UNIVERSITÄT REGENSBURG

Institute of Genomics & Practical Bioinformatics

Master of Science Computational Science

The max-min-hill-climbing algorithm an implementation in Rcpp

Report

in Practical Bioinformatics

by

Michael Bauer

Matrikelnummer: 152 8558

Tutor:	Dr. Giusi Moffa
Adviser:	Prof. Dr. Rainer Spang
Date:	September 6, 2014

Contents

List of Figures

Abstract

In this report we present an implementation of the max-min-hill-climbing algorithm, first stated in [TBA], for R. It combines both: greedy search and constraint-based learning techniques. We will discuss those two methods seperately. The main goal of this algorithm is to reconstruct Bayesian networks from estimated data. Bayesian networks play a great role in science, economics, sports and many more fields. Reconstructing them with the help of data is ongoing research and that's why there already exists a package in R which provides this algorithm. Our purpose was to write code which runs faster than the existing one. We used RCPP (C++ interface for R) for our implementation. This report tells you how our algorithm works and if we succeded with our aim.

1 Introduction

The max-min-hill-climbing algorithm (MMHC) is one of the state of the art algorithms in statistical computing. The primal aim of this algorithm is reconstructing Bayesian Networks (BN) from estimated data. A Bayesian network is a Directed Acyclic Graph (DAG) whose nodes are random variables and edges represent conditional dependencies. If two random variables are connected they are said to be conditionally dependent. If there is no connection they are said to be conditionally independent. BNs are more important than one can imagine. They play a great role in everdays life. For example [NBBCW] used them to predict the effect of missense mutations on the protein function. But not only medical scientists used Bayesian networks. Another example we found 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.

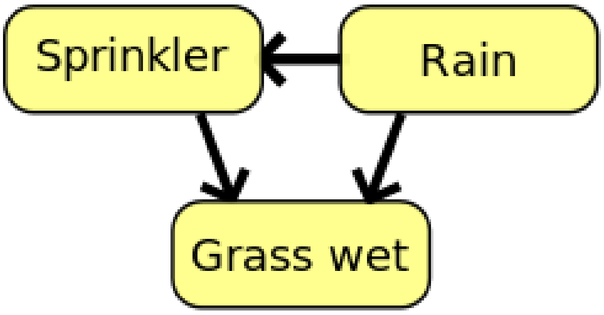


Figure 1.1: A simple example of a Bayesian network where Sprinkler & Rain, Sprinkler & Grass Wet and Grass Wet & Rain are conditionally dependent.

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 an increase of running time, also the dependencies between nodes can slow the code down.

Once we used our data to reconstruct a Bayesian network we can take a look at how long the computations lasted. Our aim was to be faster than the existing algorithm in R (part of the "bnlearn" package), which is implemented in pure C. We tried to beat their running times and to be more efficient by dealing with the data. After an introduction to the mathematical basis and our implementation we will see if it was possible to optimize our code with RCPP to beat the "bnlearn" algorithm.

2 Background

Before we are able to analyze our implementation and talk about it in detail we need some mathematical background. The words node and variable are from now on interchangable. The notiations, definitions, lemmas and theorems (given without proof) in this section are quoted verbatim from [TBA].

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.

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 $P(\mathbf{Z} = \mathbf{z}) > 0$,

$$P(X = x, Y = y|\mathbf{Z} = \mathbf{z}) = P(X = x|\mathbf{Z} = \mathbf{z})P(Y = y|\mathbf{Z} = \mathbf{z}) \tag{2.1}$$

or

$$P(X, Y|\mathbf{Z}) = P(X|\mathbf{Z})P(Y|\mathbf{Z}) \tag{2.2}$$

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

Definition (Bayesian network)

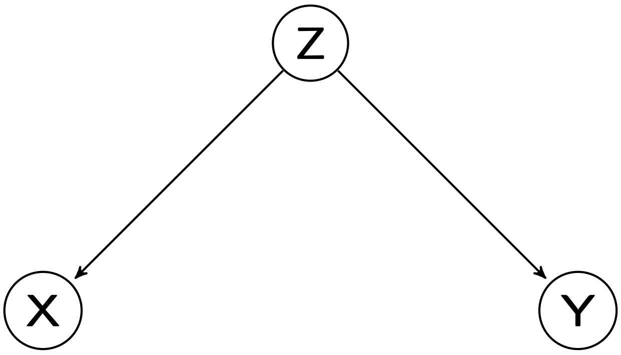
Let P be a discrete joint probability distribution of the random variables in some set $\mathcal{V}$ and $\mathcal{G} =< \mathcal{V}, E >$ be a Directed Acyclic Graph (DAG). We call $< \mathcal{G}, P >$ a (discrete) *Bayesian network* if $< \mathcal{G}, P >$ satisfies the Markov condition.

Definition (Markov condition)

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

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 2 that this graph is a Bayesian network and with section 2 we have: $P(X, Y|\mathbf{Z}) = P(X|\mathbf{Z})P(Y|\mathbf{Z})$.

Definition (collider)

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

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]).

Definition (d-seperated)

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*.

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.

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*.

Theorem

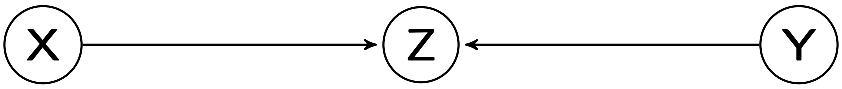
In a faithful Bayesian network $< \mathcal{G}, P >$ the following equivalence holds ([P88])

$$Dsep_{\mathcal{G}}(X;Y|\mathbf{Z}) \Longleftrightarrow Ind_P(X;Y|\mathbf{Z}) \tag{2.3}$$

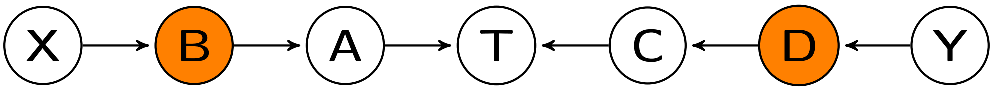
Remark and Explanation

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

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



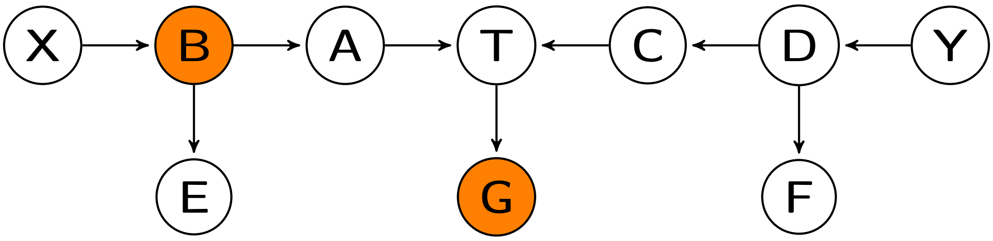
Here Z is a *collider* because it has two incoming edges. In this case if we just look for $Ind_P(X;Y|\{\})$, the path between X and Y would be blocked and for this X and Y are *d-seperated*. If we look for $Ind_P(X;Y|\mathbf{Z})$ with collider $\mathbf{Z}$ and obviously $\mathbf{z} \in \mathbf{Z}, \forall \mathbf{z} \in \mathbf{Z}$, then this path is not blocked and we state that X and Y are *d-connected*.
Because of section 2 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. We want to give you two other examples for d-seperation.



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.

So there is no element which blocks the path between A and Y.
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.

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

$$MinAssoc(X;T|\mathbf{Z}) = \min_{\mathbf{S} \subseteq \mathbf{Z}} Assoc(X;T|\mathbf{S}) \tag{2.4}$$

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

1. In Figure 4.1 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):

$$Ind(X;T|\mathbf{Z}) \iff (Assoc(X;T|\mathbf{Z} = 0)). \tag{2.5}$$

From now we denote $Ind_P(X;Y|\mathbf{Z})$ as $Ind(X;Y|\mathbf{Z})$.

Because this algorithm is based on conditional independence testing, i.o.w. measuring the strength of association between two variables, we need some formulas for our 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:

Lemma (G^2 value)

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]):

$$G^2 := 2 * \sum_{a,b,c} S_{ijk}^{ab\mathbf{c}} * ln \left(\frac{S_{ijk}^{ab\mathbf{c}} * S_k^{\mathbf{c}}}{S_{ik}^{a\mathbf{c}} * S_{jk}^{b\mathbf{c}}} \right), \tag{2.6}$$

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

$$df = (|D(X_i)| - 1)(|D(X_j)| - 1) \prod_{X_l \in \mathbf{X_k}} |D(X_l)|, \tag{2.7}$$

where $D(X_i)$ is the domain (number of distinct values) of variable X_i.

