2]+1
        C=rep(NA,d)
        C[a]=c[1]
        C[b]=c[2]
        phi[a,b,v1,v2]=phi[a,b,v1,v2]*m(C,data)/(N*pC(G_edges,phi,C))
      }
    }
    it=+1
    print(max(abs((old_phi-phi))))
  }
  return(list(phi=phi,diff=max(abs((old_phi-phi))),iteration=it))
}
# Compute IPF for the three graphs
# A
phi0a=init_phi(Ga_adj)
IPF_a=IPF(phi0a,data,Ga_edges,0.01)

## [1] 0.67168
## [1] 0.0635103
## [1] 0.005453936

#B
phi0b=init_phi(Gb_adj)
IPF_b=IPF(phi0b,data,Gb_edges,0.01)

## [1] 0.8952535
## [1] 0.270559
## [1] 0.1413328
## [1] 0.0834352
## [1] 0.05580195
## [1] 0.03923967
## [1] 0.03068312
## [1] 0.02629259
## [1] 0.02221051
## [1] 0.0185679
## [1] 0.01540343
## [1] 0.0127044
## [1] 0.01043218
## [1] 0.008537358

# Fully connected graph
phi0c=init_phi(Gc_adj)
IPF_c=IPF(phi0c,data,Gc_edges,0.01)

## [1] 0.8933857
## [1] 0.1576448
## [1] 0.1148661
## [1] 0.1011127
## [1] 0.08918518
## [1] 0.07907773
## [1] 0.0705046
## [1] 0.06315459
## [1] 0.05677672
## [1] 0.05117988
## [1] 0.04622267
## [1] 0.04180066
## [1] 0.03783549
## [1] 0.03426668
## [1] 0.0310462
## [1] 0.02813473
## [1] 0.02549931
## [1] 0.02311171
## [1] 0.02094739
## [1] 0.01898474
## [1] 0.01720457
## [1] 0.01558973
## [1] 0.01412481
## [1] 0.01279591
## [1] 0.01159048
## [1] 0.01049714
## [1] 0.009505589
```

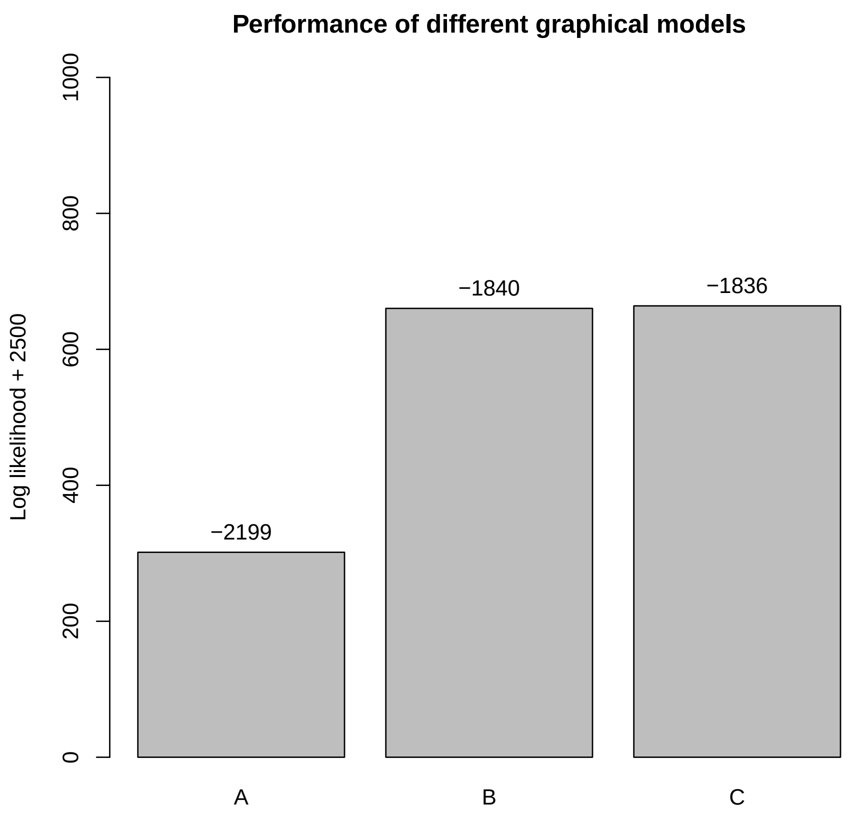
Here are the required results:

```
IPF_a$phi[3,4,,]

##           [,1]      [,2]
## [1,] 0.9089642 0.4137162
## [2,] 0.7890789 1.4239463

IPF_b$phi[3,4,,]

##           [,1]      [,2]
## [1,] 1.0726648 0.6477793
## [2,] 0.8719518 1.1019320

IPF_c$phi[3,4,,]

##           [,1]      [,2]
## [1,] 1.2096976 0.7363805
## [2,] 0.8366489 1.0676378
```

1.5 Likelihood

Instead of computing the likelihood, I choose to compute the log likelihood. I use for this the formula page 220 of the Textbook. The performance displayed on the barplot is based on the opposite of this likelihood which we have maximized with the IPF algorithm.

