732A96: Lab 1
Advanced Machine Learning

Carles Sans Fuentes

October 9, 2017

Assignment

Question 1

In this question we are asked to show that multiple runs of the hill-climbing algorithm can return non-equivalent graphs. The hill-climbing algorithm is is a local greedy search optimization technique which iteratively starts with an arbitrary solution to a problem, and then it tries to find a better solution by incrementally changing a single element of the solution, such that if the change produces a better solution, then this new one is taken for the next incremental change until no further improvements are found. Intuitively, it can be understood that different ending outcomes can be found from different initial points if the function is complex enough. In order to prove this, the asia dataset from the bnlearn package in R has been taken. Two different starts has been tried, specifying in one an initial connection to exist and in the second one just totally random. It can be seen in the information and graphs below (table and 1 that the graphs are not equivalent.

```
> mygraph
  Bayesian network learned via Score-based methods
  model:
    [partially directed graph]
  nodes:                                 8
  arcs:                                  7
    undirected arcs:                     2
    directed arcs:                       5
  average markov blanket size:           2.25
  average neighbourhood size:            1.75
  average branching factor:              0.62
  learning algorithm:                    Hill-Climbing
  score:                                 BIC (disc.)
  penalization coefficient:              4.258597
  tests used in the learning procedure:  77
  optimized:                             TRUE
> cmygraph
  Bayesian network learned via Score-based methods model:
    [partially directed graph]
  nodes:                                 8
  arcs:                                  8
    undirected arcs:                     6
    directed arcs:                       2
  average markov blanket size:           2.25
  average neighbourhood size:            2.00
  average branching factor:              0.25
  learning algorithm:                    Hill-Climbing
  score:                                 BIC (disc.)
  penalization coefficient:              4.258597
  tests used in the learning procedure:  82
  optimized:                             TRUE
> all.equal(mygraph, cmygraph)
[1] "Different number of directed/undirected arcs"
```

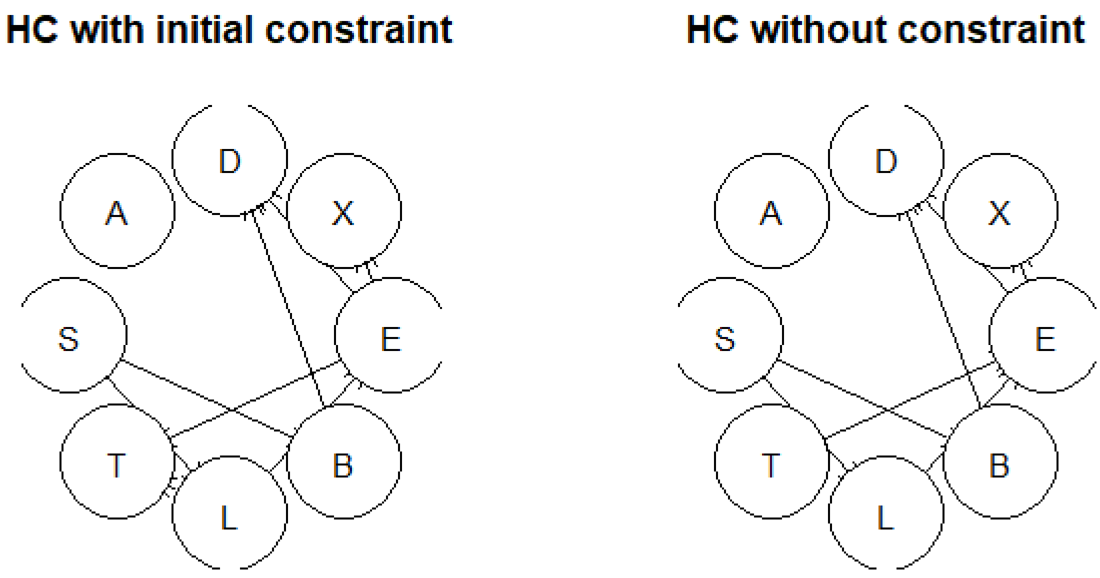

Figure 1: Graphical illustration of HC optimization for two different set ups

ALTERNATIVE SHOWING:

In order to show it in another way, I am going to use the data set alarm. I will be simulating from hill climbing as many times until it creates different graphs. And I will show this last ones graphically. This is what it can be seen in the following graph:

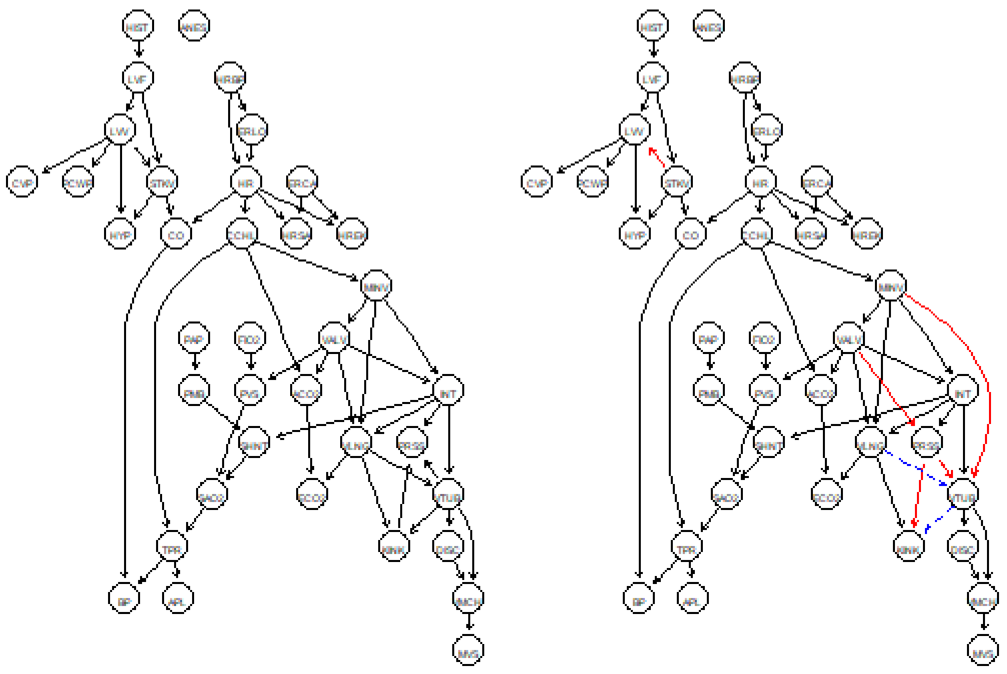

Figure 2: Graphical representation of the alarm data with different hc ending points

As explained previously, the hc() is a greedy algorithm which ends up in a local optima, so it tries to change arcs in order to get a better optima.

Question 2

In figure 3 it can be found the graphical proof that increasing the sample size the BDeu decreases regularization.

