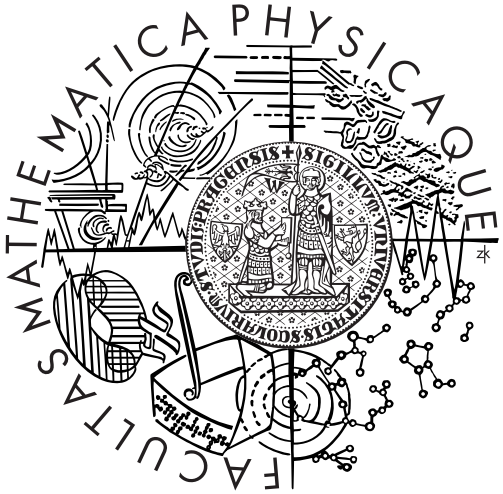
Charles University in Prague

Faculty of Mathematics and physics

MASTER THESIS



Martin Káldy

Modern evolutionary algorithms for detecting high-fitness areas

Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Ing. RNDr. Martin Holeňa, CSc.

Study Program: Computer Science

Software Systems

Prague, 2010

I would like to express my appreciation to my supervisor, Ing. RNDr. Martin Holeňa, CSc., for his invaluable advice, providing materials and kind guidance throughout writing this thesis.

I hereby declare that I have written this thesis myself using only the literature listed. I agree with using of this thesis.

Martin Káldy, Prague 27th November 2010

Contents

Název práce: *Moderní evoluční algoritmy pro hledání oblastí s vysokou fitness*
Autor: *Martin Káldy*
Katedra (ústav): *Katedra teoretické informatiky a matematické logiky*
Vedoucí diplomové práce: *Ing. RNDr. Martin Holeňa, CSc.*
e-mail vedoucího: *martin@cs.cas.cz*

Abstrakt: Evoluční algoritmy jsou optimizační techniky inspirované vývojem biologických druhů v přírodě. Používají konceptuálně jednoduchý proces střídající dvě fáze, a to reprodukci a výběr na základě fitness, a iterativně tak vyvíjejí stále lepší řešení. Evolučním algoritmům je věnováno dost pozornosti díky jejich schopnosti řešit i velmi komplikované optimizační problémy, na kterých jiné optimalizační metody mohou selhat kvůli existenci mnoha lokálních optim. Různých typů evolučních algoritmů byla navrhnuta celá šíře. V této diplomové práci se budeme věnovat skupině algoritmů "EDA" (z anglického Estimation of Distribution Algorithms), tedy algoritmy odhadující pravděpodobnostní rozdělení. Ve fázi vytváření nové generace EDA odhadne z vybrané rodičovské populace pravděpodobnostní rozložení a novou generaci generuje na základě tohoto rozdělení. V této práci naimplementujeme a použijeme několik existujících EDA tak, aby pracovaly v dohodnutém specifickém prostředí, které lze zhruba charakterizovat jako stromovité struktury obsahující jak diskrétní, tak spojité veličiny. Navíc také pro jedince zavádíme další omezení ve formě lineárních nerovnic. Implementovaná aplikace je navržená pro komunikci přes dohodnutá rozhraní, od nějž získává informace o modelu a o ukládání řešení do databáze, do které se pak všem vygenerovaným řešením přiřazuje fitness hodnota vypočítaná ze skutečných provedených experimentů.

Klíčová slova: *optimalizace, evoluční algoritmy, odhad rozdělení, fitness funkce*

Title: *Modern evolutionary algorithms for detecting high-fitness areas*
Author: *Martin Káldy*
Department: *Department of Theoretical Computer Science and Mathematical Logic*
Supervisor: *Ing. RNDr. Martin Holeňa, CSc.*
Supervisor's e-mail address: *martin@cs.cas.cz*

Abstract: Evolutionary algorithms are optimization techniques inspired by the actual evolution of biological species. They use conceptually simple process of two repeating phases of reproduction and fitness-based selection, that iteratively evolves each time better solutions. Evolutionary algorithms receive a lot of attention for being able to solve very hard optimization problems, where other optimization techniques might fail due to existence of many local optima. Wide range of different variants of evolutionary algorithms have been proposed. In this thesis, we will focus on the area of Estimation of Distribution Algorithms (EDA). When creating the next generation, EDAs transform the selected high-fitness population into a probability distribution. New generation is obtained by sampling the estimated distribution. We will design and and implement combinations of existing EDAs that will operate in business-specific environment, that can be characterized as tree-like structure of both discrete and continuous variables. Also, additional linear inequality constraints are specified to applicable solutions. Implemented application communicates with provided interfaces, retrieving the problem model specification and storing populations into database. Database is used to assign externally computed fitness from real world experiments to all solutions in the optimization process.

Keywords: *optimization, evolutionary algorithms, estimation of distribution, fitness function*

1 Organization of thesis

Purpose of this thesis is to explore and evaluate new designs of evolutionary algorithms from the family of Estimation of Distribution Algorithms (EDAs). The algorithms are not intended to simply outperform existing algorithms, but rather to fulfill some specific requirements that may be important for certain applications, most importantly, the algorithms must be able to evolve complex structures.

In section 3, we will outline state of the art in evolutionary algorithms and EDA particularly. In section 4, we will describe in detail the data structures that need to be evolved. We will examine in detail the characteristics of the structures, design algorithms for manipulation with the structures and prove their important characteristics. In section 5 we will introduce framework, that will reuse EDA algorithms and refine them to handle complex optimization structures. In section 6, we will take a closer look at details of our implementation. In section 7, we will describe tuning and testing methodology to measure effectiveness of the algorithms. We will present results of experiments with different algorithms and make some conclusions about their performance.

In appendix A, we summarized notation used throughout this thesis. Appendices B, C and D summarize agreed interface of the application; Starting with data structures in appendix B, custom function handles in Appendix C and interface functions in Appendix D.

Following sections are conclusion, bibliography, used notation and appendix and implementation reference.

2 Introduction

Optimization problems is a large family of tasks that refers to finding the best solutions from the space of all possible solutions. It is a wide field of study and has many subfields, including, for example, gradient-based methods, linear programming, dynamic programming, combinatorial optimization and genetic algorithms. Optimization is present virtually in any field of research as well as in our daily life. Although in real life problems typically include several criteria describing goodness of a solution, many optimization algorithms simplify the problem and only consider one single criterion.

In this thesis, we will only discuss the simple case of single-objective optimization, where any solution is mapped by the objective function (fitness function) onto a real value (fitness of solution). Goodness of a solution is measured solely by the fitness value. The fact that there is only one single real-valued criterion means that any two solutions are comparable and any set of solutions can be ordered. The actual value of the fitness function may have no direct interpretation except for solution comparison.

Speaking more formally, optimization may be defined as solving function:

$$\arg\max_{z \in Z} \left(fitness\left(z\right) \right)$$

where Z is the space of all available solutions. In broader sense, optimization process does not necessarily find the best solution in the space, but rather searches the space for a solution that has "reasonably high" value of the function $fitness(z)$. Normally, the more resources (number of fitness function evaluations, CPU time etc.) an algorithm has, better solution it is able to find. Different algorithms may be then compared according to fitness of the found solutions and amount of resources used. This topic will be examined in detail in chapter 5.3.

For now, we will use very basic definition of the solution space: Problem space is Cartesian product of either:

- Discrete variable with finite (usually very small) number of defined values
- Real variable from interval $(-\infty, \infty)$

That may not very well represent real world problems, but on the other hand, it is very simple. Most of the existing evolutionary algorithms are designed for exactly such solution space model. The simplicity is very important advantage, because then the algorithms can be easily analyzed. Later in the chapter 4, we will introduce much more complex models and we will design techniques to adapt classic evolutionary algorithm to operate correctly for that models.

3 State of the art in Evolutionary algorithms

Genetic or Evolutionary algorithms are optimization algorithms inspired by process of natural evolution of species, as demonstrated first by Charles Darwin in his famous book On the Origin of Species [8]. The objective of the algorithm designs is not to model the natural evolution realistically as possible, though. We want to use the principles behind the evolution and use them in a way that best suits the scientific or industrial needs. While Charles Darwin speaks of the natural selection in with regard to self-preservation and reproduction, we introduce purely artificial selection criteria of the particular optimization problems we want to solve. Modern day evolutionary algorithms may diverge quite a lot from the evolution as we can observe it in nature (as we will see in section 3.4).

In section 3.1, we will give a short overview of some well known genetic algorithms. Section 3.2 is a preface to EDA algorithms and explains their central concept of Probabilistic graphical models. Section 3.3 summarizes basic concepts of data clustering, that is widely used among various EDAs. In section 3.4, we will look closer at the family of EDAs. We will make a brief presentation of existing algorithms, explain their key ideas and show the characteristics of their operation.

3.1 Brief overview of classic genetic algorithms

Classic genetic algorithms maintain population of individuals, that are being iteratively optimized, producing new generations of offsprings. Depending on

the exact algorithm and its parameters, newly created offsprings may either completely replace the old generation, or part of the parental generation may survive and get processed again. Functionality of the genetic algorithms may be summarized to following steps:

1. Initial population is generated. It may be generated completely random, or there may be rules favoring some attribute values or combinations, or it may be provided by some concrete design.

2. Fitness of the individuals is evaluated. Note that fitness may be decided by some analytic benchmark function, but it can be also derived from real-life experiment.

3. Selection is applied. The fittest solutions are likely to be selected, the less fit are likely to be eliminated. The selection is usually randomized, thus even bad solutions have slight chance of being selected.

4. New generation of solutions is derived from the selected individuals. Offspring use crossover and mutation operators to combine and further modify parental traits. Both operators have stochastic nature. Depending on the algorithm, offsprings completely replace the old solutions, or some old solutions may survive into the next generation (unchanged).

5. Repeat process from 2.

One important thing to realize is that the actual "improvement" of the population does happen in the step 3, the selection. To use the biological analogy, herd of antelopes does not become faster when new antelopes are born, but when the slowest antelopes are eaten by a lion. Step 4 is, on the other hand, essential for innovation. Innovation does not necessarily lead to improvement, quite contrarily, many innovations turn out to be useless and will be eliminated during following selection phases.

In classic genetic algorithms, offspring genome is derived from genome of parental pairs (or parental groups in general). As we will see in section 3.4, whole selected population or clusters of selected population may contribute to offspring genome. In following subsections, we will look at two examples of most generic evolutionary algorithms: Simple Genetic Algorithm (and its modifications) and Evolution Strategies.

3.1.1 SGA - Simple Genetic Algorithm

In this section we describe Simple Genetic Algorithm as presented by Larrañaga [12]. SGA is a simplistic variant of evolutionary optimization algorithms. Solutions of a discrete problem are encoded as vectors of bits with constant length - solution space is $X = \{0, 1\}^n$. Each generation, fitness of each individual is estimated and pairs of parents are selected from the population. The more fitness a solution has, more probably it will be selected as parent, i.e. the probability of selecting an individual is proportional to its actual numeric fitness value. Parents are grouped to pairs and new generation of solutions is derived from each

of the parental pairs. Solutions behave as "hermaphrodites", any two solutions may form a parental pair. Using crossover and mutation operators, offsprings are generated.

Crossover operator mix information represented in the offspring parents. SGA uses one-point crossover: the crossing point i is selected from $\{1, .., n-1\}$, while the probabilities of each value are uniform. Offspring's values of nodes $1, .., i$ are copied from the first parent, values of nodes $(i+1), .., n$ from the second parent. Consider following example: we have two parents with chromosomes $ABCDEFGH$ and $abcdefgh$, each letter representing value 0 or 1. For crossing point at index 2 the offspring would have chromosome $ABcdefgh$.

Mutation of values is applied to any node with a probability p_m: value of mutated node is swapped from 0 to 1 or vice versa. Mutation rate 0 means that no mutation is applied at all, mutation rate 0.5 would mean completely random offsprings. In the previous example, if mutation occurred at indices 1 and 4, the chromosome would look like $\bar{A}B\bar{c}\bar{d}efgh$ for $\bar{x}$ denoting opposite value to a value x.

Very generally speaking, crossover operator provides faster and rougher convergence, trying to combine existing substrings of chromosomes. Mutation, on the other hand, serves for fine tuning and creating innovative traits in the long run. Mutation rates are generally kept very low, although optimal value may vary depending on the problems being solved.

Key advantage of SGA is its simplicity, that makes them easier to analyze. For real-life usage, there are several variants of the SGA. Some of the modifications include:

- One-point crossover can be replaced by generating more crossing points, especially for longer chromosomes. The functionality is similar, first parent provide values from the start to the first crossing point, then until the next crossing point values are taken by second parent, then until the third crossing point, values are again taken by the first parent and so on. Importantly, the length of the chromosomes is constant and value of the offspring at i-th index comes always from same index of one of the parents.

- There are several methods to select parental pairs. Sometimes the fitness value can only provide comparison between fitness of solutions and should not be linearly proportional to the actual probability of their selection. In such case, rank-based selection and tournament methods are more suitable, and will be described later (See 3.1.3).

- Bit strings are not the only possible solution representation. Different encoding may use (usually small) integers instead of bits, permutations or even tree structures. In we change the chromosome representation, we have to also redesign mutation and crossover operators.

One of disadvantages of SGA is its inability to directly process real valued properties. Real values may be encoded using binary encoding. .

3.1.2 ES - Evolution strategies

Evolution strategies is a another example from the family of genetic algorithms, proposed by Schwefel [5]. It is widely used alternative to gradient-based methods of numeric optimization, that are susceptible to failure in noisy environments or in environments with many local optima. Evolution strategies do not have rigidly defined design; They are rather a framework of algorithms, composed from many components.

Let's start with the solution representation. Unlike SGA, Evolution Strategies optimize solutions in real valued continuous space. Solutions themselves are represented as vectors of real numbers. Part of the numbers directly apply to the solution properties and part serve as metadata. We differentiate between problem space Z, consisting of n real variables $Z = \mathbb{R}^n$ and metadata space S consisting of m additional variables $S = \mathbb{R}^m$. Solution space is Cartesian product of both: $X = Z \times S$. Number of data dimensions is defined by the dimensionality of the optimization problem, and exact number of metadata parameters may vary depending on both dimensionality of the problem and type of evolution strategies. Metadata values are of two types: mutation standard deviations σ (m_σ values) and values determining covariances α (m_α values). Mutation operator is defined as adding random vector from normal distribution with zero mean. Metadata describe parameters of this normal distribution. In the simplest case, $m_\sigma = 1,\ m_\alpha = 0$, there is one value determining the standard deviation of mutations and no correlations whatsoever. Fairly common is usage of one metadata value for standard deviation of each problem dimension and no correlations: $m_\sigma = n,\ m_\alpha = 0$. The most general case additionally applies correlation factors to all dimension combinations: $m_\sigma = n,\ m_{alpha} = n \cdot (n-1)/2$. Although more general approach may model desired distribution more precisely, it also needs to learn more parameters, or in other words ruin otherwise promising solution if some of its meta-parameters are accidentally learned wrong.

We already introduced genetic operator of mutation. Mutation of problem variables is parametrized only as its initial mutation values. The actual mutation of the problem variables is evolved together with the actual values. Additionally, the mutation parameters are also submitted to mutation operators (using notation from Larrañaga [12]):

$$\sigma_i' = \sigma_i \cdot exp\left(\sigma_0 \cdot \mathcal{N}\left(0,1\right) + \sigma_\sigma \cdot \mathcal{N}_i\left(0,1\right)\right)$$
$$z_i' = z_i + \sigma_i' \cdot \mathcal{N}_i\left(0,1\right)$$

for $\mathcal{N}_i\left(\mu,\sigma\right)$ being normal distribution in i-th dimension, σ_0 being scalar parameter of collective mutation of mutation factors and σ_σ scalar parameter of individual mutation of mutation factors. By introducing correlations, we have:

$$\alpha_{ij}' = \alpha_{ij} + \sigma_\alpha \cdot \mathcal{N}_{ij}\left(0,1\right)$$

$$\sigma_i' = \sigma_i \cdot exp\left(\sigma_0 \cdot \mathcal{N}\left(0,1\right) + \sigma_\sigma \cdot \mathcal{N}_i\left(0,1\right)\right)$$

$$z'_i = z_i + \sigma'_i \cdot \mathcal{N}_i(0, C')$$

for σ_α being parameter of correlation mutation and $\mathcal{N}(\mu, C)$ being multivariate normal distribution with covariance matrix C, defined as follows:

$$c_{ij} = \begin{cases} \sigma_i & if\ i = j \\ \frac{1}{2}\left(\sigma_i^2 - \sigma_j^2\right) tan\left(2\alpha_{ij}\right) & if\ i \neq j \end{cases}$$

Crossover operator is considered less important in evolution strategies, as main focus is kept on the mutation. Crossover operator is generalized: there may be one or many parents (one parent would actually mean no recombination at all). Without going into much detail, ES may either use discrete recombination, taking value from one selected of the parents, or intermediary recombination, combining values as average of parental values. Discrete approach is generally used for problem variables and intermediary for metadata [12].

Selection and offspring generation. Selection algorithm of ES is deterministic; We differentiate two situations:

- (μ, λ) "comma" selection strategy, when λ offspring from the previous generation is put into selection pool and μ most fit are selected ($\mu < \lambda$).
- $(\mu + \lambda)$ "plus" selection strategy, when λ offspring and μ parents from the previous generation comes into selection pool and the best μ solutions of the pool are selected as new parents (no equivalent limitation for μ and λ as before).

ES may also use notation $(\mu/\rho, \lambda)$ and $(\mu/\rho + \lambda)$ where ρ denotes number of parents that each offspring is descended from.

3.1.3 Selection methods

We have already mentioned selection as a propeller of the fitness improvement. Most straightforward approach to selection is to select simply the most fit solutions. While this approach is viable, it is prone to get stuck in local optima. Sometimes, innovation should take a few steps to become actually viable, though those steps might tend to have lower fitness than the local optima. Selection should therefore not automatically dismiss less fit solutions, although it should naturally give less preference to them.

Many evolutionary algorithms have very similar selection phase interface: when the whole population is evaluated for fitness, part of the population is selected based on the fitness, potentially grouped to parental pairs or groups. In fact, we are mostly able to decompose evolutionary algorithms into two components: selection and reproduction of the selected, while both are largely independent. Few algorithms deviate from this scheme, one of such is the continuous PBIL with Gaussian distributions, proposed by Sebag and Ducolumbier [15]. Firstly, the algorithm develop generations of distributions, instead of standartly evolved generations of populations. Instantiation, selection and modification of the distribution is the optimization step in PBIL. Secondly, the selection in the

optimization step uses exactly three individuals: two best fitting solutions and one worst, used as counter-example.

In this section, we will introduce some general algorithms to perform the "generic" selection valid for most evolutionary algorithms. We will also declare their parameters, and we will intend to unify their parameters, so that they are "shared" among different algorithms as much as possible. Len N denotes size of the population. Selection should select $N \cdot selectionSize$ individuals to become parents ($selectionSize$ defines percentage, not actual number of selected individuals).

- *Truncation selection* is perhaps the most straightforward selection; It selects fixed percentage proportion ($N \cdot selectionSize$) of the best solutions. As said before, this approach shows tendency to get stuck in local optima.

- *Tournament selection* first randomly generates $N \cdot selectionSize$ tournament tuples of size $tournamentSize$ and selects best sample from each tournament. These winners form the parental population. Some low-fitness individuals may survive if they get into tournament with another low-fitness individuals. The bigger the $tournamentSize$ is, more selection pressure is applied. (in pairs, individual has to be better than just one other competitor, while in big tournaments individual has to be best of many).

- *Rank selection* selects ($N \cdot selectionSize$) samples randomly with weighted probabilities derived from ranks respectively to their fitness. Best solution will have weight N, worst solution will have weight 1. Lower-fitness solutions still retain slight chance of becoming a parent.

- *Fitness value selection* is similar to the *rank selection*, but uses the actual value of fitness function as weight. If fitness of a solution is negative, its weight is 0. Suitability of this selection depends on the exact definition of fitness function; Unlike previous cases, if we, for example, use logarithm of fitness function instead of fitness function, the selection algorithm behaves differently.

There are two additional procedures that may be applied to the selected algorithm.

Elitism. Selecting function may select small number of the very best solutions from the population into so-called elite. Elitist population will always survive unchanged into the next generation. Number of selected elitist solutions is determined by integer parameter *elitism* defined in the structure. Presence of the elitist mechanism is determined by value of parameter *elitism* being greater than zero.

Niche selection. This is not an exact algorithm, rather an additional mechanism to enhance properties of the selection algorithm. What happens in most evolutionary algorithms is when there are identified high fitness solutions near multiple local optima, algorithms tend to converge to the single most promising optimum among them. This may happen either by dominance of the slightly

better optimum or simply by genetic drift. In niche selection, we introduce additional mechanism that intends to diversify the selected population by means of fitness penalization of individuals that are similar to other selected individuals.

There are several proposals how niching may be introduced. One of the proposed methods is *shared fitness* proposed by Gao and Hu *[9]*. Let w_i denotes standard weight of selection of individual with index i, based on its fitness. Using shared fitness, this weight is modified:

$$w_i' = \frac{w_i}{\sum_{j=1}^{N} shared(d_{ij})}$$

where d_{ij} is distance between individuals i and j, and *shared* function is defined as follows:

$$shared(d_{ij}) = \begin{cases} 0 & if\ d_{ij} \geq r \\ 1 - (d_{ij}/r) & if\ d_{ij} < r \end{cases}$$

for r being radius of the niching. Note that the sum in the first expression is always at least 1, because $shared(d_{ii}) = 1$ (individual shares its fitness "with itself"). Niching is of course not the only possible method of keeping diversity among the population, but it may help to keep it such. Drawback of the niching is potentially slower convergence, measured by number of function evaluations needed to reach certain fitness.

3.2 Probabilistic graphical models

Probabilistic models have wide use in areas such as computer learning, data mining and knowledge representation.

By **probabilistic models** we will mean sets of rules and equations that describe probability distributions in the simplified solution space (i.e. unconstrained space before corrections, for more details see section 4.2.3). Probabilistic **graphical** models in particular use graph representation to describe structure of the data, modeling conditional (in)dependencies between the random variables. The graph is learned from the samples selected by the evolutionary algorithms. In following subsections, we will present existing probabilistic graphical model, Bayesian networks, in more detail and briefly outline another model, Gaussian Networks.

Both of the models share the common concept of dependencies between variables and will use similar notation. Our notation is derived from notation used by Larrañaga [12].

One dimensional variable is represented as X_i, and may be discrete or continuous. We use $X = (X_1, X_2, ..., X_n)$ to represent d-dimensional variable and $x = (x_1, x_2, ..., x_n)$ to represent its instance. *Joint generalized probability distribution* in discrete case noted as $p(X = x)$ describes actual probability of a solution, whereas in continuous case, we use density function $f(X = x)$. *Generalized conditional probability distribution* given value x_j of variable X_j

is denoted as probability $p(X_i = x_i \mid X_j = x_j)$ for a discrete variable X_i or density $f(X_i = x_i \mid X_j = x_j)$ in continuous case. The expressions above may be also substituted using more compact notation $p(x)$, $f(x)$, $p(x_i \mid x_j)$, $f(x_i \mid x_j)$. Let $x_i^{(k)}$ denotes k-th value of variable X_i. Set of values of X_i is $values(X_i) = \{x_i^{(k)} | k = 1..r_i\}$ and its size is r_i.

3.2.1 Bayesian Networks

Definition of Bayesian networks Bayesian Networks is a model that uses directed acyclic graph to model dependencies between discrete random variables. Each variable is represented by a graph node. Conditional dependencies between variables are edges from "parental" nodes to the "child" node. Two nodes are connected by edges iff there is a direct conditional dependence between them. If edges $X_{j_1} \to X_i$, $X_{j_2} \to X_i$, ..., $X_{j_K} \to X_i$ are all the edges directed to the node X_i, we will define this as K-dimensional variable $Pa_i^S = X_{j_1} \times X_{j_2} \times .. \times X_{j_K}$ consisting of subset of one dimensional variables of decomposed variable X, and we will call Pa_i^S set of parental nodes of node X_i in the structure S. We will use notation $pa_i^S = (x_{j_1}, x_{j_2}, ..., x_{j_K})$ for an instance of the variable Pa_i^S, or more specifically, $pa_i^{S(j)}$ for j-th value in value set $values(Pa_i^S)$. Important property of Bayesian networks is that the modeled probability distribution can be expressed as factorization:

$$p(x) = p(x_1) \cdot p(x_2 \mid pa_2^S) \cdot ... \cdot p(x_N \mid pa_N^S) = \prod_{i=1}^{n} p(x_i \mid pa_i^S) \tag{1}$$

For any structure S, the probabilistic distribution also depends on finite set of distribution parameters $\theta_S \in \Theta_S$. These parameters are:

- For each variable X_i without parents ($Pa_i^S = \emptyset$), and for every value $x_i^{(k)}$ from the variable value set, we need unconditional probability $p(x_i^{(k)}) = \theta_{i1k}$ (Note that value set of non-existing parents has only 1 value - empty. This will simplify later expressions where variables with and without parents are used together)

- For each variable X_i with parents ($Pa_i^S \neq \emptyset$), for every value $x_i^{(k)}$ from the variable value set and every possible combination of values $pa_i^{S(j)}$ (j-th value of the value set of parental combinations), we need to define conditional probability $p(x_i^{(k)} \mid pa_i^{S(j)}) = \theta_{ijk}$

In the simplest possible case when all variables X_i are binary, number of parameters required to specify distribution of a variable with n parents is $2^{(n+1)}$. In case that all variables X_i are k-ary (each may have k possible values), the number of parameters is $k^{(n+1)}$. This clearly demonstrates why we have to keep the network as simple as possible.

Learning a Bayesian network Larrañaga[11] demonstrated method of learning a Bayesian network from the data means determining structure S and parameters θ_S. Learning of S is called *structural learning*. Once we have a structure S, we may learn parameters θ_S and refer to such process as *parameter learning*. There is no single universal rule to induce the network.

One of the learning algorithms, known as "PC algorithm" is based on the idea of detecting conditional independence between variables. Algorithm starts with full graph and removes edge if its nodes pass statistical tests for independence.

Bayesian scores is another way to learn the structure. We express our uncertainty about the model by variable S^h describing hypothesis about the structure. Learning the structure corresponds to finding the most probable structure. Given the data D, we compute a posteriori probability $p(S^h|D)$ using Bayesian rule

$$P(S^h|D) = \frac{p(S^h) \cdot p(D|S^h)}{\sum_S p(S) \cdot p(D|S)}$$

In this thesis, we will be using different method, based on likelihood-based penalized score.

Maximum likelihood estimator for Bayesian networks Perhaps the most straightforward way to determine structure and parameters is maximum likelihood estimator (in fact, it is normally the penalized maximum likelihood estimator used, that introduces penalization heuristic to avoid induction of too complex models).