Remark

section 2 is the basis of our calculations. To know whether two variables X and Y given a set **Z** are conditionally independent, we run this statistical test and seek if it is likely that they are independent or not. In the next chapter you get an introduction on how we realized this. To complete this chapter we give you the definition of the Bayesian Dirichlet likelihood-equivalence uniform (BDeu) score. It is also a statistical test which later helps us to set the edges in our graph. After those tests we observe the graph which is most likely to be the right graph depending on the data. For that we quoted verbatim from [Ca06].

The BDeu score

The Bayesian Dirichlet likelihood-equivalence uniform score is defined as:

$$g_{BDeu}(D,G) = \sum_{i=1}^{n} \left[\sum_{j=1}^{q_i} \left[\log \left(\frac{\Gamma(\frac{\eta}{q_i})}{\Gamma(N_{ij} + \frac{\eta}{q_i})} \right) + \sum_{k=1}^{r_i} \log \left(\frac{\Gamma(N_{ijk} + \frac{\eta}{r_i q_i})}{\Gamma(\frac{\eta}{r_i q_i})} \right) \right] \right], \tag{2.8}$$

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.

Explanation

This formula takes as an input a graph and the observational data, it returns a score. Later, we are seeking for the graph with the highest score. To compute the sums we have to know how to get all the values explained above. Here is a short explanation of those variables in the formula:

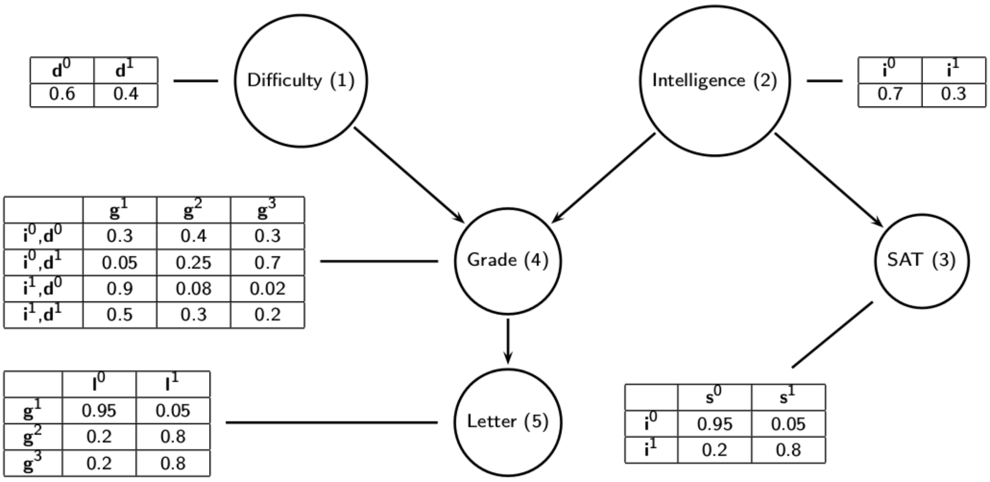
- **eta**: We set this value to 1 - the most common setting. There is no rule which value is the best and one could find many papers which try to find the "best" eta. If you are interested in that we recommend you [SKM].
- **n**: The number of variables in our BN.
- $\boldsymbol{r_i}$: How many states the $i-th$ variable can take.
- $\boldsymbol{q_i}$: 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 then $q_X = r_y \cdot r_z$.
- $\boldsymbol{w_{ij}}$: 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: $(0;0), (0;1), (1;0), (1;1)$.
- $\boldsymbol{N_{ijk}}$: Counts how often a variable X_i takes the value x_{ik} and its parents take the configuration w_{ij}.
- $\boldsymbol{N_{ij}, N_{ik}}$: Is the calculation of the sums over the depending N_{ijk} values from above.

3 Explaining the max-min-hill-climbing algorithm

In this chapter we want to introduce an example first. By this we will explain the concept behind the algorithm. We also used this example for our computations and for comparison with the bnlearn implementation.

3.1 Introduction to an example

We built a data set whose underlying structure is taken from [KoFr]. It is a BN with 5 nodes. In this we have that the intelligence of a student (high or low) and difficulty of an exam (difficult or easy) affect the students grade (good, middle, bad). The intelligence also affects the SAT (the european equivalence to this is the PISA test), which can be good or bad. The professor′s letter of recommendation can then be good or bad. The last variable depends on the grade. In the following picture also the probabilities of an event are displayed, where the smallest upper case numbers stand for high, good and easy and the higher numbers for low, bad and difficult.



d^0	d^1
0.6	0.4

i^0	i^1
0.7	0.3

	g^1	g^2	g^3
i^0,d^0	0.3	0.4	0.3
i^0,d^1	0.05	0.25	0.7
i^1,d^0	0.9	0.08	0.02
i^1,d^1	0.5	0.3	0.2

	l^0	l^1
g^1	0.95	0.05
g^2	0.2	0.8
g^3	0.2	0.8

	s^0	s^1
i^0	0.95	0.05
i^1	0.2	0.8

Figure 3.1: The graph of our example with 5 nodes and some conditional independencies.

In science you normally do not start with a graph, reconstruct data and then again reconstruct the graph. You start with (random) data and try to find out which dependencies you find within your data. For our purpose this example helped to check if the code we wrote is valid. Because of randomness in the data the output can vary. If we talk about getting the "right" graph, we mean that it looks like our example. By statistical testing there is no right or wrong.

3.2 Computing the *MMHC*

The max-min-hill-climbing algorithm generally takes your underlying data and firstly finds the skeleton of the graph. Since, it knows the structure of the graph it starts to add edges between nodes. It also reverses existing edges or deletes them, until it finds the graph which is most likely to fit best to the data.

Finding the skeleton (*MMPC*)

Finding the skeleton is the first part of the algorithm. This search is constraint-based and is called the max-min parents and children algorithm (*MMPC*). The name refers to the fact that we find for a specific variable T (target) all variables which are conditionally dependent on T. All we then know is that those variables can be a parent or a children of T.
We start with an empty graph and iterate over all variables. In each step - a target T is selected

- we try to find all the variables which are conditional dependent on T. We now assume - depending on our example - that the algorithm picked $T = "grade"$ to find its parents and children.

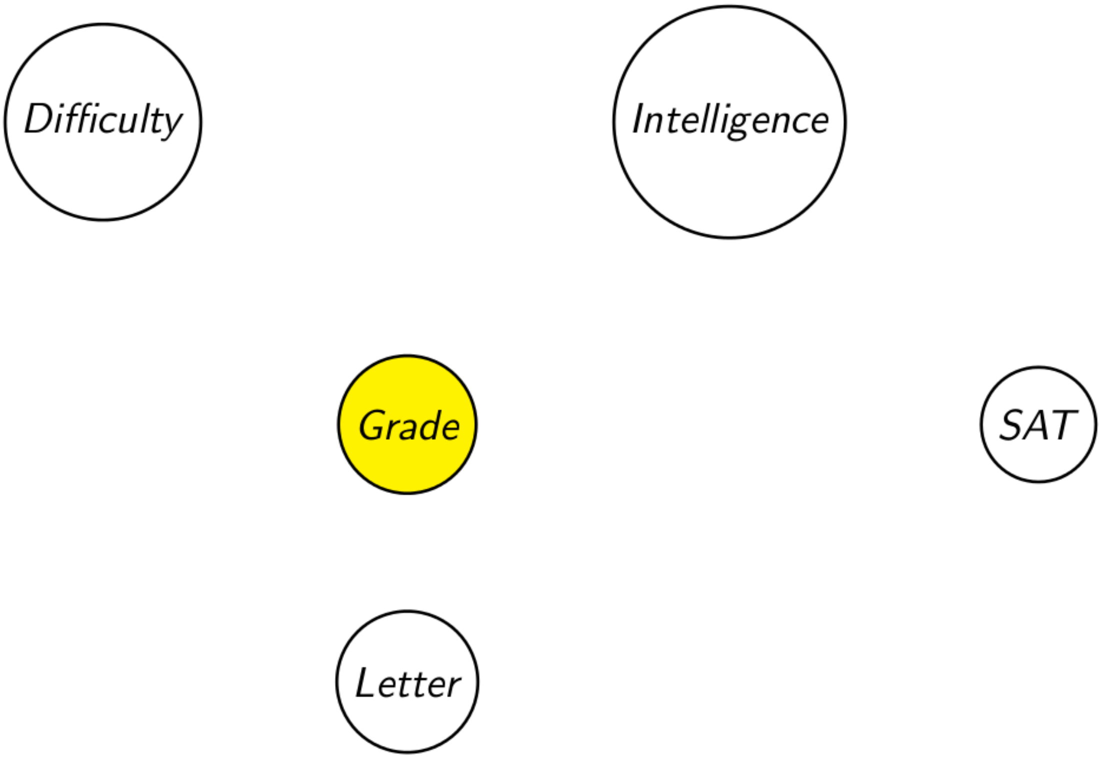


Figure 3.2: The first iterations step where "grade" is selected.

