UNIVERSITÄT REGENSBURG

Institute of Genomics & Practical Bioinformatics
Master of Science Computational Science

The max-min-hill-climbing algorithm

Report
in practical Bioinformatics

by

Michael Bauer

Matrikelnummer: 152 8558

Tutor:	Dr. Giusi Moffa
Adviser:	Prof. Dr. Rainer Spang
Date:	August 22, 2014

Contents

List of Figures

List of Tables

1 Abstract

In this report we present a new implementation of the max-min-hill-climbing algorithm (MMHC) for R, first stated in [TBA]. It combines both: greedy search and constraint-based learning techniques. We will discuss those two methods seperately and how they effect running time. We also want to work out the importance of this algorithm. The main goal of it is to reconstruct Bayesian networks from estimated data. Bayesian networks play a great role in science, economics, sports and many more fields where the observational data can get extremely big. It is not the first time R provides this algorithm but we come up by using RCPP (C++ interface for R) for our implementation to have a better chance to deal with big data. Since running time is getting more important we tried to improve it for this algorithm and beat the existing one.

2 Introduction

The max-min-hill-climbing algorithm is one of the state of the art algorithms in statistical computing. The primal aim of this algorithm is reconstructing BN out of estimated data. A Bayesian network is a Directed Acyclic Graph (DAG) whose nodes are random variables and eges represent conditional dependencies. If two random variables are connected they are said to be dependent. If there is no connection they are said to be conditional independent. BNs are more important than one can imagine. They play a great role in everdays life. For example [NBBCW] use them to predict the effect of missense mutations on the protein function. But not only medical science uses Bayesian networks. Another example where scientists used them was football. In [PKA] they refer to an article where european football clubs tried to predict injuries of their players depending on BNs. With a simple Google search you may also find the prediction of stock exchanges and many more. Wikipedia provides a simple but descriptive example which illustrates BNs in a smooth way.

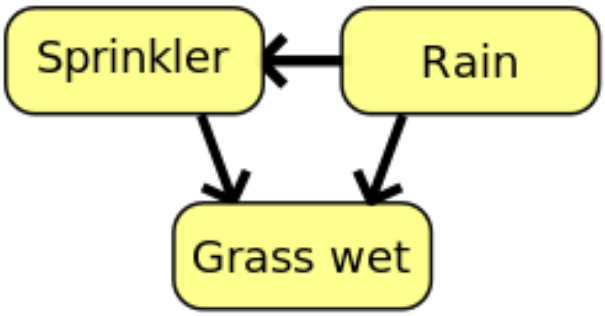


Figure 2.1: A simple example of a Bayesian network.

Reconstructing those networks is not easy, more precisely it is a np-hard problem. It is not only the amount of data which leads to a bad running time, also the dependencies between nodes can slow the code down.
In the first step of this algorithm we try to reconstruct the skeleteon of the graph. Therefor we iterate over all variables - we select one variable (we call it "target" **T**) in each iteration step - and find those variables which are dependent to the selected one. The more variables we find the longer a single iteration takes. The dependent variables are then said to be a parent or a child of **T**. It may happen that there are false positive ones in our set (which we will call

CPC). For this reason we have to check again if the parents/children in the set really belong to our selected variable **T**. The relation between **T** and its parents/children is symmetric. That means for a target **T** and $\mathbf{A} \in \mathbf{CPC}_T$ it follows:

$$\mathbf{T} \longleftrightarrow \mathbf{A} \Longleftrightarrow \mathbf{A} \longleftrightarrow \mathbf{T} \tag{2.1}$$

for a target **A** and $\mathbf{T} \in \mathbf{CPC}_A$. Since this relation holds we have to check in a second step if this symmetrie is fullfild. For that we check for every $\mathbf{X} \in \mathbf{CPC}_T$ if $\mathbf{T} \in \mathbf{CPC}_X$.
Though this is a great approach, we are interested in the whole Directed Acyclic Graph. The second part of the algorithm will then take this skeleton and add directed edges to it such that it does not become cyclic and is fully directed. We will see that this is based on one formula which is not complicated to understand but extremely tricky for implementation. Once we observe the Bayesian network from our data we then can look at the running time of the algorithm and where the time gets lost but also where we optimized to save time. But more important for us was to beat the existing algorithm for R (part of the "bnlearn" package). The goal for us was to be faster. So after a brief discussion of our implementation we will see if it was possible to optimize the code with RCPP to beat the "bnlearn" algorithm.

3 Background

Before we are able to analyze our implementation and talk about it in detail we need some mathematical background. In this section we fully follow [TBA][p. 5-7]. For the proofs of the Lemmas and Theorems we also reference to this paper.

3.1 Notation

We introduce our notation which is completely consistent to [TBA].
We denote

1. variables with an upper-case letter (e.g., A, V_i),
2. a state or a value of that variable by the same lower-case letter (e.g., a, v_i),
3. a set of variables by upper-case bold face (e.g., $\mathbf{Z}, \mathbf{Pa}_i$),
4. an assignment of state or value to each variable in the given set with the corresponding lower-case bold-face letter (e.g., $\mathbf{y}, \mathbf{pa}_i$),
5. special sets of variables (e.g. the set of all variables $\mathcal{V}$) with calligraphic fonts.

