Chapter 4
HEAD-DT: Automatic Design of Decision-Tree Algorithms

Abstract As presented in Chap. 2, for the past 40 years researchers have attempted to improve decision-tree induction algorithms, either by proposing new splitting criteria for internal nodes, by investigating pruning strategies for avoiding overfitting, by testing new approaches for dealing with missing values, or even by searching for alternatives to the top-down greedy induction. Each new decision-tree induction algorithm presents some (or many) of these strategies, which are chosen in order to maximize performance in empirical analyses. Nevertheless, the number of different strategies for the several components of a decision-tree algorithm is so vast after these 40 years of research that it would be impracticable for a human being to test all possibilities with the purpose of achieving the best performance in a given data set (or in a set of data sets). Hence, we pose two questions for researchers in the area: "is it possible to automate the design of decision-tree induction algorithms?", and, if so, "how can we automate the design of a decision-tree induction algorithm?" The answer for these questions arose with the pioneering work of Pappa and Freitas [30], which proposed the automatic design of rule induction algorithms through an evolutionary algorithm. The authors proposed the use of a grammar-based GP algorithm for building and evolving individuals which are, in fact, rule induction algorithms. That approach successfully employs EAs to evolve a generic rule induction algorithm, which can then be applied to solve many different classification problems, instead of evolving a specific set of rules tailored to a particular data set. As presented in Chap. 3, in the area of optimisation this type of approach is named hyper-heuristics (HHs) [5, 6]. HHs are search methods for automatically selecting and combining simpler heuristics, resulting in a generic heuristic that is used to solve any instance of a given optimisation problem. For instance, a HH can generate a generic heuristic for solving any instance of the timetabling problem (i.e., allocation of any number of resources subject to any set of constraints in any schedule configuration) whilst a conventional EA would just evolve a solution to one particular instance of the timetabling problem (i.e., a predefined set of resources and constraints in a given schedule configuration). In this chapter, we present a hyper-heuristic strategy for automatically designing decision-tree induction algorithms, namely HEAD-DT (Hyper-Heuristic Evolutionary Algorithm for Automatically Designing Decision-Tree Algorithms). Section 4.1 introduces HEAD-DT and its evolutionary scheme. Section 4.2 presents the individual representation adopted by HEAD-DT to evolve decision-tree algorithms, as

R.C. Barros et al., *Automatic Design of Decision-Tree Induction Algorithms*,
SpringerBriefs in Computer Science, DOI 10.1007/978-3-319-14231-9_4

well as information regarding each individual's gene. Section 4.3 shows the evolutionary cycle of HEAD-DT, detailing its genetic operators. Section 4.4 depicts the fitness evaluation process in HEAD-DT, and introduces two possible frameworks for executing HEAD-DT. Section 4.5 computes the total size of the search space that HEAD-DT is capable of traversing, whereas Sect. 4.6 discusses related work.

Keywords Automatic design · Hyper-heuristic decision-tree induction · HEAD-DT

4.1 Introduction

According to the definition by Burke et al. [7], "a hyper-heuristic is an automated methodology for selecting or generating heuristics to solve hard computational search problems". Hyper-heuristics can automatically generate new heuristics suited to a given problem or class of problems. This is carried out by combining components or building-blocks of human-designed heuristics. The motivation behind hyper-heuristics is to raise the level of generality at which search methodologies can operate. In the context of decision trees, instead of searching through an EA for the best decision tree to a given problem (regular metaheuristic approach, e.g., [1, 2]), the generality level is raised by searching for the best decision-tree induction algorithm that may be applied to several different problems (hyper-heuristic approach).

HEAD-DT (Hyper-Heuristic Evolutionary Algorithm for Automatically Designing Decision-Tree Algorithms) can be seen as a regular generational EA in which individuals are collections of building blocks of top-down decision-tree induction algorithms. Typical operators from EAs are employed, such as tournament selection, mutually-exclusive genetic operators (reproduction, crossover, and mutation) and a typical stopping criterion that halts evolution after a predefined number of generations. The evolution of individuals in HEAD-DT follows the scheme presented in Fig. 4.1.