To find the parents and children (**PC**) we first test conditional independence of T with all nodes $\mathcal{V} = \mathcal{D} \setminus \{"grade"\}$, where $\mathcal{D}$ is our observational data. Since **PC** $= \emptyset$ we are seeking for $Ind(T; X|\emptyset)$ for all $X \in \mathcal{V}$.
The procedure of the *MMPC* algorithm then works as follows:

- Compute the G^2 value.
- Compute the degrees of freedom df.
- Calculate the *pvalue* - we used the $pchisq(G^2, df)$ in Rcpp.
- Test if the *pvalue* is smaller than a threshold of 0.05, if yes then keep this *pvalue*, it's corresponding G^2 and X. If it is bigger than 0.05 then you know that T and X are conditionally independent and X can be crossed out from $\mathcal{V}$, i.e. you don't have to consider this X in your calculations again.
- At the end let the X join the **PC** set which has the smallest *pvalue*.

- If the *pvalues* of two variables are equaled then take the variable with the higher G^2 value.
- If X joined **PC** then it can be crossed out from $\mathcal{V}$, since you already know that X and T are conditionally dependent.
- Repeat this calculation but now with all subsets of **PC**. Terminate the algorithm when $\mathcal{V} = \emptyset$.

Remark: From section 2 we also know that two independent variables have an association value of zero. Later when we discuss the pseudo code, we need to compute the association value which is nothing but the G^2 value, if the *pvalue* is <0.05.

To come back to our example we assume that the strongest association between "grade" was found for "difficulty" and we found out that "grade" and "SAT" are conditionally independent. Then "difficulty" joins the **PC** set (**PC** $= \{"difficulty"\}$) and we won't consider "SAT" for the next calculations, i.e. $\mathcal{V} = \{"intelligence", "letter"\}$.

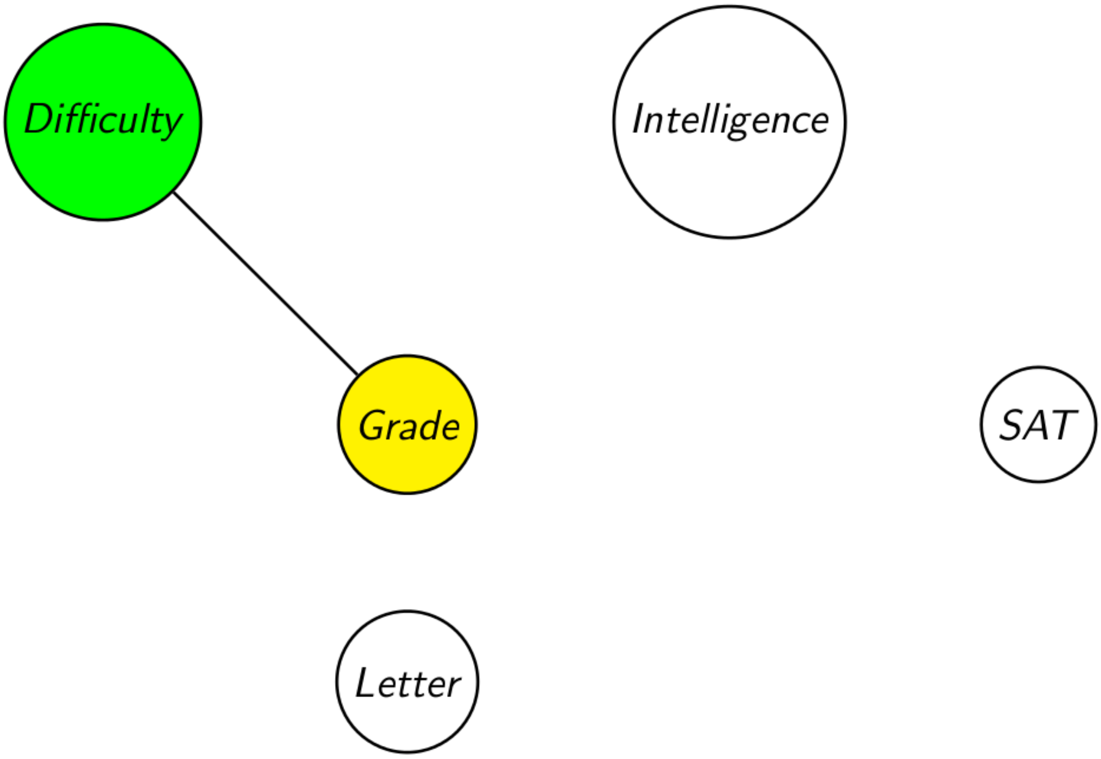


Figure 3.3: Here "difficulty" joined **PC** and is a parent or a children of "grade".

Despite the fact that we have to consider all subsets of the **PC** set, which are $\emptyset$ and "difficulty", we only have to test $Ind("grade"; "intelligence" | "difficulty")$ and $Ind("grade"; "letter" | "difficulty")$.

Since we used $\emptyset$ in our calculations one step before, we can save and use those values in this step again.
We now assume that the stronger association lies between "grade" and "intelligence". Since there is an association between "grade" and "letter", too, i.e. they are not conditionally independent, we have: $\mathcal{V} = \{"letter"\}$ and $PC = \{"difficulty", "intelligence"\}$.

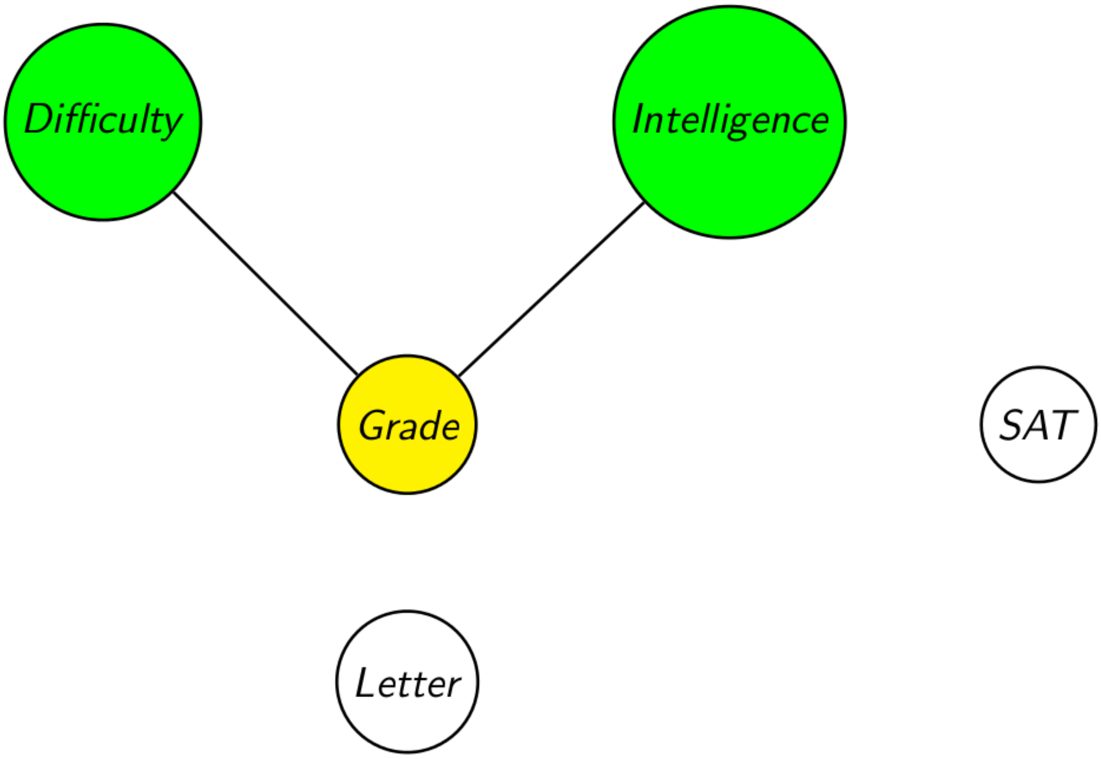


Figure 3.4: Here "intelligence" joined **PC** and is a parent or a children of "grade".

