An Algorithmic Framework For Density Estimation Based Evolutionary Algorithms

Peter A.N. Bosman
peterb@cs.uu.nl

Dirk Thierens
Dirk.Thierens@cs.uu.nl

Department of Computer Science, Utrecht University
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands

December 1999

Abstract

The direct application of statistics to stochastic optimization in evolutionary computation has become more important and present over the last few years. With the introduction of the notion of the Estimation of Distribution Algorithm (EDA), a new line of research has been named. The application area so far has mostly been the same as for the classic genetic algorithms, being the binary vector encoded problems. The most important aspect in the new algorithms is the part where probability densities are estimated. In probability theory, a distinction is made between discrete and continuous distributions and methods. Using the rationale for density estimation based evolutionary algorithms, we present an algorithmic framework for them, named IDEA. This allows us to define such algorithms for vectors of both continuous and discrete random variables, combining techniques from existing EDAs as well as density estimation theory. The emphasis is on techniques for vectors of continuous random variables, for which we present new algorithms in the field of density estimation based evolutionary algorithms, using two different density estimation models.

1 Introduction

Optimization problems are formulated using variables that can assume values from some domain. The allowed combinations of assignments of values from that domain to those variables, constitutes the search space for the optimization problem. Each such instantiation of every variable is called a *feasible* solution. Each such a solution can be graded as to how well of a solution it is for the problem at hand. The objective with respect to the optimization problem, can then be defined so as to find the *optimal feasible* solution. The definition of optimality is subject to the problem definition (eg. minimization or maximization).

The class of optimization problems can be seen as a superclass. Every optimization problem resides in some class that is a subclass of the general optimization problem class. An example of such a class of problems are the combinatorial optimization problems (COPs). These problems are in general defined so as to find the minimum subset $S \subseteq N$, given a set of n weighted elements $|N| = n$ such that $S \in \mathcal{F} \subseteq \mathcal{P}(N)$, or in other words such that S lies within the set of feasible solutions $\mathcal{F}$, which is a subset of the powerset of N that contains all possible subsets of N. As the size of set $\mathcal{F}$ is usually an exponentially growing amount as a function of n, the search space tends to be very large. A lot of well known combinatorial optimization problems are not surprisingly in the class of $\mathcal{NP}$–complete problems. This means that for such a problem we cannot expect that there will ever be a polynomial time algorithm that finds the optimal feasible solution.

Different optimization approaches focus on a subset of optimization problem classes. An algorithm that is often applied to solving $\mathcal{NP}$–complete problems, is the branch and bound method (see for instance [27]). Branch and bound is in general an exponential time algorithm in the case of exponentially large search spaces, that traverses the search space systematically to find the optimal solution.

The way that feasible solutions are graded along with which solutions are to be seen as feasible, codes the structure of the search space. All of this structural information is formulated using the problem variables. Any reasonable optimization problem has structure. Regard for instance the problems in the COP class. The common known problems in this class, such as the TSP, clearly have some structure. If we exploit this structure, we can perform optimization faster to give better results.

In black box optimization, we have no prior knowledge of the problem structure. The only thing available is a mechanism to grade a feasible solution. Assuming that the underlying problem within the black box has structure, attempting to find out and subsequently use this structure is a more preferable way to traverse the search space than a random search.

Searching for and using such structure in optimization has been an active line of research within the field of genetic and evolutionary computation. One of the latest developments in this field is to build and use probabilistic models over stochastic random variables, where one random variable is introduced for every coding variable within the optimization problem. Most of the theory and practice within this line of research has been done in the class of problems that are encoded using only binary variables. The reason for this is that the new methods have been developed in the field of genetic and evolutionary computation, wherein the genetic algorithm (GA) plays one of the most important and historical roles.

Within the standard genetic algorithm [12, 18], strings of binary variables are manipulated using operators inspired by biological evolution. Parts of information are exchanged between solutions as a result of recombination. A selection of solutions that are maintained in a population at any point in time, is appointed to undergo this recombination operation. If the structure of the encoded problem is such that it contains sets of interacting variables (building blocks), it is known that the population size required to solve the problem for a standard genetic algorithm with uniform crossover, grows exponentially with the problem size [33]. The uniform crossover operator combines two solutions by swapping the values for every variable with a certain probability.

In the compact GA (cGA) by Harik, Lobo and Goldberg [16], the dynamics of the standard GA are mimicked using only a single probability vector that codes whether a variable should be set to 1. This probability vector as a result resembles a univariate probability distribution over the binary variables. Using such a univariate probability distribution in different ways, has in addition to the cGA resulted in the PBIL approach by Baluja and Caruana [2] and the UMDA by Mühlenbein and Paaß [23]. These approaches share the problem of not being able to efficiently optimize problems that contain sets of interacting variables.

To take into account interacting variables, Holland [18] already recognized that it would be beneficial to the GA if it would exploit this so called *linkage* information. The inversion operator was proposed to accomplish this, but it unfortunately has turned out to be too slow with respect to the speed of convergence of the GA. Through other approaches, such as the mGA [14], the fmGA [13], the GEMGA [19, 4], the LLGA [21] and the BBF–GA [34], the simple GA was extended to be able to process building blocks. Other approaches have come to focus more on the probability density view of processing the problem variables. An overview of the work in this field was given by Pelikan, Goldberg and Lobo [25]. Our work in this paper contributes along those lines of research.

Staying within the field of problems that are encoded using binary variables, approaches that allowed for pairwise interactions between variables in the probability distribution were presented. The reason for this extension to higher order statistics is the equivalent of the quest for the exploitation of linkage in GAs. Because the variables are being processed independently when using a univariate distribution, problems that are defined using higher order building blocks can not be solved efficiently. In these approaches, a variable may be conditionally dependent on exactly one other variable. Allowing different structures for pairwise interactions, the MIMIC algorithm by De Bonet, Isbell and Viola [6], an optimal dependency tree approach by Baluja and Davies [3] and the BMDA algorithm by Pelikan and Mühlenbein [26] were introduced. For all of the methods in this introduction, we leave the details to section 3.

More recently, approaches allowing multivariate interactions were presented. In these approaches, a variable may be conditionally dependent on sets of variables. These methods include

the BOA by Pelikan, Goldberg and Cantú–Paz [24], the FDA by Mühlenbein, Mahnig and Rodriguez [22] and the ECGA by Harik [15]. Even though finding the correct structure to capture the sets of interacting variables is very difficult, covering multivariate interactions is the key to solving higher order building block problems and exploiting problem structure [7].

Mühlenbein, Mahnig and Rodriguez [22] first presented a general framework for this type of algorithm, named EDA (Estimation of Distribution Algorithm). This framework is a general optimization algorithm that estimates a distribution every iteration, based upon a collection of solutions and subsequently samples new solutions from the estimated distribution. In this paper, we make certain steps from the EDA more explicit within a new framework. By doing so, we allow algorithms within the new framework to follow a certain rationale for density estimation based evolutionary algorithms.

Next to presenting this new algorithmic framework, we also present three instance classes for it. One of these instances is the class of empiric probabilities for discrete variables, which follows the approaches mentioned above. The estimation of distributions is however not limited to binary or discrete spaces and can be extended to real spaces. Therefore, we also present two new instance classes for problems defined on real continuous variables.

The simple GA and the approaches discussed so far only regard binary or discrete encoded problems. In the case of real valued problems, evolution strategies (ES) [1] have been shown to be very effective. In the ES algorithm, real variables are treated as real values. Real values can be coded in binary strings, but when such a coding is processed by the GA, a few problems arise. The most important of these is that if two real variables are dependent on each other, this means that all of the binary variables that code these real variables, have to be dependent on each other in some way. This gives additional overhead because the bits that code a single real variable also have to be seen as having some sort of dependence amongst each other. In the ES, a new type of mutation is applied that adapts the value for a real variable along a certain direction according to a normal distribution, using correlations with respect to other variables if desired.

There is no direct use of a probability distribution or density estimation in the ES algorithm. However, other approaches have made a first step in this direction. Given the EDA framework, we could say that the work by Servet, Trave–Massuyes and Stern in a first approach [30], the algorithm PBIL$_C$ by Sebag and Ducoulombier [29], which is a real valued version of PBIL using normal distributions, and the more flexible approach using a mixture model of normal distributions by Gallagher, Fream and Downs [11], are first attempts to expand the use of building and using probabilistic models for optimization to real continuous spaces. In all of those approaches however, the real variables are processed independently of each other. In other words, the probability distribution structure that is used in these algorithms, is a univariate one. In this paper, we present algorithms that use real density models as well as allow for the modelling of multivariate interactions between variables. All of this is done within the new framework that we present, but can be seen to be rather independent of it.

Our goal in this paper is to apply the search for linkage in terms of probabilistic models to continuous spaces. We therefore do not introduce any new way of data or linkage information processing, but show how we can expand the existing techniques to be used in the continuous case. This means that we focus mainly on density estimation. We require density models to use for modelling distributions of continuous random variables. The problem of estimating densities is well studied. An overview on fundamental issues in density estimation has for instance been given by Scott [28] and in a more application like manner by Bishop [5]. Based on such work, we can employ commonly used density models that have convenient properties in expanding from discrete to continuous density estimation based evolutionary algorithms.

The remainder of this paper is organized as follows. In section 2 we present the general framework for density estimation based evolutionary algorithms. Next, we go over previous work in section 3 and observe what is required to establish those algorithms. We need to know the demands on the density estimation model to be able to use probability theory for defining density estimation based evolutionary algorithms in general. In section 4, we redefine the algorithms for the case of discrete random variables to fit within the new framework. Subsequently, we use the requirements information in section 5 to define new algorithms using two different density

estimation models for continuous random variables. The two density estimation models we use in this paper are the normal distribution and the histogram distribution. Notes on the running times of the algorithms and topics for further research are given in section 6. We finish by presenting our conclusions in section 7. We present all of the algorithms in somewhat detailed pseudo–code. Conventions that are used in this paper for writing pseudo–code are given in appendix F. In general, involved mathematical derivations and definitions can be found in the appendices (A through F). Before moving to present the IDEA algorithmic framework, we first introduce some notation we employ in writing probability theory.

1.1 Probability theory notation

Discrete random variables are variables that can take on only a finite or at most a countably infinite number of values. We shall denote such variables by X_i. Using this notation, X or X^j denotes a vector of random variables X_i. The domain for each X_i is mostly taken to be the same as well as finite, so we shall denote the domain by $D^X = \{d_0^X, d_1^X, \ldots, d_{n_d-1}^X\}$. The size of the domain is thus denoted by $n_d = |D^X|$. In a special case, we have $n_d = 2$, $d_0 = 0$ and $d_1 = 1$ so that $D^X = \mathbb{B}$, the binary domain that was used in most EDAs so far. Now we can define the *multivariate joint probability mass function* for n discrete random variables $X_{j_0}, X_{j_1}, \ldots X_{j_{n-1}}$:

$$p_{j_0,j_1,\ldots,j_{n-1}}(d_{k_0}^X, d_{k_1}^X, \ldots, d_{k_{n-1}}^X) = P(X_{j_0} = d_{k_0}^X, X_{j_1} = d_{k_1}^X, \ldots, X_{j_{n-1}} = d_{k_{n-1}}^X) \tag{1}$$

$$\text{such that } \sum_{k_0=0}^{n_d-1} \sum_{k_1=0}^{n_d-1} \cdots \sum_{k_{n-1}=0}^{n_d-1} p_{j_0,j_1,\ldots,j_{n-1}}(d_{k_0}^X, d_{k_1}^X, \ldots, d_{k_{n-1}}^X) = 1$$

$$\text{where } \; p_{j_0,j_1,\ldots,j_{n-1}}(\cdot) \geq 0$$

For $n = 1$ this becomes the well known univariate probability mass function $p_i(d_k^X) = P(X_i = d_k^X)$ so that the sum over all domain variables of the probability function equals 1. As the set D^X is taken to be finite, we can straightforwardly make use of the mapping $i \leftrightarrow d_i^X$. By storing this mapping[1], the numbers $\overline{D^X} = \{0, 1, \ldots, n_d - 1\} \subseteq \mathbb{N}$ can be used. This clarifies and simplifies certain notations and implementations. Therefore, when referring to discrete random variables in the remainder of the paper, we shall use $\overline{D^X}$ instead of D^X. In the one–dimensional case for instance, this means that we write $p_j(k) = P(X_j = k)$ instead of $p_j(d_k^X) = P(X_j = d_k^X)$. As a general notation, we usually write $P(X_i)$ for p_i, making $P(X_i)$ a function. This notation is straightforwardly expanded to the multivariate case.

Continuous random variables are variables that can take on a continuum of values. We shall denote such variables by Y_i and let Y or Y^j once again be a vector of variables Y_i. The domain is mostly a subset of $\mathbb{R}$, so we shall denote the domain by $D^Y = [d_0^Y, d_1^Y]$ with $d_0^Y < d_1^Y$ and $(d_0^Y, d_1^Y) \in \mathbb{R}^2$. We can now define the *multivariate joint density function* $f(y_0, y_1, \ldots, y_{n-1})$ for n *continuous* random variables $Y_{j_0}, Y_{j_1}, \ldots Y_{j_{n-1}}$:

$$\int_{a_0}^{b_0} \int_{a_1}^{b_1} \cdots \int_{a_{n-1}}^{b_{n-1}} f_{j_0,j_1,\ldots,j_{n-1}}(y_0, y_1, \ldots, y_{n-1}) dy_0 dy_1 \ldots dy_{n-1} = P((Y_{j_0}, Y_{j_1}, \ldots, Y_{j_{n-1}}) \in A) \tag{2}$$

$$\text{such that } \int_{d_0^Y}^{d_1^Y} \int_{d_0^Y}^{d_1^Y} \cdots \int_{d_0^Y}^{d_1^Y} f_{j_0,j_1,\ldots,j_{n-1}}(y_0, y_1, \ldots, y_{n-1}) dy_0 dy_1 \ldots dy_{n-1} = 1$$

$$\text{where } \; f(\cdot) \geq 0 \;\text{ and }\; A = [a_0, b_0] \times [a_1, b_1] \times \ldots \times [a_{n-1}, b_{n-1}] \subseteq (D^Y)^n$$

For $n = 1$ this becomes the well known univariate density function $\int_a^b f_i(y)dy = P(a < Y_i < b)$, so that the integral over the domain of the positive density function equals 1. As with the discrete variables, we would like to employ a notation $P(Y_i)(x) = \int_x^x f_i(y)dy$ so that given a point a, we get

[1] When programming for instance, this mapping could be stored in an **array**.

the probability that Y_i is set to a, but we cannot do so because $P(Y_i = a) = \int_a^a f_i(y)dy = 0$. Since however we do computations on $f(y)$ in the continuous case, we introduce the notation $P(Y_i)$ for f_i, making $P(Y_i)$ a (density) function. This notation is again straightforwardly expanded to the multivariate case.

We may now note that when we discuss random variables in general, regardless of their domain type, we specify them as Z_i and write Z for a vector of variables Z_i. We can now give the definition for the conditional probability of one random variable Z_i being conditionally dependent on n other variables $Z_{j_0}, Z_{j_1}, \ldots, Z_{j_{n-1}}$. This is the most fundamental probability expression in the use of density estimation based evolutionary algorithms:

$$P(Z_i|Z_{j_0}, Z_{j_1}, \ldots, Z_{j_{n-1}}) = \frac{P(Z_i, Z_{j_0}, Z_{j_1}, \ldots, Z_{j_{n-1}})}{P(Z_{j_0}, Z_{j_1}, \ldots, Z_{j_{n-1}})} \tag{3}$$

2 Optimization using IDEAs

In this section we present the algorithmic framework for evolutionary optimization algorithms using density estimation techniques. We call this framework IDEA. In section 2.1 we specify the IDEA and show its connection with the EDA. In section 2.2, we introduce additional notation and place some remarks on the fundamental part of IDEA, namely the probability distribution.

2.1 The IDEA framework

The rationale for using distribution estmation in optimization was clearly stated by De Bonet, Isbell and Viola [6]. Assume we have a problem for which the function to be optimized is defined as $C(Z)$ with $Z = (Z_0, Z_1, \ldots, Z_{l-1})$, which without loss of generality we seek to minimize. If we have no information on $C(Z)$ in advance, we might as well assume a uniform distribution over the input $P(Z)$. Now denote a probability function that has a uniform distribution over all vectors Z with $C(Z) \leq \theta$ and a probability of 0 over all other vectors by $P^\theta(Z)$:

$$P^\theta(Z) = \begin{cases} \frac{1}{|\{Z'|C(Z') \leq \theta\}|} & \text{if } C(Z) \leq \theta \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

Sampling from distribution $P^\theta(Z)$ would thus give only samples with a function value of *at most* θ. This means that if we would somehow gain information that allows us to find $P^{\theta^*}(Z)$ where $\theta^* = \min_Z\{C(Z)\}$, a sample drawn from the probability distribution $P^{\theta^*}(Z)$ would give us an optimal solution vector Z^*.

Sampling and estimating distributions is well studied within probability theory. In terms of an iterated algorithm, given a collection of n vectors Z^i ($i \in \{0, 1, \ldots, n-1\}$) at iteration t, denote the largest function value of the best $\lfloor \tau n \rfloor$ vectors with $\tau \in [\frac{1}{n}, 1]$ by θ_t. We then pose a density estimation problem, where given the $\lfloor \tau n \rfloor$ vectors, we want to find, or best approximate, the density $P^{\theta_t}(Z)$. To this end, we use density estimation techniques from probability theory. In section 3 we will investigate what techniques have been employed so far in previously introduced algorithms for discrete (binary) spaces. Once we have this approximate distribution, we sample from it to find more samples that will hopefully all have a function value lower than θ_t and select again from the available samples to approximate in the next step $P^{\theta_{t+1}}(Z)$ with $\theta_{t+1} \leq \theta_t$. Making these steps explicit, we define the general *Iterated Density Estimation Evolutionary Algorithm* (IDEA) as follows:

IDEA(n, τ, m, $sel()$, $rep()$, $ter()$)

1 Generate a collection of n random vectors.
$\{Z^i \mid i \in \{0, \ldots, n-1\}\}$

2 Evaluate function values of the vectors in the collection.
$C(Z^i) \quad (i \in \{0, 1, \ldots, n-1\})$

3 Initialize the iteration counter.
$t \leftarrow 0$

4 Select $\lfloor \tau n \rfloor$ vectors.
$\{Z^{(S)i} \mid i \in \{0, \ldots, \lfloor \tau n \rfloor - 1\}\} \leftarrow sel() \quad (\tau \in [\frac{1}{n}, 1])$

5 Set θ_t to the worst function value value among the selected vectors.

6 Determine the probability distribution $\hat{P}^{\theta_t}(Z)$.

7 Generate m new vectors O by sampling from $\hat{P}^{\theta_t}(Z)$.

8 Incorporate the new vectors O in the collection.
$rep()$

9 Evaluate the new vectors in the collection ($\subseteq O$).

10 Update the iteration counter.
$t \leftarrow t + 1$

11 If the termination condition has not been satisfied, go to step 4.
if $\neg ter()$ ***then goto*** 4

12 Denote the amount of required iterations by t_{end}.
$t_{\text{end}} \leftarrow t$

If we refer to the iterations in the IDEA as *generations*, it becomes clear that the IDEA is a true evolutionary algorithm in the sense that a population of individuals is used. From this population, individuals are selected to generate new offspring with. Using these offspring along with the parent individuals and the current population, a new population is constructed.

In the IDEA algorithm, $\hat{P}^{\theta_t}(Z)$ stands for an approximation to the true distribution $P^{\theta_t}(Z)$. An approximation is required because the determined distribution is based upon samples. This means that even though depending on the density model and search algorithm used, it is possible we might achieve $\hat{P}^{\theta_t}(Z) = P^{\theta_t}(Z)$, in general this is not the case.

If we set m to $(n - \lfloor \tau n \rfloor)$, $sel()$ to selection by taking the best $\lfloor \tau n \rfloor$ vectors and $rep()$ to replacing the worst $(n - \lfloor \tau n \rfloor)$ vectors in the collection by the new sampled vectors, we can state that $\theta_{k+1} = \theta_k - \varepsilon$ with $\varepsilon \geq 0$. This assures that the search for θ^* is conveyed through a monotonically decreasing series $\theta_0 \geq \theta_1 \geq \ldots \geq \theta_{t_{\text{end}}}$. Hopefully we end up having $\theta_{t_{\text{end}}} = \theta^*$. We call an IDEA with m, $sel()$ and $rep()$ so chosen, a *monotonic* IDEA.

If we set m in the IDEA to n and take the m new vectors as the new collection, we obtain the EDA algorithm that was originally presented by Mühlenbein, Mahnig and Rodriguez [22] as a general estimation of distribution algorithm. In the EDA however, the probability plateau θ_t cannot be enforced according to the rationale we just presented. Therefore we introduce IDEA and note how EDA is an instance of this new framework.

In figure 1 we give a graphical example of the general idea according to which IDEAs go to work. We have a continuous problem with $D^Y = [-10, 10]$ and $l = 2$. The simple function $C(Y)$ we seek to minimize has a uniquely defined minimum at $(Y_1^*, Y_2^*) = (0, 0)$. The $C(Y)$ function plots demonstrate the idea of having the threshold θ_t for the probability distribution $P^{\theta_t}(Y)$. Over the iterations (with $t_{\text{end}} = 30$) we observe that θ_t goes to the minimum value of $C(Y)$, leading to convergence to the optimal point. The contour plots show the convergence of the distribution of the vectors in the collection. The vectors were determined at the end of each iteration.

New IDEAs have been introduced under different names. The most important discriminant among these algorithms has been the way in which step 6 in IDEA is performed. The reason for this becomes apparent when looking at the case of discrete random variables. In the case when we attempt to best estimate the complete joint probability, we require to count from the selected $\lfloor \tau n \rfloor$ vectors the occurrences of every combination of assignments of every element from D^X to every X_i. This leads to $(n_d)^l$ combinations to check out for each selected vector, which is an exponential amount. The amount of vectors we would require to justify this estimation would

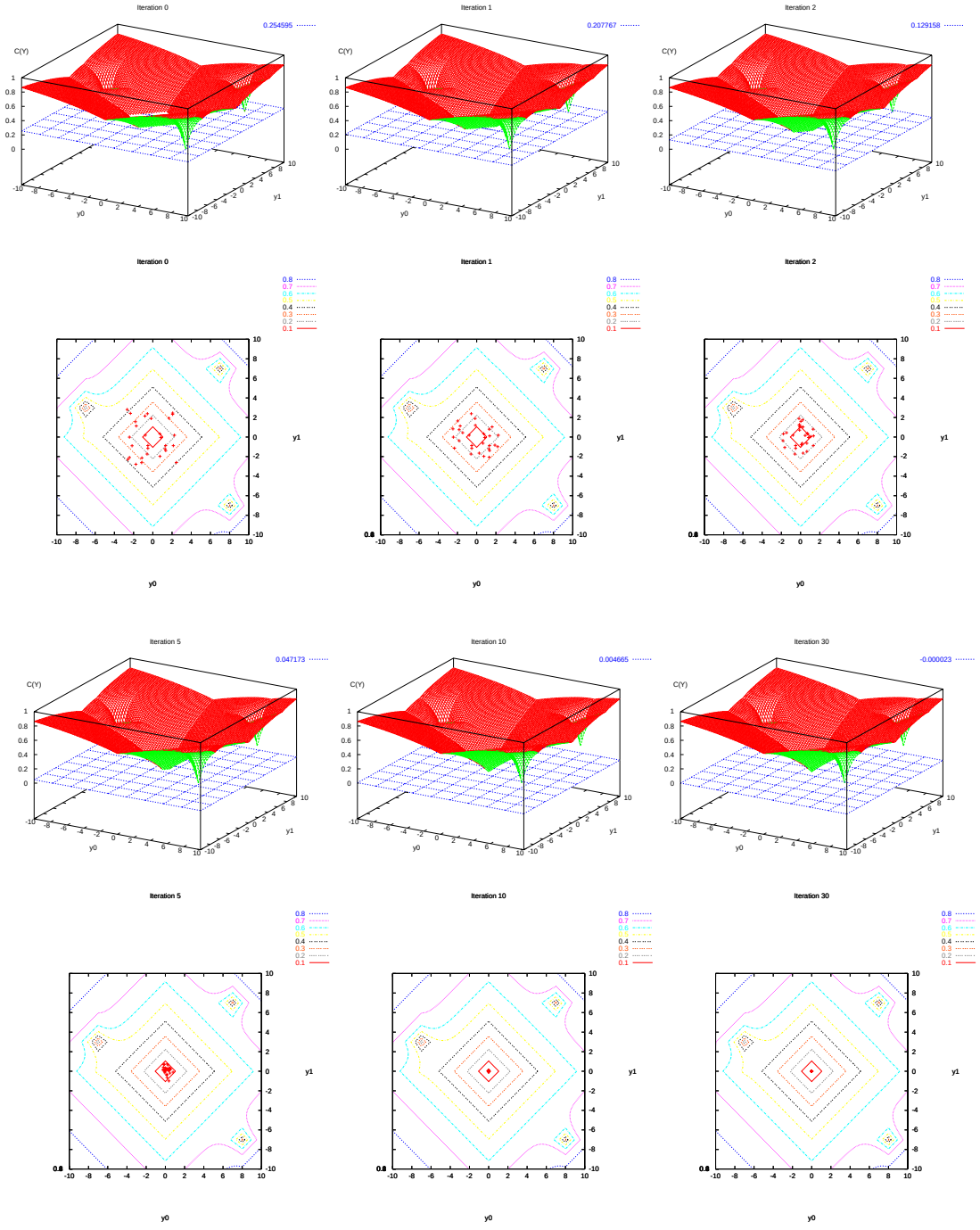

Figure 1: An example run of a monotonic IDEA on a simple test function with a fixed probability distribution structure such that $\hat{P}(Y) = \hat{P}(Y_1|Y_2)\hat{P}(Y_2)$, $\tau = 0.5$, $n = 30$ and $m = 15$.

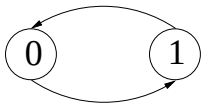

Figure 2: An unfeasible PDS graph on two variables: $\hat{P}(Z_0|Z_1)\hat{P}(Z_1|Z_0)$

also be enormous. We would be better off trying every possible vector X. The other extreme would be to regard every variable X_i independently. Such has been the approach by algorithms such as PBIL [2] and UMDA [26]. However, just as the notion of *linkage* was first acknowledged by Holland to be of importance [18], it has been shown by Bosman and Thierens [7] that IDEA algorithms require to find higher order density structures as an approximation to $P^{\theta_t}(X)$ as well in order to find solutions to higher order building block problems. This is why quick and efficient density estimation is of the greatest importance to IDEA algorithms and why multiple different algorithms have been introduced with different types of distribution structures. In section 3 we shall go over these algorithms and note what is required to rationally find such a distribution by observing what has been done so far. First however, we introduce some notation regarding these probability distributions and introduce the general notion of how these distributions are used in IDEAs.

2.2 Probability distribution issues

After updating the probability density structure, we can write this structure in the following general form:

$$\hat{P}_\pi(Z) = \prod_{i=0}^{l-1} \hat{P}(Z_i|Z_{\pi(i)_0}, Z_{\pi(i)_1}, \ldots, Z_{\pi(i)_{|\pi(i)|-1}}) \tag{5}$$

Where Z_i represents the i–th random variable in vector Z and $\pi(i)$ is a function that returns a vector $\pi(i) = (\pi(i)_0, \pi(i)_1, \ldots, \pi(i)_{|\pi(i)|-1})$ of indices denoting the variables that Z_i is conditionally dependent on.

We call the graph that results when drawing the variables Z_i as nodes and drawing an arc from node i to j when Z_j is conditionally dependent on Z_i, the *Probability Density Structure* (PDS) graph. The PDS graph needs to be acyclic. This can be seen from the simple example in the discrete case when we might want to model $\hat{P}(X_0|X_1)\hat{P}(X_1|X_0)$ as shown in figure 2, with $\overline{D^X} = \mathbb{B} = \{0,1\}$. This is not a valid probability distribution over X. From an IDEA standpoint, even though we will be able to determine the two required probabilities by counting, we will not be able to generate new samples according to this distribution because we need to set X_1 to 0 or 1 before we can sample X_0, but we also need to set X_0 to 0 or 1 before we can sample X_1. To see that an acyclic graph is sufficient, we note that because $P(Z_{j_0}|Z_{j_1}, Z_{j_2}, \ldots, Z_{j_{n-1}}) \cdot P(Z_{j_1}, Z_{j_2}, \ldots, Z_{j_{n-1}}) = P(Z_{j_0}, Z_{j_1}, \ldots, Z_{j_{n-1}})$, the complete joint probability can be written as follows (see figure 3 as an example):

$$P(Z_0, Z_1, \ldots, Z_{l-1}) = \prod_{i=0}^{l-1} P(Z_i|Z_{i+1}, Z_{i+2}, \ldots, Z_{l-1}) \tag{6}$$

To formalize the notion of having an acyclic graph, we state that we require to have a vector $\boldsymbol{\omega} = (\omega_0, \omega_1, \ldots, \omega_{l-1})$, so that we can rewrite the general form of the approximated probability density structure:

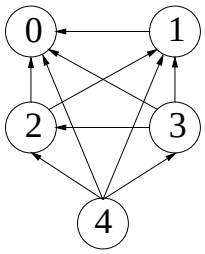

Figure 3: A complete PDS graph on five variables: $\hat{P}(Z_0|Z_1,Z_2,Z_3,Z_4)\hat{P}(Z_1|Z_2,Z_3,Z_4)\cdot \hat{P}(Z_2|Z_3,Z_4)\hat{P}(Z_3|Z_4)\hat{P}(Z_4) = \hat{P}(Z_0,Z_1,Z_2,Z_3,Z_4)$

$$\hat{P}_{\pi,\boldsymbol{\omega}}(Z) = \prod_{i=0}^{l-1} \hat{P}(Z_{\omega_i}|Z_{\pi(\omega_i)_0}, Z_{\pi(\omega_i)_1}, \ldots, Z_{\pi(\omega_i)_{|\pi(\omega_i)|-1}}) \tag{7}$$

$$\text{such that } \forall_{i\in\{0,1,\ldots,l-1\}}\langle \omega_i \in \{0,1,\ldots,l-1\} \wedge \forall_{k\in\{0,1,\ldots,l-1\}-\{i\}}\langle \omega_i \neq \omega_k\rangle\rangle$$

$$\forall_{i\in\{0,1,\ldots,l-1\}}\langle \forall_{k\in\pi(\omega_i)}\langle k \in \{\omega_{i+1},\omega_{i+2},\ldots,\omega_{l-1}\}\rangle\rangle$$

Even though this definition is tricky and defines what permutations in combination with parent functions are allowed, the construction method for the PDS in the form of $(\pi,\boldsymbol{\omega})$ will in general only create feasible structures.

We should note that next to finding a distribution, we must also be able to *sample* from $\hat{P}_{\pi,\boldsymbol{\omega}}(Z)$. It is clear to see that because the PDS graph is acyclic, there is an ordering of the variables such that we can always sample a value for a variable that is conditionally dependent only on variables we have already sampled for. This ordering is given by the ω_i from equation 7. Note that given the conditional variables, the probability density for a variable takes on a one dimensional form, since it is given in the form of equation 7. From such a one dimensional density it is sometimes straightforward to sample. Still, the method of sampling is dependent on the type of density estimation that is used, so we parameterize this method in the IDEA.