Let $D = \{x^1, x^2, ..., x^N\}$ is a database of N samples of the variable X. Once we know the Bayesian network structure S, we may compute parameters estimate $\hat{\theta_S}$ of parameter θ_S that would maximize likelihood of the data $L(D|S, \hat{\theta_S})$. We define that good estimate of parameters of Bayesian network S and θ_S is such that maximizes the likelihood of the data as much as possible. To evaluate goodness of the structure, we may compute maximum possible likelihood score for all possible parameters θ_S

$$score(S) = \max_{\theta_S \in \Theta_S} (\log L(D|S, \theta_S)$$

Note that we are not using the likelihood, but its logarithm instead. Logarithm is monotonic function and using it on the likelihood does not change point where it is maximal. Purpose of the logarithm is convenience: to simplify likelihood expressions that usually contain lot of products and exponents (A_i, e_i are dummy variables):

$$\log(\prod_i (A_i)^{e_i}) = \sum_i e_i \cdot \log A_i$$

As we declared before, N stands for number of samples in data D. Let N_{ijk} denotes number of samples x^w in data D such that $x_i = x_i^{(k)}$ and $pa_i^S = pa_i^{S(j)}$.

Let $r_i = |dom(X_i)|$, $q_i = |dom(Pa_i^S)|$ and $N_{ij} = \sum_{k=1..r_i} N_{ijk}$. Using equation 1 of factorized probability distribution in Bayesian networks, we obtain

$$\log L(D|S, \theta_S) = \log \prod_{w=1}^{N} \prod_{i=1}^{n} p(x_i^w | pa_i^{S,w}, \theta_{S,i}) = \sum_{i=1}^{n} \sum_{j=1}^{q_i} \sum_{k=1}^{r_i} \log(\theta_{ijk})^{N_{ijk}}$$

It is easy to demonstrate that maximum likelihood estimator of the θ_{ijk} is given by $\hat{\theta_{ijk}} = \frac{N_{ijk}}{Nij}$. Likelihood of the structure is then:

$$\log L(D|S, \theta_S) = \sum_{i=1}^{n} \sum_{j=1}^{q_i} \sum_{k=1}^{r_i} N_{ijk} \cdot \log \frac{N_{ijk}}{N_{ij}}$$

If $N_{ij} = 0$ for some i, j, we have to eliminate this combination from the sum. Logically, such combination have no effect on the actual likelihood and estimate of θ_{ijk} for $k = 1, .., r_i$ may be arbitrary. We may use some default value as $\theta_{ijk} = \frac{1}{r_i}$ or $\theta_{ijk} = \frac{\sum_{j=1}^{q_i} N_{ijk}}{\sum_{j=1}^{q_i} N_{ij}}$. Again, we have to omit any j that yields zero N_{ij}.

Penalized maximum likelihood greedy algorithm for Bayesian networks Not only do we have maximum likelihood estimator for a complete structure, we may limit the estimate to one single node. Suppose without loss of generality that variables $X_i, ..., X_n$ are topologically sorted, i.e. there is no edge $X_j \longrightarrow X_k$ for some $j > k$. Greedy algorithm starts with no dependencies and evaluates parents for the variables in their topological order. For each variable X_i, algorithm tries in a loop to add the parent that would be most profitable in terms of penalized maximum likelihood:

$$\sum_{j=1}^{q_i} \sum_{k=1}^{r_i} N_{ijk} \cdot \log \frac{N_{ijk}}{N_{ij}} - penalization(X_i, S, D)$$

When a parent is added to the graph, it is fixed and algorithm continues with adding next parents to that node. With each parent, likelihood is usually increasing, and also the penalization grows. Loop of parent addition is stopped when no new parent would be able to increase the penalized likelihood. When this happens, algorithm continues to next node.

There are several proposals for the penalization function. Bayesian Information Criterion (BIC) cited by Larrañaga [11] evaluates penalization function as:

$$penalization(S, D) = dim(S) \cdot \frac{1}{2} \log(N) = \sum_{i=1}^{n} q_i(r_i - 1) \cdot \frac{1}{2} \log(N)$$

that can be decomposed for each node as:

$$penalization(X_i, S, D) = q_i(r_i - 1) \cdot \frac{1}{2} \log(N)$$

Generally speaking, greater penalization induces simpler structure. In presented algorithms, we will be using BIC penalization multiplied by algorithm parameter *penalizationFactor*, that controls the simplicity or complexity of the induced network.

3.2.2 Gaussian Networks

Gaussian networks (Shachter and Kenley [16]) are multidimensional probabilistic models that model certain interactions among dimensions. Similarly to Bayesian Networks, Gaussian networks construct acyclic graph of dependencies among problem variables, this case the variables are continuous. We will use the same notation for K-dimensional "parental" variable $Pa_i^S = X_{j_1} \times X_{j_2} \times .. \times X_{j_K}$ consisting of subset of one dimensional variables of decomposed multidimensional space X, and we will call Pa_i^S set of parental nodes of node X_i in the structure S. We will use notation $pa_i^S = (x_{j_1}, x_{j_2}, ..., x_{j_K})$.

Marginal distributions of Gaussian networks are modeled as univariate normal distributions with regression coefficients for their parental values:

$$f(x_i|pa_i^S) = \mathcal{N}\left(m_i + \sum_{X_j \, in \, Pa_i} b_{ji} \cdot (x_j - m_j), v_i\right)$$

where $\mathcal{N}(m, v)$ is univariate normal distribution with mean m and variance v. It can be shown that this actually can be rewritten to the standard definition of multivariate normal distribution[12]. One important observation is that correlation between two modeled variables does not directly mean dependence in the network, a classic example that correlation does not imply causality.

If we compare the model to the Bayesian network scheme, we my note that the Gaussian model is generally simpler. For a variable X_i with n parents, we need to specify:

- mean m_i of the normal distribution
- variance v_i of the normal distribution
- n linear factors b_{ij} for all $X_j \in Pa_i$

That is $(n+2)$ parameters. Compare this to the number of parameters needed to specify distribution of variable with n parents in Bayesian network, that is even in the simplest case $2^{(n+1)}$. This difference is caused by the fact that Gaussian networks derive influence of combination of parental values as combination of their influences. Bayesian networks describe explicitly influence of each parental value combination.

Maximum likelihood estimator for Gaussian networks Likelihood in the Gaussian networks will be estimated for one variable, depending on the values of the parental variables. Let $D = \{x^1, x^2, ..., x^N\}$ is a database of N samples x^w, $w = 1, .., N$ of the variable X, and let S defines structure of the Gaussian network. Given S, we compute parameters estimate $\hat{\theta_S}$ of parameter θ_S (stands for set of aforementioned parameters m, v and b) that would maximize likelihood of the data $L(D|S, \hat{\theta_S})$, exactly the same way as we did for Bayesian networks.

By definition of likelihood of normal distributions, we obtain:

$$L(D|S,\theta) = \prod_{w=1}^{N}\prod_{i=1}^{n} \frac{1}{\sqrt{2\pi v_i}} \cdot e^{-\frac{1}{2v_i}\left(x_i^w - m_i - \sum_{x_j \in pa_i} b_{ji}(x_j - m_j)\right)^2}$$

We will naturally use logarithm of likelihood instead:

$$\log L(D|S,\theta) = \sum_{r=1}^{N}\sum_{i=1}^{n}\left(-\frac{1}{2}\log\left(2\pi v_i\right) - \frac{1}{2v_i}\left(x_i^w - m_i - \sum_{x_j \in pa_i} b_{ji}\left(x_j - m_j\right)\right)^2\right)$$

Maximum likelihood estimates are (Larrañaga [11]):

$$\hat{m_i} = \overline{X_i}$$

$$\hat{b_{ji}} = \frac{S_{X_j X_i}}{S_{X_j X_j}}$$

$$\hat{v_i} = S_{X_i X_i} - \sum_{X_j \in Pa_i} \frac{S_{X_j X_i}^2}{S_{X_j X_j}} + 2 \sum_{X_j \in Pa_i} \sum_{X_k \in Pa_i, k>j} \frac{S_{X_j X_k} \cdot S_{X_j X_i} \cdot S_{X_k X_i}}{S_{X_j X_j} \cdot S_{X_k X_k}}$$

where

$$\overline{X_i} = \frac{1}{N}\sum_{w=1}^{N} x_i^w$$

$$S_{X_i X_j} = \frac{1}{N}\sum_{w=1}^{N}\left(x_i^w - \overline{X_i}\right)\left(x_j^w - \overline{X_j}\right)$$

being mean and (co)variance of data of variables X_i and X_j.

Penalized maximum likelihood greedy algorithm for Gaussian networks The algorithm for constructing Gaussian network by greedy algorithm using penalized maximum likelihood estimator is the same as in the case of Bayesian networks, only the actual likelihood estimator is the one from the previous paragraph. Suppose we have sorted nodes $X_1, .., X_n$. We will iterate by the nodes and at each step on node X_i, we suppose that parameters for all previous nodes $X_1, .., X_{i-1}$ are already set.

We will estimate likelihood separately for the node X_i and add penalization for size of the set of model parameters for node X_i, resulting in expression:

$$\sum_{r=1}^{N} \left(-\frac{1}{2} \log\left(2\pi v_i\right) - \frac{1}{2v_i} \left(x_i^w - m_i - \sum_{x_j \in pa_i} b_{ji} \left(x_j - m_j\right) \right)^2 \right) - penalization(X_i, S, D)$$

for parameters m_i, v_i and b_{ij} estimated as in previous paragraph. The algorithm is greedy, when a parent is added to the graph, it is fixed. Algorithm continues with adding next parents to that node and stops when increase in penalization outweights the improvement in likelihood. Then the algorithm moves to estimating of the next node.

There are multiple options of choosing a penalization function. We may use the same penalization as in the case of Bayesian networks. In this thesis, BIC criterion is used:

$$penalization(X_i, S, D) = (2 + |Pa_i|) \cdot \frac{1}{2} \log(N)$$

In presented algorithms, we will be using this penalization multiplied by algorithm parameter *penalizationFactor*, that controls the simplicity or complexity of the induced network.

3.3 Clustering algorithms

In cluster analysis, we assign data samples from dataset D into C disjoint subsets. Clusters contain data that are close in space or have some similar characteristics. Formally, we will define clustering as a function that gets N solutions and assigns each solution its index of the cluster, thus defining clusters $D^1, .., D^C$ as follows:

$$clust_C : X^N \longrightarrow \{1, .., C\}^N$$

$$D^c = \{x^w \in D | clust(x^w) = c\}$$

Determining how the samples are divided into clusters is a problem from the area of unsupervised machine learning. We will present two methods for clustering, K-means clustering and hierarchical clustering.

3.3.1 K-means standard clustering algorithm

K-means algorithm divides data samples into predefined number of clusters. Sample is always assigned to the cluster with the closest mean:

$$clust_C(x^w) = \arg \min_{c \in \{1,..,C\}} distance(x^w, mean(D^c))$$

$$mean(D^c) = \frac{1}{|D^c|} \sum_{x \in D^c} x$$

Algorithm operates in iterations:

1. First, C random samples are selected, acting as first means.
2. In each step, all solutions are assigned to the cluster with closest mean $argmin_{c \in \{1,..,C\}}\{distance(x^w, mean(D^c))\}$.
3. if step 2 performed any change in cluster assignment, means $mean(D^c)$ are recomputed and step 2 is repeated.

3.3.2 Hierarchical clustering

Hierarchical clustering itself does not create flat division of solutions to clusters; As the name indicates, the result obtained by hierarchical clustering is a binary tree of subclusters. The top level cluster is the whole dataset, the leaves of the hierarchy are one-item clusters containing each single solution. Algorithm for deciding cluster tree is fairly simple, constructing the tree step by step from bottom:

1. First, make a leaf cluster for each single solution
2. Count distances between all top level clusters. Depending on the algorithm parameters, distance is computed:
   - as average of distances in solution pairs with one solution from each cluster
   - as median of distances in solution pairs with one solution from each cluster
   - between two closest solutions in each cluster
   - between two furthest solutions in each cluster
   - another criterion

   Also, different metrics may be used to compute distances in solution space.
3. Merge two closest top level clusters into a newly created cluster
4. Repeat from step 2 until there is only one top level cluster

The second phase of the algorithm is interpretation of the tree. EDAs usually operate on flat set of clusters without hierarchy. We would like to partition solutions into cluster using some natural divisions between data. We don't even fix the number of clusters beforehand, as in the case of k-means.

Essential number for merging or splitting two clusters is the inconsistency value. Each non-leaf cluster is a **link** between two subclusters. **Height** of the link is the distance between the linked subclusters, computed in the step 2 of the hierarchical clustering algorithm. Inconsistency of a link in cluster reflect how different is the link length from lengths of all the links in clusters inside; The formula is:

$$inconsistency(c) = \frac{height(c) - mean_{s \in subclust(c)}(height(s))}{stddev_{s \in subclust(c)}(height(s))}$$

Where $subclust(c)$ is set of all subclusters of c, including the cluster c. Solutions in a cluster are partitioned into subclusters if the inconsistency of the cluster is greater than some fixed limit. Partitioned cluster is naturally also removed from its parent cluster. Although we do not directly specify the number of clusters, that number will depend on the limit for inconsistency limit to split a subcluster node.

3.3.3 Problems with clustering

Clustering algorithm used on points in space with multiple continuous dimensions have to use some distance measure to decide which cluster does a point belong to. Defining a distance measure over different dimensions may be tricky, because sometimes the "units" (in the physical sense) do not correspond. Consider, for example, three points in 2D space $weight \times time$: $A = (2kg, 10s)$, $B = (4kg, 10s)$, $C = (4kg, 20s)$. Is $distance(A, B) < distance(B, C)$? The answer depends if 2 kilograms is more than 10 seconds, which is undecidable and any solution is artificial not natural.

There are several different reasonable ways to resolve this difficulty:

1. Map all dimensions intervals to canonical intervals $[-1, 1]$ and use the numeric values.
2. If we know maximal precision for each dimension, we may use it as the basic unit.
3. For the current population, compute the smallest bounding axis-aligned hyper-rectangle. We may then map the hyper rectangle to canonical intervals $[-1, 1]$ and use the numeric values.
4. "Ostrich solution": Use any numeric value "as is" without any modification or mapping

We generally discourage using solution 4, because very different scales in each dimension may cause that only larger dimensions will be practically taken into account. We will use the solution 1 in this thesis, although there is no reason to discourage the other two.

When clustering points in discrete space, we have to define a distance measure that would respect qualitative discrete values. We propose using the Hamming distance (for each dimension, sum up 0 if values equal, 1 if not).

Another problem arises with usage of the $NULL$ values. In discrete case, $NULL$ may be considered a special value for computation of the Hamming Distance. However, in continuous space, there is undefined distance between $NULL$ and non-$NULL$ values.

3.4 Brief overview of EDAs

In section 3.1 we have presented two basic evolutionary algorithms, the Simple genetic algorithm and Evolution strategies. These algorithms are based on

mutation and crossover operators, that more or less resemble the evolutionary process observed in nature. Estimation of distribution algorithms take radically different approach to the derivation of next generation. Instead of forming parental groups, whole selected population (or big clusters of the population) are used to generate a probabilistic model for the offsprings. Bossman [7] considers the model estimation as a very special example of crossover operator.

In SGA (and derived similar algorithms), one-point or multi-point crossovers are used, usually preserving parts of the chromosomes that are close together. Contrarily, genetic attributes encoded at distanced positions are likely to be destructed when the crossover point is in between. This phenomenon is known as "Genetic linkage".

Feature of EDAs is that they do not use the local linkage of genome strings; instead they model linkages among attributes by observing correlation between attribute values of the fittest individuals. Related variables tend to be correlated, and unrelated variables produce correlation merely accidentally.

The algorithm of EDAs is fairly similar to the classic genetic algorithms:

1. Initial population is generated. It may be generated completely random, or there may be rules favoring some attribute values or combinations, or it may be provided by some concrete design.

2. Fitness of the individuals is evaluated. Note that fitness may be decided by some analytic benchmark function, but it can be also derived from real-life experiment.

3. Selection is applied. The fittest solutions are likely to be selected, the less fit are likely to be eliminated. The selection is usually randomized, thus even bad solutions have slight chance of being selected.

4. **Based on the selected individuals, parameters of probabilistic model is estimated. The model usually consist of the 1) structure and 2) parameters of the structure. Set of possible structures of the model depends on the concrete EDA.**

5. **New generation is sampled by the estimated parametrized probabilistic model. Few samples may survive from the previous generation by using elitism (see section 3.1.3)**

6. Repeat process from 2.

As we can see, the only difference is in the reproduction phase in steps 4 and 5 (marked bold). During the iterative optimization process, each step of EDA usually produces a generation represented by the population - algorithm usually does not store any meta-information. Exception to this rule is for instance the algorithm PBIL, that considers result of an iteration to be the estimated probabilistic model, instead of the population.

That is basic introduction of the EDA algorithms. We will now introduce some discrete and continuous EDAs and their respective probabilistic models.

The notation used is based on notation used in [12], but may be changed occasionally. For exact reference see section Used Notation at the end of the thesis.

Shortly about the implementation details. Please note that the presented algorithms (unless stated otherwise) are existing published evolutionary algorithms and the author of this thesis does not claim authorship. Their description is however necessary to document implemented methods in the application, or were at least candidates to be implemented.

3.4.1 UMDA / UMDAc - Univariate Marginal Distribution Algorithm

UMDA is perhaps the simplest of the algorithms from the EDA family. UMDA has been proposed by Pelikan [14]. UMDA uses univariate distributions separately in each problem dimensions and does not model any interactions between variables. If a problem is linear or nearly linear, UMDA is highly efficient.

UMDA has discrete and continuous variants (UMDAc) and both estimators are quite straightforward. First, the discrete case: Let N is number of samples and N_{ik} is number of samples having k-th value from its discrete value set: $N_{ik} = \left|\left\{x^w \in D,\ x_i^w = x_i^{(k)}\right\}\right|$. Then estimated distribution is:

$$p_l(X_i = x_i^{(k)}) = \frac{N_{ik}}{N}$$

In continuous case:

$$f_l(X_i) = \mathcal{N}(m_i, v_i)$$

$$m_i = \frac{1}{N}\sum_{w=1}^{N} x_i^w$$

$$v_i = \frac{1}{N}\sum_{w=1}^{N} (x_i^w - m_i)^2$$

Simplicity of the algorithm and small number of distribution parameters make its performance comparable to even very sophisticated algorithms (results by Bosman [7]). This is also quite notable when only very small number of fitness function evaluations are available. However, when the problems are more complex and more function evaluations are available, other algorithms tend to outperform UMDA.

3.4.2 PBIL(PBILc) - Population Based Incremental Learning

The discrete variant of PBIL was first introduced by Baluja[2]. The PBIL algorithm is close to UMDA in the way the probabilistic model is defined. There are no interactions between variables. The interesting feature, unique among the EDA algorithms is that it PBIL does not optimize populations, but the distributions. In each generation, set of parameters is created, describing model of the distribution.

Let's first look at the discrete case. Distribution in generation t is by parameters $p_l^t(X)$. The iteration follows:

- Population of size M is generated by the distribution $p_l^t(X)$
- Fitness value is computed for each individual
- Best N individuals are selected (Truncation selection method)
- Distribution of the selected samples $p_\Delta^t(X)$ is computed the same way as it is learned in UMDA.
- New distribution is computed by following "Hebbian" rule:

$$p_l^{t+1}(X) = (1-\alpha) \cdot p_l^t(X) + \alpha \cdot p_\Delta^t(X)$$

Parameter $\alpha \in (0, 1]$ is learning rate. Bigger the α is, more quickly the algorithm forgets previous distributions. For $\alpha = 1$, the algorithm is equivalent to UMDA.

Now we are going to present continuous variant of the algorithm. PBILc was introduced by Sebag and Ducolumbier [15]. The modeled distributions are uncorrelated marginal normal (Gaussian) distributions $\mathcal{N}(m_i, v_i)$ as in the case of UMDAc. The learning process of distribution means m_i takes in account both positive and negative samples. Let m_i^t and v_i^t stands for learned mean and variance in iteration t. The algorithm is as follows:

- Population of size M is generated by the distribution at iteration $\mathcal{N}(m_i^t, v_i^t)$
- Fitness value is computed for each individual
- Best 2 individuals and worst 1 individual are selected
- New distribution is computed by following "Hebbian" rule, using both positive and negative samples:

$$m_i^{t+1} = (1-\alpha) \cdot m_i^t + \alpha \cdot \left(x_i^{best1} + x_i^{best2} - x_i^{worst}\right)$$

The tricky part is estimation of variances v_i. Several methods have been proposed:

1. Use constant variance for all the generations. Drawback is that the algorithm can not well adjust when the optimum is close.
2. Use evolved metadata as in evolution strategies (section 3.1.2)
3. Use variance of K best individuals in the generated population

$$v_i^{t+1} = \frac{1}{N} \sum_{w=1}^{K} (x_i^w - m_i)^2$$

4. Use "Hebbian method" for K best individuals in the population:

$$v_i^{t+1} = (1 - \alpha) \cdot v_i^t + \alpha \cdot \frac{1}{N} \sum_{w=1}^{K} (x_i^w - m_i)^2$$

PBIL is specific member of EDA family, sometimes not even considered to be pure EDA.

For the purposes dictated by the business specification, this algorithm is unsuitable. The interface for algorithm interaction does not support storing distributions instead of populations. When data are sent for fitness evaluation, additional storage for distribution memory would be necessary. We show this algorithm only for reference.

3.4.3 EBNA - Estimation of Bayesian Network Algorithm

Presented by Etxeberria and Larrañaga [11], EBNA is a discrete problem solving EDA that uses the idea of learning the data distribution using structure of Bayesian networks. EBNA represents a typical example of EDA, using steps from section 3.4. In the phase of distribution estimation, the greedy algorithm for learning Bayesian network presented in section 3.2.1 is used.

Briefly, distribution is described by structural parameters and by probabilistic parameters. Structural parameters describe which dependencies between dimensions are reflected in the model, which is the shape of the Bayesian network. Each dimension has set of "parental" dimensions that it is derived from. Parental dimensions are typically selected using penalized score metrics. Only the most significant dependencies are modeled, though. Penalization is used to limit number of parents (way to keep low number of parameters that have to be estimated), and upper limit to number of parents may be set. Parameters of the network are estimated during the computation of structure.

According to Larrañaga [12], EBNA algorithm performs well if compared to other algorithms and also scales well if the number of dimensions is high. Note that the model (probability for each value and each parent value combination) allows for a large variability of "shapes" of the estimated distributions, as value distribution on a certain node are estimated independently for each combination of parental values.

For further reference on Bayesian networks, see section 3.2.1.

3.4.4 EMNA - Estimation of Multivariate Normal Distribution

EMNA is algorithm that learns model of multivariate normal distribution in continuous space. While UMDA does not model any interactions between problem variables, EMNA models all of them, i.e. it learns whole covariance matrix of the multivariate normal distribution:

$$f_l(X) = \mathcal{N}(m, C)$$

For learned vector of means m and covariance matrix C. This yields excessive number of parameters when the number of dimensions grow; For each node, there is mean, variance and covariances with all other nodes. The total number of parameters is thus $2n + \binom{n}{2}$. The computation is however very simple:

$$m_i = \frac{1}{N} \sum_{w=1}^{N} x_i^w$$

$$C_{ij} = \frac{1}{N} \sum_{w=1}^{N} \left(x_i^w - m_i\right) \left(x_j^w - m_i\right)$$

The number of parameters to learn causes that EMNA does not scale well if the problem dimensionality is high, which is reason why this algorithm is not as popular compared to other EDAs (results by Larrañaga [12]). The algorithm EGNA presented in the following section will address this issue and will represent a adaptive but lightweight alternative to EMNA.

3.4.5 EGNA - Estimation of Gaussian Network Algorithm

EGNA, presented by Etxeberria and Larrañaga [11] is a "continuous counterpart" to the discrete algorithm EBNA. In this case, probabilistic model used is Gaussian Network. Structural parameters are treated in the same way as they are in EBNA. They are used to describe dependencies between dimensions in the model. Dependencies are incomplete, each dimension has set of "parental" dimensions that it is derived from. Parental nodes are typically selected using penalized score metrics. Penalization occurs in order to limit number of parents (therefore limit number of parameters that have to be estimated). Probabilistic parameters quantify the distribution at each dimension. Model uses univariate normal distribution, although parents influence distribution of child node by shifting its mean. The distribution in a dimension is:

$$f_l(X_i) = N\left(m_i + \sum_{X_j \; in \; Pa_i} b_{ij} \left(x_j - m_j\right)\right)$$

According to Larrañaga [12] and Bosman [7], EGNA performs well and is more robust regarding the number of dimensions. It is largely due to the fact that, unlike EMNA, number of learned parameters does not scale quadratically with the problem dimensionality. As for possible shapes of the distribution, it is in the end always a multivariate normal distribution. In another words, it is always an ellipse that may be rotated in space.

To be more precise about EBNA and EGNA algorithms, penalized maximum likelihood score is not the only used option to generate structure. For example, Bayesian score estimators and edge exclusion techniques may be used. Larrañaga [11] presents detailed view on these methods.

3.4.6 MIMIC - Mutual Information Maximization for Input Clustering

DeBonet et al. [6] presented EDA algorithm modeling bivariate dependencies between problem variables. In MIMIC, these dependencies are constructed in string. The algorithm will find permutation p, such that direct dependencies are between nodes $X_{p_1} \to X_{p_2}$, $X_{p_2} \to X_{p_3}$ ..., $X_{p_{n-1}} \to X_{p_n}$. The permutation is constructed step-by-step by greedy algorithm, trying to find node X_{p_i} so that entropy of $h(X_{p_1})$ is minimized, and then following nodes so that the entropy based on values of previous node $h(X_{p_{i+1}}|X_{p_i})$ is minimized.

The entropy is in discrete case defined as (first equation for the first node, second for node X_i with parent X_j):

$$h(X_i) = -\sum_{k=1}^{r_i} p(X_i = x_i^{(k)}) \cdot \log p(X_i = x_i^{(k)})$$

$$h(X_i|X_j) = \sum_{l=1}^{r_j} h(X_i|X_j = x_j^{(l)}) \cdot p(X_j = x_j^{(l)})$$

$$h(X_i|X_j) = -\sum_{l=1}^{r_j}\sum_{k=1}^{r_i} p(X_i = x_i^{(k)}|X_j = x_j^{(l)}) \cdot \log p(X_i = x_i^{(k)}|X_j = x_j^{(l)}) \cdot p(X_j = x_j^{(l)})$$

using the usual notation $x_i^{(k)}$ being k-th value of variable X_i and r_i being number of values in discrete value set of X_i. Note the similarity to maximal likelihood of the structure defined by estimators θ_S over structure S. (see section 3.2.1 about more information on Bayesian networks and notation). Let $L_i(D|S, \theta_S)$ stands for likelihood for variable X_i, for data D and linear MIMIC structure S:

$$\log L_i(D|S, \theta_S) = \sum_{k=1}^{r_i} N_{i \bullet k} \cdot \log\left(\frac{N_{i \bullet k}}{N}\right) = N \cdot \sum_{k=1}^{r_i} \frac{N_{i \bullet k}}{N} \cdot \log\left(\frac{N_{i \bullet k}}{N}\right)$$

$$\log L_i(D|S, \theta_S) = \sum_{k=1}^{r_i}\sum_{l=1}^{r_l} N_{ilk} \cdot \log\left(\frac{N_{ilk}}{N_{il}}\right) = N \cdot \sum_{k=1}^{r_i}\sum_{l=1}^{r_l} \frac{N_{ilk}}{N_{il}} \cdot \log\left(\frac{N_{ilk}}{N_{il}}\right) \cdot \frac{N_{il}}{N}$$

The MIMIC algorithm was also extended to continuous form MIMICc. The node structure algorithm is exactly the same as in discrete case, only the expression for entropy is redefined to continuous space [12]:

$$h(X_i) = \frac{1}{2}(1 + \log 2\pi) + \log S_{X_i X_i}$$

$$h(X_i|X_j) = \frac{1}{2}(1 + \log 2\pi) + \log\left(S_{X_i X_i} - \frac{S_{X_i X_j}^2}{S_{X_j X_j}}\right)$$

MIMIC algorithm can be considered bivariate modification of the multivariate algorithms EBNA and EGNA. Major difference is that in EBNA/EGNA, ordering of nodes is fixed from the start and algorithm selected parents for each node. In MIMIC, algorithm decides ordering of the nodes and parents are always the last node in the ordering.