Now we have to calculate the association given all subsets of **PC**, i.e. We test $Ind(T; X|Z), \forall X \in \mathcal{V}, Z \in \mathbf{PC}^2$ where $\mathbf{PC}^2$ is the power set of **PC**. The bigger **PC** gets, the more calculations we have to do. Some of those calculations in the power set of **PC** are done before (e.g. $Ind("grade"; "letter"|"difficulty")$ c.f. above), so we just saved the corresponding values and used it again. We now assume that "letter" also joined our set and we terminate the algorithm. Now we have all possible parents and children of "grade", which we already knew.

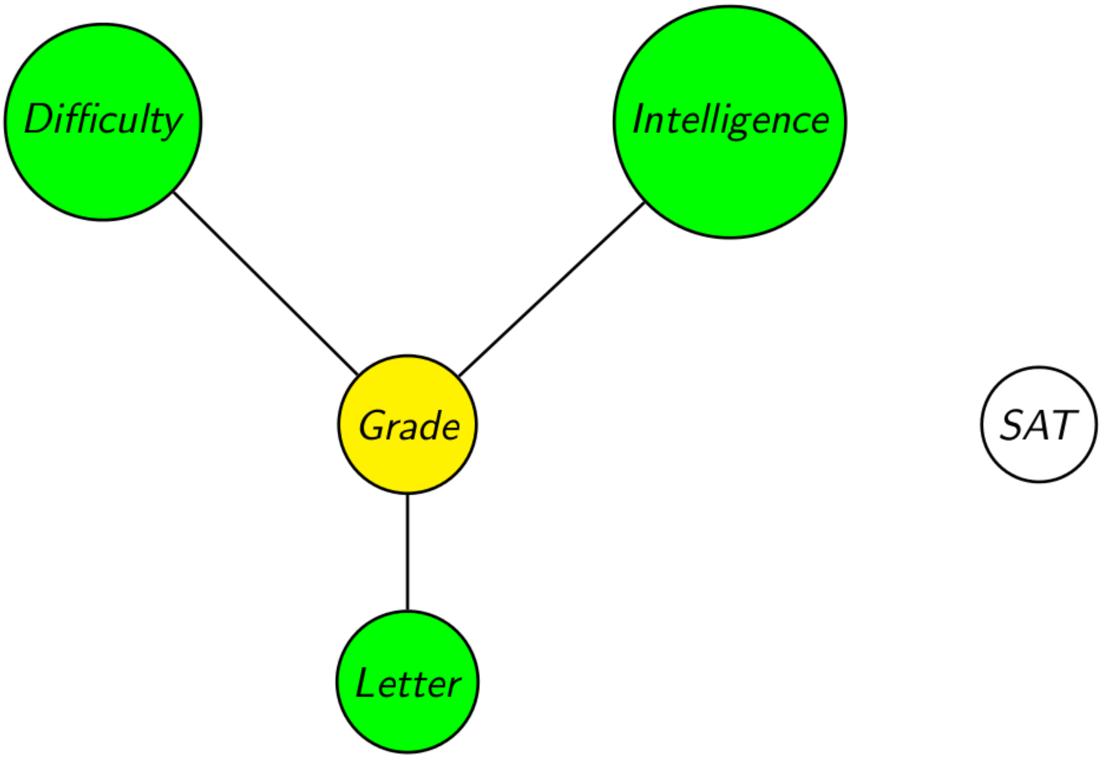


Figure 3.5: Here "letter" joined **PC** and is a parent or a children of "grade".

It may happen that there are false positive variables in the **PC** set of a target variables **T**. For this reason we have to check if the parents/children in $\mathbf{PC}_T$ really belong to our selected variable **T**. The relation between **T** and its parents/children is symmetric. That means for a target **T** and $\mathbf{A} \in \mathbf{PC}_T$ it follows:

$$\mathbf{T} \longleftrightarrow \mathbf{A} \Longleftrightarrow \mathbf{A} \longleftrightarrow \mathbf{T} \tag{3.1}$$

for a target **A** and $\mathbf{T} \in \mathbf{PC}_A$. Since this relation holds we have to check in a second step if this symmetrie is fullfild. For that we check for every $\mathbf{X} \in \mathbf{PC}_T$ if $\mathbf{T} \in \mathbf{PC}_X$.

As we have finished all those calculations we have the structure/the skeleton of our graph.

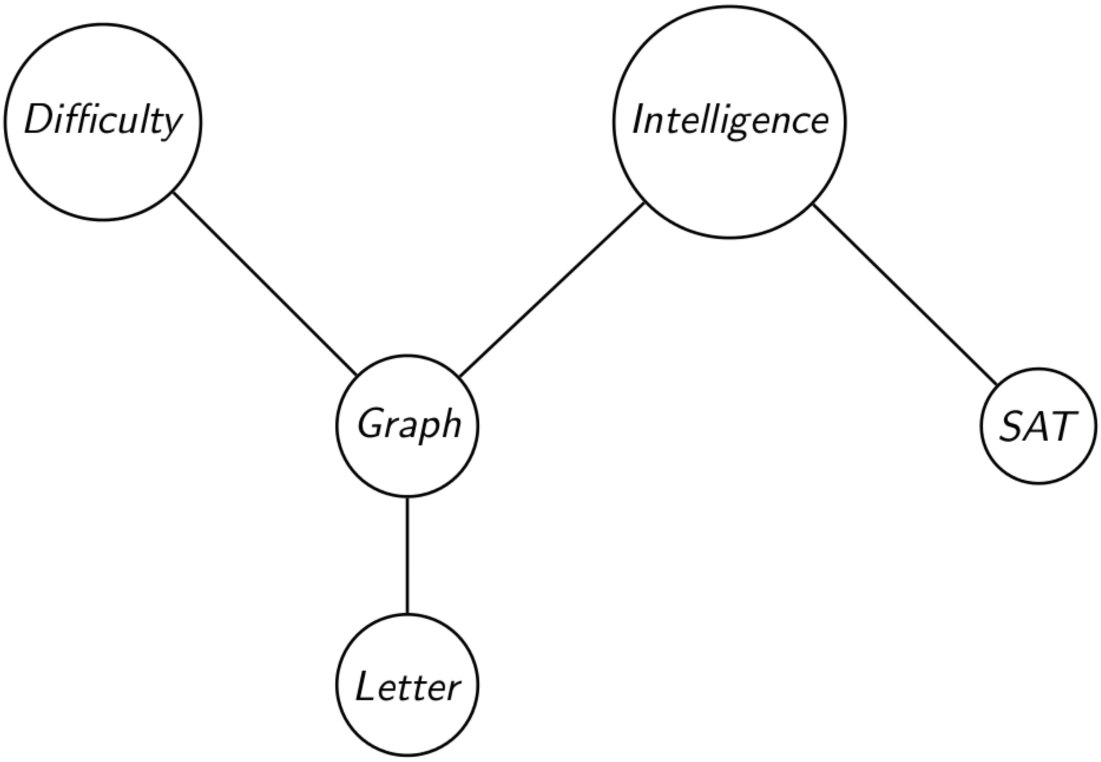


Figure 3.6: The skeleton we should observe after our calculations.

Though we now the structure and that there are connection we do not know how they affect each other. For this we have to direct the edges which is done by the BDeu score.

3.2.1 Computing the $BDeu$ score

The Bayesian Dirichlet likelihood-equivalence uniform score takes as seen in section 2 the data set $\mathcal{D}$ and a graph G as an input. What we do in one iteration step is:

- Calculate the $BDeu$ score of the current graph. In the first iteration score the empty graph, i.e. the graph without any edges.
- Add an edge randomly between two conditional dependent nodes X and Y. From $MMPC$ we already know those variables which could have a connection.
- Calculate the $BDeu$ score with the new set edge.
- If the score is higher than before, the edge stays, else the edge is removed. This is the greedy search part. We are seeking for an local optimum.

- Do this as long as the score stays in its (possible) maximum for at least 10 times in a row.
- Also apply reversing edges and deleting edges to the possible operations on the graph during one iteration step.