Once more we note that step 6 in the IDEA is an important discriminant. A general discrimination can be made between algorithms that use a factorization a priori and the ones that search for a probability density structure. Note that the factorization we refer to is not the full specification of $\hat{P}_{\pi,\boldsymbol{\omega}}(Z)$, but only of the structure of $\hat{P}_{\pi,\boldsymbol{\omega}}(Z)$. This structure is fully specified by a pair $(\pi,\boldsymbol{\omega})$ that is subject to the constraints from equation 7. The actual probability values are found through estimations on the selected points. This means that even though we might have the correct structure over all points, say the complete joint probability, we yet have to find the probability *values* over the domains that assign probabilities of 1.0 to the assignment of the right domain values to the variables. This estimating however needs always to be done. If we specify a search procedure for finding $(\pi,\boldsymbol{\omega})$ by $sea()$, we note that algorithms that use a factorized probability distribution simply use a static $sea()$ procedure that always returns the same structure. Therefore we need not to introduce two new algorithms, but we can suffice by elaborating on the first framework to end up with the final specification of IDEA:

IDEA(n, τ, m, $sel()$, $rep()$, $ter()$, $sea()$, $est()$, $sam()$)
1 ... 5 *Identical to* IDEA(n, τ, m, $sel()$, $rep()$, $ter()$)
6 Determine the probability distribution $\hat{P}^{\theta_t}_{\pi,\boldsymbol{\omega}}(Z)$ in two steps:
6.1 Find a PDS subject to the constraints from equation 7.
$(\pi, \boldsymbol{\omega}) \leftarrow sea()$
6.2 Estimate the density functions.
$\{\hat{P}(Z_{\omega_i}|Z_{\pi(\omega_i)_0}, Z_{\pi(\omega_i)_1}, \ldots, Z_{\pi(\omega_i)_{|\pi(\omega_i)|-1}}) \mid i \in \{0, \ldots l-1\}\} \leftarrow est()$
7 Generate m new vectors O by sampling from $\hat{P}^{\theta_t}_{\pi,\boldsymbol{\omega}}(Z)$ in three steps:
7.1 $O \leftarrow \emptyset$
7.2 ***repeat*** m ***times***
7.3 $O \leftarrow O \cup sam()$
8 ... 12 *Identical to* IDEA(n, τ, m, $sel()$, $rep()$, $ter()$)

Concluding, we now have a framework that, given implementations for the functions that are parameters and the values for the other parameters, completely specifies density estimation based evolutionary algorithms. This makes it clear what variant of the IDEA is exactly used, as any parameter that is not specified leaves it unclear as to what the algorithm does. In addition however, add–ons may be introduced on top of the framework. For instance a local search heuristic can be added, which makes the algorithm hybrid.

3 Finding requirements from existing EDAs: Previous Work

The use of distribution estimation in optimization algorithms has been introduced through various approaches. These approaches use various types of density structures and have mostly been applied to discrete and in particular binary random variables. We seek to investigate what approaches to the search for a PDS have been used. By making this algorithmic part explicit, we isolate what we require from certain distribution models in order to use them in the case of continuous random variables. In all cases, we will find that the metric to guide the search for a PDS is the entropy measure over random variables. We shall write h_i and h_{ij} respectively for the univariate and bivariate joint empirical entropy (either discrete or differential) over variables Z_i and Z_j. In the pseudo–code that we present in the remainder of this paper, we assume that h_i ($\forall_{i \in \{0,1,\ldots,l-1\}}$) and h_{ij} ($\forall_{(i,j) \in \{0,1,\ldots,l-1\}^2}$) are global variables, for which we do not allocate additional memory in the algorithms.

3.1 PBIL

The basic idea in PBIL by Baluja and Caruana [2], is to update a single probability vector $P[i], i \in \{0, 1, \ldots, l-1\}$. This vector has an entry for every binary random variable. Each entry denotes the probability that the corresponding variable should be set to 1. The update procedure is performed by moving the probability vector toward a few (M) of the best vectors in the population. This is achieved by updating each entry *independently* of the others at the rate of a certain learning parameter η:

for $j \leftarrow 0$ **to** $M-1$ **do**
  for $i \leftarrow 0$ **to** $l-1$ **do**
    $P[i] \leftarrow P[i](1-\eta) + X_i^{(S)j}\eta$

The requirements for the density model are nothing special because the density structure is simply the univariate distribution. In other words, we need no special aspects of the density model to compute $(\pi, \boldsymbol{\omega})$. We do note however, that PBIL uses a different approach to *updating* the distribution. In PBIL, a memory is used as the information from previous iterations is preserved by multiplication of the probability vector entries with $(1-\eta)$. This is different from our IDEA algorithmic model as we use only the population to update the distribution from.

In extensions of the PBIL framework to continuous spaces [30, 29, 11], all variables continue to be regarded *independently* of one another. Even though the density model for each variable may be different, the requirements on these models are nothing special from the viewpoint of searching for the structure $(\pi, \boldsymbol{\omega})$. Concluding, the requirements for implementing the search algorithm employed by PBIL are *none*.

3.2 MIMIC

In MIMIC, a framework by De Bonet, Isbell and Viola [6], all binary variables except for one are conditionally dependent on exactly *one* other variable. These dependencies are ordered in such a way that the resulting PDS is a *chain*:

$$\hat{P}(Z) = \left(\prod_{i=0}^{l-2} \hat{P}(Z_{j_i} | Z_{j_{i+1}}) \right) \hat{P}(Z_{j_{l-1}}) \tag{8}$$

$$\text{such that } \forall_{(i,k) \in \{0,1,\ldots,l-1\}^2} \langle i \neq k \rightarrow j_i \neq j_k \rangle$$

To determine the j_i that constitute the chain, the agreement between the complete joint distribution and $\hat{P}(X)$ is maximized. This comes down to minimizing the following expression when using the Kullback–Leibler divergence, written using *discrete* variables X_i:

$$J(X) = \left(\sum_{i=0}^{l-2} H(X_{j_i} | X_{j_{i+1}}) \right) + H(X_{j_{l-1}}) \tag{9}$$

In this expression, $H(X_i)$ stands for the entropy of variable X_i. The notion of entropy was first introduced by Shannon when presenting his information theory [31]. It is a measure of the average amount of information conveyed per message. The multivariate definition in n dimensions for discrete random variables $X_{j_0}, X_{j_1}, \ldots, X_{j_{n-1}}$, is the following:

$$H(X_{j_0}, X_{j_1}, \ldots, X_{j_{n-1}}) = \tag{10}$$

$$- \sum_{k_0=0}^{n_d-1} \sum_{k_1=0}^{n_d-1} \cdots \sum_{k_{n-1}=0}^{n_d-1} p_{j_0, j_1, \ldots, j_{n-1}}(k_0, k_1, \ldots, k_{n-1}) \ln(p_{j_0, j_1, \ldots, j_{n-1}}(k_0, k_1, \ldots, k_{n-1}))$$

Furthermore, we require the notion of *conditional* entropy. For *one* discrete random variable X_i conditioned on n other discrete random variables $X_{j_0}, X_{j_1}, \ldots, X_{j_{n-1}}$, this definition equals:

$$H(X_i | X_{j_0}, X_{j_1}, \ldots, X_{j_{n-1}}) = \tag{11}$$

$$H(X_i, X_{j_0}, X_{j_1}, \ldots, X_{j_{n-1}}) - H(X_{j_0}, X_{j_1}, \ldots, X_{j_{n-1}})$$

The definition in equation 10 cannot be used for *continuous* random variables. To this end, we require the definition of *differential entropy*, which is the continuous variant of the entropy measure. This entropy measure is usually denoted with a small h instead of a capital letter H:

$$h(Y_{j_0}, Y_{j_1}, \ldots, Y_{j_{n-1}}) = \tag{12}$$

$$- \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \cdots \int_{-\infty}^{\infty} f_{j_0, j_1, \ldots, j_{n-1}}(y_0, y_1, \ldots, y_{n-1}) \ln(f_{j_0, j_1, \ldots, j_{n-1}}(y_0, y_1, \ldots, y_{n-1})) dy_0 dy_1 \ldots dy_{n-1}$$

The minimization of the divergence in equation 9 can be done in a greedy way, which is the approach used in MIMIC. First, the variable with minimal unconditional entropy is selected. Then, new variables are subsequently selected by choosing the one with minimal conditional entropy, given the previously selected variable. This gives an $\mathcal{O}(l^2)$ algorithm (without computing the entropies):

$\omega_{l-1} \leftarrow arg\ \min_j\{\hat{h}(Z_j) \mid j \in \{0,1,\ldots,l-1\}\}$
$\pi(\omega_{l-1}) \leftarrow \emptyset$
for $i \leftarrow l-2$ **downto** 0 **do**
    $\omega_i \leftarrow arg\ \min_j\{\hat{h}(Z_j, Z_{\omega_{i+1}}) - \hat{h}(Z_{\omega_{i+1}}) \mid$
        $j \in \{0,1,\ldots,l-1\} - \{\omega_{i+1}, \omega_{i+2}, \ldots, \omega_{l-1}\}\}$
    $\pi(\omega_i) \leftarrow \omega_{i+1}$

In the above, we have written $\hat{h}(Z_i)$ instead of $h(Z_i)$ to remark that the entropy measure used in an actual algorithm based on vectors $Z^{(S)j}$ is the *empirical entropy*. The empirical entropy differs from the true entropy in that the empirical entropy is based upon samples instead of upon the true distribution.

Concluding, the one requirement we have on the distribution model to be used in order to apply the MIMIC chain search procedure, is that we are able to compute one dimensional and two dimensional (joint) empirical entropies.

3.3 Optimal Dependency Trees

In the approach using optimal dependency trees by Baluja and Davies [3], all variables except for one may still be conditionally dependent on exactly one other variable, but multiple variables may be dependent on one and the same variable, yielding a tree structure:

$$\hat{P}(Z) = \left(\prod_{i=0}^{l-2} \hat{P}(Z_{j_i} | Z_{e_i})\right) \hat{P}(Z_{j_{l-1}}) \tag{13}$$

$$\text{such that } \forall_{(i,k)\in\{0,1,\ldots,l-1\}^2} \langle (i \neq k \rightarrow j_i \neq j_k) \wedge e_i \in \{j_{i+1} \ldots j_{n-1}\}\rangle$$

To determine the j_i that constitute the tree, the Kullback–Leibler divergence is again minimized. In the algorithm, the measure named *mutual information* is used. The definition of mutual information for two random variables Z_i and Z_j can be expressed in terms of the entropy measure we saw earlier in section 3.2:

$$I(Z_i, Z_j) = h(Z_i) + h(Z_j) - h(Z_i, Z_j) \tag{14}$$

The greater the mutual information, the greater the interaction between variables Z_i and Z_j. As a result, the algorithm builds the tree by *maximizing* the mutual information measure. The algorithm runs in $\mathcal{O}(l^2)$ time (without computing the entropies):

$b^t \leftarrow$ ***new array*** of ***integer*** with size $l-1$
$\omega_{l-1} \leftarrow$ RandomNumber(l)
$\pi(\omega_{l-1}) \leftarrow \emptyset$
$b^t[i] \leftarrow \omega_{l-1} \quad (\forall_{i\in\{0,1,\ldots,l-1\}-\{\omega_{l-1}\}})$
for $i \leftarrow l-2$ ***downto*** 0 ***do***
    $\omega_i \leftarrow arg\ \max_j\{\hat{h}(Z_j) + \hat{h}(Z_{b^t[j]}) - \hat{h}(Z_j, Z_{b^t[j]}) \mid$
        $j \in \{0,1,\ldots,l-1\} - \{\omega_{i+1}, \omega_{i+2}, \ldots, \omega_{l-1}\}\}$
    $\pi(\omega_i) \leftarrow b^t[\omega_i]$
    if $\hat{h}(Z_j) + \hat{h}(Z_{b^t[j]}) - \hat{h}(Z_j, Z_{b^t[j]}) < \hat{h}(Z_j) + \hat{h}(Z_{\omega_i}) - \hat{h}(Z_j, Z_{\omega_i})$
      then $b^t[j] \leftarrow \omega_i \quad (\forall_{j\in\{0,1,\ldots,l-1\}-\{\omega_i,\omega_{i+1},\ldots,\omega_{l-1}\}})$

In the above, we have used algorithm RandomNumber(x), which is assumed to return a (pseudo) random integer from the set $\{0,1,\ldots,x-1\}$. We may conclude that because we only require (empirical) entropy measures, we require nothing new for implementing the optimal dependency tree search procedure in addition to the requirements for implementing the MIMIC chain search procedure. The only requirement is thus to be able to determine $\hat{h}(Z_j)$ and $\hat{h}(Z_i, Z_j)$.

3.4 BOA

One of the most general approaches, allowing the most flexible network structure, was proposed by Pelikan, Goldberg and Cantú–Paz [24]. The algorithm named BOA uses a PDS in which each node may have up to κ successors, allowing variables now to be conditionally dependent on *sets* of variables. Because of the very general structure, we may write this using the same variables as before in equation 7 and with the introduction of *one* additional constraint:

$$\hat{P}(Z) = \prod_{i=0}^{l-1} \hat{P}(Z_{\omega_i} | Z_{\pi(\omega_i)_0}, Z_{\pi(\omega_i)_1}, \ldots, Z_{\pi(\omega_i)_{|\pi(\omega_i)|-1}}) \tag{15}$$

$$\begin{aligned} \text{such that } & \forall_{i \in \{0,1,\ldots,l-1\}} \langle \omega_i \in \{0,1,\ldots,l-1\} \wedge \forall_{k \in \{0,1,\ldots,l-1\}-\{i\}} \langle \omega_i \neq \omega_k \rangle \rangle \\ & \forall_{i \in \{0,1,\ldots,l-1\}} \langle \forall_{k \in \pi(\omega_i)} \langle k \in \{\omega_{i+1}, \omega_{i+2}, \ldots, \omega_{l-1}\} \rangle \rangle \\ & \forall_{i \in \{0,1,\ldots,l-1\}} \langle |\pi(i)| \leq \kappa \rangle \end{aligned}$$

As was the case in MIMIC and the use of the optimal dependency trees, a metric is required that should be maximized or minimized so as to find a network structure. In the original implementation of BOA, the so called $K2$ metric is used, which is a metric that should be maximized. In contrast to that approach and more in the light of previously dicussed approaches, we use a metric based on conditional entropies. In appendix A, the difference between the K2 metric and the entropy metric is elaborated upon. In the general graph search algorithm that will result, we will attempt to *minimize* the conditional entropy of the PDS imposed by equation 15:

$$\breve{\psi}((\pi, \boldsymbol{\omega})) = \min_{(\pi,\boldsymbol{\omega})} \left\{ \sum_{i=0}^{l-1} \hat{h}(X_{\omega_i} | X_{\pi(\omega_i)_0}, X_{\pi(\omega_i)_1}, \ldots, X_{\pi(\omega_i)_{|\pi(\omega_i)|-1}}) \right\} \tag{16}$$

Note the obvious similarity with the metric used by MIMIC which we discussed in section 3.2. It follows from this metric that we require again only to be able to compute empirical entropies. The only difference is that this time we are required to be able to compute multivariate entropies for the applied density model in $\kappa + 1$ dimensions.

The only thing that is now missing is the algorithm used in BOA to find a PDS. To this end, three different cases are distinguished for κ. When $\kappa = 0$, the PDS regards every variable univariately. In other words, we end up with a fixed network in which $\forall_i \langle \pi(i) = \emptyset \rangle$. In the case of $\kappa = 1$, there is a polynomial algorithm by Edmonds [10] to construct the optimal network. A direct combinatorial proof of the correctness of the algorithm was given by Karp [20]. The definitions that are required to understand the algorithm and fill in the details, are based on the work by Karp and are clarified in appendix B. If $\kappa > 1$, the problem is $\mathcal{NP}$–complete for the proposed metric [17]. Therefore, a greedy algorithm can be used as has been done in the MIMIC approach. The original implementation of BOA starts with a PDS with $\forall_i \langle \pi(i) = \emptyset \rangle$. Next, from the set of arcs that are not yet in the PDS graph, the arc that moves the metric the most in the direction of the optimum is selected and added to the PDS graph. This process is repeated until the metric can no longer be improved. These three cases constitute a search algorithm that runs in $\mathcal{O}(\kappa l^3)$ time (without computing the entropies). This time bound is due to the fact that for $\kappa = 0$, the running time is $\mathcal{O}(l)$, for $\kappa = 1$, the running time is $\mathcal{O}(l^3)$ and for $\kappa > 1$, the running time is $\mathcal{O}(\kappa l^3)$. Along with these running times, we note that Tarjan [32] has given a more efficient implementation of finding optimal branchings for the case when $\kappa = 1$, which runs in $\mathcal{O}(l(\log(l))^2 + l^2)$ time. Seen to the equivalent running time of the general graph search algorithm with respect to the case when $\kappa > 1$ is taken as a constant, this is of no influence on the running time. We are now ready to present the algorithm itself:

```
if κ = 0
  then i ← 0 to l − 1 do
       ω_i ← i
       π(ω_i) ← ∅
else if κ = 1
  then P ← ConstructProblem()
       G = (V, A) ← ({0, 1, ..., l − 1}, {0, 1, ..., l − 1}^2 − ⋃_{i=0}^{l−1} {(i, i)})
       set s(i, j) ← i and t(i, j) ← j
       compute c(i, j) for every arc (i, j) so that min_{(i,j)}{c(i, j) | (i, j) ∈ A} = 1
       P ← (G, s, t, c)
       H = (V, A_H) ← ComputeCriticalGraph(P)
       if acyclic(H)
          then B ← A_H
          else P̄ ← ConstructDerivedProblem(P, H)
               B̄ ← repeat computation, but start with P̄
               B ← optimum branching for P corresponding to B̄ for P
       perform topological sort on B to find (π, ω)
else
  then G = (V, A) ← ({0, 1, ..., l − 1}, ∅)
       compute change(i, j) for every arc (i, j)
       ψ̆ ← Σ_{i=0}^{l−1} ĥ(Z_i)
       while ¬finished do
            (v_1, v_2) ← arg min_{(i,j)}{change(i, j) | allowed(i, j) ∧ (i, j) ∈ V^2}
            if change(v_1, v_2) > 0
               then breakwhile
            mark (v_1, v_2) and arcs (i, j) that make cycles as not allowed
            A ← A ∪ (v_1, v_2)
            if v_2 has κ parents
               then mark each arc (i, v_2), i ∈ {0, 1, ..., l − 1} as not allowed
            ψ̆ ← Σ_{i=0}^{l−1} ĥ(Z_i|{Z_j|(j, i) ∈ A})
            compute change(i, j) for every arc (i, j) with allowed(i, j) = true
       perform topological sort on G to find (π, ω)
```

3.5 UMDA, BMDA and FDA

The EDA framework has resulted from insights gained through the introduction of algorithms along the line of UMDA, BMDA and FDA. The UMDA by Mühlenbein and Paaß [23] uses the same type of distribution as does the PBIL algorithm discussed in section 3.1. The main difference is that a memory is used in PBIL, whereas in UMDA the collection of vectors is used mainly. This latter approach is therefore more in concordance with our IDEA approach. It follows from the use of only the univariate distribution that, for the same reasons as PBIL, we have no additional requirements for the UMDA.

The BMDA by Pelikan and Mühlenbein [26] covers second order interactions just as is the case for MIMIC and the optimal dependency trees discussed in sections 3.2 and 3.3 respectively. However, the structure used in BMDA is the *most* general one for second order relations, meaning that the PDS graph has the form of a collection of trees as used in the optimal dependency trees approach. If we look even closer, we observe that because the BOA from section 3.4 also uses the most general approach for modeling relations with maximum order κ, the model for BMDA equals the model for the BOA with $\kappa = 1$. As noted elsewhere [24], the BOA covers both the UMDA and the BMDA. Hence, we may decline from going into further detail on this algorithm.

Finally, the FDA by Mühlenbein, Mahnig and Rodriguez [22] is of importance. In this algorithm a complete factorization of the distribution, specified by n_s sets s_i with $i \in \{0, 1, \ldots, n_s - 1\}$, is taken as input and used to build the PDS:

$$\hat{P}(Z) = \left(\prod_{i=1}^{n_s-1} \hat{P}\left(\prod_{Z_j \in b_i} Z_j \,\middle|\, \prod_{Z_j \in c_i} Z_j \right) \right) \hat{P}\left(\prod_{Z_j \in b_0} Z_j \right) \tag{17}$$

$$\text{where } \begin{cases} c_i & = s_i \cap d_{i-1} \\ b_i & = s_i \;-\; d_{i-1} \\ d_i & = \begin{cases} \bigcup_{j=0}^{i} s_j & \text{if } i \geq 0 \\ \emptyset & \text{otherwise} \end{cases} \\ \bigcup_{i=0}^{n_s-1} s_i & = \{Z_0, Z_1, \ldots, Z_n\} \end{cases}$$

Using an actual factorization for the problem at hand, FDA can very efficiently perform optimization. As this factorization is given a priori, it can be written in terms of the general definition of equation 7. This means that we have *no* additional requirements for implementing FDA in our IDEA framework, since the PDS that is used, is fixed at the outset. In other words, a *sea*() function that returns the PDS given as input by the user, suffices.

3.6 CGA and ECGA

The compact genetic algorithm (cGA) was introduced by Harik, Lobo and Goldberg [16] in 1997. The main idea behind this algorithm, is that it is able to mimic the order–1 behaviour of the simple genetic algorithm (sGA) with uniform crossover. Again, just as in the UMDA and PBIL, for each binary variable, an entry in a probability vector is created that is initially set to 0.5. The algorithm then creates tournaments and moves the probability vector toward the winner of the tournament. The way in which the probability vector is adapted, is dependent on the assumed population size in the sGA that the cGA mimics. Again, because of the fact that a univariate distribution is used, we do not have any additional requirements.

An extension to the cGA, named the extended compact genetic algorithm (ECGA), was given by Harik [15] in 1999. In the ECGA, the PDS consists of joint structures which are all considered univariately. This means that variables can be grouped together in a joint distribution, but they are processed independently of any other variables. Such a PDS follows the idea of having non–overlapping building blocks which must be processed as a whole in order to solve a problem. Such problems have been shown to be solvable by the ECGA in a very efficient manner. The search algorithm for the PDS is a greedy algorithm that seeks to combine two sets of variables into a single new set as long as a certain metric can still be increased. Initially, all of the variables are placed in a singleton set. The metric used is based upon the minimium discription length from information theory (see for instance [8]). The advantage of this metric is that is penalizes models that are more complex than required. This is important because the density estimation step is an approximation to the true density because of the model used as well as the fact that we are using a limited set of samples.

The metric used in the ECGA (as well as the one used in BMDA) differs from the entropy measure we proposed to use in all of the other cases. The search algorithm is a special case of finding a multivariate PDS. The most general structure is used within the FDA and the BOA. Therefore, even though it deserves full attention, especially with respect to completely decomposable problems, we shall see this method as being covered by the level of allowed interactions in a general PDS search and focus on that. The ECGA search algorithm, along with the metric used, can however easily be incorporated in the IDEA framework. Even though the used metric has important and interesting properties, we shall refrain from exploring it further in this paper.

4 IDEAs for discrete random variables

Having noted the type of information that is required in previously created EDAs for binary random variables, we are ready to revisit the work in terms of the IDEA framework. In this section, we show how in the case of discrete random variables, the algorithms discussed in section 3

can be implemented. We directly cast these algorithms in the form of general discrete random variables as introduced in section 1 as opposed to the special case of the binary random variables that was implicitly used in most of the discussed algorithms. The running times of the algorithms are stated in section 6.

Even though the incorporation of the pseudo–code for the search procedures that return the PDS $(\pi, \boldsymbol{\omega})$ in the case of discrete random variables is something of a repetition of earlier work, it serves three purposes:

1. It gives a clear overview of the main ingredients of the algorithms presented in the literature so far by using pseudo–code to clearly specify the operational procedures.
2. It shows how those algorithms can be incorporated within the IDEA framework.
3. It provides a reference to check back with when defining the algorithms for continuous random variables, allowing to note the similarities and see how *density estimation* lies at the heart of these methods.

In the discrete case, we compute probabilities by counting frequencies as we would in making a histogram. We call these probabilities *empiric* probabilities and denote them by $\hat{p}(\cdot)$. The basis of computing probabilities in these algorithms in the discrete case lies within the use of equation 3 and the following:

$$\hat{p}_{j_0,j_1,\ldots,j_{n-1}}(k_0,k_1,\ldots,k_{n-1}) = \frac{m(\boldsymbol{j},\boldsymbol{k})}{\lfloor \tau n \rfloor} \tag{18}$$

$$\text{where} \begin{cases} \boldsymbol{j} & = (j_0, j_1, \ldots, j_{n-1}) \\ \boldsymbol{k} & = (k_0, k_1, \ldots, k_{n-1}) \\ m(\boldsymbol{\nu}, \boldsymbol{\lambda}) & = \sum_{q=0}^{\lfloor \tau n \rfloor} \begin{cases} 1 & \text{if } \forall_{i \in \{0,1,\ldots,|\boldsymbol{\lambda}|-1\}} \langle X_{\nu_i}^{(S)q} = \lambda_i \rangle \\ 0 & \text{otherwise} \end{cases} \end{cases}$$

We need to compute the empiric probabilities in order to sample from the conditional probabilities imposed by the PDS. We do this by using *bins* (b_i and b_i^p, $i \in \{0, 1, \ldots, l-1\}$) in which we perform a frequency count exactly as imposed by function $m(\cdot,\cdot)$ from equation 18. This leads us to define algorithm DIEST to estimate the density functions in the discrete case once the PDS is known:

DIEST()
1 **for** $i \leftarrow 0$ **to** $l-1$ **do**
  1.1 $b_{\omega_i} \leftarrow$ **new array** of **real** in $|\pi(\omega_i)|+1$ dimensions with size $n_d \times n_d \times \ldots \times n_d$
  1.2 ${b^p}_{\omega_i} \leftarrow$ **new array** of **real** in $|\pi(\omega_i)|$ dimensions with size $n_d \times n_d \times \ldots \times n_d$
  1.3 **for** $(a_0, a_1, \ldots, a_{|\pi(\omega_i)|}) \leftarrow (0,0,\ldots,0)$ **to** $(n_d-1, n_d-1, \ldots, n_d-1)$ **in crossproduct do**
    1.3.1 $b_{\omega_i}[a_0, a_1, \ldots, a_{|\pi(\omega_i)|}] \leftarrow 0$
  1.4 **for** $(a_0, a_1, \ldots, a_{|\pi(\omega_i)|-1}) \leftarrow (0,0,\ldots,0)$ **to** $(n_d-1, n_d-1, \ldots, n_d-1)$ **in crossproduct do**
    1.4.1 $b^p_{\omega_i}[a_0, a_1, \ldots, a_{|\pi(\omega_i)|-1}] \leftarrow 0$
  1.5 **for** $k \leftarrow 0$ **to** $\lfloor \tau n \rfloor - 1$ **do**
    1.5.1 $a_0 \leftarrow X_{\omega_i}^{(S)k}$
    1.5.2 **for** $q \leftarrow 0$ **to** $|\pi(\omega_i)|-1$ **do**
      1.5.2.1 $a_{q+1} \leftarrow X^{(S)k}_{\pi(\omega_i)_q}$
    1.5.3 $b_{\omega_i}[a_0, a_1, \ldots, a_{|\pi(\omega_i)|}] \leftarrow b_{\omega_i}[a_0, a_1, \ldots, a_{|\pi(\omega_i)|}] + \frac{1}{\lfloor \tau n \rfloor}$
    1.5.4 ${b^p}_{\omega_i}[a_1, a_2, \ldots, a_{|\pi(\omega_i)|}] \leftarrow {b^p}_{\omega_i}[a_1, a_2, \ldots, a_{|\pi(\omega_i)|}] + \frac{1}{\lfloor \tau n \rfloor}$

1.6 $\forall_{(k_0,k_1,\ldots,k_{|\pi(\omega_i)|})\in\overline{D^X}^{|\pi(\omega_i)|+1}}\langle$
$\hat{P}(X_{\omega_i}=k_0|X_{\pi(\omega_i)_0}=k_1, X_{\pi(\omega_i)_1}=k_2,\ldots,$
$X_{\pi(\omega_i)_{|\pi(\omega_i)|-1}}=k_{|\pi(\omega_i)|}) \leftarrow \varphi(\omega_i,k_0,k_1,\ldots,k_{|\pi(\omega_i)|})\rangle$

where $\varphi(\omega_i,k_0,k_1,\ldots,k_{|\pi(\omega_i)|}) = \begin{cases} \frac{b_{\omega_i}[k_0,k_1,\ldots,k_{|\pi(\omega_i)|}]}{b^p_{\omega_i}[k_1,k_2,\ldots,k_{|\pi(\omega_i)|}]} & \text{if } |\pi(\omega_i)|>0 \\ b_{\omega_i}[k_0,k_1,\ldots,k_{|\pi(\omega_i)|}] & \text{otherwise} \end{cases}$

2 return$(\hat{P}(\cdot))$

Note that in step 1.6 of algorithm DiEst the probabilities are assigned to the approximation of the probability mass function. This step does not actually have to be computed, because the required information is stored in the arrays b_i and b^p_i. The only reason it is stated here is for the clarity of the mapping to be established for sampling from the probability distribution. Note that the above algorithm takes an exponential amount of time, which can become a problem if the amount of parents gets larger. This sampling is straightforward as the probability information can be directly accessed through the arrays. This is coded in algorithm DiHiSam:

DiSam()
1 ***for*** $i \leftarrow l-1$ ***downto*** 0 ***do***
1.1 $\varsigma \leftarrow$ Random01()
1.2 $\vartheta \leftarrow 0$
1.3 $\eta \leftarrow 1$
1.4 ***if*** $|\pi(\omega_i)|>0$ ***then***
1.4.1 $\eta \leftarrow b^p_{\omega_i}[X_{\pi(\omega_i)_0}, X_{\pi(\omega_i)_1},\ldots,X_{\pi(\omega_i)_{|\pi(\omega_i)|-1}}]$
1.5 ***for*** $k \leftarrow 0$ ***to*** n_d-1 ***do***
1.5.1 $\vartheta \leftarrow \vartheta + \frac{1}{\eta} b_{\omega_i}[k, X_{\pi(\omega_i)_0}, X_{\pi(\omega_i)_1},\ldots,X_{\pi(\omega_i)_{|\pi(\omega_i)|-1}}]$
1.5.2 ***if*** $\varsigma \le \vartheta$ ***then*** $X_{\omega_i} \leftarrow k$; ***breakfor***

Having specified how to estimate the distribution and how to sample from these estimations in the case of discrete random variables, we still have to specify how to find $(\pi,\boldsymbol{\omega})$. Two special cases arise when we fix the distribution in a certain way. In general, a search procedure that returns some fixed PDS that may be dependent on user–input on beforehand, gives an approach similar to the FDA [22]. The two special cases of such a fixed distribution that we wish to outline on beforehand, are the univariate distribution and the complete joint distribution. In the univariate distribution, each variable is regarded independently of the others. This leads to an approach in the line of to PBIL [2] and UMDA [23]. For problems without linked parameters, these methods can be very efficient. However, when parameters are linked, we require higher order building block processing in order to find efficient and effective optimization algorithms [7]. In the joint distribution, which is the other extreme, all the variables are regarded in a joint fashion, conveying $\frac{1}{2}l(l-1)$ dependencies. This PDS will convey too many dependencies for most problems. We can formalize the two algorithms as follows:

UnivariateDistribution()
1 ***for*** $i \leftarrow 0$ ***to*** $l-1$ ***do***
1.1 $\omega_i \leftarrow i$
1.2 $\pi(\omega_i) \leftarrow \emptyset$
2 return$((\pi,\boldsymbol{\omega}))$

JointDistribution()
1 ***for*** $i \leftarrow 0$ ***to*** $l-1$ ***do***
1.1 $\omega_i \leftarrow i$
1.2 $\pi(\omega_i) \leftarrow (i+1, i+2,\ldots,l-1)$
2 return$((\pi,\boldsymbol{\omega}))$

Note that these two fixed PDS algorithms only need to compute the information once and may return this information every time anew. The running time of these algorithms thus needs to be

counted only once, contrary to the algorithms that search for a PDS. The search algorithms will require their running time every iteration.