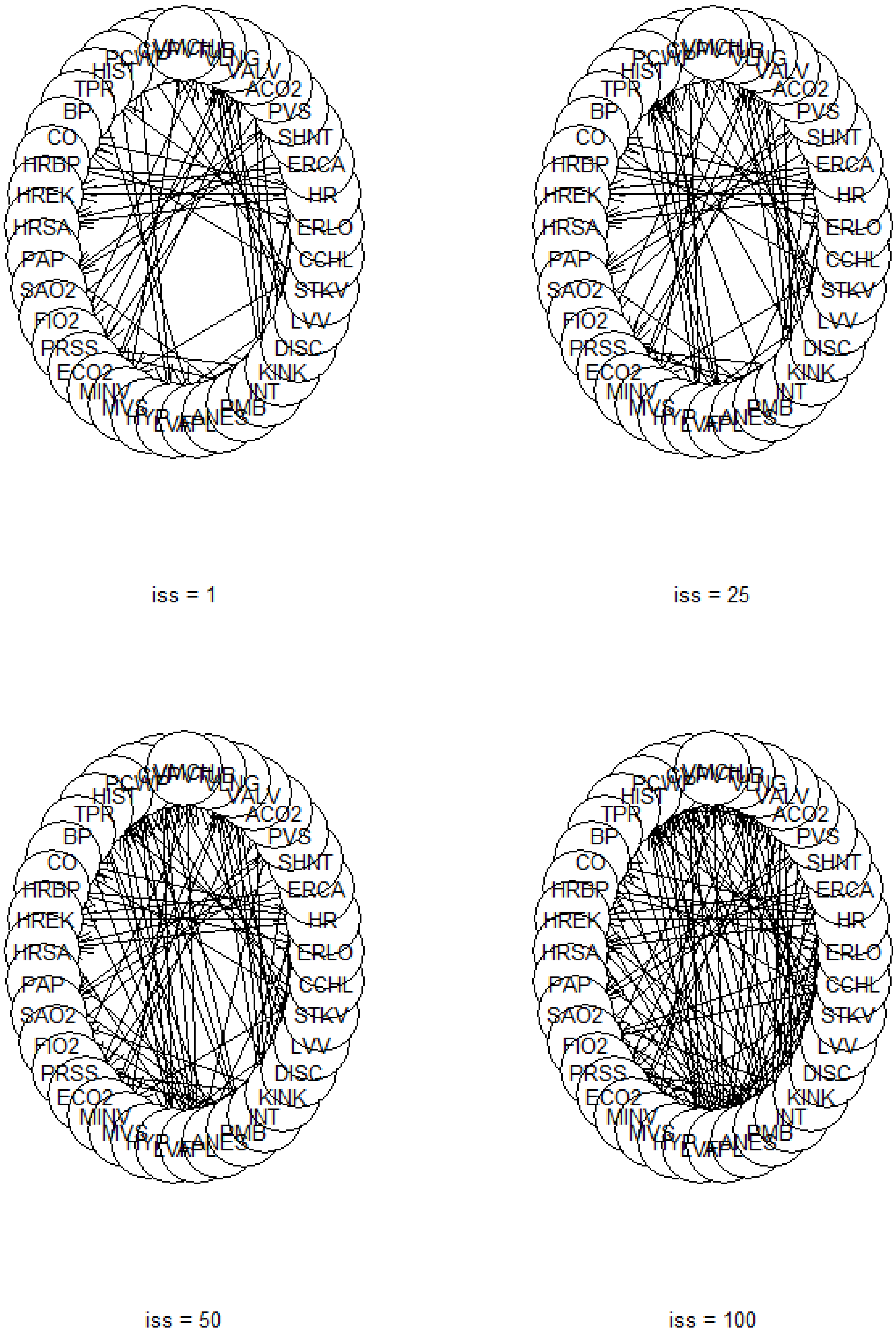

Figure 3: Graphical representation of the alarm data with different iss scores

The alpha.star results are the following ones:

```
[1] 6.779801 #for iss = 1
[1] 7.439901 #for iss = 25
[1] 10.34823 #for iss = 50
[1] 16.62416 #for iss = 100
```

The score got is lower because the more arc a graph has, the more likely to represent a graph from the data but the more complex it is so the more penalized the model is. Also, it is much probable that there will be a lot of parameters (e.g. arcs that do not exist in the real data) which will worsen the result.

Question 3

2nd rewritten EXPLANATION

Now we are asked to compare the answers given some queries by exact and approximate inference algorithms.

The approximate algorithm uses samples from the distribution while it discards the samples that are not consistent with the conditioning set. Now we will iterate 4 times on both algorithm in order to compare how well they perform. For that, I have used the dataset called data.learning and I have compared the LS and the approximate algorithm in order to find the conditional distribution of A, E given $B = b$ and $F = b$. and then further below it can be found the conditional distribution of A, E given $B = b$ and $F = b$, $D = b$ and $C = b$.

```
> res2
[[1]]
           A A_Approx.Freq         E E_Approx.Freq
a 0.07394366    0.06010929 0.3167939     0.3333333
b 0.64964789    0.65391621 0.3664122     0.3679417
c 0.27640845    0.28597450 0.3167939     0.2987250

[[2]]
           A A_Approx.Freq         E E_Approx.Freq
a 0.07394366    0.08791209 0.3167939     0.3150183
b 0.64964789    0.64285714 0.3664122     0.3791209
c 0.27640845    0.26923077 0.3167939     0.3058608

[[3]]
           A A_Approx.Freq         E E_Approx.Freq
a 0.07394366        0.0752 0.3167939        0.2992
b 0.64964789        0.6352 0.3664122        0.3840
c 0.27640845        0.2896 0.3167939        0.3168

[[4]]
           A A_Approx.Freq         E E_Approx.Freq
a 0.07394366    0.08448276 0.3167939     0.2948276
b 0.64964789    0.66724138 0.3664122     0.3862069
c 0.27640845    0.24827586 0.3167939     0.3189655
```

As previously mentioned, it can be found below the conditional distribution of A, E given $B = b$ and $F = b$, $D = b$ and $C = b$

```
> res5
[[1]]
           A A_Approx.Freq         E E_Approx.Freq
a 0.05115203    0.07142857 0.3167939     0.4285714
b 0.23753656    0.07142857 0.3664122     0.3214286
c 0.71131142    0.85714286 0.3167939     0.2500000

[[2]]
           A A_Approx.Freq         E E_Approx.Freq
a 0.05115203     0.0000000 0.3167939     0.3529412
b 0.23753656     0.1764706 0.3664122     0.2352941
c 0.71131142     0.8235294 0.3167939     0.4117647

[[3]]
           A A_Approx.Freq         E E_Approx.Freq
a 0.05115203    0.08333333 0.3167939     0.3333333
b 0.23753656    0.16666667 0.3664122     0.3333333
c 0.71131142    0.75000000 0.3167939     0.3333333

```

```
[[4]]
            A A_Approx.Freq         E E_Approx.Freq
a 0.05115203           0.05 0.3167939          0.40
b 0.23753656           0.15 0.3664122          0.25
c 0.71131142           0.80 0.3167939          0.35
```

Answer: As a result, the LS algorithm gives the same results every time (being exact) whereas the approximate one gives every time different result.