```r
# Log likelihood
l = function(G_edges,phi){
  res=0
  for (edge in G_edges){
    a=edge[1]
    b=edge[2]
    for (c in list(c(0,0),c(1,0),c(0,1),c(1,1))){
      v1=c[1]+1
      v2=c[2]+1
      C=rep(NA,d)
      C[a]=c[1]
      C[b]=c[2]
      res=res+m(C,data)*log(phi[a,b,v1,v2])
    }
  }
  res-N*log(Z(G_edges,phi))
}


l_a=l(Ga_edges,IPF_a$phi)
l_b=l(Gb_edges,IPF_b$phi)
l_c=l(Gc_edges,IPF_c$phi)
# Log likelihood of our models
cte=2500
barplot(c(l_a,l_b,l_c)+cte,
  ylab= paste('Log likelihood +',cte), names.arg = c('A','B','C'),
  main= 'Performance of different graphical models',ylim=c(0,1000))
text(.7,l_a+cte+30, round(l_a))
text(1.9,l_b+cte+30, round(l_b))
text(3.1,l_c+cte+30, round(l_c))
```



The likelihood is minimized with the fully connected graph (the KL divergence for this graph must be zero). However, this is not necessarily a good idea because the complexity of the model is much greater than the B case which has a very close likelihood to the fully connected graph. Therefore, because of the simplicity of the model and the good overall perfomance of it, I would recommand using the B graph here.

1.6 Mutual Informaion

I first compute the mutual information, given the provided function.

```r
# Mutual Information
I = function(i,j,data){
  res=0
  for (c in list(c(0,0),c(1,0),c(0,1),c(1,1))){
    # set pij
    v=rep(NA,d)
    v[i]=c[1]
    v[j]=c[2]
    pij=m(v,data)/N
    # set pi
    v[j]=NA
    pi=m(v,data)/N
    # set pj
    v[j]=c[2]
    v[i]=NA
    pj=m(v,data)/N
    res=res+pij*log(pij/(pi*pj))
  }
  return(res)
}
```

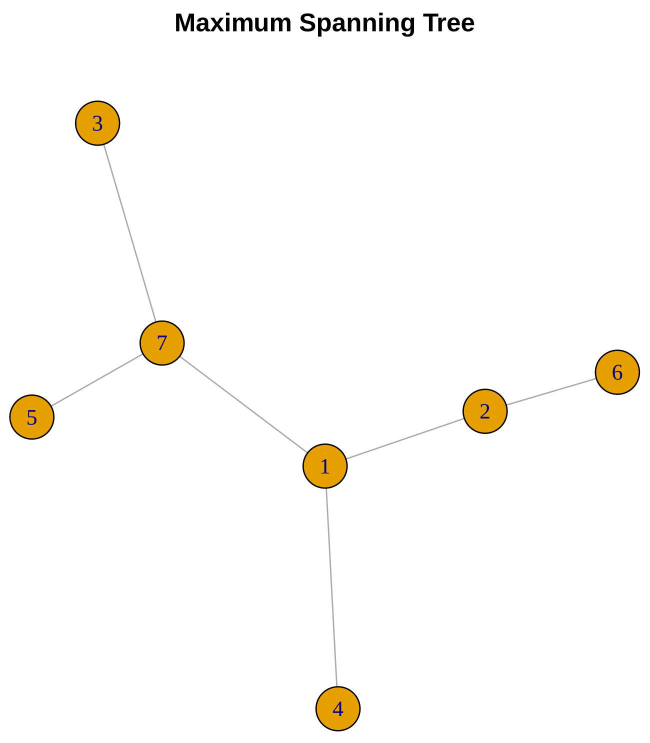
Then, I can initialize a weight matrix. Since there exist efficient algorithm for the minimum spanning tree problem such as Kruskal's or Prim's, I put the negative information in the initial weights and then use Prim's algorithm to find the minimum spanning tree on this matrix. This is equivalent to find the maximum spanning tree for the weighted matrix with I(i,j).