For each of the three different search approaches discussed in section 3, we now present them in the setting of discrete random variables. The first of these was MIMIC. In order to write down this algorithm, we noted in section 3.2 that we need to be able to compute the entropy. In the case of discrete random variables, combining equations 10 and 18 provides us with enough information to write out algorithm DiChainSea:

DiChainSea()
1 $a \leftarrow$ ***new array*** of ***integer*** with size l
2 ***for*** $i \leftarrow 0$ ***to*** $l-1$ ***do***
2.1 $a[i] \leftarrow i$
2.2 $p_i \leftarrow$ ***new array*** of ***real*** with size n_d
3 $e \leftarrow 0$
4 $\omega_{l-1} \leftarrow a[e]$
5 ***for*** $i \leftarrow 0$ ***to*** $l-1$ ***do***
5.1 ***for*** $j \leftarrow 0$ ***to*** $n_d - 1$ ***do***
5.1.1 $p_{a[i]}[j] \leftarrow 0$
5.2 ***for*** $j \leftarrow 0$ ***to*** $\lfloor \tau n \rfloor - 1$ ***do***
5.2.1 $p_{a[i]}[X_{a[i]}^{(S)j}] \leftarrow p_{a[i]}[X_{a[i]}^{(S)j}] + \frac{1}{\lfloor \tau n \rfloor}$
5.3 $h_{a[i]} \leftarrow -\sum_{k=0}^{n_d-1} p_{a[i]}[k] \ln(p_{a[i]}[k])$
5.4 ***if*** $h_{a[i]} < h_{\omega_{l-1}}$ ***then***
5.4.1 $\omega_{l-1} \leftarrow a[i]$
5.4.2 $e \leftarrow i$
6 $\pi(\omega_{l-1}) \leftarrow \emptyset$
7 $a[e] \leftarrow a[l-1]$
8 ***for*** $i \leftarrow l-2$ ***downto*** 0 ***do***
8.1 $e \leftarrow 0$
8.2 $\omega_i \leftarrow a[e]$
8.3 ***for*** $q \leftarrow 0$ ***to*** i ***do***
8.3.1 $p_{\omega_{i+1}a[q]} \leftarrow$ ***new array*** of ***real*** in 2 dimensions with size $n_d \times n_d$
8.3.2 $p_{a[q]\omega_{i+1}} \leftarrow$ ***new array*** of ***real*** in 2 dimensions with size $n_d \times n_d$
8.3.3 ***for*** $j_0 \leftarrow 0$ ***to*** $n_d - 1$ ***do***
8.3.3.1 ***for*** $j_1 \leftarrow 0$ ***to*** $n_d - 1$ ***do***
8.3.3.1.1 $p_{\omega_{i+1}a[q]}[j_0, j_1] \leftarrow 0$
8.3.3.1.2 $p_{a[q]\omega_{i+1}}[j_1, j_0] \leftarrow 0$
8.3.4 ***for*** $j \leftarrow 0$ ***to*** $\lfloor \tau n \rfloor - 1$ ***do***
8.3.4.1 $p_{\omega_{i+1}a[q]}[X_{\omega_{i+1}}^{(S)j}, X_{a[q]}^{(S)j}] \leftarrow p_{\omega_{i+1}a[q]}[X_{\omega_{i+1}}^{(S)j}, X_{a[q]}^{(S)j}] + \frac{1}{\lfloor \tau n \rfloor}$
8.3.4.2 $p_{a[q]\omega_{i+1}}[X_{a[q]}^{(S)j}, X_{\omega_{i+1}}^{(S)j}] \leftarrow p_{a[q]\omega_{i+1}}[X_{a[q]}^{(S)j}, X_{\omega_{i+1}}^{(S)j}] + \frac{1}{\lfloor \tau n \rfloor}$
8.3.5 $h_{\omega[i+1]a[q]} \leftarrow -\sum_{k_0=0}^{n_d-1} \sum_{k_1=0}^{n_d-1} p_{\omega_{i+1}a[q]}[k_0, k_1] \ln(p_{\omega_{i+1}a[q]}[k_0, k_1])$
8.3.6 $h_{a[q]\omega_{i+1}} \leftarrow h_{\omega_{i+1}a[q]}$
8.3.7 ***if*** $h_{a[q]\omega_{i+1}} - h_{\omega_{i+1}} < h_{\omega_i\omega_{i+1}} - h_{\omega_{i+1}}$ ***then***
8.3.7.1 $\omega_i \leftarrow a[q]$
8.3.7.2 $e \leftarrow q$
8.4 $\pi(\omega_i) \leftarrow \omega_{i+1}$
8.5 $a[e] \leftarrow a[i]$
9 ***for*** $i \leftarrow 0$ ***to*** $l-1$ ***do***
9.1 ***if*** $|\pi(\omega_i)| > 0$ ***then***
9.1.1 $b_{\omega_i} \leftarrow p_{\omega_i \pi(\omega_i)_0}$
9.1.2 $b_{\omega_i}^p \leftarrow p_{\pi(\omega_i)_0}$
9.2 ***else***
9.2.1 $b_{\omega_i} \leftarrow p_{\omega_i}$
10 return$((\pi, \boldsymbol{\omega}))$

Note that from algorithm DiChainSea it follows that we do not need to recompute the arrays b_i and b_i^p in algorithm DiEst. In other words, because of the fact that line 1.6 of algorithm DiEst was only to clarify the construction of $\hat{P}$, algorithm DiEst can be skipped when using algorithm DiChainSea. This follows from the definition of IDEA because $sea()$ is executed before $est()$. This reduces computation time every iteration.

The second search procedure is taken from the optimal dependency trees approach. To this end, we require again the entropy measure. Since we have already shown what this measure is, we can immediately write down algorithm DiTreeSea:

```
DiTreeSea()
 1  a ← new array of integer with size l
 2  for i ← 0 to l − 1 do
    2.1   a[i] ← i
    2.2   p_i ← new array of real with size n_d
 3  b^t ← new array of integer with size l − 1
 4  e ←RandomNumber(l)
 5  ω_{l−1} ← e
 6  π(ω_{l−1}) ← ∅
 7  for j ← 0 to n_d − 1 do
    7.1   p_{ω_{l−1}}[j] ← 0
 8  for j ← 0 to ⌊τn⌋ − 1 do
    8.1   p_{ω_{l−1}}[X^{(S)j}_{ω_{l−1}}] ← p_{ω_{l−1}}[X^{(S)j}_{ω_{l−1}}] + 1/⌊τn⌋
 9  h_{ω_{l−1}} ← − Σ_{k=0}^{n_d−1} p_{ω_{l−1}}[k] ln(p_{ω_{l−1}}[k])
10  a[e] ← a[l − 1]
11  for i ← 0 to l − 2 do
      11.1   for j ← 0 to n_d − 1 do
               11.1.1   p_{a[i]}[j] ← 0
      11.2   for j ← 0 to ⌊τn⌋ − 1 do
               11.2.1   p_{a[i]}[X^{(S)j}_{a[i]}] ← p_{a[i]}[X^{(S)j}_{a[i]}] + 1/⌊τn⌋
      11.3   h_{a[i]} ← − Σ_{k=0}^{n_d−1} p_{a[i]}[k] ln(p_{a[i]}[k])
      11.4   b^t[a[i]] ← ω_{l−1}
      11.5   p_{b^t[a[i]]a[i]} ← new array of real in 2 dimensions with size n_d × n_d
      11.6   p_{a[i]b^t[a[i]]} ← new array of real in 2 dimensions with size n_d × n_d
      11.7   for j_0 ← 0 to n_d − 1 do
               11.7.1   for j_1 ← 0 to n_d − 1 do
                          11.7.1.1   p_{b^t[a[i]]a[i]}[j_0, j_1] ← 0
                          11.7.1.2   p_{a[i]b^t[a[i]]}[j_1, j_0] ← 0
      11.8   for j ← 0 to ⌊τn⌋ − 1 do
               11.8.1   p_{b^t[a[i]]a[i]}[X^{(S)j}_{b^t[a[i]]}, X^{(S)j}_{a[i]}] ← p_{b^t[a[i]]a[i]}[X^{(S)j}_{b^t[a[i]]}, X^{(S)j}_{a[i]}] + 1/⌊τn⌋
               11.8.2   p_{a[i]b^t[a[i]]}[X^{(S)j}_{a[i]}, X^{(S)j}_{b^t[a[i]]}] ← p_{a[i]b^t[a[i]]}[X^{(S)j}_{a[i]}, X^{(S)j}_{b^t[a[i]]}] + 1/⌊τn⌋
      11.9   h_{b^t[a[i]]a[i]} ← − Σ_{k_0=0}^{n_d−1} Σ_{k_1=0}^{n_d−1} p_{b^t[a[i]]a[i]}[k_0, k_1] ln(p_{b^t[a[i]]a[i]}[k_0, k_1])
      11.10  h_{a[i]b^t[a[i]]} ← h_{b^t[a[i]]a[i]}
```

```
12   for i ← l − 2 downto 0 do
     12.1   e ← arg max_j {h_{a[j]} + h_{b^t[a[j]]} − h_{a[j]b^t[a[j]]} | j ∈ {0,1,...,i}}
     12.2   ω_i ← a[e]
     12.3   π(ω_i) ← b^t[ω_i]
     12.4   a[e] ← a[i]
     12.5   for j ← 0 to i − 1 do
            12.5.1  p_{ω_i a[j]} ← new array of real in 2 dimensions with size n_d × n_d
            12.5.2  p_{a[j]ω_i} ← new array of real in 2 dimensions with size n_d × n_d
            12.5.3  for j_0 ← 0 to n_d − 1 do
                    12.5.3.1   for j_1 ← 0 to n_d − 1 do
                               12.5.3.1.1   p_{ω_i a[j]}[j_0, j_1] ← 0
                               12.5.3.1.2   p_{a[j]ω_i}[j_1, j_0] ← 0
            12.5.4  for k ← 0 to ⌊τn⌋ − 1 do
                    12.5.4.1   p_{ω_i a[j]}[X^{(S)k}_{ω_i}, X^{(S)k}_{a[j]}] ← p_{ω_i a[j]}[X^{(S)k}_{ω_i}, X^{(S)k}_{a[j]}] + 1/⌊τn⌋
                    12.5.4.2   p_{a[j]ω_i}[X^{(S)k}_{a[j]}, X^{(S)k}_{ω_i}] ← p_{a[j]ω_i}[X^{(S)k}_{a[j]}, X^{(S)k}_{ω_i}] + 1/⌊τn⌋
            12.5.5  h_{ω_i a[j]} ← − Σ_{k_0=0}^{n_d−1} Σ_{k_1=0}^{n_d−1} p_{ω_i a[j]}[k_0, k_1] ln(p_{ω_i a[j]}[k_0, k_1])
            12.5.6  h_{a[j]ω_i} ← h_{ω_i a[j]}
            12.5.7  I_best ← h_{a[j]} + h_{b^t[a[j]]} − h_{a[j]b^t[a[j]]}
            12.5.8  I_add ← h_{a[j]} + h_{ω_i} − h_{a[j]ω_i}
            12.5.9  if I_best < I_add then
                    12.5.9.1   b^t[a[j]] ← ω_i
13   for i ← 0 to l − 1 do
     13.1   if |π(ω_i)| > 0 then
            13.1.1   b_{ω_i} ← p_{ω_i π(ω_i)_0}
            13.1.2   b^p_{ω_i} ← p_{π(ω_i)_0}
     13.2   else
            13.2.1   b_{ω_i} ← p_{ω_i}
14   return((π, ω))
```

Note that from algorithm DiTreeSea we have again that we do not need to recompute arrays b_i and b_i^p in algorithm DiEst.

The third and final search procedure is the BOA approach. As noted in section 3.4, the general algorithm consists out of more than one case, depending on the value of κ. This means that first of all, we may present the general algorithm DiGraphSea for the case of discrete random variables:

```
DiGraphSea(κ)
1   if κ = 0 then
    1.1   (π, ω) ← UnivariateDistribution()
2   else if κ = 1 then
    2.1   (π, ω) ← DiGraphSeaExact()
3   else then
    3.1   (π, ω) ← DiGraphSeaGreedy(κ)
4   return((π, ω))
```

When $\kappa = 0$, the result is simply the univariate distribution, for which we had already defined algorithm UnivariateDistribution. When we have $\kappa = 1$, we noted in section 3.4 that we can use Edmond's algorithm. Edmond's algorithm requires for each arc (i, j) in the graph to have assigned to it a positive real value $c((i, j))$ as defined in appendix B. The optimum branching is then found with respect to these values. This optimum branching is in our sense a PDS graph where each node has at most 1 parent, meaning that the $\pi(\cdot)$ vector for all nodes has length at most 1 $(\forall_{i \in \{0,1,\ldots,l-1\}} \langle |\pi(i)| \leq 1 \rangle)$. Note that Emond's algorithm computes a *maximum* branching, whereas we proposed in section 3.4 to *minimize* the entropy measure. In order to be able to maximize and to have all positive values, we maximize the negative entropy, which is equivalent to minimizing the entropy measure. As the PDS that results holds no information on the measure

or metric used, this is completely irrelevant to the result. For Edmond's algorithm, we require to have only positive real values. To this end, we go over the metric for each of the arcs and find the minimum value. By subtracting this minimum value from the value for each arc, the value $c((i,j))$ becomes a real value larger *or equal* to 0. To get only positive values as required, we add 1 to every value in addition. We can now specify algorithm DiGraphSeaExact that employs Edmond's optimum branching algorithm to find the PDS in the case of discrete random variables:

```
DiGraphSeaExact()
 1  define type edarc as (stack of integer, stack of integer, stack of real)
 2  v^p ← new array of vector of integer with size l
 3  V ← new vector of integer
 4  A ← new vector of edarc
 5  for i ← 0 to l − 1 do
    5.1  p_i ← new array of real with size n_d
    5.2  for j ← 0 to n_d − 1 do
         5.2.1  p_i[j] ← 0
    5.3  for j ← 0 to ⌊τn⌋ − 1 do
         5.3.1  p_i[X_i^(S)j] ← p_i[X_i^(S)j] + 1/⌊τn⌋
    5.4  h_i ← −Σ_{k=0}^{n_d−1} p_i[k] ln(p_i[k])
 6  for i ← 0 to l − 1 do
    6.1  for j ← i + 1 to l − 1 do
         6.1.1  p_ij ← new array of real in 2 dimensions with size n_d × n_d
         6.1.2  p_ji ← new array of real in 2 dimensions with size n_d × n_d
         6.1.3  for j_0 ← 0 to n_d − 1 do
                6.1.3.1  for j_1 ← 0 to n_d − 1 do
                         6.1.3.1.1  p_ij[j_0, j_1] ← 0
                         6.1.3.1.2  p_ji[j_1, j_0] ← 0
         6.1.4  for k ← 0 to ⌊τn⌋ − 1 do
                6.1.4.1  p_ij[X_i^(S)k, X_j^(S)k] ← p_ij[X_i^(S)k, X_j^(S)k] + 1/⌊τn⌋
                6.1.4.2  p_ji[X_j^(S)k, X_i^(S)k] ← p_ji[X_j^(S)k, X_i^(S)k] + 1/⌊τn⌋
    6.2  for j ← 0 to l − 1 do
         6.2.1  if i ≠ j then
                6.2.1.1  h_ji ← −Σ_{k_0=0}^{n_d−1} Σ_{k_1=0}^{n_d−1} p_ji[k_0, k_1] ln(p_ji[k_0, k_1])
                6.2.1.2  c ← h_ji − h_i
                6.2.1.3  S^s ← new stack of integer
                6.2.1.4  S^t ← new stack of integer
                6.2.1.5  S^c ← new stack of real
                6.2.1.6  PUSH(S^s, i)
                6.2.1.7  PUSH(S^t, j)
                6.2.1.8  PUSH(S^c, −c)
                6.2.1.9  A_|A| ← (S^s, S^t, S^c)
 7  γ ← min_{(S^s,S^t,S^c)∈A}{TOP(S^c)}
 8  for i ← 0 to |A| − 1 do
    8.1  (S^s, S^t, S^c) ← A_i
    8.2  c ← POP(S^c)
    8.3  c ← c + 1 − γ
    8.4  PUSH(S^c, c)
 9  B ← GraphSeaOptimumBranching(V,A,l)
10  for i ← 0 to |B| − 1 do
    10.1  (S^s, S^t, S^c) ← B_i
    10.2  s ← POP(S^s)
    10.3  t ← POP(S^t)
    10.4  v^p[t]_|v^p[t]| ← s
```

```
11  (π, ω) ← GraphSeaTopologicalSort(v^p)
12  for i ← 0 to l − 1 do
    12.1  if |π(ω_i)| > 0 then
              12.1.1  b_{ω_i} ← p_{ω_i π(ω_i)_0}
              12.1.2  b^p_{ω_i} ← p_{π(ω_i)_0}
    12.2  else
              12.2.1  b_{ω_i} ← p_{ω_i}
13  return((π, ω))
```

Again, we find from algorithm DiGraphSeaExact that we do not need to recompute arrays b_i and b_i^p in algorithm DiEst. Algorithm DiGraphSeaExact calls other algorithms to compute the optimum branching. These algorithms are *independent* of the type of density model that is used and concern only the finding of the optimum branching, given $c(i,j)$. Therefore, we need to state them only once. The additional algorithms can be found in appendix E. The running time for algorithm DiGraphSeaExact is $\mathcal{O}(l^3 + l^2 n_d^2 + l^2 \tau n)$, which is subsumed by the running time of the greedy graph search algorithm for the case when $\kappa > 1$.

We conclude the summary of the search algorithms for discrete random variables by presenting the greedy graph search algorithm for the case when $\kappa > 1$. The outline for the algorithm was given in section 3.4. Combining that outline with the discrete entropy we have already used in the above search algorithms, we can now state algorithm DiGraphSeaGreedy:

```
DiGraphSeaGreedy(κ)
1   a ← new array of boolean in 2 dimensions with size l × l
2   v^p, v^s ← 2 new arrays of vector of integer with size l
3   h^n, c ← 2 new arrays of real in 2 dimensions with size l × l
4   h^o ← new array of real with size l
5   for i ← 0 to l − 1 do
    5.1  p^p_{0i} ← new array of real with size n_d
    5.2  for j ← 0 to n_d − 1 do
             5.2.1  p^p_{0i}[j] ← 0
    5.3  for j ← 0 to ⌊τn⌋ − 1 do
             5.3.1  p^p_{0i}[X^{(S)j}_{ω_{l−1}}] ← p^p_{0i}[X^{(S)j}_{ω_{l−1}}] + 1/⌊τn⌋
    5.4  for q ← 1 to l − 1 do
             5.4.1  p^p_{qi} ← new array of real with size n_d
             5.4.2  for j ← 0 to n_d − 1 do
                        5.4.2.1  p^p_{qi}[j] ← p^p_{0i}[j]
    5.5  h^o[i] ← −Σ_{k=0}^{n_d−1} p^p_{0i}[k] ln(p^p_{0i}[k])
    5.6  b_i ← p^p_{0i}
6   for i ← 0 to l − 1 do
    6.1  for j ← i + 1 to l − 1 do
             6.1.1  p_{ij} ← new array of real in 2 dimensions with size n_d × n_d
             6.1.2  p_{ji} ← new array of real in 2 dimensions with size n_d × n_d
             6.1.3  for j_0 ← 0 to n_d − 1 do
                        6.1.3.1  for j_1 ← 0 to n_d − 1 do
                                     6.1.3.1.1  p_{ij}[j_0, j_1] ← 0
                                     6.1.3.1.2  p_{ji}[j_1, j_0] ← 0
             6.1.4  for k ← 0 to ⌊τn⌋ − 1 do
                        6.1.4.1  p_{ij}[X_i^{(S)k}, X_j^{(S)k}] ← p_{ij}[X_i^{(S)k}, X_j^{(S)k}] + 1/⌊τn⌋
                        6.1.4.2  p_{ji}[X_j^{(S)k}, X_i^{(S)k}] ← p_{ji}[X_j^{(S)k}, X_i^{(S)k}] + 1/⌊τn⌋
```

6.2 ***for*** $j \leftarrow 0$ ***to*** $l-1$ ***do***
6.2.1 ***if*** $i \neq j$ ***then***
6.2.1.1 $a[i,j] \leftarrow$ ***true***
6.2.1.2 $h^n[i,j] \leftarrow -\sum_{k_0=0}^{n_d-1}\sum_{k_1=0}^{n_d-1} p_{ji}[k_0,k_1]\ln(p_{ji}[k_0,k_1])$
6.2.1.3 $h^n[i,j] \leftarrow h^n[i,j] - h^o[i]$
6.2.1.4 $c[i,j] \leftarrow h^n[i,j] - h^o[j]$
6.3 $a[i,i] \leftarrow$ ***false***
7 $\gamma \leftarrow l^2 - l$
8 ***while*** $\gamma > 0$ ***do***
8.1 $(v_0, v_1) \leftarrow arg\ \min_{(i,j)}\{c[i,j] \mid a[i,j] \wedge (i,j) \in \{0,1,\ldots,l-1\}^2\}$
8.2 ***if*** $c[v_0,v_1] > 0$ ***then***
8.2.1 ***breakwhile***
8.3 $\gamma \leftarrow \gamma -$ GraphSeaArcAdd(κ, v_0, v_1, a, v^p, v^s)
8.4 $h^o[v_1] \leftarrow h^n[v_0,v_1]$
8.5 $b^p_{v_1} \leftarrow p^p_{v_1 v_0}$
8.6 $b_{v_1} \leftarrow p_{v_1 v_0}$
8.7 ***for*** $i \leftarrow 0$ ***to*** $l-1$ ***do***
8.7.1 ***if*** $a[i,v_1]$ ***then***
8.7.1.1 $p_{v_1 i} \leftarrow$ ***new array*** of ***real*** in $|v^p[v_1]|+2$ dimensions with size $n_d \times n_d \times \ldots \times n_d$
8.7.1.2 $p^p_{v_1 i} \leftarrow$ ***new array*** of ***real*** in $|v^p[v_1]|+1$ dimensions with size $n_d \times n_d \times \ldots \times n_d$
8.7.1.3 ***for*** $(j_0, j_1, \ldots, j_{|v^p[v_1]|+1}) \leftarrow (0,0,\ldots,0)$ ***to*** $(n_d-1, n_d-1, \ldots, n_d-1)$ ***in crossproduct do***
8.7.1.3.1 $p_{v_1 i}[j_0, j_1, \ldots, j_{|v^p[v_1]|+1}] \leftarrow 0$
8.7.1.4 ***for*** $(j_0, j_1, \ldots, j_{|v^p[v_1]|}) \leftarrow (0,0,\ldots,0)$ ***to*** $(n_d-1, n_d-1, \ldots, n_d-1)$ ***in crossproduct do***
8.7.1.4.1 $p^p_{v_1 i}[j_0, j_1, \ldots, j_{|v^p[v_1]|}] \leftarrow 0$
8.7.1.5 ***for*** $j \leftarrow 0$ ***to*** $\lfloor \tau n \rfloor - 1$ ***do***
8.7.1.5.1 $k_0 \leftarrow X^{(S)j}_{v_1}$
8.7.1.5.2 $k_q \leftarrow X^{(S)j}_{v^p[v_1]_{q-1}} \quad (\forall_{q \in \{1,2,\ldots,|v^p[v_1]|\}})$
8.7.1.5.3 $k_{|v^p[v_1]|+1} \leftarrow X^{(S)j}_i$
8.7.1.5.4 $p_{v_1 i}[k_0, k_1, \ldots, k_{|v^p[v_1]|+1}] \leftarrow p_{v_1 i}[k_0, k_1, \ldots, k_{|v^p[v_1]|+1}] + \frac{1}{\lfloor \tau n \rfloor}$
8.7.1.5.5 $p^p_{v_1 i}[k_1, k_2, \ldots, k_{|v^p[v_1]|+1}] \leftarrow p_{v_1 i}[k_1, k_2, \ldots, k_{|v^p[v_1]|+1}] + \frac{1}{\lfloor \tau n \rfloor}$
8.7.1.6 $h^n[i,j] \leftarrow -\sum_{k_0=0}^{n_d-1}\sum_{k_1=0}^{n_d-1}\cdots\sum_{k_{|v^p[v_1]|+1}}^{n_d-1} p_{v_1 i}[k_0, k_1, \ldots, k_{|v^p[v_1]|+1}]\ln(p_{v_1 i}[k_0, k_1, \ldots, k_{|v^p[v_1]|+1}])$
8.7.1.7 $h^n[i,j] \leftarrow h^n[i,j] - (-\sum_{k_0=0}^{n_d-1}\sum_{k_1=0}^{n_d-1}\cdots\sum_{k_{|v^p[v_1]|}}^{n_d-1} p^p_{v_1 i}[k_0, k_1, \ldots, k_{|v^p[v_1]|}]\ln(p^p_{v_1 i}[k_0, k_1, \ldots, k_{|v^p[v_1]|}]))$
8.7.1.8 $c[i,j] \leftarrow h^n[i,v_1] - h^o[v_1]$
9 return(GraphSeaTopologicalSort(v^p))

Also in the general graph search algorithm DiGraphSeaGreedy, which runs in $\mathcal{O}(l^3\kappa + l^2 n_d + l^2\tau n + l\kappa n_d^{\kappa+1} + l\kappa^2\tau n)$ time, we find that we do not need to recompute arrays b_i and b^p_i in algorithm DiEst. The reason for this in all cases is that the information that is used to guide the search is largely the same information that is used to build the estimates. In the algorithm, we have called algorithms GraphSeaArcAdd and GraphSeaTopologicalSort. Just as was the case for the exact algorithm for the case when $\kappa = 1$, these subalgorithms are *independent* of the type of density model that is used. More detailed information on the additional algorithms can be found in appendix E.

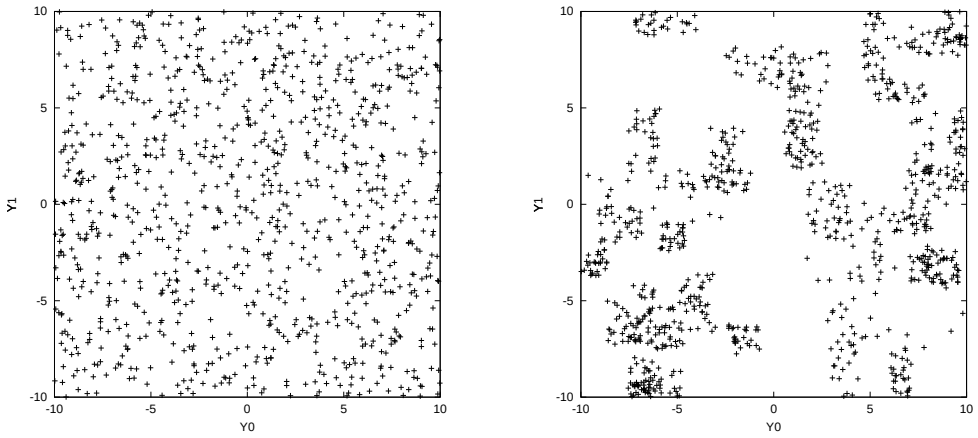

Figure 4: A uniform and a clustered data set over Y_0 and Y_1 with $D^Y = [-10, 10]$.

To complete this section on the algorithms for discrete random variables, we refer to the articles that presented the original algorithms for obtained results. Given the general framework of IDEA and the implementation of the original algorithms, it will probably immediately become clear as to how a similar algorithm within the IDEA framework would perform.

5 IDEAs for continuous random variables

Having noted the type of information that is required in previously created algorithms in section 3 and having presented the algorithms for discrete random variables in section 4, we are ready to present IDEA strategies in which density estimation for continuous random variables is performed. Density estimation can be seen as model fitting where given the structure from equation 7, we have to estimate the individual probabilities in the product. In general, we find from probability theory that we can distinguish three important classes of density estimation models: *parametric*, *non–parametric* and *mixture*. None of these have ever been explicitly mentioned in previous work on algorithms for discrete random variables. A reason for this might be that the frequency count approach is the most straightforward and most precise one in the case of discrete random variables. The distinction between the three classes is classicly found in continuous models.

In this section, we concentrate on the continuous model. To provide additional insight in the distribution estimations, we give examples of the resulting density function for two data sets. In both cases we have only two dimensions with $D^Y = [-10, 10]$. The data sets are plotted in figure 4, which shows a uniform distribution on the left and a clustered distribution on the right. We shall make figures for the joint distribution that was estimated by using a certain density model. In addition, we shall also make figures for the univariate distribution over both variables Y_0 and Y_1.

In this paper, we present two approaches with different density models. One approach is from the class of parametric distributions and uses the *normal distribution*. This approach is presented in section 5.1. The other approach is from the class of non–parametric distributions and uses the *histogram distribution*, which is closely related to the empiric probabilities for discrete random variables from section 4. The histogram approach is presented in section 5.2. In section 6, some preliminary work and ideas are presented for another model in the class of non–parametric distributions, namely the *normal kernel distribution*. Also, the relation of that kernel distribution to the *normal mixture model distribution*, which belongs to the class of mixture models, is presented. Finally, we note again as we did for the case of using discrete random variables, that the running times of the algorithms are stated in section 6.

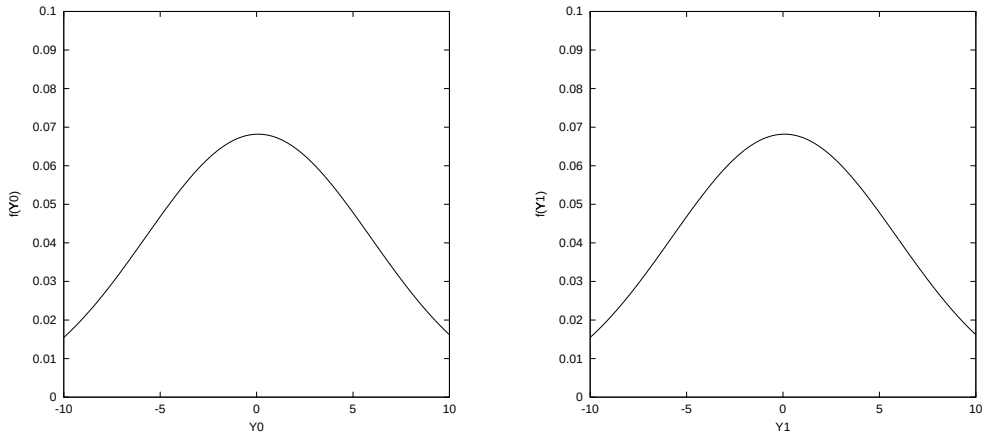

Figure 5: Fitting the univariate data from the uniform data set with a normal distribution parametric model.

5.1 Parametric model: normal distribution

The first type of distribution estimation model we discuss, is the *parametric* model. In this model, a density function as stated in equation 2 is given on beforehand. This means that the general form of the density function is fixed. The goal is to set the parameters for the density function in such a way that the result represents the distribution of the data points as good as possible.

The most widely used parametric density function for real spaces is the density function that underlies the normal distribution, which is a normalized Gaussian. It follows from section 2.2 that we require to have the *conditional* density function conditioned on *one* variable. In appendix C it is shown that this density function for one variable Y_0 conditioned on n other variables $Y_1, Y_2, \ldots, Y_n$ equals:

$$f(y_0|y_1, y_2, \ldots, y_n) = g(y, \tilde{\mu}_0, \tilde{\sigma}_0) \tag{19}$$

$$\text{where} \begin{cases} g(y, \mu, \sigma) &= \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(y-\mu)^2}{2\sigma^2}} \\ \tilde{\sigma}_0 &= \frac{1}{\sqrt{\sigma'_{00}}} \\ \tilde{\mu}_0 &= \frac{\mu_0 \sigma'_{00} - \sum_{i=1}^{c} (y_i - \mu_i)\sigma'_{i0}}{\sigma'_{00}} \end{cases}$$

In equation 19, we use the notation $\sigma'_{ij} = (\mathbf{\Sigma}^{-1})(i, j)$ with $\mathbf{\Sigma}$ the nonsingular covariance matrix over variables $Y_0, Y_1, \ldots, Y_n$. The density function over the variables taken univariately for the two datasets is plotted in figures 5 and 6. Note that for the uniform data set, the centers are placed just about over 0 as expected, whereas this is not completely the case for the clustered data set. Still, it is clear that there is no great distinction between the two estimated distributions, which shows the obvious deficit of this parametric model in modelling clusters. This can be observed again from the joint distributions as depicted in figure 7. The distributions are quite identical. The difference lies in that the joint distribution over the clustered data set is somewhat skewed and lies more around the diagonal from $(-10, -10)$ to $(10, 10)$ instead of being complete uniform in all directions. By observing again the clustered data set in figure 4, this can be seen to indeed slightly be the trend for this data set. So even though this model is able to represent the *general* features of the data, it fails in representing the *specific* features.