3.4.7 Mixture models

Weakness of all previously listed algorithms is that if they focus on a single area:

- Firstly, if the high-fitness area has complicated shapes, the Gaussian distributions will have big difficulties modeling them.
- Secondly, if there are multiple isolated islands of high fitness, standard algorithms will have to focus only on one of them.
- Thirdly, the mixture models address the problem of symmetry. Consider continuous one-dimensional problem with fitness function defined as $fitness(x) = |x|$ for $x \in (-10, 10)$ and starting population distributed closely around 0. Increasing fitness on the "positive" side drags the population in opposite direction to the drag of increasing fitness on the "negative" side. Consequently, the two optimization forces may effectively nullify each other and slow down convergence considerably.

To add flexibility to the standard models, mixture models have been introduced by Pelikan and Goldberg [13]. Mixture model distribution is a union of independent sub-distributions:

$$pf_l(X) = \sum_{c=1}^{C} \pi_c \cdot pf_{l,c}(X)$$

Where $pf_{l,c}$ are the sub-distributions and π_c their respective weights. Their model is flexible - they may be any of the models of previously listed EDAs - both discrete and continuous. The key factor that assures that right samples participate on learning corresponding sub-distribution is clustering. When the population is evaluated and high fitness solutions are selected, the selected individuals are divided into clusters and distribution is learned separately for each cluster. Weight of clusters is determined by number of individual forming that cluster.

The clustering may be carried out using various clustering techniques. Two examples of clustering algorithms are K-means clustering and hierarchical clustering. More detailed information on clustering was provided in section 3.3.

Obvious drawback of the clustering is the fact that the more regions are being explored in parallel, the less samples are available to efficiently explore each single region.

3.4.8 Histogram EDA

Bosman [7] presented probability model based on histograms. Continuous problem space is divided into several bins in each dimension. Since number of bins is given, width of bins in different dimensions may vary accordingly to value range for that dimension. With greater number of bins in a dimension, better precision may be achieved. Speaking about bounds between dimensions, theoretically the maximum likelihood estimate may be used to compute probability

in every n-dimensional cube in solution space. This is highly impractical, because number of cubes grows exponentially with the problem dimensionality. Consider 7-dimensional problem and 10 bins in each dimension (which are both very small requirements). Number of bins would be in such case $10^7 = 1000000$, which is probably far more than allows our budget for number of fitness function evaluations for whole algorithm run. Bosman [7] argues that only univariate version (without any variable dependencies) may succeed to produce any positive results.

4 Optimization model

In section 4 we have only introduced a very basic definition of the solution space. That model is very simple and its key advantage is that most of the proposed EDA algorithms were designed for . Definition and handling of different structures is one of the key points of this thesis.

In this section, we will describe our optimization models and ways to optimize solutions in it. In section 4.1, we will present general metamodel, that defines possible optimization structures, whose instances are being optimized. In section 4.2, we will describe specific constraints, that the optimization structures may define and we will present algorithms to enforce these constraints. In section 4.3, we will propose an automated way of classification of solutons. These methods will be used to decide validity of solutions and will make necessary corrections, so that malformed solutions coud be repaired. Section 4.4 will shortly describe special types and characteristics of variables used in optimization model.

4.1 Solution space model and metamodel definition

Each problem can define its own data model, and each of the models is instance of a common metamodel, which we will call the **optimization metamodel**. The metamodel is defined as a business requirement and can not be changed, because it is a part of a defined interface. However, the optimization metamodel is very general and widely reusable.

Instance of the optimization metamodel is also called the Optimization structure and is provided to the algorithms as a parameter. The optimization structure basically defines nodes and oriented edges between the nodes via references. The class diagram representing the optimization metamodel is shown on figure 1.

Not all instances of the optimization metamodel are valid optimization structures. There are following rules that an optimization structure must comply with:

- Each node has a unique text identifier
- The oriented graph of connected nodes must not contain oriented circles.
- The model must define exactly one **root** node.

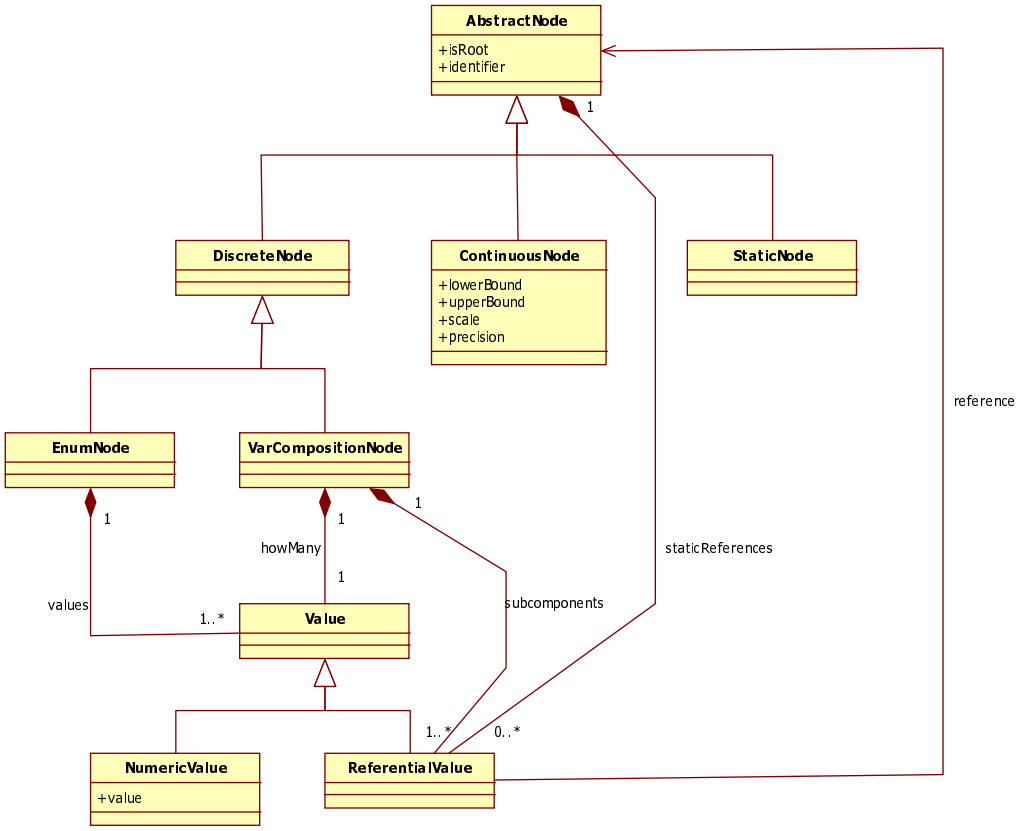


Figure 1: Optimization metamodel

- All non-root nodes in the structure are reachable by oriented path originating from the root node.

Most but not all considered optimization structures will be tree-like and thus we will use tree nomenclature: For a node, all nodes connected by forward edges are its **child nodes**, nodes connected by backward edges are its **parent nodes**. A node may have more than one parent. Root node has no parents and is the only parentless node. Important property is that nodes may be **topologically ordered**. **Topological rank** of the root node is 1, topological ranks of all child nodes are always greater, topological ranks of all parent nodes are always smaller.

Population in the evolution algorithm is instance of the optimization structure. Population consists of zero or more individuals and each individual is graph of connected **node instances**. Structure of node instances copies structure the optimization structure, although an individual may have some nodes

uninstantiated. Rules for instantiation of nodes are following:

- Nodes are never repeatedly instantiated (for one individual).
- Root node is always instantiated.
- Other nodes are instantiated if and only if they are referenced by instance of another node of the individual.

Node instances have properties depending on its type:

- Any Node may define static references to other nodes meaning its instance will always reference instances of all these nodes. The metamodel defines five types of static references: static composition, proportion, preparation, lowerBoundsReference, upperBoundsReference.[1]Node may have many static composition staticReferences and at maximally 1 staticReference of each other type. The types of staticReferences differ in business interpretation, although evolution algorithms will treat those references in the same way. Some nodes may be **numerically evaluable**, which is not to be confused with actual "value" or syntactic representation of instance of a node. **Numerical evaluation** is a real value that represents actual value of a particular measure that a node corresponds to. Not all nodes are necessarily numerically evaluable. Numerical evaluation may be derived from numerical evaluations of other nodes via chains of **references**.
- RealNode instance contains a real value from the closed interval $(lowerBound, upperBound)$. RealNode is always numerically evaluable and numerical evaluation of its instance is the selected real value. The RealNode attribute intervalParameter is a "business" information indicating used scale, that can be linear or logarithmic. Parameter precision defines maximal resolution for the variable. Precision does not function like a constraint (effectively turning real variable into discrete), it is merely a hint to the algorithms when more precise values are unwanted.
- EnumNode instance contains selection of exactly one of the values defined by the EnumNode. Values of EnumNode are either numeric or referential, in latter case the node instance references another node. EnumNode may or may not be numerically evaluable. An EnumNode is numerically evaluable if all of its enumValues are either NumericValues or ReferentialValues that all reference numerically evaluable Nodes. Numerical evaluation of the EnumNode instance is value of the selected NumericValue, or numerical evaluation of the node instance referenced by selected ReferentialValue.

[1]In the original specification, upperBounds and lowerBounds of RealNodes are defined as either NumericValue or ReferentialValue. In order to simplify the models, we reinterpret the lower and upper bounds as numeric attributes - for NumericValue bounds; Or as StaticReferences of type lowerBoundsReference (or upperBoundsReference respectively) plus numeric attributes equal to the least possible numerical evaluation of the referenced node - for ReferentialValue bounds. Least possible numerical evaluation is obtained recursively by iterative assessing of minima and maxima of referenced nodes - see numerical evaluation of enum nodes.

Note that this definition is recursive and may result in chain of references. Because the metamodel graph is acyclic, the chain is also acyclic and eventually ends at some node. If all possible sources of numerical evaluations of EnumNode instances are enum nodes with exclusively integer values, the node is **integer-evaluable**.

- VarCompositionNode instance contains combination of *howMany* subcomponents from all Node subcomponents (i.e. the value of the composition is subcomponent subset and the composition is not numerically evaluable). The *howMany* count is either integer NumericValue or numerical evaluation of ReferentialValue referencing instance of another node. In the case of a howMany reference, the referenced node must be integer-evaluable. VarComposition stands for "variable composition", because each individual may have different subset of the selected subcomponent nodes, as opposed to the static compositions where all individuals always statically reference whole set of subcomponents. Size of the subset of selected subcomponents by a variable composition is either constant (in case numerical *howMany* count) or variable (in case of referential *howMany* count).

VarCompositionNodes and StaticNodes are never numerically evaluable. An EnumNode may reference other referencing EnumNodes, thus forming a chain of retaking numerical evaluations, that eventually ends at a node without references which we refer to as **source of numerical evaluation** (i.e. source is the referenced node, not the referencing ones). **Source index** of a node is number of EnumNodes for which this node may become source of their numerical evaluation.

There problem may define additional constraints on the individuals, that will be described in the following chapter.

We have now defined individuals as graph of node instances. However, existing evolution algorithms are usually interfaced for numeric vectors, from either integer or real domains. Because we want to reuse these algorithms with as little modifications as possible, we will also represent individuals or populations in this way; A simple mapping used to represent node instances to numbers and vice versa is following:

- StaticNode instances are represented as value 0.
- EnumNode instance is represented as index of selected value.
- VarCompositionNodes are represented as integer value that is index of the selected combination from sorted list of all possible combinations (of any length) from the powerset of its subcomponents (e.g. $1 = \{\}$, $2 = \{first\}$, $3 = \{second\}$, $4 = \{first, second\}$, $5 = \{third\}, ...$ for subcomponents $\{first, second, third, ...\}$).
- RealNode instances as represented as the selected real value.

- If a node is undefined, it is represented as $NULL$, which is a special value different from all integer and real numbers. Note that $NULL$ value of VarComposition is different from value indicating empty combination and different from representation of instantiated StaticNode.

Using this representation, we define variable X_i for each node N_i of the optimization structure, and we define its value set straightforwardly according to the node properties. Each variable value set X_i has at least two values, one $NULL$ and at least one $non - NULL$. The solution space X of a problem is subset of Cartesian product $X'$of variables X_i:

$$X \subseteq X' = X_1 \times X_2 \times ... \times X_n$$

Number n denotes number of problem dimensions or simply **problem dimensionality.** Each variable X_i is associated with one node of the optimization structure. Depending on the class of the corresponding node, value set of its variable X_i is discrete when the node is instance of DiscreteNode or continuous when the node is instance of RealNode.

Lastly, we would like to present alternative approach to variable compositions:

Decomposition: So far we presented variable composition as a single discrete variable with value for any combination of the subcomponents. We also experimented with decomposition of such variable to list of binary discrete variables, each representing inclusion of one subcomponent of the variable composition. Value 1 represents that the subcomponent is included, 0 that the subcomponent is excluded. The decomposition only takes place when the distribution is estimated by the evolutionary algorithm. Usage of decomposition of variable composition nodes will be considered as boolean parameter of the evolutionary algorithm.

Decomposition is an alternative implementation of composition handling and may be used optionally.

4.2 Solution space constraints

As stated in the last chapter, the representation of solution space X is defined as subset of Cartesian product X'. More specifically, X contains all values of X' that satisfy some additional constraints. In particular, there are three types of constraints:

1. Nullness. As said before, node is $NULL$ if and only if is not root and is not referenced by any other node of the individual. $NULL$ node does not reference any other node, not even by static references. Note that occurrence of $NULL$ values in individual depends entirely on two factors: values of variable compositions and referential values of enums. All other references have static character and only "forward" non-nullness towards their child nodes.

2. Variable compositions. A $Non-NULL$ variable composition must select subset of subcomponents of the size $howMany$. $howMany$ is either numeric value or numerical evaluation of another node. Note that howMany referenced node of a $non-NULL$ variable composition is by definition never $NULL$, because $howMany$ reference is already a type of a static reference.

3. Linear constraints. Solution $x \in X'$(x is column vector in this case) satisfies linear constraints iff

$$A \cdot numEvaluation(x) \leq a$$

where A is a $m \times n$ linear inequality matrix and a is right side inequality vector consisting of m rows. Note that only numerically evaluable nodes may be constrained by linear inequalities. Upper and lower bounds of RealNodes are always part of linear constraints but there may be additional rules. Function $numEvaluation$ assigns numerical evaluation of the node instance if the node is numerically evaluable or zero for numerically non-evaluable or $NULL$ nodes. We only define linear constraints as inequalities "less than or equal". Linear inequality "greater than or equal" can be obtained by multiplying both sides by-1. Linear inequalities of type "less than" and "greater than" (excluding equal) are not supported. Linear equation ("equals") constraints can be expressed as two inequalities:

$$A_{i\bullet} \cdot numEvaluation(x) = a_i$$

$$\begin{pmatrix} A_{i\bullet} \\ -A_{i\bullet} \end{pmatrix} \cdot numEvaluation(x) \leq \begin{pmatrix} a_i \\ -a_i \end{pmatrix}$$

each row of the inequality matrix A describes one linear constraint. To be precise, rule in row $A_{i\bullet}$ is not evaluated at all if x has $NULL$ values at all positions corresponding to non-zero coefficients of the row $A_{i\bullet}$.

Linear constraints are part of the optimization model. The metamodel presented in the previous section thus additionally contain two classes. Class LinearConstraint represents one linear constraint. LinearConstraint aggregates nodes (that must be numerically evaluable) and for each of the aggregated node, relational class LinearCoefficient describes coefficient of the node in the linear inequality. Parameter $lessThan$ of the LinearConstraint describes the "right side" of the inequality, that is maximal allowed value of the linear combination of node values.

4.2.1 Node allocations

We define **allocation** as a set of all valid solutions, that are defined by having certain values (including nullness) on the allocated nodes and having any values on the non-allocated nodes. If there is at least one solution for an allocation that satisfies rules for nullness and variable composition (regardless of linear constraints), it is **constructive allocation**. If no solutions with the exact defined

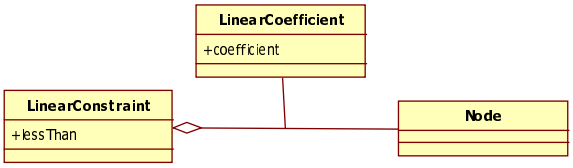


Figure 2: Metamodel classes for linear constraints

values on allocated nodes satisfy rules for nullness and variable composition, it is **inconstructive allocation**.

If there is at least one solution of an allocation that satisfies linear constraints, it is **feasible allocation**. If no solutions with the exact defined values on allocated nodes satisfy linear constraints, it is **infeasible allocation**.

If an allocation is both feasible and constructive, it is **valid allocation**. Otherwise, it is **invalid allocation**.

Important type of allocations are **discrete allocation,** that are defined by values of all discrete nodes.

Observation 1: discrete allocations already contain enough information to decide nullness of all model nodes, including nullness of real and static nodes. As stated in previous section, nullness is entirely determined by static references, by referential values of enums and by variable compositions. Static references are already defined by the model, while enums and varCompositions are both subclasses of discrete nodes, thus both included in discrete allocations.

Observation 2: discrete allocations partially define numerical evaluations of all numerically evaluable model nodes:

- where the source of numerical evaluation is numeric value of EnumNode, discrete allocation defines the exact numerical evaluation
- where the source of numerical evaluation is value of a real node, discrete allocation identifies the source real node
- where the node is real, it is trivially its own source
- in any other case, the node is not numerically evaluable

Only enumNodes form chain of references retaking numerical evaluation from other nodes; Discrete allocation defines values of all enum nodes, thus providing us information about the source.

Lemma: all solutions in a discrete allocation equally satisfy (or not satisfy) rules for nullness and variable compositions.
Nullness of all nodes is defined, as stated in Observation 1, equally for

all solutions in the discrete allocation. Selected subcomponent subsets and numerical evaluation of referential $howMany$ values of variable compositions are, by Observation 2, already fixed - The source of $howMany$ numerical evaluation is never a real node, because $howMany$ must be by definition integer-evaluable. Thus, all solutions in a discrete allocation have equally satisfied rules of variable compositions.

If allocation L_1 allocates all nodes allocated by allocation L_2 and possibly more, then we say that L_2 is **suballocation** of L_1 , but from perspective of sets $L_1 \subseteq L_2$.

Note: The original geometric interpretation of the valid solution space is union of several polyhedra in different spaces, each space with different number of dimensions according to which nodes are defined. In our representation, we model the missing dimensions as $NULL$ values. Solution space is then union of polyhedra in one big multidimensional space where each dimension also allows the special $NULL$ value. Exact shape of the polyhedra is determined by upper and lower bounds of real nodes and by linear constraints. Each discrete allocation corresponds to one polyhedron.
Given the fact that discrete values determine nullness of all nodes (Observation 1) and define all referential values of the solution (Observation 2), all the remaining $non - NULL$ real nodes are only constrained by set of linear equations and inequalities (upper and lower bounds being also linear inequalities). The delimited space is thus convex polyhedron or empty set. Discrete allocation is feasible exactly when its corresponding polyhedron is not empty.

4.2.2 Linear constraint satisfaction considerations

EDA Algorithms in their raw form as proposed in the scientific papers are not adapted to this type of constraints. There are several ways to resolve this issue:

- Simple approach is to eliminate all solutions that do not satisfy constraints. This may be a reasonable approach when there are only linear inequalities and majority of the solutions are likely to satisfy the constraints; Consider points in three dimensional space (x_1, x_2, x_3) that have to satisfy $b - \epsilon \leq x_1 + x_2 + x_3 \leq b + \epsilon$. For example, a three dimensional non-correlated (spherical) probabilistic model will put solution away from the tightly constrained zone with probability close to 1 when ϵ is close to 0, thus generating nearly all broken solutions.

- Another solution is to place a fitness penalty to all solutions that are not satisfying the constraints, the bigger the penalty the more away from the constrained space it is. However, the problems of inequalities persists, it is only transformed to problem of the algorithm solving primarily constraints satisfaction instead of constrained optimization.

- Another method is to include constraints directly into the probabilistic models generated by EDAs, but this may result an overcomplicated approach. (Consider adapting probabilistic model to fit into, for instance, a zone shaped as triangle)

- Fourth method is generate solution $x' \in X'$ and correct all constraints violations afterward. Solution $x' \in X'$ is thus converted to new valid solution $x \in X$ so that x is as close to x' as possible.

Taking into account advantages and disadvantages of each approach, we decided we will be using the last of the methods for correcting invalid solutions in this thesis.

4.2.3 Constraint satisfaction algorithm

Assume that we have a solution $x' \in X'$ that is sampled with no $NULL$ values. (it is important that the evolution algorithms do not generate them). We will proceed with three steps, correcting each type of constraints:

1. First, we enforce the variable composition sizes. Variable composition instance of individual that do not select exactly $howMany$ subcomponents must be repaired: we must additionally select or deselect subcomponents to reach the actual $howMany$ value. The exact algorithm is based on the idea of estimation of distribution. We compute probability of selection for each of the subcomponents among the population. For the repaired individual, Subcomponents are additionally selected or deselected randomly, with the estimated probability serving as weight of selecting or deselecting each one.

2. Then, we assess $NULL$ values. First, all variables of the individual except root are considered $NULL$. Then, we process nodes in topological order; If the node is not considered $NULL$, we set all nodes it references to $non-NULL$. This way, $NULL$ and $non-NULL$ values are propagated throughout the whole graph. After this second step, all solutions should be already constructive.

3. Lastly, linear constraints are repaired (if possible). Let's assume that x is constrained vector of real variables. When fixing linear constraints, the matrix A of linear constraints is reduced to coefficients (columns of A) corresponding to $non-NULL$ real values. Also the vector $x'$is reduced to $non-NULL$ positions. By eliminating $NULL$ positions, we do not change satisfaction of the inequalities, but we exclude the $NULL$ positions from corrections, as $NULL$s can not be corrected. All rows of the matrix A where there are no remaining nonzero coefficients are removed, together with the corresponding right side of the inequality in vector a. Remaining rows in A are normal vectors of the hyperplanes of the linear limits. Linear programming is used to determine smallest (in euclidean metric) non-negative linear combination $shift$ (vector of size m) of the normal vectors

A^T that added to the original solution x' yields a correct solution x:

$$x = x' + \left(-A^T\right) \cdot shift$$

$$A \cdot x \leq a$$

$$shift \geq 0$$

which implies:

$$\left(-A \cdot A^T\right) \cdot shift \leq a - A \cdot x'$$

$$-shift \leq 0$$

These inequalities may be processed by the linear programming procedure, setting the weight scalar w being minimized as:

$$w = shift \cdot \left(\sqrt{\sum_{i=1}^{n} A_{ji}^2} \right)_{j=1,..,m}$$

By solving the linear programming problem, we obtain solution from the same discrete allocation (i.e. with changes made only on the real nodes).

Note that we use value of x instead of $numEvaluation(x)$. So far we have only defined step 3 for real variables and we take advantage of the fact that for real variables, values are equal to node numerical evaluations. The linear correction step is considerably more complicated when there are constrains for enum variables. Enums can be always "eliminated" from the linear constraints by simple arithmetic operations:

- Numerical evaluations of numeric enum x_k may be subtracted from the right side a after multiplication with corresponding coefficients in A:

$$\sum_{i \in I} A_{\bullet i} \cdot numEvaluation(x_i) \leq a_j \iff$$

$$\iff \sum_{i \in I}^{i \neq k} A_{\bullet i} \cdot numEvaluation(x_i) \leq a_j - A_{\bullet k} \cdot numEvaluation(x_k)$$

- Coefficients $A_{\bullet k}$ of referential enum x_k referencing node x_{ref} may be added to corresponding coefficients $A_{\bullet ref}$ at column for x_{ref}:

$$\sum_{i \in I} A_{\bullet i} \cdot numEvaluation(x_i) \leq a \iff$$

$$\iff (A_{\bullet k} + A_{\bullet ref}) \cdot numEvaluation(x_{ref}) + \sum_{i \in I}^{i \notin \{ref,k\}} A_{\bullet i} \cdot numEvaluation(x_i) \leq a$$

In the first case, we take advantage of the fact that $numEvaluation(x_k)$ is constant. In the second case, numerical evaluation of node k is the same as numerical evaluation of the referenced node ref. In both cases, we effectively removed constraint of node k from the linear inequalities, and by multiple application of the rules (in topological order of enum variables), we always get system of linear inequalities constraining only real nodes. Correct solution of the new constraint inequalities will yield correct solution of the original inequalities.

Using this correction, we are able to repair any solution, provided that there exist any solution for the discrete allocation. The problem is, that for some discrete allocations (those are infeasible discrete allocations), the solution does not exist. It would be perfectly valid inequality one that only constraints numerical evaluations of enum nodes and no real values - in which case, any real value modification has no effect. If the solution being corrected is from feasible discrete allocation, the linear correction will always be able to correct it.

If we expect the number of infeasible solutions to be small percentage of generated solutions, we may simply eliminate them, possibly replacing them with additionally generated solutions. However, this approach is not favored by our business requirements.

If we require that all solutions must be repairable, we may add a correction step number "2.5" between steps 2 and 3:

- Test if the (constructive) discrete allocation of the individual is feasible. If the allocation is infeasible, find "closest" constructive and feasible discrete allocation, rewrite discrete values of the individual according to the new discrete allocation, keeping the real values where defined, invent rest of the real values. Continue with linear correction of real values.

(Distance of discrete allocations is measured as hamming distance, i.e. number of differently allocated nodes). However, the feasibility correction is a non-trivial task. In the following chapters, we will inspect this topic into detail and propose a way to realize this step.

4.3 Exploring validity of discrete allocations

In this subchapter, we will take a closer look at exploring validity of discrete allocations - the possibility of satisfying null rules, variable composition rules and linear inequalities. There are two major challenges:

1. expressivity of the model, that creates difficulties when we try to draw conclusions about its behavior or behavior of the parts.

