MŰEGYETEM 1782

**Budapesti Műszaki és Gazdaságtudományi Egyetem**
Villamosmérnöki és Informatikai Kar

Fábián Gábor Benedek

# LEARNING GENE REGULATORY NETWORKS WITH GAUSSIAN BAYESIAN NETWORKS

KONZULENS

Gézsi András

BUDAPEST, 2016

# Table of contents

# HALLGATÓI NYILATKOZAT

Alulírott **Fábián Gábor Benedek**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2016. 12. 09.

...…………………………………………….
Fábián Gábor Benedek

# Összefoglaló

Napjaink robbanásszerű információs fejlődésének egyik oka a rohamosan növekvő adatkészletek, amik alapján lényegesen több információt tudunk kinyerni az adott területről, mint arra azelőtt lehetőségünk volt. Ez a fejlődés a humánbiológiai területekre is nagy hatással volt. Jó példa erre a humán genom feltérképezése.

Kutatásom során egy ilyen területtel, a génregulációs hálózatokkal foglalkoztam. Az elmúlt években a microarray mérések megjelenése következtében nagy mennyiségű adat áll rendelkezésre olyan weboldalakon, mint a Gene Expression Omnibus.

Fejlesztésem ilyen adatok feldolgozására irányult. Készítettem egy alkalmazást, amely egy bemeneti adathalmaz alapján megpróbálja rekonstruálni a hálózatban található összefüggéseket folytonos Bayes hálókkal. Implementáltam egy scoring függvényt, amely pontozni képes egyes szülőhalmazok valószínűségét az adatok alapján. A struktúra pontos felépítését két elterjedt lokális keresési módszert használva próbáltam megkeresni: ez a hegymászó algoritmus és a szimulált lehűtés volt. Ezeken felül bevezettem egy regularizációs algoritmust, amellyel tetszőlegesen korlátozni lehet a gráfban szereplő élek számát.

Az eredmények kiértékeléséhez bevezettem különböző kiértékelési metrikákat, megvizsgáltam, hogy a módszer hogyan reagál különböző mennyiségű adatokra. Összehasonlítottam a két keresési függvényt, és arra a következtetésre jutottam, hogy a feladat speciális típusa miatt érdemesebb hegymászó algoritmussal dolgozni. Ezen felül megvizsgáltam a regularizáció hatását a keresési eredményeken, és ajánlottam egy módszert arra, hogy hogyan érdemes a regularizáció súlyozását ideálisan beállítani.

# Abstract

One of the main reasons the informational revolution is taking place is the emergence of rapidly growing datasets that enables us to extract more information from the given field than we could before. This evolution had a huge impact on human biological areas as well: a good example is the mapping of the human genome.

In my research I studied the area of gene regulatory networks: in the last few years, the appearance of DNA microarray technology has led to the availability of huge datasets on different websites, such as the Gene Expression Omnibus.

My development is aimed to process data acquired from such sources. I created an application that tries to reconstruct the actual connections with continuous Bayesian Networks. I implemented a scoring function that is able to calculate the likelihood of different sets of parent nodes based on the data. I try to find the exact structure by using two of the most common local searching algorithms: hill climbing and simulated annealing. In addition to that I introduced a regularization technique as well, which allows us to keep the number of edges in the graph to a minimum.

To assess the performance of the algorithms I introduced various evaluation metrics and examined how they perform with different amounts of data. I compared the two searching algorithms and came to the conclusion that in case of this unique problem hill climbing may perform better. Furthermore, I analyzed the effect of regularization on the test results, and offered a method to find the ideal weight of the regularization term.

# List of abbreviations

| | |
|---|---|
| CPDAG | completed partially directed acyclic graph |
| DAG | directed acyclic graph |
| DNA | deoxyribonucleic acid |
| GEO | Gene Expression Omnibus |
| GRN | Gene Regulatory Network |
| HC | hill climbing |
| MCMC | Markov Chain Monte Carlo |
| PDAG | partially directed acyclic graph |
| RNA | ribonucleic acid |
| SA | simulated annealing |

# 1 Introduction

The constant technological revolution and advancement changes our world constantly. One of the biggest trends of the last decade or so has been the increased amount of data that we can collect and process, or as it is summarized in one phrase: big data. If used correctly, it helps us understand our environment and systems better, thus enables us to enhance them and create products that are more efficient, better tailored for the real world.

Biologists are joining the big data movement. In their quest to understand how for example the human body works, for a long time there were no new ways to make discoveries: they could not collect enough data, and even if they somehow could, there was just simply not enough computational power to properly analyze it. For instance, sequencing the complete human genome requires nearly 100 gigabytes of storage.[2] Even though it does not require as much space after processing it, it demonstrates well how much data is generated daily in this industry.

But technology increases exponentially, and we will be able to solve an increasing amount of problems that we could not before. One of the specialty of biological data is that it is very noisy: usually there is an extremely complex system behind it, and it is hard to understand how it actually works just by looking at the data. Therefore, it is truly important to carefully select our approach: what algorithms we would like to use and how we can use our knowledge to simplify the problem to a manageable complexity.

My research focuses on a complex network called Gene Regulatory Networks (GRNs), and explore how we can reconstruct the network from the collected data. GRNs have very important roles in basically every process of life including metabolism, cell cycle etc. It basically describes how genes influence each other: for instance, if one of them activates, will it activate other ones as well? If we knew how these networks worked, it would help shed light on how certain diseases develop or progress, and would lead to for instance better, more personalized treatments.

The reason why GRNs became an emerging field in bioinformatics is mainly because of the invention of microarray analysis [5]. With microarrays we can measure the gene expression levels in messenger RNAs, and do it a lot cheaper than we were previously able to. It is an efficient, high throughput technique, which can measure all genes’ expression level. As its popularity started rising, the amount of data recorded and shared via online portals increased rapidly. On websites like the Gene Expression Omnibus (GEO)[6], there are over 2 million samples shared as of the writing of this paper.

There are many ways to approach this problem, but the one I chose was trying to create a Bayesian network that models the inferences. The Bayesian networks are consisted of nodes that represent the variables (or genes) that we acquired data of, and edges, that represent the direct dependencies between them. It creates models that are easy to understand, because they represent the relationship between variables very well. Also, it is a robust method that has been tried many times, and proved to work proficiently. They are especially practical when the dataset is very noisy, for example in the case when the data is collected from gene expression.

# 2 Biological background

All information needed to describe and build the human body is encrypted in our DNA. The DNA is a collection of genes, which are different regions of the DNA with a specific role that encode specific characteristics of a person. This is done by mainly encoding proteins, which are the building blocks of the human body: they coordinate nearly every function of every cell. Since all of our cells contain the same DNA, not all genes are needed in every part of our body. So the genes that are expressed are the ones that are actually taking part in creating the required proteins. The process in which the information from a gene is used to create a protein is called gene expression.

Gene expression consist of two main parts: transcription, in which a particular segment of the DNA is copied to a messenger RNA; and translation, in which the messenger RNA is „translated” to create the protein. For this project, the messenger RNA has an important role: this is the part where we can measure exactly the expression levels of the different genes. Naturally, this short overview is very oversimplified, but it is necessary to understand where the data come from.

During the process of gene regulation, genes can have an impact on other gene’s expression. This can happen in different ways: they can impede, enhance or even completely silence the expression of other genes.

As mentioned in the introduction, collecting data of the expression levels of genes has become easier mostly because the use of DNA microarrays has become widespread. DNA microarrays are microscopic devices (illustrated in Figure 2.1), that have thousands of spots that correspond to different genes. Each of these spots contain identical strands of DNA that represent one of the genes, and can measure the expression of that selected gene. This enables the scientists to perform measurements on thousands of genes simultaneously.

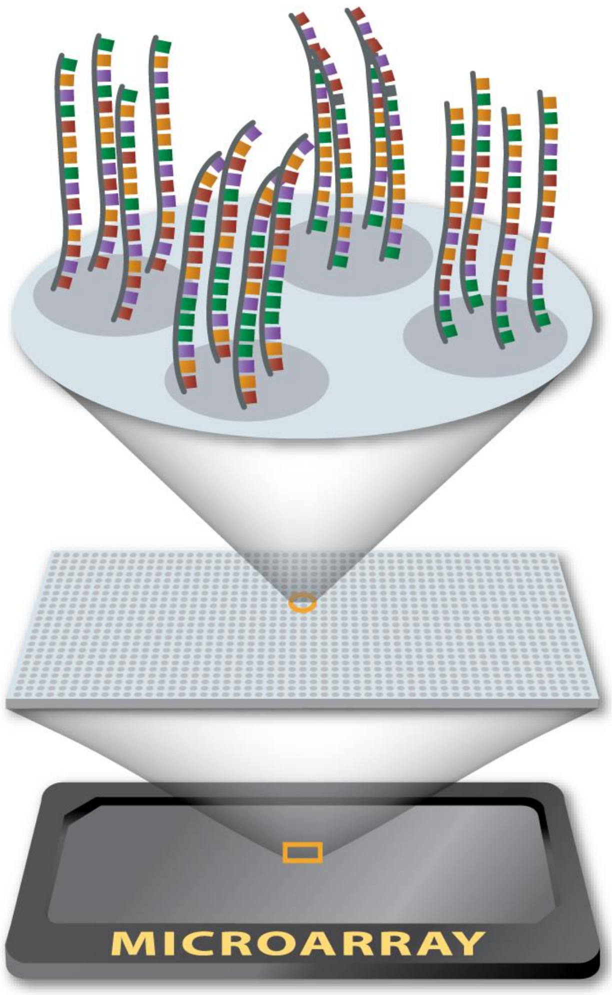


**Figure 2.1. A simplified image of how a DNA microarray is built up. Microarrays are made of a glass case containing thousands of spots that contain identical strands of DNA of certain genes. [7]**

# 3 Bayesian networks

## 3.1 In general

Bayesian networks or belief networks are directed acyclic graphs, where the vertices are probabilistic variables, and the probability assigned to the variable is only influenced by its parents. The edges represent direct dependencies between the corresponding variables, where the direction of the arrow shows the direction of the relationship. For example, as shown in Figure 3.1, P's value influences the probability distribution of Q, and although the value of Q influences the probability distribution of P, there is no causal relationship in that direction.