In order to sample from equation 19, we need to compute $\tilde{\mu}_0$ and $\tilde{\sigma}_0$. It becomes clear that to this end, we require to compute the inverse of the covariance matrix, as we need values from $\mathbf{\Sigma}^{-1}$. Computing the inverse requires $\mathcal{O}(n^3)$ computing time, where $n \times n$ is the size of the matrix

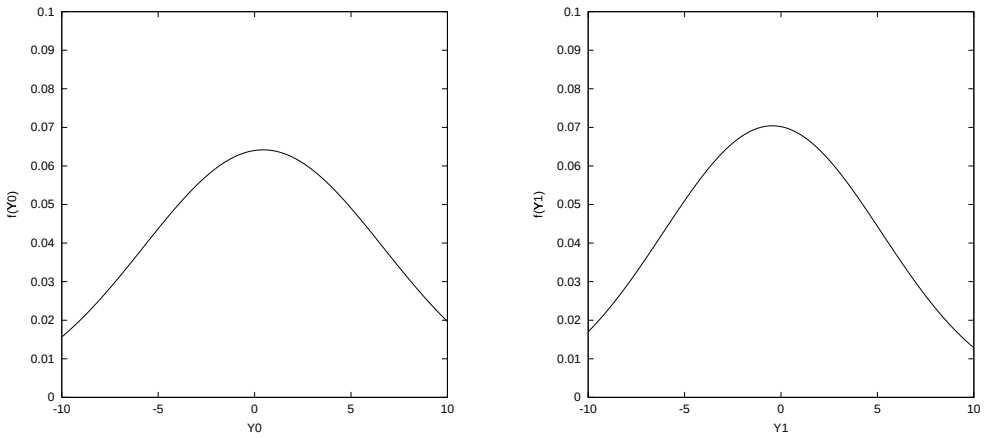

Figure 6: Fitting the univariate data from the clustered data set with a normal distribution parametric model.

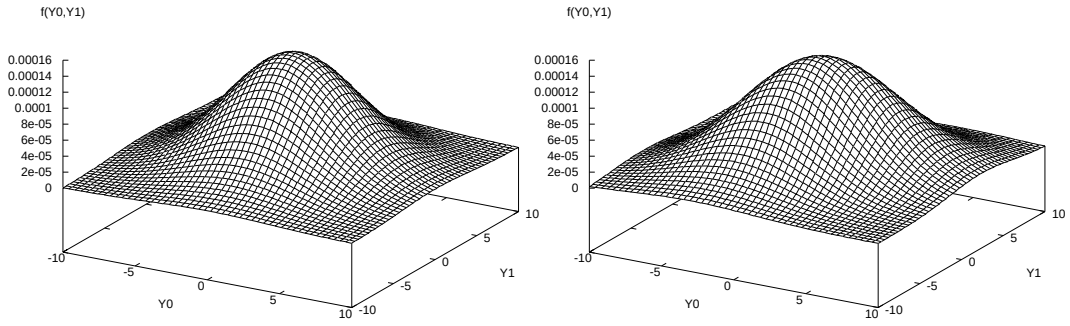

Figure 7: Fitting the joint data from the uniform (left) and clustered (right) data sets with a normal distribution parametric model.

to be inverted. However, we observe from equation 19 that we only require entries from the first column of the inverse matrix, meaning that we have to solve the linear system of equations:

$$\sum_{i=0}^{n-1} \sigma_{0i}\sigma'_{i0} = 1 \ \wedge \ \forall_{j\in\{1,2,\ldots,n-1\}} \left\langle \sum_{i=0}^{n-1} \sigma_{ji}\sigma'_{i0} = 0 \right\rangle \tag{20}$$

We shall use matrix notation for the remainder of this paper to write systems such as the one from equation 20. The use of matrix notation gives better insight into the linear system to be solved and eases both notation and implementation. In the case of equation 20, we may write:

$$\begin{bmatrix} \sigma_{00} & \sigma_{01} & \cdots & \sigma_{0(n-1)} \\ \sigma_{10} & \sigma_{11} & \cdots & \sigma_{1(n-1)} \\ \sigma_{20} & \sigma_{21} & \cdots & \sigma_{2(n-1)} \\ \vdots & \vdots & \vdots & \vdots \\ \sigma_{(n-1)0} & \sigma_{(n-1)1} & \cdots & \sigma_{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} \sigma'_{00} \\ \sigma'_{10} \\ \sigma'_{20} \\ \vdots \\ \sigma'_{(n-1)0} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \equiv \mathbf{\Sigma}(\mathbf{\Sigma}^{-1}(*,0)) = I(*,0) \tag{21}$$

Solving this system of equations unfortunately also takes $\mathcal{O}(n^3)$ time, but can be done faster than computing the complete inverse, especially when the diameter of the matrix becomes rather large. The above linear system of equations is of the form $\mathbf{\Sigma}(\mathbf{\Sigma}^{-1}(*,0)) = I(*,0)$. The matrix $\mathbf{\Sigma}$ is the non–singular symmetric covariance matrix, where $\sigma_{ij} = \mathbf{\Sigma}(i,j)$ and $\sigma'_{ij} = \mathbf{\Sigma}^{-1}(i,j)$. In the algorithms that follow, we shall first have to construct $\mathbf{\Sigma}$. At this point, we *must* note that this matrix concerns certain variables and that so far, we have not pointed out exactly which these variables are. We therefore introduce the following notation:

$$\begin{cases} \mathbf{\Sigma}\langle j_0, j_1, j_2, \ldots, j_{n-1}\rangle = \begin{bmatrix} \breve{\sigma}_{j_0 j_0} & \breve{\sigma}_{j_0 j_1} & \cdots & \breve{\sigma}_{j_0 j_{n-1}} \\ \breve{\sigma}_{j_1 j_0} & \breve{\sigma}_{j_1 j_1} & \cdots & \breve{\sigma}_{j_1 j_{n-1}} \\ \breve{\sigma}_{j_2 j_0} & \breve{\sigma}_{j_2 j_1} & \cdots & \breve{\sigma}_{j_2 j_{n-1}} \\ \vdots & \vdots & \vdots & \vdots \\ \breve{\sigma}_{j_{n-1} j_0} & \breve{\sigma}_{j_{n-1} j_1} & \cdots & \breve{\sigma}_{j_{n-1} j_{n-1}} \end{bmatrix} \\ \breve{\sigma}_{ij} \ (= \breve{\sigma}_{ji}) = \frac{\sum_{k=0}^{\lfloor \tau n \rfloor - 1} (Y_i^{(S)k} - \mu_i)(Y_j^{(S)k} - \mu_j)}{\lfloor \tau n \rfloor} \end{cases} \tag{22}$$

It should be clear that by using equation 22, we have that $\sigma_{ik} = \mathbf{\Sigma}\langle j_0, j_1, \ldots, j_{n-1}\rangle(i,k) = \breve{\sigma}_{j_i j_k}$. In this way, we have filled the positions of the matrix $\mathbf{\Sigma}$ that we use in equation 21 with the actual covariances $\breve{\sigma}_{ij}$. Just as we did for the entropy in the univariate and bivariate case as stated in section 3, we shall assume in the pseudo–code that μ_i $(\forall_{i\in\{0,1,\ldots,l-1\}})$ and $\breve{\sigma}_{ij}$ $(\forall_{(i,j)\in\{0,1,\ldots,l-1\}^2})$ are global variables, for which we do not allocate additional memory in the algorithms. Using these definitions, we can now specify algorithm RePaNoEst to estimate the density functions once the structure is known:

```
RePaNoEst()
1   m ← new array of boolean in 2 dimensions with size l × l
2   for i ← 0 to l − 1 do
    2.1   for j ← 0 to l − 1 do
          2.1.1   m[i, j] ← false
    2.2   m[i, i] ← true
3   for i ← 0 to l − 1 do
    3.1   μ_i ← (Σ_{k=0}^{⌊τn⌋−1} Y_i^{(S)k}) / ⌊τn⌋
```

4 ***for*** $i \leftarrow 0$ ***to*** $l-1$ ***do***

4.1 $\breve{\sigma}_{ii} \leftarrow \frac{\sum_{k=0}^{\lfloor \tau n \rfloor -1}(Y_i^{(S)k} - \mu_i)^2}{\lfloor \tau n \rfloor}$

4.2 ***for*** $j \leftarrow 0$ ***to*** $|\pi(i)| - 1$ ***do***

4.2.1 ***if*** $\neg m[i, \pi(i)_j]$ ***then***

4.2.1.1 $\breve{\sigma}_{i\pi(i)_j} \leftarrow \frac{\sum_{k=0}^{\lfloor \tau n \rfloor -1}(Y_i^{(S)k} - \mu_i)(Y_{\pi(i)_j}^{(S)k} - \mu_{\pi(i)_j})}{\lfloor \tau n \rfloor}$

4.2.1.2 $\breve{\sigma}_{\pi(i)_j i} \leftarrow \breve{\sigma}_{i\pi(i)_j}$

4.2.1.3 $m[i, \pi(i)_j] \leftarrow$ ***true***

4.2.1.4 $m[\pi(i)_j, i] \leftarrow$ ***true***

4.2.2 ***for*** $q \leftarrow 0$ ***to*** $j-1$ ***do***

4.2.2.1 ***if*** $\neg m[\pi(i)_q, \pi(i)_j]$ ***then***

4.2.2.1.1 $\breve{\sigma}_{\pi(i)_q \pi(i)_j} \leftarrow \frac{\sum_{k=0}^{\lfloor \tau n \rfloor -1}(Y_{\pi(i)_q}^{(S)k} - \mu_{\pi(i)_q})(Y_{\pi(i)_j}^{(S)k} - \mu_{\pi(i)_j})}{\lfloor \tau n \rfloor}$

4.2.2.1.2 $\breve{\sigma}_{\pi(i)_j \pi(i)_q} \leftarrow \breve{\sigma}_{\pi(i)_q \pi(i)_j}$

4.2.2.1.3 $m[\pi(i)_q, \pi(i)_j] \leftarrow$ ***true***

4.2.2.1.4 $m[\pi(i)_j, \pi(i)_q] \leftarrow$ ***true***

5 ***for*** $i \leftarrow 0$ ***to*** $l-1$ ***do***

5.1 ***Find*** σ'_{*0} ***in*** $\mathcal{O}((|\pi(\omega_i)|+1)^3)$ ***from***

$\mathbf{\Sigma}\langle \omega_i, \pi(\omega_i)_0, \pi(\omega_i)_1, \ldots, \pi(\omega_i)_{|\pi(\omega_i)|-1}\rangle(\mathbf{\Sigma}^{-1}(*,0)) = I(*,0)$

5.2 $\tilde{\sigma}_{\omega_i} \leftarrow \frac{1}{\sqrt{\sigma'_{00}}}$

5.3 $\tilde{\mu}_{\omega_i} \leftarrow \frac{\mu_{\omega_i}\sigma'_{00} - \sum_{k=0}^{|\pi(\omega_i)|-1}(y_{\pi(\omega_i)_k} - \mu_{\pi(\omega_i)_k})\sigma'_{(k+1)0}}{\sigma'_{00}}$

5.4 $\hat{P}(Y_{\omega_i}|Y_{\pi(\omega_i)_0}, Y_{\pi(\omega_i)_1}, \ldots, Y_{\pi(\omega_i)_{|\pi(\omega_i)|-1}}) \leftarrow \frac{1}{\tilde{\sigma}_{\omega_i}\sqrt{2\pi}} e^{\frac{-(y_{\omega_i} - \tilde{\mu}_{\omega_i})^2}{2(\tilde{\sigma}_{\omega_i})^2}}$

6 return$(\hat{P}(\cdot))$

Step 5.1 from algorithm RePaNoEst requires the polynomially bounded time of $\mathcal{O}((|\pi(\omega_i)|+1)^3)$ for solving the linear system of equations. This means that if we specify a complete acyclic PDS (which models the complete joint probability), we still have a polynomially bounded algorithm. That algorithm is bounded by $\sum_{i=0}^{l-1}\mathcal{O}(i^3) = \mathcal{O}(l^4)$. Note that we mentioned earlier in section 2.1 that for the complete joint probability distribution in the case of discrete random variables, the computational requirements would be an exponentially growing function. The reason why this is not the case here, is because we are using an estimation instead of counting frequencies for all possible combinations. Indeed, in the case of using histograms, we *do* end up with an exponentially growing function as we shall see in section 5.2.

Steps 5.2 and 5.3 in algorithm RePaNoEst compute the $\tilde{\sigma}_{\omega_i}$ and $\tilde{\mu}_{\omega_i}$ parameters for the one–dimensional normal distribution to sample from. In the expression for $\tilde{\mu}_{\omega_i}$ however, we have variables $y_{\pi(\omega_i)_k}$. These are function variables in accordance with equation 2. This means that $\tilde{\mu}_{\omega_i}$ cannot actually be computed until values for those $y_{\pi(\omega_i)_k}$ are given. This is exactly according to the definition of conditionality, where the probability density function is specified, *given* values for the conditional variables. Values for these conditional variables are given when we sample for a new vector. This means that when sampling from the distribution generated by algorithm RePaNoEst, we must first actually compute $\tilde{\mu}_{\omega_i}$.

Sampling from the distribution thus made, requires to be able to sample from a normal distribution. Sampling from a "standard" normal distribution $Y_i \sim \mathcal{N}(0,1)(\mu = 0, \sigma = \sqrt{1})$ is mostly available, so we shall assume to have an algorithm for this. From the form of the normal distribution it follows that if we have $Y_i' = \mu + \sigma Y_i$, variable Y_i' is normally distributed with parameters μ and σ ($Y_i \sim \mathcal{N}(\mu, \sigma^2)$). We call this one line algorithm sampleNormalDistribution(μ, σ). Using this primitive, we can now sample a vector $Y = (Y_0, Y_1, \ldots, Y_{l-1})$ from the distribution made by algorithm RePaNoEst() as follows[2]:

[2]Note that σ' values here refer to the inverse of the covariance matrix for variables with indices $\{\omega_i\} \cup \pi(\omega_i)$, so we need to have stored these values globally in an actual implementation of RePaNoEst().

```
REPANOSAM()
1   for i ← l − 1 downto 0 do
    1.1  μ̃_{ω_i} ← (μ_{ω_i} σ'_{00} − Σ_{k=0}^{|π(ω_i)|−1} (Y_{π(ω_i)_k} − μ_{π(ω_i)_k}) σ'_{(k+1)0}) / σ'_{00}
    1.2  Y_{ω_i} ← SAMPLENORMALDISTRIBUTION(μ̃_{ω_i}, σ̃_{ω_i})
```

A special case for using the normal distribution parametric model, is when using univariate probabilities. This is modelled in algorithm UNIVARIATEDISTRIBUTION, which we introduced earlier in section 4. Using the univariate distribution gives an approach that is close to being similar to the approach named $PBIL_C$ [29]. All variables are regarded independently of another.

As we did in the discrete case, we shall now derive the application of using the normal distribution as an instance of the parametric distribution class in the search procedures. In order to implement the framework of MIMIC in our parametric setting, we require to know the differential entropy for the normal distribution. It has been shown [8] that the joint entropy over n normally distributed variables $y_0, y_1, \ldots, y_{n-1}$ can be written as follows:

$$h(y_0, y_1, \ldots, y_{n-1}) = \frac{1}{2}(n + \ln((2\pi)^n(\det\ \mathbf{\Sigma})) \tag{23}$$

It is clear that the entropy of a normally distributed random variable is completely determined by the value of σ. Using the above differential entropy information, we can specify algorithm REPANOCHAINSEA that uses the greedy algorithm employed in MIMIC to pick the chain of conditional dependencies and algorithm REPANOTREESEA that builds the tree of conditional dependencies:

```
REPANOCHAINSEA()
1   a ← new array of integer with size l
2   for i ← 0 to l − 1 do
    2.1  a[i] ← i
3   e ← 0
4   ω_{l−1} ← a[e]
5   for i ← 0 to l − 1 do
    5.1  μ_{a[i]} ← (Σ_{k=0}^{⌊τn⌋−1} Y_{a[i]}^{(S)k}) / ⌊τn⌋
    5.2  σ̆_{a[i]a[i]} ← (Σ_{k=0}^{⌊τn⌋−1} (Y_{a[i]}^{(S)k} − μ_{a[i]})^2) / ⌊τn⌋
    5.3  h_{a[i]} ← ½(1 + ln(2πσ̆_{a[i]a[i]}))
    5.4  if h_{a[i]} < h_{ω_{l−1}} then
         5.4.1  ω_{l−1} ← a[i]
         5.4.2  e ← i
6   π(ω_{l−1}) ← ∅
7   a[e] ← a[l − 1]
8   for i ← l − 2 downto 0 do
    8.1  e ← 0
    8.2  ω_i ← a[e]
    8.3  for j ← 0 to i do
         8.3.1  σ̆_{ω_{i+1}a[j]} ← (Σ_{k=0}^{⌊τn⌋−1} (Y_{ω_{i+1}}^{(S)k} − μ_{ω_{i+1}})(Y_{a[j]}^{(S)k} − μ_{a[j]})) / ⌊τn⌋
         8.3.2  σ̆_{a[j]ω_{i+1}} ← σ̆_{ω_{i+1}a[j]}
         8.3.3  h_{ω_{i+1}a[j]} ← ½(2 + ln(4π^2(σ̆_{ω_{i+1}ω_{i+1}} σ̆_{a[j]a[j]} − σ̆_{ω_{i+1}a[j]} σ̆_{a[j]ω_{i+1}})))
         8.3.4  h_{a[j]ω_{i+1}} ← h_{ω_{i+1}a[j]}
         8.3.5  if h_{a[j]ω_{i+1}} − h_{ω_{i+1}} < h_{ω_iω_{i+1}} − h_{ω_{i+1}} then
                8.3.5.1  ω_i ← a[j]
                8.3.5.2  e ← j
    8.4  π(ω_i) ← ω_{i+1}
    8.5  a[e] ← a[i]
9   return((π, ω))
```

RePaNoTreeSea()

1 $a \leftarrow$ ***new array*** of ***integer*** with size l
2 ***for*** $i \leftarrow 0$ ***to*** $l-1$ ***do***
2.1 $a[i] \leftarrow i$
3 $b^t \leftarrow$ ***new array*** of ***integer*** with size $l-1$
4 $e \leftarrow$RandomNumber(l)
5 $\omega_{l-1} \leftarrow e$
6 $\pi(\omega_{l-1}) \leftarrow \emptyset$
7 $\mu_{\omega_{l-1}} \leftarrow \frac{\sum_{k=0}^{\lfloor \tau n \rfloor - 1} Y_{\omega_{l-1}}^{(S)k}}{\lfloor \tau n \rfloor}$
8 $\breve{\sigma}_{\omega_{l-1}\omega_{l-1}} \leftarrow \frac{\sum_{k=0}^{\lfloor \tau n \rfloor - 1} (Y_{\omega_{l-1}}^{(S)k} - \mu_{\omega_{l-1}})^2}{\lfloor \tau n \rfloor}$
9 $h_{\omega_{l-1}} \leftarrow \frac{1}{2}(1 + \ln(2\pi\breve{\sigma}_{\omega_{l-1}\omega_{l-1}}))$
10 $a[e] \leftarrow a[l-1]$
11 ***for*** $i \leftarrow 0$ ***to*** $l-2$ ***do***
11.1 $\mu_{a[i]} \leftarrow \frac{\sum_{k=0}^{\lfloor \tau n \rfloor - 1} Y_{a[i]}^{(S)k}}{\lfloor \tau n \rfloor}$
11.2 $\breve{\sigma}_{a[i]a[i]} \leftarrow \frac{\sum_{k=0}^{\lfloor \tau n \rfloor - 1} (Y_{a[i]}^{(S)k} - \mu_{a[i]})^2}{\lfloor \tau n \rfloor}$
11.3 $h_{a[i]} \leftarrow \frac{1}{2}(1 + \ln(2\pi\breve{\sigma}_{a[i]a[i]}))$
11.4 $b^t[a[i]] \leftarrow \omega_{l-1}$
11.5 $\breve{\sigma}_{b^t[a[i]]a[i]} \leftarrow \frac{\sum_{k=0}^{\lfloor \tau n \rfloor - 1} (Y_{b^t[a[i]]}^{(S)k} - \mu_{b^t[a[i]]})(Y_{a[i]}^{(S)k} - \mu_{a[i]})}{\lfloor \tau n \rfloor}$
11.6 $\breve{\sigma}_{a[i]b^t[a[i]]} \leftarrow \breve{\sigma}_{b^t[a[i]]a[i]}$
11.7 $h_{b^t[a[i]]a[i]} \leftarrow \frac{1}{2}(2 + \ln(4\pi^2(\breve{\sigma}_{b^t[a[i]]b^t[a[i]]}\breve{\sigma}_{a[i]a[i]} - \breve{\sigma}_{b^t[a[i]]a[i]}\breve{\sigma}_{a[i]b^t[a[i]]})))$
11.8 $h_{a[i]b^t[a[i]]} \leftarrow h_{b^t[a[i]]a[i]}$
12 ***for*** $i \leftarrow l-2$ ***downto*** 0 ***do***
12.1 $e \leftarrow arg\ \max_j \{h_{a[j]} + h_{b^t[a[j]]} - h_{a[j]b^t[a[j]]} \mid j \in \{0, 1, \ldots, i\}\}$
12.2 $\omega_i \leftarrow a[e]$
12.3 $\pi(\omega_i) \leftarrow b^t[\omega_i]$
12.4 $a[e] \leftarrow a[i]$
12.5 ***for*** $j \leftarrow 0$ ***to*** $i-1$ ***do***
12.5.1 $\breve{\sigma}_{\omega_i a[j]} \leftarrow \frac{\sum_{k=0}^{\lfloor \tau n \rfloor - 1} (Y_{\omega_i}^{(S)k} - \mu_{\omega_i})(Y_{a[j]}^{(S)k} - \mu_{a[j]})}{\lfloor \tau n \rfloor}$
12.5.2 $\breve{\sigma}_{a[j]\omega_i} \leftarrow \breve{\sigma}_{\omega_i a[j]}$
12.5.3 $h_{\omega_i a[j]} \leftarrow \frac{1}{2}(2 + \ln(4\pi^2(\breve{\sigma}_{\omega_i\omega_i}\breve{\sigma}_{a[j]a[j]} - \breve{\sigma}_{\omega_i a[j]}\breve{\sigma}_{a[j]\omega_i})))$
12.5.4 $h_{a[j]\omega_i} \leftarrow h_{\omega_i a[j]}$
12.5.5 $I_{best} \leftarrow h_{a[j]} + h_{b^t[a[j]]} - h_{a[j]b^t[a[j]]}$
12.5.6 $I_{add} \leftarrow h_{a[j]} + h_{\omega_i} - h_{a[j]\omega_i}$
12.5.7 ***if*** $I_{best} < I_{add}$ ***then***
12.5.7.1 $b^t[a[j]] \leftarrow \omega_i$
13 return$((\pi, \boldsymbol{\omega}))$

It follows from both algorithm RePaNoChainSea and algorithm RePaNoTreeSea that we do not need to recompute the values for μ_i and $\breve{\sigma}_{ij}$ in algorithm RePaNoEst if one of these search procedures is employed. This follows from the definition of algorithm IDEA for the same reasons as in the case of discrete random variables in section 4.

It remains to specify the most involved approach, which is the general graph search approach. First of all, we present the general algorithm RePaNoGraphSea for the real parametric case using a normal distribution that distinguishes between the different cases of the amount of allowed parents κ:

```
RePaNoGraphSea(κ)
1   if κ = 0 then
    1.1   (π, ω) ← UnivariateDistribution()
2   else if κ = 1 then
    2.1   (π, ω) ← RePaNoGraphSeaExact()
3   else then
    3.1   (π, ω) ← RePaNoGraphSeaGreedy(κ)
4   return((π, ω))
```

In section 4, we already elaborated on the implementation of Edmond's algorithm for the case when $\kappa = 1$. Because the entropy measures are the same as in the case of algorithms RePaNoChainSea and RePaNoTreeSea, since in all cases only a single parent is allowed, the computation of the cost of each arc is now straightforward. The remainder of the search procedure consists of calling the optimum branching algorithm and deriving the PDS from the so obtained results. Referring to appendix E for the subalgorithms used, we can therefore now present algorithm RePaNoGraphSeaExact:

```
RePaNoGraphSeaExact()
 1   define type edarc as(stack of integer, stack of integer, stack of real)
 2   v^p ← new array of vector of integer with size l
 3   V ← new vector of integer
 4   A ← new vector of edarc
 5   for i ← 0 to l − 1 do
     5.1   μ_i ← (Σ_{k=0}^{⌊τn⌋−1} Y_i^{(S)k}) / ⌊τn⌋
     5.2   σ̆_ii ← (Σ_{k=0}^{⌊τn⌋−1} (Y_i^{(S)k} − μ_i)^2) / ⌊τn⌋
     5.3   V_{|V|} ← i
 6   for i ← 0 to l − 1 do
     6.1   for j ← i + 1 to l − 1 do
           6.1.1   σ̆_ij ← (Σ_{k=0}^{⌊τn⌋−1} (Y_i^{(S)k} − μ_i)(Y_j^{(S)k} − μ_j)) / ⌊τn⌋
           6.1.2   σ̆_ji ← σ̆_ij
     6.2   for j ← 0 to l − 1 do
           6.2.1   if i ≠ j then
                   6.2.1.1   c ← ½(2 + ln(4π²σ̆_jj σ̆_ii − σ̆_ji σ̆_ij)) − ½(1 + ln(2πσ̆_ii))
                   6.2.1.2   S^s ← new stack of integer
                   6.2.1.3   S^t ← new stack of integer
                   6.2.1.4   S^c ← new stack of real
                   6.2.1.5   push(S^s, i)
                   6.2.1.6   push(S^t, j)
                   6.2.1.7   push(S^c, −c)
                   6.2.1.8   A_{|A|} ← (S^s, S^t, S^c)
 7   γ ← min_{(S^s,S^t,S^c)∈A}{top(S^c)}
 8   for i ← 0 to |A| − 1 do
     8.1   (S^s, S^t, S^c) ← A_i
     8.2   c ← pop(S^c)
     8.3   c ← c + 1 − γ
     8.4   push(S^c, c)
 9   B ← GraphSeaOptimumBranching(V,A,l)
10   for i ← 0 to |B| − 1 do
     10.1   (S^s, S^t, S^c) ← B_i
     10.2   s ← pop(S^s)
     10.3   t ← pop(S^t)
     10.4   v^p[t]_{|v^p[t]|} ← s
11   return(GraphSeaTopologicalSort(v^p))
```

When using algorithm RePaNoGraphSeaExact, the conditional entropy is computed for every arc (i,j) with $i \neq j$. This requires to compute all variables μ_i and $\breve{\sigma}_{ij}$. This implies that again we do not need to recompute those variables when using algorithm RePaNoEst. The running time for algorithm RePaNoGraphSeaExact is $\mathcal{O}(l^3 + l^2\tau n)$, which is subsumed by the running time of the greedy graph search algorithm for the case when $\kappa > 1$..

To implement the greedy algorithm for the case when $\kappa > 1$, we note from the algorithm in section 3.4 that we require to *update* the entropy measures once an arc is added to the graph. As the entropy measure is a decomposable sum of values, it is of the form $\sum_{i=0}^{l-1} \alpha_i$. Moreover, the α_i factors are *independent* of each other, meaning that when we add arc (v_0, v_1) to the graph, we only have to recompute α_{v_1}.

If we now recall equation 23, the joint entropy is a function of the determinant of the covariance matrix of the involved variables. This covariance matrix for vertex v_1 is enlarged by adding one additional row and column when we add an arc (v_0, v_1) to the graph. We can avoid recomputing the complete determinant by using an update rule. To this end, we note that we have the following from linear algebra:

$$\frac{\det(\mathbf{\Sigma}\langle j_0, j_1, \ldots, j_n\rangle)}{\det(\mathbf{\Sigma}\langle j_0, j_1, \ldots, j_{n-1}\rangle)} = \frac{1}{\mathbf{\Sigma}\langle j_0, j_1, \ldots, j_n\rangle^{-1}(n,n)} \tag{24}$$

Therefore, by computing $\sigma'_{*,n}$ from the linear system of equations $\mathbf{\Sigma}\langle j_0, j_1, \ldots, j_n\rangle \mathbf{\Sigma}^{-1}(*,n) = I(*n)$, we get σ'_{nn}. If we have stored the determinant for the previous state of the vertex to which we are to add another parent in d^o, by using equation 24 we can determine the new determinant as $d^n = d^o/\sigma'_{nn}$. Formalizing the remainder of the concepts from the algorithm in section 3.4, we can now present the greedy algorithm RePaNoGraphSeaGreedy:

RePaNoGraphSeaGreedy(κ)
1 $a \leftarrow$ ***new array*** of ***boolean*** in 2 dimensions with size $l \times l$
2 $v^p, v^s \leftarrow 2$ ***new arrays*** of ***vector*** of ***integer*** with size l
3 $h^n, d^n_f, d^n_p, c \leftarrow 4$ ***new arrays*** of ***real*** in 2 dimensions with size $l \times l$
4 $h^o, d^o_f, d^o_p \leftarrow 3$ ***new arrays*** of ***real*** with size l
5 ***for*** $i \leftarrow 0$ ***to*** $l-1$ ***do***
  5.1 $\mu_i \leftarrow \frac{\sum_{k=0}^{\lfloor \tau n \rfloor - 1} Y_i^{(S)k}}{\lfloor \tau n \rfloor}$
  5.2 $\breve{\sigma}_{ii} \leftarrow \frac{\sum_{k=0}^{\lfloor \tau n \rfloor - 1} (Y_i^{(S)k} - \mu_i)^2}{\lfloor \tau n \rfloor}$
  5.3 $d^o_f[i] \leftarrow \breve{\sigma}_{ii}$
  5.4 $d^o_p[i] \leftarrow 0$
  5.5 $h^o[i] \leftarrow \frac{1}{2}(1 + \ln(2\pi d^o_f[i]))$
6 ***for*** $i \leftarrow 0$ ***to*** $l-1$ ***do***
  6.1 ***for*** $j \leftarrow i+1$ ***to*** $l-1$ ***do***
    6.1.1 $\breve{\sigma}_{ij} \leftarrow \frac{\sum_{k=0}^{\lfloor \tau n \rfloor - 1} (Y_i^{(S)k} - \mu_i)(Y_j^{(S)k} - \mu_j)}{\lfloor \tau n \rfloor}$
    6.1.2 $\breve{\sigma}_{ji} \leftarrow \breve{\sigma}_{ij}$
  6.2 ***for*** $j \leftarrow 0$ ***to*** $l-1$ ***do***
    6.2.1 ***if*** $i \neq j$ ***then***
      6.2.1.1 $a[i,j] \leftarrow$ ***true***
      6.2.1.2 $d^n_f[i,j] \leftarrow \breve{\sigma}_{jj}\breve{\sigma}_{ii} - \breve{\sigma}_{ji}\breve{\sigma}_{ij}$
      6.2.1.3 $d^n_p[i,j] \leftarrow \breve{\sigma}_{ii}$
      6.2.1.4 $h^n[i,j] \leftarrow \frac{1}{2}(2 + \ln(4\pi^2 d^n_f[i,j])) - \frac{1}{2}(1 + \ln(2\pi d^n_p[i,j]))$
      6.2.1.5 $c[i,j] \leftarrow h^n[i,j] - h^o[j]$
  6.3 $a[i,i] \leftarrow$ ***false***
7 $\gamma \leftarrow l^2 - l$

8 ***while*** $\gamma > 0$ ***do***
  8.1 $(v_0, v_1) \leftarrow arg\ \min_{(i,j)}\{c[i,j] \mid a[i,j] \wedge (i,j) \in \{0,1,\ldots,l-1\}^2\}$
  8.2 ***if*** $c[v_0, v_1] > 0$ ***then***
    8.2.1 ***breakwhile***
  8.3 $\gamma \leftarrow \gamma -$ GraphSeaArcAdd(κ, v_0, v_1, a, v^p, v^s)
  8.4 $d_f^o[v_1] \leftarrow d_f^n[v_0, v_1]$
  8.5 $d_p^o[v_1] \leftarrow d_p^n[v_0, v_1]$
  8.6 $h^o[v_1] \leftarrow h^n[v_0, v_1]$
  8.7 ***for*** $i \leftarrow 0$ ***to*** $l-1$ ***do***
    8.7.1 ***if*** $a[i, v_1]$ ***then***
      8.7.1.1 ***Find*** $\sigma'_{*(|v^p[v_1]|+2)}$ ***in*** $\mathcal{O}((|v^p[v_1]|+2)^3)$ ***from***
        $\boldsymbol{\Sigma}\langle v_1, v^p[v_1]_0, v^p[v_1]_1, \ldots, v^p[v_1]_{|v^p[v_1]|-1}, i\rangle \cdot$
        $(\boldsymbol{\Sigma}^{-1}(*, |v^p[v_1]|+2)) = I(*, |v^p[v_1]|+2)$
      8.7.1.2 $d_f^n[i, v_1] \leftarrow d_f^o[v_1] / \sigma'_{(|v^p[v_1]|+1)(|v^p[v_1]|+1)}$
      8.7.1.3 $h^n[i, v_1] \leftarrow \frac{1}{2}(|v^p[v_1]| + 2 + \ln((2\pi)^{|v^p[v_1]|+2} d_f^n[i, v_1]))$
      8.7.1.4 ***Find*** $\sigma'_{*(|v^p[v_1]|+1)}$ ***in*** $\mathcal{O}((|v^p[v_1]|+1)^3)$ ***from***
        $\boldsymbol{\Sigma}\langle v^p[v_1]_0, v^p[v_1]_1, \ldots, v^p[v_1]_{|v^p[v_1]|-1}, i\rangle \cdot$
        $(\boldsymbol{\Sigma}^{-1}(*, |v^p[v_1]|+1)) = I(*, |v^p[v_1]|+1)$
      8.7.1.5 $d_p^n[i, v_1] \leftarrow d_p^o[v_1] / \sigma'_{|v^p[v_1]||v^p[v_1]|}$
      8.7.1.6 $h^n[i, v_1] \leftarrow h^n[i, v_1] -$
        $\frac{1}{2}(|v^p[v_1]| + 1 + \ln((2\pi)^{|v^p[v_1]|+1} d_p^n[i, v_1]))$
      8.7.1.7 $c[i, j] \leftarrow h^n[i, v_1] - h^o[v_1]$
9 return(GraphSeaTopologicalSort(v^p))

As was the case for the discrete random variables, we have that we do not need to recompute variables μ_i and $\check{\sigma}_{ij}$ in all cases of applying a search algorithm, because they will already have been computed.