2. possibly vast amount of all valid and invalid discrete allocations allowed by the model

In order to be valid, discrete allocation must be both constructive and feasible. Assessing and repairing constructiveness of discrete allocations is simple and was already explained in section 4.2.3. The hard part of the validity is deciding feasibility, alias possibility of satisfaction of linear inequality constraints.

To show why the feasibility is problematic, let's suppose following problem:

There are k boolean variables $v_1, ..., v_k$ and expression in conjunctive normal form:

$$(l_{11} \vee l_{12} \vee l_{13}) \wedge (l_{21} \vee l_{22} \vee l_{23}) \wedge ... \wedge (l_{m1} \vee l_{m2} \vee l_{m3})$$

where each $l_{ij} \in \{v_1, ..., v_k, \neg v_1, ..., \neg v_k\}$. Our task is to decide if there exist set of evaluations of variables $v_1, ..., v_k$ such that the CNF expression is satisfied. This is a well known NP-complete problem 3-SAT that is not computable in polynomial time unless P=NP. We will show that there exists polynomial reduction to a problem defined by an optimization model with linearly constrained enum values.

Let N_{k+1} is a root node, instance of class StaticNode. Let $N_1, ..., N_k$ are nodes of class EnumNode and node N_{k+1} has static references to all nodes $N_1, .., N_k$. Each of nodes $N_1, ..., N_k$ has two enumValues of class NumericValue, allowing values 1 and -1. There is a linear inequality $A.evaluation(x) \leq a$ constraining evaluations $e_1, .., e_{k+1}$ of variables $X_1, ..., X_{k+1}$ of the nodes $N_1, .., N_{k+1}$. Matrix A has m rows and $k+1$ columns; Its value A_{ij} for $j \in \{1, .., k\}$ and $i \in \{1, .., m\}$ is number of literals $\neg v_j$ in i-th clause minus number of literals v_j in i-th clause. Values of variable X_{k+1} are unconstrained, i.e. $A_{i(k+1)} = 0$ for all $i \in \{1, .., m\}$. Vector a is column vector of length m with all values equal to 2.

For example, a clause $(v_1 \vee \neg v_3 \vee \neg v_4)$ would be translated to rule:

$$\begin{pmatrix} -1 & 0 & 1 & 1 & \cdots & 0 \end{pmatrix} \cdot \begin{pmatrix} e_1 & e_2 & e_3 & e_4 & \cdots & e_{k+1} \end{pmatrix}^T \leq 2$$

Each rule is satisfied if and only if at least one of the multiplications is -1, i.e. $e_j = 1$ for literal x_j or $e_j = -1$ for literal $\neg x_j$, which occurs exactly when the according clause is satisfied. The model has feasible solutions if and only if the CNF expression is satisfiable. The model is defined by $O(m.k)$ parameters, thus the reduction to the new problem of model feasibility is undoubtedly a polynomial reduction. Thus, even deciding if a optimization model has any valid solutions is a NP-hard problem. If no heuristics is applied, we would have to search through all discrete combinations in order to discover validity of all discrete allocations.

Moreover, the number of discrete allocations may be enormous. If there is a variable composition selecting 10 out of 20 binary enum subcomponents (similar to nodes in previous examples), there are $\begin{pmatrix} 20 \\ 10 \end{pmatrix} \cdot 2^{10} = 184756 \cdot 1024 = 189190144$ combinations. The big numbers are consequence of the fact that number of all discrete allocations is product of value set sizes for values of discrete nodes. Each discrete allocation may yield a different linear problem, because of different distribution of $NULL$ values that cannot be "repaired". Generally there is no method to factorize overall feasibility to feasibility of decomposed parts. If every validity test means running linear programming routine, the task may prove to be far too expensive in terms of CPU and memory resources (for storing the results).

There is no universal method, no "silver bullet" solution to the allocation validity problem. We may, however, use several heuristics that should simplify this task if the definition of the model follows some reasonable rules. It is always a big challenge to design heuristics to be as realistic as possible, because real life scenario might prove to be far less ideal than we expect during the design stage.

After consultation of the business requirements, we have concluded following assumptions:

1. The solution space may be heavily constrained by linear inequalities, leaving only small part of feasible solutions among all possible solutions. The constraints will be usually applied to both real and enum variables. Mapping of feasibility and performing correction of solutions is essential.

2. The graph of the model tends to be mostly tree-like: few nodes will have more than one parent. Nodes with more than one parent might exist but will probably not be the bottleneck of research of allocation validity.

3. If there is a bottleneck in feasibility research, it will probably be represented by variable compositions with big number of subcomponents, although these subcomponents (and their respective subgraphs) will be usually constrained in a very similar way. This has nothing to do with the node value interpretation, it only reflects validity of the allocation.

According to these assumptions, we are going to design a system that will not decide feasibility for each possible discrete allocation separately, but for whole classes of equivalent discrete allocations. Our goal is to define a classification of discrete allocations, where allocations in a class are all feasible or all unfeasible, while keeping the number of classes as low as possible. Moreover, not all classes necessarily contain valid discrete allocations. We want to be able to identify all valid discrete allocation classes without having to test all discrete allocation classes.

Once we have such classification and have identified all valid classes, we do not need to test every valid discrete allocation, but only one sample for each of the valid classes.

4.3.1 Null dependence

$NULL$ values in solution are defined by referential values of enums and variable compositions. By definition, nodes have $non-NULL$ value if and only if they are root or they are referenced by another $non-NULL$ node instance. This rule creates dependencies between "nullness" of different nodes. We want to define property that $NULL$ value of one node is dependent on values of one or more other nodes. According to the model, nodes reference other nodes by various means, that may be variable composition, enum referential value, howMany reference or some kind of a static reference. Unlike nodes, node instances of an individual may generally reference only part of the child nodes; In case of

variable composition, its value determines which of the children are selected and which are not selected. On the contrary, for a $non-NULL$ node, all the nodes that it statically references are always "selected", regardless of the parent node value. Thus, we can define two type of references among nodes:

Static-type and variable-type child nodes: Variable-type children are child nodes referenced variable-type references, that are either referential values of EnumNode or subcomponent references of a VarCompositionNode. Instance of a variable-type child node is not necessarily referenced by the parent node. Static-type children are child nodes referenced by static-type references, namely staticReference and howMany reference of a VarCompositionNode. Instances of static-type child nodes are always "selected" by the parent node (if it is not $NULL$).

Static-type descendant is a node connected by forward references of static type. Obviously, instances of static-type descendant nodes are always $non-NULL$ if the ancestor node is $non-NULL$.

Static-type references are references that are always present in the individual - the parent may have any value unless it is $NULL$ itself. In variable-type references, existence of the actual reference among node instances is not mandatory.

The rule is not generally true in the opposite direction, if a node instance is $non-NULL$ and the node is static-type child of some parent nodes, part of parents still may be $NULL$. This is due to the fact that a node may be child of many parents and it's $non-NULL$ value secured by another path in the graph.

Observation: Given the fact that root node is always $non-NULL$, any node that is static-type descendant of the root node is always $non-NULL$.

4.3.2 Model symmetries

In this section, we will outline the general idea of proposed equivalence of different discrete allocations. The model is defined as nodes connected by references is oriented graph without oriented circles.

The basic idea is to simplify selection of subcomponents by variable compositions. In following chapters, we will establish equivalence between selection of certain nodes and we would only have to test validity for selection of one "sample" branch and reuse this information to avoid testing equivalent branches.

Goal: we want to design an equivalence that for two equivalent subcomponents N_i and N_j of variable composition N_v and for any discrete allocation L_i that allocates node N_v with selection of N_i but not N_j, there exists discrete allocation L_j that allocates node N_v with selection of node N_j but not N_i such that constructiveness and feasibility of L_i and L_j are equal. That equivalence would imply that exploring branch of subcomponent N_i will be sufficient and will already provide us with information about feasibility in branch N_j.

We will differentiate between two types of equivalences:

- equivalence of nodes in the optimization structure, that describes similarity of constraints applied to values of the nodes.
- equivalence of feasibility of discrete allocations, that ensures equal feasibility for all individuals in each equivalence class. Equivalence of nodes will be used to define and evaluate equivalence of discrete allocations.

Before we design the equivalences, we will take a closer look on how nodes are connected in the node graph and how it will influence possible equivalences. There are two ways the nodes may be anchored in the optimization structure. All nodes have to be descendants of the root node. If for a node N_i there is only one oriented path from the root to this node, then the node is **singly descended**. If there are multiple oriented paths from the root node, he node is **multiply descended** (If a child is referenced by the parent by multiple references, it is considered multiply descended). Single descendence of the nodes is the case which the heuristics are optimized for.

In following examples, we will refer to a vaguely defined "symmetry", that would imply the aforementioned goal. If the symmetry is reflexive, symmetric and transitive (A symmetric to B and B symmetric to C imply A symmetric to C), it is an equivalence relation. The examples should only explain what properties such equivalence would have to have in different scenarios.

Example 1: Let's suppose following optimization model example, where all nodes are singly descended.

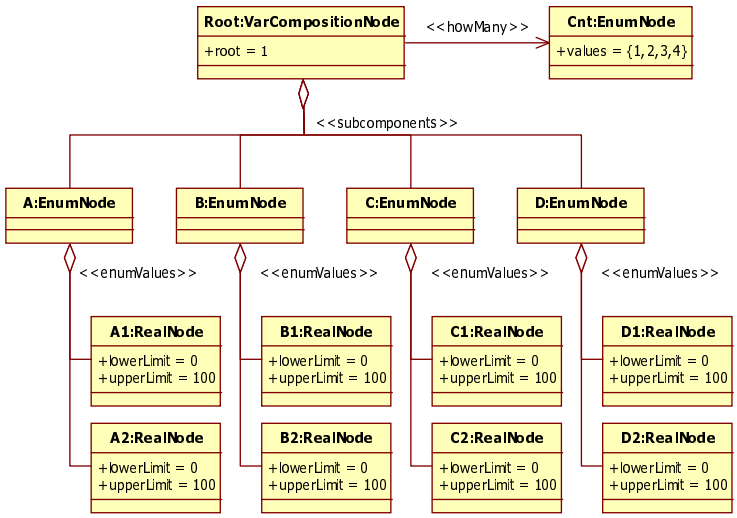


Figure 3: Example of model having singly descended nodes

We have root variable composition, selecting 1-4 child enumerations, where each enumeration chooses one of two child nodes. The diagram is simplified that

referentialValues are represented by aggregations/associations with according stereotype. There are additional linear inequalities for numerical evaluations E_A, E_B, E_C, E_D, E_{A1}, E_{B1}, E_{C1}, E_{D1}, E_{A2}, E_{B2}, E_{C2}, E_{D2} of nodes A, B, C, D, A1, B1, C1, D1. A2, B2, C2 and D2 respectively;

$$E_A + E_B + E_C + E_D \geq 150$$

$$E_{A1} + E_{B1} + E_{C1} + E_{D1} + E_{A2} + E_{B2} + E_{C2} + E_{D2} \leq 100$$

The graph is a tree, but its branches are not totally independent, because the linear inequalities creates dependencies between them. However, If we compare nodes A, B, C, D, their subtrees and constraints, we see a symmetry. For example, if the Root variable composition has to select 1 subcomponent, from the perspective of constraint satisfaction it is unimportant which one it is. It is also unimportant which of the real values are referenced by the selected subcomponents. The same applies for selection of 2 subcomponents. In fact, there are four different types of situations, that only differ in number of selected subcomponents. Unfortunately, by simple calculation we may deduct that none of the situations lead to feasible allocation. In this example, the symmetry seems to be an equivalence.

Example 2: We continue with another example, where there are nodes with multiple descendence. Multiple descendence brings additional complexity into dependencies between nodes. Firstly, there is more complicated dependence of nullness, because the node may be referenced by multiple possible paths. Secondly, the multiple descendence may complicate numerical evaluations of nodes that retake value from multiply descended sources. The example model is shown on figure 4.

We have root variable composition, selecting 1-4 child enumerations, where each enumeration chooses one of two child nodes. The diagram is simplified: referentialValues are represented by aggregations with according stereotype. There are additional linear inequalities for numerical evaluations E_A, E_B, E_C, E_D, E_{AB}, E_{BC}, E_{CD}, E_{DA} of nodes A, B, C, D, AB, BC, CD, DA respectively;

$$E_A + E_B + E_C + E_D \geq 150$$

$$E_{AB} + E_{BC} + E_{CD} + E_{DA} \leq 100$$

We want to study feasibility of different discrete allocations of this model and we want to avoid repeating feasibility tests when the discrete allocations are symmetric.

If the Root variable composition has howMany value 1, thus selecting only 1 subcomponent. If we look at the diagram, we clearly see similarities between situations when any of the nodes A, B, C, D is selected. Evaluating feasibility for one subcomponent choices is sufficient to get information about all 3 remaining choices. We may call this a symmetry, but not an equivalence. We will explain why: The situation is radically different when there are 2 subcomponents selected: combination of subcomponents A-B allows that both subcomponents

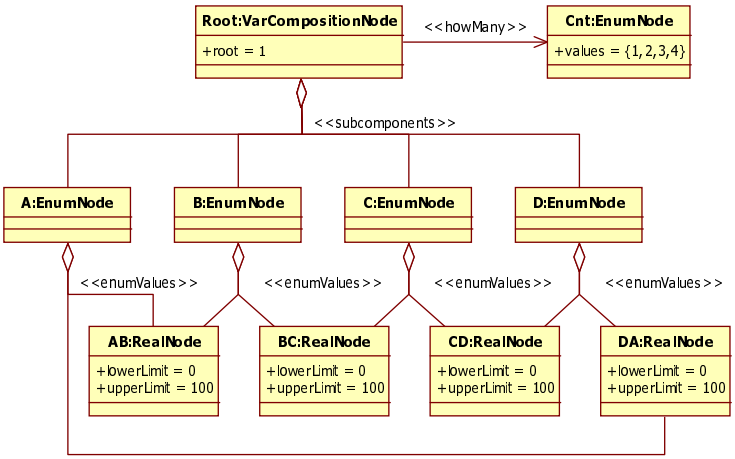


Figure 4: Example 2: model having multiply descended nodes prevents symmetry classes

select "shared" value AB, whereas combination of subcomponents A-C provides no such option. Moreover, both situations have different constraints:

In the first case $E_A = E_{AB}$ and $E_C = E_{CD}$ and we have unsolvable constraints for values of real nodes:

$$E_{AB} + 0 + E_{CD} + 0 \geq 150$$

$$E_{AB} + 0 + E_{CD} + 0 \leq 100$$

In the second case, $E_A = E_B = E_{AB}$ gives us solvable inequalities (for example, $E_{AB} = 80$):

$$E_{AB} + 0 + E_{AB} + 0 \geq 150$$

$$E_{AB} + 0 + 0 + 0 \leq 100$$

Thus, nodes B and C should never be equivalent. The multiple descendence limits our possibilities to define equivalences of constraints among different multiple descended nodes.

Example 3: In this example, we will show that existence of multiply descended nodes does not completely prohibit establishing equivalences.

The linear constraints are:

$$E_A + E_B + E_C + E_D \geq 150$$

$$E_{A0} + E_{B0} + E_{C0} + E_{D0} + E_{ABCD} \leq 100$$

Without going into deeper analysis, we claim that subcomponents A, B, C and D are symmetric and that symmetry is an equivalence relation. For any

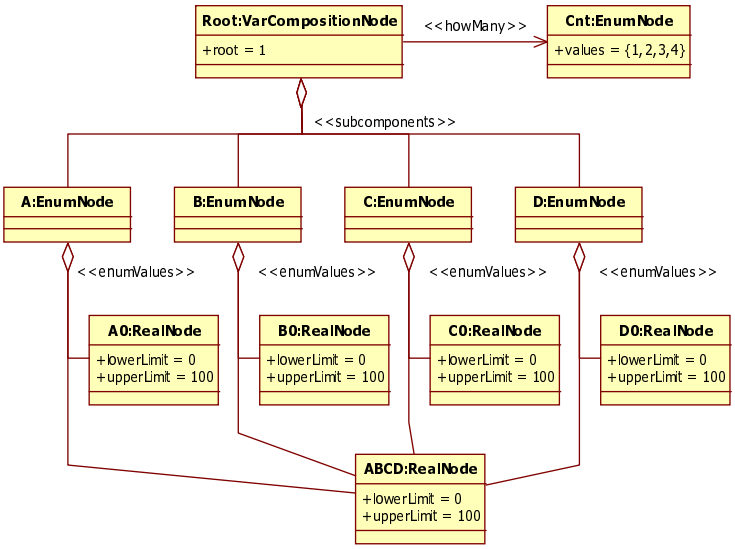


Figure 5: Example 3: model with multiply descended node, where branches are "symmetric"

count of selected subcomponents, exact identity of selected subcomponents is (from the feasibility perspective) irrelevant as any two branches are interchangeable.

Contrary to the situation in the first example, It is not equivalent situation if a subcomponent references the shared node (ABCD) or the exclusive real node (A0, B0, C0 or D0). Thus, A0 should never be equivalent to ABCD.

4.3.3 Linear equivalence

In this section, we will formally define equivalence of linear constraints for nodes in the optimization structure, but we will ignore all other constraints for nullness and composition. This equivalence will be used in following sections to define the "final" equivalence used for generating all valid classes of discrete allocations.

The heuristics supposes that many nodes in the model may have very similar linear inequality constraints, although the meaning of the nodes may be different. We want to divide nodes into classes of equivalence, such that they are treated the same way in the matrix of linear inequalities.

Notation: Permutation $B = A_{\{p(1),..,p(m)\}\{q(1),..,q(n)\}}$ of $m \times n$ matrix A with coefficients A_{ij} (where p is permutation of rows and q is permutation of columns) has coefficients $B_{ij} = A_{p(i)q(j)}$. Permutation $b = a_{\{p(1),..,p(m)\}}$ of row (or column) vector $a = (a_1, .., a_m)$ is defined as $b = (a_p(1), .., a_p(m))$.

Definition: N_i and N_j are **linear equivalent** (denoted $\sim_{lin}$) iff the set of

inequalities $A \cdot x \leq a$ is identical to the same inequalities with swapped columns i and j of the constraints matrix A:

$$A_{\bullet\{1,2,..,i,..,j,..,n\}} \cdot x \leq a \quad \Leftrightarrow \quad A_{\bullet\{1,2,..,j,..,i,..,n\}} \cdot x \leq a$$

or in other words that there exist permutation p such that:

$$A_{\{1,..,m\}\{1,2,..,i,..,j,..,n\}} = A_{\{p(1),..,p(m)\}\{1,2,..,j,..,i,..,n\}}$$

$$a_{\{1,..,m\}} = a_{\{p(1),..,p(m)\}}$$

We have to show that these relations defined for node pairs really are equivalences.

Lemma: conditions of linear equivalence define reflexive, symmetric and transitive relation.

Proof: Reflexivity and symmetry are self-evident. Proof of transitivity follows: For linear equivalences of nodes $N_i \sim_{lin} N_j$ and $N_j \sim_{lin} N_k$ there are permutations p and r such that:

$$A_{\{1,..,m\}\{1,2,..,i,..,j,..,k,..,n\}} = A_{\{p(1),..,p(m)\}\{1,2,..,j,..,i,..,k,..,n\}} \tag{2}$$

$$a_{\{1,..,m\}} = a_{\{p(1),..,p(m)\}}$$

$$A_{\{1,..,m\}\{1,2,..,i,..,j,..,k,..,n\}} = A_{\{r(1),..,r(m)\}\{1,2,..,i,..,k,..,j,..,n\}} \tag{3}$$

$$a_{\{1,..,m\}} = a_{\{r(1),..,r(m)\}}$$

We rewrite matrix A in the right side of equation 3 with equal matrix $A_{\{p(1),..,p(m)\}\{1,2,..,j,..,i,..,k,..,n\}}$ (such equality is proven by equation 2) so that composed permutations of rows and columns are applied:

$$A_{\{1,..,m\}\{1,2,..,i,..,j,..,k,..,n\}} = A_{\{r(p(1)),..,r(p(m))\}\{1,2,..,j,..,k,..,i,..,n\}} \tag{4}$$

$$a_{\{1,..,m\}} = a_{\{r(p((1)),..,r(p(m))\}}$$

Then, we rewrite matrix A (before applying row and column permutations) in the right side of equation 2 with equal matrix $A_{\{r(p(1)),..,r(p(m))\}\{1,2,..,j,..,k,..,i,..,n\}}$ (such equality is proven by equation 4):

$$A_{\{1,..,m\}\{1,2,..,i,..,j,..,k,..,n\}} = A_{\{p(r(p(1))),..,p(r(p(m)))\}\{1,2,..,k,..,j,..,i,..,n\}}$$

$$a_{\{1,..,m\}} = a_{\{p(r(p((1))),..,p(r(p(m)))\}}$$

Which is a proof that the permutation $p \circ r \circ p$ defines Linear equivalence $N_i \sim_{lin} N_k$. QED.

Linear equivalence of two nodes means that we may switch coefficients of linear constraints for these nodes, and the constrained space will remain unchanged. It is very important fact that linear equivalence is truly an equivalence - it classifies nodes into equivalence classes and is reflexive, symmetric and transitive.

The linear equivalence may be used for comparing feasibility of two solutions;

Lemma: Let there are two solutions (allocations of all model nodes) A, B with node numerical evaluations E_A and E_B. If numerical evaluation E_A can be obtained from numerical evaluation E_B by permuting its numerical evaluations inside each class of linear equivalent nodes, then A is feasible exactly when B is feasible.

Proof: The lemma is symmetrical - we will prove without loss of generality that feasible numerical evaluation E_A implies feasible numerical evaluation E_B: Let q is the permutation from lemma antecedent, that for any node N_i defines equality of numerical evaluations on i-th a $q(i)$-th positions:

$$E_B = E_{A\{q(1),\ldots,q(n)\}} \tag{5}$$

By definition of linear equivalence, switching the linear constraints among linear yield identical set of linear inequalities. Let us decompose permutation q as series of binary swaps $q_1 \circ q_2 \circ \ldots \circ q_k$ that always swap two linear equivalent nodes. This decomposition is possible, because permutation q only reorders positions inside of equivalence classes of linear equivalence. E_A satisfies linear constraints:

$$A \cdot E_A \leq a \tag{6}$$

By decomposition of permutation q into swaps q_i, by definition of linear equivalent nodes there exists for each q_i permutation p_i such that:

$$A_{\{p_i(1),..,p_i(m)\}\{q_i(1),..,q_i(n)\}} = A_{\{1,..,m\}\{1,..,n\}}$$

by repetitive application of these rules we obtain:

$$A_{\{p(1),..,p(m)\}\{q(1),..,q(n)\}} = A_{\{1,..,m\}\{1,..,n\}} \tag{7}$$

$$a_{\{p(1),..,p(m)\}} = a_{\{1,..,m)\}} \tag{8}$$

for permutation $p = p_1 \circ p_2 \circ \ldots \circ p_k$. We rewrite 6 reordered by permutation of rows p and permutation of columns q (simple arithmetic operations):

$$A_{\{p(1),..,p(m)\}\{q(1),..,q(n)\}} \cdot E_{A\{q(1),..,q(n)\}} \leq a_{\{p(1),..,p(m)\}} \tag{9}$$

and by application of 7, 8 and 5 we obtain:

$$A \cdot E_B \leq a \tag{10}$$

QED.

This lemma is a very important property. If by setting values of real nodes we can force two discrete allocations to have (after permutation of linear equivalent nodes) equal numerical evaluation of all constrained nodes, then these two discrete allocations are equally feasible. Nevertheless, the remaining problem with the linear equivalence is that:

- Linear equivalence does not differentiate between real nodes and enum nodes that may have various value sets. When linear equivalent nodes have different value sets, their feasibility does not behave equivalently.
- Numerical evaluations of enums with referential values retake numerical evaluation of their children. Referential values may be modeled as additional linear equation constraint (numerical evaluation of the parent equal to the child). Unlike standard linear inequalities, linear inequalities caused by referential values are conditional, required only when the solution has according referential value of the enum node.
- Lastly, linear equivalence does not give us information about equal validity of selecting different subcomponents of variable compositions, which would be helpful for identification of all valid equivalent groups of discrete allocations.

4.3.4 F-equivalence

In previous section, we have classified nodes by equivalence of linear constraints. In this section, we will define new node equivalence, that will include both constructiveness and feasibility. The F-equivalence of nodes is finer than linear equivalence. Evaluation of this relation has recursive definition and can be tested by processing nodes in reverse topological order (children first):

Nodes N_i and N_j are **F-equivalent** ($\sim_F$) if following conditions are true:

1. Nodes N_i and N_j are of the same type (EnumNode, VarCompositionNode, RealNode, StaticNode)
2. Nodes N_i and N_j are both singly descended, or $i = j$ (multiply descended nodes are not F-equivalent to any other node, they are only F-equivalent reflexively)
3. Nodes N_i and N_j are linear equivalent. By definition of linear inequalities (see section 4.2), this also includes equal lowerBounds and upperBounds of RealNodes.
4. For EnumNodes, sets of numeric values for both nodes are equal. Sets of nodes referenced by ReferentialValues are F-equivalent.
5. For VarCompositionNodes, howMany values are equal numeric values or references to F-equivalent nodes. Sets of subcomponents are for both nodes F-equivalent.
6. For RealNodes, source index is at most 1 (see definition in section 4.1), or $i = j$ (nodes that can be sources for multiple EnumNodes are only F-equivalent to themselves, they are only F-equivalent reflexively)
7. Nodes referenced by static references of N_i and N_j are F-equivalent.

When two sets of nodes are F-equivalent, then both contain the same number of nodes from each F-equivalence class. Note that all the conditions 1-6 are reflexive, symmetric and transitive, therefore the relation of F- equivalence is an equivalence. The name "F-equivalence" was chosen to indicate intended use of this equivalence for classification of nodes according to their influence on **f**easibility, but at the same time we wanted to avoid confusion that this equivalence is defined by the property of feasibility itself.

Important part of definition of F-equivalent nodes N_i and N_j is that all their children have to be F-equivalent. This property has two interesting implications:

Observation 1: using induction, it can be easily demonstrated that F-equivalent nodes have subgraphs of the same size.

Observation 2: using induction, it can be easily demonstrated that all multiple descended nodes in subgraph of node N_i are also present in subgraph of any F-equivalent node N_j. We use the fact that multiple descended nodes are only equivalent reflexively.

4.3.5 Signature equivalence of discrete allocations

Definition: Signature $sig(L)$ of a discrete allocation L is vector with one entry $sig_i(L)$ for each node N_i of the optimization structure. Entries are defined as follows:

1. for real nodes, the value is nullness of that node in the discrete allocation
2. for enum nodes, the value is either:
   (a) nullness, it the enum node is not numerically evaluable
   (b) numeric value, if the enum node itself has numeric value, or source of its numeric value is enum with a numeric value
   (c) linear equivalence class of the source of its numeric value, if the source is a real node
3. zero for any other type of node

Definition: Two discrete allocations L_X, L_Y are **signature equivalent** (denoted $sig(L_X) \sim_{sig} sig(L_Y)$) if there exist permutation p among nodes of the same F-equivalence class, such that $sig_i(L_X) = sig_{p(i)}(L_Y)$ holds for all i. In other words, that there exist bijection among F-equivalent nodes of both allocations L_X and L_Y, such that they have equal numeric value or equal real source of values.