3.2 Definition (conditional independence)

Two variables X and Y are conditionally independent given $\mathbf{Z}$ with respect to a probability distribution P, denoted as $Ind_P(X; Y|\mathbf{Z})$, if for all $x, y, \mathbf{z}$ where $P(\mathbf{Z} = \mathbf{z}) > 0$,

$$P(X = x, Y = y|\mathbf{Z} = \mathbf{z}) = P(X = x|\mathbf{Z} = \mathbf{z})P(Y = y|\mathbf{Z} = \mathbf{z}) \tag{3.1}$$

or

$$P(X, Y|\mathbf{Z}) = P(X|\mathbf{Z})P(Y|\mathbf{Z}) \tag{3.2}$$

for short. If X, Y are dependent given $\mathbf{Z}$ we denote $Dep_P(X; Y|\mathbf{Z}$.

3.3 Definition (Bayesian network)

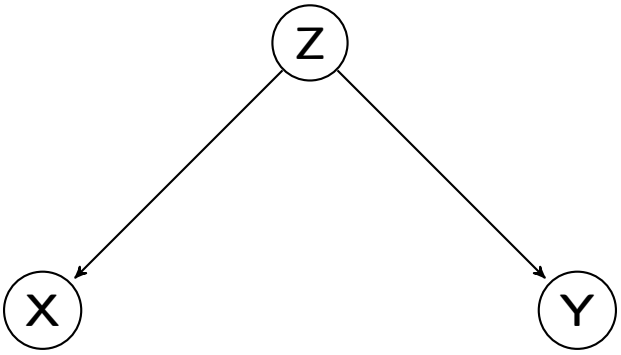
Let P be a discrete joint probability distribution of the random variables in some set $\mathcal{V}$ and $\mathcal{G} =< \mathcal{V}, E >$ be a Directed Acyclic Graph (DAG). We call $< \mathcal{G}, P >$ a (discrete) *Bayesian network* if $< \mathcal{G}, P >$ satisfies the Markov condition.

3.4 Definition (Markov condition)

Any node in a Bayesian network is conditionally independent of its non-descendants, given its parents.

3.5 Explanation

With those definitions we have our first concept we will discuss briefly. We will explain the definitions by using the following picture:



The Markov condition states that X and Y given $\mathbf{Z}$ must be conditionally independent. This comes from the fact that X is a non-descendant of Y and vice versa and $\mathbf{Z}$ is a parent of both nodes. By fullfilling this condition we get from section 3.3 that this graph is a Bayesian network and with section 3.2 we have: $P(X, Y|\mathbf{Z}) = P(X|\mathbf{Z})P(Y|\mathbf{Z})$.

3.6 Defintion (collider)

A node W of a path p is a *collider* if p contains two incomming edges into W.

3.7 Definition (blocked path)

A path p from node X to node Y is *blocked* by a set of nodes $\mathbf{Z}$, if there is a node W on p for which one of the following two conditions hold:

1. W is not a collider and $W \in \mathbf{Z}$, or
2. W is a collider and neither W or its descendants are in $\mathbf{Z}$ [P88]

3.8 Definition (d-seperation)

Two nodes X and Y are *d-seperated* by $\mathbf{Z}$ in graph $\mathcal{G}$ (denoted as $Dsep_{\mathcal{G}}(X; Y|\mathbf{Z})$) if and only if every path from X to Y is blocked by $\mathbf{Z}$. Two nodes are *d-connected* if they are not *d-seperated*.

3.9 Definition (faithful)

If all and only the conditional independencies true in the distribution P are entailed by the Markov condition applied to $\mathcal{G}$, we will say that P and $\mathcal{G}$ are *faithful to each other* ([SGSN]). Furthermore, a distribution P is *faithful* if there exists a graph $\mathcal{G}$, to which it is faithful.

3.10 Definition (faithfulness condition)

A Bayesian network $< \mathcal{G}, P >$ satisfies the *faithfulness condition* if P embodies only independencies that can be represented in the DAG $\mathcal{G}$ ([SGSN]). We will call such a Bayesian network a *faithful network*.

3.11 Theorem

In a faithful Bayesian network $< \mathcal{G}, P >$ the following equivalence holds ([P88])

$$Dsep_{\mathcal{G}}(X; Y|\mathbf{Z}) \Longleftrightarrow Ind_P(X; Y|\mathbf{Z}) \tag{3.3}$$

3.12 Remark and Explanation

Remark: For the rest of this report we assume faithfulness of the networks to learn. For this reason we don't want to explain the corresponding definitions in detail. Just note, that the definitions are neccessary for mathematical correctness.

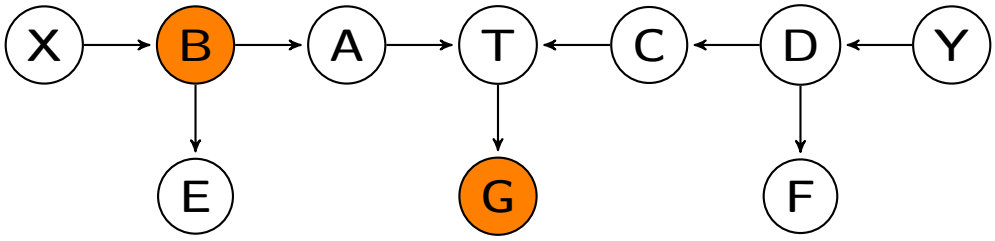
Explanation: The definition of a collider already tells everything about it. To illustrate a collider, we have:

Here Z is a *collider* because it has two incoming edges. In this case if we just look for $P(X;Y|\{\})$, the path between X and Y would be blocked and for this X and Y are *d-seperated*. If we look for $P(X;Y|\mathbf{Z})$, then this path is not blocked and we state that X and Y are *d-connected*.
Because of section 3.11 and the faithfulness assumptions we state for the rest of our report that the terms d-seperation and conditional independence are equivalent. With this we already know that X and Y are conditional dependent given $\mathbf{Z}$ in the example above. This brings us a big step closer to learn the structure of a Bayesian network from observational data. Before we start looking at the algorithms, we want to give you two other examples for d-seperation of variables.