5.2 Non–parametric model: histogram distribution

In a non–parametric model, the form of the distribution is not determined at the outset as is the case for the parametric model. There are no parameters that need to be chosen to best fit a function over data points. Instead, a transformation of the data points constitutes the resulting density function. Note that we may still have parameters, but they do not solely determine the form of the fit. Examples of non–parametric models are histograms and kernel–based methods.

Out of all models, the histogram is perhaps the most straightforward way to estimate a probability density. In the continuous case, the histogram can arbitrarily well represent the distribution of given datapoints by choosing an arbitrarily small bin width. In the case of discrete random variables, the histogram is actually what has been used implicitly so far in all discrete EDA approaches.

If we have to estimate the distribution of the data points $Y_i^{(S)k}$ in a certain range $[\min^{Y_i}, \max^{Y_i}\rangle$, we specify r bins $b_i[j]$, $j \in \{0,1,\ldots,r-1\}$ that divide the range in r subsequent and equally sized subranges. Define range^{Y_i} by $\max^{Y_i} - \min^{Y_i}$. The basis of the empiric probability that y falls into bin $b_i[j]$, is a frequency count:

$$b_i[j] = \left| \left\{ Y_i^{(S)k} \;\middle|\; k \in \{0,1,\ldots,\lfloor \tau n \rfloor - 1\} \wedge j \leq \frac{Y_i^{(S)k} - \min^{Y_i}}{\text{range}^{Y_i}} r < j+1 \right\} \right| \tag{25}$$

An exponentially growing amount of bins in the form of r^n is required in the multidimensional case for variables $Y_{i_0}, Y_{i_1}, \ldots, Y_{i_{n-1}}$ and according bins $b_{i_0 i_1 \ldots i_{n-1}}[j_0, j_1, \ldots, j_{n-1}]$. It is clear that r^n bins are required for a PDS with a parent arity of $n-1$ as equation 3 dictates we require to estimate the density for a probability over n variables in that case. We should realize this *curse of dimensionality* as it limits the amount of bins we can use when the parent arity goes up.

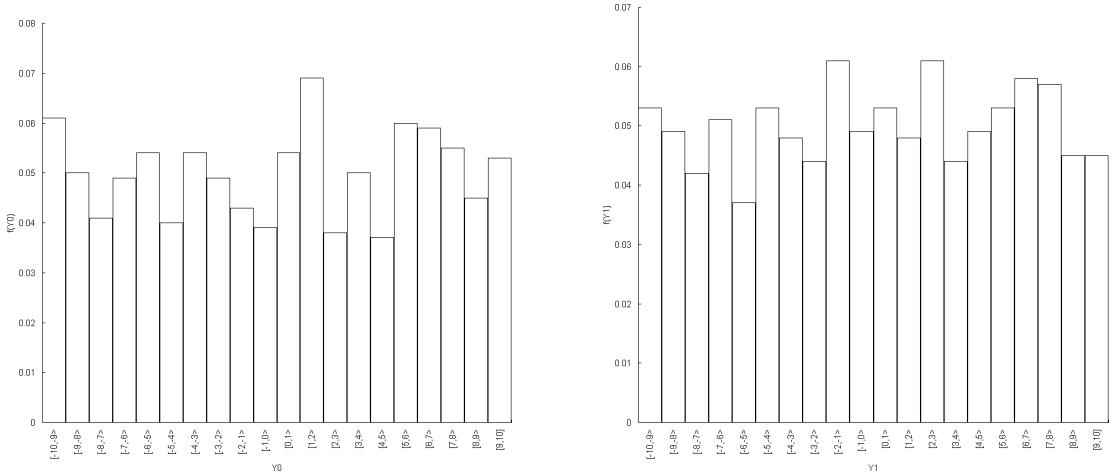

Figure 8: Fitting the univariate data from the uniform data set with a histogram density model, using 20 bins.

In figures 8 and 9, the density function using histograms over the variables taken univariately is plotted. We should be aware of the scale differences for the two sets of graphs. This difference amplifies the non–uniformity of the fit over the clustered data set. As opposed to the resulting fits with the normal distribution in section 5.1, from these figures, as well as from the joint density function in figure 10, it is clear which data set is sampled from a uniform distribution and which is sampled from a clustered distribution. Using histograms is in general a more localized method. Depending on the bin width, more or less details are conveyed in the resulting density estimation. As the figures show however, we might very well require quite a large amount of bins to representably estimate the distribution. If the amount of parents κ in the PDS graph grows, this results in a drastic increase of computation time because of the exponential growing function $r^{\kappa+1}$. In the case of binary random variables, the situation was comparable to using histograms with $r = 2$. If we however use 10 bins in each dimension, two parents already amounts to $10^3 = 1000$ bins, whereas in the binary case we could go as far as $\kappa = 9$ parents to get $2^{10} = 1024$ bins. Clearly, we are trying to solve different types of problems in the case of discrete and continuous random variables, but we should nevertheless be aware of the underlying problem in using histograms.

In the case of discrete random variables, we don't have real valued data. We can however set the minimum and maximum of the range we want to use bins for, to 0 and $n_d - 1$ respectively and specify to use n_d bins. By doing so, we have defined bins for the discrete case. This stresses again the strong correspondence between the algorithms for discrete random variables and the use of the histogram density model for continuous random variables.

We now move to specify the algorithms in the continuous case using the non–parametric histogram model. The estimation of the density functions are exactly the empiric probabilities that result from the frequency count. This leads to the algorithm ReHiEst to estimate the distribution once the structure $(\pi, \boldsymbol{\omega})$ is known. The algorithm requires a parameter r for the amount of bins to use in each dimension:

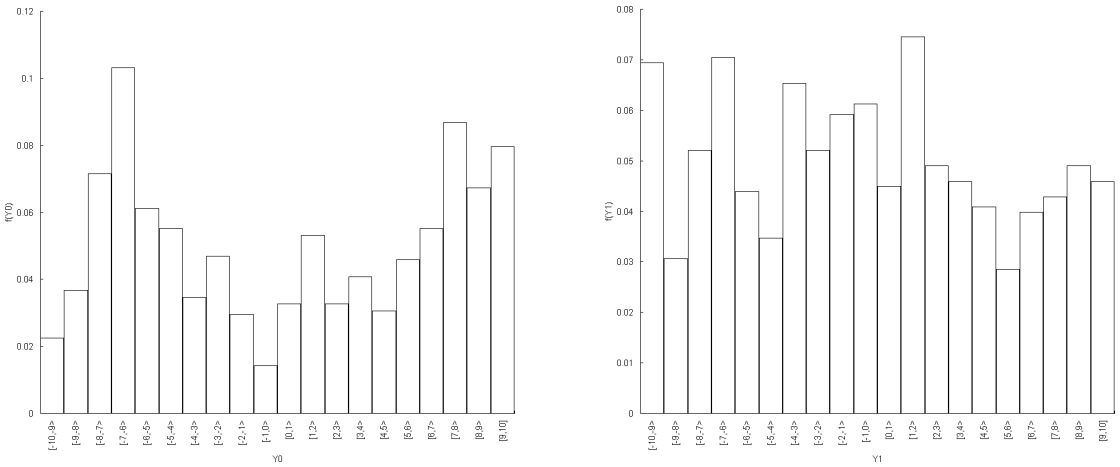

Figure 9: Fitting the univariate data from the clustered data set with a histogram density model, using 20 bins.

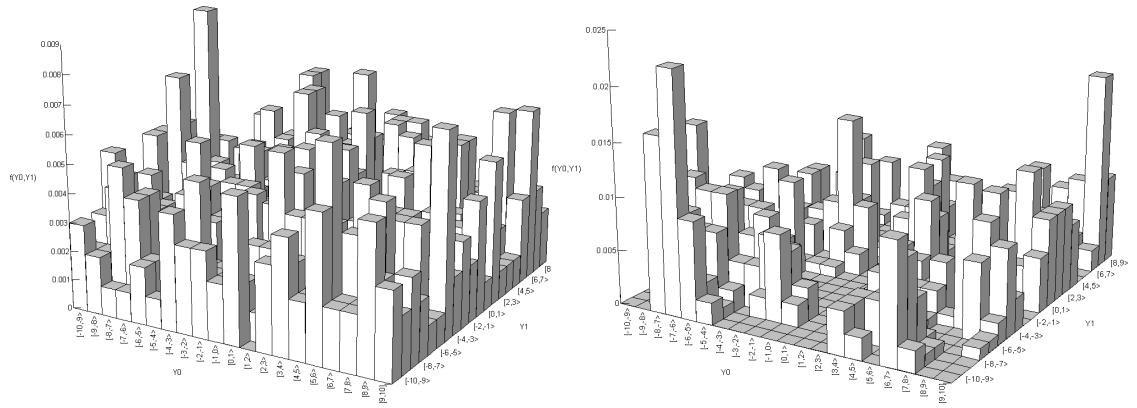

Figure 10: Fitting the joint data from the uniform (left) and clustered (right) data sets with a histogram density model, using 20 bins in each dimension.

ReHiEst(r)

1 ***for*** $i \leftarrow 0$ ***to*** $l-1$ ***do***

1.1 $\min^{Y_i} \leftarrow \min_{k\in\{0,1,\ldots,\lfloor \tau n\rfloor -1\}}\{Y_i^{(S)k}\}$

1.2 $\max^{Y_i} \leftarrow \max_{k\in\{0,1,\ldots,\lfloor \tau n\rfloor -1\}}\{Y_i^{(S)k}\}$

1.3 $\text{range}^{Y_i} \leftarrow \max^{Y_i} - \min^{Y_i}$

1.4 $\beta^{Y_i} \leftarrow \frac{\text{range}^{Y_i}}{r}$

2 ***for*** $i \leftarrow 0$ ***to*** $l-1$ ***do***

2.1 $b_{\omega_i} \leftarrow$ ***new array*** of ***real*** in $|\pi(\omega_i)|+1$ dimensions with size $r \times r \times \ldots \times r$

2.2 ${b^p}_{\omega_i} \leftarrow$ ***new array*** of ***real*** in $|\pi(\omega_i)|$ dimensions with size $r \times r \times \ldots \times r$

2.3 ***for*** $(a_0, a_1, \ldots, a_{|\pi(\omega_i)|}) \leftarrow (0,0,\ldots,0)$ ***to*** $(r-1, r-1, \ldots, r-1)$ ***in crossproduct do***

2.3.1 $b_{\omega_i}[a_0, a_1, \ldots, a_{|\pi(\omega_i)|}] \leftarrow 0$

2.4 ***for*** $(a_0, a_1, \ldots, a_{|\pi(\omega_i)|-1}) \leftarrow (0,0,\ldots,0)$ ***to*** $(r-1, r-1, \ldots, r-1)$ ***in crossproduct do***

2.4.1 $b^p_{\omega_i}[a_0, a_1, \ldots, a_{|\pi(\omega_i)|-1}] \leftarrow 0$

2.5 ***for*** $k \leftarrow 0$ ***to*** $\lfloor \tau n\rfloor - 1$ ***do***

2.5.1 $a_0 \leftarrow \min\{r-1, \lfloor (Y_{\omega_i}^{(S)k} - \min^{Y_{\omega_i}})/\beta^{Y_{\omega_i}}\rfloor\}$

2.5.2 ***for*** $q \leftarrow 0$ ***to*** $|\pi(\omega_i)|-1$ ***do***

2.5.2.1 $a_{q+1} \leftarrow \min\{r-1, \lfloor (Y^{(S)k}_{\pi(\omega_i)_q} - \min^{Y_{\pi(\omega_i)_q}})/\beta^{Y_{\pi(\omega_i)_q}}\rfloor\}$

2.5.3 $b_{\omega_i}[a_0, a_1, \ldots, a_{|\pi(\omega_i)|}] \leftarrow b_{\omega_i}[a_0, a_1, \ldots, a_{|\pi(\omega_i)|}] + \frac{1}{\lfloor \tau n\rfloor}$

2.5.4 ${b^p}_{\omega_i}[a_1, a_2, \ldots, a_{|\pi(\omega_i)|}] \leftarrow {b^p}_{\omega_i}[a_1, a_2, \ldots, a_{|\pi(\omega_i)|}] + \frac{1}{\lfloor \tau n\rfloor}$

2.6 $\hat{P}(Y_{\omega_i}|Y_{\pi(\omega_i)_0}, Y_{\pi(\omega_i)_1}, \ldots, Y_{\pi(\omega_i)_{|\pi(\omega_i)|}}) \leftarrow$

$$\begin{cases} \varphi(\omega_i, a_0, a_1, \ldots, a_{|\pi(\omega_i)|}) & \text{if } \begin{array}{l} y_{\omega_i} \in [\min^{Y_{\omega_i}}, \max^{Y_{\omega_i}}] \wedge \\ \forall_j \langle Y_{\pi(\omega_i)_j} \in [\min^{Y_{\pi(\omega_i)_j}}, \max^{Y_{\pi(\omega_i)_j}}]\rangle \end{array} \\ 0 & \text{otherwise} \end{cases}$$

where

$$\begin{cases} a_j = \min\{r-1, \overline{a}_j\} \\ \overline{a}_j = \begin{cases} \left\lfloor \frac{(y_{\omega_i} - \min^{Y_{\omega_i}})}{\beta^{Y_{\omega_i}}} \right\rfloor & \text{if } j = 0 \\ \left\lfloor \frac{(y_{\pi(\omega_i)_{j-1}} - \min^{Y_{\pi(\omega_i)_{j-1}}})}{\beta^{Y_{\pi(\omega_i)_{j-1}}}} \right\rfloor & \text{otherwise} \end{cases} \\ \varphi(\omega_i, k_0, k_1, \ldots, k_{|\pi(\omega_i)|}) = \begin{cases} \frac{b_{\omega_i}[k_0, k_1, \ldots, k_{|\pi(\omega_i)|}]}{b^p_{\omega_i}[k_1, k_2, \ldots, k_{|\pi(\omega_i)|}]} & \text{if } |\pi(\omega_i)| > 0 \\ b_{\omega_i}[k_0, k_1, \ldots, k_{|\pi(\omega_i)|}] & \text{otherwise} \end{cases} \end{cases}$$

3 return($\hat{P}(\cdot)$)

The exponential factor in the running time for algorithm ReHiEst when we take a fully connected graph, is $\mathcal{O}(r^{\kappa+1})$ because of the fact that in step 2.4, at most $\kappa+1$ counters are enumerated in crossproduct. Each enumeration consists of exactly $\mathcal{O}(r)$ steps. This can be compared to the algorithm for discrete random variables from section 4 that has an exponential factor of $\mathcal{O}(n_d^{\kappa+1})$. Just as was the case for algorithms RePaNoEst and DiEst, line 1.6 does not actually have to computed, because the required information is stored in arrays b_{ω_i} and $b^p_{\omega_i}$. Line 1.6 does however give us a way to sample from the distribution. Based on the proportionality of the frequencies stored in the bins and a randomly drawn real number between 0 and 1, we select a bin. In the case of discrete random variables, we would select the so found bin as the result for the discrete random variable. Now however, the bin stands for a range of values for a continuous random variable. To justify this, after we have selected a bin, the resulting value for the continuous random variable is a random number drawn from the uniform distribution on the range that the selected bin resembles. This is formalized in algorithm ReHiSam:

```
REHISAM(r)
1   for i ← l − 1 downto 0 do
    1.1   β ← range^{Y_{ω_i}} / r
    1.2   α ← RANDOM01()
    1.3   ς ← RANDOM01()
    1.4   ϑ ← 0
    1.5   for j ← 0 to |π_{ω_i}| − 1 do
          1.5.1   a_{j+1} ← min{r − 1, ⌊r(Y_{π(ω_i)_j} − min^{Y_{π(ω_i)_j}})/range^{Y_{π(ω_i)_j}}⌋}
    1.6   η ← 1
    1.7   if |π(ω_i)| > 0 then
          1.7.1   η ← b^p_{ω_i}[a_1, a_2, ..., a_{|π_{ω_i}|}]
    1.6   for a_0 ← 0 to r − 1 do
          1.6.1   ϑ ← ϑ + (1/η) b_{ω_i}[a_0, a_1, ..., a_{|π_{ω_i}|}]
          1.6.2   if ς ≤ ϑ then Y_{ω_i} ← min^{Y_{ω_i}} + (a_0 + α)β; breakfor
```

As in the parametric case, we are now able to create a full algorithm by using UNIVARIATEDISTRIBUTION, JOINTDISTRIBUTION or any other a priori fixed distribution. It may be clear that because of quite similar reasons as we came across for the algorithms for discrete random variables, the recomputation of arrays b_i and b_i^p in method REHIEST is not required after having applied any of the search algorithms we will present next.

In order to define the search algorithms, we should remind ourselves again that we need to be able to compute the entropy. In the case of the histogram distribution, this can be done in a similar way as in the case of discrete random variables. In appendix D it is shown that the multivariate entropy measure for continuous random variables that have a histogram distribution, equals:

$$h(Y_{j_0}, Y_{j_1}, \ldots, Y_{j_{n-1}}) = \tag{26}$$

$$-\frac{\prod_{i=0}^{n-1} \text{range}^{Y_{j_i}}}{r^n} \sum_{k_0=0}^{r-1} \sum_{k_1=0}^{r-1} \cdots \sum_{k_{n-1}=0}^{r-1} \left(b_{j_0,j_1,\ldots,j_{n-1}}[k_0, k_1, \ldots, k_{n-1}] \ln(b_{j_0,j_1,\ldots,j_{n-1}}[k_0, k_1, \ldots, k_{n-1}]) \right)$$

$$\text{where} \quad b_{j_0,j_1,\ldots,j_{n-1}}[k_0, k_1, \ldots, k_{n-1}] \;=$$

$$\left| \left\{ Y_{j_0}^{(S)q} \;\middle|\; q \in \{0, 1, \ldots, \lfloor \tau n \rfloor - 1\} \;\wedge \forall_{i \in \{0,1,\ldots,n-1\}} \left\langle k_i \leq \frac{Y_{j_i}^{(S)q} - \min^{Y_{j_i}}}{\text{range}^{Y_{j_i}}} r < k_i + 1 \right\rangle \right\} \right|$$

We now have enough tools to write out the algorithms for the chain search, the tree search and the graph search. We denote these algorithms as REHICHAINSEA, REHITREESEA and REHIGRAPHSEA respectively. When going over the algorithms, note the obvious similarity with the algorithms for discrete random variables from section 4. The running time for algorithm REHIGRAPHSEAEXACT is $\mathcal{O}(l^3 + l^2 r^2 + l^2 \tau n)$, which is subsumed by the running time of the greedy graph search algorithm for the case when $\kappa > 1$.

```
REHICHAINSEA(r)
1   a ← new array of integer with size l
2   for i ← 0 to l − 1 do
    2.1   a[i] ← i
    2.2   p_i ← new array of real with size r
3   e ← 0
4   ω_{l−1} ← a[e]
```

5 ***for*** $i \leftarrow 0$ ***to*** $l-1$ ***do***
  5.1 $\min^{Y_{a[i]}} \leftarrow \min_{k\in\{0,1,\ldots,\lfloor \tau n\rfloor -1\}}\{Y_{a[i]}^{(S)k}\}$
  5.2 $\max^{Y_{a[i]}} \leftarrow \max_{k\in\{0,1,\ldots,\lfloor \tau n\rfloor -1\}}\{Y_{a[i]}^{(S)k}\}$
  5.3 $\mathrm{range}^{Y_{a[i]}} \leftarrow \max^{Y_{a[i]}} - \min^{Y_{a[i]}}$
  5.4 $\beta^{Y_{a[i]}} \leftarrow \frac{\mathrm{range}^{Y_{a[i]}}}{r}$
  5.5 ***for*** $j \leftarrow 0$ ***to*** $r-1$ ***do***
    5.5.1 $p_{a[i]}[j] \leftarrow 0$
  5.6 ***for*** $j \leftarrow 0$ ***to*** $\lfloor \tau n\rfloor - 1$ ***do***
    5.6.1 $k \leftarrow \min\{r-1, \lfloor (Y_{a[i]}^{(S)j} - \min^{Y_{a[i]}})/\beta^{Y_{a[i]}}\rfloor\}$
    5.6.2 $p_{a[i]}[k] \leftarrow p_{a[i]}[k] + \frac{1}{\lfloor \tau n\rfloor}$
  5.7 $h_{a[i]} \leftarrow -\frac{\mathrm{range}^{Y_{a[i]}}}{r}\sum_{k=0}^{r-1} p_{a[i]}[k]\ln(p_{a[i]}[k])$
  5.8 ***if*** $h_{a[i]} < h_{\omega_{l-1}}$ ***then***
    5.8.1 $\omega_{l-1} \leftarrow a[i]$
    5.8.2 $e \leftarrow i$
6 $\pi(\omega_{l-1}) \leftarrow \emptyset$
7 $a[e] \leftarrow a[l-1]$
8 ***for*** $i \leftarrow l-2$ ***downto*** 0 ***do***
  8.1 $e \leftarrow 0$
  8.2 $\omega_i \leftarrow a[e]$
  8.3 ***for*** $q \leftarrow 0$ ***to*** i ***do***
    8.3.1 $p_{\omega_{i+1}a[q]} \leftarrow$ ***new array*** of ***real*** in 2 dimensions with size $r \times r$
    8.3.2 $p_{a[q]\omega_{i+1}} \leftarrow$ ***new array*** of ***real*** in 2 dimensions with size $r \times r$
    8.3.3 ***for*** $j_0 \leftarrow 0$ ***to*** $r-1$ ***do***
      8.3.3.1 ***for*** $j_1 \leftarrow 0$ ***to*** $r-1$ ***do***
        8.3.3.1.1 $p_{\omega_{i+1}a[q]}[j_0, j_1] \leftarrow 0$
        8.3.3.1.2 $p_{a[q]\omega_{i+1}}[j_1, j_0] \leftarrow 0$
    8.3.4 ***for*** $j \leftarrow 0$ ***to*** $\lfloor \tau n\rfloor - 1$ ***do***
      8.3.4.1 $k_0 \leftarrow \min\{r-1, \lfloor (Y_{\omega_{i+1}}^{(S)j} - \min^{Y_{\omega_{i+1}}})/\beta^{Y_{\omega_{i+1}}}\rfloor\}$
      8.3.4.2 $k_1 \leftarrow \min\{r-1, \lfloor (Y_{a[q]}^{(S)j} - \min^{Y_{a[q]}})/\beta^{Y_{a[q]}}\rfloor\}$
      8.3.4.3 $p_{\omega_{i+1}a[q]}[k_0, k_1] \leftarrow p_{\omega_{i+1}a[q]}[k_0, k_1] + \frac{1}{\lfloor \tau n\rfloor}$
      8.3.4.4 $p_{a[q]\omega_{i+1}}[k_1, k_0] \leftarrow p_{a[q]\omega_{i+1}}[k_1, k_0] + \frac{1}{\lfloor \tau n\rfloor}$
    8.3.5 $h_{\omega_{i+1}a[q]} \leftarrow -\frac{\mathrm{range}^{Y_{\omega_{i+1}}}\mathrm{range}^{Y_{a[q]}}}{r^2}\cdot \sum_{k_0=0}^{r-1}\sum_{k_1=0}^{r-1} p_{\omega_{i+1}a[q]}[k_0, k_1]\ln(p_{\omega_{i+1}a[q]}[k_0, k_1])$
    8.3.6 $h_{a[q]\omega_{i+1}} \leftarrow h_{\omega_{i+1}a[q]}$
    8.3.7 ***if*** $h_{a[q]\omega_{i+1}} - h_{\omega_{i+1}} < h_{\omega_i\omega_{i+1}} - h_{\omega_{i+1}}$ ***then***
      8.3.7.1 $\omega_i \leftarrow a[q]$
      8.3.7.2 $e \leftarrow q$
  8.4 $\pi(\omega_i) \leftarrow \omega_{i+1}$
  8.5 $a[e] \leftarrow a[i]$
9 ***for*** $i \leftarrow 0$ ***to*** $l-1$ ***do***
  9.1 ***if*** $|\pi(\omega_i)| > 0$ ***then***
    9.1.1 $b_{\omega_i} \leftarrow p_{\omega_i\pi(\omega_i)_0}$
    9.1.2 $b_{\omega_i}^p \leftarrow p_{\pi(\omega_i)_0}$
  9.2 ***else***
    9.2.1 $b_{\omega_i} \leftarrow p_{\omega_i}$
10 return$((\pi, \boldsymbol{\omega}))$

ReHiTreeSea(r)

1 $a \leftarrow$ ***new array*** of ***integer*** with size l
2 ***for*** $i \leftarrow 0$ ***to*** $l-1$ ***do***
2.1 $a[i] \leftarrow i$
2.2 $p_i \leftarrow$ ***new array*** of ***real*** with size r
3 $b^t \leftarrow$ ***new array*** of ***integer*** with size $l-1$
4 $e \leftarrow$RandomNumber(l)
5 $\omega_{l-1} \leftarrow e$
6 $\pi(\omega_{l-1}) \leftarrow \emptyset$
7 $\min^{Y_{\omega_{l-1}}} \leftarrow \min_{k \in \{0,1,\ldots,\lfloor \tau n \rfloor - 1\}} \{Y^{(S)k}_{\omega_{l-1}}\}$
8 $\max^{Y_{\omega_{l-1}}} \leftarrow \max_{k \in \{0,1,\ldots,\lfloor \tau n \rfloor - 1\}} \{Y^{(S)k}_{\omega_{l-1}}\}$
9 $\text{range}^{Y_{\omega_{l-1}}} \leftarrow \max^{Y_{\omega_{l-1}}} - \min^{Y_{\omega_{l-1}}}$
10 $\beta^{Y_{\omega_{l-1}}} \leftarrow \frac{\text{range}^{Y_{\omega_{l-1}}}}{r}$
11 ***for*** $j \leftarrow 0$ ***to*** $r-1$ ***do***
11.1 $p_{\omega_{l-1}}[j] \leftarrow 0$
12 ***for*** $j \leftarrow 0$ ***to*** $\lfloor \tau n \rfloor - 1$ ***do***
12.1 $k \leftarrow \min\{r-1, \lfloor (Y^{(S)j}_{\omega_{l-1}} - \min^{Y_{\omega_{l-1}}})/\beta^{Y_{\omega_{l-1}}} \rfloor\}$
12.2 $p_{\omega_{l-1}}[k] \leftarrow p_{\omega_{l-1}}[k] + \frac{1}{\lfloor \tau n \rfloor}$
13 $h_{\omega_{l-1}} \leftarrow -\frac{\text{range}^{Y_{\omega_{l-1}}}}{r} \sum_{k=0}^{r-1} p_{\omega_{l-1}}[k] \ln(p_{\omega_{l-1}}[k])$
14 $a[e] \leftarrow a[l-1]$
15 ***for*** $i \leftarrow 0$ ***to*** $l-2$ ***do***
15.1 $\min^{Y_{a[i]}} \leftarrow \min_{k \in \{0,1,\ldots,\lfloor \tau n \rfloor - 1\}} \{Y^{(S)k}_{a[i]}\}$
15.2 $\max^{Y_{a[i]}} \leftarrow \max_{k \in \{0,1,\ldots,\lfloor \tau n \rfloor - 1\}} \{Y^{(S)k}_{a[i]}\}$
15.3 $\text{range}^{Y_{a[i]}} \leftarrow \max^{Y_{a[i]}} - \min^{Y_{a[i]}}$
15.4 $\beta^{Y_{a[i]}} \leftarrow \frac{\text{range}^{Y_{a[i]}}}{r}$
15.5 ***for*** $j \leftarrow 0$ ***to*** $r-1$ ***do***
15.5.1 $p_{a[i]}[j] \leftarrow 0$
15.6 ***for*** $j \leftarrow 0$ ***to*** $\lfloor \tau n \rfloor - 1$ ***do***
15.6.1 $k \leftarrow \min\{r-1, \lfloor (Y^{(S)j}_{a[i]} - \min^{Y_{a[i]}})/\beta^{Y_{a[i]}} \rfloor\}$
15.6.2 $p_{a[i]}[k] \leftarrow p_{a[i]}[k] + \frac{1}{\lfloor \tau n \rfloor}$
15.7 $h_{a[i]} \leftarrow -\frac{\text{range}^{Y_{a[i]}}}{r} \sum_{k=0}^{r-1} p_{a[i]}[k] \ln(p_{a[i]}[k])$
15.8 $b^t[a[i]] \leftarrow \omega_{l-1}$
15.9 $p_{b^t[a[i]]a[i]} \leftarrow$ ***new array*** of ***real*** in 2 dimensions with size $r \times r$
15.10 $p_{a[i]b^t[a[i]]} \leftarrow$ ***new array*** of ***real*** in 2 dimensions with size $r \times r$
15.11 ***for*** $j_0 \leftarrow 0$ ***to*** $r-1$ ***do***
15.11.1 ***for*** $j_1 \leftarrow 0$ ***to*** $r-1$ ***do***
15.11.1.1 $p_{b^t[a[i]]a[i]}[j_0, j_1] \leftarrow 0$
15.11.1.2 $p_{a[i]b^t[a[i]]}[j_1, j_0] \leftarrow 0$
15.12 ***for*** $j \leftarrow 0$ ***to*** $\lfloor \tau n \rfloor - 1$ ***do***
15.12.1 $k_0 \leftarrow \min\{r-1, \lfloor (Y^{(S)j}_{b^t[a[i]]} - \min^{Y_{b^t[a[i]]}})/\beta^{Y_{b^t[a[i]]}} \rfloor\}$
15.12.2 $k_1 \leftarrow \min\{r-1, \lfloor (Y^{(S)j}_{a[i]} - \min^{Y_{a[i]}})/\beta^{Y_{a[i]}} \rfloor\}$
15.12.3 $p_{b^t[a[i]]a[i]}[k_0, k_1] \leftarrow p_{b^t[a[i]]a[i]}[k_0, k_1] + \frac{1}{\lfloor \tau n \rfloor}$
15.12.4 $p_{a[i]b^t[a[i]]}[k_1, k_0] \leftarrow p_{a[i]b^t[a[i]]}[k_1, k_0] + \frac{1}{\lfloor \tau n \rfloor}$
15.13 $h_{b^t[a[i]]a[i]} \leftarrow -\frac{\text{range}^{Y_{b^t[a[i]]}} \text{range}^{Y_{a[i]}}}{r^2} \cdot \sum_{k_0=0}^{r-1} \sum_{k_1=0}^{r-1} p_{b^t[a[i]]a[i]}[k_0, k_1] \ln(p_{b^t[a[i]]a[i]}[k_0, k_1])$
15.14 $h_{a[i]b^t[a[i]]} \leftarrow h_{b^t[a[i]]a[i]}$