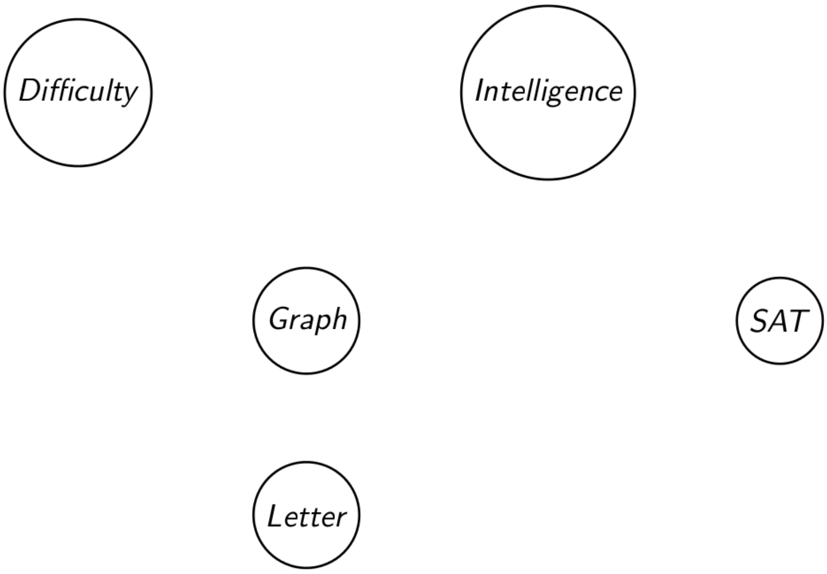


Figure 3.7: Starting with the empty graph.

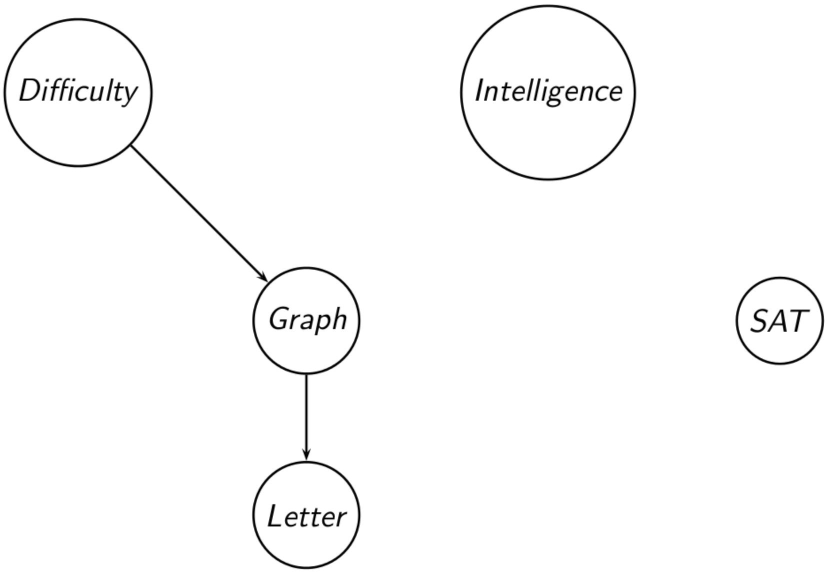


Figure 3.8: Adding edges between nodes.

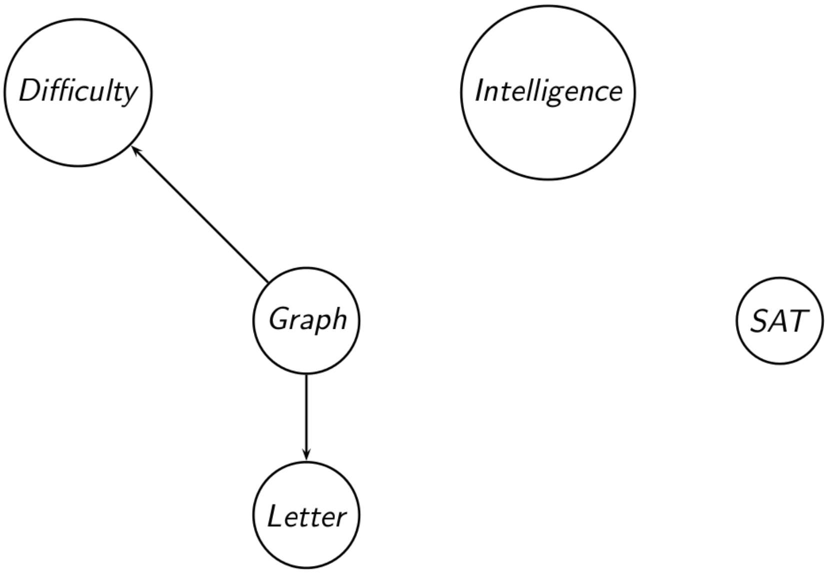


Figure 3.9: Reverse an edge.

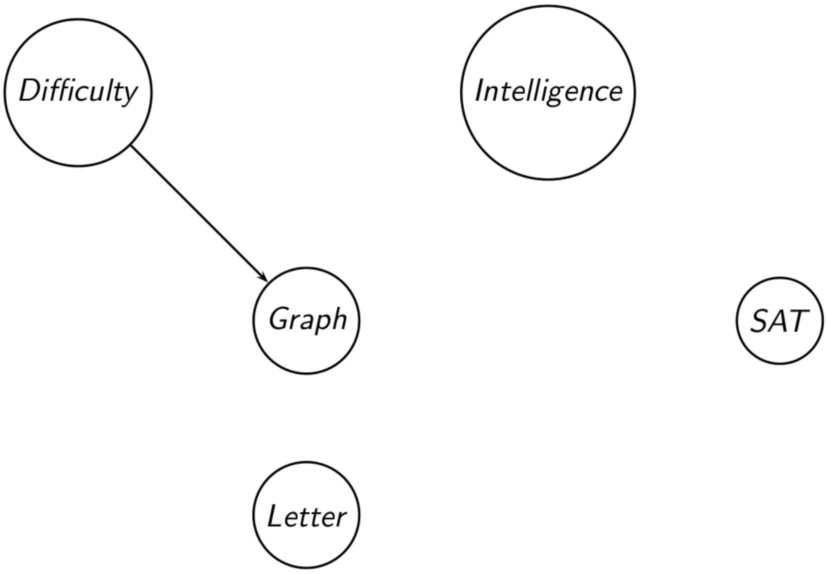


Figure 3.10: Reversed and deleted edge.

4 The max-min-hill-climbing algorithm

As we stated before our main aim was to beat the existing algorithm provided by the bnlearn package in R, which is implemented in C. There are a lot of bottlenecks which slow the code down. To have the chance to beat an existing algorithm which runs quite fast, we needed to optimize the code as much as possible. For that we first want to present the pseudo code of the algorithm to be able to talk about optimization.

```
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 4.1: 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 between two nodes in the skeleton graph. We first discuss the *MMPC* (max-min parents and children) function which is executed with 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 (*BDeu* score).

4.1 max-min parents and children (MMPC))

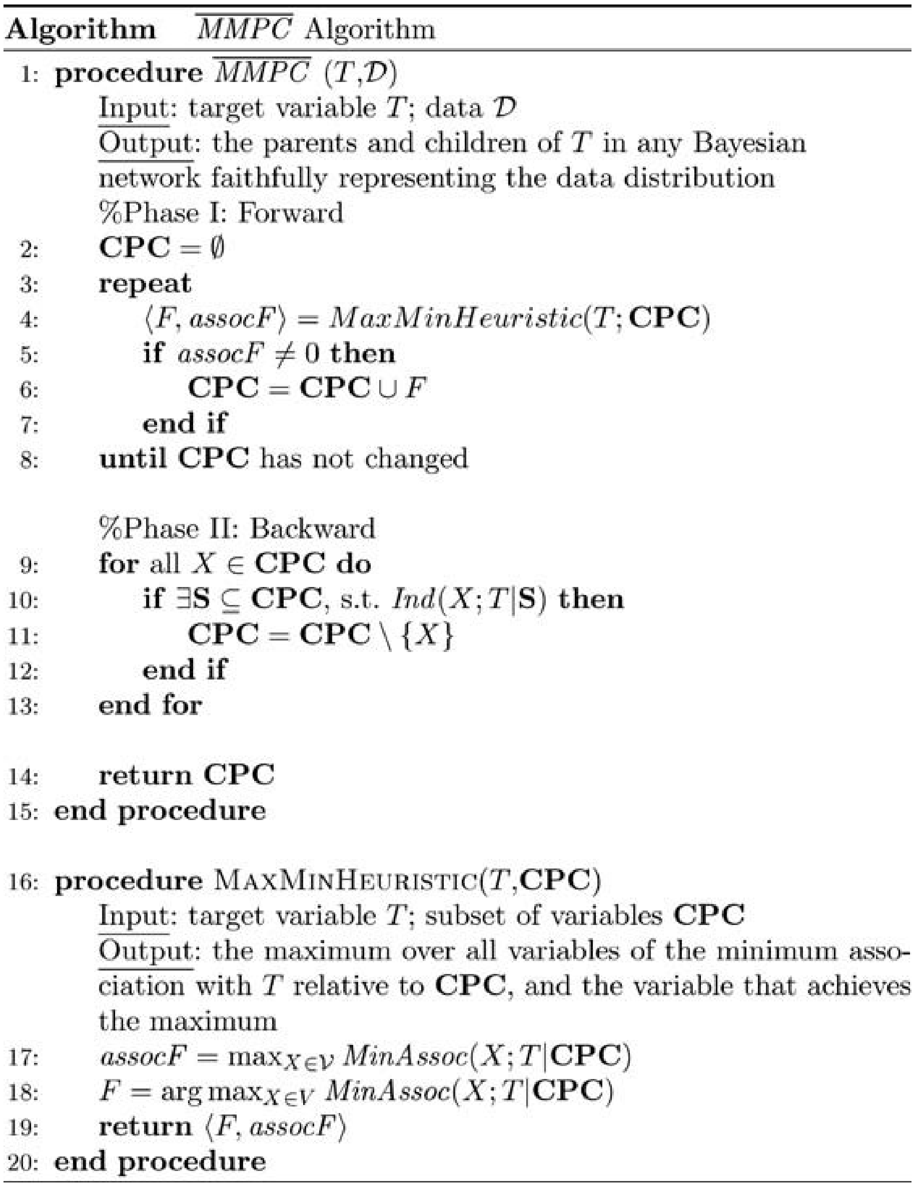
The max-min parents and children (MMPC) is the algorithm which reconstructs the graph skeleton from data.