If we are looking for $Ind_P(X;Y|\mathbf{Z})$ with $\mathbf{Z} = \{B, D\}$ we learn that X and Y are conditionally independent given $\mathbf{Z}$. In other words they are d-seperated in the path because of the following reasons:

- T is a collider with $T \notin \mathbf{Z}$ and blocks the path between X and Y.
- The nodes B and D are no colliders but they are elements of $\mathbf{Z}$.

The situation becomes a bit more difficult if we take a look at the next example:



We learn that the path between X and Y remains blocked by looking for $Ind_P(X;Y|\mathbf{Z})$ with $\mathbf{Z} = \{B, G\}$, i.e. X and Y are conditionally independent given $\mathbf{Z}$. If we would look at the path between A and Y we would learn that A and Y are d-connected. This comes from:

- T is a collider but its descendant $G \in \mathbf{Z}$, i.e. T would not block the path.
- The node B is no collider but it is an element of $\mathbf{Z}$. For that it blocks the path.

So there is no element which blocks the path between A and Y, but for X and Y, B blocks the path.
As we could see, detecting conditional independence of two nodes is quite difficult in small graphs. Since we normally observe large data sets with a couple of nodes, a concept for this is needed. As we will see, statistical methods, such as hypothesis testing will be a useful friend for this task.

3.13 Definition

We define the minimum association of X and T relative to a feature subset $\mathbf{Z}$, denoted as $MinAssoc(X;T|\mathbf{Z})$, as

$$MinAssoc(X;T|\mathbf{Z}) = \min_{\mathbf{S}\subseteq\mathbf{Z}} Assoc(X;T|\mathbf{S}) \tag{3.4}$$

i.e., as the minimum association achieved between X and T over all subsets of $\mathbf{Z}$.
Remark:

1. In Figure 5.1.1 we will see this $MinAssoc$ statement again as a function. With that we measure conditional independence of X and T given $\mathbf{Z}$.
2. For that the following equivalence holds (providing without a proof):

$$Ind(X;T|\mathbf{Z}) \iff (Assoc(X;T|\mathbf{Z} = 0). \tag{3.5}$$

4 Functions of bnlearn

As we stated, our purpose was to implement the max-min-hill-climbing algorithm in a way that it will be faster and more efficient than the existing is. That's why we want to give you a brief introduction to the mmpc() and mmhc() functions of the "bnlearn" package. We also want to present some numbers depending on running time which show you the effectiveness of the two algorithms. Afterwards we explain our algorithm and we present our results. You will see that there will be some differences on the running time of both implementations.

4.1 mmpc(data.frame)

This function represents the first part of the algorithm, which returns the skeleton of the graph. You can choose between several methods of independence testing. The input is a R data frame and the return value of it is of the class "bn". With the plot function you are able to plot the whole skeleton of the graph, like:

```
> graph <- mmpc(data.frame)
> plot(graph)
```

4.2 mmhc(data.frame)

This function is similiar to the mmpc(). The differences are, that it returns a DAG and you can additionally choose a score function which is used to direct the edges. Again you execute this with:

```
> graph <- mmhc(data.frame)
> plot(graph)
```

5 The max-min-hill-climbing algorithm

We now want to go into the implementation of our algorithm. First of all we want you present the pseudo code and then talk about it. Afterwards we present the statistical methods behind the algorithm and how they effect running time. In the end we want to present an example which should be reconstructed by our algorithm.

5.1 max-min-hill-climbing algorithm

Algorithm *MMHC* Algorithm

1: **procedure** MMHC($\mathcal{D}$)
   Input: data $\mathcal{D}$
   Output: a DAG on the variables in $\mathcal{D}$
   % Restrict
2: **for** every variable $X \in \mathcal{V}$ **do**
3: $\mathbf{PC}_X$ = MMPC($X, \mathcal{D}$)
4: **end for**
   % Search
5: Starting from an empty graph perform Greedy Hill-Climbing with operators *add-edge*, *delete-edge*, *reverse-edge*. Only try operator *add-edge* $Y \rightarrow X$ if $Y \in \mathbf{PC}_X$.
6: **Return** the highest scoring DAG found
7: **end procedure**

Figure 5.1: The pseudo code of the MMHC algorithm. Line 2-4 represent the loop over all nodes calling the MMPC function. At line 5 the scoring starts.

As mentioned before, the MMHC function has to parts. First we find the skeleton and then direct the edges of the skeleton graph. We first discuss the *MMPC* (max-min parents and children) function which is called the observational data $\mathcal{D}$ in the for loop (over all possible nodes in the graph) and then we take a closer look at the scoring function starting at line 5. This function will later be stated as *BDeu* which stands for Bayesian Dirichlet likelihood-equivalence uniform.