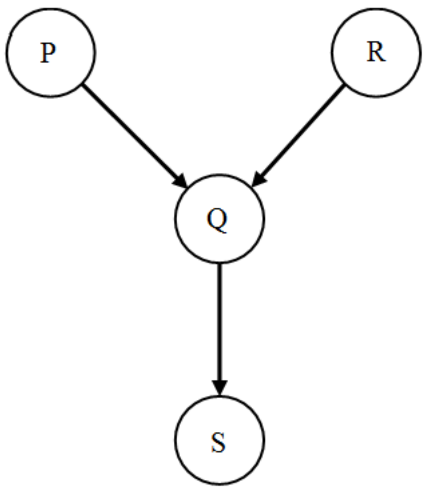


**Figure 3.1. Simple example of a Bayesian network. Here, the probability distribution of Q only depends on the values of P and R. Similarly, the probability distribution of S is only influenced by the value of Q.**

Bayesian networks can be thought of two different ways [1], but before going into details, we need to know what it means if two set of variables are conditionally independent given another set of variables. So let A, B and C be three disjoint subsets of the set of variables. We can say that A is conditionally independent from B given C, if by knowing the values of the variables in C, the set A is independent from B.

$$A \perp B \mid C \iff P(A \mid B \cap C) = P(A \mid C)$$

Now, the first way to look at it is as a compact representation of the joint probability distribution. In a discrete space, there are exponential number of possible combination of events, and by using the chain rule, we can separate the joint probability distribution into a multiplication with N members (where N equals to the number of random variables). Since every variable is only influenced by its parents, the members of the multiplication can be greatly simplified. This way, we can describe the distribution by combining the local probability distributions, reducing the complexity from exponential to polynomial. For example, if we would like to know what $P(P,Q,R,S)$ equals to, we can separate it into multiple probabilities:

$$P(P,Q,R,S) = P(S|Q)P(Q|P,R)P(R)P(P).$$

Another way to understand Bayesian networks is that it represents the conditional independencies between variables, and every node is conditionally independent from its non-descendants, given its parents. In general, there are two types of conditional independencies we can look at:

1. A node is conditionally independent from its non-descendants, if its parents are given. For example, in Figure 3.1, if we know the value of Q, S is independent from P and R.
2. A node is conditionally independent from every other variable given the values of its Markov blanket (which consist of the variable's parents, children and parents of its children). [1]

## 3.2 Types of Bayesian networks

Bayesian networks have three main types: discrete, continuous and hybrid.

In the discrete case, every variable can have a finite number of different values. There are many cases, where we only need to represent whether a certain event does happen or not and we do not need more information beyond that, for example we keep whether it rains or not, but not how many mm of rain we received.

In the continuous case, every variable can have an infinite number of values, for example, if we deal with continuous variables, such as height, price or temperature. In fact, most of real world problems need to deal with continuous variables. In several cases, we can use a method called discretization, when we try to split up the range into multiple intervals, and use the categories we get as a result. There are many tested algorithms that try to accomplish this by wasting as little information as possible.[8]

Hybrid Bayesian networks consist of variables from both discrete and continuous domains. Since it is not required to understand this topic, I will not go into more detail.

## 3.3 Gaussian Bayesian networks

Gaussian Bayesian networks are a subclass of Bayesian networks with continuous variables. The name Gaussian implies that it is connected to a Gaussian distribution: the variables have a Gaussian distribution, or normal distribution. So for a node $X_i$, the conditional density function looks like the following:

$$p(x_i|pa_i) = N(x_i;\ \mu_i + \sum_{X_j \in Pa_i} b_{ij}(x_j - \mu_j), \qquad \sigma_i^2)$$

**Equation 3.1**

As it shows, the normal distribution’s expected value is the linear combination of its own expected value and the difference between the parent nodes’ actual and expected value. The distribution’s variance is fixed. Because of the linear property, the network, in which all the variables show the above described property, is called linear Gaussian Bayesian model.

## 3.4 Reasons for using Gaussian Bayesian models

I chose to use linear Gaussian Bayesian network as my model. One of the main reasons was that we can say that the data collected from gene expression has a lognormal distribution [9], which means, that the logarithm of the distribution is a normal distribution. Therefore, after collecting the data, it is easy to transform it to fit our model.

We do not know whether the variables from the data collected from gene expression have in fact a linear relationship, but even if they are not linear, we have seen many cases where linear models performed well in terms of representing reality. In addition to that, we have many proven methods for working with linear Gaussian Bayesian models. Even if the variables have nonlinear relationships, it may be able to capture the dependencies, and it will be fairly easy to work with.

Of course, we could discretize our variables, and use non-continuous variable spaces, but this was not part of my goals. Besides that, by sticking to the original data, we avoid a transformation that would definitely cause to lose some information.

## 3.5 Partially Directed Acyclic Graphs (PDAGs)

As stated above, when we use Bayesian networks, we represent random variables and their connections with directed acyclic graphs. However, when we would like to simplify our joint probability distribution and create a Bayesian network from it, it has been shown [3] that there are usually multiple DAGs that represent the same set of conditional independence assertions. This does not mean that any of them is wrong: it just means, that from the joint probability distribution alone, there are some connections where the direction is not unequivocal – there is no evidence to decide the orientation. These DAGs, if they represent the same set of conditional independence assertions, are equivalent to each other. The set of possible DAGs that are equivalent to each other represent an equivalence class, and to properly illustrate these classes, we use partially directed acyclic graphs.

Generally, partial DAGs are graphs, that have both directed and undirected edges. It has been shown, by Verma and Pearl [4], that DAGs are in the equivalence class if and only if they have the same skeleton and the same v-structures. By skeleton, I mean the graph that we get if all directed edges are replaced with undirected edges. Also, by v-structure, I mean a set of three nodes that have a structure such that one of the nodes is the child of the other two: $A \rightarrow B \leftarrow C$. PDAG is a graph, which has the same skeleton as all the graphs in its equivalence class, and only the edges that are part of a v-structure are directed. This way, we can illustrate easily what an equivalence class really consists of.

However, not all the remaining edges can be directed in both ways. For example, if we look at Figure 3.2. , there are three undirected edges: $T - P$, $T - Q$ and $R - S$. None of them are part of a v-structure, so they should not be directed in the PDAG representation. However, if we look at all the possible valid extensions of the PDAG graph, we will see, that in every one of them, the R-S edge will have the same orientation. Therefore, we can group the remaining undirected edges into two groups: compelled (the edges that in fact have the same orientation in every instance of the equivalence class), and reversible (where the edges actually can take on both directions). The edge $R - S$ is compelled and have a direction of $R \rightarrow S$, because the $R \leftarrow S$ edge would be part of the v-structure, therefore it would already be directed. On the other hand, $T - P$ and $T - Q$ are reversible, because there are DAGs where orientation of these edges are reversed. If we add a direction to all compelled edges as well, we get a completed PDAG (CPDAG), which now perfectly represents the equivalence class.

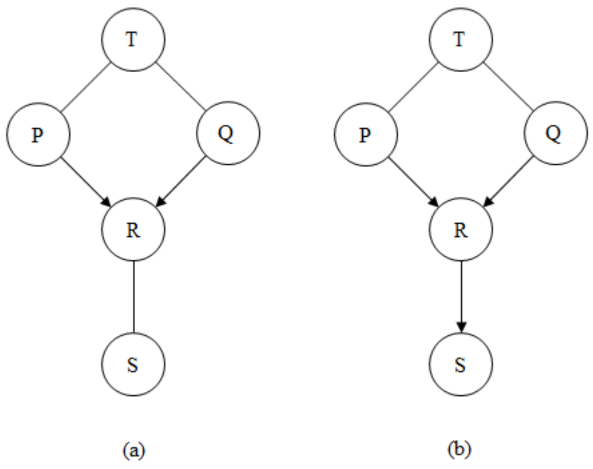


**Figure 3.2. An example of a PDAG and completed PDAG representation of a simple graph**

## 3.6 Convert DAGs to CPDAGs

To be able to evaluate the network the algorithm found, we need to convert it into a completed PDAG that represents its equivalence class.

The algorithm has two main parts. First, we need to make all edges undirected, except the ones that are in a v-structure. It is important to realize, that only those subgraphs are considered as a v-structure, that have the exact same structure as Figure 3.3, i.e. there can be no immediate dependency between P and Q.

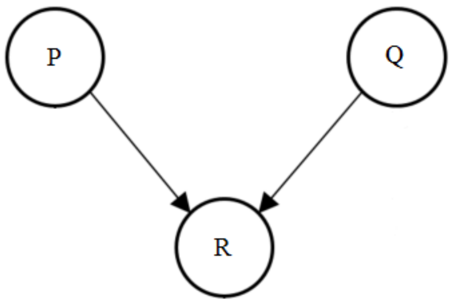


**Figure 3.3. V-structure**

The second part of the algorithm draws the compelled edges. Here is a short pseudocode:

```
while the graph changed in the last iteration
    for every edge that has an end node that has an incoming directed edge
        make it directed, so that it does not create a v-structure
```

## 3.7 Structure learning in Bayesian networks

Structure learning is when we have certain random variables and we have a data sample from the joint probability distribution, and we would like to construct a DAG that best represents the distribution. There are two main approaches to structure learning in Bayesian networks that I will discuss shortly here: Bayesian approach and constraint-based approach.[12]

### 3.7.1 Bayesian approach to structure learning

In the case of Bayesian approach, I mean algorithms that try to find the best structure with global approach. There are two ways I will discuss here: classical approach [10] and Markov Chain Monte Carlo (MCMC) [11].

In the classical case, we have a scoring function, that computes the likelihood of a G graph with the data available, we have a set of steps, that we can take to change the graph and we have a searching algorithm helps us finding the best solution in the space of possible G' graphs. This approach is mostly used, when we have a few solutions that are substantially better than the others.

When we do not have solutions that are a lot better than others, either due to the nature of the problem, or to the amount of data we have, we can use algorithms based on Markov Chain Monte Carlo methods. In this case, our goal is not to find an exact solution, rather to have a distribution over the space of graphs that tells us the most probable components in the graph, let it be edges or directed paths etc.

### 3.7.2 Constraint-based approach to structure learning

In the case of constraint-based structure learning, the aim is to find a DAG that represents all the conditional independencies in our probability distribution. Here, we look at sets of variables locally, and find the connections between them, and construct the graph from these local relationships. In this case, first we find the d-separations that are true for our probability distribution, then find the local connections with classical statistical algorithms. After that, we try to construct the graph from these independent subsets of our graph, paying attention to keeping the graph acyclic.