```
Algorithm    Algorithm $MMPC$
1: procedure MMPC(T,$\mathcal{D}$)
2:     CPC = $\overline{MMPC}(T, \mathcal{D})$
3:     for every variable $X \in$ CPC do
4:         if $T \notin \overline{MMPC}(X, \mathcal{D})$ then
5:             CPC = CPC $\setminus X$
6:         end if
7:     end for

8:     return CPC
9: end procedure
```

Figure 4.2: 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 4.1 this function - including the *MaxMinHeuristic* function - is shown. Figure 4.1, 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 3.1, 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 for the target **T**. The more interesting part here is the $\overline{MMPC}$ function.



```
Algorithm  $\overline{MMPC}$ Algorithm
procedure $\overline{MMPC}$ $(T,\mathcal{D})$
    Input: target variable $T$; data $\mathcal{D}$
    Output: the parents and children of $T$ in any Bayesian
    network faithfully representing the data distribution
    %Phase I: Forward
    $\mathbf{CPC} = \emptyset$
    repeat
        $\langle F, assocF \rangle = MaxMinHeuristic(T; \mathbf{CPC})$
        if $assocF \neq 0$ then
            $\mathbf{CPC} = \mathbf{CPC} \cup F$
        end if
    until CPC has not changed

    %Phase II: Backward
    for all $X \in \mathbf{CPC}$ do
        if $\exists \mathbf{S} \subseteq \mathbf{CPC}$, s.t. $Ind(X; T|\mathbf{S})$ then
            $\mathbf{CPC} = \mathbf{CPC} \setminus \{X\}$
        end if
    end for

    return CPC
end procedure

procedure MaxMinHeuristic$(T,\mathbf{CPC})$
    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
    $assocF = \max_{X \in \mathcal{V}} MinAssoc(X; T|\mathbf{CPC})$
    $F = \arg\max_{X \in \mathcal{V}} MinAssoc(X; T|\mathbf{CPC})$
    return $\langle F, assocF \rangle$
end procedure
```

Figure 4.3: 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 one interesting procedure: the *MaxMinHeuristic*.

4.1.1 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**. This is done via the G^2 statistic as mentioned before.

4.2 The scoring function

The scoring works as we stated earlier in this report. The important thing here is that we only add an edge from X to Y iff Y is a member of the parents/children set of X. In the case of scoring the most interesting variable is the eta. There is no "perfect" value one could take. There is a lot of research done to find the optimal eta for an algorithm. In our case there are two possibilities: we could set $eta = 1$ or we could calculate it with $eta = \frac{1}{N} \sum_{i=1}^{N} |X_i|$ where N is the number of variables and $|X_i|$ is the cardinality of the variable X_i. In our tests it did not affect the results or the running time. For that reason we decided to set it to one.

4.3 Computational optimization

We decided to write our algorithm object oriented. If we instantiate our class, the constructor of the class will do some precomputations like getting the cardinality of the variables. Since the class holds all information the algorithm needs, we have our information in every method of our class available. This has the effect that we do not pass the objects (vectors, lists, etc.) from function to function. We work with them right in our class.

Another thing which is important to know is, that the underlying data matrix is stored differently in R and C++. R saves a matrix firstly by column and then by row, i.e. you have a pointer to a column which holds the pointer to the row. In C++ it is vice versa. We had to change the normal way of iterating over matrices in C++ - from first row then column to first column then row.

To compute the G^2 you need to count the $S_{ijk}^{ab\mathbf{c}}$, etc. values. For this we implemented a special application: hash tables. At first side we tried to use the unordered_map of C++. This is a hash table within you can look up your elements in constant time. We precomputed all possible combinations of our variables, i.e. we tested $Ind(X;Y|\mathbf{Z})$ for all possible X, Y and sets $\mathbf{Z}$. Our idea was that we only need to look up the values we need in our methods. Of course, this did not improve running time since you had to compute all permutations. With 5 variables and all possible $X, Y, \mathbf{Z}$ you have $2! + 3! + 4! + 5! = 152$ calculations.

We then decided to use n-dimensional arrays hash tables where we convert the values of a variable into integer from $1, ...n$ where $n = |X|$ are the possible states of one variable. We then used those integer values as indices for our arrays and increment the value of the array everytime it is accessed. Afterwards we just needed to iterate over the array again an use the values in the arrays for our calculations. The biggest array we allocate is 5-dimensional. For small problems with less nodes in the graph and less dependencies within those nodes this code runs extremely fast. For bigger problems there also exists an implementation to compute the G^2 value, but it is quite slow and we try to avoid using it. The difficulty behind

our arrays is that we use pointer arrays - otherwise we could not allocate 5-dimensional arrays. Here we needed to be careful to free the memory after usage.
To present the validity of our code we recall the example from the previous section. For this example 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 time our algorithm needs to compute the skeleton of the graph, i.e. using the *MMPC* function, is as follows:

```
df <- student(10000)
bm <- benchmark(C$mmpc(),
                columns = c("test", "elapsed", "relative"),
                replications = 1)

             test elapsed relative
1         C$mmpc()   0.043    1.000
```

where the elapsed time is measured in seconds. We see that our implementation with only 0.043 seconds of running time is extremely fast, even for a high number of observations. In the next chapter we will see if this time is good enough to be faster than the bnlearn function. By looking at the whole algorithm we have the following running time:

```
df <- student(10000)
bm <- benchmark(C$mmhc()
                columns = c("test", "elapsed", "relative"),
                replications = 1)