5.1.1 max-min parents and children (MMPC))

The max-min parents and children (MMPC) is the algorithm which reconstructs the graph skeleton from data. To bring you the concept of this closer, let us first present the pseudo code of it:

Algorithm Algorithm *MMPC*

```
1: procedure MMPC(T,D)
2:     CPC = MMPC(T, D)
3:     for every variable X ∈ CPC do
4:         if T ∉ MMPC(X, D) then
5:             CPC = CPC \ X
6:         end if
7:     end for

8:     return CPC
9: end procedure
```

Figure 5.2: The pseudo code of the MMPC function. First the **CPC** set is computed by the $\overline{MMPC}$ function. Afterwards the false positives are erased.

In this algorithm another function $\overline{MMPC}$ is executed. In Figure 5.1.1 this function - including the *MaxMinHeuristic* function - is shown. Back to Figure 5.1.1, we see, that $\overline{MMPC}$ is executed twice. Firstly at line 2 and secondly in the if-statement at line 4. As we stated in Equation 2.1, for two variables they are connected, the symmetrie holds. At line 2 we only now that some X are in the set **CPC** for a target variable T, i.e. the X is a parent or a child of T. By testing if T is also a parent or child of X, we see if the symmetrie holds, if not, X is removed from the **CPC** set. Let's talk about the $\overline{MMPC}$ function. First take a look at is:

Algorithm $\overline{MMPC}$ Algorithm

```
1: procedure MMPC (T,D)
       Input: target variable T; data D
       Output: the parents and children of T in any Bayesian
       network faithfully representing the data distribution
       %Phase I: Forward
2:     CPC = ∅
3:     repeat
4:         ⟨F, assocF⟩ = MaxMinHeuristic(T; CPC)
5:         if assocF ≠ 0 then
6:             CPC = CPC ∪ F
7:         end if
8:     until CPC has not changed

       %Phase II: Backward
9:     for all X ∈ CPC do
10:        if ∃S ⊆ CPC, s.t. Ind(X;T|S) then
11:            CPC = CPC \ {X}
12:        end if
13:    end for

14:    return CPC
15: end procedure

16: procedure MaxMinHeuristic(T,CPC)
       Input: target variable T; subset of variables CPC
       Output: the maximum over all variables of the minimum asso-
       ciation with T relative to CPC, and the variable that achieves
       the maximum
17:    assocF = max_{X∈V} MinAssoc(X;T|CPC)
18:    F = arg max_{X∈V} MinAssoc(X;T|CPC)
19:    return ⟨F, assocF⟩
20: end procedure
```

Figure 5.3: Here you see both: the $\overline{MMPC}$ function and the calculation of the association between X and T by the *MaxMinHeuristic* function.

In this subfunction we have two important routines: the execution of $\overline{MMPC}$ and the *MaxMinHeuristic*.

5.1.1.1 The *MaxMinHeuristic* function

This function finds the X which maximizes the measure of association between X and T given the current **CPC** set. The current **CPC** set is therefor passed into the function. The return of the function is the X and the value of association between X and T given

CPC. At this point, we left one important question: How do we measure this association.

5.1.1.2 The G^2 value

Because this algorithm is based on conditional independence testing and measuring the strength of association between two variables, we need some formulas for implementation to get this measure. We followed [SGSN] and calculated the G^2 statistic, under the null hypothesis of the conditional independence holding. For that we have (from [TBA]):
Let $S_{ijk}^{ab\mathbf{c}}$ be the number of times in the data where $X_i = a$, $X_j = b$ and $X_k = \mathbf{c}$. We define in a similar fashion, $S_{ik}^{a\mathbf{c}}$, $S_{jk}^{b\mathbf{c}}$ and $S_k^{\mathbf{c}}$, then the G^2 statistic is defined as (c.f. [SGSN]):

$$G^2 := 2 * \sum_{a,b,c} S_{ijk}^{ab\mathbf{c}} * ln\left(\frac{S_{ijk}^{ab\mathbf{c}} * S_k^{\mathbf{c}}}{S_{ik}^{a\mathbf{c}} * S_{jk}^{b\mathbf{c}}}\right), \tag{5.1}$$

The G^2 statistic is asymptotically distributed as χ^2 with appropriate degrees of freedom. To compute these degrees of freedom we use:

$$df = (|D(X_i)| - 1)(|D(X_j)| - 1) \prod_{X_l \in \mathbf{X_k}} |D(X_l)|, \tag{5.2}$$

where $D(X_i)$ is the domain (number of distinct values) of variable X_i. After bringing this to code, we can work with the output of this function.

5.1.1.3 How $\overline{MMPC}$ works

The procedure of the $\overline{MMPC}$ function then works as follows:

- Compute the G^2 value. You observe a $Svalue$ of type *double* value.
- Compute the degrees of freedom df.
- Assign the $pvalue$ to the output of the function $pchisq(Svalue, df)$.
- Test if the $pvalue$ is smaller than 0.05, if yes then keep this $pvalue$, it's corresponding $Svalue$ and X.
- At the end let the X join the **CPC** set which has the smallest $pvalue$.
- If two pvalues are equaled then take the variable X with the higher $Svalue$.
- If you checked all variables over all subsets of the **CPC** set, stop the calculations.

Hint: At the moment you find that a target T and a variable X are conditionally independent, i.e. the association values equales zero, you don't need to check this connection again since you already know of their independence. The same holds for the fact, if they are dependent. This saves running time.
After symmetrie testing in the $MMPC$ function, the first part of the algorithm returns all the parents and children of all variables. We realized this within a list. Every element of the list - numbered from 1 to n - stands for a variable and contains a vector which holds the parents and the children of this variable. The second part of the max-min-hill-climbing algorithm now takes this list and directs the edges from one variable to another.