### 3.7.3 My approach

I implemented the variation of the classical Bayesian approach, where our goal is to find the best DAG that has the highest likelihood based on our data. I will define a scoring function, and then try two local searching techniques to try to find the best solution.

# 4 Scoring function

The scoring function is mainly based on the scoring function defined in Chapter 8.6 of the book Learning Bayesian Networks by Richard E. Neapolitan [13]. Since it is a main component of my algorithm, I will give a short overview of how it works.

The scoring function has the following form:

$$\rho(d|gp) = \rho(d|G) = \prod_{i=i}^{n} \frac{\rho(d_{PA_i^{(G)} \cup \{x_i\}}|gp_C)}{\rho(d_{PA_i^{(G)}}|gp_C)}$$

where $\rho(d|gp)$ denotes the probability density function of $d$ data if $gp$ is the DAG structure, $PA_i^{(G)}$ denotes the set of parents of $x_i$ given G structure and $gp_C$ denotes a complete DAG i.e. where no conditional independencies exist. Therefore, the likelihood of a given G graph structure is the product of all its substructures, which is very convenient, because if we only change the graph locally (for example add a new edge), only one set of parents will change, so it is enough if we calculate that element in the product.

I will not go into detail about how $\rho(d|gp_C)$ is calculated exactly, it is explained in great detail in the book mentioned.

I did not use exactly the equation specified, because after implementing it resulted in extremely big numbers. Therefore, I calculated the natural logarithm of the function, and used that as my scoring function. In addition, I added a regularization member as well to the scoring function, so the end result looked the following:

$$\begin{aligned} score(gp|d) &= \ln(d|gp) - regularization \\ &= \sum_{i=1}^{n} \left( \ln \rho \left( d_{PA_i^{(G)} \cup \{x_i\}} \middle| gp_C \right) - \ln \rho \left( d_{PA_i^{(G)}} \middle| gp_C \right) \right) - \lambda \cdot |gp|^2 \end{aligned}$$

In the regularization term, $|gp|$ means the size of the network, defined as the number of edges present in the graph, lambda is a regularization constant. In the next section, I will go into detail why I chose to use this form of regularization.

## 4.1 Reasons for regularization

When testing the algorithm, if the only constraint of the possible graphs was that it must remain acyclic, I usually ended up with a lot more edges than I should have, because it seemed to find minor connections between entirely separate variables. One of the technique I tried in the beginning was a local constraint, that the number of parents of any of the nodes should not exceed a certain number. This is important not primarily because of regularization, but because there is not enough data to support more complicated connections, so even if in fact there are more parents of a node than we would allow, the goal should rather be to find the most important ones. In terms of regularization, having the same constraint on all the nodes does not seem rational, because variables usually have very different number of parents, so having the same rigid constraint for all of them is not sufficient.

I had the idea of using the global regularization technique that is common in the field of neural networks, called the L2 regularization. Even though I could not apply the same method, since in deep learning it is used as an extra term to the cost function, which keeps the norm of the weight matrix to the minimum, I added a simple quadratic function that punishes graphs that have too many edges.

It produced better results, the only downside was, that when using simulated annealing as the searching method, in the beginning of the search the graph filled up with edges, limiting the possible moves to only a few, thus slowing down the searching process.

I realized that the best approach was combining the two methods: setting a maximum number of parents for all nodes helps keeping the simulated annealing fast, and as the temperate cools down, the global constraint would become more influential, and would keep down the number of edges to the desired point.

# 5 Search

## 5.1 Search space

I defined the search space as possible DAG graphs over the given variables, and the possible moves as edge addition, edge removal and edge reversal.

Initially, I did not consider reversal as one of the possible moves, but many times having an edge in the wrong way have a higher score than not having it at all, so the reversal would not happen in two steps, because there is a local minimum between the two states. Therefore, having a move that reverses the edge in one step should help the searching algorithm considerably.

## 5.2 Searching algorithms

Since the searching space is super exponential, it is impossible to go through all the possible graphs (if the number of variables is more than just a few), we cannot use exhaustive searching algorithms such as breadth-first search or depth-first search. We unfortunately cannot use heuristic so I used two types of local searching algorithms, hill climbing [14], and simulated annealing [15].

### 5.2.1 Hill climbing

The most basic local searching algorithm is called hill climbing. It is a very simple but not optimal searching algorithm, which will stop at the nearest local maximum. Works the following way:

```
starts from a random state

do
     if best adjacent state’s score > 0
          current state ← best adjacent state
until the score of all possible moves are less than 0

return final state
```

Since the stop condition is to not have a neighboring state with a better score than the current one, it is guaranteed to finish in a local maximum. In general, there are many advantages of using this algorithm: the two main ones are that it is easy to implement and it requires no memory. Unfortunately, if the search space is complex (as it is in this case) there are many local maxima, and it is very likely that it will stop at a local but not global maximum.

One of the most common ways to enhance the performance of this algorithm is to start search threads simultaneously from different random points in the search space, and in the end find the best solution from the found local maximums. In this case, random starting points can be acyclic graphs with a set of randomly generated edges. I did not implement the parallel search, but I will examine the effect random edges had on the hill climbing algorithm in the evaluation chapter (Chapter 6.3).

### 5.2.2 Simulated annealing

The name comes from a technique in metallurgy, where in order to create more durable metals, first the metal is heated to a certain temperature, and then it is cooled down in a controlled manner.

The algorithm is analogous to this process, because there is an initial high temperature as well, and we slowly decrease it, and as the temperature lowers we should be getting close to a global maximum. The algorithm works the following way:

```
while numberOfSteps < maxNumberOfSteps and T > 0
     T ← Tmax * ((maxNumberOfSteps – numberOfSteps) / maxNumberOfSteps))
     if T = 0 then return current state
     select random next state
     calculate its score
     if score > 0 or random(0, 1) < exp(score / T)
          current state ← next state
     numberOfSteps = numberOfSteps + 1
```

So initially, we start from a random state (empty graph for instance), and find a maximum temperature, from where we will slowly cool down. As we diminish the temperature, we slowly give less and less probability to not making a positive move. Initially, the aim is to set the temperature to a very high value, that lets the algorithm make almost any move regardless of what its score is, like a random walk in the search space. As we get to a lower temperature, the random walk transforms into hill climbing. The reason it is better than the hill climbing is because in the beginning, during the random walk, the algorithm should be able to step out of local maximums, but stuck in the global one.

## 5.3 DAG condition

Maintaining an acyclic structure is critical to solving this problem. There are two algorithms I implemented, one of them checks if a new edge would violate the DAG condition, the other one checks whether the graph is in fact acyclic.

The first one works in a simple way: checks if there is a directed path between the prospective child and the prospective parent, and if there is not, then the new edge will not create a directed cycle. It simply does a breadth-first search starting from the child, and checks whether the parent is part of the descendants.

When I had to check if the graph is acyclic in general, I used Kahn's algorithm [16]. It is used to determine the topological order, which is necessary to exist if the graph is acyclic. It works the following way:

```
START ← set containing all the nodes
RESULT ← empty list, will contain the topological order
NEXT ← empty set, contains the next possible nodes in the topological order

NEXT ← nodes from START that have no parents
while NEXT is not empty
     current ← first element of NEXT
     children ← current node's children
     move current to RESULT
     move all elements from children to NEXT that have no parents that are
not in RESULT

if RESULT = START return true
     else return false
```

It basically tries to build a topological order, starting from a node that has no parents, and then always considering only those nodes that have no parents other than the ones already in the list. After the topological order is determined, I check whether all the elements are in the list, if so, it is acyclic, if not, there was a cycle in the graph.

## 5.4 Keeping track of possible moves

There are two operations that are very costly and should be executed as few times as possible: checking whether a new edge would violate the DAG condition, and calculating the score of a certain move. The naive approach, that before every step we collect all possible moves and calculate all of their scores, does not work unfortunately – even with graphs that have a 100 nodes, takes too much time.

First of all, we have to keep track of possible moves, because by collecting them before every step, we need to check for every move whether making that move would cause the graph to become cyclic. This is very computationally expensive and unnecessary, since only a limited number of moves' legality changes. For example, deleting an edge creates new possible moves: there may be new legal edges between the descendants and the ancestors of the just deleted edge that were not previously possible because it would have created a cycle. An example is shown in Figure 5.1. Adding a new edge creates even more change: it is possible that there are edges that could have been reversed before, but they are not anymore, because with the new edge they may violate the DAG condition. Reversing an edge is even more complicated. Even though keeping track of all these changes is a very complex process, it is still worth it, because collecting every legal move before every step turned out to be 30 times slower.

Since calculating the score is also slow, we should not recalculate moves with scores that have not changed. In general, we only need to recalculate moves, where the child's set of parents have changed.

As I implemented the hill climbing algorithm, I needed to keep these aspects in mind, because the algorithm always chooses the best possible move, so all legal moves need to be tracked. But it did not work as well when searching by the other method, simulated annealing.

For the simulated annealing to work well it needs to make a considerably higher number of steps – it is basis of how it works. I realized that the algorithm did not work fast enough to achieve a good result in a reasonable time, and I could not optimize the tracking of possible moves any more. Two possible solutions have come up. Firstly, we can try keeping track of the possible moves with an adjacency matrix, and maintain that: there is a proposal of how to use ancestor and descendant matrices here [17]. On one hand, finding these matrices would only require a matrix multiplication, so checking the acyclic condition would get faster, but on the other hand it would not eliminate the number of moves we would have to go through, and also, I would have needed to redo the basis of my implementation. Altogether, it looked like a promising idea, but not the best one.

The other possibility was to not keep track of the moves at all and do the following:

```
while move = null
     X, Y ← two random nodes
     if there is no edge between X and Y and does not create a cycle
          move ← add edge from X to Y
     else if there is an edge between X and Y
          if random(0,1) > 0.5
               move ← delete edge
          else
               if reversing does not create cycle
                    move ← reverse edge between X and Y
```

It is a very simple algorithm that does not require keeping track of either score or legal moves, and is extremely fast.