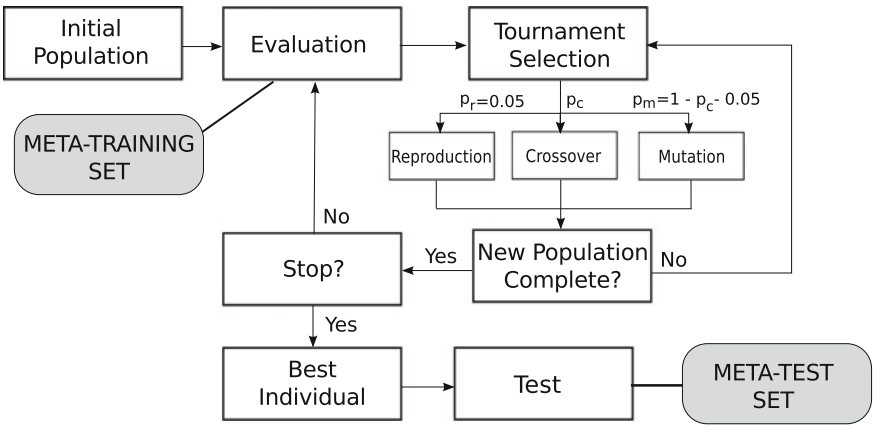

Fig. 4.1 HEAD-DT's evolutionary scheme

4.2 Individual Representation

Each individual in HEAD-DT is encoded as an integer vector, as depicted in Fig. 4.2, and each gene has a different range of supported values. We divided the genes into four categories, representing the major building blocks (design components) of a decision-tree induction algorithm:

- split genes;
- stopping criteria genes;
- missing values genes;
- pruning genes.

4.2.1 Split Genes

The linear genome that encodes individuals in HEAD-DT holds two genes for the **split** component of decision trees. These genes represent the design component that is responsible for selecting the attribute to split the data in the current node of the decision tree. Based on the selected attribute, a decision rule is generated for filtering the input data in subsets, and the process continues recursively.

To model this design component, we make use of two different genes. The first one, **criterion**, is an integer that indexes one of the 15 splitting criteria that are implemented in HEAD-DT (see Table 4.1). The most successful criteria are based on Shannon's entropy [36], a concept well-known in information theory. Entropy is a unique function that satisfies the four axioms of uncertainty. It represents the average amount of information when coding each class into a codeword with ideal length according to its probability. Examples of splitting criteria based on entropy are *global mutual information* (GMI) [18] and *information gain* [31]. The latter is employed by Quinlan in his ID3 system [31]. However, Quinlan points out that information gain

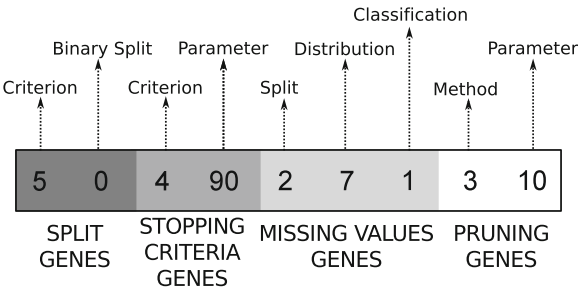

Fig. 4.2 Linear-genome for evolving decision-tree algorithms

Table 4.1 Split criteria implemented in HEAD-DT

Criterion	References
Information gain	[31]
Gini index	[4]
Global mutual information	[18]
G statistic	[26]
Mantáras criterion	[24]
Hypergeometric distribution	[25]
Chandra-Varghese criterion	[9]
DCSM	[10]
χ^2	[27]
Mean posterior improvement	[37]
Normalized gain	[21]
Orthogonal criterion	[15]
Twoing	[4]
CAIR	[11]
Gain ratio	[35]

is biased towards attributes with many values, and thus proposes a solution named *gain ratio* [35]. Gain ratio normalizes the information gain by the entropy of the attribute being tested. Several variations of the gain ratio have been proposed, such as the normalized gain [21].