Theorem 1: Two Signature-equivalent discrete allocations are both feasible or both unfeasible.

Proof: Because of the fact that the conditions for F-equivalence of discrete allocations are symmetric, without loss of generality we will only prove that feasible allocation L_X implies feasible allocation L_Y.

Let's have two discrete allocations L_X and L_Y and let permutation p defines bijection of nodes that satisfies conditions for F-equivalence of discrete allocations. Given that F-equivalence implies linear equivalence (linear equivalence is one of conditions for F-equivalence), permutation p only permutates nodes inside classes of linear equivalent nodes, thus all permutated real nodes allow same range of values. If there exist feasible solution in L_X for some allocation of the real nodes, we will use the same allocation of permutated real nodes for the discrete allocation L_Y and we will show that such solutions must also satisfy all constraints.

More formally: Let $x = (x_1, .., x_n)$ is a feasible solution from allocation L_X. Its evaluation satisfies:

$$A \cdot numEvaluation(x) \leq a$$

We create solution $y = (y_1, .., y_n)$ from allocation L_Y:

1. when $x_{p(i)}$ is $non - NULL$ real (case 1), then we set $y_i := x_i$. Value y_i is from its defined interval because bounds are part of linear constraints and node x_i is from interval.

2. when $x_{p(i)}$ is $non - NULL$ value of enum node $N_{p(i)}$, its value is already defined by the allocation L_X, and also the value of y_i is already defined by allocation L_Y. Note that by definition of Signature equivalence, $numEvaluation(x_{p(i)})$ has source either numeric value that is equal to $numEvaluation(y_i)$, or source nodes of both evaluations are F-equivalent real nodes, or both nodes are not numerically evaluable.

3. when $x_{p(i)}$ is $non - NULL$ value of variable composition node $N_{p(i)}$, its selected subset is defined by allocation L_X and the value of y_i is defined by allocation L_Y. Note that signature does not explicitly describe concrete selections of variable compositions, but it does describe nullness of all enum and real nodes.

4. when $x_{p(i)}$ is $NULL$ value, y_i is also $NULL$ value. Same nullness (of any enum or real) node $x_{p(i)}$ and y_i result from definition of Signature equivalence of allocations L_X and L_Y.

Solution y is evidently member discrete allocation L_Y. If we compare evaluations of solutions x and y, we obtain

$$numEvaluation\left(x_{p(i)}\right) = numEvaluation\left(y_i\right)$$

unless values $x_{p(i)}$ and y_i are both referential values of enum nodes, and source of theirs evaluations are real nodes. Although the source real nodes are linear equivalent, their evaluations are not necessarily equal.

- If source real nodes of a pair are multiply descended or their source index is bigger than 1, they must be one node, because of second and sixth conditions of F-equivalence (it states that node with these properties are only F-equivalent reflexively). Evaluation of these nodes is therefore necessarily equal.
- Otherwise, the source real nodes are single descended and have source index at most 1. We will denote J to be set of these real nodes.

We create modification z of solution y from discrete allocation L_Y:

- for all nodes $N_i \notin J$, we set $z_i = y_i$.
- for nodes from each F-equivalent class of nodes in J, we reorder assignment of values of y to z in a way that
  - if node N_i is source of numerical evaluation of EnumNode N_h in discrete allocation L_Y, then we take the parent's paired value $N_{p(h)}$ in discrete allocation L_X. By definition of the bijection, $N_{p(h)}$ is also EnumNode that references F-equivalent real node N_{h2} ($N_{h2} \sim_F N_i$). We set $z_i = x_{h2}$ which is in fact $z_i = y_{p^{-1}(h2)}$, note that $N_{p^{-1}(h2)} \sim_F N_{h2} \sim_F N_i$.
  - for the rest of values z_i of nodes N_i that are not source of numerical evaluation for other nodes, we redistribute unused values of solution y from the F-equivalent class.

Let permutation r denotes the reordering of real values from y to z in all F-equivalent classes. Because we only reordered $non-NULL$ real values, solution z is from discrete allocation L_Y. Solution z has important property:

$$numEvaluation\left(x_{r(p(i))}\right) = numEvaluation\left(z_i\right)$$

For all nodes. Given the fact that permutations r and p only reorder values of nodes inside F-equivalent classes, thus also inside linear equivalent classes, composed permutation $r \circ p$ also has this property. Using lemma about linear equivalences from section 4.3.3, we obtain that:

$$A \cdot numEvaluation(z) \leq a$$

QED.

Theorem 2: Two constructive Signature equivalent discrete allocations are both valid or invalid.

Proof: The lemma about discrete allocation from section 4.2.1 states that all solutions from single discrete allocation are all constructive or all inconstructive. Therefore, if there exists constructive solution in L_Y, the constructed solution z from the previous proof is also constructive.

4.3.6 Generating constructive Signature-equivalent discrete allocation classes

In this section, we will show an algorithm that uses F-equivalence of nodes to generate all classes of signature-equivalent discrete allocations.

1. We create set of all possible enum allocations (i.e. allocations of all enum nodes) without any $NULL$s. We compute numerical evaluations of all integer-evaluable nodes.

2. For each allocation, we define auxiliary mask that defines proposal for nullness. At start, this proposal states $non - NULL$ for root node and $NULL$ for all other nodes. The mask proposal will be iteratively changed in later steps.

3. For each node N_i in topological order

   (a) for all allocations with mask set to $non - NULL$ for N_i, we reset masks for statically referenced nodes to $non - NULL$.

   (b) if node N_i is enum or variable composition, for all allocations with mask set to $NULL$, we modify these allocations to have $NULL$ allocated for node N_i.

   (c) if node N_i is enum, for all allocations with referenceValue and mask set to $non - NULL$ for N_i, we reset masks for these referenced nodes to $non - NULL$.

   (d) if node N_i is variable composition:

      i. we compute howMany value, that is either constant or referential, in which case we use the pre-generated numerical evaluations, which we have since they are integer-evaluable.

      ii. We generate all combinations of counts of selected nodes for each F-equivalent classes of subcomponents, so that the total count is equal to the computed howMany value, and that does not exceed number F-equivalent subcomponents of this variable composition.

      iii. For each combination we take one random sample of composition.

      iv. For all allocations with mask set to $non - NULL$ for N_i, we replace each of this allocations with set of allocations, each with different sample for the node N_i. We set their masks to $non - NULL$ for the selected subcomponents.

4. Use generated allocations as samples to define discrete allocations classes. Use linear programming (see section 4.2.3) to determine feasibility of each class.

Note: algorithm is not optimal. Due to pre-generation of all combinations of enum allocations, some solutions will be duplicated when the enums are

set to $NULL$ due to not being referenced. Moreover, some generated allocations will fit into the same class of signature-equivalence, although they have different structure and thus are generated separately. These duplicities can be easily removed afterwards and should not significantly increase number of generated allocations.

We will now demonstrate that the algorithm does not omit any constructive signature-equivalent class of discrete allocations.

Lemma 1: For any two F-equivalent nodes $N_i \sim_F N_j$, and all constructive allocations of subgraph of node N_i (i.e. node N_i and its descendants) where N_i is $non - NULL$, we can find corresponding allocation of subgraph of node N_j with N_j being $non - NULL$, such that nodes in subgraph of N_i and nodes in subgraph for N_j produce equivalent signature (see section 4.3.5). To remind the readers, those conditions are same nullness of both nodes and either:

1. both paired nodes are real
2. both paired nodes are enums and either:
   (a) none of them is numerically evaluable
   (b) nodes have equal numeric evaluation whose sources are enums with numeric values
   (c) nodes have referential values whose sources are F-equivalent real nodes
3. both paired nodes are of another type

Proof: This proof is very technical and its only idea is to use definition of F-equivalent nodes to demonstrate that they have F-equivalent sets of possible instances of their subgraphs.

Note that definition of F-equivalence of nodes does not permit recursivity - node may not reference F-equivalent subnode, because by definition of F-equivalence, such subnode would also have to reference F-equivalent node, and so on, leading to cyclic or infinite structure.

We will use induction, i.e. start from leaves of the graph and continue upwards. Subgraph roots N_i and N_j will be presumed in both cases to be $non - NULL$. When we process the existence of corresponding allocations for two F-equivalent nodes, we may consider this property proven for all classes present in their subtrees. Thus if we pick two F-equivalent $non - NULL$ subnodes of N_i and N_j, having certain allocation on the subgraph of subnode of N_i, we are able o find allocation of the subgraph of the subnode of N_j producing equivalent signature.

First step of induction are rules for leaves:

- If N_i and N_j are two F-equivalent reals, condition 1 is always satisfied because we presume the nodes to be $non - NULL$.

- If N_i and N_j are two F-equivalent leaf enums, they have same set of numeric values and no referential values (because we are processing leaves). For any allocation N_i the same allocation of N_j is available and satisfy condition 2b.

- In this first step we are only processing leaves, so N_i and N_j cannot be variable compositions.

- If N_i and N_j are two F-equivalent nodes of other type, they satisfy condition 3.

In all cases, nodes N_i and N_j are paired and produce equivalent signature

Provided that we have proven the property for all deeper nodes, we prove it for their parents. From induction we are able to find signature equivalent allocations for F-equivalent $non-NULL$ subnodes and their subgraphs. We may ensure signature equivalent allocation for F-equivalent $NULL$ subnodes and their subgraphs: All singly descended nodes in $NULL$ subnode subgraph must also be also $NULL$ and pairing of these $NULL$ nodes satisfies conditions 1, 2a or 3. Multiply descended F-equivalent nodes are always the same node - by condition 2 of F-equivalence. If they are in $NULL$ subnode subgraphs, they may either be $NULL$, in which case the signature equivalence is assured, or $non-NULL$, in which case these nodes must appear in some $non-NULL$ branch elsewhere in the structure. Signature equivalence will be ensured when processing that $non-NULL$ branch.

We have proven that we can ensure equivalent allocations of subgraphs of two F-equivalent subnodes of N_i and N_j. We can easily use these allocations to construct allocation of subgraphs of F-equivalent **sets** of subcomponents of N_i and N_j; For singly descended nodes this is trivial as they appear only in one of the subgraphs. Multiply descended F-equivalent nodes may appear in subgraphs of multiple nodes, but they are actually one node and always have the same value.

Now we prove the step of induction by pairing the nodes N_i and N_j (both are $non-NULL$), and finding an allocation of N_j that would produce same signature value as allocation of N_i and at the same time N_j would have the same set of $non-NULL$ subnodes as N_i:

- First, N_i and N_j have by F-equivalence also F-equivalent set of static referenced nodes.

- If N_i and N_j are two F-equivalent enums, the nodes have same set of possible numeric values and same set of possible referential values. For allocations of N_i selecting numeric values, corresponding allocation of N_j selects equal numeric value, and pair N_i and N_j satisfy condition 2b. For allocations selecting referential values, corresponding allocations of N_j selects F-equivalent reference value, and signature equivalence of the selected subnode ensure signature equivalent signature of allocation of N_i and N_j (satisfying conditions 2a, 2b or 2c).

- If N_i and N_j are reals (non-leaf reals are possible if they have static references), N_i and N_j can be paired according to condition 1.

  - If N_i and N_j are variable compositions, they combine allocations of howMany reference (if howMany is referential) and allocations of subcomponents. By induction, for any allocation of the howMany reference of N_i there exist allocation of F-equivalent howMany reference N_j with bijection of their subgraphs. Moreover, because also the howMany referenced nodes have same signature, and because both must be integer-evaluable, they must be paired according to condition 2b and thus their numerical evaluations are thus equal.
    Given the howMany value (regardless if numeric or referential), for any allocation of node N_i selecting certain combination of subcomponents, there exists correspondent allocation of F-equivalent subcomponents of N_j. Nodes N_i and N_j can be paired themselves according to condition 4.

QED.

Lemma 2: Variable composition selecting subcomponent N_i instead of another F-equivalent subcomponent N_j, will generate signature-equivalent set of constructive discrete allocations.

Proof: The idea is to widen Lemma 1 from F-equivalent node subgraphs to whole discrete allocations. For any discrete allocation L_X, we will construct similar discrete allocation L_Y differing only in subgraphs of N_i and N_j and equal elsewhere. The discrete allocations L_Y will use Lemma 1 to swap subgraph under N_i of allocation L_X to subgraph under N_j. By equality of L_X and L_Y outside subgraphs of N_i and N_j and equivalence inside the subgraphs, we always obtain signature-equivalent discrete allocation.

Now formally: Let x solution from constructive discrete allocation L_X selecting subcomponent N_i and not selecting F-equivalent subcomponent N_j of a variable composition. Solution x has $non-NULL$ value of N_i, while all $non-NULL$ values in subgraph of N_j must be multiply descended as N_j was not selected.

- Let L_{xi} is allocation defined by values of x on nodes in subgraph of N_i, L_{xj} allocation defined by values of x on nodes in subgraph of N_j and let L_0 is allocation defined by values of x on nodes outside both subgraphs of N_i and N_j. Thus L_X is a suballocation of $L_0 \cap L_{xi} \cap L_{xj}$ restricted to allocating only discrete nodes.

- For allocation L_{xi} of subgraph of node N_i, there exist (using the Lemma 1) corresponding allocation L_{yj} of subgraph of node N_j with $non-NULL$ N_j.

- For allocation of L_{xj} of subgraph of node N_j, we may construct allocation L_{yi} of subgraph of node and a bijection among the nodes, such that:

  - multiply descended nodes in L_{yi} are the same as multiply descended nodes in L_{xj} (by Observation 2 in section 4.3.4, multiply descended nodes in subgraphs of F-equivalent nodes must be shared) and their pairing is reflexive.
  - singly descended nodes are all $NULL$ and are paired arbitrarily in each F-equivalence class (by Observation 1 and 2 in section 4.3.4, there must be equal number of singly descended nodes in subgraphs of F-equivalent nodes)

- Let allocation L_Y is defined as suballocation of $L_0 \cap L_{yi} \cap L_{yj}$ restricted to allocating discrete nodes. Allocation L_Y is evidently constructive. Given that $L_{xi} \sim_F L_{yj}$ and $L_{xj} \sim_F L_{yi}$, then also $L_X \sim_F L_Y$.

QED.

4.3.7 Feasibility - conclusion

In this chapter, we have explored into more detail how constraints may be enforced, even though we have demonstrated that this is a complicated task. We have designed methods to explore feasibility throughout the solution space, using various equivalences to simplify the task as much as possible. We are however relying on heuristics rather than "secure" or proven algorithms, thus we cannot assure universal suitability of the proposed methods. If the feasibility exploring fails due to excessive number of equivalence classes, we may try eliminating solutions as proposed in section 4.2.3. If neither approach is suitable for the problem, evolutionary algorithms may not be suitable to solve such type of problem. In following sections, we will introduce several algorithms to perform the evolutionary optimization. As mentioned before, these algorithms generally operate over solution spaces that are fairly simple and do not consider any special constraints or particular structure of the solution space. Solution space representation as Cartesian product and constraint enforcement will serve as intermediate tier, that allows us to apply general algorithms to particularly designed optimization problems.

Exact implementation of the signatures and equivalences will be explained on a simple example in section 6.

4.4 Variables in semidiscrete problems

In this section we take a closer look at interpretation of the discrete and continuous variables.

4.4.1 Types of variables

In generic discrete evolution problems, values of variables are typically binary or from a small set of numbers or symbols. We will distinguish cases when a discrete variable is qualitative (nominal) or quantitative (ordinal or cardinal). In section 4.1 we have introduced two particular examples of usage of discrete

variables. Composition is qualitative discrete variable, the variable defining size of the composition is quantitative (cardinal) discrete variable. From the modeling point of view, there will be no difference, because except previously stated examples, the business-defined optimization structure does not contain such information, and also because EDA algorithms themselves do not differentiate between these variable types.

Variables of continuous problems have always quantitative character. Possible values are real numbers from intervals, because in practice, there are always constraints such as lower and upper bounds or maximal numerical precision. Although application of numerical precision may resemble that the variable is actually discrete, we will always consider these variables to have continuous value sets and numerical precision will be only a hint for algorithms to stop trying to be more precise.

Theories often only consider purely discrete or purely continuous problems. What happens in the real world (and specifically in our problem model) is that optimization problems are typically semi-discrete: partly discrete, partly continuous. Typical problem that should be solved by the evolutionary algorithm framework is evolution of a chemical catalysts for enabling certain chemical reactions. If the catalyst has to be mixed from at most three components out of ten, then it would be modeled as variable composition. Its $howMany$ value would be cardinal discrete variable with values 1, 2 or 3. Subcomponents would be static nodes having static references to component proportion in the catalyst solution. Proportions would have only parent each, which would be the static node of the chemical component. (We might equally model the situation without using the static nodes, putting the proportions as subcomponents of the composition, but this way our solution is more correct formally and closer to the business defined interfaces). In any case, proportions would be real variables with linear constraints enforcing that the total sum of proportions would be 1. This is an example of usage, naturally we want to keep the framework as general as possible.

4.4.2 Nonlinear variables

Some continuous variables may be further specified to behave non-linearly for the purpose of generating the initial population and for adjusting densities of probabilistic models. In our model, variable may be either linear or logarithmic. If a variable in the original problem is logarithmic, we use logarithm of the variable in the density estimation.

Usage of logarithmic variables is defined by attribute intervalParameters of RealNode. The inclusion of the nonlinear behavior is surprisingly simple. The algorithms do not operate with the population in the form as it is read from the database, but after set of transformations (see section about population normalization in section 5.4). The normalization process will use the concrete nonlinear transformation on the population, so the EDA algorithm does not have to know anything about it. Once the next generation is sampled, the population is denormalized and reverse nonlinear transformation is applied.

5 Algorithms for semidiscrete problems

In this section, we will focus on construction of algorithms, that will be able to operate in complex semidiscrete environment. Our algorithms will have to overcome difficluties arising from complex definition of solution space and its constraints and semidiscrete character of the solutions.

In section 5.1, we will identify specific difficulties that arise when we are optimizing solutions defined and constrained by optimization structures. In section 5.2, we will present frameworks for construction of semidiscrete algorithms. In section 5.3, we will present an general abstraction describing operation of algorithms. In section 5.4, we will describe transformations of population performed in our framework. In section 5.5, we will present how our framework operates with $NULL$ values in the population.

5.1 Challenges of optimization structures

So far, algorithms were designed for discrete OR continuous problems. Considering our problem model, we have to overcome this limitation and make the algorithms semi-discrete. Many existing algorithms have both discrete and continuous versions, but they very often only share the key idea and their discrete and continuous implementations are slightly incompatible.

This problem can be best explained on example: Consider algorithms EBNA and EGNA - both are using the key idea of graph representation of interactions that reflects only the most likely variable interactions. Both can be, for example, estimated by penalized maximum likelihood. Bayesian network in EBNA defines interaction of $discrete-discrete$ dimensions, whether Gaussian network in EGNA defines $continuous-continuous$ interactions. To make a EBN+GNA hybrid, we must also design $discrete-continuous$ interactions. To be complete, we might also want to use interactions, where a discrete or continuous variable is depending on multiple, both discrete and continuous parental variables.

Another issue are $NULL$ values. As indicated in section 4, values will not be all defined for some solutions. Presence of $NULL$ values may well affect several procedures present in existing EDA algorithms. We must define the way the $NULL$s will be handled in each of the procedures:

- Clustering. There is no clue what is the distance between $NULL$ and non-$NULL$ value. If we set this distance in a dimension to be always 0, we are breaking the triangle inequality: distance between any two points A, B must be smaller or equal than distance between A and $NULL$ plus distance between B and $NULL$. If both of the latter distances are by definition 0, then all distances in the space must be 0, which is not desirable. For the purpose of clustering, we will enumerate $NULL$ values to be equal to average of values in the dimension.

- Aggregate values including mean, variance, sum. $NULL$s can be ignored, but we must also make correction to the number of values being aggre-

gated. Let's put that in simple example: average of $\{2, 4, 6, NULL\}$ is $\frac{12}{3} = 4$. Similar rules must be created for other aggregate values.

- Covariances. Only pairs where both values are non-$NULL$ are taken into account when computing covariance. In probabilistic models, if a variable is sampled and the covariance is used to modify its distribution, whenever there is dependence on the $NULL$ value, the dependency is omitted.

Special business requirement for the algorithm is necessity of batch operation: algorithm should compute multiple fitness function evaluations in one phase and estimate the fitness and generate new solutions in another phase, where number of the phases should be reasonably limited. Algorithms like PBIL are clearly not appropriate due to this requirement. Reason behind this requirement is proposed software architecture of the system.

5.2 Constructing semidiscrete algorithms

We want to base our comparison on performance of existing algorithms, because they are widely reviewed by the scientific community and well tested. Because of the complexities of our solution space, we must consider additional modifications that will adapt existing algorithms to our needs. We have proposed multiple of such adaptation.

5.2.1 Plain algorithm combination

Perhaps the most straightforward solution is to use some discrete algorithm on the discrete part and a continuous algorithm on the continuous part of the problem. This means, however, that interactions between discrete and continuous variables will not be reflected in the probabilistic model in any way. Some algorithms (UMDA, UMDAc) are not modeling dependencies and plain combination is thus a natural solution.

Two algorithms, discrete and continuous, are required to specify this approach. The algorithms may be discrete and continuous versions of one algorithmic principle (for instance, UMDA and UMDAc, EBNA and EGNA), but not necessarily.

5.2.2 Shared clustering

In previous construction, if any of the algorithms uses clustering, it is not shared with other half of the problem. The discrete and continuous parts may even both use clustering, while clusters are selected differently in each case. In Shared Clustering construction, we explicitly use same clustering on both parts of the problem. The clustering is an information passed to both parts and leverages coordination between them.

The construction is rather simple: First we separate selected solutions into several clusters. We use discrete algorithm with clustering on the discrete part

of each cluster. Then, we use continuous algorithm on the continuous part of each cluster.

Two algorithms (discrete and continuous) and discrete space clustering are required to specify this approach. No other clustering (except the shared one) will be performed by the underlying algorithms.

5.2.3 ClustEBNA

Third proposed construction maps continuous variables to discrete space in order to infer one big model (e.g. EBNA or MIMIC) of interactions of both discrete and continuous variables. In each continuous dimension, existing values of solutions are divided to segments. Segmentation may be performed by one-dimensional clustering algorithm with fixed number of clusters, e.g. K-means; Unlike standard clustering, the segmentation is not performed in order to identify separated clusters of solutions. Multiple segments are created even if all the data is in one cluster. However, if the solutions actually are forming separated clusters, it is highly desirable to have the classes created accordingly to these existing divisions. For each segment in a dimension, simple one dimensional model is used to model its distribution. We decided to use Gaussian distribution due to simplicity and robustness of its estimation; Index of segment is then used as a discrete descriptor of a continuous variable. Discrete EDA algorithm is used to model interaction among all variables, operating with both discrete values and indices of continuous segments.

The sampling is straightforward. Once the discrete distribution is sampled, segment indices sampled for continuous nodes are transformed to continuous values using estimated distributions of segments.

The name clustEBNA is derived from "clustering" and "EBNA", that are suitable algorithms whose combination is used to solve semidiscrete problems. Interestingly, clustEBNA algorithm is able to model continuous distributions with complex shapes. The more segments are created in each dimension, the more precise the shapes can be. Excessive number of segments may be, however, stress on the discrete algorithm.

We also experimented with variation of this construction, that use histogram models used for segmentation in continuous dimensions (histEBNA). The most challenging task is to adapt intervals and refine them in time, in order to get more precision in following generations. The key idea is to have relatively high number of segments, but keep most of them empty, in order not to send lot of different indices to the discrete algorithm. In this respect, clustEBNA was naturally self-adapting, while in case of histograms, the adaptation must be specifically designed. Experimenting with histEBNA might be done in future if clustEBNA provides promising results.

5.2.4 Modifications of conventional genetic algorithms

We also propose adaptations of conventional genetic algorithms. The main difference is their crossover operator, which only include information from very

limited number of parents.

Adapted SGA: This modification replaces real values by their binary representation to certain precision.

Adapted Evolution Strategies: With this algorithm, we are in opposite situation, when we must define a way to process discrete variables. We reused the approach of SGA, where values are directly inherited from the parents and with certain probability mutated to certain third value. In the genome, we put first values of discrete variables and then values of continuous ones. Variability metadata are only created for continuous variables. To bring more coherence to algorithm definitions, we use the concept of elitist population and use approach of repacing the parental population otherwise.

When using adaptations of crossover-based algorithms, we decompose variable compositions by default. We prefer to have bigger number of small variables than vice versa.

5.3 Algorithm abstraction

In previous sections, we have shown how Estimation of Distribution Algorithms use distribution estimation and distribution sampling as two central operators to reproduce parental population. We have already outlined few examples of algorithms from the EDA family and briefly described their specifics. In this section, we will introduce an abstraction of the algorithm operation.

To formalize a genetic algorithm, it is a function:

$$Pop^{t+1} = Alg_{params}(Pop^t)$$

Where Alg_{params} is the genetic algorithm, parametrized by tuple of algorithm parameters *params*. Pop^t is the population at generation t, and population is defined as multiset of N instances of variable Z (See precise definition of solution space Z in chapter 4); $Pop^t = \{z^{t,w} \in Z | w = 1, .., N\}$. $Distribution^1$ is the initial population distribution, typically a uniform distribution over the solution space. We call Pop^1 the initial population.

Algorithms from the EDA family may be further decomposed to subalgorithms of selection, actual function of distribution estimation and function of distribution sampling. Different implementations of these subalgorithms may be combined arbitrarily and further parametrized:

$$(Parents^{t+1}, ElitePop^{t+1}) = Select_{selectParams}(selectMethod, fitnessFunction, Pop^t)$$

$$Distribution^{t+1} = EDA_{edaParams}(algorithm, Parents^{t+1})$$

$$Pop^{t+1} = ElitePop^{t+1} \uplus Corrections(Sample_{sampleParams}(Distribution^{t+1}))$$

Function *Select* transforms population to parental and elitist subsets in multiset sense. Operator $\uplus$ stands for multiset union, generating multiset containing all (possibly duplicated) samples from both multiset operands.

$$\forall x : x \in Parents^{t+1} \Longrightarrow x \in Pop^t$$

$$\forall x : x \in ElitePop^{t+1} \Longrightarrow x \in Pop^t$$

We might be using different methods *Selectmethod*, picking one of selecting algorithms introduced in section 3.1.3. Part of the population will be selected for reproduction, and potentially part of the population will be marked as "elite", passing to the following generation unchanged.

EDA function performs the actual *algorithm* for estimation of distribution. Structure $Distribution^t$ has not a fixed definition - it is a structure of distribution parameters that defines the estimated probability distribution, specific for each EDA algorithm. Function *Sample* samples $N - elitism$ instances of the population accordingly to the probability distribution. *Sample* functions defined for EDA algorithms do not take into account constraints of the solution space as defined in section 4. For this reason, function *Correction* is applied to transform raw sampled solutions from unconstrained space into space with constraints as we defined it. Population Pop^t always consist of correct solutions in constrained space.