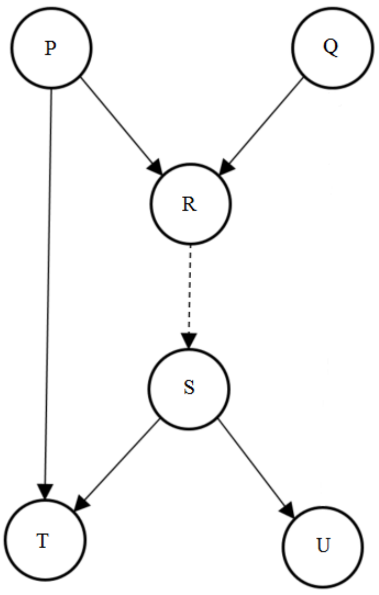


**Figure 5.1. Example of what needs to be considered when maintaining possible edges. When deleting R → S, there are many moves that are now legal: edge addition of U → Q, U → R, U → P, T → R, T → Q, S → R, S → P, S → Q, even the reversal of P → T is legal.**

# 6 Evaluation

## 6.1 Data generating program

Since it is hard to test the algorithm on real data (we have only guesses of what the real structure could resemble), I used András Gézsi's data generating application. It can be used for many forms of data generation, inter alia, generating data from randomly generated Gaussian Bayesian networks.

The application first builds up a network: creates as many random variables as it is specified in the parameters, and determines a specific order that will be equivalent with one of the possible topological orders. Then from the beginning of the list of variables, for each one it generates a number from a geometric distribution that will be the number of parents it will have. If the number is greater than the possible number of parents (which equals to the number of variables in the list before this one), regenerates the number. Then it chooses the exact parents, all of them with equal probability: this is how the edges are created. During the creation, every edge gets a random value from a standard normal distribution: this will be the coefficient to how the parent's value influences the child's value.

After the structure of the graph is built, the program generates as many samples as we specify. It picks values for all variables from a standard normal distribution, and calculates the value from the formula described in Equation 3.1.

The program produces multiple files: a .csv file containing the data lines; a .structure file containing the information about the generated model, including edges and its weights; a .variables file containing a list of generated variables and even an .xgmml file is generated, which contains all information about the model in a format that can easily be processed by software such as Cytoscape [18].

## 6.2 Metrics for comparing PDAGs

During evaluation, my goal was to use the same metrics as we would use in case of a general classification problem: precision, accuracy or even the F1 score. Since over the course of the evaluation I compare the completed PDAG versions of the real and the found networks, there is no clear method in which we should interpret these metrics. I will present multiple ways of comparing these graphs.

Since the variables are given, only the edges need to be compared. There are three main methods of defining whether two edge should be considered as the same. First there is what I will call the *Strict* evaluation: only then is it the same, if the edge in both graphs have the exact same form (directed/undirected) and orientation. The second is the more permissive *PDAG* version: it is a hit, if and only if there is an edge in both cases, and they are not directed in the opposite direction. The last one is what I will call *Structure* evaluation: it only compares the graphs after converting them into undirected graphs. This is the most permissive out of the three: v-structures are the only structures that define an equivalence class, so the aim is to not have a big difference between the performance calculated by the PDAG and the Structure metrics.

By defining these metrics, we can calculate the number of true positives, true negatives and false negatives, and by combining them we can calculate the accuracy, precision and their harmonic mean: F1 score.

One of the deficiencies of these metrics is that they give the same weights to the edges that represent the strongest connections and weakest ones, so to solve this issue, we can calculate weighted sensitivity the following way:

$$\frac{sum\ of\ strengths\ of\ TP\ edges}{sum\ of\ strength\ of\ TP\ and\ FN\ edges}$$

Unfortunately, we do not know the strengths of those edges that are not present in the original network, therefore we cannot calculate neither a weighted precision nor a weighted F1 score.

## 6.3 Evaluating hill climbing from different starting positions

If we look at only the pseudocode of the hill climbing algorithm, we can see that it does not allow any random movement: if we start the search from the same starting position, the results should be deterministic. However, this is not what I experienced when testing the algorithm (as it is shown in the first row of Figure 6.1). By all metrics, the variation turned out to be significant.

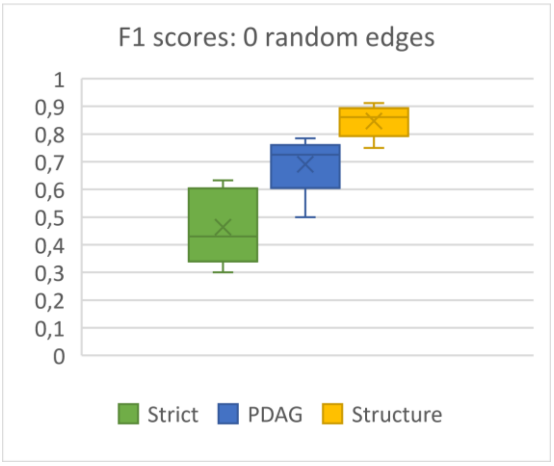
To figure out what is behind it, I examined what the best moves were in the first few steps of the algorithm, and I found that usually there was no difference between the two top scores, and only the order of their evaluation decided which move were actually made (which is completely random, given that I used HashSet to store my moves, where the order is arbitrary). The two top moves were always the addition of the same edge, but in two different directions. This is understandable, since the algorithm only checks the variables common likelihood, and without other edges, this evaluation is symmetric – without v-structures, it is impossible to find the direction of the connection from only observed data.

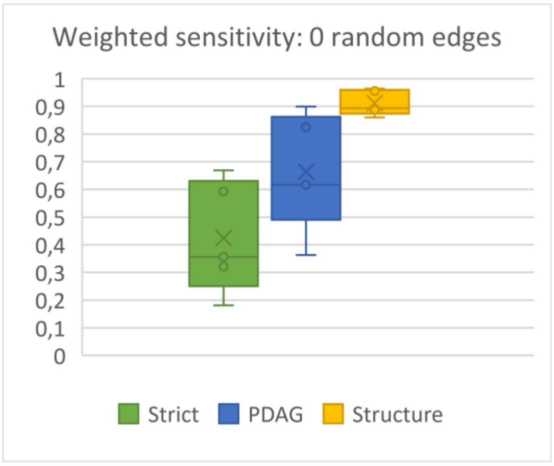
The main conclusion is that it is very important to find the first edges, since in many cases it influences the directions of other edges as well. This will be important when determining the initial temperature in case of simulated annealing.

On Figure 6.1, I tested the hill climbing algorithm on a 20-node network, first starting from the empty network, then a random network containing 10 random edges, then finally a network with 100 random edges (not violating the DAG condition).

There are a few takeaways: firstly, we can see a tendency if we look at the different metrics vertically: the more random edges there are, the worse the results are. However, the F1 scores decline more sharply than weighted sensitivities: even in the case of 100 random edges, the algorithm is able to find the strongest connections (by looking at the Structure metric), but usually not in the right direction (by looking at the PDAG metric).

One of the other interesting results is how small the variance is in case of the F1 scores when starting from 100 edges. An explanation may be, that it usually cannot get rid of many of the random edges, which keeps the F1 scores low.





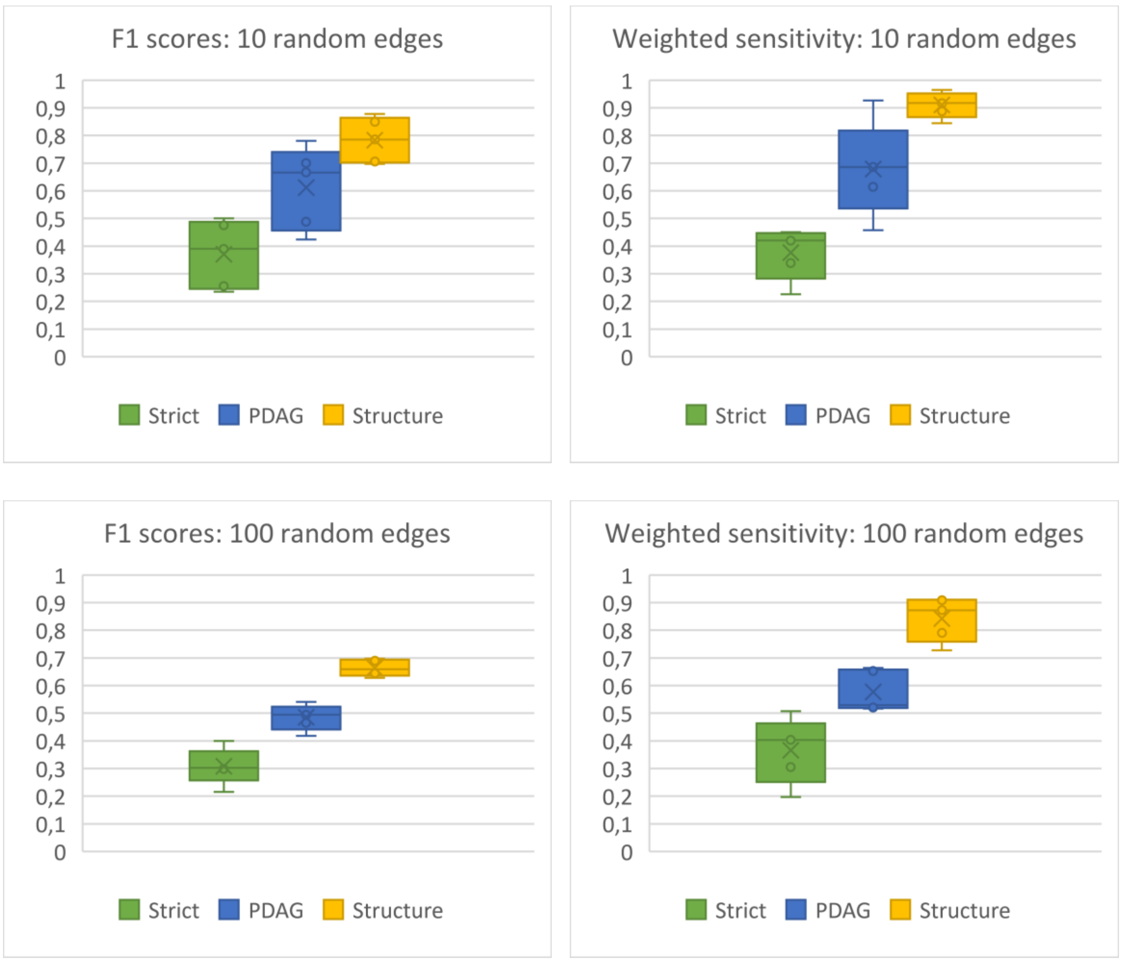


**Figure 6.1. Boxplots showing the results of the hill climbing search from different starting graphs. I used weighted sensitivity and F1 score to evaluate the outcomes (by all three different viewpoints I described in Chapter 6.2).**

## 6.4 Finding optimal maximum temperature

When using simulated annealing as the searching algorithm, it is very important to find an initial temperature that helps the algorithm perform at an optimal level. If the initial temperature is set too high, during the first part of the search the algorithm will do too much random walking, mostly ignoring whether the chosen step has a negative or a positive score, and by the time the temperature gets low enough, there are not enough steps to get rid of all the bad moves. On the other hand, if the initial temperature is set too low, it will be too similar to hill climbing, not allowing to get out of local maximums to get to a global one.