Alternatives to entropy-based criteria are the class of distance-based measures. These criteria evaluate separability, divergency, or discrimination between classes. Examples are the Gini index [4], the twoing criterion [4], the orthogonality criterion [15], among several others. We also implemented as options for HEAD-DT lesser-known criteria such as CAIR [11] and mean posterior improvement [37], as well as the more recent Chandra-Varghese [9] and DCSM [10], to enhance the diversity of options for generating splits in a decision tree.

The second gene that controls the split component of a decision-tree algorithm is **binary split**. It is a binary gene that indicates whether the splits of a decision tree are going to be binary or multi-way. In a binary tree, every split has only two outcomes, which means that nominal attributes with many categories are aggregated into two subsets. In a multi-way tree, nominal attributes are divided according to their number of categories—one edge (outcome) for each category. In both cases, numeric attributes always partition the tree in two subsets, represented by tests $att \leq \Delta$ and $att > \Delta$.

4.2.2 Stopping Criteria Genes

The top-down induction of decision trees is recursive and it continues until a stopping criterion (also known as *pre-pruning*) is satisfied. The linear genome in HEAD-DT holds two genes for representing this design component: **criterion** and **parameter**.

The first gene, **criterion**, selects among the following five different strategies for stopping the tree growth:

1. Reaching class homogeneity: when every instance that reaches a given node belong to the same class, there is no reason to split this node any further. This strategy can be the only single stopping criterion, or it can be combined with the next four strategies;
2. Reaching the maximum tree depth: a parameter *tree depth* can be specified to avoid deep trees. Range: [2, 10] levels;
3. Reaching the minimum number of instances for a non-terminal node: a parameter *minimum number of instances for a non-terminal node* can be specified to avoid/alleviate the data fragmentation problem in decision trees. Range: [1, 20] instances;
4. Reaching the minimum percentage of instances for a non-terminal node: same as above, but instead of the actual number of instances, a percentage of instances is defined. The parameter is thus relative to the total number of instances in the training set. Range: [1 %, 10 %] the total number of instances;
5. Reaching an accuracy threshold within a node: a parameter *accuracy reached* can be specified for halting the growth of the tree when the accuracy within a node (majority of instances) has reached a given threshold. Range: {70 %, 75 %, 80 %, 85 %, 90 %, 95 %, 99 %} accuracy.

Gene **parameter** dynamically adjusts a value in the range [0, 100] to the corresponding strategy. For example, if the strategy selected by gene **criterion** is *reaching the maximum tree depth*, the following mapping function is executed:

$$param = (value \bmod 9) + 2 \tag{4.1}$$

This function maps from [0, 100] (variable *value*) to [2, 10] (variable *param*), which is the desired range of values for the parameter of strategy *reaching the maximum tree depth*. Similar mapping functions are executed dynamically to adjust the ranges of gene **parameter**.

4.2.3 Missing Values Genes

The next design component of decision trees that is represented in the linear genome of HEAD-DT is the *missing value treatment*. Missing values may be an issue during tree induction and also during classification. We make use of three genes to

represent missing values strategies in different moments of the induction/deduction process. During tree induction, there are two moments in which we need to deal with missing values: splitting criterion evaluation (**split gene**), and instances distribution (**distribution gene**). During tree deduction (classification), we may also have to deal with missing values in the test set (**classification gene**).

During the split criterion evaluation in node t based on attribute a_i, we implemented the following strategies:

- Ignore all instances whose value of a_i is missing [4, 17];
- Imputation of missing values with either the mode (nominal attributes) or the mean/median (numeric attributes) of all instances in t [12];
- Weight the splitting criterion value (calculated in node t with regard to a_i) by the proportion of missing values [34];
- Imputation of missing values with either the mode (nominal attributes) or the mean/median (numeric attributes) of all instances in t whose class attribute is the same of the instance whose a_i value is being imputed.