Comparing both algorithms, the approximate one performs worse when the conditioning set is large because it is based on first obtaining a sample from the distribution and then discarding the samples that are not consistent with the conditioning set (the less samples the higher variance). So, when the conditioning set is large, most of the samples are discarded leading to a poor approximation of the exact result.

In other words, the variance of the output prediction is greater the more nodes are observed because there is less data to calculate it, leading to a worse prediction of the approximate model compared to the exact one. Nevertheless, the goodness of the model is higher since the prediction will be more accurate.

Question 4

In this section We are asked to compute approximately the fraction of the 29281 DAGs that represent different independence models. In order to do so, I have used the random.graph function() sampling 10000 graphs. From these ones, I have check the unique ones and transformed them into DAGS. After that, I accounted for the the fraction of graphs resulted for different "burn.in" and "every" parameters getting the following results:

```
> result
            burn in = 1 burn in = 100 burn in = 10000 burn in = 1e+06
every = 2     0.6178384     0.6073096       0.6229086       0.6252186
every = 20    0.5471494     0.5539439       0.5519067       0.5567743
every = 100   0.5531273     0.5551819       0.5510988       0.5536594
every = 200   0.5454000     0.5534984       0.5536512       0.5552473
```

Results seem to show that the true proportion of essential graphs is approximately 0.55-0.6 of the DAGS. In light of the results, it is preferable to perform structure learning in the space of essential graphs for computation simplicity given that the result must be quite well accurate. The "Burn in" in this case does not change the result. These might mean that the process is directly stationary from the first observations. On the other side, the "every" parameter affects more to the output. By doing each observation less correlated with the previous ones (e.g. increasing the "every" parameter) the existent correlation through data tends to a certain level (e.g. in our case with 20 it already converges). leading to a more accurate fraction of the number of unique elements graphs.

Contributions

All results and comments presented have been developed and discussed together by the members of the group.

Appendix

Poisson regression-the MCMC way

```r

##Advanced Machine Learning lab 1


# source("http://bioconductor.org/biocLite.R")
# biocLite(c("graph", "RBGL", "Rgraphviz"))
#install.packages("gRain", dependencies=TRUE)
#install.packages("bnlearn")
#install.packages("gRbase")
#install.packages("gRain")
library(bnlearn)
library(gRain)
library(gRbase)

###1
data(asia)

plot(asia)

par(mfrow=c(1,2))

wl = matrix(c("E", "T"), ncol = 2, byrow = TRUE,
            dimnames = list(NULL, c("from", "to")))

plot(hc(asia, whitelist = wl), main ="HC with initial constraint")
plot(hc(asia, whitelist = NULL),main ="HC without constraint")

mygraph<-cpdag(hc(asia, whitelist = NULL, restart = 10))
cmygraph<-cpdag(hc(asia, whitelist = wl))
mygraph
cmygraph
all.equal(mygraph, cmygraph)

#################Alternative
set.seed(12345)
countinue <- TRUE
while(countinue){
  hc1 <- hc(alarm, restart = 10)
  hc2 <- hc(alarm, restart = 10)
  continue <- ifelse(all.equal(vstructs(hc1), vstructs(hc2)) == TRUE, TRUE, FALSE)
  if (continue !=TRUE){
    par(mfrow = c(1,2))
    graphviz.compare(hc1, hc2)
    par(mfrow = c(1,1))
    break
  }
}

###2
data("alarm")
n<-4
iss<- c(1,25,50,100)
mydag<-list()
par(mfrow=c(2,2))
for(i in 1:n){
  mydag[[i]]<- hc(alarm, restart = n,
                  score = "bde", iss = iss[i])
  plot(mydag[[i]], sub = paste0("iss = ", iss[i] ))
  print(alpha.star(mydag[[i]], alarm, debug = FALSE))
}


par(mfrow=c(1,1))

####3
data(learning.test)
pdag = iamb(learning.test)
pdag
plot(pdag)
dag = set.arc(pdag, from = "B", to = "A")
dag = pdag2dag(pdag, ordering = c("A", "B", "C", "D", "E", "F"))
plot(dag)
LS<- as.grain(fit)## it creates LS
plot(LS)
fit = bn.fit(dag, learning.test)##It creates all conditional tables from one edge with another
    with his possible outcomes

MM<- compile(LS)##it triangulates and moralizes
plot(MM)
basic<-querygrain(MM, nodes = c("A","E"))