Finding the optimal temperature depends on the distribution of the scores of the possible moves, so there is no applicable algorithm to find the initial temperature in every case. After examining the distribution of scores, I found that it is similar to the distribution of the strengths of the edges, which has a standard normal distribution. Since the positive scores will definitely pass every time, I only looked at moves with negative scores. I needed to figure out, how much probability the worst moves should have in the beginning. I decided to collect the absolute values of the moves that had a negative score, and after sorting it, I chose to look at the score that is at the 90th percentile as an anchor, and adjust the temperature and thus the probabilities to that number. I did not know exactly how much the initial probability of that anchor should be, so I examined it: I tested the F1 scores and the weighted sensitivities of different initial probabilities of a 20-node network. Figure 6.2 shows the results:

The data looks very arbitrary: the best results may be between 0.5 and 0.7, but there is no clear tendency.

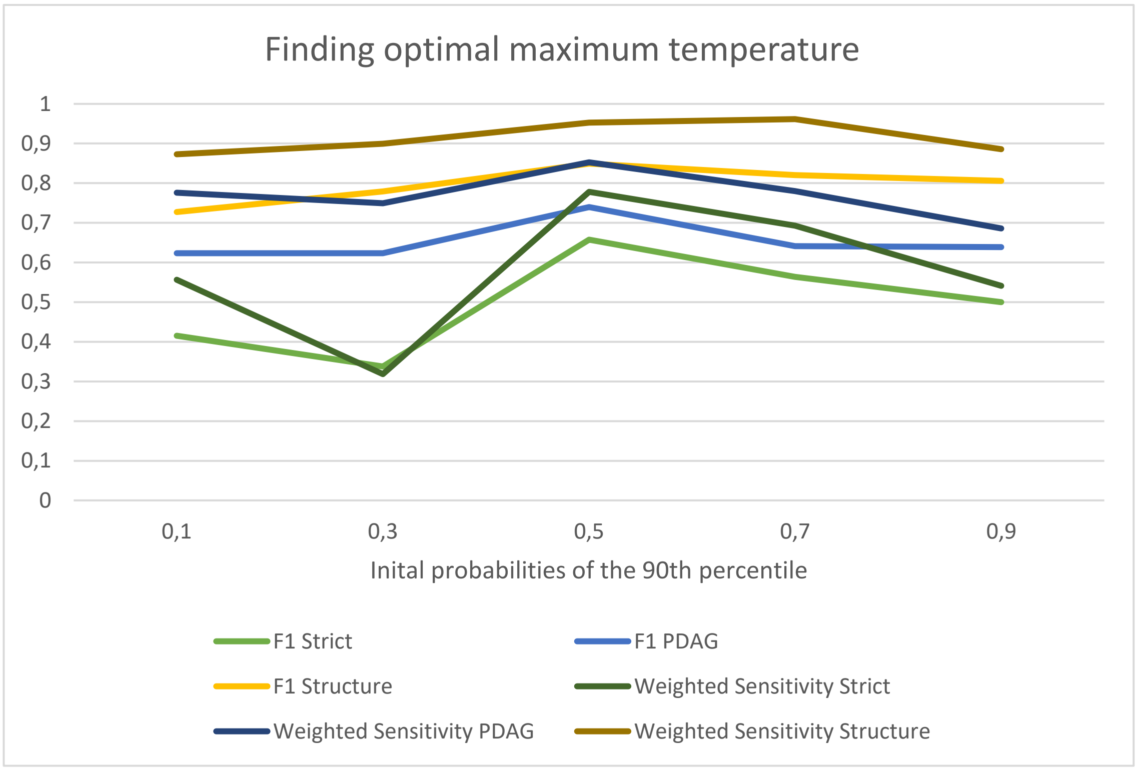


**Figure 6.2. Weighted sensitivities and F1 scores of the results of different initial probabilities, if the anchor was set to the 90th percentile.**

I figured that maybe the problem is not with the initial probability, but with the anchor. This is reasonable, since as I discussed in the previous chapter, if the most important connections are found in the wrong direction, it influences greatly the other edges as well. The problem may be that the simulated annealing cannot get out of positions where the strongest connection is found in the wrong way, so I moved the anchor from the 90th percentile to the 99th, expecting it to enable the algorithm to reverse even the edges with the highest scores. Here are the results of the comparisons:

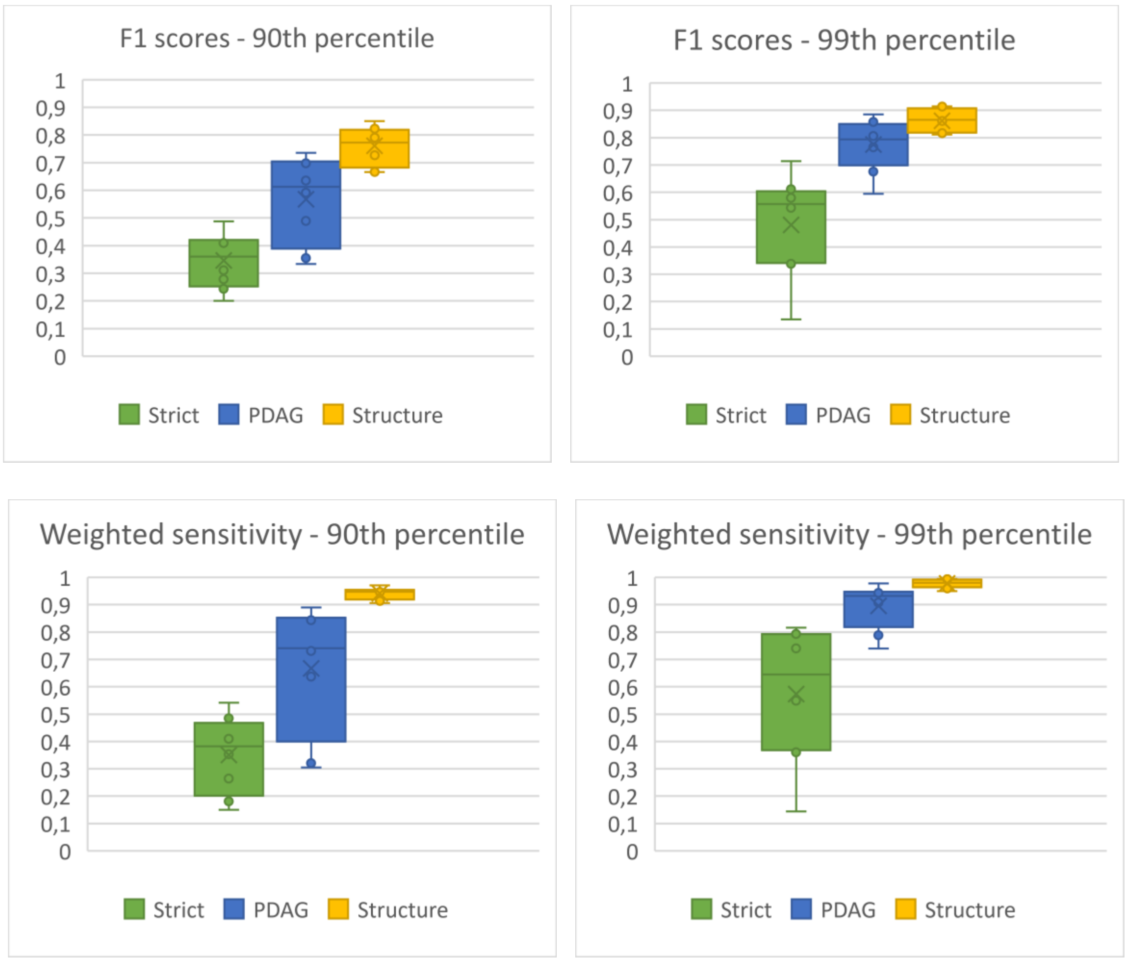


**Figure 6.3. Comparing results between two instances, where we set the anchor to the 90th and 99th percentile.**

The most important difference in my opinion is that the PDAG metrics, both the sensitivity and the F1 score, are a lot closer to the Structure metrics. This means, that there is much higher chance that we find the edges with the right orientation.

## 6.5 Comparing the two searching algorithms

### 6.5.1 Different network sizes

I wanted to compare the two algorithms in terms of how they perform with different network sizes. I used a network with 20, 35, 60, and 100 nodes and ran both of these algorithms. Since the hill climbing only does about as many steps as the number of edges in it in the end (it only adds every single edge once, rarely reverses or deletes any of them), I rather compared how long an average step takes. Figure 6.4 shows the results.