For deciding which child node training instance x_j should go to, considering a split in node t over a_i, we adopted the options:

- Ignore instance x_j [31];
- Treat instance x_j as if it has the most common value of a_i (mode or mean), regardless of the class [34];
- Treat instance x_j as if it has the most common value of a_i (mode or mean) considering the instances that belong to the same class than x_j;
- Assign instance x_j to all partitions [17];
- Assign instance x_j to the partition with largest number of instances [34];
- Weight instance x_j according to the partition probability [22, 35];
- Assign instance x_j to the most probable partition, considering the class of x_j [23].

Finally, for classifying an unseen test instance x_j, considering a split in node t over a_i, we used the strategies:

- Explore all branches of t combining the results [32];
- Take the route to the most probable partition (largest subset);
- Halt the classification process and assign instance x_j to the majority class of node t [34].

4.2.4 Pruning Genes

Pruning was originally conceived as a strategy for tolerating noisy data, though it was found to improve decision tree accuracy in many noisy data sets [4, 31, 33]. It has now become an important part of greedy top-down decision-tree induction algorithms. HEAD-DT holds two genes for this design component. The first gene, **method**, indexes one of the five well-known approaches for pruning a decision tree

Table 4.2 Pruning methods implemented in HEAD-DT

Method	References
Reduced-error pruning	[33]
Pessimistic error pruning	[33]
Minimum error pruning	[8, 28]
Cost-complexity pruning	[4]
Error-based pruning	[35]

presented in Table 4.2, and also the option of not pruning at all. The second gene, **parameter**, is in the range [0, 100] and its value is again dynamically mapped by a function according to the selected pruning method.

(1) Reduced-error pruning (REP) is a conceptually simple strategy proposed by Quinlan [33]. It uses a pruning set (a part of the training set) to evaluate the goodness of a given subtree from T. The idea is to evaluate each non-terminal node t with regard to the classification error in the pruning set. If such an error decreases when we replace the subtree $T^{(t)}$ rooted on t by a leaf node, then $T^{(t)}$ must be pruned. Quinlan imposes a constraint: a node t cannot be pruned if it contains a subtree that yields a lower classification error in the pruning set. The practical consequence of this constraint is that REP should be performed in a bottom-up fashion. The REP pruned tree T' presents an interesting optimality property: it is the smallest most accurate tree resulting from pruning original tree T [33]. Besides this optimality property, another advantage of REP is its linear complexity, since each node is visited only once in T. A clear disadvantage is the need of using a pruning set, which means one has to divide the original training set, resulting in less instances to grow the tree. This disadvantage is particularly serious for small data sets. For REP, the parameter gene is regarding the percentage of training data to be used in the pruning set (varying within the interval [10 %, 50 %]).

(2) Also proposed by Quinlan [33], the pessimistic error pruning (PEP) uses the training set for both growing and pruning the tree. The apparent error rate, i.e., the error rate calculated over the training set, is optimistically biased and cannot be used to decide whether pruning should be performed or not. Quinlan thus proposes adjusting the apparent error according to the continuity correction for the binomial distribution in order to provide a more realistic error rate. PEP is computed in a top-down fashion, and if a given node t is pruned, its descendants are not examined, which makes this pruning strategy quite efficient in terms of computational effort. Esposito et al. [14] point out that the introduction of the continuity correction in the estimation of the error rate has no theoretical justification, since it was never applied to correct over-optimistic estimates of error rates in statistics. For PEP, the parameter gene is the number of standard errors (SEs) to adjust the apparent error, in the set {0.5, 1, 1.5, 2}.

(3) Originally proposed by Niblett and Bratko [28] and further extended in [8], minimum error pruning (MEP) is a bottom-up approach that seeks to minimize the *expected error rate* for unseen cases. It uses an ad-hoc parameter m for controlling

the level of pruning. Usually, the higher the value of m, the more severe the pruning. Cestnik and Bratko [8] suggest that a domain expert should set m according to the level of noise in the data. Alternatively, a set of trees pruned with different values of m could be offered to the domain expert, so he/she can choose the best one according to his/her experience. For MEP, the parameter gene comprises variable m, which may range within [0, 100].