The *algorithm run* is sequence $\{(Distribution^t, Pop^t)|t = 1..T\}$, obtained by application of functions above in loop. Many of the procedures in different algorithms are modeled to behave like random (they are, in fact, always pseudorandom. It is our goal to make some of the procedures "as random as possible").

5.4 Normalized and denormalized populations

When the population is passed to the EDA algorithms, it is transformed to **normalized form**. Normalized form is abstraction used for easier processing by EDA and clustering procedures. Normalized population has following properties

- The population is splitted into two populations, that contain data of discrete and continuous respective parts of each individual.
- The variable compositions in discrete population may be decomposed to number of binary discrete variables and composed back (for details, see decomposition at the end of section 4.1).
- The variable composition maps all continuous variables to intervals $[-1; 1]$. For variables with logarithmic scale, logarithm of values is computed and then mapped onto the $[-1; 1]$ interval. This is crucial for correct operation of clustering and correct behavior of EDAs on the nonlinear variables.
- To the structure of normalized population, additional data may be added about mean and variance of variables and covariance of variable pairs.

The **denormalized form** of population is the standard format defined by the optimization structure. The denormalized form is used for storing population into database. The normalized form is used for processing the next generation.

5.5 NULLs in the population

Another aspect unknown to proposed EDA algorithms are $NULL$ values. Their presence in the solution cannot be modeled stochastically, because optimization structures define precise rules where $NULLs$ must be and must not be. As stated in section 4.2.3, we assume that the population is sampled without any null values and nullness is computed later, depending on the values of each individual. The algorithm is very simple, We start with sampled values and null mask set to zero everywhere except the root. Then we process nodes in their topological order, and we propagate non-$NULL$ values in mask from each processed non-$NULL$ node to all statically referenced children and selected children (if the current node is a variable composition or enum node with referential children). Thanks to topological ordering, only one walkthrough is needed.

High number of $NULLs$ in present in population for one variable may also be a problem, because it prevents us from realistically model distribution of that variable due to low number of relevant samples. We consider this to be a problem especially for continuous variables. In ClustEBNA, number of segments for a variable is lowered if there is not enough non-$NULL$ values to realistically estimate them.

Special adjustments are done for Simple Genetic Algorithm and Evolution Strategies. If a child inherit $NULL$ value from its parent, it will try to replace it by value of the second parent. If even that is $NULL$, it will look for a value to other solutions in the population. If no individual defines non-$NULL$ value of that variable, it is instantiated by default, with uniform distribution, discrete or continuous.

6 Architecture and implementation

In this section, we will outline architecture, data formats and procedures in our implementation. During the process, the population is transformed to different formats, each corresponding to task that needs to be performed. We will also present other structures that are used to describe data and data model.

The general architecture of the program is designed to cooperate with **external source of fitness values** for populations. For this purpose, when a population is sampled, it is stored into a database. The exact table model of the data depends on the optimization model and is further specified by **database structure** parameter, provided by the user. Structures are described in Appendix: Data Structures. The program is run each time, when next generation has to be sampled. Program loads last population from the database and estimates distribution . Then, next generation is sampled, that will be stored back into the database. After this one step, program ends. Alternatively, only for

the initial population, the program does not load any samples from database and uses initial distribution, that is described in the optimization structure.

When a generation is sampled, its fitness is evaluated. We expect that this process will be relatively slow and costly, as it may involve preparation and execution of real-life laboratory experiments. Each individual in the population serves as set of parameters for the experiment. When a fitness is computed, it is stored into the database on a position, that is defined by the database structure.

For purposes of benchmarking and testing, program may also take additional parameter of a testing fitness function, that will for each individual compute its fitness. Testing fitness functions are custom made, they only have defined interface they must satisfy. When running the algorithm, either testing function or externally provided fitness values must be provided.

The program is written in MATLAB language, while the requirement is to be compatible with version 7.X. We implemented connection to the Mysql database, using the appropriate driver. Besides stardard matlab functions, we have used following standartly implemented features:

- Mysql database connector and database toolbox
- *linprog*: matlab implementataion of linear programming algorithm (optimization toolbox)
- *kmeans*: matlab implementation of K-means clustering (clustering toolbox)
- *ttest2*: matlab implementation of Student's t-test (statistics toolbox)
- *fracfact*: matlab implementation of fractional factorial designs (statistics toolbox)
- built-in Java: used for structures in computation of feasibility

The program is modular and extensible; The moduularity allows users to customize many parts:

- Custom semidiscrete EDA algorithms
- Custom discrete and continuous EDA algorithms and clustering algorithms for the semidiscrete "merger" algorithms
- Custom benchmarking functions
- Customizable set of parameters, that will be sent to the algorithms from the interface

Routines to extract population and fitness from the database are implemented. Other databases may be used by using database driver with compatible API, or completely remade data load may be used, as the EDA interface may work with already extracted populations.

6.1 Example model

We will be using simple example optimization model, composed of five nodes: Root, Cnt, A, B and C in this order;

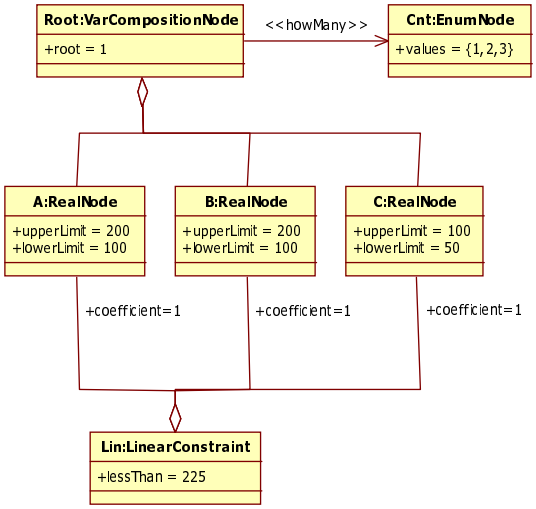


Figure 6: Example optimization model

There is also a linear constraint, that represents inequality of evaluations of nodes A, B and C:

$$E_A + E_B + E_C \leq 225 \tag{11}$$

The Root node is discrete node having defined 8 possible $non - NULL$ values, as listed in table 1.

value encoding	composition
1	$\varnothing$
2	$\{A\}$
3	$\{B\}$
4	$\{A, B\}$
5	$\{C\}$
6	$\{A, C\}$
7	$\{B, C\}$
8	$\{A, B, C\}$

Table 1: Encoding of variable compositions of node 'Root'

Note that Root value 1 representing $\varnothing$ yields no constructive solution, because allowed $howMany$ values are from $\{1, 2, 3\}$. If decomposition of variable compositions is set, the Root will be separated into three binary variables, one for each potential subcomponent A, B and C. Encoding of enum variable Cnt is very straightforward, as shown in table 2.

value encoding	numeric evaluation	allowed compositions
1	1	$\{A\}, \{B\}, \{C\}$
2	2	$\{A, B\}, \{A, C\}, \{B, C\}$
3	3	$\{A, B, C\}$

Table 2: Encoding of values of enum node 'Cnt'

Real variables A and B have value from interval $[100, 200]$, real variable C from interval $[50, 100]$. All three variables A, B and C may have $NULL$ value, if they are not selected by the composition Root.

$NULL$ values are internally encoded by binary mask. $Non - NULL$ nodes have mask 1 and their value is encoded as described above, whereas $NULL$ nodes have mask 0 and their value is standartly set to zero (It must never be set to special value Inf, $-Inf$ or NaN).

Let us focus on geometric interpretation of solution space; Each possible variable composition value yields a generalized n-dimensional polygon, as shown on figure 7.

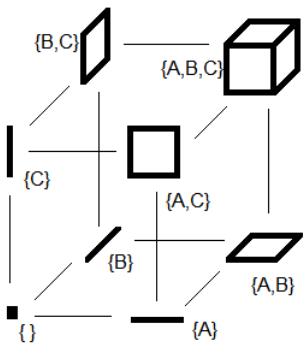


Figure 7: Geometric representation of composition with three subcomponents

Solution space is union of these disjunct polygons. We have to be careful, though, because not all these solutions are valid. We may immediately dismiss

solution {}, because is not constructive: no Cnt value allows Root to have 0 subcomponents. Other generalized polygons, associated with appropriate Cnt value 1, 2 or 3 are constructive. Such solutions are, however, not necessarily feasible, because not all of them satisfy linear constraint defined for nodes A, B and C. We will evaluate feasibility in the following section.

6.2 Equivalences and Feasibility

Linear equivalence is derived from the linear constraints. Note that upper and lower bounds are also considered linear constraints. The linear equivalence and F-equivalence classes of the model Nodes is shown in table 3:

node	linear eq. class	F-equivalence class
Root	1	1
Cnt	1	2
A	2	3
B	2	3
C	3	4

Table 3: Equivalence classes of nodes in the example model

For Root and Cnt, no linear constraints are applied, thus falling into one class. A and B are also in one linear equivalence class, because they are both constrained in the same way by the constraint 11 and their lower and upper bounds are equivalent. C falls into separate class.

The F-equivalence further refines linear equivalence classes according to Node parameters. Incompatible node types of Root and Cnt automatically split them into separate classes.

6.3 Signatures

In previous section we have defined signatures and their equivalences, whose definition is based upon definition of F-equivalence. In this short section, we will take a closer look at the implementation of the signature mechanism.

From the implementation point of view, signature is a data structure created solutions produced by the evolutionary algorithms. Signature is used to classify solutions of discrete allocations according to some equivalence defined on the optimization model nodes. This equivalence is generally a parameter passed to the signature generator method. In this thesis, we generate signatures **based on F-equivalence**. [2]

[2] We also experimented with usage of linear equivalence, although by now only experimental results are available and deeper analytical insight into foolproofness must be carried out.

Format of the signature is array of integer values, one for each enum or real node. Static and composition nodes have signature defined as zero and are thus irrelevant.

Let's now look at the definition of signature values. For each node N_i, the defining equivalence defines value $equiv(N_i)$ that is equivalence class of node N_i, that is from set $\{1, .., maxEquiv\}$. We create list of all numeric values in the model, on all enum nodes in the model. This list defines for each numeric value its index $valueIndex(numEvalueation(X_i))$. As we know from section 4.2.1, discrete allocation have three types of values of enum nodes:

1. EnumNode N_i is not numerically evaluable. Signature of allocation for such node is $equiv(N_i)$ when $non - NULL$ or 0 when $NULL$ (in fact, we only need to recognize nullness).

2. EnumNode N_i is numerically evaluable, but it is $NULL$. Signature of allocation for that node is 0.

3. EnumNode N_i is numerically evaluable, and it evaluates to numeric value. Signature of allocation for that node is $-valueIndex(numEvalueation(X_i))$. Note that the value is negative. Negative value distinguish numeric evaluations from evaluations leading to real nodes.

4. EnumNode N_i is numerically evaluable, its value is referential and it evaluates to a real node. Let $\{N_{c_1}, N_{c_2}, ..., N_{c_k}\}$ is the chain of nodes of the numeric evaluation, where N_{i_1} is the source real node and N_{c_k} is the node N_i itself. Signature of allocation for node N_i is defined as:

$$\sum_{j=1}^{k} (maxEquiv + 1)^{j-1} \cdot equiv(N_{c_j})$$

5. RealNode N_i. Signature of allocation for node N_i is defined as $equiv(N_i)$.

Signatures are often tested for equivalence. Signatures are equivalent when there exists a permutation of nodes inside defined equivalence classes, such that application of the permutation on one signature yields array equal to the other signature. To simplify that process, signatures are translated into **canonical form**. In canonical form, signature of values in each equivalence class are sorted.

Let's now look at concrete F-equivalence signatures of discrete allocations in our example. Each of the generalized polygons (excluding the empty one) corresponds to exactly one constructive discrete allocation. We will thus call all these allocations by their set of selected subcomponents. The signatures of the allocations are shown in table 4.

As we can see, canonical form only sorts signature values in F-equivalent classes, which is in our case on the 3th and 4th positions. Constant values $-1, -2, -3$ correspond to values of enum node Cnt $1, 2, 3$. We may see that discrete allocations for $\{A\}$ and $\{B\}$ are signature equivalent and so are discrete

allocation	non-canonical signature	canonical signature
$\{A\}$	(1,-1,3,0,0)	(1,-1,3,0,0)
$\{B\}$	(1,-1,0,3,0)	(1,-1,3,0,0)
$\{A,B\}$	(1,-2,3,3,0)	(1,-2,3,3,0)
$\{C\}$	(1,-1,0,0,4)	(1,-1,0,0,4)
$\{A,C\}$	(1,-2,3,0,4)	(1,-1,3,0,4)
$\{B,C\}$	(1,-2,0,3,4)	(1,-1,3,0,4)
$\{A,B,C\}$	(1,-3,3,3,4)	(1,-1,3,3,4)

Table 4: Simple signatures generated for discrete allocations in the example model

allocation for $\{A,C\}$ and $\{B,C\}$; On figure 8, classes of signature equivalent discrete allocations are outlined.

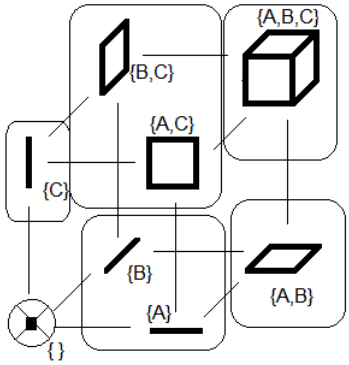


Figure 8: Signature equivalence classes of discrete allocations in the example model

For the equivalent discrete allocations, we only have to test feasibility of one and automatically get feasibility of all. It is a kind of symmetry, that is especially useful when there are lot of F-equivalent subcomponents of a variable composition. In our example, we have to test 5 discrete allocations instead of 7. If the variable composition twice as many subcomponents, adding nodes A2, B2 and C2 with the same properties as A, B, C (thus 6 subcomponents total from 2 different F-equivalence classes), thanks to signature equivalence we would have to test only 8 out of 41 constructive discrete allocations.

Without going into much detail, we will conclude that allocation $\{A,B,C\}$ is infeasible (sum of lower bounds of the three nodes is 250, which is over limit). All other constructive discrete allocations are feasible, though their polygons are sometimes truncated by the additional linear constraint 11 limiting sum of

real nodes to be at most 225. The resulting valid solution space has geometric interpretation of six generalized polygons, as shown on figure 9.

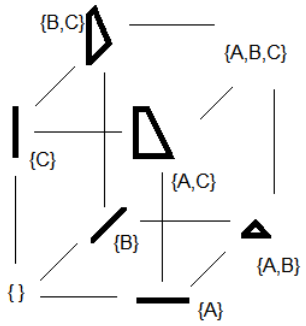


Figure 9: Geometric representation of constrained example model

Some of the generalized polygons are related to each other, because they share the same dimension. By EDA operators, solutions from one polygon may "inspire" solutions in different polygon with similar values in the shared dimensions.

In section 4.3.6, we provided an algorithm to efficiently generate all signature equivalent discrete allocations. It allows us to identify and repair solutions that are from unfeasible discrete allocations. The general correcting algorithm from section 4.2.3 is then able to correct any inconstructive or infeasible solution.

6.4 Further implementation details

More details about basic interface of the framework (data structures, algorithms and function interfaces) are to be found in appendices. Please note that it is not a complete function reference nor a user manual, but review of agreed specification of interfaces. The functions and data structures are partly business-defined and partly our original interfaces.

7 Tuning and testing of algorithm performance

In this section, we will outline processes, that will evaluate algorithm according to their experimental performance. In section 7.1, we will focus on definition and measurement of performance of algorithms. In section 7.2 we will present analytical problems used to specify our experiments. In section 7.3, we will show how parameters of algorithms were tuned. In section 7.4, we will describe experiments with algorithms to test their performance. In section 7.5, we will show results of that testing and comments to algorithm performance.

7.1 Algorithm performance

Although some attempts to analytically compute performance of genetic algorithms have been made, the research still mainly depends on empirical approach. In this chapter, we will describe experimental process used to assess performance of evolutionary algorithms using benchmarking testing fitness functions.

Bartz Beielstein [4] demonstrated that experimental approach to measuring performance may be convincing in many aspects. Specific nature of experimental research, however, must follow certain standards to overcome its intristic limitations. In this section, we will attempt to identify the limitations, decide if there is a way to overcome them and design procedures that will do so.

7.1.1 Measuring algorithm performance

Until now, we vaguely described our intention of finding and comparing "good algorithms" that find "fit solutions". In this and following sections, we will examine what could be exact definition of these criteria.

The primary performance measure of an algorithm is the fitness of the best solution:

- PM_Best: Best fitness among the population:

$$PM\_Best(D, f) = \max_{x^w \in D} fitness_f(x^w)$$

There may be, however, additional criteria to evaluate other important or interesting aspects of an algorithm. These *algorithm performance measures* may in the end influence our decision about which algorithm is the most suitable to solve a specific real-world problem. The importance of the algorithm best fitness criterion is, for example, its usage for algorithm tuning.

Usually the performance increases with greater number of generations. The problem is that longer algorithm runs tend to be more costly. Some algorithms may prefer more generations and smaller population, other algorithms operate better with fewer generations and bigger population, therefore we do not fix neither the population size nor the number of generations. All algorithm runs are limited only by number of fitness function evaluations. (even the CPU time needed to process the algorithm steps is neglected). The ratio of population size to number of generations is an additional parameter of all algorithms.

Let's take a closer look at some specific performance measures:

- PM_Average Average: average fitness of solutions in a generation

$$PM\_Average(D, f) = \frac{1}{N} \sum_{x^w \in D} fitness_f(x^w)$$

- PM_Reached: Number of reached peaks. We already know high fitness areas for some continuous testing functions. Each high fitness area is decided by "expert knowledge" of the particular fitness function shape.

Area is always continuous (in continuous dimensions) subset of solution space where fitness is greater than some limit value. Performance measure is number of these areas "visited" by the solutions in a generation.

$$PM\_Reached(D, f, limit) = |\{a \in highFitnessAreas(f, limit),\ \exists x^w \in D : x^w \in a\}|$$

- PM_Fail: Similar to PM_Reached, PM_Fail also measures ability of algorithm to reach pre-defined high-fitness zone. In this case, we don't count number of reached peaks, we only want the algorithm not to fail too often. Failure is defined as algorithm run not reaching that *limit*:

  $$PM\_Fail(D, f) = \delta(\forall x^w \in D :\ fitness_f(x^w) < limit_f)$$

  The $limit_f$ is computed by generating sizable initial population and averaging fitness of three highest obtained fitness values. The size of the population is set to be as high as number of total function evaluations available to the algorithm. This criterion effectively express if using algorithm outperforms random distribution. For one run, this measure take values 1 for failed run or 0 for successful one. If we measure multiple runs, aggregate PM_Fail measure is expressed as percentage of failed runs.

- PM_Outperform: This criterion measures relative performance of couples of algorithms. We create set of initial populations and run EDA optimization run starting from this population by many algorithms. Then we perform paired test of one algorithm performing better than the other one, paired by the same problem and initial population.

7.1.2 Importance of randomness

One of the biggest difficulty we meet when comparing performance of evolutionary algorithms is their stochastic nature. Randomness is essential for correct operation of the algorithms, however at the same time it limits our ability to draw conclusions from the experiments. Generally speaking, we can identify these major factors of success or failure of an algorithm runs (see section 5.3):

1. Initial population. Initial population is generated pseudorandomly. In our model, the problem definition already contains information how the initial values of all problem variables should be distributed. For discrete variables (non-compositions), probability of each value is specified. In continuous case, there is lower bound and upper bound for the variable. Initial population has uniform distribution for linear continuous variables, or logarithm of the variable is uniformly distributed in case of logarithmic variables (See section 4.4.2 about nonlinear continuous variables). All values of all variables are generated independently. We must also keep in mind the generated solutions have to be corrected in order to satisfy all the problem constraints, as described in section 4.2.

2. Algorithms. Naturally, the EDA algorithm itself is a major factor of optimization success or failure. We use rather complicated models of solution space that imply necessity of cooperation of both discrete and continuous optimization procedures. Algorithms will mostly be composed of independent parts as indicated in section 5.

3. Parameters. Evolutionary algorithms usually need many parameters that specify their actual behavior, quantify limits for decisions at some points, decide particular method for subtasks that the algorithm needs to solve, etc. Parameter variables are usually designed specifically for each algorithm, although we intend to unify "similar" parameters of different algorithms and their variants. Finding parameter values is optimization task that has to be carried out before the algorithms are put into production stage on real-life problems. The finding of good parameter values is referred to as "algorithm tuning".

4. Selection method. Selection is sometimes a fixed part of algorithm proposal. However, in section 5.3 we demonstrated that selection may be actually considered modular component. In fact, selection procedure and its parameters may be well considered as part of the algorithm parameters.

5. Randomness of algorithms. This mostly refers to pseudorandom sampling of estimated models in the reproduction phase of the EDAs. Randomness is also widely used in selection procedure (except truncation selection method).

We will now focus at the meaning and importance of each of the listed factors. Our intention is to make comparison of performance of algorithms (point 2). For this, parameters plus selection (points 3 and 4) should be fixed to some "good" set of values, that will be determined by algorithm tuning. Influence of initial randomness and randomness during optimization (points 1 and 5) are disturbing factors. Influence of initial population may be minimized in such way, that same initial population will be passed to all compared algorithms. Nevertheless, some initial distributions may be beneficial for only part of the algorithms and suboptimal for another ones. Influence of randomness (point 5) cannot be mitigated; We have to rely on statistics and make sure that enough algorithm runs (with different initial populations) will be performed, so that the results are statistically significant.

7.2 Testing problems

In the field of evolutionary optimization, many testing functions have been proposed [3]. Functions often model tricky situations that are known to be hard for conventional (gradient-based etc.) optimization techniques. The proposed problems often contain various interactions between problem variables, deceptive functions etc. The existing proposed problems are divided to two categories,

discrete problems and continuous problems, both categories operation in unconstrained space. In this thesis, we will use these problems, and modify them so that they can be combined into constrained semidiscrete problems.

Sometimes, fitness functions are originally defined for searching minimum. In our examples, we always want to seek maximum. Minimum seeking functions are switched to maximum seeking simply by multiplication by -1. We also try to generalize fitness, so that is is not limited to 2 values 0 and 1.

In following sections, we will present various discrete and continuous fitness functions, that were implemented and used for construction of more complex fitness used in algorithm tuning and testing. These tuning and testing fitness functions will be described later in sections dedicated to tuning and testing.

7.2.1 Review of discrete problems

Discrete sum: Fitness function returns sum of indices of discrete values.

N-max: This function is derived from basic two-max function. Two-max function is defined in binary space and returns total number of ones or total number of zeros, whichever is greater. Function N-max allows more discrete values and returns count of the most repeated value. Maximums are solutions with all values equal. Algorithm may try to converge to whichever maximum, however, combinations of two solutions close to two different optima tend to be suboptimal themselves.

Checkerboard: Function takes $n = a^2$ input variables that are represented as square $a \times a$ cells. If the number of input variables is not square, the side of square has length $ceiling(\sqrt{a})$ and cells with no matching variable are left zero. Variables may have value 0 or 1. Value of fitness is 1 for every pair of adjacent square cells, whose value is different (by adjacent cells we mean orthogonal, or diagonal). The optimum for this fitness is a board where zeros and ones alter just like black and white fields on a checkerboard There are two optima, with inverted values; For example, when $a = 5$, global maxima are:

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$

Interesting property of this function is that if algorithms try to build several parts of checkerboards simultaneously, they would fail when these parts should be assembled together. The "correct" way of solving this problem is to capture the interaction between values of adjacent cells.

Six-peaks: *Head* is length of initial sequence in the solution, that have the same values. *Tail* is length of ending sequence in the solution, that have the same values. Fitness function six-peaks returns length of head or

length of tail, whichever is greater plus bonus. Bonus is only applied if both head and tail is bigger than 40% of the total length. Value of bonus is length of the total length. The maxima are at:

$$\overbrace{00...0}^{40\%}\overbrace{1...1111}^{60\%}$$

$$\overbrace{0000..0}^{60\%}\overbrace{1...11}^{40\%}$$

and two more solutions with reversed 0 and 1. There are two local maxima, to which some algorithm may wrongly converge: all zeros and all ones. The four global optima are relatively isolated and hard to reach.

7.2.2 Review of continuous problems

In this section, we assume that continuous variables are roughly in interval $(-1; 1)$, although exact shape of the solution space may be further constrained. Real variables being not in such interval are normalized to it. More information on the process of normalization in section 5.4. Constants defining the testing functions were adjusted to achieve good shape in the defined zone.

Sphere: The function is very straightforward and basically equals euclidean distance from the center at 0:

$$fitness_{sphere}(x) = n - \sum_{i=1}^{n} x_i^2$$

There are no local optima and global optimum is at 0.

Griewank: Function is similar to sphere, but adds various local maxima:

$$fitness_{griewank}(x) = 1 - 2.5 \cdot \sum_{i=1}^{n} x_i^2 + \prod_{i=1}^{n} \cos\left(\frac{5}{\sqrt{n+1}} \cdot x_i\right)$$

The maximum is at point 0 and its value is 2.

Rosenbrock (modification): This is a modified version of Rosenbrock function. Fitness value is derived from the distance from a multidimensional parabola, with peak added at $[0.5, 0.5, .., 0.5]$. The function is defined as:

$$fitness_{rosenbrock}(x) = n - \log\left(100 \cdot \left(0.5 \cdot x_1 - \sum_{i=2}^{n} x_i^2\right)^2 + \sum_{i=1}^{n} (0.5 - x_i)^2 + 0.0001\right)$$

Rosenbrock function does not have any local maxima, only one global maxima. The "difficulty" is that algorithm tend to converge near to $[0, 0, .., 0]$ and then it starts to "rediscover" the real optimum. However, following gradient on the nonlinear shape valley is hard.

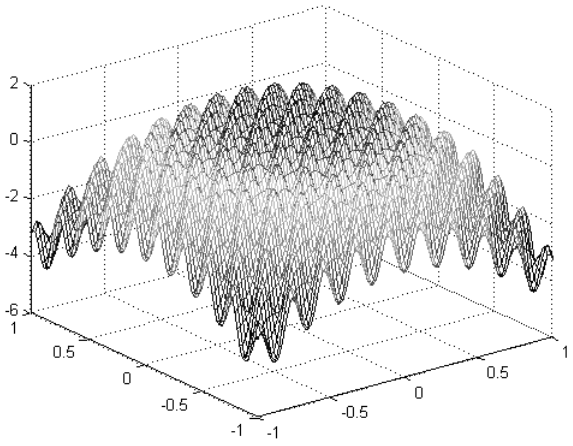


Figure 10: Griewank fitness function

Summation cancellation: Summation cancellation creates series of linear dependencies among input variables. The function is defined as:

$$y_j = \sum_{i=1}^{j} x_i \cdot (-1)^{j-i}$$

$$fitness_{summation}(x) = \log\left(\frac{100}{10^{-3} + \sum_{i=1}^{n} |y_i|}\right)$$

The only local optimum is at $[0, 0, .., 0]$ and its value is $5 \cdot \log(10)$. Function must minimize sums for all y_j, but change of a single variable has effect to many sums. In higher dimensions, this is very difficult. Graph of the fitness in two dimensions follows on figure (although it does not really explain the difficulty).