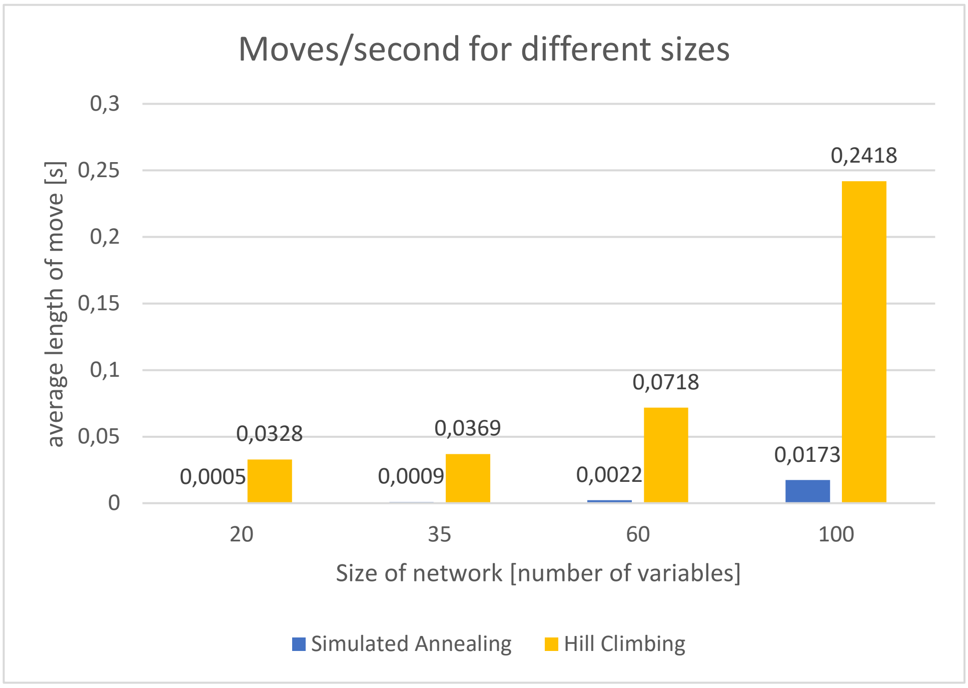


**Figure 6.4. Average length of moves of the two algorithms with different network sizes.**

As it can be seen on Figure 6.4, the length of evaluating all possible moves, as in case of hill climbing, is significantly longer than in case of simulated annealing. However, if we take into account, that SA takes a minimum of 100000 steps to work, while HC only

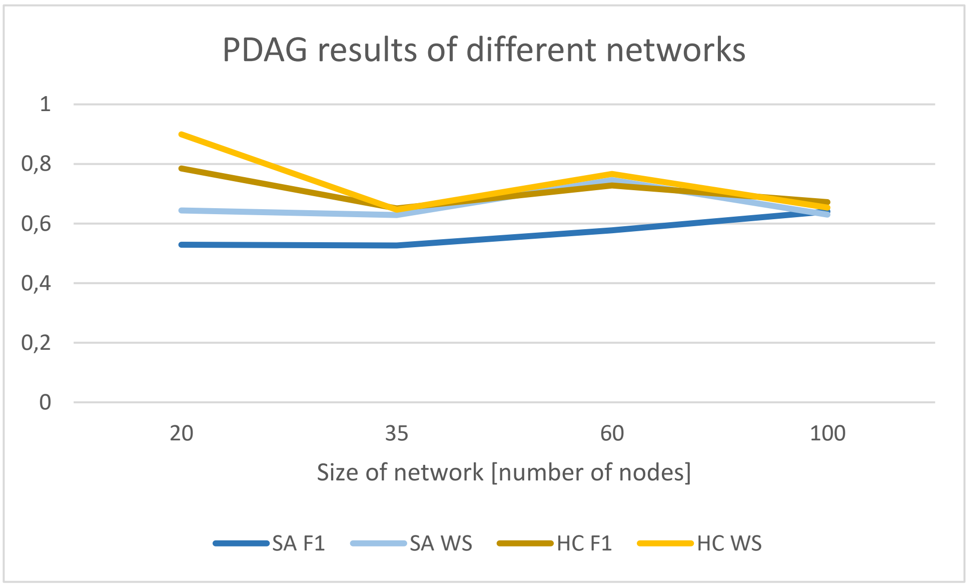


**Figure 6.5. Results of PDAG metrics for different network sizes with the searching algorithms.**

takes steps that is proportional to the number of edges, hill climbing is significantly faster every time (at least the sizes I examined).

I also examined how they performed (shown on Figure 6.5), and I do not think anything can be said definitely, since on Figure 6.1 and Figure 6.3 the graphs show how big the variance is between two instances with the same parameters. However, the graph may suggest that hill climbing is not worse in this situation than simulated annealing, especially, if we factor in the runtimes, hill climbing takes less than a minute in every instance I tried, while simulated annealing took 2,5 hours in case of a 100 variables and 250000 steps.

### 6.5.2 Different data sizes

I was really interested in how the two algorithms work with different sizes of data. However, the scoring function works in a way that the scores are proportional to the number of rows we use, which leads to a problem that I cannot use the same regularizations in cases with different data sizes. So I tested them without regularization, and the number of edges on these networks are shown on Figure 6.6. So when only working with a 100 rows of data, the algorithms did not build up a size as it should have, and it needed more than a 1000 lines to have a need for regularization at all. This is especially true when we used simulated annealing: it built up a big network, with a lot more connections than it was actually needed.

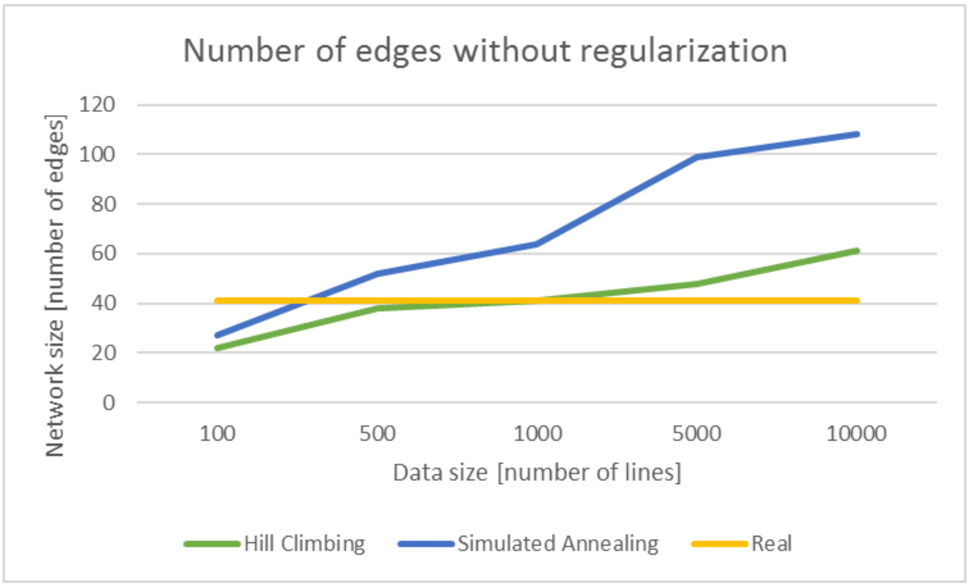


**Figure 6.6. Number of edges found without regularization for different number of data lines used.**

I also looked at how they performed without regularization. I chose to illustrate only the PDAG metrics: I think it gives us the best illustration out of the three options. As I did before, I chose to show the F1 scores and the weighted sensitivities on Figure 6.7.

Since it was only examined on one network instance, the exact numbers are less important, rather the tendency that is shown on the graph. It is obvious that using 100

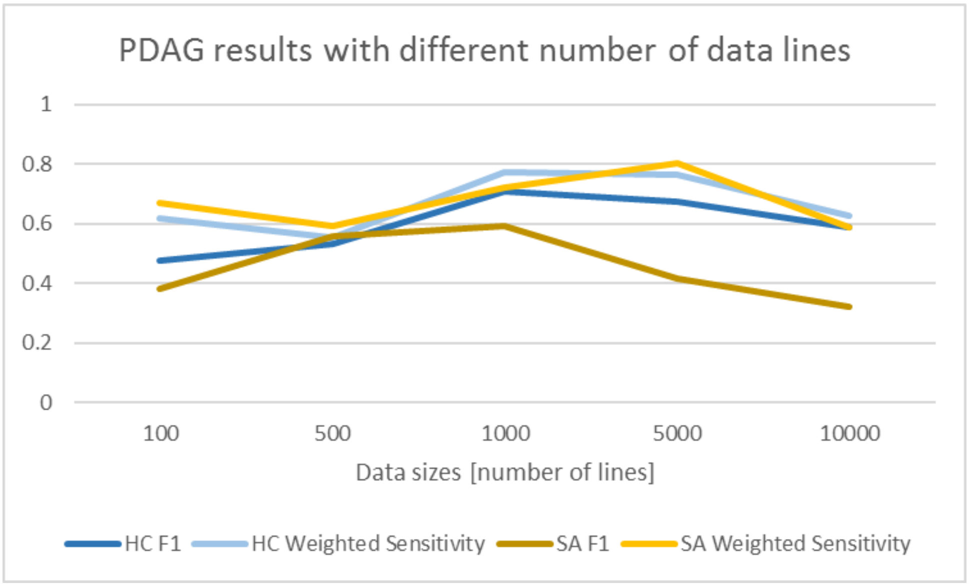


**Figure 6.7. PDAG results of the two algorithms with different number of data lines used.**

rows did not create enough confidence in the algorithm to add enough edges, so both metrics have a very low value in case of both algorithms. On the other hand, after reaching 5000 and 10000 rows, the results drop, since they have too many edges, which lead to too many false positives. This leads to a decline regarding the F1 scores, which is apparent when looking at the results.

### 6.5.3 Comparing performances

Since many of the previous conclusions were not convincing, I did another couple of tests, to decide which algorithms are better in case of my problem. I chose to use a random graph with 20 nodes, since the runtime of 250000 steps of simulated annealing with a 100 nodes takes 2.5 hours, whereas it only takes about 5-6 minutes to make 500000 steps in a 20-nodes network. I ran each algorithms 8 times, and the results are shown on Figure 6.8.