```
# Set the weights equals to the opposite of I(i,j)
init_weights=function(data){
  m=matrix(NA,d,d)
  for (i in 1:d){
    for (j in 1:d){
      m[i,j]=-I(i,j,data)
    }
  }
  return (m)
}
W=init_weights(data)
```

```
# Minimum spanning tree
library(ape)
T_adj=mst(W)

# Plot Graph
library('igraph');

##
## Attaching package: 'igraph'
##
## The following objects are masked from 'package:ape':
```

```
##
##     edges, mst, ring
##
## The following objects are masked from 'package:stats':
##
##     decompose, spectrum
##
## The following object is masked from 'package:base':
##
##     union
```

```
g1<-as.undirected(graph.adjacency(T_adj))
plot(g1,main= "Maximum Spanning Tree")
```



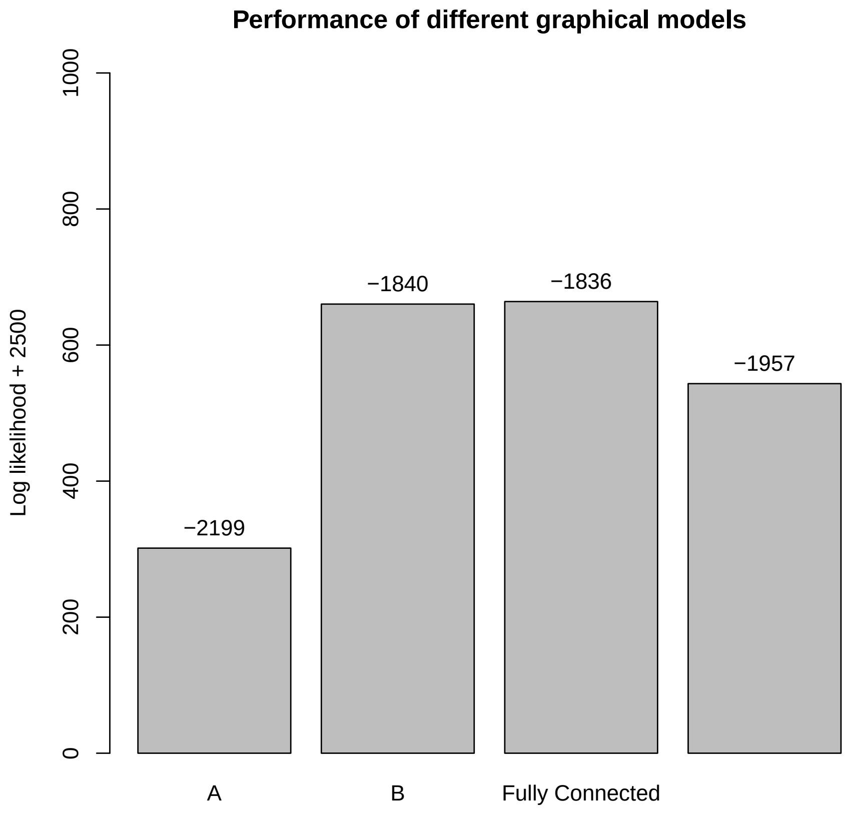
```
# Likelihood of the tree
T_adj=T_adj*upper.tri(T_adj)
T_edges=list(c(1,2),c(1,7),c(2,6),c(3,7),c(5,7))
phi0T=init_phi(T_adj)
IPF_T=IPF(phi0T,data,T_edges,0.001)
```

```
## [1] 0.8940494
## [1] 0.09358389
## [1] 0.01162324
## [1] 0.002174112
## [1] 0.0003678723
```

```
l_T=l(T_edges,IPF_T$phi)
```

Finally, I display a barplot of my work. We can see that the maximum spanning tree might be a good idea since it is very sparse and provide a likelihhod close to the likelihood of the fully connected graph.

```
# Summary
barplot(c(l_a,l_b,l_c,l_T)+cte,
        ylab= paste('Log likelihood +',cte),
        names.arg = c('A','B','Fully Connected','Max spanning tree'),
      main= 'Performance of different graphical models',ylim=c(0,1000))
text(.7,l_a+cte+30, round(l_a))
text(1.9,l_b+cte+30, round(l_b))
text(3.1,l_c+cte+30, round(l_c))
text(4.3,l_T+cte+30, round(l_T))
```


Performance of different graphical models
Log likelihood + 2500
0
200
400
600
800
1000
−2199
−1840
−1836
−1957
A
B
Fully Connected