```

```r
###################
###################comparison of results for one change
## I want the conditional distribution of A, E given B = b

##Exact one
ChangeEvidence<-setEvidence(MM, c("B"), c("b"))
finalstate<-querygrain(ChangeEvidence, nodes = c("A","E"))

##Approximate algorithm
rsample<-cpdist(fit, nodes = c("A", "E"), evidence= (B=="b"), method = "ls")
condprob<-lapply(rsample, FUN = function(x){table(x)/length(x)})

res1<-cbind(A_LS= as.data.frame(finalstate)[1], A_Greedy= condprob$A,
      E_LS= as.data.frame(finalstate)[2], E_Greedy= condprob$E)
res1<-res1[,c(1,3,4,6)]
colnames(res1)<- c("A_LS", "A_Greedy","E_LS", "E_Greedy" )
res1

##################comparison of results for two changes
## I want the conditional distribution of A, E given B = b and F = b
ChangeEvidence2<-setEvidence(MM, c("B","F"), c("b", "b"))
finalstate2<-querygrain(ChangeEvidence2, nodes = c("A","E"))


res2<- list()
for(i in 1:4){
  ##Exact one
  ChangeEvidence2<-setEvidence(MM, c("B","F"), c("b", "b"))
  finalstate2<-querygrain(ChangeEvidence2, nodes = c("A","E"))
  ##Greedy algorithm
  rsample2<-cpdist(fit, nodes = c("A", "E"), evidence= (B=="b")&(F=="b"), method = "ls")
  condprob2<-lapply(rsample2, FUN = function(x){table(x)/length(x)})

  res2[[i]]<-cbind(A_LS= as.data.frame(finalstate2)[1], A_Approx= condprob2$A,
              E_LS= as.data.frame(finalstate2)[2], E_Approx= condprob2$E)
  res2[[i]]<-res2[[i]][,c(1,3,4,6)]

}
res2


###wITHOUT EVIDENCE
## I want the conditional distribution of A, E

##Exact one
finalstate3<-querygrain(MM, nodes = c("A","E"))


##################comparison of results for two changes
## I want the conditional distribution of A, E given B = b and F = b, D=d
ChangeEvidence4<-setEvidence(MM, c("B","F","D", "A"), c("b", "b","b", "b"))
finalstate4<-querygrain(ChangeEvidence4, nodes = c("A","E"))

i<-1
res5<- list()
for(i in 1:4){
  ##Exact one
  ChangeEvidence5<-setEvidence(MM, c("B","F","D", "C"), c("b", "b","b", "b"))
  finalstate5<-querygrain(ChangeEvidence5, nodes = c("A","E"))
  ##Greedy algorithm
  rsample5<-cpdist(fit, nodes = c("A", "E"), evidence= (B=="b")&(F=="b")&(D=="b")&(C=="b"),
      method = "ls")
  condprob5<-lapply(rsample5, FUN = function(x){table(x)/length(x)})

  res5[[i]]<-cbind(A_LS= data.frame(finalstate5)[1], A_Approx= data.frame(condprob5$A),
                  E_LS= data.frame(finalstate5)[2], E_Approx= data.frame(condprob5$E))
  res5[[i]]<-res5[[i]][,c(1,3,4,6)]

}
res5



###4
burn_in<- c(1,100, 10000, 1000000)
every<-c(2,20,100, 200)

nodes<- LETTERS[1:5]
num <- 1000


checkingDags<- function(burnin, every, num, nodes){
  mymat<- matrix(ncol =length(burnin), nrow=length(every))
  for(i in 1:length(burnin)){
    for(j in 1:length(every)){
      z<-random.graph(nodes = nodes, num = num,  method = "ic-dag",
                      every = every[j], burn.in =burnin[i])

      m<-unique(z)
      print(length(m))# This gives the proportion the number of repeated graphs out my

      dagGraph<-lapply(m, FUN= function(x){cpdag(x)})
      mymat[j,i]<- length(unique(dagGraph))/length(m)
      }
  }
  return(mymat)
}

result<-checkingDags(burnin = burn_in, every = every, num = num, nodes = nodes)
colnames(result)<- paste0("burn in = " , burn_in)
rownames(result)<- paste0("every = " , every)
result
```