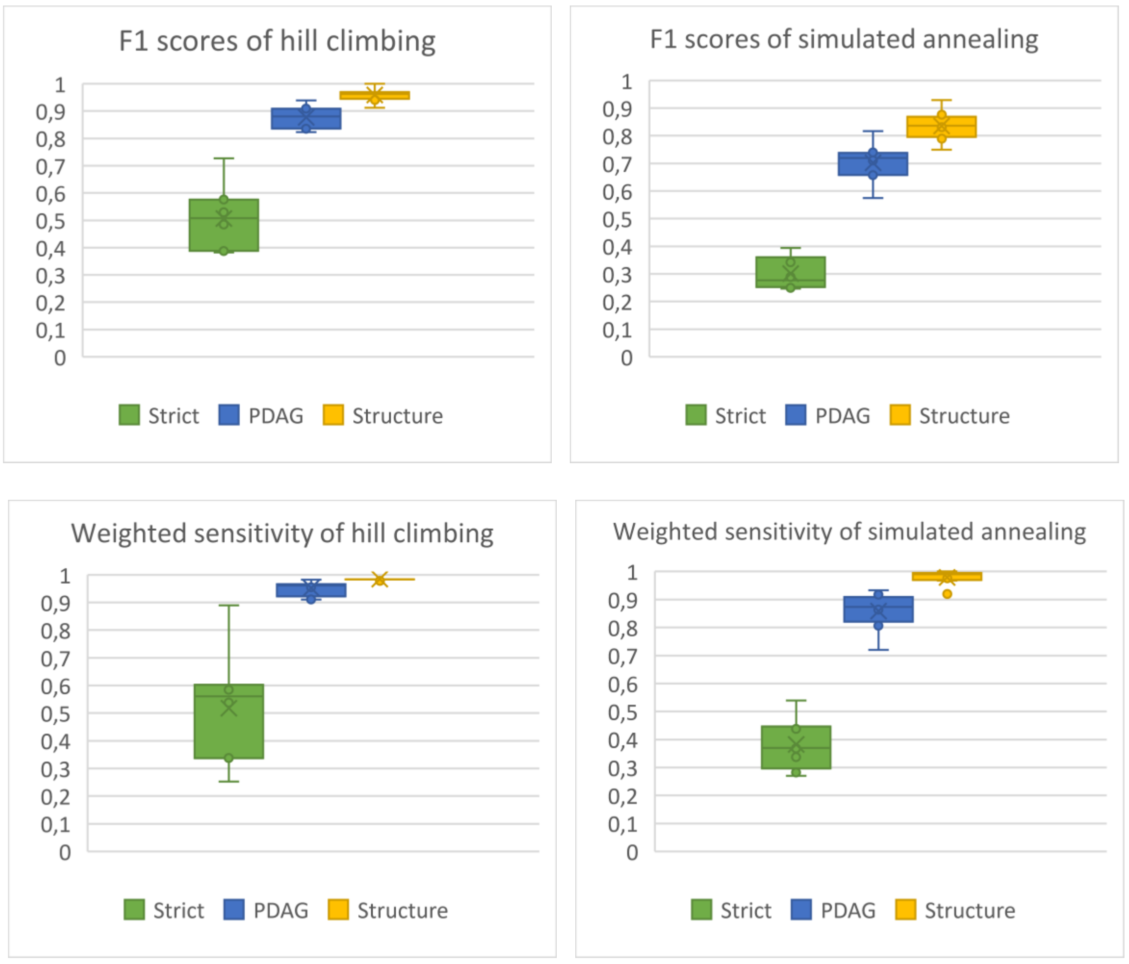


**Figure 6.8. Comparison of the performances of the two algorithms on the same graph.**

It can be seen that in case of almost every single metric hill climbing performed better than simulated annealing. If we combine this evidence with the fact that the runtimes are almost incomparable, these findings suggest that in this particular case hill climbing may work better than simulated annealing, which is widely considered as a method that performs better than hill climbing.

I believe that the reason behind the success of the simpler algorithm is that it starts from such a good location, that in many cases it reaches one of the optimums that is close to a global one. Usually this is the exact reason why hill climbing is not used: if we do not know where to start the algorithm from, it usually stops at a suboptimal state. However, in this case, if we start the hill climbing from an empty graph, it knows exactly which the most important edges are, but usually it can only guess its direction, especially at the start of the search. But it may be a lot faster to run the algorithm maybe a 1000 times and find the best solution out of those than run simulated annealing for that long.

One of the other solutions is to do a search that precedes hill climbing, that helps find the direction of the most important selections, and then start hill climbing, utilizing the speed of algorithm while averting its biggest disadvantage.

## 6.6 Finding the ideal regularization constant

Chapter 6.5.2 leads to a question of how to find the ideal regularization constant. Its value depends on two parameters: the size of the network, and the number of data lines used, so finding what the relationship between these three variables is would help using the regularization properly.

The connection between the number of data lines used and the scores are directly proportional to each other: using 10 times as much data results in 10 times higher scores. Since the regularization constant linearly influences the regularization, changing it proportional to the data size should help keep the effect of the regularization the same.

So now we need to find the connection between the number of nodes used in a graph and the needed regularization. To find that, we first need to calculate the average number of edges of a network with N nodes. We have no knowledge of how the number of connections shape up in real gene regulatory networks, so I will use the generated networks as my basis of evaluation. Fortunately, we know how these networks are generated: each variable’s number of parents is generated from a geometric distribution with a probability of 0.5. It is known that the mean of a geometric distribution is $\frac{1}{p}$, therefore the expected value of the number of edges in a network of N nodes is $\frac{1}{p} * (N-1) = 2 * (N-1)$. Therefore, to be able to ignore the random variation of the network generation, I will use these numbers as a target variable when finding the optimal regularization.

Also, since we know the relationship between the number of data lines and the constant, I will set the number of data lines to a 10000, and only examine this case.

I tested different data sizes and different values for lambda, and I expected to find a clear tendency: as I increase the size of the network, the lambda should be lowered, since the regularization term influences the scores in a quadratic way. But the results I got turned out to be very random: the number of edges found greatly depends on the size of the actual network, and every runtime, the first few edges influence the results. For four networks with different sizes I iteratively found the lambdas that resulted in a network with the preferred size. To find what the values are between the tested network sizes, I tried to fit different curves on the found data points. First, I tried to fit a curve with a form of $y = \frac{P}{x^2}$. I concluded, that since the regularization term is quadratic, and I found that the edges and the nodes have a linear relationship, I figured that having a lambda that is inversely proportional would make sense, and also, the value of a function of this type goes under 0. However, as Figure 6.9 shows, it did not fit as well as I hoped. So I tried a quadratic function as well, which led to a function that was much closer to the points, but did not make as much sense, since the function of this type's limit is infinity as x approaches infinity, and there should be no point in this function where the lambda starts increasing.

As I mentioned before, the ideal lambda depends very much on the actual network, but if we need to find a lambda, that works well, the following function should be used

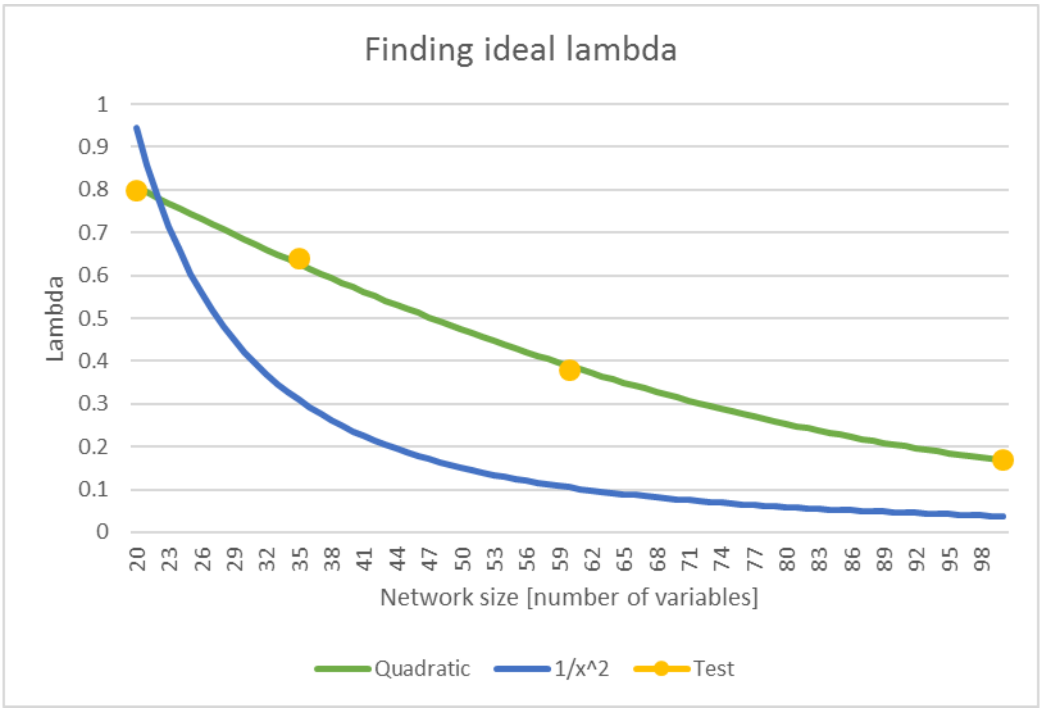


**Figure 6.9. The chart shows the lambda values I found through testing, and two curves, that are fitted to be closest to the given points. The functions are the following:**

$$\boldsymbol{y = 0.00006214243x^2 - 0.01543723x + 1.090721}$$

$$\boldsymbol{y = \frac{377.6663}{x^2}}$$

as a starting point:

$$lambda = \frac{data\ size}{10000}(0.00006214243N^2 - 0.01543723N + 1.090721)$$

where N is the number of variables, and it does not exceed a 100 variables.

## 6.7 Effect of regularization

Regularization plays a big part in finding the ideal networks, so it is crucial to be sure that it does not ruin rather helps the performance of our algorithms. As it was described in Chapter 4.1, local regularization is rather used not because we would like to limit the size of our network, but because the scoring function cannot find a pattern of more than a certain amount of parents with a limited amount of data. Therefore, I chose to test only the global regularization.