5.1.2 The BDeu score

Before we present our realization of the BDeu score, we want to give a short definition of it partly following [Ca06].

5.1.2.1 Definition (BDeu score)

The Bayesian Dirichlet likelihood-equivalence uniform score (short: BDeu score) is defined as:

$$g_{BDeu}(D,G) = \sum_{i=1}^{n}\left[\sum_{j=1}^{q_i}\left[\log\left(\frac{\Gamma(\frac{\eta}{q_i})}{\Gamma(N_{ij}+\frac{\eta}{q_i})}\right)+\sum_{k=1}^{r_i}\log\left(\frac{\Gamma(N_{ijk}+\frac{\eta}{r_i q_i})}{\Gamma(\frac{\eta}{r_i q_i})}\right)\right]\right], \qquad (5.3)$$

where G is a graph, D is the underlying data, the number of states of the variable X_i is r_i, the number of possible configurations of the parent set $Pa_G(X_i)$ of X_i is q_i, with $q_i = \prod_{X_j \in Pa_G(X_i)} r_j$, $w_{ij}, j = 1, ..., q_i$ represents a configuration of $Pa_G(X_i)$, N_{ijk} is the number of instances in the data set D where the variable X_i takes the value x_{ik} and the set of variables $Pa_G(X_i)$ take the value w_{ij}, N_{ij} is the number of instances in the data set where the variable in $Pa_G(X_i)$ take their $j-th$ configuration w_{ij}, with $N_{ij} = \sum_{k=1}^{r_i} N_{ijk}$, N_{ik} is the number of instances in D where the variable X_i takes its $k-th$ value x_{ik} and therefor $N_{ik} = \sum_{j=1}^{q_i} N_{ijk}$ and the total number of instances in D is n.

5.1.2.2 Explanation

Of course, this definition needs some explanation. In simple words this formula takes a graph and returns a score. That is what we need for our computations. We are seeking for the graph with the highest score. But first of all back to the calculation of the score. To compute the

sums you just have to know how to get all those values explained above. Let us give you a short overview over all those variable:

- **eta**: We set this value to 1 - the most common setting. There is no rule which value is the best and one could find many papers which try to find the "best" eta. If you are interested in that we recommend you [1]
- **n**: The number of variables in our BN.
- r_i: How many states the $i-th$ variable can take.
- q_i: The product of all states of the parent's variables of the $i-th$ variable, e.g. if node X has two parents Y and Z $q_X = r_y \cdot r_z$.
- w_{ij}: This is a configuration of the parents of a certain variable, i.e. recall X with parents Y, Z and we assume that the states of Y, Z are binary. Then w_{ij} can have four different configurations: $(0;0), (0;1), (1;0), (1;1)$.
- N_{ijk}: Counts how often a variable X_i takes the value x_{ik} and its parents take the configuration w_{ij}.
- N_{ij}, N_{ik}: Is the calculation of the sums over the depending N_{ijk} values from above.

As already mentioned this function takes a graph and returns a value. This value should be the best score which can be achieved depending on the list returned by the *MMPC* algorithm. For this we take an empty graph, i.e. we have all our nodes but no edges, and compute the score of this empty graph. Then we choose by random one node/variable X and add an edge $X \longrightarrow Y$, if Y is a parent/child of X. This we know from our *PC* set, returned by *MMPC*. We compute the score again and if the new score is higher than the one of the empty graph, this score is saved and the edge stays. If it is not higher, the edge will be removed. We also apply the operations "revers edge" and "delete edge" to the graph in a similar fashion. If the score reaches a maximum we determine the search and return the "best" graph. The word best is in quotes, because randomness in our data ensures that we find the graph with the best score, but this must not be the correct one.

5.2 Introduction to an example

We now want to talk about the main issue we had when implenting this algorithm. We wanted to provide code which is valid and beat the running time of the functions of the bnlearn package. Our comparisons depend on a data set which is build from a self written function. The rules to create these data are taken from [KoFr]. All our benchmarks and comparisons between our implementation and the bnlearn package depend on this data. For that reason we want to introduce it in the following section.

5.2.1 Student data frame

The following graph is the one where we started. Normally you don't start with a graph. You start with data and end up with a graph. In our case we needed a posibility to test if the code works properly. The output of the *MMHC* should return this graph or a one which is likely to this.