(4) Cost-complexity pruning (CCP) is the post-pruning strategy adopted by the CART system [4]. It consists of two steps: (i) generate a sequence of increasingly smaller trees, beginning with T and ending with the root node of T, by successively pruning the subtree yielding the lowest *cost complexity*, in a bottom-up fashion; (ii) choose the best tree among the sequence based on its relative size and accuracy (either on a pruning set, or provided by a cross-validation procedure in the training set). The idea within step 1 is that pruned tree T_{i+1} is obtained by pruning the subtrees that show the lowest increase in the apparent error (error in the training set) per pruned leaf. Regarding step 2, CCP chooses the smallest tree whose error (either on the pruning set or on cross-validation) is not more than one standard error (SE) greater than the lowest error observed in the sequence of trees. For CCP, there are two parameters that need to be set: the number of SEs (in the same range than PEP) and the pruning set size (in the same range than REP).

(5) Error-based pruning (EBP) was proposed by Quinlan and it is implemented as the default pruning strategy of C4.5 [35]. It is an improvement over PEP, based on a far more pessimistic estimate of the expected error. Unlike PEP, EBP performs a bottom-up search, and it carries out not only the replacement of non-terminal nodes by leaves but also *grafting* of subtree $T^{(t)}$ onto the place of parent t. For deciding whether to replace a non-terminal node by a leaf (subtree replacement), to graft a subtree onto the place of its parent (subtree raising) or not to prune at all, a pessimistic estimate of the expected error is calculated by using an upper confidence bound. An advantage of EBP is the new *grafting* operation that allows pruning useless branches without ignoring interesting lower branches. A drawback of the method is the presence of an ad-hoc parameter, *CF*. Smaller values of *CF* result in more pruning. For EBP, the parameter gene comprises variable *CF*, which may vary within [1 %, 50 %].

4.2.5 Example of Algorithm Evolved by HEAD-DT

The linear genome of an individual in HEAD-DT is formed by the building blocks described in the earlier sections: (*split criterion, split type, stopping criterion, stopping parameter, pruning strategy, pruning parameter, mv split, mv distribution, mv classification.*) One possible individual encoded by that linear string is [4, 1, 2, 77, 3, 91, 2, 5, 1], which accounts for Algorithm 1.

Algorithm 1 Example of a decision-tree algorithm automatically designed by HEAD-DT.

1: Recursively split nodes with the G statistics criterion;
2: Create one edge for each category in a nominal split;
3: Perform step 1 until class-homogeneity or the maximum tree depth of 7 levels ((77 mod 9) + 2) is reached;
4: Perform MEP pruning with $m = 91$;
When dealing with missing values:
5: Distribute missing-valued instances to the partition with the largest number of instances;
6: Distribute missing values by assigning the instance to all partitions;
7: For classifying an instance with missing values, explore all branches and combine the results.

4.3 Evolution

The first step of HEAD-DT is the generation of the initial population, in which a population of 100 individuals (default value) is randomly generated (generation of random numbers within the genes' acceptable range of values). Next, the population's fitness is evaluated based on the data sets that belong to the meta-training set. The individuals then participate of a pairwise tournament selection procedure ($t = 2$ is the default parameter) for defining those that will undergo the genetic operators. Individuals may participate in either uniform crossover, random uniform gene mutation, or reproduction, the three mutually-exclusive genetic operators employed in HEAD-DT (see Fig. 4.3).