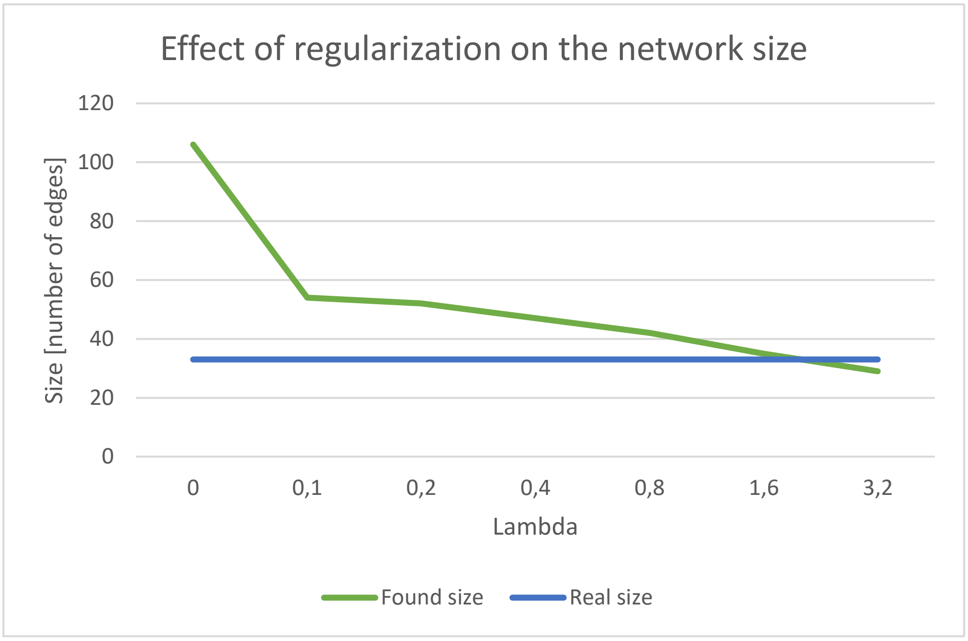
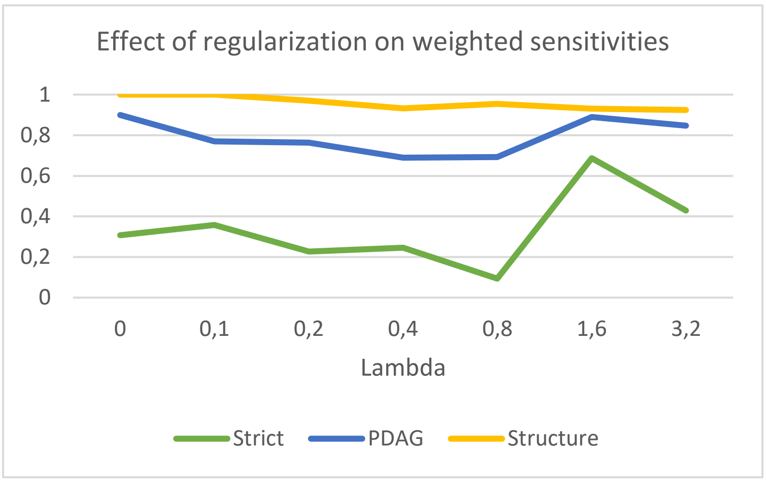


**Figure 6.10. Effect of the global regularization term (lambda) on the number of edges found.**

First, the goal was to see what its effect is on the number of edges – whether it in fact works or not. I chose a network with 20 nodes and used simulated annealing as my searching algorithm, and I tested it on an exponential scale. The results are on Figure 6.10.

It is apparent from the graph that it works well, and that it changes in a way that is similar to exponential: from 0.1 to 3.2, the line that shows the found network's size is close to linear. However, it is possible that it works differently if we run the simulated annealing for more than 100000 steps: the network only gets rid of its edges at the end of the process, so giving it more time may lead to different results (probably a more effective regularization – less edges).



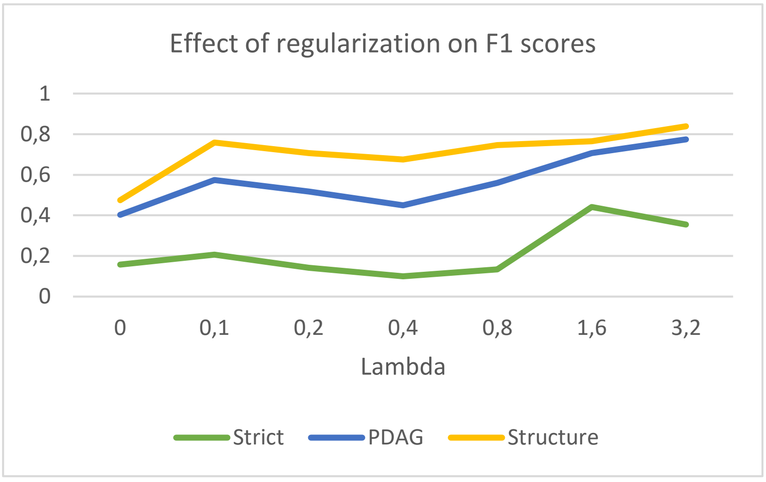


**Figure 6.11. Results of different metrics when trying out different regularizations.**

Now, that it is shown that it works, I was curious to see what its effect is on the metrics I examined – Figure 6.11 shows the results. The results are, again, somewhat arbitrary, since the results variate with every single test run, but there are some apparent tendencies. Firstly, on the first graph, it can be seen that without regularization, the algorithm finds almost all of the real connections, but second graph shows that it finds just as many wrong ones as well. Secondly, using regularization does not result in significant declines on first graph, whereas the F1 scores increase considerably.

# 7 Conclusion and discussion

In this thesis my aim was to create an application that is able to process data acquired from examining the process of gene expression. By doing that, I implemented a scoring function, that is able to calculate the likelihood of a collection of nodes, and thereby it can successfully find a Bayesian network, that approximates the real network. It does that by using local searching algorithms, hill climbing and simulated annealing, whose performances I have assessed extensively. I also applied a regularization technique to keep the size of the network under control, and offered a function to find an ideal lambda.

One of the main conclusions as the result of the evaluation is that this type of problem is unique in a respect that the simulated annealing did not perform better than the hill climbing algorithm. The latter starts from such a good position, that it is very close to the optimal solution, and their runtimes are not even comparable: the hill climbing algorithm finishes the process in less than a minute, even if we test it on a 100 node network.

The other main conclusion is that it is extremely important to find the first few edges in the right direction, because it influences the rest of the graph very significantly. The strongest connections are almost always successfully discovered, but the direction of those relationships are very arbitrary.

The third lesson is that using a regularization increases the performance of the various metrics of the graph, and still retains the most important connections.

There are many ways this application could be developed further. Making the application multithreaded would improve on many of the performance limitations, which would enable the use of larger, more realistic data sizes. Also, the global regularization could be improved if we could not only specify the regularization coefficient’s (lambda) value, but also specify what the target network size is, and then let the application find an ideal lambda for us. The most important possible development is definitely an algorithm that helps finding the strongest connections’ direction. It could be done different ways: enabling the algorithm to start its search from a semi-discovered graph, or use other structure finding methods, such as constraint-based statistical methods (mentioned in Chapter 3.7.2) to find the directions of these edges.

# Bibliography

[1] Russell, Stuart J., and Peter Norvig. "Probabilistic Reasoning." *Artificial Intelligence: A Modern Approach*. 2nd ed. Englewood Cliffs, NJ: Prentice Hall, 2003. Print.

[2] Orcutt, Mike. "*Bases to Bytes.*" MIT Technology Review. MIT, 25 Apr. 2012. Web. 24 Nov. 2016.

[3] Pearl, Judea, and Thomas S. Verma. "*Equivalence and Synthesis of Causal Models*" 1990

[4] Pearl, Judea, and Thomas S. Verma. "*A Theory of Inferred Causation.*" 1991

[5] Schena, Mark, et al. "Quantitative monitoring of gene expression patterns with a complementary DNA microarray." *Science* 270.5235 (1995): 467.

[6] Edgar R, Domrachev M, Lash AE. Gene Expression Omnibus: NCBI gene expression and hybridization array data repository, Nucleic Acids Res. 2002 Jan 1;30(1):207-10

[7] Genetic Science Learning Center. (2013, February 14) DNA Microarray. Retrieved November 22, 2016, from http://learn.genetics.utah.edu/content/labs/microarray/

[8] Irani, Keki B. "Multi-interval discretization of continuous-valued attributes for classification learning." (1993).

[9] Bengtsson, Martin et al. "Gene Expression Profiling in Single Cells from the Pancreatic Islets of Langerhans Reveals Lognormal Distribution of mRNA Levels." *Genome Research* 15.10 (2005): 1388–1392. PMC. Web. 24 Nov. 2016.

[10] Neapolitan, Richard E. "Bayesian Structure Learning" *Learning Bayesian Network*s. Upper Saddle River, NJ: Pearson Prentice Hall, 2004. Print.

[11] Neapolitan, Richard E. "Approximate Bayesian Structure Learning" *Learning Bayesian Network*s. Upper Saddle River, NJ: Pearson Prentice Hall, 2004. Print.

[12] Neapolitan, Richard E. "Constraint-Based Learning" *Learning Bayesian Network*s. Upper Saddle River, NJ: Pearson Prentice Hall, 2004. Print.

[13] Neapolitan, Richard E. "Learning Structure: Continuous Variables" *Learning Bayesian Network*s. Upper Saddle River, NJ: Pearson Prentice Hall, 2004. Print.

[14] Russell, Stuart J., and Peter Norvig. "Informed Search and Exploration" *Artificial Intelligence: A Modern Approach*. 2nd ed. Englewood Cliffs, NJ: Prentice Hall, 2003. Print.

[15] Aarts, E., and J. Korst. *Simulated Annealing and Boltzmann Machines*. United States: New York, NY; John Wiley and Sons Inc., 1988. Print.

[16] Kahn, Arthur B. (1962), "Topological sorting of large networks", *Communications of the ACM*, 5 (11): 558–562, doi:10.1145/368996.369025.

[17] Giudici, P. & Castelo, "Legal moves for DAG models" R. Machine Learning (2003) 50: 127. doi:10.1023/A:1020202028934

[18] Shannon, Paul, et al. "Cytoscape: a software environment for integrated models of biomolecular interaction networks." *Genome research* 13.11 (2003): 2498-2504.

# Appendix

The application has a console input, where many of the parameters can be specified. Here are a few simple examples:

```
java –jar GBNStructureLearner.jar –df res/sample.0.data.csv –sf
res/sample.0.structure –sa hillclimbing –re 100 –dr 10000

java –jar GBNStructureLearner.jar –df res/sample.0.data.csv –sf
res/sample.0.structure –sa simulatedannealing –ns 100000 –lb 0.2

java –jar GBNStructureLearner.jar -d res/sample.0.data.csv -st
res/sample.0.structure -e -ld myNetwork.txt
```

The complete list of options are detailed in Table 1.

| **Option** | **Long option** | **Parameter?** | **Desciption** |
|---|---|---|---|
| -df | data-filename | Yes | specify file containing data |
| -dr | data-rows | Yes | specify the first how many rows should be used for searching |
| -eo | evaluation-only | No | if you only wish to evaluate the network |
| -lb | lambda | Yes | specify how strong the regularization should be (default=0) |
| -ld | load-from-file | Yes | if you wish to load the network you saved previously, specify the file |
| -ns | number-of-steps | Yes | specify the number of steps the search algorithm should make (default=10000) |
| -re | random-edges | Yes | specify how many random edges should the graph contain (default=0) |
| -sa | search-algorithm | Yes | choose searching algorithm (default=sa) (options: sa/simulatedannealing/hc/hillclimbing) |
| -sf | structure-filename | Yes | specify file containing network structure |

**Table 1. Commands to control the behavior of the application.**