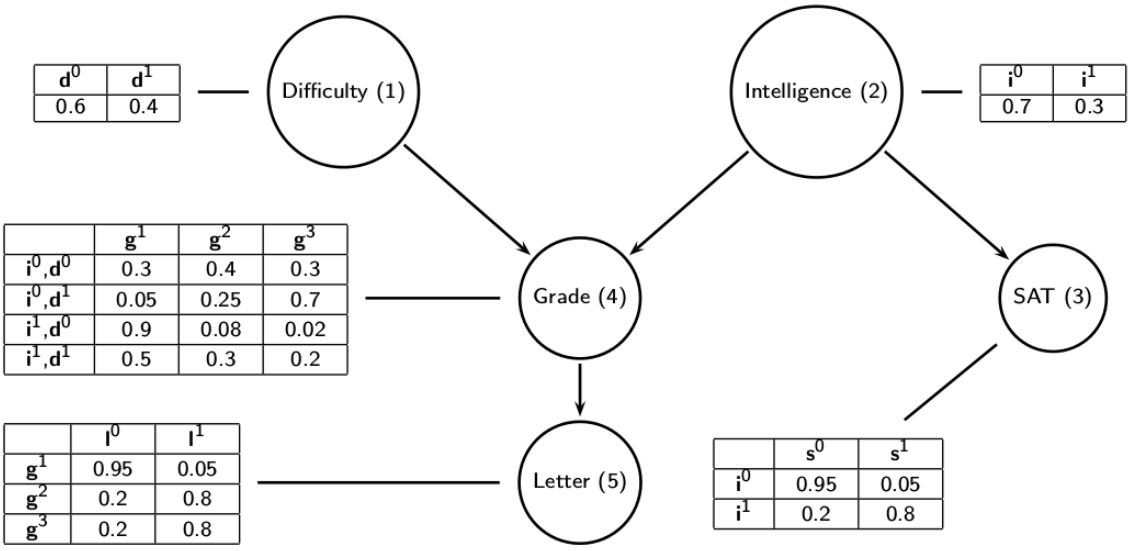


Figure 5.4: This graph should later be returned from the *MMHC* algorithm.

The probabilities and rules in this graph are computed by a R function which you find in our package. We won't provide further information here.

5.3 Computational effort

We decided to write our algorithm object oriented. With this we save a lot of running time. We have a second implementation - which will not be provided - that is written procedural. So before we will see, if our algorithm could beat the bnlearn package, we want to present the differences depending on running time between object oriented and procedural programming style. Of course, this comparison only holds for this case. Somebody else could have a greater effort on this by procedural programming.
As we saw above we have five variables. For this example case we computed 10,000 observations, i.e. we have a R data frame $df \in \mathbb{N}^{10000\times 5}$, where all nodes have binary values $1, 2$

except the node holding the grade. This has three states: $1, 2, 3$.

The first comparison we want to show is between the *MMPC* version where $C\$mmpc()$ is the object oriented and $MMPC(mat, 0.05)$ the procedural programming version:

```
df <- student(10000)
bm <- benchmark(C$mmpc(), MMPC(mat, 0.05),
                columns = c("test", "elapsed", "relative"),
                replications = 1)
print(bm)
             test elapsed relative
1        C$mmpc()   0.043    1.000
2 MMPC(mat, 0.05)   0.148    3.442
```

Clearly, the object oriented version is faster than the other. By comparing only the BDeu scoring we observe:

```
df <- student(10000)
bm <- benchmark(C$mmpc(), MMPC(mat, 0.05),
                columns = c("test", "elapsed", "relative"),
                replications = 1)
-27812.9 # score returned by BDeu(...)
-27812.9 # score returned by C$mmpc()
print(bm)
                          test elapsed relative
2 BDeu(mat, PC, as.integer(5))   1.419    8.446
1                     C$mmhc()   0.168    1.000
```

At this point it becomes clear, why we focused on our object oriented version. To complete this we present the benchmark of both by running the full algorithms. We also present the graph which is returned by the computations.

```
df <- student(10000)
bm <- benchmark(MMHC_oop(df), MMHC_pp(df),
                columns = c("test", "elapsed", "relative"),
                replications = 1)
-27619.2 # score of the graph returned by MMHC_pp(df)
```

```
-27619.2 # score of the graph returned by MMHC_oop(df)
print(bm)
          test elapsed relative
1 MMHC_oop(df)   0.253    1.000
2  MMHC_pp(df)   2.204    8.711
```

For plotting the graph we used the "igraph" R package. The resulting graph is then:

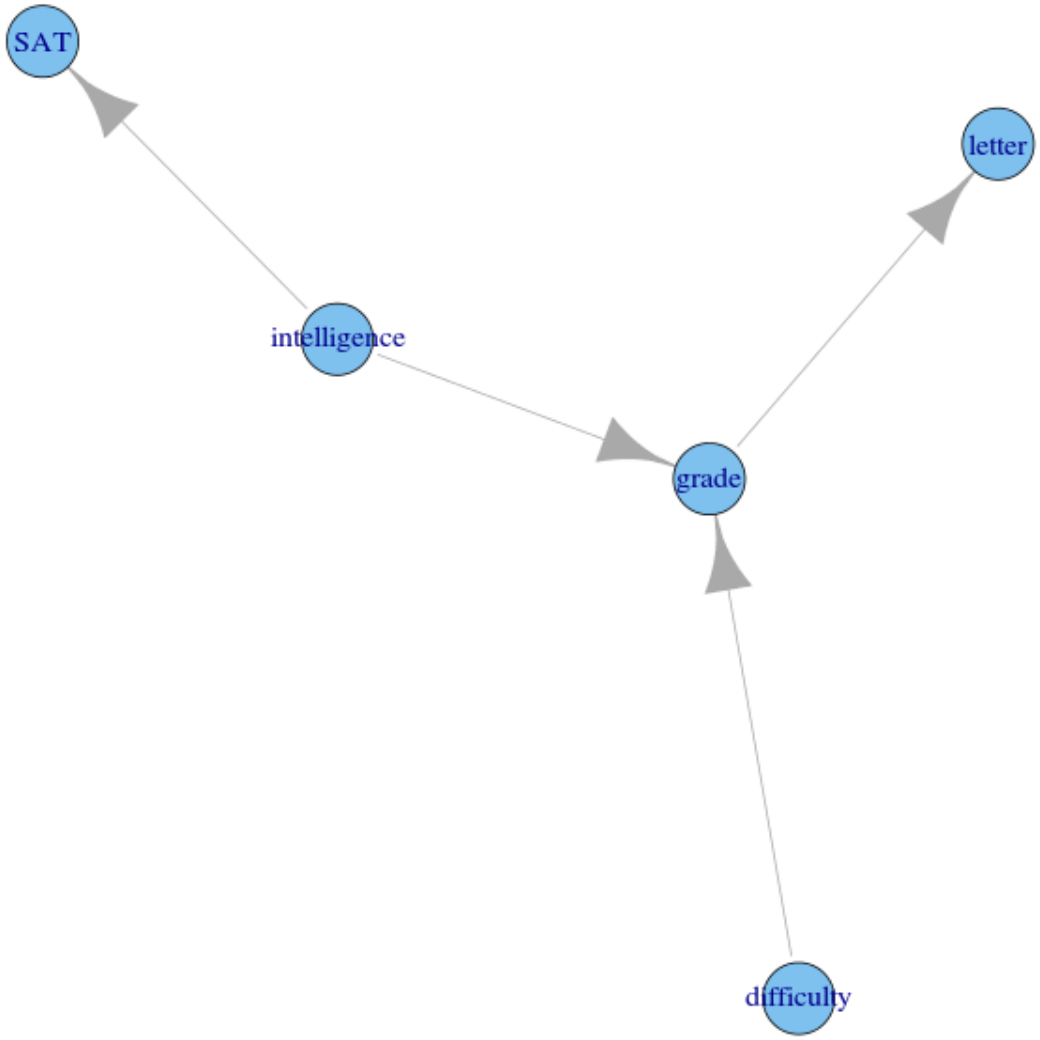


Figure 5.5: The graph returned by both functions looks like the one we started with.

This section also showed that the *MMPC* algorithm is much faster then the *BDeu* score. This will play a great role in the next chapter.

6 Beating bnlearn

Beating the bnlearn was a very challenging and difficult task. This package was developed over years. It was a PhD thesis project which went over at least one year. We only had three months for this task. But with some tricks and computational knowledge we were able to implement an algorithm which is fast and valid. The question now is, if we beated the bnlearn package and the answer is no. This comes from the fact, that the bnlearn package has a very smooth and kind of constant running time. We guessed that they were precomputing some values and then just look them up. We also tried this method, but it failed. In the end it sounds like we mist our goal, but that is not the case. As we will see our running time is also quite good and developing goes on. We will also release a package for our algorithm, because for less number of observations, i.e. up to 7,500, our *MMPC* is a bit faster. In some computiations on other machines, e.g. a Mac Book Pro, our algorithm is extremely good, only bnlearn is a bit better.

Now, let's take a look at the running times. First of all we took a range of observations from 1,000 up to 20,000 for *MMPC* and 10,000 for *MMHC* respectively in steps of 1,000. We then ran our benchmark 10 times, added up the running times and divided by 10 to avoid large peaks in our plots. What you see on the picture is the number of observations against running time in seconds:

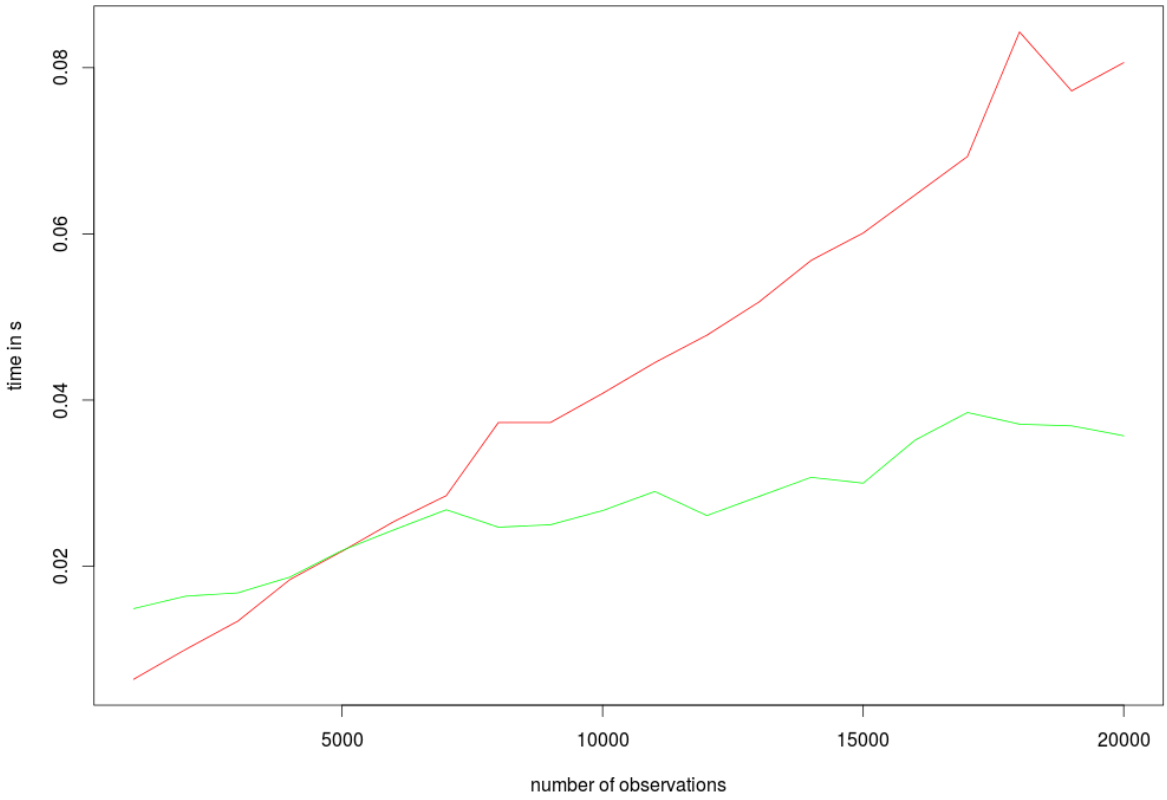


Figure 6.1: From 1,000 to 7,500 observations our algorithm (red line) is quite fast. Above this number bnlearn one (green line) is much faster.