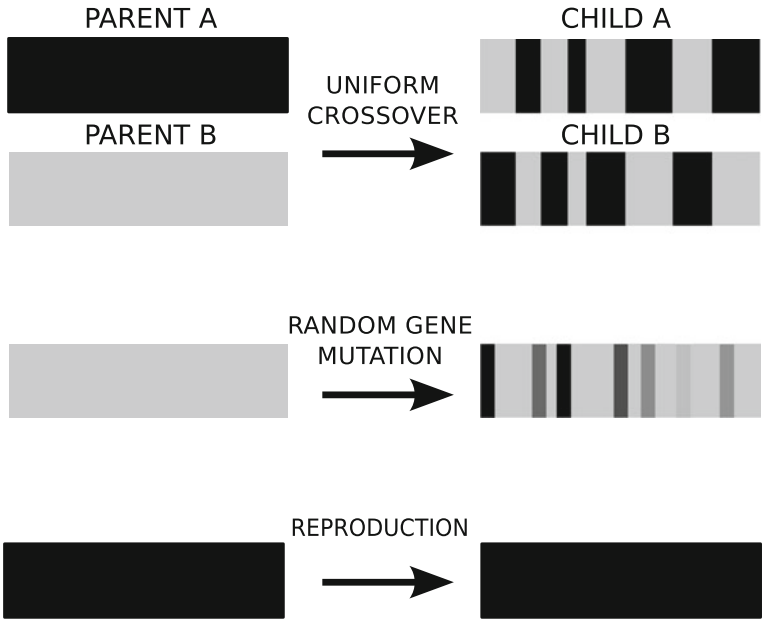

Fig. 4.3 HEAD-DT's genetic operators

The uniform crossover is guided by a swap probability p_s (default value $= 0.5$) that ultimately indicates whether a child's gene should come from parent A or from parent B. Algorithm 2 depicts the pseudocode of the uniform crossover operator implemented in HEAD-DT.

Algorithm 2 Uniform crossover employed by HEAD-DT.

```
Let A and B be two parents chosen by tournament selection;
Let C and D be the two resulting offspring;
for each gene g in genome do
    Choose a uniform random real number u from [0,1];
    if u ≤ p_s then
        //swap genes
        C[g] = B[g];
        D[g] = A[g];
    else
        //do not swap
        C[g] = A[g];
        D[g] = B[g];
    end if
end for
```

The mutation operator implemented in HEAD-DT is the random uniform gene mutation. It is guided by a replacement probability p_{rep} (default value $= 0.5$), which dictates whether or not a gene's value should be replaced by a randomly generated value within the accepted range of the respective gene. Algorithm 3 depicts the pseudocode of the random uniform gene mutation operator implemented in HEAD-DT.

Algorithm 3 Random uniform gene mutation employed by HEAD-DT.

```
Let A be a single individual chosen by tournament selection;
Let B be the individual resulting from mutating A;
for each gene g in genome do
    Choose a uniform random real number u from [0,1];
    if u ≤ p_rep then
        //mutate gene
        Randomly generate a value v within the range accepted by g;
        B[g] = v;
    else
        //do not mutate gene
        B[g] = A[g]
    end if
end for
```

Finally, reproduction is the operation that copies (clones) a given individual to be part of the EA's next generation in a straightforward fashion. A single parameter p controls the mutually-exclusive genetic operators: crossover probability is given by p, whereas mutation probability is given by $(1 - p) - 0.05$, and reproduction probability is fixed in 0.05. For instance, if $p = 0.9$, then HEAD-DT is executed with a crossover probability of 90 %, mutation probability of 5 % and reproduction probability of 5 %. HEAD-DT employs an elitist strategy, in which the best e % individuals are kept from one generation to the next ($e = 5$ % of the population is