16 ***for*** $i \leftarrow l-2$ ***downto*** 0 ***do***
  16.1 $e \leftarrow arg\ \max_j \{h_{a[j]} + h_{b^t[a[j]]} - h_{a[j]b^t[a[j]]}\}\ \ (j \in \{0,1,\ldots,i\})$
  16.2 $\omega_i \leftarrow a[e]$
  16.3 $\pi(\omega_i) \leftarrow b^t[\omega_i]$
  16.4 $a[e] \leftarrow a[i]$
  16.5 ***for*** $j \leftarrow 0$ ***to*** $i-1$ ***do***
    16.5.1 $p_{\omega_i a[j]} \leftarrow$ ***new array*** of ***real*** in 2 dimensions with size $r \times r$
    16.5.2 $p_{a[j]\omega_i} \leftarrow$ ***new array*** of ***real*** in 2 dimensions with size $r \times r$
    16.5.3 ***for*** $j_0 \leftarrow 0$ ***to*** $r-1$ ***do***
      16.5.3.1 ***for*** $j_1 \leftarrow 0$ ***to*** $r-1$ ***do***
        16.5.3.1.1 $p_{\omega_i a[j]}[j_0, j_1] \leftarrow 0$
        16.5.3.1.2 $p_{a[j]\omega_i}[j_1, j_0] \leftarrow 0$
    16.5.4 ***for*** $k \leftarrow 0$ ***to*** $\lfloor \tau n \rfloor - 1$ ***do***
      16.5.4.1 $k_0 \leftarrow \min\{r-1, \lfloor (Y_{\omega_i}^{(S)k} - \min^{Y_{\omega_i}})/\beta^{Y_{\omega_i}} \rfloor\}$
      16.5.4.2 $k_1 \leftarrow \min\{r-1, \lfloor (Y_{a[j]}^{(S)k} - \min^{Y_{a[j]}})/\beta^{Y_{a[j]}} \rfloor\}$
      16.5.4.3 $p_{\omega_i a[j]}[k_0, k_1] \leftarrow p_{\omega_i a[j]}[k_0, k_1] + \frac{1}{\lfloor \tau n \rfloor}$
      16.5.4.4 $p_{a[j]\omega_i}[k_1, k_0] \leftarrow p_{a[j]\omega_i}[k_1, k_0] + \frac{1}{\lfloor \tau n \rfloor}$
    16.5.5 $h_{\omega_i a[j]} \leftarrow -\frac{\text{range}^{Y_{\omega_i}} \text{range}^{Y_{a[j]}}}{r^2} \cdot \sum_{k_0=0}^{r-1} \sum_{k_1=0}^{r-1} p_{\omega_i a[j]}[k_0, k_1] \ln(p_{\omega_i a[j]}[k_0, k_1])$
    16.5.6 $h_{a[j]\omega_i} \leftarrow h_{\omega_i a[j]}$
    16.5.7 $I_{best} \leftarrow h_{a[j]} + h_{b^t[a[j]]} - h_{a[j]b^t[a[j]]}$
    16.5.8 $I_{add} \leftarrow h_{a[j]} + h_{\omega_i} - h_{a[j]\omega_i}$
    16.5.9 ***if*** $I_{best} < I_{add}$ ***then***
      16.5.9.1 $b^t[a[j]] \leftarrow \omega_i$
17 ***for*** $i \leftarrow 0$ ***to*** $l-1$ ***do***
  17.1 ***if*** $|\pi(\omega_i)| > 0$ ***then***
    17.1.1 $b_{\omega_i} \leftarrow p_{\omega_i \pi(\omega_i)_0}$
    17.1.2 $b^p_{\omega_i} \leftarrow p_{\pi(\omega_i)_0}$
  17.2 ***else***
    17.2.1 $b_{\omega_i} \leftarrow p_{\omega_i}$
18 return$((\pi, \boldsymbol{\omega}))$

ReHiGraphSea(κ, r)
1 ***if*** $\kappa = 0$ ***then***
  1.1 $(\pi, \boldsymbol{\omega}) \leftarrow$ UnivariateDistribution()
2 ***else if*** $\kappa = 1$ ***then***
  2.1 $(\pi, \boldsymbol{\omega}) \leftarrow$ ReHiGraphSeaExact(r)
3 ***else then***
  3.1 $(\pi, \boldsymbol{\omega}) \leftarrow$ ReHiGraphSeaGreedy(κ, r)
4 return$((\pi, \boldsymbol{\omega}))$

```
ReHiGraphSeaExact(r)
1   define type edarc as
                (stack of integer, stack of integer, stack of real)
2   v^p ← new array of vector of integer with size l
3   V ← new vector of integer
4   A ← new vector of edarc
5   for i ← 0 to l − 1 do
    5.1   min^{Y_i} ← min_{k∈{0,1,...,⌊τn⌋−1}}{Y_i^{(S)k}}
    5.2   max^{Y_i} ← max_{k∈{0,1,...,⌊τn⌋−1}}{Y_i^{(S)k}}
    5.3   range^{Y_i} ← max^{Y_i} − min^{Y_i}
    5.4   β^{Y_i} ← range^{Y_i} / r
    5.5   p_i ← new array of real with size r
    5.6   for j ← 0 to r − 1 do
          5.6.1   p_i[j] ← 0
    5.7   for j ← 0 to ⌊τn⌋ − 1 do
          5.7.1   k ← min{r − 1, ⌊(Y_{ω_{l−1}}^{(S)j} − min^{Y_{ω_{l−1}}})/β^{Y_{ω_{l−1}}}⌋}
          5.7.2   p_i[k] ← p_i[k] + 1/⌊τn⌋
    5.8   h_i ← −(range^{Y_i}/r) Σ_{k=0}^{r−1} p_i[k] ln(p_i[k])
6   for i ← 0 to l − 1 do
    6.1   for j ← i + 1 to l − 1 do
          6.1.1   p_ij ← new array of real in 2 dimensions with size r × r
          6.1.2   p_ji ← new array of real in 2 dimensions with size r × r
          6.1.3   for j_0 ← 0 to r − 1 do
                  6.1.3.1   for j_1 ← 0 to r − 1 do
                            6.1.3.1.1   p_ij[j_0, j_1] ← 0
                            6.1.3.1.2   p_ji[j_1, j_0] ← 0
          6.1.4   for k ← 0 to ⌊τn⌋ − 1 do
                  6.1.4.1   k_0 ← min{r − 1, ⌊(Y_i^{(S)k} − min^{Y_i})/β^{Y_i}⌋}
                  6.1.4.2   k_1 ← min{r − 1, ⌊(Y_j^{(S)k} − min^{Y_j})/β^{Y_j}⌋}
                  6.1.4.3   p_ij[k_0, k_1] ← p_ij[k_0, k_1] + 1/⌊τn⌋
                  6.1.4.4   p_ji[k_1, k_0] ← p_ji[k_1, k_0] + 1/⌊τn⌋
    6.2   for j ← 0 to l − 1 do
          6.2.1   if i ≠ j then
                  6.2.1.1   h_ji ← −(range^{Y_j} range^{Y_i} / r^2) ·
                                Σ_{k_0=0}^{r−1} Σ_{k_1=0}^{r−1} p_ji[k_0, k_1] ln(p_ji[k_0, k_1])
                  6.2.1.2   c ← h_ji − h_i
                  6.2.1.3   S^s ← new stack of integer
                  6.2.1.4   S^t ← new stack of integer
                  6.2.1.5   S^c ← new stack of real
                  6.2.1.6   PUSH(S^s, i)
                  6.2.1.7   PUSH(S^t, j)
                  6.2.1.8   PUSH(S^c, −c)
                  6.2.1.9   A_{|A|} ← (S^s, S^t, S^c)
7   γ ← min_{(S^s,S^t,S^c)∈A}{TOP(S^c)}
8   for i ← 0 to |A| − 1 do
    8.1   (S^s, S^t, S^c) ← A_i
    8.2   c ← POP(S^c)
    8.3   c ← c + 1 − γ
    8.4   PUSH(S^c, c)
9   B ← GraphSeaOptimumBranching(V,A,l)
```

```
10  for i ← 0 to |B| − 1 do
      10.1  (S^s, S^t, S^c) ← B_i
      10.2  s ← POP(S^s)
      10.3  t ← POP(S^t)
      10.4  v^p[t]_{|v^p[t]|} ← s
11  (π, ω) ← GRAPHSEATOPOLOGICALSORT(v^p)
12  for i ← 0 to l − 1 do
      12.1  if |π(ω_i)| > 0 then
              12.1.1  b_{ω_i} ← p_{ω_i π(ω_i)_0}
              12.1.2  b^p_{ω_i} ← p_{π(ω_i)_0}
      12.2  else
              12.2.1  b_{ω_i} ← p_{ω_i}
13  return((π, ω))
```

```
REHIGRAPHSEAGREEDY(κ, r)
1   a ← new array of boolean in 2 dimensions with size l × l
2   v^p, v^s ← 2 new arrays of vector of integer with size l
3   h^n, c ← 2 new arrays of real in 2 dimensions with size l × l
4   h^o ← new array of real with size l
5   for i ← 0 to l − 1 do
      5.1   min^{Y_i} ← min_{k∈{0,1,...,⌊τn⌋−1}}{Y_i^{(S)k}}
      5.2   max^{Y_i} ← max_{k∈{0,1,...,⌊τn⌋−1}}{Y_i^{(S)k}}
      5.3   range^{Y_i} ← max^{Y_i} − min^{Y_i}
      5.4   β^{Y_i} ← range^{Y_i} / r
      5.5   p^p_{0i} ← new array of real with size r
      5.6   for j ← 0 to r − 1 do
              5.6.1   p^p_{0i}[j] ← 0
      5.7   for j ← 0 to ⌊τn⌋ − 1 do
              5.7.1   k ← min{r − 1, ⌊(Y^{(S)j}_{ω_{l−1}} − min^{Y_{ω_{l−1}}})/β^{Y_{ω_{l−1}}}⌋}
              5.7.2   p^p_{0i}[k] ← p^p_{0i}[k] + 1/⌊τn⌋
      5.8   for q ← 1 to l − 1 do
              5.8.1   p^p_{qi} ← new array of real with size r
              5.8.2   for j ← 0 to r − 1 do
                        5.8.2.1   p^p_{qi}[j] ← p^p_{0i}[j]
      5.9   h^o[i] ← −(range^{Y_i}/r) Σ_{k=0}^{r−1} p^p_{0i}[k] ln(p^p_{0i}[k])
      5.10  b_i ← p^p_{0i}
6   for i ← 0 to l − 1 do
      6.1   for j ← i + 1 to l − 1 do
              6.1.1   p_{ij} ← new array of real in 2 dimensions with size r × r
              6.1.2   p_{ji} ← new array of real in 2 dimensions with size r × r
              6.1.3   for j_0 ← 0 to r − 1 do
                        6.1.3.1   for j_1 ← 0 to r − 1 do
                                    6.1.3.1.1   p_{ij}[j_0, j_1] ← 0
                                    6.1.3.1.2   p_{ji}[j_1, j_0] ← 0
              6.1.4   for j ← 0 to ⌊τn⌋ − 1 do
                        6.1.4.1   k_0 ← min{r − 1, ⌊(Y_i^{(S)k} − min^{Y_i})/β^{Y_i}⌋}
                        6.1.4.2   k_1 ← min{r − 1, ⌊(Y_j^{(S)k} − min^{Y_j})/β^{Y_j}⌋}
                        6.1.4.3   p_{ij}[k_0, k_1] ← p_{ij}[k_0, k_1] + 1/⌊τn⌋
                        6.1.4.4   p_{ji}[k_1, k_0] ← p_{ji}[k_1, k_0] + 1/⌊τn⌋
```

6.2 ***for*** $j \leftarrow 0$ ***to*** $l-1$ ***do***
6.2.1 ***if*** $i \neq j$ ***then***
6.2.1.1 $a[i,j] \leftarrow$ ***true***
6.2.1.2 $h^n[i,j] \leftarrow (-\frac{\text{range}^{Y_i}\text{range}^{Y_j}}{r^2} \cdot \sum_{k_0=0}^{r-1}\sum_{k_1=0}^{r-1} p_{ji}[k_0,k_1]\ln(p_{ji}[k_0,k_1])) - h^o[i]$
6.2.1.3 $c[i,j] \leftarrow h^n[i,j] - h^o[j]$
6.3 $a[i,i] \leftarrow$ ***false***
7 $\gamma \leftarrow l^2 - l$
8 ***while*** $\gamma > 0$ ***do***
8.1 $(v_0, v_1) \leftarrow arg\ \min_{(i,j)}\{c[i,j] \mid a[i,j] \wedge (i,j) \in \{0,1,\ldots,l-1\}^2\}$
8.2 ***if*** $c[v_0,v_1] > 0$ ***then***
8.2.1 ***breakwhile***
8.3 $\gamma \leftarrow \gamma -$ GraphSeaArcAdd(κ, v_0, v_1, a, v^p, v^s)
8.4 $h^o[v_1] \leftarrow h^n[v_0,v_1]$
8.5 $b^p_{v_1} \leftarrow p^p_{v_1 v_0}$
8.6 $b_{v_1} \leftarrow p_{v_1 v_0}$
8.7 ***for*** $i \leftarrow 0$ ***to*** $l-1$ ***do***
8.7.1 ***if*** $a[i,v_1]$ ***then***
8.7.1.1 $p_{v_1 i} \leftarrow$ ***new array*** of ***real*** in $|v^p[v_1]|+2$ dimensions with size $r \times r \times \ldots \times r$
8.7.1.2 $p^p_{v_1 i} \leftarrow$ ***new array*** of ***real*** in $|v^p[v_1]|+1$ dimensions with size $r \times r \times \ldots \times r$
8.7.1.3 ***for*** $(j_0, j_1, \ldots, j_{|v^p[v_1]|+1}) \leftarrow (0,0,\ldots,0)$ ***to*** $(r-1, r-1, \ldots, r-1)$ ***in crossproduct do***
8.7.1.3.1 $p_{v_1 i}[j_0, j_1, \ldots, j_{|v^p[v_1]|+1}] \leftarrow 0$
8.7.1.4 ***for*** $(j_0, j_1, \ldots, j_{|v^p[v_1]|}) \leftarrow (0,0,\ldots,0)$ ***to*** $(r-1, r-1, \ldots, r-1)$ ***in crossproduct do***
8.7.1.4.1 $p^p_{v_1 i}[j_0, j_1, \ldots, j_{|v^p[v_1]|}] \leftarrow 0$
8.7.1.5 ***for*** $j \leftarrow 0$ ***to*** $\lfloor \tau n \rfloor - 1$ ***do***
8.7.1.5.1 $k_0 \leftarrow \min\{r-1, \lfloor (Y_{v_1}^{(S)j} - \min^{Y_{v_1}})/\beta^{Y_{v_1}} \rfloor\}$
8.7.1.5.2 $k_q \leftarrow \min\{r-1, \lfloor (Y^{(S)j}_{v^p[v_1]_{q-1}} - \min^{Y_{v^p[v_1]_{q-1}}})/\beta^{Y_{v^p[v_1]_{q-1}}} \rfloor\}$ $(\forall_{q \in \{1,2,\ldots,|v^p[v_1]|\}})$
8.7.1.5.3 $k_{|v^p[v_1]|+1} \leftarrow \min\{r-1, \lfloor (Y_i^{(S)j} - \min^{Y_i})/\beta^{Y_i} \rfloor\}$
8.7.1.5.4 $p_{v_1 i}[k_0, k_1, \ldots, k_{|v^p[v_1]|+1}] \leftarrow p_{v_1 i}[k_0, k_1, \ldots, k_{|v^p[v_1]|+1}] + \frac{1}{\lfloor \tau n \rfloor}$
8.7.1.5.5 $p^p_{v_1 i}[k_1, k_2, \ldots, k_{|v^p[v_1]|}] \leftarrow p^p_{v_1 i}[k_1, k_2, \ldots, k_{|v^p[v_1]|}] + \frac{1}{\lfloor \tau n \rfloor}$
8.7.1.6 $\alpha \leftarrow \text{range}^{Y_i} \prod_{q=0}^{|v^p[v_1]|-1} \text{range}^{Y_{v^p[v_1]_q}}$
8.7.1.7 $h^n[i,j] \leftarrow -\frac{\text{range}^{Y_{v_1}}\alpha\text{range}^{Y_j}}{r^{|v^p[v_1]|+2}} \sum_{k_0=0}^{r-1}\sum_{k_1=0}^{r-1}\cdots\sum_{k_{|v^p[v_1]|+1}}^{r-1} p_{v_1 i}[k_0, k_1, \ldots, k_{|v^p[v_1]|+1}]\ln(p_{v_1 i}[k_0, k_1, \ldots, k_{|v^p[v_1]|+1}])$
8.7.1.8 $h^n[i,j] \leftarrow h^n[i,j] - (-\frac{\alpha\text{range}^{Y_j}}{r^{|v^p[v_1]|+1}} \sum_{k_0=0}^{r-1}\sum_{k_1=0}^{r-1}\cdots\sum_{k_{|v^p[v_1]|}}^{r-1} p^p_{v_1 i}[k_0, k_1, \ldots, k_{|v^p[v_1]|}]\ln(p^p_{v_1 i}[k_0, k_1, \ldots, k_{|v^p[v_1]|}]))$
8.7.1.9 $c[i,j] \leftarrow h^n[i,v_1] - h^o[v_1]$
9 return(GraphSeaTopologicalSort(v^p))

6 Discussion

The IDEA framework allows to implement density estimation based evolutionary algorithms in a way that decomposes certain aspects within these algorithms. This allows us to clearly show how different density models map onto different procedures that are parameters to the algorithmic

framework. Next to showing how the framework allows for applying certain density models to optimization, the derived results should be brought into practice. Currently, we are in the process of implementing all of the algorithms presented in this paper. The first aspect to be published on short notice, is the results of running the proposed algorithms on benchmark tests and comparing them with other methods.

Next to bringing the algorithms into practice in the sense of a runnable program, there are some important additional issues in general in the use of the IDEA framework. For instance, many theoretical questions can be addressed. Such questions vary from population size requirements to selection issues and convergence. Here, we make special note of a few other issues that should be taken into account. In section 6.1 we present the running times of the search algorithms we presented in this paper and place some important notes. In section 6.2, we give some directions for future work in applying different types of density estimation models.

6.1 A note on the running times

In sections 4 and 5, we have presented algorithms to be used within the IDEA framework. These algorithms take a certain amount of time every iteration. If the search method becomes more involved, the running time that is required every iteration goes up. This is of course hopefully accompanied by better results, but we should be aware of how much time is spent on these algorithms. Recalling that l stands for the length of the coding vector of variables, n stands for the amount of samples in the collection, τ stands for the truncation percentile, n_d stands for the size of the discrete domain set, r stands for the amount of bins in the histogram density model and κ stands for the maximum amount of allowed parents in the PDS for every variable, the running times without regarding any of these as constants, are the following:

Algorithm	**Complexity**
UnivariateDistribution()	$\mathcal{O}(l)$
JointDistribution()	$\mathcal{O}(l^2)$
DiChainSea()	$\mathcal{O}(l^2 n_d^2 + l^2 \tau n)$
DiTreeSea()	$\mathcal{O}(l^2 n_d^2 + l^2 \tau n)$
DiGraphSea()	$\mathcal{O}(l^3 \kappa + l^2 n_d + l^2 \tau n + l \kappa n_d^{\kappa+1} + l \kappa^2 \tau n)$
RePaNoChainSea()	$\mathcal{O}(l^2 \tau n)$
RePaNoTreeSea()	$\mathcal{O}(l^2 \tau n)$
RePaNoGraphSea()	$\mathcal{O}(l^3 \kappa + l^2 \tau n + l \kappa^4)$
ReHiChainSea()	$\mathcal{O}(l^2 r^2 + l^2 \tau n)$
ReHiTreeSea()	$\mathcal{O}(l^2 r^2 + l^2 \tau n)$
ReHiGraphSea()	$\mathcal{O}(l^3 \kappa + l^2 r + l^2 \tau n + l \kappa r^{\kappa+1} + l \kappa^2 \tau n)$
DiEst()	$\mathcal{O}(l n_d^{\kappa+1} + l \kappa \tau n)$
RePaNoEst()	$\mathcal{O}(l \kappa^3 + l^2 + l \tau n + \kappa^2 \tau n)$
ReHiEst()	$\mathcal{O}(l r^{\kappa+1} + l \kappa \tau n)$
DiSam()	$\mathcal{O}(l n_d + l \kappa)$
RePaNoSam()	$\mathcal{O}(l \kappa)$
ReHiSam()	$\mathcal{O}(l r + l \kappa)$

If κ, r, τ and n_d are seen as constants, the running times of the algorithms in that case are easily derived using the table above.

It is common practice to measure the competence of an algorithm by the amount of evaluations required to reach a certain value. However, in the case of using more sophisticated methods for processing the available solution vectors, such as the methods discussed in this paper, such a measure is not entirely fair unless the running time for evaluating a solution dominates all of the running times above. Therefore, it is better to account for the time taken by the algorithm to generate new samples. One way of doing this, is by recording the time in seconds or milliseconds that the algorithm takes instead of the amount of function evaluations. However, in such a case, results obtained on a certain computer are hard to compare with results from the literature because

the computer(s) used in the literature are very likely to differ in many ways from that of your own. A way to compensate for this, is to use a factor based on time in which no additional dimensions are introduced in the resulting expression. If we assume that the running of the algorithm is not hindered by the operating system for swapping memory or something like that, this will be a fair comparison[3]. If we have n_e evaluations, a total running time T in (milli)seconds for the algorithm of and a total running time T_e that the algorithm spent on performing evaluations measured in the same precision, a measure that could be used to compare the results on instead of the amount of evaluations, is the following normalized expression:

$$\overline{n_e} = n_e \frac{T - T_e}{T_e} \tag{27}$$

Another way would be to have a piece of benchmark code that for instance simulates some operations that most evolutionary approaches would do. This would mean some iterations of a lot of integer and real computations. Next, the running time for this benchmark program on the computer to run the tests on should be recorded as T_b. An alternative to using the amount of evaluations (again unless the evaluations clearly dominate the running time) is the normalized running time of the algorithm, which is an index to be named according to the benchmark test:

$$\overline{T} = \frac{T}{T_b} \tag{28}$$

The results that we are still to obtain for the algorithms presented in this paper, will be subjected to such a measure. In general, it should be noted that because of the time spent in finding and using models for covering the interactions between variables in an optimization algorithm might become substantially different from other algorithms, the total time taken by the algorithm should be taken into account in drawing conclusions about the efficiency of the algorithm in terms of running time.

6.2 Density estimation models

In this paper, we have presented algorithms that use density estimation techniques in optimization. Moreover, we have presented algorithms for continuous random variables within two different classes, namely the parametric normal distribution and the non–parametric histogram distribution. There are however more density models that are commonly used in practice. Two of these are the normal kernel distribution and the normal mixture model. In sections 6.2.1 and 6.2.2 we give some insights into using these methods within the IDEA framework.

Another thing that we would like to note in this discussion section, is that we have seen that a problem of using histograms, is their exponential requirements in computation time. However, if $r = 1$, the computation time requirements are no longer exponential. Taking $r = 1$ in the histogram method, we get the uniform distribution, which is somewhat like the Gaussian approach in that we have a very global technique instead of local. Because of the fact that we introduced *ranges* for the histogram distribution, we expect to get convergence. It would be interesting to see how this very simple approach would work on real problems.

6.2.1 Normal kernel distribution

In section 5.2, we used a non–parametric model for density estimation. Next to that density model, there are other density models with interesting properties. If we for instance again take the normal distribution density function, but now fix the value of σ, center one such density function on every sample point and divide the sum of all of these density functions by the amount of sample points, we get the normal kernel distribution. The use of the density functions centered around the sample points is called a kernel method, where the density functions themselves are the kernels. The method of using the Gaussian kernels underlying the normal distribution, has

[3]Not taking into account the quality of the code

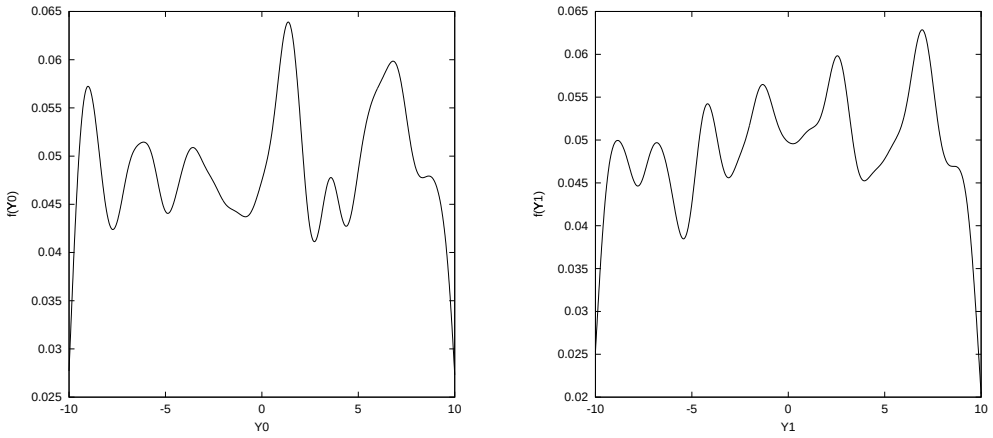

Figure 11: Fitting the univariate data from the uniform dataset with a Gaussian kernel model with $\sigma = 0.5$.

some interesting properties. For instance, by changing the value of σ, we can either get more or less detail in the resulting density estimation. For a very large value of σ for example, there will hardly be any local features in the resulting density estimation. Like with the use of the histogram distribution, we can scale an initially chosen value for σ with the range of the sample points and thereby keep the level of detail during the optimization process constant.

The density function underlying the univariate normal kernel distribution with fixed standard deviation σ defined on N sample points $y^{(S)i}$, $i \in \{0, 1, \ldots, N-1\}$, is defined by:

$$f(y) = \frac{1}{N} \sum_{i=0}^{N-1} \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(y-y^{(S)i})^2}{2\sigma^2}} \tag{29}$$

The multivariate version of the density function in n dimensions with fixed standard deviations σ_i, is defined by:

$$f(y_0, y_1, \ldots, y_{n-1}) = \frac{1}{N} \sum_{i=0}^{N-1} \frac{(2\pi)^{-\frac{n}{2}}}{\prod_{j=0}^{n-1} \sigma_j} e^{-\sum_{j=0}^{n-1} \frac{(y_j - y_j^{(S)i})^2}{2\sigma_j^2}} \tag{30}$$

In figures 11 and 12, the density function taken over the variables univariately is plotted. Note that the localized aspect of the kernel method becomes clear immediately. Even though the samples were drawn from an uniform distribution, there are still peaks in the distribution. It is clear that this method is well suited for representing local features of the distribution. Because of the local aspect, the kernels show a much better representation of the clusters as can be seen in the *joint* probability distribution using Gaussian kernels, which is depicted in figure 13. The joint distributions are not identical at all.

Equation 3 dictates that we require to know the conditional density function for n variables conditionally dependent on one single variable. It is not clear however whether that density function will be convenient to sample from or whether more sophisticated techniques are required to this end. However, it would be very interesting to investigate the use of this density function because of its properties.

6.2.2 Normal mixture distribution

The kernel method for density estimation has some useful properties, amongst which the possibility to select the amount of detail in the resulting density estimation, based on the value for σ. However,

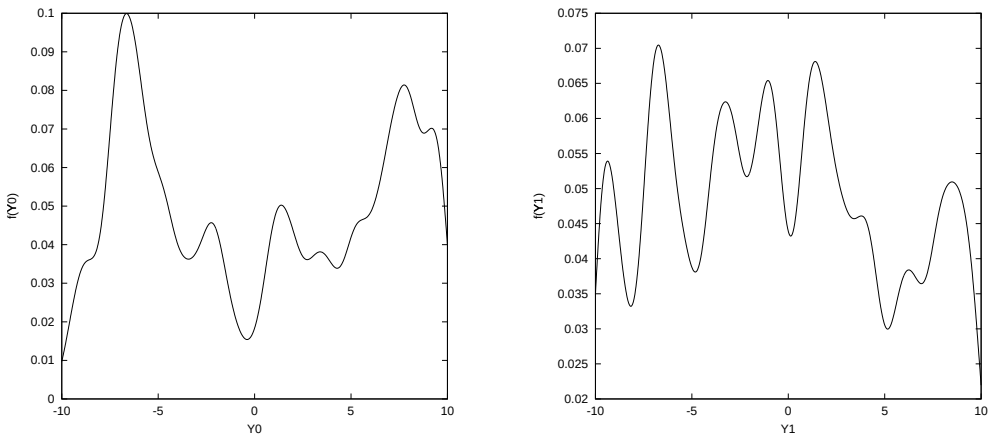

Figure 12: Fitting the univariate data from the clustered dataset with a normal kernel distribution model with $\sigma = 0.5$.

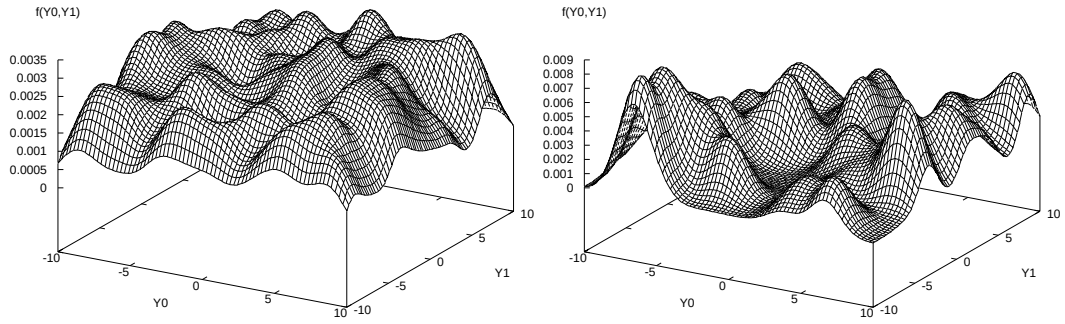

Figure 13: Fitting the joint data from the uniform (left) and clustered (right) datasets with a Gaussian kernel model with $\sigma_0 = \sigma_1 = 1.0$.

if we have n data points, we require n kernels. This might become a problem when the conditional density function is hard to sample from, as it will take a lot of time to evaluate it. As a trade–off between the very global model of the normal distribution as a parametric model on the one side and the normal kernel distribution as a non–parametric model on the other side, we have the normal mixture distribution as a mixture model in between. The intuitive idea is to take more than one kernel, but not to take as many kernels as we have data points or to place them necessarily over the data points. Furthermore, each density function within such a mixture need not be weighted just as heavily as any other. This gives the following density function underlying the univariate normal mixture distribution defined on M models:

$$f(y) = \sum_{i=0}^{M-1} \alpha_i \frac{1}{v_i\sqrt{2\pi}} e^{\frac{-(y-c_i)^2}{2v_i^2}} \tag{31}$$

$$\text{where} \quad \begin{cases} \forall_{i\in\{0,1,\ldots,M-1\}} \langle 0 \leq \alpha_i \leq 1 \rangle \\ \sum_{i=0}^{M-1} \alpha_i = 1 \end{cases}$$

In equation 31, c_i and v_i stand for the center and the variance of the i–th normal distribution density function in the mixture respectively. We use this notation to prevent confusion with the single normal distribution or the normal kernel distribution. To find the centers c_i and the variances v_i, different approaches can be used. One well known method to this end is the EM–algorithm (see for instance [5]).

The multivariate version of the density function in n dimensions with $\boldsymbol{y} = (y_0, y_1, \ldots, y_{n-1})$, is defined by:

$$f(y_0, y_1, \ldots, y_{n-1}) = \sum_{i=0}^{M-1} \alpha_i \frac{(2\pi)^{-\frac{n}{2}}}{(\det \boldsymbol{V}_i)^{\frac{1}{2}}} e^{-\frac{1}{2}(\boldsymbol{y}-\boldsymbol{c}_i)^T \boldsymbol{V}_i^{-1} (\boldsymbol{y}-\boldsymbol{c}_i)} \tag{32}$$

$$\text{where} \quad \begin{cases} \forall_{i\in\{0,1,\ldots,M-1\}} \langle 0 \leq \alpha_i \leq 1 \rangle \\ \sum_{i=0}^{M-1} \alpha_i = 1 \end{cases}$$

In equation 32, we have that $\boldsymbol{c}_i = (c_{i0}, c_{i1}, \ldots, c_{in-1})$ is the vector of means and $\boldsymbol{V}_i = E[(\boldsymbol{y} - \boldsymbol{c}_i)^T (\boldsymbol{y} - \boldsymbol{c}_i)]$ is the nonsingular symmetric covariance matrix. Both of these are defined with respect to index i, which stands for the i–th multivariate normal distribution density function. The EM algorithm can for instance again be applied to find values for these variables.