But if you look at the right boundary of the function you discover that our implementation only needs about 0.08 seconds to find the skeleton of a graph out of 100,000 values. But also the running time for the *MMHC* algorithm is not bad. About 1.4 seconds for the biggest data frame. To beat bnlearn with this is extremely difficult. They only need about 0.1 seconds and this is amazing.

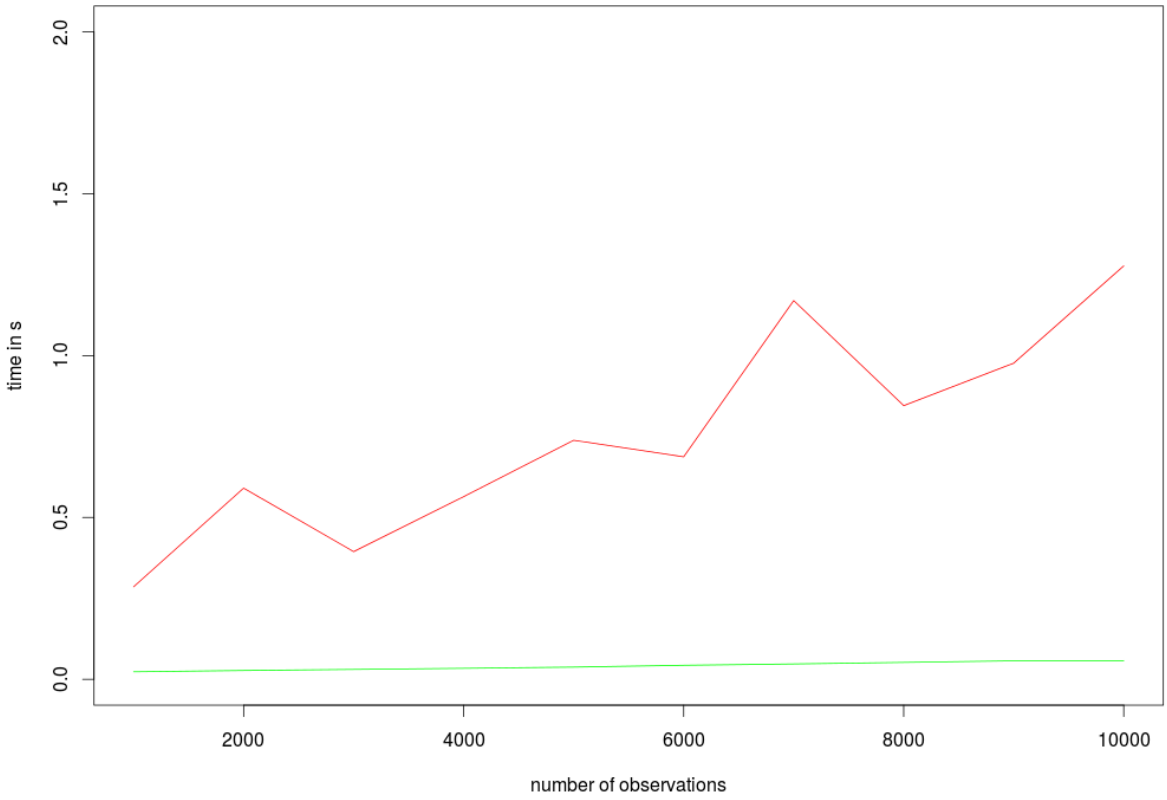


Figure 6.2: Hear you see that our algorithm (red line) grows linearly where as the bnlearn one (green one) is constant.

7 Conclusion

My conclusion is that I don’t have a conclusion :)

Bibliography

[TBA] Ioannis Tsamardinos, Laura E. Brown, Constantin F. Aliferis,
The max-min hill-climbing Bayesian network structure learning algorithm,
Springer Science + Business Media, Inc. 2006

[NBBCW] Chris J. Needham1, James R. Bradford, Andrew J. Bulpitt, Matthew A. Care and David R. Westhead,
Predicting the effect of missense mutations on protein function: analysis with Bayesian networks,
http://www.biomedcentral.com/1471-2105/7/405

[PKA] http://www-ekp.physik.uni-karlsruhe.de/županc/WS1011/docs/Datenanalyse2010_3.pdf

[P88] Pearl, 1988

[SGSN] Spirtes, Glymour & Scheines, 1993, 2000;
Neapolitan, 2003

[Ca06] Luis M. de Campos,
A Scoring Function for Learning Bayesian Networks based on Mutual Information and Conditional Independence Tests,
Journal of Machine Learning Research 7, 2006

[1] [SKM] On Sensitivity of the MAP Bayesian Network Structure to the Equivalent Sample Size Parameter,
Tomi Silander and Petri Kontkanen and Petri Myllymäki,
UAI, 2007

[KoFr] Daphne Koller, Nir Friedman,
Probabilistic Graphical Models: Principles and Techniques (Adaptive Computation and Machine Learning),
The MIT Press,
First edition (16. November 2009)