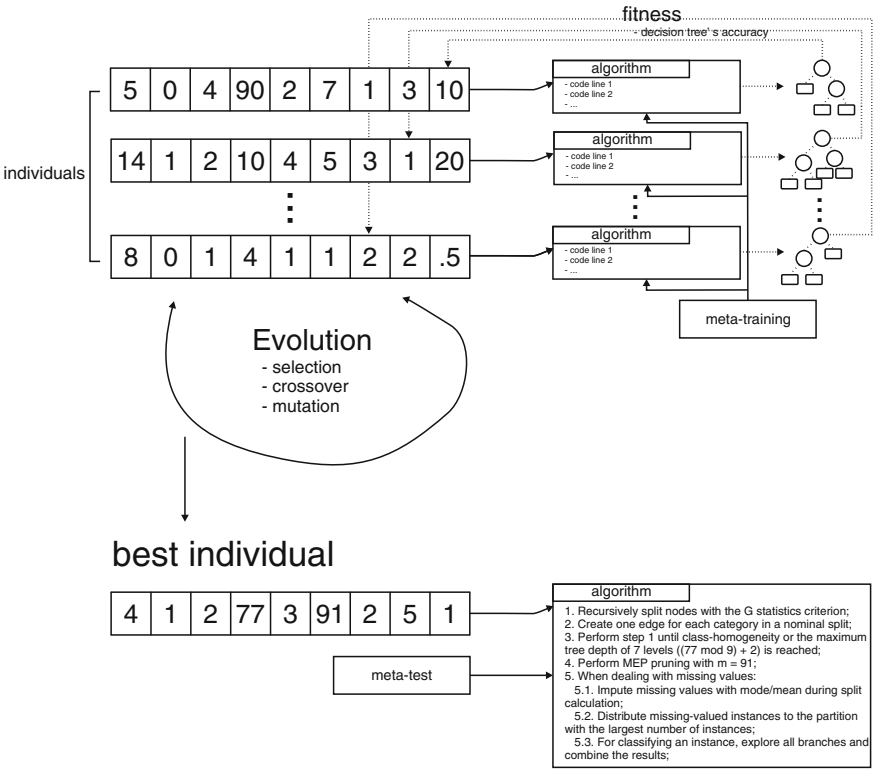

Fig. 4.4 Evolution of individuals encoded as integer vectors

the default parameter). Evolution ends after a predefined number of generations is achieved (100 generations is the default value), and the best individual returned by HEAD-DT is then executed over the meta-test set, so its performance in unseen data can be properly assessed.

Figure 4.4 presents an example of how linear genomes are decoded into algorithms, and how they participate of the evolutionary cycle. For decoding the individuals, the building blocks (indexed components and their respective parameters) are identified, and this information is passed to a skeleton decision-tree induction class, filling the gaps with the selected building blocks.

4.4 Fitness Evaluation

During the fitness evaluation, HEAD-DT employs a *meta-training set* for assessing the quality of each individual throughout evolution. A *meta-test set* is used for assessing the quality of the evolved decision-tree induction algorithm (the best individual

in Fig. 4.1). There are two distinct frameworks for dealing with the meta-training and test sets:

1. Evolving a decision-tree induction algorithm tailored to one specific data set.
2. Evolving a decision-tree induction algorithm from multiple data sets.

In the first case, named *the specific framework*, we have a specific data set for which we want to design a decision-tree algorithm. The meta-training set comprises the available training data from the data set at hand. The meta-test set comprises test data (belonging to the same data set) we have available for evaluating the performance of the algorithm (see Fig. 4.5a). For example, suppose HEAD-DT is employed to evolve the near-optimal decision-tree induction algorithm for the *iris* data set. In such a scenario, both meta-training and meta-test sets comprise distinct data folds from the *iris* data set.

In the second case, named *the general framework*, there are multiple data sets composing the meta-training set, and possibly multiple (albeit different) data sets comprising the meta-test set (see Fig. 4.5b). For example, suppose HEAD-DT is employed to evolve the near-optimal algorithm for the problem of credit risk assessment. In this scenario, the meta-training set may comprise public UCI data sets [16] such as *german credit* and *credit approval*, whereas the meta-test set may comprise particular credit risk assessment data sets the user desires to classify.

The general framework can be employed with two different objectives, broadly speaking:

1. Designing a decision-tree algorithm whose predictive performance is consistently good in a wide variety of data sets. For such, the evolved algorithm is applied to data sets with very different structural characteristics and/or from very distinct application domains. In this scenario, the user chooses distinct data sets to be part of the meta-training set, in the hope that evolution will be capable of generating an algorithm that performs well in a wide range of data sets. Pappa [29] calls this strategy "*evolving robust algorithms*";
2. Designing a decision-tree algorithm that is tailored to a particular application domain or to a specific statistical profile. In this scenario, the meta-training set comprises data sets that share *similarities*, and so the evolved decision-tree algorithm is specialized in solving a specific type of problem. Unlike the previous