The normal mixture model has convenient properties in that it is a trade off between the global distribution that is the parametric normal distribution and the local distributions that are the histogram distribution and the normal kernel distribution. The amount of computing time can be regularized because of the limit M on the amount of density functions within the mixture model. Further exploring the possibilities of using the mixture model seems to be very worthwhile.

Still, the same problem arises as with the normal kernel distritubion when computing the conditional density function as it is not clear yet whether that distribution will be straightforward to sample from. This problem within the use of conditional distributions is not present when using univariate distributions as has been the case so far in density estimation based evolutionary algorithms for real spaces [30, 29, 11]. The mixture model was used by Gallagher, Fream and Downs [11] in a univariate distribution, but the full covariance matrix $\boldsymbol{V}_i$ for density function i in the model was reduced to a matrix with only entries on the diagonal.

7 Conclusions

We have presented the algorithmic framework IDEA for modelling density estimation optimization algorithms. These algorithms make use of density estimation techniques to build a probability distribution over the variables that code a problem in order to perform optimization. To this end, a probability density structure must be found and subsequently used in density estimation. For a

set of existing search algorithms, we have presented their application within the IDEA framework, using three different density estimation models. One out of these is used in the case of discrete random variables, whereas the other two are used in the case of continuous random variables. This, in combination with the rationale of modelling solutions below a threshold in a probability distribution, shows that the framework is both general and applicable.

Even though testing of the algorithms on short notice is required, together with different density models that may be implemented in the future, the new approaches seem promising for optimization of problems with continuous variables.

References

[1] T. Bäck and H-P. Schwefel. Evolution strategies i: Variants and their computational implementation. In G. Winter, J. Priaux, M. Galn, and P. Cuesta, editors, *Genetic Algorithms in Engineering and Computer Science, Proceedings of the First Short Course EUROGEN'95*, pages 111–126. Wiley, 1995.

[2] S. Baluja and R. Caruana. Removing the genetics from the standard genetic algorithm. Technical report, Carnegie Mellon University, 1995.

[3] S. Baluja and S. Davies. Using optimal dependency–trees for combinatorial optimization: Learning the structure of the search space. In D.H. Fisher, editor, *Proceedings of the 1997 International Conference on Machine Learning*. Morgan Kauffman publishers, 1997. Also available as Technical Report CMU-CS-97-107.

[4] S. Bandyopadhyay, H. Kargupta, and G. Wang. Revisiting the gemga: Scalable evolutionary optimization through linkage learning. In *Proceedings of the 1998 IEEE International Conference on Evolutionary Computation*, pages 603–608. IEEE Press, 1998. Also available as Technical Report EECS-97-004, Washington State University, Pullman.

[5] C.M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.

[6] J.S. De Bonet, C. Isbell, and P. Viola. Mimic: Finding optima by estimating probability densities. *Advances in Neural Information Processing*, 9, 1996.

[7] P.A.N. Bosman and D. Thierens. Linkage information processing in distribution estimation algorithms. In W. Banzhaf, J. Daida, A.E. Eiben, M.H. Garzon, V. Honavar, M. Jakiela, and R.E. Smith, editors, *Proceedings of the GECCO–1999 Genetic and Evolutionary Computation Conference*, pages 60–67. Morgan Kaufmann Publishers, 1999.

[8] T.M. Cover and J.A. Thomas. *Elements of Information Theory*. John Wiley & Sons Inc., 1991.

[9] K. Deb and D.E. Goldberg. Sufficient conditions for deceptive and easy binary functions. *Annals of Mathematics and Artificial Intelligence*, 10:385–408, 1994.

[10] J. Edmonds. Optimum branchings. *J. Res. Nat. Bur. Standards*, 71B:233–240, 1967. Reprinted in Math. of the Decision Sciences, *Amer. Math. Soc. Lectures in Appl. Math.*, 11:335–345, 1968.

[11] M. Gallagher, M. Fream, and T. Downs. Real–valued evolutionary optimization using a flexible probability density estimator. In W. Banzhaf, J. Daida, A.E. Eiben, M.H. Garzon, V. Honavar, M. Jakiela, and R.E. Smith, editors, *Proceedings of the GECCO–1999 Genetic and Evolutionary Computation Conference*, pages 840–846. Morgan Kaufmann Publishers, 1999.

[12] D.E. Goldberg. *Genetic Algorithms In Search, Optimization, And Machine Learning*. Addison–Wesley, Reading, 1989.

[13] D.E. Goldberg, K. Deb, H. Kargupta, and G. Harik. Rapid, accurate optimization of difficult problems using fast messy genetic algorithms. In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 56–64. Morgan Kauffman publishers, 1993. Also available as IlliGAL Report 93004.

[14] D.E. Goldberg, B. Korb, and K. Deb. Messy genetic algorithms: Motivation, analysis and first results. *Complex Systems*, 10:385–408, 1989.

[15] G. Harik. Linkage learning via probabilistic modeling in the ecga. IlliGAL Technical Report 99010. ftp://ftp-illigal.ge.uiuc.edu/pub/papers/IlliGALs/99010.ps.Z, 1999.

[16] G. Harik, F. Lobo, and D.E. Goldberg. The compact genetic algorithm. In *Proceedings of the 1998 IEEE International Conference on Evolutionary Computation*, pages 523–528. IEEE Press, 1998.

[17] D. Heckerman, D. Geiger, and D.M. Chickering. Learning bayesian networks: The combination of knowledge and statistical data. In R. Lopez de Mantaras and D. Poole, editors, *Proceedings of the 10th Conference on Uncertainty in Artificial Intelligence*, pages 293–301. Morgan Kauffman publishers, 1994. Also available as Technical Report MSR-TR-94-09.

[18] J.H. Holland. *Adaptation in Natural and Artificial Systems.* Ann Arbor: University of Michigan Press, 1975.

[19] H. Kargupta. The gene expression messy genetic algorithm. In *Proceedings of the 1996 IEEE International Conference on Evolutionary Computation*, pages 631–636. IEEE Press, 1996.

[20] R.M. Karp. A simple derivation of edmonds' algorithm for optimum branchings. *Networks*, 1:265–272, 1971.

[21] F.G. Lobo, K. Deb, D.E. Goldberg, G.R. Harik, and L. Wang. Compressed introns in a linkage learning genetic algorithm. In W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D.B. Fogel, M.H. Garzon, D.E. Goldberg, H. Iba, and R. Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 551–558. Morgan Kauffman publishers, 1998. Also available as IlliGAL Report 97010.

[22] H. Mühlenbein, T. Mahnig, and O. Rodriguez. Schemata, distributions and graphical models in evolutionary optimization. *Journal of Heuristics*, 5:215–247, 1999.

[23] H. Mühlenbein and G. Paaß. From recombination of genes to the estimation of distributions i. binary parameters. In A.E. Eiben, T. Bäck, M. Schoenauer, and H.-P. Schwefel, editors, *Parallel Problem Solving from Nature – PPSN V*, pages 178–187. Springer, 1998.

[24] M. Pelikan, D.E. Goldberg, and E. Cantú-Paz. Boa: The bayesian optimization algorithm. In W. Banzhaf, J. Daida, A.E. Eiben, M.H. Garzon, V. Honavar, M. Jakiela, and R.E. Smith, editors, *Proceedings of the GECCO–1999 Genetic and Evolutionary Computation Conference*, pages 525–532. Morgan Kaufmann Publishers, 1999.

[25] M. Pelikan, D.E. Goldberg, and F. Lobo. A survey of optimization by building and using probabilistic models. IlliGAL Technical Report 99018. ftp://ftp-illigal.ge.uiuc.edu/pub/papers/IlliGALs/99018.ps.Z, 1999.

[26] M. Pelikan and H. Mühlenbein. The bivariate marginal distribution algorithm. In R. Roy, T. Furuhashi, K. Chawdry, and K. Pravir, editors, *Advances in Soft Computing – Engineering Design and Manufacturing.* Springer–Verlag, 1999.

[27] A. Ravidran, D.T. Philips, and J.J. Solberg. *Operations Research Principles and Practice.* John Wiley & Sons Inc., 1987.

[28] D.W. Scott. *Multivariate Density Estimation.* John Wiley & Sons Inc., 1992.

[29] M. Sebag and A. Ducoulombier. Extending population–based incremental learning to continuous search spaces. In A.E. Eiben, T. Bäck, M. Schoenauer, and H.-P. Schwefel, editors, *Parallel Problem Solving from Nature – PPSN V*, pages 418–427. Springer, 1998.

[30] I. Servet, L. Trave-Massuyes, and D. Stern. Telephone network traffic overloading diagnosis and evolutionary computation technique. In J.K. Hao, E. Lutton, E. Ronald, M. Schoenauer, and D. Snyers, editors, *Proceedings of Artificial Evolution '97*, pages 137–144. Springer Verlag, LNCS 1363, 1997.

[31] C.E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, 623–656, 1948.

[32] R. Tarjan. Finding optimal branchings. *Networks*, 7:25–35, 1977.

[33] D. Thierens and D.E. Goldberg. Mixing in genetic algorithms. In S. Forrest, editor, *Proceedings of the fifth conference on Genetic Algorithms*, pages 38–45. Morgan Kaufmann, 1993.

[34] C.H.M. van Kemenade. Building block filtering and mixing. In *Proceedings of the 1998 IEEE International Conference on Evolutionary Computation.* IEEE Press, 1998.

A PDS graph search metrics

Given is a PDS:

$$(\pi, \boldsymbol{\omega}) = (\pi, (\omega_0, \omega_1, \ldots, \omega_{l-1})) \tag{33}$$

$$\text{such that } \forall_{i \in \{0,1,\ldots,l-1\}} \langle \omega_i \in \{0, 1, \ldots, l-1\} \wedge \forall_{k \in \{0,1,\ldots,l-1\}-\{i\}} \langle \omega_i \neq \omega_k \rangle \rangle$$

$$\forall_{i \in \{0,1,\ldots,l-1\}} \langle \forall_{k \in \pi(\omega_i)} \langle k \in \{\omega_{i+1}, \omega_{i+2}, \ldots, \omega_{l-1}\} \rangle \rangle$$

The Bayesian Dirichlet metric [17] in the discrete case, which we denote $\psi((\pi, \boldsymbol{\omega}))$, is defined by:

$$\psi((\pi, \boldsymbol{\omega})) = p((\pi, \boldsymbol{\omega})|\xi) \prod_{i=0}^{l-1} \prod_{k_1=0}^{n_d-1} \prod_{k_2=0}^{n_d-1} \cdots \prod_{k_{|\pi(\omega_i)|}=0}^{n_d-1} \tag{34}$$

$$\left(\frac{m'(\pi(\omega_i), \boldsymbol{\varrho}^i)!}{(m'(\pi(\omega_i), \boldsymbol{\varrho}^i) + m(\pi(\omega_i), \boldsymbol{\varrho}^i))!} \prod_{k_0=0}^{n_d-1} \frac{(m'(\omega_i \sqcup \pi(\omega_i), k_0 \sqcup \boldsymbol{\varrho}^i) + m(\omega_i \sqcup \pi(\omega_i), k_0 \sqcup \boldsymbol{\varrho}^i))!}{m'(\omega_i \sqcup \pi(\omega_i), k_0 \sqcup \boldsymbol{\varrho}^i)!} \right)$$

$$\text{where} \begin{cases} \boldsymbol{\varrho}^i & = (k_1, k_2, \ldots, k_{|\pi(\omega_i)|}) = (\varrho_0^i, \varrho_1^i, \ldots, \varrho_{|\boldsymbol{\varrho}^i|}^i) \\ m(\boldsymbol{\nu}, \boldsymbol{\lambda}) & = \sum_{q=0}^{\lfloor \tau n \rfloor} \begin{cases} 1 & \text{if } \forall_{r \in \{0,1,\ldots,|\boldsymbol{\lambda}|-1\}} \langle X_{\nu_r}^{(S)q} = \lambda_r \rangle \\ 0 & \text{otherwise} \end{cases} \end{cases}$$

In the above, we have used $\sqcup$ for the notation of appending an element at the head of a vector: $x_0 \sqcup (x_1, x_2, \ldots, x_n) = (x_0, x_1, \ldots, x_n)$. The expression $p((\pi, \boldsymbol{\omega})|\xi)$ stands for the prior probability of network $(\pi, \boldsymbol{\omega})$. The function $m'(\boldsymbol{\nu}, \boldsymbol{\lambda})$ stands for prior information on $m(\boldsymbol{\nu}, \boldsymbol{\lambda})$. In the BOA implementation [24], the prior information is disregarded:

$$\begin{cases} p((\pi, \boldsymbol{\omega})|\xi) & = 1 \\ m'(\boldsymbol{\nu}, \boldsymbol{\lambda}) & = 1 \end{cases}$$

Disregarding the prior information results in the so called K2 metric. For numerical purposes, the actual metric that we work with is a logarithm over the $K2$ metric, wich we denote in the case of discrete random variables $\tilde{\psi}((\pi, \boldsymbol{\omega}))$, given the same *where* clause as in equation 34:

$$\tilde{\psi}((\pi, \boldsymbol{\omega})) = \tag{35}$$

$$\ln \left(\prod_{i=0}^{l-1} \prod_{k_1=0}^{n_d-1} \prod_{k_2=0}^{n_d-1} \cdots \prod_{k_{|\pi(\omega_i)|}=0}^{n_d-1} \left(\frac{1}{(1 + m(\pi(\omega_i), \boldsymbol{\varrho}^i))!} \prod_{k_0=0}^{n_d-1} (1 + m(\omega_i \sqcup \pi(\omega_i), k_0 \sqcup \boldsymbol{\varrho}^i))! \right) \right) =$$

$$\sum_{i=0}^{l-1} \sum_{k_1=0}^{n_d-1} \sum_{k_2=0}^{n_d-1} \cdots \sum_{k_{|\pi(\omega_i)|}=0}^{n_d-1} \left(- \sum_{j=1}^{1+m(\pi(\omega_i), \boldsymbol{\varrho}^i)} \ln(j) + \sum_{k_0=0}^{n_d-1} \sum_{j=1}^{1+m(\omega_i \sqcup \pi(\omega_i), k_0 \sqcup \boldsymbol{\varrho}^i)} \ln(j) \right)$$

The MIMIC algorithm [6] uses a metric based upon the Kullback–Leibler divergence distance metric to the chain probability distribution structure. The same is done in the optimal dependency trees approach [3], but with respect to the tree probability distribution structure. We find that if we write $\mathfrak{p}(\boldsymbol{\nu}, \boldsymbol{\lambda}) = \hat{p}_{\nu_0, \nu_1, \ldots, \nu_{|\boldsymbol{\lambda}|-1}}(\lambda_0, \lambda_1, \ldots, \lambda_{|\boldsymbol{\lambda}|-1})$, we have in accordance with equation 18 that $m(\boldsymbol{\nu}, \boldsymbol{\lambda}) = \lfloor \tau n \rfloor \mathfrak{p}(\boldsymbol{\nu}, \boldsymbol{\lambda})$. If we now apply the Kullback–Leibler divergence to the general model from equation 7, we can derive another metric. Staying within the case of discrete random variables, we may write the negative of this metric, which we denote $\breve{\psi}((\pi, \boldsymbol{\omega}))$, as follows[4] (still keeping the same *where* clause as in equation 34):

[4]The Kullback–Leibler divergence is thus equal to $-\breve{\psi}((\pi, \boldsymbol{\omega}))$.

$$\breve{\psi}((\pi,\boldsymbol{\omega})) = -\sum_{i=0}^{l-1} \hat{h}(X_{\omega_i}|X_{\pi(\omega_i)_0}, X_{\pi(\omega_i)_1}, \ldots, X_{\pi(\omega_i)_{|\pi(\omega_i)|-1}}) \;=\; (36)$$

$$-\sum_{i=0}^{l-1}\left(-\sum_{k_0=0}^{n_d-1}\sum_{k_1=0}^{n_d-1}\cdots\sum_{k_{|\pi(\omega_i)|}=0}^{n_d-1} \mathfrak{p}(\omega_i \sqcup \pi(\omega_i), k_0 \sqcup \boldsymbol{\varrho}^i)\ln(\mathfrak{p}(\omega_i \sqcup \pi(\omega_i), k_0 \sqcup \boldsymbol{\varrho}^i))\right.$$

$$\left. -\left(-\sum_{k_1=0}^{n_d-1}\sum_{k_2=0}^{n_d-1}\cdots\sum_{k_{|\pi(\omega_i)|}=0}^{n_d-1} \mathfrak{p}(\pi(\omega_i), \boldsymbol{\varrho}^i)\ln(\mathfrak{p}(\pi(\omega_i), \boldsymbol{\varrho}^i))\right)\right) \;=$$

$$\sum_{i=0}^{l-1}\sum_{k_1=0}^{n_d-1}\sum_{k_2=0}^{n_d-1}\cdots\sum_{k_{|\pi(\omega_i)|}=0}^{n_d-1}$$

$$\left(-\mathfrak{p}(\pi(\omega_i), \boldsymbol{\varrho}^i)\ln(\mathfrak{p}(\pi(\omega_i), \boldsymbol{\varrho}^i)) + \sum_{k_0=0}^{n_d-1} \mathfrak{p}(\omega_i \sqcup \pi(\omega_i), k_0 \sqcup \boldsymbol{\varrho}^i)\ln(\mathfrak{p}(\omega_i \sqcup \pi(\omega_i), k_0 \sqcup \boldsymbol{\varrho}^i))\right)$$

The goal using the $K2$ metric is to *maximize* $\psi((\pi,\boldsymbol{\omega}))$. Using the Kullback–Leibler divergence, according to the above, we can also propose to maximize $\breve{\psi}((\pi,\boldsymbol{\omega}))$ or in other words to minimize $-\breve{\psi}((\pi,\boldsymbol{\omega}))$. Introducing the notation $\bar{\psi}((\pi,\boldsymbol{\omega}))$, we can state this minimization problem as follows:

$$\min_{(\pi,\boldsymbol{\omega})}\left\{\bar{\psi}((\pi,\boldsymbol{\omega})) = -\breve{\psi}((\pi,\boldsymbol{\omega})) = \sum_{i=0}^{l-1} \hat{h}(X_{\omega_i}|X_{\pi(\omega_i)_0}, X_{\pi(\omega_i)_1}, \ldots, X_{\pi(\omega_i)_{|\pi(\omega_i)|-1}})\right\} \quad (37)$$

As the entropy measure has certain interesting and useful properties, we choose to work with the metric $\bar{\psi}((\pi,\boldsymbol{\omega}))$. However, there seems to be a certain correspondence between metrics $\tilde{\psi}((\pi,\boldsymbol{\omega}))$ and $\bar{\psi}((\pi,\boldsymbol{\omega}))$. In a direct sense, because we have written out the definition of conditional entropy for metric $\breve{\psi}((\pi,\boldsymbol{\omega}))$, it is clear that a single substitution transforms $\tilde{\psi}((\pi,\boldsymbol{\omega}))$ into $\breve{\psi}((\pi,\boldsymbol{\omega}))$. This substition is the following:

$$\sum_{j=1}^{1+m(\boldsymbol{\nu},\boldsymbol{\lambda})} \ln(j) \quad \rightsquigarrow \quad \frac{m(\boldsymbol{\nu},\boldsymbol{\lambda})}{\lfloor \tau n \rfloor}\ln\left(\frac{m(\boldsymbol{\nu},\boldsymbol{\lambda})}{\lfloor \tau n \rfloor}\right) \;=\; \mathfrak{p}(\boldsymbol{\nu},\boldsymbol{\lambda})\ln(\mathfrak{p}(\boldsymbol{\nu},\boldsymbol{\lambda}))$$

We have introduced this substitution by writing $\rightsquigarrow$ because it is non–monotonic. However, in the composition with the positive sum over all instances of the considered node, the two metrics might distinguish between two networks in the same manner. If we observe closely, both of the lastly bracketed expressions from $\tilde{\psi}((\pi,\boldsymbol{\omega}))$ and $\breve{\psi}((\pi,\boldsymbol{\omega}))$ are a way of expressing conditionality of a single variable on a set of parent variables. Whereas in $\tilde{\psi}((\pi,\boldsymbol{\omega}))$ it is a factorial of instance numbers, in $\breve{\psi}((\pi,\boldsymbol{\omega}))$ it is close to the logarithm of the empiric probabilities. We will not attempt to give an analysis of the correspondence between the two metrics. However, to give some insight into the behaviour of the metrics, we have portrayed some experiments. Consider the following trap functions:

$$f(X) = \begin{cases} -\frac{1-d}{l-1}o(X) + 1 - d & \text{if } o(X) < l \\ 1 & \text{if } o(X) = l \end{cases}$$

Here $o(X)$ stands for the amount of ones in vector X of length l. The maximum value of 1 is achieved for a vector with l ones and the suboptimum of $1-d$ is found for a vector with l zeros. When d goes to 0, the problem becomes fully deceptive for some value of d. This means that all schemata of an order smaller than l lead to the suboptimum (deceptive attractor) [9]. We have set $l = 5$ and have taken only one such subvector as the problem. We then fixed the ordering of the variables to $\omega = (0,1,2,3,4)$. Because of the fact that the problem is not dependent on the ordering of the variables in the vector, this will not restrict the results to a special case. Using this ordering, we enumerated all of the 1024 possible combinations of parents and computed both

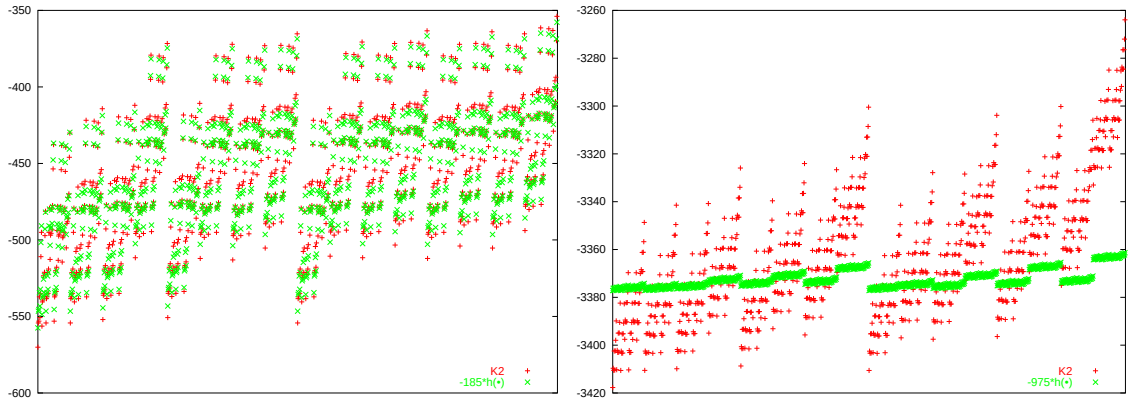

Figure 14: Metrics for $\tau = 0.2$ (left) and $\tau = 1.0$ (right), $d = 0.2$.

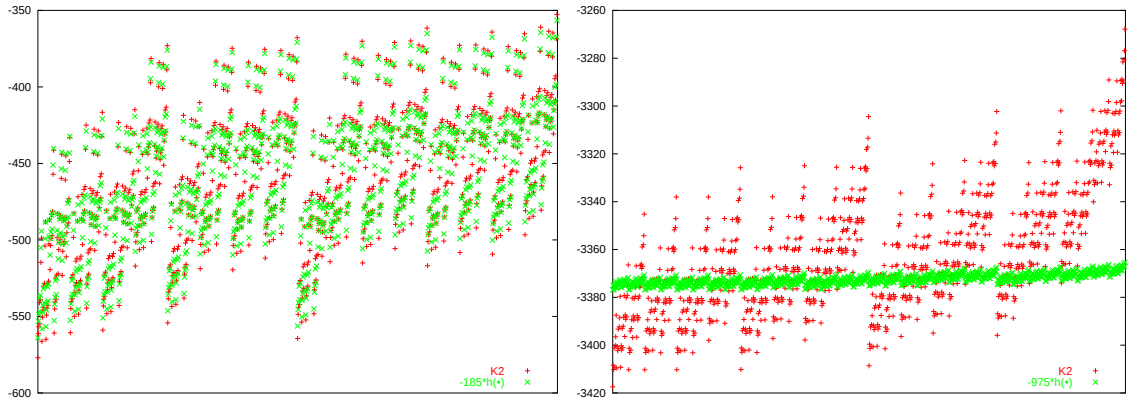

Figure 15: Metrics for $\tau = 0.2$ (left) and $\tau = 1.0$ (right), $d = 0.8$.

the $K2$ metric as well as the negative entropy metric $\breve{\psi}((\pi, \boldsymbol{\omega}))$. The computation of these metrics is done using a set of samples. To simulate the IDEA approach, we have taken 1000 samples and evaluated them according to function $f(\cdot)$. We then selected the top 1000τ samples with respect to this measure and computed the metrics based upon this selection. To show the results for a few variants of the problem, we have chosen $\tau \in \{0.2, 1.0\}$ and $d \in \{0.2, 0.8\}$. The results are shown in figures 14 and 15.

B Edmond's algorithm for optimum branchings

Given is the following:

$$D = \begin{cases} V = \{0, 1, \ldots, n-1\} \\ A \subseteq V \times V \\ G = (V, A) \\ s : A \to V \quad \text{such that } s((i,j)) = i \\ t : A \to V \quad \text{such that } t((i,j)) = j \\ c : A \to \mathbb{R}^+ \end{cases} \tag{38}$$

The functions $s(\cdot)$ and $t(\cdot)$ respectively return the source and target of an arc, whereas function $c(\cdot)$ returns the *weight* of an arc in the graph. We seek to find a *maximum* weighted branching $B \subseteq A$. The weight $C(\cdot)$ of a branching B is the sum of the weights of the arcs in the branching. A set $B \subseteq A$ is called a branching if B does not contain a *cycle* and no two arcs in B have the same target. We may now formulate problem $\mathcal{P}(\cdot)$ as a function of the data D as follows:

$$\mathcal{P}(D) : \text{maximize } C(B) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} c(i,j) x_{ij} \tag{39}$$

$$\text{such that } \begin{cases} B \subseteq A \\ \forall_{(i,j)\in B} \left\langle x_{ij} = \begin{cases} 1 & \text{if } (i,j) \in B \\ 0 & \text{otherwise} \end{cases} \right\rangle \\ \neg\exists_{\boldsymbol{a}=(a_0,a_1,\ldots,a_{|\boldsymbol{a}|-1})\in B^{|\boldsymbol{a}|}} \langle \forall_{i\in\{(0,1,\ldots,|\boldsymbol{a}|-2)\}} \langle t(a_i) = s(a_{i+1}) \rangle \wedge t(a_{|\boldsymbol{a}|-1}) = s(a_0)) \rangle \\ \forall_{(a,a')\in B\times B} \langle a \neq a' \rightarrow t(a) \neq t(a') \rangle \end{cases}$$

The first step of the algorithm requires the notion of a *critical graph*. To this end, we introduce predicates $\mathcal{C}_{\mathcal{A}}(\cdot)$, $\mathcal{C}_{\mathcal{G}}(\cdot)$ and $\mathcal{C}'_{\mathcal{G}}(\cdot)$, which stand for the definition of a *critical arc*, a *critical graph* and a help predicate for the definition of a critical graph respectively. The following three definitions hold given the data D from equation 38 and given that we write a critical graph H for graph $G = (V, A)$ as $H = (V, A^H)$:

$$\mathcal{C}_{\mathcal{A}}(a) \Leftrightarrow c(a) > 0 \wedge \forall_{a'\in A} \langle t(a') = t(a) \rightarrow c(a') \leq c(a) \rangle \tag{40}$$

$$\mathcal{C}'_{\mathcal{G}}(H) \Leftrightarrow A^H \subseteq A \wedge \forall_{a_H\in A^H} \langle \mathcal{C}_{\mathcal{A}}(a_H) \rangle \wedge \forall_{(a_H,a'_H)\in A^H\times A^H} \langle a_H \neq a'_H \rightarrow t(a_H) \neq t(a'_H) \rangle \tag{41}$$

$$\mathcal{C}_{\mathcal{G}}(H) \Leftrightarrow \mathcal{C}'_{\mathcal{G}}(H) \wedge \forall_{H'=(V,A^{H'})} \langle \mathcal{C}'_{\mathcal{G}}(H') \rightarrow |A^{H'}| \leq |A^H| \rangle \tag{42}$$

In the algorithm, we have to be able to construct new data $\overline{D}$ for problem $\overline{\mathcal{P}}$ from data D if the critical graph has cycles. This is done by contracting the nodes in each cycle to a single new vertex. Assuming that the critical graph H contains k cycles $C_0, C_1, \ldots, C_{k-1}$ such that $\forall_{i\in\{0,1,\ldots,k-1\}} \langle C_i \subseteq A \rangle$, these cycles concern disjoint sets of vertices: $\forall_{i\in\{0,1,\ldots,k-1\}} \langle V_i = \{t(a) | a \in C_i\} \rangle$. We may now define the new data $\overline{D}(\cdot)$ as a function of a critical graph H:

$$\overline{D}(H) = \begin{cases} \overline{V} & = \hat{V} \cup \{z_0, z_1, \ldots, z_{k-1}\} \quad \text{such that } z_i \in \mathbb{N} \text{ and unused} \\ \overline{A} & = \{a \mid a \in A \wedge (s(a) \in \hat{V} \vee t(a) \in \hat{V})\} \\ \overline{G} & = (\overline{V}, \overline{A}) \\ \overline{s}(a) & = \begin{cases} s(a) & \text{if } s(a) \in \hat{V} \\ z_i & \text{if } s(a) \in V_i \end{cases} \\ \overline{t}(a) & = \begin{cases} t(a) & \text{if } t(a) \in \hat{V} \\ z_i & \text{if } t(a) \in V_i \end{cases} \\ \overline{c}(a) & = \begin{cases} c(a) & \text{if } t(a) \in \hat{V} \\ c(a) - c(\tilde{a}) + c(a_i^{\min}) & \text{if } t(a) \in V_i \end{cases} \end{cases} \tag{43}$$

$$\text{where } \begin{cases} \hat{V} = V - \bigcup_{i=0}^{k-1} V_i \\ \forall_{i\in\{0,1,\ldots,k-1\}} \langle a_i^{\min} = arg\ \min_{a\in C_i} \{c(a)\} \rangle \\ \forall_{a_H\in A^H} \langle f(t(a_H)) = a_H \rangle \\ \forall_{a\in\overline{A}} \langle \forall_{i\in\{0,1,\ldots,k-1\}} \langle t(a) \in V_i \rightarrow \tilde{a} = f(t(a)) \rangle \rangle \end{cases}$$

Karp [20] has shown that the following one–to–one correspondence holds between an optimum branching B in $\mathcal{P}(D)$ and an optimum branching $\overline{B}$ in problem $\mathcal{P}(\overline{D}(\cdot))$:

$$B = \overline{B} \cup \left(\bigcup_{i=0}^{k-1} (C_i - a_i^{\text{break}}) \right) \tag{44}$$

$$\text{where } a_i^{\text{break}} = \begin{cases} \tilde{a} & \text{if } \exists_{\overline{a} \in \overline{B}} \langle \overline{t}(\overline{a}) = z_i \rangle \\ a_i^{\min} & \text{otherwise} \end{cases}$$

Edmond's algorithm [10] computes a critical graph H, which is a graph H such that $\mathcal{C}_{\mathcal{G}}(H)$ holds. It has been shown by Karp [20] that if H is acyclic, it is an optimum branching. Otherwise, the algorithm recursively computes an optimum branching $\overline{B}$ using data $\overline{D}(H)$ in problem $\mathcal{P}(\overline{D}(H))$. By using the correspondence from equation 44, the optimum branching B for the original problem can be determined.