7.2.3 Design of semidiscrete problems

In this section, we will introduce methods used to create semidiscrete problems, combining both discrete and continuous problems and adding dependencies between them. Some of the discrete and continuous problems are easily adaptable to $NULL$ values, others can be hardly modified that way. For fitness functions evaluation, we don't use $numEvaluate$ numeric evaluations, but normalized values. but direct index of value is used for evaluating

We propose three methods of creating semidiscrete fitness:

Simple fitness composition: This is the simplest approach. Fitness is computed as sum of discrete and continuous fitness functions. No special

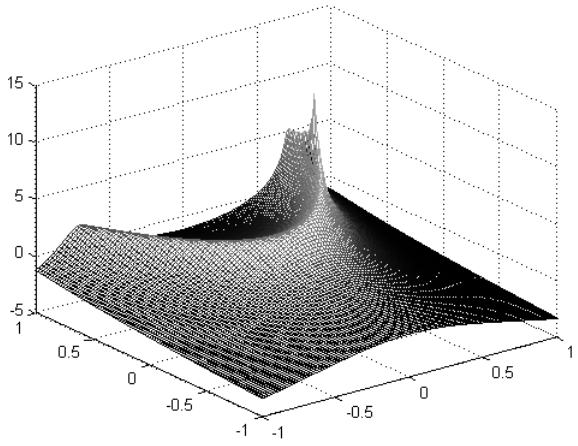


Figure 11: Rosenbrock fitness function

interactions are modeled.

Shift fitness composition: fitness is composed of 2 problems, one for discrete variables (discrete problem is optional) and one for continuous part of the population. Discrete variables "shifts" values of real variables in interactions defined by shifting matrix (matrix is set of vectors that for each discrete variable decides coefficient how much it displaces values in the real population). In order to get to the optimum of continuous problems, continuous part must compensate shift created by discrete variables. Shifting creates one-way interaction from discrete variables to continuous variables. Lets suppose solution x consist of discrete part x_{disc} of length n_{disc} and x_{cont} of length n_{cont}. The shift fitness composition is defined by shifting matrix $Shift$ (of size $n_{cont} \times n_{disc}$):

$$x_{shift} = x_{disc} \times Shift$$

$$fitness(x) = fitness_{disc}(x_{disc}) + fitness_{cont}(x_{cont} + x_{shift})$$

The shift matrix can model any linear set of interaction from discrete variables to continuous ones.

Merge fitness composition: transforms enum population (stressing enum, compositions are left without any effect) into additional real values and evaluates them as real population together with the real variables. One or usually many discrete variables will merge into a new real variable, added to the set of standard real variables. The additional variable is linear combination of numeric evaluations of the merged enums. The combination is normalized, in the way similar to normalization of real values (see normalization in section 5.4).

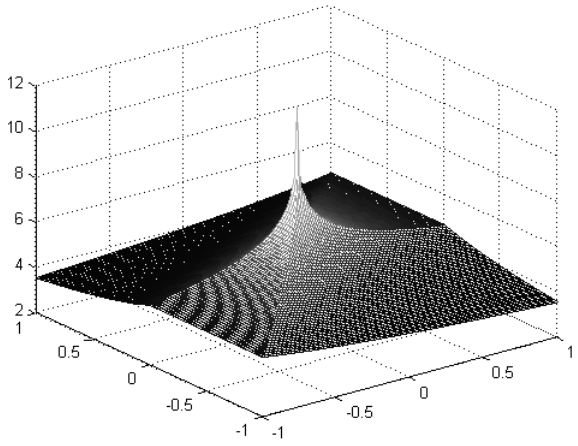


Figure 12: Summation cancellation fitness function

Now the exact definition will follow. Parameters $mergeWeights$ defines weights, that each variable has in the transformation. Let us have $mergeWeights = (w_1, w_2, ..., w_g)$ vector of length g where

$$\sum_{i=1}^{g} w_i = 1$$

Let population has count $k \cdot g$ of enumNodes, labeled as $x_{enum,j}$ for $j \in \{1, .., k \cdot g\}$. Then k new real variables $x_{merg} = \{x_{merg,1}, ..., x_{merg,k}\}$ are created. Each of that new merged variables $x_{merg,j}$ uses sequence of values of nodes $x_{enum,(j-1)\cdot g+1}, ..., x_{enum,(j-1)\cdot g+g}$, that is j-th sequence of length g of enum nodes. Value of $x_{merg,j}$ is defined as:

$$x_{merg,j} = \sum_{i=1}^{g} w_i \cdot normalize\left(x_{(j-1)\cdot g+i}\right)$$

$$normalize(x_i) = \frac{numEvaluation\left(x_i\right) - \min numEvaluation\left(X_i\right)}{\max numEvaluation\left(X_i\right) - \min numEvaluation\left(X_i\right)} \cdot 2 - 1$$

$$fitness(x) = fitness_{real}\left(\left(x_{real}, x_{merg}\right)\right)$$

Function $normalize$ normalizes numeric evaluation of the variable to interval $(-1, 1)$. The usage of numeric evaluations of discrete nodes is specific, note that standard discrete testing functions use discrete indices instead of discrete values. Note that in the special trivial case when $g = 1$ and thus necessarily $mergeWeights = (1)$, the enums are treated exactly as normalized real variables where the numeric evaluation of the enum variable is placed in a real-value dimension.

Additionally, mergeFitness may take parameter of enum and real filters. Filtered variables are not passed to the fitness function.

7.2.4 Testing functions with NULL values

The metamodel allows solutions to have some values $NULL$, thus undefined for the usual fitness functions. We have already devised a methods to process $NULL$s in the EDA algorithm. We would like to be able to test performance of the algorithms when $NULL$ values are present. Standard testing fitness functions are not designed to deal with $NULL$ functions. Naive way to include $NULL$ values into benchmarking may be to assign predefined value to them;

Our premise is to allow solutions to reach global optimum even with $NULL$ values. To discourage convergence to easier solutions with plenty of $NULL$ values we include slight penalization for each $NULL$ value. The penalization may be constant or pseudorandomly generated and specific for each dimension, thus labeling some dimensions as more valuable than others. Algorithms with specified number of selected variables need to seek those dimensions and put $NULL$s on the less valuable.

Some variables may be filtered out from the fitness evaluation. Those are especially auxiliary enum variables serving as count for variable compositions. They are completely excluded from fitness evaluation and are not penalized.

7.3 Algorithm tuning

Evolutionary algorithms can be tuned by adjusting values of several parameters. Often there are no obvious self-explanatory values for algorithm parameters and learning of algorithm parameters must be performed before comparing its performance with other algorithms.

Parameters may be learned to adjust algorithm to specific conditions given by certain optimization structure and fitness function, which is however not our case. We want to have a globally acceptable parameter values, that we would be able to use for unknown new problems.

7.3.1 Overfitting

Algorithm parameters optimized for some problems may not be optimal generally. When a learning process is fixed to certain problem, it eventually starts to gather too problem-specific knowledge and loses its generality. This phenomenon is known as *overfitting*. The philosophy of algorithm tuning is to prepare the algorithms to solve yet unknown future problems without overfitting. Two measures are taken to avoid that:

1. Learning will be performed on multiple sample problems to increase generality. This measure will directly lower possibility of overfitting, because parameters efficient for bigger number of problems are more likely to be efficient generally.

2. When tuned algorithms are finally compared, the comparison will be done on a different set of testing problems, independent from the tuning problems. This measure itself will not prevent phenomenon of overfitting, but it will to some extent measure ability of tuned algorithms to overcome overfitting. In fact, if the algorithms were compared on the same problems used for tuning, overfitting would be wrongly perceived as positive effect on algorithm performance, rather than an obstacle.

Nevertheless, expert knowledge is important at the beginning of tuning, substantially simplifying the process.

In the tuning process, we were limited by big costs of computation, thus we only used low number of tuning functions. If the resources allowed us to have more tuning functions, we would highly recommend using cross validation. Parameters would be adjusted by results of tuning of all functions except for a few functions, whose result would be used for validating the learned parameters. By different marking of function results as learning and validation sets, we would obtain multiple settings of learned parameters. We would select the parameters with best validation results.

7.3.2 Factorial Designs

In this section, we will focus on experimental designs, as presented by Bartz-Beielstein [4]. Process of tuning the algorithm parameters values is an optimization task per se. Tuning takes place in a parameter space, space of all parameter value's combinations. In order to simplify this space, set of possible values of each parameter will be decided by expert knowledge. To further simplify the parameter space, we will model it as Cartesian product of p parameter variables $P = P_1 \times P_2 \times .. \times P_p$, each of that are either continuous interval $[-1; 1]$, or discrete set $\{-1; 1\}$. Experiment settings is a value from the parameter space: $p \in P$.

The **naive design** of parameter tuning is optimizing one parameter at a time, while rest of the parameters is fixed to some value $\overline{p_i}$, that is 0 in the continuous case and any of the values in discrete case. To capture parameter influence in the parameter space more realistically, more complex methods have been proposed. Well known approach is **Full factorial design**, that takes two or more values of each parameter and plans an experiment of all combinations.

The good parameter space exploration and moderate number of experiments are combined in third design, the **fractional factorial design** (FFD). Fractional factorial design is defined as subset of two-valued full fractional design; Part of the parameters are modeled as in the full fractional design and additional parameters are set accordingly to interaction values. In the example in table 7.3.2, four basic factors ($p_1 = A$, $p_2 = B$, $p_3 = C$, $p_4 = D$) and four additional parameters are modeled ($p_5 = A{\cdot}B{\cdot}C$, $p_6 = A{\cdot}B{\cdot}D$, $p_7 = A{\cdot}C{\cdot}D$, $p_8 = B{\cdot}C{\cdot}D$).

If we wanted to model eight parameters using Full Factorial Design, we would need 256 experiments, thus at 16 times bigger cost. The price we pay using fractional factorial design is possible confusion of variable interaction; In

experiment	A	B	C	D	ABC	ABD	ACD	BCD
1	1	1	1	1	1	1	1	1
2	1	1	1	-1	1	-1	-1	-1
3	1	1	-1	1	-1	1	-1	-1
4	1	1	-1	-1	-1	-1	1	1
5	1	-1	1	1	-1	-1	1	-1
6	1	-1	1	-1	-1	1	-1	1
7	1	-1	-1	1	1	-1	-1	1
8	1	-1	-1	-1	1	1	1	-1
9	-1	1	1	1	-1	-1	-1	1
10	-1	1	1	-1	-1	1	1	-1
11	-1	1	-1	1	1	-1	1	-1
12	-1	1	-1	-1	1	1	-1	1
13	-1	-1	1	1	1	1	-1	-1
14	-1	-1	1	-1	1	-1	1	1
15	-1	-1	-1	1	-1	1	1	1
16	-1	-1	-1	-1	-1	-1	-1	-1

Table 5: Fractional factorial design with 8 factors

previous example, we are unable to differentiate effect of two factors combination $p_1 \cdot p_2$ $(A \cdot B)$ and two factors combination $p_3 \cdot p_5$ $(C \cdot ABC)$. In such case, we have to decide by expert knowledge which of the confounded combinations is more likely.

By choosing the right design, we may ensure that main factors or two factors combinations will only be confused with combinations of high number of other factors. The separation of factors is referred to as **resolution** of the design. In our example, we have defined resolution IV design.

We use k^p notation for full factorial designs, where k is number of values in each dimension and p number of dimensions. For fractional factorial designs, notation 2^{p-d}_r is used, where p is number of factors, d is number of additional factors whose settings is derived from settings of basic factors and r is the resolution. In our example of fractional factorial design, we have presented 2^{8-4}_{IV} design.

7.3.3 Tuning process

We will determine set of tuning functions, that will be used for optimization of parameters of the EDA algorithms. For each algorithm, fractional factorial design will be used to set up the experiment parameters. In our evolutionary algorithms, we have identified 13 parameters, although they are never used all together. Most of the parameters are by their nature two-valued. Some parameters may have more values, typically real numbers from an interval. For each of these parameters we will provide four possible values and each parameter will be modeled using two factors; One factor will decide low values or high

values, the other factor will decide central values or extreme values.

Since we are able to limit number of factors to 15, we decided to mostly use design 2_V^{15-7}, that is defined as:

$$A, B, C, D, E, F, G, H, ABDG, ACEG, BCFG, ABEH, ACFH, BCDH, ABCDEFGH$$

The number of variables may vary, leaving unused factors, these will be simply omitted, creating derived design, for example 2_V^{13-5}, by omitting factors $BCDH$ and $ABCDEFGH$.

Resolution V is enough to distinguish any two-factor interactions, it may confound interactions of two factor and three factors, or one factor and four factor. It is able to distinguish influence of both 2-valued and four-valued parameters. It is also able to distinguish two parameter interactions, limited to interaction with high/low or central/extreme values for 4-valued parameters.

Tuning is performed over multiple tuning problems, and overall performance is computed. Each problem may have fitness values in quite different intervals and simple sum of fitness values may be misleading. We calibrate fitness function f for each problem to return fitness relatively to fitness of randomly generated solutions: let μ_f is average fitness and σ_f is estimated standard deviation of random solutions for a tuning problem function f, then calibrated function $\hat{f}$ is defined as follows:

$$fitness_{\hat{f}}(x) = \frac{(fitness_f(x) - \mu_f)}{\sigma_f}$$

The purpose of calibrating is to make fitness functions comparable, so that high-fitness problems do not have bigger significance than low-fitness ones. By no means do we assume, that the distribution of fitness among solutions is normal or close to it - for example, because there is usually well defined upper limit for fitness.

In tuning process, we assume that the most appropriate performance measure is fitness of best generated solutions (PM_Best), because this is the most important testing performance measure. Overall performance of an algorithm over multiple tuning functions is computed as sum of performances of calibrated tuning functions.

Tuning is thus performed for each algorithm, for each settings of factors (given by fractional design) and each tuning function. For each algorithm, we compute best parameter settings from these fractional design experiments, using following algorithm:

1. We compute results for each single factor and propose usage of the better performing value.

2. For all two-factor pairs, we test if some combination of values yields better results, than simple combination of values proposed in step 1. If so, we store record of that change suggestion.

3. We order these change suggestions by statistical significance (i.e. p-value) of outperforming the original factor values, starting from the most obvious changes.

4. We perform two-factor change suggestions in their order, provided that they would not change values that appeared in any of the preceding change suggestions.

Computed factors are used to generate parameter values for both 2-valued and 4-valued factors. We were unable to test all possible parameter combinations, but we have insight to factor interactions to some degree. Which parameter combination performs the best may depend on selection of tuning algorithms. Thus we do not seek one best combination of parameters, but rather parameters reasonably competent for various optimization tasks.

7.3.4 Tuning functions

We have constructed set of tuning functions, each defined with appropriate optimization structure. The functions are designed to tune specific abilities of the algorithm, but not necessarily all at the same time. The structures are mostly designed as two trees of discrete and continuous variables. The tuning functions and structures are as follows:

1. Structure with 8 real and 8 three-valued enum variables, selecting 2 or 3 of each group (thus including two more enum variables for count of variable compositions, that are however not included in fitness computation). Structure has linear constraint: sum of odd variables must be equal to sum of even variables. Fitness is constructed by Simple Fitness Composition, composed of Griewank function and Discrete Sum function. Dimensions are penalized pseudorandomly, favoring some dimensions to be selected and others to be left $NULL$. This tuning function has not very complex shape, but forces algorithms to behave well in models where $NULL$s are frequent.

2. Structure has 3 real and 3 nine-valued discrete non-$NULL$ variables. Merge Fitness Composition is used to process all values by Summation Cancellation function. Function tests ability to solve problems with limited dimensionality, but complex interactions. High number of values defined for discrete variables is another feature.

3. Structure has 15 three-valued discrete variables and 6 continuous, all non-$NULL$. Shift Fitness Composition is used. Fifteen discrete variables are evaluated by Sixpeaks function, while six continuous variables are sent to simplistic Sphere function. For each discrete variable, there is pseudorandomly generated set of continuous variables that interacts with certain intensity, and selected discrete values displace continuous population. Tuning function simulates interactions between discrete and continuous variables and puts more stress on the ability to solve discrete problem.

Because the tuning process is expensive, we want to keep the list of tuning functions limited and specialized.

7.4 Algorithm testing

7.4.1 Tested algorithms

In chapter 5, we presented three frameworks for construction of semidiscrete algorithms. Using appropriate existing discrete, continuous algorithms from section 3.4, we are able to construct 20 composed algorithms; The eight most natural combinations are:

- Simple combination of UMDA, UMDAc
- Simple combination of MIMIC, MIMICc
- Simple combination of EBNA, EGNA
- Shared clustering for UMDA, UMDAc
- Shared clustering for MIMIC, MIMICc
- Shared clustering for EBNA, EGNA
- ClustEBNA with EBNA
- ClustEBNA with MIMIC

Twelve more algorithms may be created by combinations of different discrete and continuous model types, like UMDA and EGNA, or MIMIC and UMDAc. Even more algorithm combinations may be created using non-shared clustering, which we decided not to include.

For reference, we also included adapted version Simple genetic algorithm and Evolution strategies. We did not mix, however, these conventional genetic algorithms with EDA algorithms, because their crossover operator is local in nature and conceptually different from estimation of overall distribution.

7.4.2 Testing procedure and testing functions

Once we have set of selected evolutionary algorithms and have its parameters tuned, we will perform testing runs in order to compare their performance over set of testing functions. To avoid overfitting of parameters (i.e. situation when the parameters are very fit for testing functions only and poor for other functions), we have different set for testing and validation.

For each testing problem we generate set of initial populations. Initial population distribution is defined by the optimization structure and not by algorithm. For each algorithm, each testing problem and each initial population, test will be performed and the result stored. When we compare the algorithm performance, we will use paired T-tests, that should reduce variance of the results and bring more precise conclusions with less computing time.

For testing, we use analytical functions designed to challenge certain algorithm abilities, but also empirical functions derived from actual data. Now we are going to present the functions:

1. Fitness Shift Construction, with 8 two-valued discrete and 4 continuous non-$NULL$ variables. Zero to all discrete variables are selected, leaving the others $NULL$ with slight penalization. Fitness function combines Rosenbrock continuous function and N-max discrete function, while weight of the continuous part is increased. The shift is modeled by pseudorandom matrix in the same way as we modeled it before for the tuning functions.

2. Fitness Shift Construction, with 16 two-valued discrete and 6 continuous non-$NULL$ variables. Real variables are constrained that sum of odd values and sum of even values is equal. Fitness function combines Griewank continuous function and Checkerboard discrete function, where main weight is on the discrete problem. The shift is again modeled by pseudorandom matrix.

3. Last function is surrogate model of an unknown empirical function, describing the yield of synthesis of HCN depending on the composition of materials catalyzing that reaction [10]. The surrogate model is based on radial basis function networks, kindly provided by M. Holeňa and L. Bajer [1]. The function is operating within optimization structure with one discrete variable "support" with values $1, 2, .., 15$ and two or three out of eleven continuous variables from interval $(0, 100)$. Those variables represent proportions of chemicals in the catalyst and their sum is100. We were given three instances of the model, thus three different fitness functions operating in the same optimization structure.

Our budget is 2000 function evaluations in each algorithm run.

7.5 Experimental results

We have run tuning and testing process for our 20 algorithms. The results of testing are measured in criteria PM_Best, PM_Fail and $PM\_Outperform$, that discussed in section 7.1.1. We use calibrated fitness values (described section 7.3.3) and show results for each function and for all three testing functions averaged (functions identified in the table header by their continuous parts in the table of results). Outperforming is done for statistical significance $\alpha = 0.05$, and excluding conventional genetic algorithms because of their rather specific results. Failures are expressed in percents. Four best performing algorithms in each category were marked in bold font (or more if equally good results can reach the best places). The results are shown in table 6.

Generally, we may say that the most satisfactory performance was shown by EBNA and EGNA algorithms. Mixtures were successful, but tuning proved that low number of clusters is preferable (the lowest number 3 was picked). This outcome may be caused by low population sizes due to low number of

Algorithm	overall	Rosenbrock	Griewank	HCN	Outperf.	Fails %
UMDA/UMDAc	3.4931	4.1409	3.1214	3.2171	0	15.38
UMDA/EGNA	3.7096	4.6439	**3.1625**	3.3225	0	**3.85**
UMDA/MIMICc	3.6367	4.5128	2.9914	3.4058	0	7.69
EBNA/UMDAc	3.5604	4.3741	3.0343	3.2727	0	13.46
EBNA/EGNA	**3.8052**	**4.9554**	3.0228	3.4374	**5**	**5.77**
EBNA/MIMICc	**3.7455**	4.7083	3.0925	3.4357	0	7.69
MIMIC/UMDAc	3.4809	4.1170	2.9418	3.3839	0	23.08
MIMIC/EGNA	3.5886	4.7544	2.9919	3.0194	1	7.69
MIMIC/MIMICc	3.6698	**5.0566**	2.9719	2.9810	0	9.62
UMDA/UMDAc/Kmeans	3.5927	4.1910	**3.2298**	3.3573	0	9.62
UMDA/EGNA/Kmeans	3.7429	4.6668	**3.2389**	3.3229	1	**5.77**
UMDA/MIMICc/Kmeans	3.6701	4.7852	**3.1732**	3.0520	1	11.54
EBNA/UMDAc/Kmeans	3.6471	4.1885	3.0691	**3.6836**	0	15.38
EBNA/EGNA/Kmeans	**3.7663**	**4.8841**	2.9825	3.4324	**11**	**5.77**
EBNA/MIMICc/Kmeans	3.6084	4.2357	2.9134	**3.6760**	0	**3.85**
MIMIC/UMDAc/Kmeans	3.3604	4.1861	2.9107	2.9843	0	11.54
MIMIC/EGNA/Kmeans	3.4921	4.5215	2.8964	3.0584	**2**	**5.77**
MIMIC/MIMICc/Kmeans	3.4651	4.3572	2.9976	3.0405	0	13.46
clustEBNA/EBNA	3.6868	4.5507	3.1094	3.4003	0	11.54
clustEBNA/MIMIC	3.4746	4.0205	3.0127	3.3907	0	17.31
S. Genetic Algorithm	3.3568	3.8062	2.5166	**3.7475**	-	25.00
Evolution Strategies	**4.2247**	**5.1996**	3.0096	**4.4649**	-	9.62

Table 6: Experimental results

total available function evaluations. ClustEBNA algorithms were performing decently, and the the EBNA variant is preferable. We expected the algorithms to have lower rate of failures (measure PM_Fail), which was about 10%.

Performance of adapted Simple genetic algorithm was mixed. It had outstanding performance on empirical functions, but poor for analytical ones, especially the fitness with Rosenbrock function. We assume that binary representation of continuous variables is not the most efficient approach.

Performance of adapted Evolution Strategies are very impressive. They have the best overall performance. High performance over HCN fitness function was pushed high by few very good optimization runs.

Tuning and testing has been run on intel core 2 duo processor and took approximately 2.5 days each task, 5 days in total. Bottleneck of the computation was getting values of the empiric fitness function HCN. Next possible bottleneck were linear corrections, although they were significally less demanding of CPU time compared to the empiric function requirements. (We used Matlab implementation of linear programming).

8 Conclusion

In this thesis, we have designed and implemented framework of EDA algorithms that operates in solution space defined by user accordingly to our metamodel. We have reviewed and incorporated several proposed existing algorithms. We have overcome major obstacles that evolution in complex models create, most importantly the requirements for feasibility and ability to correct solutions. The corrections are based on automatic processor of models, that searches the user models for presence of precisely designed equivalences. Equivalences simplify the model instances to equivalence classes that we are able to classify accordingly to the feasibility properties. Our solution corrections is heuristic and we have proven that given the nature of allowed models, there is no silver bullet solution. We have tested our approach on multiple structures from real life scenarios and obtained results with reasonable system requirements. We have performed testing optimization runs derived from regression of actual real-life data.

The framework is designed for real-life usage, particularly for evolving catalysts for chemical synthesis. The API is designed in such a way that is ready to be connected to experiments via shared database. We have learned that with the very limited budget on number of function evaluations, we may be unable to find the best solution if the fitness function is complex enough. Using the EDA algorithms, we also mostly converge to most promising solution spot, rather than estimating whole high-fitness area. However, we succeeded to converge to good solutions in most of the cases.

The evolutionary approach is not the only possible way of optimization in our business domain. Due to low number of allowed function evaluations, evolutionary approach should be compared to other competing methods. Attention may be focused on learning of probabilistic models by both positive (high fitness) and negative examples (low fitness), or samples from previous generations. To learn distribution of high-fitness areas in the solution space instead of converging to one spot, niching techniques should be employed and number of evaluations should be increased.

References

[1]

[2] Shumeet Baluja. Population based incremental learning: A method for integrating genetic search based function optimization and competitve learning. Technical report, 1994.

[3] Shumeet Baluja and Scott Davies. Using optimal dependency-trees for combinational optimization. In *ICML '97: Proceedings of the Fourteenth International Conference on Machine Learning*, pages 30–38, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[4] Thomas Bartz-Beielstein. *Experimental Research in Evolutionary Computation: The New Experimentalism (Natural Computing Series)*. Springer, 2006.

[5] Hans-Georg Beyer and Hans-Paul Schwefel. Evolution strategies –a comprehensive introduction. *Natural Computing: an international journal*, 1(1):3–52, 2002.

[6] Jeremy S. De Bonet, Charles L. Isbell, Jr., and Paul Viola. Mimic: Finding optima by estimating probability densities. In *Advances in Neural Information Processing Systems*, page 424. The MIT Press, 1996.

[7] Peter A. N. Bosman. *Design and Application of Iterated Density Estimation Evolutionary Algorithms*. PhD thesis, Universiteit Utrecht, Utrecht, The Netherlands, 2003.

[8] Charles Darwin. *On the Origin of Species by Means of Natural Selection or the Preservation of Favored Races in the Struggle for Life*. Murray, London, 1859.

[9] Lan Gao and Youwei Hu. Multi-target matching based on niching genetic algorithm, 2006.

[10] S. Möhmel, N. Steinfeldt, S. Endgelschalt, M. Holeňa, S. Kolf, U. Dingerdissen, D. Wolf, R. Weber, and M. Bewersdorf. New catalytic materials for the high-temperature synthesis of hydrocyanic acid from methane and ammonia by high-throughput approach. *Applied Catalysis A: General*, 334:73–83, 2008.

[11] P. Larra naga, R. Etxeberria, J. A. Lozano, and J.M. Pe na. Optimization by learning and simulation of bayesian and gaussian networks, 1999.

[12] José A. Lozano Pedro Larrañaga. *Estimation of distribution algorithms: a new tool for evolutionary computation*. Springer, 2002.