                          test elapsed relative
1                     C$mmhc()   0.371   1.000
```

again with the elapsed time in seconds. As we can see the most time we lost was in the *BDeu* scoring. Till the end of this work we did not find a way to make this function faster. If you are interested in the progress we recommend you to follow our Github account, stated at the end of this report.
The "proof" of the validity now follows from the plot we present. Here you can see that the graph looks like the one we started with. To plot our results we used the "igraph" R-package.

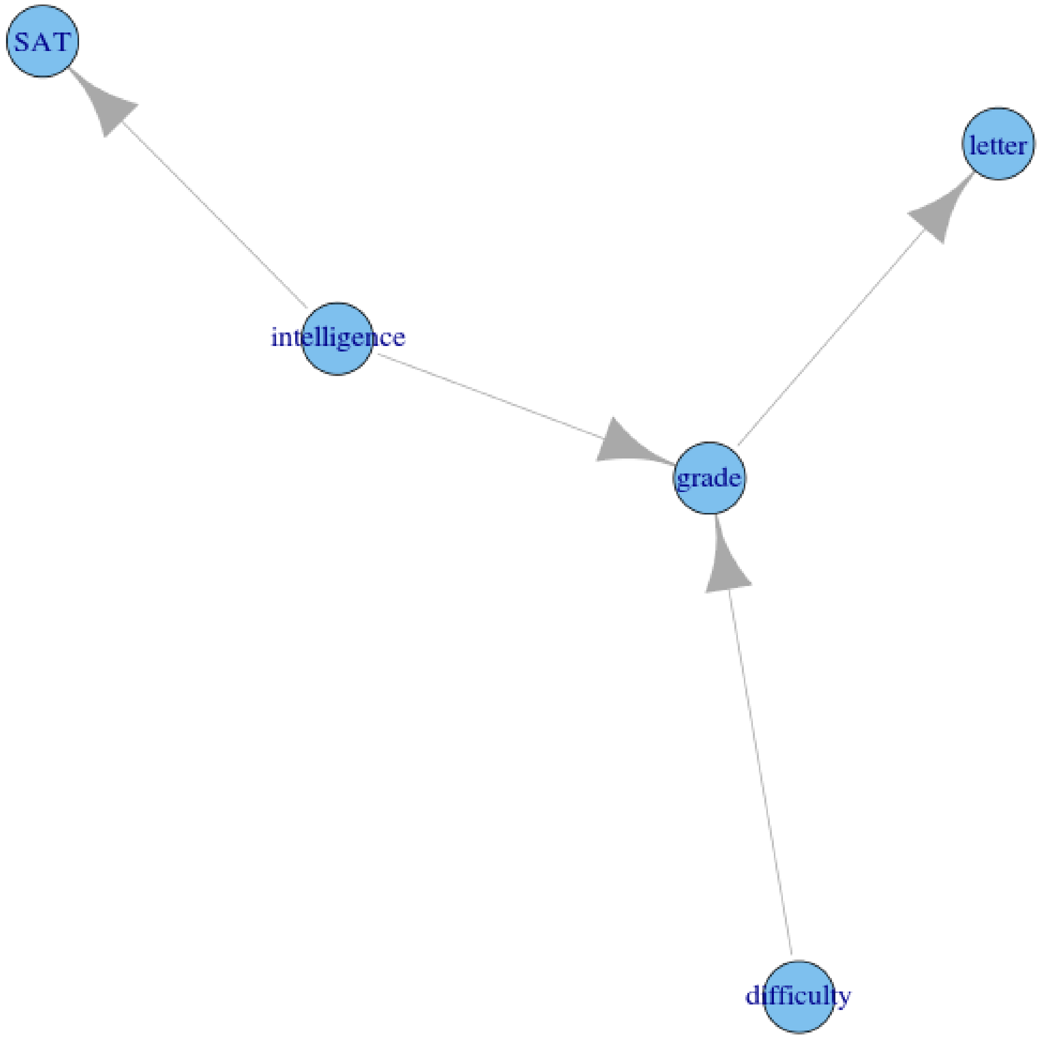


Figure 4.4: The graph returned looks like the one we started with.

5 A comparison to the bnlearn package

Beating the bnlearn was a very challenging and difficult task. With some tricks and computational knowledge we were able to implement an algorithm which is fast and valid. The question now is, if our version is faster then the one of bnlearn and the answer is no. In some cases we were faster, but as the amount of data or the amount of observations grows we did not have a chance to be faster. As we will see our running time is also quite good and developing goes on. For the discussion we consider our example from section 3.1. First of all we want to compare the running times of the *MMPC* function. For that we run our implementation and the one of bnlearn with the example having 1,000, 5,000 and 10,000 numbers of observations. For the rest of this section we measure the elapsed time in secondes. The R code looks as follow, where we used the R package "rbenchmark" to be able to run the benchmark function:

```
MMPC <- function(myDataFrame) {
    C <- new(MMHC, myDataFrame)
    C$mmpc()
}

myDataFrame <- student(1000)
C <- new(MMHC, myDataFrame)

bm <- benchmark(MMPC(myDataFrame), mmpc(myDataFrame),
                    columns = c("test", "elapsed", "relative"),
                    replications = 1)
```

The parameter of the *MMPC* function is a R data frame. This function allocates our class *C*. The method *mmpc*() is executed. To have access to the **PC** set we would need to type *C*\$*pc*() to the console. This method returns the **PC** set. If we compare our implementation with the bnlearn function with 1,000 observations we get:

```
               test elapsed relative
```

```
2 mmpc(myDataFrame)   0.018    2.571
1 MMPC(myDataFrame)   0.007    1.000
```

If we repeat this for 5,000 observations we have:

```
               test elapsed relative
2 mmpc(myDataFrame)   0.029    1.074
1 MMPC(myDataFrame)   0.027    1.000
```

and for 10,000 observations:

```
               test elapsed relative
2 mmpc(myDataFrame)   0.021    1.000
1 MMPC(myDataFrame)   0.050    2.381
```

As we can see, our implementation is much faster for a smaller amount of data. The bigger the data gets, the slower our code runs. The interesting thing is, that the bnlearn package seems to have constant running time. Another thing is that the tests this algorithm uses are not exactly the same as ours. We tried to set the parameters of the function rightly but it did not work with our data (setting test to "x2" and score to "bde"). This seems to be an advantage of our implementation since it has no problem with the data. We will discuss another example later, where setting those values was possible.

We want to show you another plot where we computed the benchmark of both implementations using the data with 1,000 to 20,000 observations in steps of 1,000 (, i.e. 20 benchmarks).

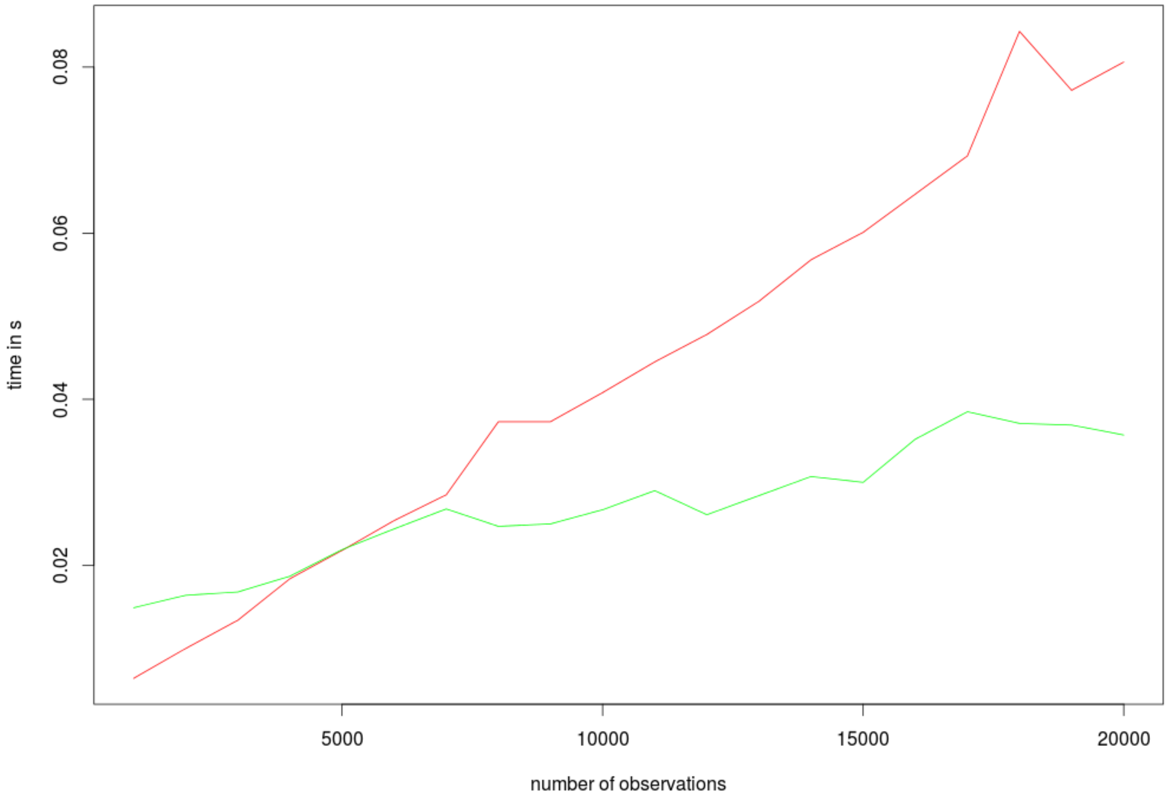


Figure 5.1: From 1,000 to 7,500 observations our algorithm (red line) is quite fast. Above this number bnlearn one (green line) is much faster.

It seems that the break even point of those to implementations lies at about 7,500 numbers of observations. At this point our algorithm starts to be slower than the other one.

Now we want to present the comparison of the whole *MMHC* algorithms. You may ask why we do not compare the scoring functions with each other. The reason is, that we can not run the scoring explicitely in the bnlearn function - only within our package. This seems to be a second advantage of our implementation, because we seperated our calculations. But before we compare the running time of the *MMHC* function we want to present again the R code where *C* is again a class:

```
MMHC <- function(myDataFrame) {
    C <- new(MMHC, myDataFrame)
    C$mmpc()
    C$mmhc()
}
```

```
myDataFrame <- student(1000)
C <- new(MMHC, myDataFrame)

bm <- benchmark(MMHC(myDataFrame), mmhc(myDataFrame),
                    columns = c("test", "elapsed", "relative"),
                    replications = 1)
```

The running time varies a lot because our loop is terminated if the score did not increase at least 10 times in a row. If it did not change 9 times in a row, but then changes the counter starts from zero again. This means we apply all our operations to the graph quite often if we do not find the best fit with our data after the first computations. So for the comparison we took the most common outcomes to compare:

```
                test elapsed relative
2  mmhc(myDataFrame)   0.024    1.000
1 MMHCs(myDataFrame)   0.038    1.583
with 1,000 observations
```

```
                test elapsed relative
2  mmhc(myDataFrame)   0.039    1.000
1 MMHCs(myDataFrame)   0.228    5.846
with 5,000 observations
```

```
                test elapsed relative
2  mmhc(myDataFrame)   0.068    1.000
1 MMHCs(myDataFrame)   0.335    4.926
with 10,000 observations
```

As we mentioned, the time often varies. For that we also plotted a line graph where we took our data with a number of observations from 1,000 to 10,000 in steps of 1,000 (, i.e. 10 benchmarks). This time we did those computations 10 times and then took the mean of the every outcome of the benchmarks.

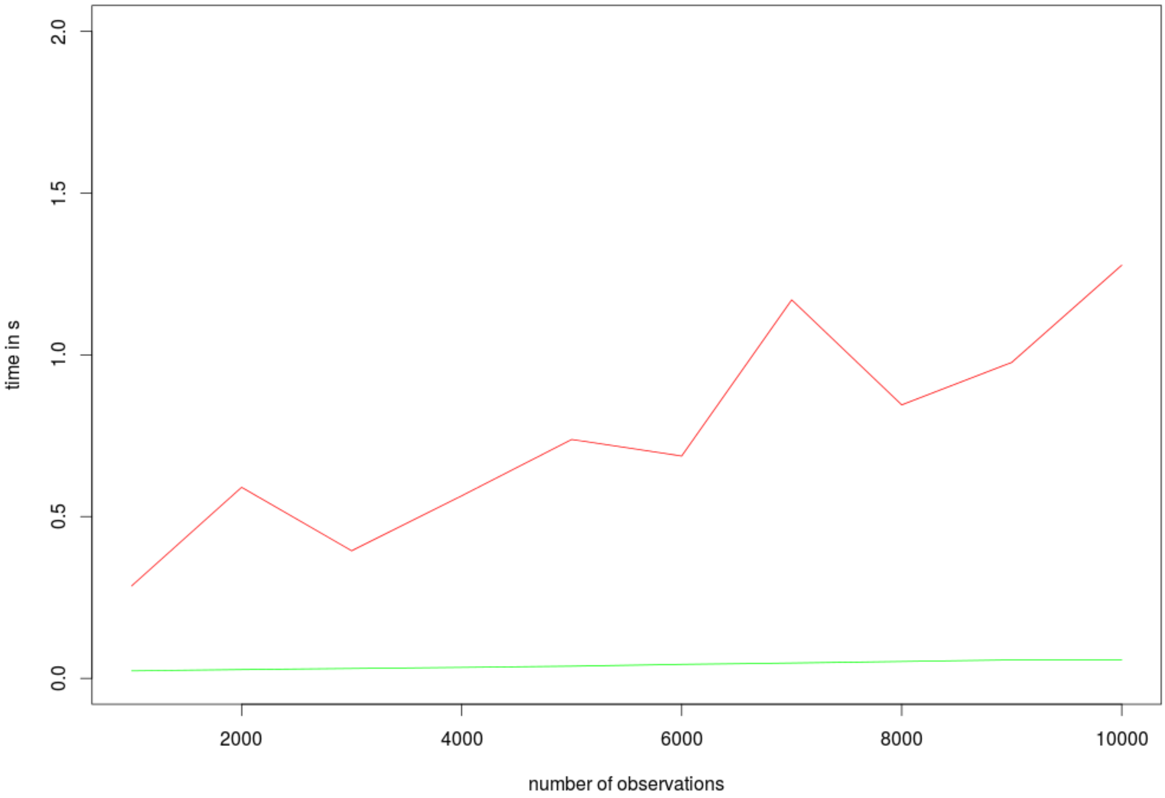


Figure 5.2: Hear you see that our algorithm (red line) grows linearly where as the bnlearn one (green one) is constant.

As the running time of our code grows linearly the bnlearn package seems to stay about constant. We could not figure out how and why the bnlearn package has no growth in its running time.
But our running time is maximally 1.5 seconds (at average) for a huge data frame. This is still a very good outcome. Maybe in the next months we can improve our algorithm depending on running time for the scoring function.

To the end of this section we want to present a last comparison between the two implementations. Here we were able to use the exact same algorithms. The underlying data is called "learning.test" and is from the bnlearn package. Normally it does not return a complete DAG (one edge without an arrow should be in ther), but for our testing purpose this was no problem. It is a data frame which holds observational data of 6 nodes with 5,000 observations. We first want to compare the *MMPC* algorithms again and then the *MMHC*. For the max-min parents and children we get:

```
                               test elapsed relative
1                            C$mmpc()   0.025    1.136
2 mmpc(learning.test, test = "x2")   0.022    1.000
```

And again we see that our *MMPC* implementation is approximately as fast as the bnlearn function. But the scoring will slow the code down for this example. We took the average time elapsed for comparison after computing the benchmark 100 times:

```
                 test elapsed relative
1            C$mmhc()   0.323    8.500
2 mmhc(learning.test)   0.038    1.000
```

We saw that the main work which has to be done in the next months is to find a better and more performant implementation for the scoring part of the algorithm. But we are pretty sure, that we will be able to improve running time. To this end, we want to present the graph which is produced using the "learning.test" data. We present both: our outcome and the outcome of the bnlearn package.

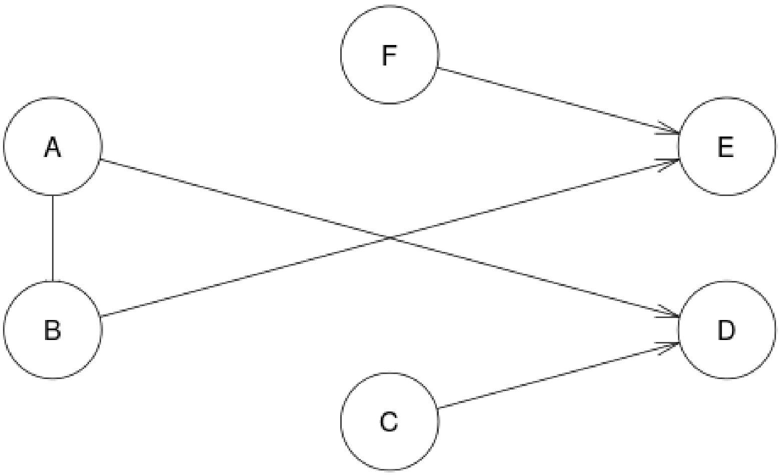


Figure 5.3: The graph of the data frame "learning.test" where one connection is not directed.

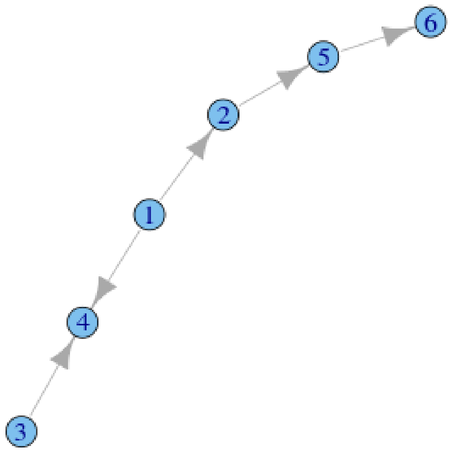


Figure 5.4: The graph of the data frame "learning.test" where all connections are directed.

6 Conclusion

Despite the fact that we could not be faster than the bnlearn package, it was worth to implement this algorithm. In the future there will be the chance to improve running time and make our algorithm faster. But for now we provide the opportunity to construct Bayesian Networks from a smaller amount of data with an unreached running time. We really hope that research goes on and maybe there will be an implementation in the future (maybe we can do that) or even a new algorithm which reconstructs Bayesian Networks and runs faster than all existing ones.

Acknowledgment

There are several people I like to thank. Since this is not a published paper I did not ask if I am allowed to mention those people, but I am pretty sure that they are happy about it.

- My girlfriend who is always there for me.
- Markus Hofstetter to whom I could talk about this project a couple of times and getting new ideas.
- Tomi Silander: a scientist who not only answered an e-mail within 30 minutes, who also gave me a lot of input and help.
- Prof. Dr. Rainer Spang who always has a sympathetic ear, not only for me also for other students.
- And last but of course not least Dr. Giusi Moffa who supported me even in times were she was already gone to Basel. Thanks a lot for being so patient with me and explaining complicated stuff in an easy way.

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

[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)