C Multivariate conditional normal distribution

The univariate density function for the normal distribution is defined by:

$$f(y) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(y-\mu)^2}{2\sigma^2}} = g(y, \mu, \sigma) \tag{45}$$

The multivariate density function for the normal distribution in d dimensions with $\boldsymbol{y} = (y_0, y_1, \ldots, y_{d-1})$ is defined by:

$$f(\boldsymbol{y}) = \frac{(2\pi)^{-\frac{d}{2}}}{(\det \boldsymbol{\Sigma})^{\frac{1}{2}}} e^{-\frac{1}{2}(\boldsymbol{y}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{y}-\boldsymbol{\mu})} \tag{46}$$

In equation 46, we have that $\boldsymbol{\mu} = (\mu_0, \mu_1, \ldots, \mu_{d-1})$ is the vector of means and $\boldsymbol{\Sigma} = E[(\boldsymbol{y} - \boldsymbol{\mu})^T (\boldsymbol{y} - \boldsymbol{\mu})]$ is the nonsingular covariance matrix, which is symmetric. It is common practice to define σ_{ij} to be the entry in row i and column j in matrix $\boldsymbol{\Sigma}$, meaning $\sigma_{ij} = \boldsymbol{\Sigma}(i, j)$. Note that it follows from the definition of $\boldsymbol{\Sigma}$ that $\sigma_{ii} = \sigma_i^2$. We similarly define $\sigma'_{ij} = (\boldsymbol{\Sigma}^{-1})(i, j)$. We may now rewrite matrix equation 46 as an equation involving only element variables (with the exception of the determinant):

$$f(y_0, y_1, \ldots, y_{d-1}) = \frac{(2\pi)^{-\frac{d}{2}}}{(\det \boldsymbol{\Sigma})^{\frac{1}{2}}} e^{-\frac{1}{2}(\sum_{j=0}^{d-1} (\sum_{i=0}^{d-1} (y_i - \mu_i)\sigma'_{ij})(y_j - \mu_j))} \tag{47}$$

We derive the density function for the conditional normal distribution for a single variable y_0 that is dependent on c other variables $y_1, y_2, \ldots, y_c$. In the case of sampling, all μ_i and σ'_{ij} parameters are assumed to be known as well as are instance values for $y_1, y_2, \ldots, y_c$. This leaves us only with a single parameter y_0 and the remainder as constants, making the resulting pdf one–dimensional. In a closed form, this means we are searching to find $\tilde{\mu}_0$ and $\tilde{\sigma}_0$ so that the resulting density function can be expressed as a one–dimensional normalized Gaussian:

$$f(y_0 | y_1, y_2, \ldots, y_c) = \frac{1}{\tilde{\sigma}_0\sqrt{2\pi}} e^{\frac{-(y-\tilde{\mu}_0)^2}{2\tilde{\sigma}_0^2}} = g(y, \tilde{\mu}_0, \tilde{\sigma}_0) \tag{48}$$

First, we note that we acquire from probability theory that

$$f(y_0 | y_1, y_2, \ldots, y_c) = \frac{f(y_0, y_1, y_2, \ldots, y_c)}{f(y_1, y_2, \ldots, y_c)} \tag{49}$$

Combining equation 49 with equation 46, we note that we get two covariance matrices of sizes $(c+1) \times (c+1)$ and $c \times c$ for the the multivariate expressions in the fraction of equation 49. We shall write the elements from the matrix containing $(c+1) \times (c+1)$ elements as $\sigma_{ij}^{(1)}$ and the elements from the matrix containing $c \times c$ elements as $\sigma_{ij}^{(2)}$. Actually, we require to denote the elements from the inverse matrices, so we shall write these as σ'_{ij} and σ''_{ij} respectively. Note that from the definition of $\boldsymbol{\Sigma}$ it follows that $\sigma_{ij}^{(1)} = \sigma_{ji}^{(1)}$ and even that $\sigma_{ij}^{(1)} = \sigma_{(i-1)(j-1)}^{(2)}$ when $i > 1 \wedge j > 1$. However, in general we cannot state that $\sigma'_{ij} = \sigma''_{(i-1)(j-1)}$. What we *may* state in general, is that because $\boldsymbol{\Sigma}$ is by definition symmetric, so is $\boldsymbol{\Sigma}^{-1}$. This means that $\sigma'_{ij} = \sigma'_{ji}$ and that $\sigma''_{ij} = \sigma''_{ji}$ It is *very* important in the following to note that σ''_{ij} concerns variables y_{i+1} and

y_{j+1} in terms of covariance. The mismatch in variable index is because of matrix indexing. We can now derive the required expressions $\tilde{\mu}_0$ and $\tilde{\sigma}_0$:

$$f(y_0|y_1,y_2,\ldots,y_c) = \frac{\frac{(2\pi)^{-\frac{c+1}{2}}}{(\det \mathbf{\Sigma}^{(1)})^{\frac{1}{2}}} e^{-\frac{1}{2}(\sum_{j=0}^{c}(\sum_{i=0}^{c}(y_i-\mu_i)\sigma'_{ij})(y_j-\mu_j))}}{\frac{(2\pi)^{-\frac{c}{2}}}{(\det \mathbf{\Sigma}^{(2)})^{\frac{1}{2}}} e^{-\frac{1}{2}(\sum_{j=1}^{c}(\sum_{i=1}^{c}(y_i-\mu_i)\sigma''_{(i-1)(j-1)})(y_j-\mu_j))}} = ae^b$$

From the form ae^b it must become apparent what the the required expressions $\tilde{\mu}_0$ and $\tilde{\sigma}_0$ are. It is clear to see from the equation in definition 45 that from computing a we will be able to find $\tilde{\sigma}_0$ and that from computing b we will be able to find both $\tilde{\mu}_0$ and $\tilde{\sigma}_0$. Even though only deriving b will be enough, we will derive both and see whether we arrive at matching results for $\tilde{\sigma}_0$. First, we determine a to get $\tilde{\sigma}_0$:

$$a = \frac{(2\pi)^{-\frac{c+1}{2}}}{(\det \mathbf{\Sigma}^{(1)})^{\frac{1}{2}}} \frac{(\det \mathbf{\Sigma}^{(2)})^{\frac{1}{2}}}{(2\pi)^{-\frac{c}{2}}} = \frac{1}{\sqrt{\frac{\det \mathbf{\Sigma}^{(1)}}{\det \mathbf{\Sigma}^{(2)}}}\sqrt{2\pi}} \Rightarrow \tilde{\sigma}_0 = \sqrt{\frac{\det \mathbf{\Sigma}^{(1)}}{\det \mathbf{\Sigma}^{(2)}}}$$

Second, we determine b to get both $\tilde{\mu}_0$ and $\tilde{\sigma}_0$:

$$\begin{aligned}
b = \; & \tfrac{1}{2}(-(\textstyle\sum_{j=0}^{c}(\sum_{i=0}^{c}(y_i-\mu_i)\sigma'_{ij})(y_j-\mu_j)) \\
& +(\textstyle\sum_{j=1}^{c}(\sum_{i=1}^{c}(y_i-\mu_i)\sigma''_{(i-1)(j-1)})(y_j-\mu_j))) \\
= \; & \tfrac{1}{2}(-(\textstyle\sum_{j=1}^{c}(\sum_{i=1}^{c}(y_i-\mu_i)\sigma'_{ij})(y_i-\mu_i)) \\
& -(y_0-\mu_0)\textstyle\sum_{i=1}^{c}(y_i-\mu_i)\sigma'_{i0} \\
& -(y_0-\mu_0)^2\sigma'_{00} \\
& -(y_0-\mu_0)\textstyle\sum_{j=1}^{c}(y_j-\mu_j)\sigma'_{0j} \\
& +(\textstyle\sum_{j=1}^{c}(\sum_{i=1}^{c}(y_i-\mu_i)\sigma''_{(i-1)(j-1)})(y_j-\mu_j))) \\
= \; & \tfrac{1}{2}(-(\textstyle\sum_{j=1}^{c}(\sum_{i=1}^{c}(y_i-\mu_i)\sigma'_{ij})(y_i-\mu_i)) \\
& -2y_0\textstyle\sum_{i=1}^{c}(y_i-\mu_i)\sigma'_{i0} \\
& +2\mu_0\textstyle\sum_{i=1}^{c}(y_i-\mu_i)\sigma'_{i0} \\
& -y_0^2\sigma'_{00} \\
& +2y_0\mu_0\sigma'_{00} \\
& -\mu_0^2\sigma'_{00} \\
& +(\textstyle\sum_{j=1}^{c}(\sum_{i=1}^{c}(y_i-\mu_i)\sigma''_{(i-1)(j-1)})(y_j-\mu_j))) \\
=_{\text{def}} & \left\{\begin{array}{l}
a_0 = -(\sum_{j=1}^{c}(\sum_{i=1}^{c}(y_i-\mu_i)\sigma'_{ij})(y_i-\mu_i)) \\
a_1 = -2\sum_{i=1}^{c}(y_i-\mu_i)\sigma'_{i0} \\
a_2 = 2\mu_0\sum_{i=1}^{c}(y_i-\mu_i)\sigma'_{i0} \\
a_3 = -\sigma'_{00} \\
a_4 = 2\mu_0\sigma'_{00} \\
a_5 = -\mu_0^2\sigma'_{00} \\
a_6 = (\sum_{j=1}^{c}(\sum_{i=1}^{c}(y_i-\mu_i)\sigma''_{(i-1)(j-1)})(y_j-\mu_j)) \\
\alpha = -a_3 \\
\beta = -a_1-a_4 \\
\gamma = -a_0-a_2-a_5-a_6
\end{array}\right\}
\end{aligned}$$

$$\frac{1}{2}(a_0+a_1y_0+a_2+a_3y_0^2+a_4y_0+a_5+a_6) = -\frac{1}{2}(\alpha y_0^2+\beta y_0+\gamma) = \frac{-(y_0^2+\frac{\beta}{\alpha}y_0+\frac{\gamma}{\alpha})}{2\frac{1}{\alpha}}$$

So that when we realize that $(y_0-\tilde{\mu}_0)^2 = y_0-2\tilde{\mu}_0y_0+\tilde{\mu}_0^2$, we find that if we have $(\frac{\beta}{-2\alpha})^2 = \frac{\gamma}{\alpha}$ (which can be checked to be the case), we may write:

$$\left\{\begin{array}{l}
\tilde{\sigma}_0 = \sqrt{\frac{1}{\alpha}} \\
\tilde{\mu}_0 = \sqrt{\frac{\gamma}{\alpha}} = \frac{\beta}{-2\alpha}
\end{array}\right.$$

Note that we now have a different expression for $\tilde{\sigma}_0$, namely $\sqrt{\alpha^{-1}} = \sqrt{1/\sigma'_{00}}$. It can be checked from linear algebra that indeed $(\det \mathbf{\Sigma}^{(1)})/(\det \mathbf{\Sigma}^{(2)}) = 1/\sigma'_{00}$. Finally, we thus arrive at the definition of the required multivariate density function for the conditional normal distribution:

$$f(y_0|y_1, y_2, \ldots, y_c) = g(y, \tilde{\mu}_0, \tilde{\sigma}_0) \tag{50}$$

$$\text{where } \begin{cases} \tilde{\sigma}_0 &= \frac{1}{\sqrt{\sigma'_{00}}} \\ \tilde{\mu}_0 &= \frac{\mu_0 \sigma'_{00} - \sum_{i=1}^{c} (y_i - \mu_i)\sigma'_{i0}}{\sigma'_{00}} \end{cases}$$

D Differential entropy for the histogram distribution

As defined in algorithm REHIEST in section 5.2, the modeled distribution for variable Y_i is defined to be 0 anywhere outside of $[\min^{Y_i}, \max^{Y_i}]$. Inside that range, it is a stepfunction where the amount of steps equals the amount of bins. First we need a generalized version of equation 25:

$$b_{j_0,j_1,\ldots,j_{n-1}}[k_0, k_1, \ldots, k_{n-1}] = \tag{51}$$
$$\left| \left\{ Y_{j_0}^{(S)q} \;\middle|\; q \in \{0, 1, \ldots, \lfloor \tau n \rfloor - 1\} \;\wedge \forall_{i \in \{0,1,\ldots,n-1\}} \left\langle k_i \leq \frac{Y_{j_i}^{(S)q} - \min^{Y_{j_i}}}{\text{range}^{Y_{j_i}}} r < k_i + 1 \right\rangle \right\} \right|$$

Now, starting from equation 12, for n variables $Y_{j_0}, Y_{j_1}, \ldots, Y_{j_{n-1}}$ that have a histogram distribution as defined by algorithms REHIEST and REHISAM, we can derive the differential entropy as follows:

$$h(Y_{j_0}, Y_{j_1}, \ldots, Y_{j_{n-1}}) =$$
$$-\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\cdots\int_{-\infty}^{\infty} \left(\begin{array}{c} f_{j_0,j_1,\ldots,j_{n-1}}(y_0, y_1, \ldots, y_{n-1}) \\ \ln(f_{j_0,j_1,\ldots,j_{n-1}}(y_0, y_1, \ldots, y_{n-1})) \end{array} \right) dy_0 dy_1 \ldots dy_{n-1} =$$
$$-\int_{\min^{Y_{j_0}}}^{\max^{Y_{j_0}}}\int_{\min^{Y_{j_1}}}^{\max^{Y_{j_1}}}\cdots\int_{\min^{Y_{j_{n-1}}}}^{\max^{Y_{j_{n-1}}}} \left(\begin{array}{c} f_{j_0,j_1,\ldots,j_{n-1}}(y_0, y_1, \ldots, y_{n-1}) \\ \ln(f_{j_0,j_1,\ldots,j_{n-1}}(y_0, y_1, \ldots, y_{n-1})) \end{array} \right) dy_0 dy_1 \ldots dy_{n-1} =_{\text{def}}$$
$$\left\{ \rho(\iota, \lambda) = \min^{Y_\iota} + \frac{\lambda}{r} \text{range}^{Y_\iota} \right\}$$
$$-\sum_{k_0=0}^{r-1}\sum_{k_1=0}^{r-1}\cdots\sum_{k_{n-1}=0}^{r-1} \left(\int_{\rho(j_0,k_0)}^{\rho(j_0,k_0+1)} \int_{\rho(j_1,k_1)}^{\rho(j_1,k_1+1)} \cdots \int_{\rho(j_{n-1},k_{n-1})}^{\rho(j_{n-1},k_{n-1})} \right.$$
$$\left. \left(\begin{array}{c} f_{j_0,j_1,\ldots,j_{n-1}}(y_0, y_1, \ldots, y_{n-1}) \\ \ln(f_{j_0,j_1,\ldots,j_{n-1}}(y_0, y_1, \ldots, y_{n-1})) \end{array} \right) dy_0 dy_1 \ldots dy_{n-1} \right) =$$
$$-\sum_{k_0=0}^{r-1}\sum_{k_1=0}^{r-1}\cdots\sum_{k_{n-1}=0}^{r-1} \left(\left(\frac{\text{range}^{Y_{j_0}}}{r} \frac{\text{range}^{Y_{j_1}}}{r} \cdots \frac{\text{range}^{Y_{j_{n-1}}}}{r} \right) \cdot \right.$$
$$\left. b_{j_0,j_1,\ldots,j_{n-1}}[k_0, k_1, \ldots, k_{n-1}] \ln(b_{j_0,j_1,\ldots,j_{n-1}}[k_0, k_1, \ldots, k_{n-1}]) \right)$$

The only fundamental difference with equation 10 for the discrete multivariate entropy lies in the factor $\prod_{i=0}^{n-1} \text{range}^{Y_{j_i}} / r^n$. In the case of discrete random variables, $r = n_d$ and the range is also equal to n_d, which makes this factor evaluate to 1 and disappear in the equation. We have arrived at the definition of the multivariate differential entropy for n continuous random variables with a histogram distribution according to algorithms REHIEST and REHISAM from section 5:

$$h(Y_{j_0}, Y_{j_1}, \ldots, Y_{j_{n-1}}) = \tag{52}$$
$$-\frac{\prod_{i=0}^{n-1} \text{range}^{Y_{j_i}}}{r^n} \sum_{k_0=0}^{r-1}\sum_{k_1=0}^{r-1}\cdots\sum_{k_{n-1}=0}^{r-1} \left(b_{j_0,j_1,\ldots,j_{n-1}}[k_0, k_1, \ldots, k_{n-1}] \ln(b_{j_0,j_1,\ldots,j_{n-1}}[k_0, k_1, \ldots, k_{n-1}]) \right)$$

E Additional algorithms

Algorithm GRAPHSEAOPTIMUMBRANCHING computes an optimum branching using Edmond's algorithm as specified in appendix B. In order to do so, it first computes a critical graph and then checks if the graph contains cycles. If it does not, the resulting selection of arcs is the optimum branching. If it does contain cycles, the derived problem is created following the definitions in appendix B. The algorithm is subsequently recursively applied to this smaller problem. From the result obtained by the recursive call, the final result is constructed using the correspondence, which is also stated in appendix B. The data that is used to achieve the computation of the optimum branching, mainly consists of the nodes V and the arcs A in the graph. As the source, target and weight of an arc are subject to changes for the derived problem while the actual arcs remain the same, the history of the source, target and weight for an arc during recursive calls is placed on a stack.

```
GRAPHSEAOPTIMUMBRANCHING(V,A,z)
1   (V, A^H) ← GRAPHSEACRITICALGRAPH(A,z)
2   (C, f) ← GRAPHSEACYCLES(V,A^H,z)
3   if |C| = 0 then
      3.1   B ← A^H
4   else
      4.1   (V̄, Ā, z̄, a^min) ← GRAPHSEADERIVEDPROBLEM(V,A,C,f,z)
      4.2   B̄ ← GRAPHSEAOPTIMUMBRANCHING(V̄,Ā,z̄)
      4.3   B ← GRAPHSEACORRESPONDENCE(B̄,Ā,z,f,C,a^min)
5   return(B)
```

Algorithm GRAPHSEACRITICALGRAPH computes a critical graph, given collections V and A of nodes and arcs respectively. To this end, all the arcs are considered. At each node v, the arc with target node v and with minimum weight is stored. All the arcs so stored, constitute the critical graph according to the definition as stated in appendix B.

```
GRAPHSEACRITICALGRAPH(A,z)
1   C ← new array of edarc with size z
2   A^H ← new vector of edarc
3   for i ← 0 to z − 1 do
      3.1   C[i] ← (⊥, ⊥, ⊥)
4   for i ← 0 to |A| − 1 do
      4.1   (S^s, S^t, S^c) ← A_i
      4.2   t ← TOP(S^t)
      4.3   c ← TOP(S^c)
      4.4   if C[t] = ⊥ then
              4.4.1   if c > 0 then
                        4.4.1.1   C[t] ← (S^s, S^t, S^c)
      4.5   else
              4.5.1   (S^s', S^t', S^c') ← C[t]
              4.5.2   c' ← TOP(S^c')
              4.5.3   if 0 < c' ≤ c then
                        4.5.3.1   C[t] ← (S^s, S^t, S^c)
5   for i ← 0 to z − 1 do
      5.1   if C[i] ≠ ⊥ then
              5.1.1 A^H_|A^H| ← C[i]
6   return((V, A^H))
```

Algorithm GRAPHSEACYCLES checks whether a critical graph has cycles and returns all of these cycles. To achieve this, a depth first search is performed at each non–visited node and as

soon as a node is visited that has been encountered earlier in a single depth–first run, the nodes are traversed backwards and the cycle is constructed.

```
GraphSeaCycles(V,A^H,z)
1   f ← new array of edarc with size z
2   m,m' ← 2 new arrays of boolean with size z
3   C ← new vector of (vector of integer, vector of edarc)
4   for i ← 0 to z − 1 do
    4.1   m[i] ← true
    4.2   m'[i] ← true
    4.3   f[i] ← (⊥,⊥,⊥)
5   for i ← 0 to |V| − 1 do
    5.1   m[V_i] ← false
    5.2   m'[V_i] ← false
6   for i ← 0 to |A^H| − 1 do
    6.1   (S^s,S^t,S^c) ← A^H_i
    6.2   t ← top(S^t)
    6.3   f[t] ← (S^s,S^t,S^c)
7   for i ← 0 to |V| − 1 do
    7.1   if ¬m[V_i] then
          7.1.1   (b,V^C,A^C,v^c,b^c) ← GraphSeaCyclesVisit(V_i,m,m',f)
          7.1.2   if b then
                  7.1.2.1   C_|C| ← (V^C,A^C)
8   return(C,f)
```

```
GraphSeaCyclesVisit(v,m[·],m'[·],f[·])
1   T ← (false,⊥,⊥,−1,false)
2   if m'[v] then
    2.1   V^C ← new vector of integer
    2.2   A^C ← new vector of edarc
    2.3   T ← (true,V^C,A^C,v,true)
3   if ¬m[v] then
    3.1   m[v] ← true
    3.2   m'[v] ← true
    3.3   (S^s,S^t,S^c) ← f[v]
    3.4   if S^s ≠ ⊥ then
          3.4.1   s ← top(S^s)
          3.4.2   (b,V^C,A^C,v^c,b^c) ← GraphSeaCyclesVisit(s,m,m',f)
          3.4.3   m'[v] ← false
          3.4.4   if b then
                  3.4.4.1   if b^c then
                            3.4.4.1.1   V^C_|V^C| ← v
                            3.4.4.1.2   A^C_|A^C| ← (S^s,S^t,S^c)
                            3.4.4.1.3   if v^c = v then
                                        3.4.3.1.3.1   b^c ← false
                  3.4.4.2   T ← (true,V^C,A^C,v^c,b^c)
4   return(T)
```

Algorithm GraphSeaDerivedProblem constructs the derived problem by first determining for each node in V whether it lies in $\hat{V}$ or in the collection of nodes of some cycle V_i. It then straightforwardly constructs $\overline{V}$ and $\overline{A}$. For each arc in $\overline{A}$, the new source $\overline{s}$, destination $\overline{t}$ and cost $\overline{c}$ are pushed onto the stacks of the arc, so that the top of the stacks hold the relevant problem data for the next recursive call to algorithm GraphSeaOptimumBranching.

```
GraphSeaDerivedProblem(V,A,C,f[·],z)
 1  e^v ← new array of integer with size z
 2  a^min new array of edarc with size |C|
 3  V̄ ← new vector of integer
 4  Ā ← new vector of edarc
 5  u ← new vector of (integer, integer, real)
 6  for i ← 0 to z − 1 do
    6.1  e^v[i] ← −1
 7  for i ← 0 to |V| − 1 do
    7.1  e^v[V_i] ← |C|
 8  for i ← 0 to |C| − 1 do
    8.1  (V^C, A^C) ← C_i
    8.2  for j ← 0 to |V^C| − 1 do
         8.2.1  e^v[V^C_j] ← i
    8.3  a^min[i] ← A^C_0
    8.4  for j ← 1 to |A^C| − 1 do
         8.4.1  (S^s, S^t, S^c) ← A^C_j
         8.4.2  c ← TOP(S^c)
         8.4.3  (S^s′, S^t′, S^c′) ← a^min[i]
         8.4.4  c′ ← TOP(S^c′)
         8.4.5  if c < c′ then
                8.4.5.1  a^min[i] ← (S^s, S^t, S^c)
 9  for i ← 0 to z − 1 do
    9.1  if e^v[i] = |C| then
         9.1.1 V̄_|V̄| ← i
10  z̄ ← z + |C|
11  for i ← 0 to |C| − 1 do
    11.1  V̄_|V̄| ← z + i
12  for i ← 0 to |A| − 1 do
    12.1  (S^s, S^t, S^c) ← A_i
    12.2  s ← TOP(S^s)
    12.3  t ← TOP(S^t)
    12.4  c ← TOP(S^c)
    12.5  (s̄, t̄, c̄) ← (s, t, c)
    12.6  if e^v[s] = |C| ∨ e^v[t] = |C| then
          12.6.1  if 0 ≤ e^v[s] < |C| then
                  12.6.1.1  s̄ ← z + e^v[s]
          12.6.2  if 0 ≤ e^v[t] < |C| then
                  12.6.2.1  t̄ ← z + e^v[t]
                  12.6.2.2  (S̃^s, S̃^t, S̃^c) ← f[t]
                  12.6.2.3  c̃ ← TOP(S̃^c)
                  12.6.2.4  (S^smin, S^tmin, S^cmin) ← a^min[e^v[t]]
                  12.6.2.5  c^min ← TOP(S^cmin)
                  12.6.2.6  c̄ ← c − c̃ + c^min
          12.6.3  u_|u| ← (s̄, t̄, c̄)
          12.6.4  Ā_|Ā| ← (S^s, S^t, S^c)
13  for i ← 0 to |Ā| − 1 do
    13.1  (S^s, S^t, S^c) ← Ā_i
    13.2  (s̄, t̄, c̄) ← u_i
    13.3  PUSH(S^s, s̄)
    13.4  PUSH(S^t, t̄)
    13.5  PUSH(S^c, c̄)
14  return((V̄, Ā, z̄, a^min))
```

Algorithm GRAPHSEACORRESPONDENCE computes the resulting optimum branching by first taking all of the arcs from $\overline{B}$ and then adding arcs from the cycles while leaving some arc out for each cycle to prevent cycles in the resulting branching B. To this end, the source and target of an arc in the original problem are required as well as the source and target of an arc in the derived problem. As these were stored on a stack, this data is readily available. Finally, as directly opposed to the end of algorithm GRAPHSEADERIVEDPROBLEM, the relevant problem data is removed from the top of the stack for each arc in the derived problem.

```
GRAPHSEACORRESPONDENCE(B̄,Ā,z,f[·],C,a^min[·])
1   f̄ ← new array of edarc with size |C|
2   for i ← 0 to |C| − 1 do
    2.1   f̄[i] ← (⊥,⊥,⊥)
3   for i ← 0 to |B̄| − 1 do
    3.1   (S̄^s,S̄^t,S̄^c) ← B̄_i
    3.2   t̄ ← TOP(S̄^t)
    3.3   if 0 ≤ t̄ − z < |C| then
          3.3.1   f̄[t̄ − z] ← (S̄^s,S̄^t,S̄^c)
4   B ← {(S̄^s,S̄^t,S̄^c) | (S̄^s,S̄^t,S̄^c) ∈ B̄}
5   for i ← 0 to |C| − 1 do
    5.1   (V^C,A^C) ← C_i
    5.2   (S^s break,S^t break,S^c break) ← a^min[i]
    5.3   s^break ← TOP(S^s break)
    5.4   t^break ← TOP(S^t break)
    5.5   (S̄^s,S̄^t,S̄^c) ← f̄[i]
    5.6   if S̄^t ≠ ⊥ then
          5.6.1   t̄ ← POP(S̄^t)
          5.6.2   (S^s break,S^t break,S^c break) ← f[TOP(S̄^t)]
          5.6.3   s^break ← TOP(S^s break)
          5.6.4   t^break ← TOP(S^t break)
          5.6.5   PUSH(S̄^t,t̄)
    5.7   for j ← 0 to |A^C| − 1 do
          5.7.1   (S^s,S^t,S^c) ← A^C_j
          5.7.2   s ← TOP(S^s)
          5.7.3   t ← TOP(S^t)
          5.7.4   if (s^break,t^break) ≠ (s,t) then
                  5.7.4.1   B_|B| ← (S^s,S^t,S^c)
6   for i ← 0 to |Ā| − 1 do
    6.1   (S̄^s,S̄^t,S̄^c) ← Ā_i
    6.2   POP(S̄^s)
    6.3   POP(S̄^t)
    6.4   POP(S̄^c)
7   return(B)
```

Algorithm GRAPHSEATOPOLOGICALSORT performs a topological sort for an array v^p of vectors that contains for each node its parents. The algorithm places the ordering of the variables in the ω vector as required. Note that the topological sort is performed using the arcs backwards as the base of an arc points to a parent and the parents have to be on the side with the larger indices in the ω vector.

```
GraphSeaTopologicalSort(v^p[·])
1   m ← new array of boolean with size l
2   for i ← 0 to l − 1 do
    2.1   m[i] ← false
    2.2   π(i) ← v^p[i]
3   z ← l − 1
4   for i ← 0 to l − 1 do
    4.1   if ¬m[i] then
              4.1.1   z ← GraphSeaTopologicalSortVisit(i,m,v^p,z)
5   return((π, ω))
```

```
GraphSeaTopologicalSortVisit(i,m[·],v^p[·],z)
1   m[i] ← true
2   for j ← 0 to |v^p[i]| − 1 do
    2.1   if ¬m[v^p[i]_j] then
              2.1.1   z ← GraphSeaTopologicalSortVisit(v^p[i]_j,m,v^p,z)
3   ω_z ← i
4   return(z − 1)
```

Algorithm GraphSeaArcAdd updates which arcs are still allowed to be added to the graph. All arcs that will cause the graph to become cyclic when they are added, are marked as *not allowed*. This is achieved by a depth–first search through the successor nodes v_s of v_1 including v_1 and for each encountered node to start another depth–first search through the precedessor nodes v_p of v_0 including v_0 so as to mark each arc (v_s, v_p) as *not allowed*. If the maximum amount of parents is achieved for node v_1, all arcs $(i, v_1), i \in \{0, 1, \ldots l-1\}$ are marked as *not allowed*. The algorithm returns the amount of arcs that have become newly marked.

```
GraphSeaArcAdd(κ, v_0, v_1, a[·,·], v^p[·], v^s[·])
 1   m ← new array of boolean with size l
 2   n^s, n^p ← 2 new vectors of integer
 3   S ← new stack of integer
 4   δ ← 0
 5   v^s[v_0]_{|v^s[v_0]|} ← v_1
 6   v^p[v_1]_{|v^p[v_1]|} ← v_0
 7   if a[v_0, v_1] then
     7.1   a[v_0, v_1] ← false
     7.2   δ ← δ + 1
 8   for i ← 0 to l − 1 do
     8.1 m[i] ← false
 9   PUSH(S, v_1)
10   while ¬EMPTY(S) do
     10.1   v ← POP(S)
     10.2   if ¬m[v] then
                10.2.1   m[v] ← true
                10.2.2   n^s_{|n^s|} ← v
                10.2.3   for k ← 0 to |v^s[v]| do
                             10.2.3.1   PUSH(S, v)
11   PUSH(S, v_0)
12   while ¬EMPTY(S) do
     12.1   v ← POP(S)
     12.2   if ¬m[v] then
                12.2.1   m[v] ← true
                12.2.2   n^p_{|n^p|} ← v
                12.2.3   for k ← 0 to |v^p[v]| do
                             12.2.3.1   PUSH(S, v)
```

```
13    for i ← 0 to |n^s| do
      13.1    for j ← 0 to |n^p| do
              13.1.1    if a[i, j] then
                        13.1.1.1    a[i, j] ← false
                        13.1.1.2    δ ← δ + 1
14    if |v^p[v_1]| = κ then
      14.1    for i ← 0 to l − 1 do
              14.1.1    if a[i, v_1] then
                        14.1.1.1    a[i, v_1] ← false
                        14.1.1.2    δ ← δ + 1
15    return(δ)
```

F Pseudo–code conventions in notation and data usage

All algorithms are written as functions by using capital letters, such as FUNCTION(). The assignment operation is written using $A \leftarrow X$, assigning X to A. Pseudo–code reserved words appear in boldface, such as **while**. The scope of structures is related to the indentation and numbering of lines. The scope of a grouping statement in line l is $l.*$. The remainder of the conventions concern the assumed to be available data structures and elementary data types:

Elementary data types	
Type	**Description**
boolean	Truth value (element of $\{\boldsymbol{false}, \boldsymbol{true}\}$)
integer	Element of $\mathbb{Z}$
real	Element of $\mathbb{R}$

Composite data structures	
Type	**Description**
array	Typed array, memory block with fixed size. • Starting index: 0 • Notation for indexing: $A[i]$. • Notation for determining the length of an array: $\|A\|$.
vector	Typed dynamic array, memory block with non–fixed size. • Starting index: 0 • Notation for indexing: V_i. • Notation for determining the amount of elements in a vector: $\|V\|$. • Special assignment notation: $V_{\|V\|} \leftarrow X$, appends X at the end vector V, increasing its size.
stack	Typed collection of elements with non–fixed size, allowing the following operations, all of which run in $\mathcal{O}(1)$ time: • EMPTY(S), returns ***true*** if stack S contains no elements, ***false*** otherwise. • PUSH(S, X), places X onto the top of stack S, increasing its size. • POP(S), returns the element that resides on the top of stack S and removes it from the stack, decreasing its size. • TOP(S), returns the element that resides on the top of stack S without removing it from the stack.