[13] Martin Pelikan and David E. Goldberg. Genetic algorithms, clustering, and the breaking of symmetry. In *PPSN VI: Proceedings of the 6th International Conference on Parallel Problem Solving from Nature*, pages 385–394, London, UK, 2000. Springer-Verlag.

[14] Martin Pelikan and Heinz Muehlenbein. Marginal distributions in evolutionary algorithms. In *In Proceedings of the International Conference on Genetic Algorithms Mendel 98*, pages 90–95, 1999.

[15] Michele Sebag and Antoine Ducoulombier. Extending population-based incremental learning to continuous search spaces, 1998.

[16] Ross D. Shachter and C. Robert Kenley. Gaussian influence diagrams. *Manage. Sci.*, 35(5):527–550, 1989.

[17] Inc. The MathWorks. Learning matlab, 2001.

[18] David H. Wolpert and William G. Macready. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.

Appendix A: Used notation

$N = N_1, N_2, ..., N_n$ problem model nodes
n number of dimensions
$z_1, z_2, .., z_n$ instances (values) of nodes $N_1, N_2, .., N_n$ defined by optimization model
$X = X_1 \times X_2 \times ... \times X_n$ solution space variables (representation comprehensible to evolution algorithms)
$x = (x_1, x_2, .., x_n)$ instance of variable X
L_i node allocation
x_i^w value in i-th dimension of w-th instance of variable X
$x_i^{(k)}$ k-th value in value set of variable X_i
r_i size of value set of a discrete variable X_i
D dataset of instances of variable X
N number of instances in dataset D
$N_{i \bullet k}$ number of variable instances in D with k-th value on i-th position
S structure of a probabilistic graphical model
$p_l^t(X_i)$ learned distribution probabilities of discrete variable X_i at iteration t.
$f_l^t(X_i)$ learned distribution function of continuous variable X_i at iteration t.
$pf_l^t(X_i)$ learned distribution of discrete or continuous variable X_i - covers both cases, with respective meaning in each situation
$p_l(X_i = x_i^{(k)})$ parameter of learned marginal distribution of discrete variable X_i: probability of variable X_i having value $x_i^{(k)}$
$Pa_i^S = X_{j_1} \times X_{j_2} \times ... \times X_{j_K}$ variable composed of variables $X_{j_1}, .., X_{j_K}$ of parents of X_i in structure S
pa_i^S instance of variable Pa_i^S composed of parental values in x
q_i size of value set of discrete parental variable Pa_i^S
N number of variable instances in data D
N_{ij} number of variable instances in data D whose parents have j-th of their value combinations
$N_{i \bullet k}$ number of variable instances in data D with k-th value on variable X_i
N_{ijk} number of variable instances in data D that with k-th value on variable X_i and j-th parental value combination
C number of clusters
D^c dataset of population in cluster c
$\hat{m}_i$ estimated mean of variable X_i in the dataset D
$\hat{\sigma_{X_i^2}}$ estimated variance of variable X_i in the dataset D
$\hat{\sigma_{X_i X_j}}$ estimated covariance between variables X_i and X_j in the dataset D

Appendix B: Interface - Data structures

Optimization Structure (unparsed)

Optimization structure defines structure and properties of the model. It is primarily intended for testing of chemical catalysts. Our model is slightly more

general and abstract, so we transform this structure to Parsed Optimization Structure that is instance of the metamodel from section 4. Some information in Unparsed Optimization Structure is irrelevant respectively to evolutionary algorithms and was thus omitted. Optimization structure is an existing business-defined interface and the only mean of passing structural information to our algorithms.

OptimizationStructure is a struct with four fields:

- *KnownIdentifiers*
- *GlobalParameters*
- *Constraints*
- *ExternalFunctions*

KnownIdentifiers is a structure array with one entry for each node of the structure. Its fields may hold some types of values; Those types are boolean (0 or 1), identifier (unique string identifying node names), integer or real numbers. In the structure, we distinguish nodes for data structure and nodes for quantities. Quantities are nodes describing actual measurable chemical quantities (typically proportions of chemical components in a catalyst), while non-quantities are essentially structural metadata.

Structure *KnownIdentifiers* has following fields:

- *Identifier* - (identifier) name of the node
- *QuantityNumber* - 0 if not a quantity, otherwise a positive integer for index of the quantity
- *KnownComponent* - (boolean) (unused information for EDAs)
- *InProportion* - (boolean) determines if Proportion field is set
- *Proportion* - (identifier or real) describes proportion of a component. If it is identifier, it creates static reference to the referenced child node
- *PreparedUsing* - (boolean) determines if Preparation field is set
- *Preparation* - (identifier) method used for preparation of the component. Creates static reference to the referenced child node.
- *Evolvable* - (boolean) if set, field is ignored by EDA
- *Count* - (boolean) describes value serving as count for variable compositions
- *ComposedOf* - (boolean) defines compositions, both static and variable ones
- *ComposedOfRoot* - (boolean) identifies root node, always a composition

- *FromAmong* - (boolean) node is variable composition
- *HowMany* - (identifier or positive integer) number of subcomponents of a variable composition, that may be numeric or referential
- *Subcomponents* - (cell vector of identifiers) list of available subcomponents of static or variable composition
- *FromInterval* - (boolean) defines that node is real
- *LowerBound* - (identifier or real) lower bound of a real node, potentially referential value
- *LowerBoundMultiplier* - (real number) multiplier for referential lower bound
- *UpperBound* - (identifier or real) upper bound of a real node, potentially referential value
- *UpperBoundMultiplier* - (real) multiplier for referential upper bound
- *Precision* - (0 or real that is a power of 0.1) precision of the real node value
- *FromIntervalParameters* - ('linear' or 'loglinear') scale of the real variable, that influences distribution.
- *IsPrimitive* - (boolean) node is primitive, not a quantity or composition node.
- *OneOf* - (boolean) defines that node is enum
- *Choices* - (cell vector of identifiers and reals) values defined for the enum node.
- *DistributedAs* - (boolean) defines if distribution field is set
- *Distribution* - (vector of reals) initial distribution for enum node

Next member of Optimization Structure is *Constraints*. It is a struct describing linear constraints for the quantities (both reals and numerically evaluable enums), with following fields:

- *Aeq* - (matrix $number\_of\_rules \times number\_of\_constrained\_quantities$) matrix for linear equations
- *beq* - (column vector) right side of linear equations
- *Aineq* - (matrix $number\_of\_rules \times number\_of\_constrained\_quantities$) matrix for linear inequalities
- *bineq* - (column vector) right side of linear inequalities

- *Quantities* - (cell vector of identifiers) list of constrained quantities that are passed to the linear equations and inequalities
- *Precisions* - (cell vector of zeros and reals that are power of 0.1) same as *KnownIdentifiers(i).Precision*.

The linear constraints have following form:

$$Aeq \times x_{Quantities} = beq$$

$$Aineq \times x_{Quantities} \leq bineq$$

Quantities are subset of nodes that are constrained. $NULL$s are counted as zero and rules with no non-$NULL$ quantity are not evaluated at all.

The *GlobalParameters* is a struct that contains metadata in fields:

- *ExperimentInformation* – (string) Description of the experiment
- *PopulationSize* – (positive integer) Default size of the population (Population size is otherwise passed to EDA interfaces as a parameter).
- *CurrentExperimentId* – (string) unique identifying string for the experiment
- *ExperimentIdField* – (string) database field for experiment id as string
- *GenerationField* – (string) database field for number of generation as number
- *SequentialNrField* – (string) database field for number of individual in a generation as number
- *SimulationFlagField* – (string) database field to store special value.
- *ExternalIdsTable* – (string) unused for EDAs
- *ExternalIdsField* – (string) unused for EDAs
- *FeedbackTable* – (string) database table where information about solution fitness is stored
- *FeedbackField* – (string) database field for fitness of individuals
- *EvolutionMethod, EvolutionParameters*, *EvolutionField*, *OverviewTable* – fields are unused in current implementation.

The last field of Optimization Structure ExternalFunctions is unused in current implementation.

Database structure (unparsed)

Database is another object defined by business to pass information controlling the optimization process. It is not actually a MATLAB struct, but an array of structs. It is holding specifications for storing values of a model into database. Length of the database is 5 plus number of evolved model fields stored into the database. Before we move to describe meaning of entries and their fields, we will explain how the data is stored into a database. The composite key of an entry is *(ExperimentId, Generation, SequentialNr)* with types (VARCHAR, INTEGER, INTEGER). The values of the individual may be redistributed to multiple tables and each such table is indexed by the composite key. Each table also holds field *Simulation* (of type INTEGER) that holds information if an experiment is simulation. All the evolved data are stored in INTEGER (enum nodes and variable composition nodes) or DECIMAL (real nodes) database fields, where *NULL*s are stored as database *NULL*s. Name of the database field and table where node is stored is defined in entries behind the firs five entries.

Each entry has following fields:

- *Content* - (string) type of metadata or (positive integer) index of node that is stored into database
- *DataColumn* - (positive integer) index of the entry in the array
- *Field* - (string) name of the column in the database tables where node value or metadata (key) are stored
- *Table* - (string) table to store the value of a node (field is empty for metadata)
- *Precision* - (0 or real that is power of 0.1) describes DECIMAL type precision of the database field

First five entries are metadata: Overview (name of the experiment, field is unused in current implementation), ExperimentId, SequentialId, SimulationFlag, Generation. Following entries serve for the actual data.

Params structure

This structure is passed throughout the EDA algorithms and stores data about values of parameters. Fields of the structure are not fixed, *params* object is treated as associative array (map). It may contain any type of field. For any parameter, undefined field results in usage of a default value of a parameter. Parameters re passed to several components of the EDA framework, namely:

- EDA and feasEDA main functions
- EDA algorithms
- Selection algorithm

- Clustering algorithm

Because all parts use the same structure, care must be taken not to duplicate names of parameters in different parts. Parameters applicable to each function will be described in help to the function, or in appendix D: Function Reference, under subsection **Available params**.

Population (denormalized format)

Matrices *Population* and *PopulationNull* (also called null mask of population) represent one generation of an evolutionary algorithm. *Population* and *PopulationNull* are matrices ($population\_size \times number\_of\_nodes$). Population contains actual values: Real numbers for real nodes, Integer indices for enum nodes, integers for variable composition, 0 for nodes without defined values. *PopulationNull* has the same size and contains values 0 for $NULL$s and 1 for non-$NULL$s.

Normalized population

Structure for normalized population (see section 5.4). Population is divided into discrete and continuous parts. Nodes unused by EDA algorithms (e.g. static compositions) are completely excluded. Normalized population is a struct with following fields:

- *intPop*: discrete population (enums and variable compositions). It is a matrix ($popsize \times number\_of\_discrete\_nodes$) of either indices of selected enum value (starting from 1) or indices of combination of selected subcomponents.
- *intNull*: null mask matrix for *intPop* (same matrix size, values 0 for $NULL$s, 1 for non-$NULL$s).
- *realPop*: continuous population (real nodes). It is a matrix ($popsize \times number\_of\_continuous\_nodes$) of real values transformed to interval $[-1; 1]$. If the field has value $NULL$, the value in *realPop* is undefined. This rule is to simplify certain computations where replacing $NULL$s with mean value (instead of zero or other constant value) makes the computation simples or faster.
- *realNull*: null mask for the *realPop*
- *intMax*: (row vector of positive integers, length $number\_of\_discrete\_nodes$) Maximal allowed values for each of the discrete variables. Matrix *intPop* may contain in each column integer values less than or equal to the corresponding maximal value in *intMax*.
- *intComposition*: (row vector of integers, length $number\_of\_discrete\_nodes$) vector indicate which ints are variable compositions (value 1) and which are enums (value 0).

- *popsize*: (integer) size of the population
- *stat*: (0 or 1) indicate if simple or extended form of normalized population is used.

Extended form has also following fields where statistical data is included:

- *realMean*: (row vector) means of the real population, excluding nulls
- *realVar*: (row vector) variance of real non-null population
- *realCov*: (square matrix) covariances of real non-null population
- *realDef*: (row vector) percentage of real variables that are non-$NULL$
- *intDef*: (row vector) percentage of discrete variables that are non-$NULL$

Additional note for *intComposition* field: In special case, all variable compositions may be decomposed to number of two-valued enum variables, one per each defined subcomponent. Each two-valued variable indicates if the subcomponent is selected or not. Number of discrete variables in normalized population is then bigger than number of discrete nodes in the optimization model.

The decomposed variable can be distinguished by negative value assigned to the first decomposed two valued variable. The negative number also indicate to how many variables was the composition decomposed. For example, a composition selecting three out of seven variables will be decomposed to seven two-valued variables whose intComposition values will be $[..., -7, 0, 0, 0, 0, 0, 0, ...]$.

Feasibility structure

Matlab object holding data about feasibility of all classes of F-Equivalent solutions. Its interpretation is non-trivial; It contains information about classes and equivalences of discrete allocations and related data. See section 4 for more details. User does not have to understand the feasibility data. The structure is constructed by function *feasibilityStructure(Optimization)* and passed to *FEDA* function as argument. It is hard to compute feasibility data, so we recommend to store them once computed.

For reference, there are following fields of *feasibilityStructure*:

- *filter*: (row vector) filter for nodes that are constrained or dependent on constraints
- *nodeqdata*: (row cell vector) equivalence data for variable compositions. Format of entries will be described later.
- *equivalences*: (row vector of indices of equivalence class) F-equivalences of nodes
- *lineq*: (row vector of indices of equivalence class) Linear-equivalences of nodes

- *feasmap*: mapping signatures to cell triples *{feasible, foundSolutions, corrections}*. Signature describes one class of discrete allocations. Values of the triple represent

  - *1:* feasible: (0 or 1) according to feasibility of the class.
  - *2:* foundSolutions*:* (matrix of individuals) one or more solutions generating this class
  - *3:* corrections*:* (matrix of individuals or []) empty for feasible allocation classes, or selection of up to 5 solutions offered as correction if allocation class is infeasible.

  And there are methods for accessing the data:

  - *[feasibilityDataTriple, exists] = get(signature)*: retrieve feasibility data with given signature. Output argument *exists* indicate if such key exists in the structure
  - *[exists] = put(key, value)*: putting data into the map (*output exists* indicates if data already existed)
  - *clear()*: clears the data

  And data are held in attribute:

  - *data*: cell matrix ($number\_of\_entries \times 2$) of rows *{signature,* feasibilityDataTriple*}*

The structure was created using constructs from embedded Java language. There is no alternative due to system requirements of using these rather complex procedures.

Appendix C: Interface - Custom function handles

One of our goals is to make the implementation modular. Programmers may implement new algorithms, selection functions, testing fitness functions etc, that may be used in the very same environment as currently implemented algorithms. The new functions must have arguments according to these specified interfaces and must fulfill the expected behavior.

Interfaces are defining function handle input parameters of EDA framework functions. Function handles in MATLAB are created from existing functions using prefix "@" and they can be recognized by Matlab construct

$$isa(object,'function\_handle')$$

We provide many predefined functions that may be used for function handle parameters with the "@" prefix. Note that some functions (e.g. *composeAlgorithm*) serve as constructors for function handles, returning already prepared function handles, thus not needing the reference symbol "@" to be used.

Fitness function interface

fit = fitness(optim, pop)

- *optim*: parsed optimization structure
- *pop*: normalized population
- *fit*: returned column vector of fitness

Interface used for testing/tuning fitness function implementation.

Selection function interface

[parents, elite] = selectionFunction(pop, fitness)

- *pop*: normalized population
- *fitness*: row vector of fitness values
- *parents*: normalized population of selected parents
- *elite*: normalized population of elite population, that is good enough to be passed to next generation without modification.

Interface used for function of evolutionary algorithm selection.

Algorithm function interface

[nextgen, distribution] = algorithm(pop, distribution, nextgensize, params)

- *pop*: (normalized population or []) selected parents
- *distribution*: (distribution of the population or []) This is both input and output parameter. The actual format of distribution is not fixed and depends on the particular algorithm. The only requirement is that algorithm must understand the distribution it produces. If the distribution input parameter is empty [], the distribution will be estimated from the population *pop*.
- *nextgensize*: (non-negative integer) size of the population that should be sampled. Algorithm function returns sampled population if the *nextgensize* parameter is above zero, otherwise only distribution output parameter *distribution* is returned and *nextgen* is left empty.
- *nextgen*: (normalized population) returned population sampled according to distribution. Size of the population is *nextgensize*.

Output population is sampled either from distribution on input, or from distribution estimated from the population on input. Exactly one of input parameters *pop*, *distribution* must be defined.

Note that some constructions (functions *composeAlgorithms* and *semidiscreteClustebna*) use modified version of this interface, where *nextgen* output argument is not normalized population but its *intPop* or *realPop* fields only.

Appendix D: Interface - Function reference

This appendix is not a complete function reference nor a manual. In this appendix we state our specifications for the function interfaces. Standartly, some function arguments are optional and may be omitted by setting them as empty structures: in the text it will be noted as "[]", obviously because of the MATLAB syntax. Exact meaning of using [] is described in the function description.

FEDA

Usage:

[population, populationNull] = FEDA(optimStructure, feasibilityStructure, population, populationNull, fitness, selectionFunction, algorithm, nextgensize, params)

Function arguments:

- *optimStructure*: (unparsed optimization structure)
- *feasibilityStructure*: (feasibility object) feasibility computed for the optimization model
- *population*: (denormalized population or [])
- *populationNull*: (denormalized population or [])
- *fitness*: (column vector of fitness values, or fitness function handle)
- *selectionFunction*: (selection function handle or string) Custom selection function is used, or string identifying one of predefined selection methods; Predefined selection names are *'ranks'*, *'tournament'* and *'truncation'*.
- *algorithm*: (algorithm function handle) EDA algorithm used.
- *nextgensize*: size of the next generation
- *params*: (params structure) optional parameters

Details:

population and *populationNull* are both input and output parameters. When *population* and *populationNull* are both set to [], the initial population is sampled according to initial distribution rules given by optimization structure and corrections are applied. No EDA estimation is performed in such case.

Function *FEDA* creates initial population or performs one step of selected evolutionary algorithm. After new generation of population is sampled, corrections are applied. If some solutions are not correctable - they are from infeasible discrete allocation (see section 4.2.1 and following for more details), they are corrected to some other feasible discrete allocation. No solutions are discarded.

Available params:

- *simpleLinearCorrection*: (0 for classic or 1 for simple, 0 by default) Simple mode prefers to make bigger corrections to smaller number of real nodes, while Classic correction makes small corrections to bigger number of real nodes (orthogonally to hyperplanes of linear constraints)
- *verbose*: (0 or 1, 0 by default) prints information about algorithm runs and number of corrections of feasibility

And parameters applicable to standard selection methods (unless overriden by a custom selection method)

- *selectionSize*: (positive real, default 1.0) size of selected parental population relatively to population size, for ranks or tournament selection only
- *tournSize*: (positive integer, at least 2, default 3) size of the tournaments in tournament selection. Bigger the tournament, bigger selection pressure is put on the individuals
- *rankExponent*: (positive real, by default 1.0) exponent applied to numeric rank in rank selection. The bigger the exponent is, bigger selection pressure is put on the individuals
- *truncationSize*: (real from $[0, 1]$, by default 0.5) size of truncated population relatively to population size. For truncate selection only. Smaller the truncation is, bigger selection pressure is put on the individuals.
- *elitism*: (non-negative integer, by default 1) number of elitist individuals

EDA (lightweight version of FEDA)

Usage:

[population, populationNull] = EDA(optimStructure, population, populationNull, fitness, selectionFunction, algorithm, nextgensize, params)

Function arguments:

identical to FEDA function arguments, with exception of feasibilityStructure argument that is missing for EDA.

Details:

EDA function is lightweight variant of FEDA. EDA does not need feasibility information about the optimization model. When a sampled solution cannot be corrected by linear algorithm (i.e. within its discrete allocation), it is discarded. In order to sample enough solutions into next generation, population is being created in a loop, until the requested number is reached. Population is

always sampled in numbers reusing once estimated distribution. Size of population sampled in each round starts with slightly bigger number than *nextgensize* and in each successive round, number is twice as high. Repetitions continue until enough correct solutions are created or until maximal number of rounds is reached (In which case EDA function returns smaller than expected number of individuals).

The advantage of this lightweight variant is that the feasibility does not have to be computed. For some more complex models, exploration of all classes of discrete allocations may be too costly. The disadvantage is that sampling of next generation have to be repeated several times if proportion of discarded unrepairable solutions is high.

Available params:

Available parameters include all parameters for *FEDA* function, and additionally

- *maxEdaSteps*: (positive integer, 8 by default) maximal number of repetitions of sampling process
- *firstEdaStepMultiplier*: (real, 1.3 by default) relative size of the first created round of sampled solutions

databaseMysqlConnect

Usage:

conn=databaseMysqlConnect(databaseName, user, password)

Function arguments:

- *databaseName, user, password*: data for connection to the database. Contact the database administrator to obtain those.
- *conn*: MATLAB object holding connection to the database

Details:

Creates connection to the database. When the work with the database is finished, be sure to close the connection: *close(conn)*.

databaseSave

Usage:

databaseSave(conn, Database, Optimization, experiment, generation, population, populationNull, simulation)

Function arguments:

- *conn*: (database connection object)
- *Database*: (database structure)
- *Optimization*: (optimization structure)
- *experiment*: (string or []) id of the experiment (use [] for default experiment id defined by *Database*)
- *generation*: (positive integer) number of the generation
- *simulation*: (numeric value | NaN) sets simulation flag that value
- *population*: denormalized population to store in the database
- *populationNull*: denormalized population *NULL* mask

Details:

Inserts data to the database tables.

databaseLoad

Usage:

[population, populationNull] = databaseLoad(conn, Database, Optimization, experiment, generation)

Function arguments:

- *conn*: (database connection object)
- *Database*: (database structure)
- *Optimization*: (optimization structure)
- *experiment*: (string) id of the experiment
- *generation*: (positive integer) number of the generation
- *simulation*: (one of values numeric value | NaN) sets simulation flag that value
- *population*: denormalized population loaded from the database
- *populationNull*: denormalized population *NULL* mask

Details:

Loads population from the database. The individuals are ordered by sequential field.

databaseFitnessLoad

Usage:

fitness = databaseFitnessLoad(conn, Optimization, experiment, generation)

Function arguments:

- *conn*: (database connection object)
- *Optimization*: (optimization structure)
- *experiment*: (string) id of the experiment
- *generation*: (positive integer) number of the generation
- *fitness*: column vector of fitness loaded from the database feedback tables

Details:

Note that Database structure is actually not needed to load fitness. Fitness values are ordered by sequential field, having the same order as population loaded by *databaseLoad* function.

composeAlgorithm

Usage:

algorithm = composeAlgorithm(discreteAlgorithm, continuousAlgorithm, clusteringAlgorithm)

Function arguments:

- *discrete algorithm*: (Algorithm function handle, see details) EDA algorithm for discrete part
- *continuousAlgorithm*: (Algorithm function handle, see details) EDA algorithm for continuous part
- *clusteringAlgorithm*: (function handle) clustering algorithm as function handle of interface *clust = clustering(pop, params)*, where *pop* is normalized population and params is the param structure. Returning row vector *clust* of cluster indices for each individual in the population.
- *algorithm*: (Algorithm function handle) returned algorithm composed of the three parts, first dividing parental population by clustering and applying discrete algorithm on the discrete part and continuous algorithm to the continuous part.

Details:

This function returns function handle for algorithm in the FEDA function arguments. The partial algorithms have to satisfy the Algorithm interface with one difference, that the *nextgen* population output is NOT in normalized population format, but only its *intPop* or *realPop* fields (matrices of size $nextgensize \times number\_discete\_variables$ or $nextgensize \times number\_continuous\_variables$). By definition, at this stage there can be no $NULL$s. Available algorithm implementations are listed in section Available EDA algorithms below.

Available params:

- *decomposition*: (0 or 1) decides if variable compositions are decomposed to number of two-valued enum variables or left as single discrete variable with value for each combination of subcomponents.

semidiscreteClustebna

Usage:

algorithm = semidiscreteClustebna(discreteAlgorithm)

Function arguments:

- *discrete algorithm*: (Algorithm function handle, see details) EDA algorithm for discrete and discretized continuous parts.
- *algorithm*: (Algorithm function handle) returned algorithm, mapping continuous variables to dscrete space by one-dimensional clustering.

Details:

This function returns function handle for algorithm in the FEDA function arguments. The partial algorithm *discreteAlgorithm* have to satisfy the Algorithm interface with one difference, that the *nextgen* population output is NOT in normalized population format, but only its *intPop* field (matrix of size $nextgensize \times number\_discrete\_discretized\_nodes$). By definition, at this stage there can be no $NULL$s. Available algorithm implementations are listed in section Available EDA algorithms below.

Available params:

- *decomposition*: (0 or 1, 0 by default) decides if variable compositions are decomposed to number of two-valued enum variables or left as single discrete variable with value for each combination of subcomponents.
- *numSegments*: (positive integer, 4 by default) number of segments in each continuous dimension

- *minSegmentSize*: (positive integer, 5 by default) minimal number of non-$NULL$ variables needed for each segment.

Available EDA algorithms

In this section, we will list algorithms implemented for usage in *composeAlgorithm* and *semidiscreteClustebna* functions. The function interface is the modified algorithm function handle interface returning *intPop* or *realPop*.

We have implemented following algorithms with available parameters:

- discreteUmda, discreteMimic
  - *mutations*: (positive real number, default 0) weight of mutation to random discrete value
- discreteEbna
  - *mutations*: (positive real number, default 0) weight of mutation to random discrete value
  - *penal*: (non-negative real number, default 0.25) coefficient for penalization of maximum likelihood score
  - *maxParents*: (positive integer, default 3) maximal allowed number of parents of a node
- continuousUmda, continuousCmimic
  - *varmult*: (real number greater or equal to 1) multiplier for estimated standard deviation (for keeping the distributon more varied)
- continuousEgna
  - *varmult*: (real number greater or equal to 1, default 1.0) multiplier for estimated standard deviation (for keeping the distributon more varied)
  - *penal*: (non-negative real number, default 0.25) coefficient for penalization of maximum likelihood score
  - *maxParents*: (positive integer, default 3) maximal allowed number of parents of a node

And default clustering algorithm

- clusteringKmeans
  - *numClusters*: (positive integer, default 4) number of clusters
  - *clusterReals*: (0 or 1, default 1) indicate if clustering is done respectively to both discrete and continuous variables (value 1), or discrete values only (value 0). The latter is essentially clustering of discrete allocations.

– *clusterQualitative*: (0 or 1, default 1) indicate if discrete values are clustered as nominals or ordinals. The former assumes existing ordering of discrete values, thus putting numerically close values together, whereas the latter considers any two different values equally distanced.

feasibilityStructure (constructor of the structure)

Usage:

feasibilityStructure = *feasibilityStructure(Optimization)*

Function arguments:

- *Optimization*: (unparsed Optimization structure)
- *feasibilityStructure*: constructed Feasibility structure.

Details:

This is constructor for feasibilityStructure, that is a parameter for *FEDA* function. Construction may be time consuming task. If possible, keep this structure and reuse it for all generations.