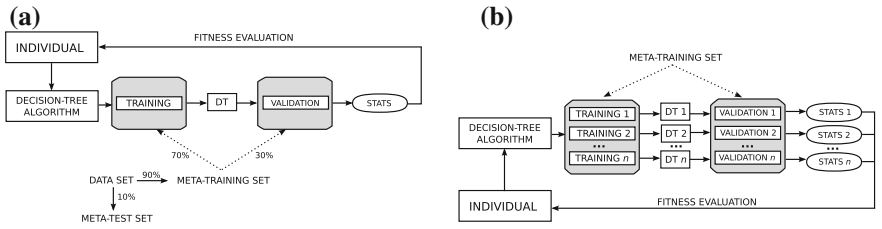

Fig. 4.5 Fitness evaluation schemes. **a** Fitness evaluation from one data set in the meta-training set. **b** Fitness evaluation from multiple data sets in the meta-training set

strategy, in this case we have to define a *similarity* criterion for creating specialized algorithms. We highlight the following *similarity* criteria: (i) choosing data sets that share the same *application domain* (e.g., gene expression data); (ii) choosing data sets with *provenance resemblance* (e.g., data sets generated from data collected by a specific sensor); and (iii) choosing data sets with *structural resemblance* (e.g., data sets with statistically-similar features and/or with similar geometrical complexity [19, 20]).

In Fig. 4.5b, we can observe how the fitness evaluation of a decision-tree induction algorithm evolved from multiple data sets occurs. First, a given individual is mapped into its corresponding decision-tree algorithm. Afterwards, each data set that belongs to the meta-training set is divided into training and validation—typical values are 70 % for training and 30 % for validation [39]. The term "validation set" is used in here instead of "test set" to avoid confusion with the meta-test set, and also due to the fact that we are using the "knowledge" within these sets to reach for a better solution (the same cannot be done with test sets, which are exclusively used for assessing the performance of an algorithm).

After dividing each data set from the meta-training set into "training" and "validation", a decision tree is induced for each training set available. For evaluating the performance of these decision trees, we use the corresponding validation sets. Statistics regarding the performance of each decision tree are recorded (e.g., accuracy, F-Measure, precision, recall, total number of nodes/leaves, etc.), and can be used individually or combined as the fitness function of HEAD-DT. The simple average is probably the most intuitive way of combining the values per data set, but other possible solutions are the median of the values, or their harmonic mean. Depending on the data sets used in the meta-training set, the user may decide to give greater weight of importance to a more difficult-to-solve data set than to an easier one, and hence a weighted scheme may be a good solution when combining the data set values. Some of these possibilities are discussed in Chap. 6.

A typical fitness function would be the average F-Measure of the decision trees generated by a given individual for each data set from the meta-training set. F-Measure (also known as F-score or F1 score) is the harmonic mean of precision and recall:

$$precision = \frac{tp}{tp + fp} \tag{4.2}$$

$$recall = \frac{tp}{tp + fn} \tag{4.3}$$

$$fmeasure = 2 \times \frac{precision \times recall}{precision + recall} \tag{4.4}$$

$$Fitness = \frac{1}{n} \sum_{i=1}^{n} fmeasure_i \tag{4.5}$$

where *tp* (*tn*) is the number of true positives (negatives), *fp* (*fn*) is the number of false positives (negatives), $fmeasure_i$ is the F-Measure obtained in data set i and n is the total number of data sets in the meta-training set.

This formulation assumes that the classification problem at hand is binary, i.e., composed by two classes: positive and negative. Nevertheless, it can be trivially extended to multi-class problems. For instance, we can compute the measure for each class—assuming each class to be the "positive" class in turn—and (weight-)average the per-class measures. Having in mind that we would like HEAD-DT to perform well in both balanced and imbalanced data sets, we believe that the average F-Measure is a more suitable fitness function than the average accuracy.

4.5 Search Space

To compute the search space reached by HEAD-DT, consider the linear genome presented in Sect. 4.2: (*split criterion, split type, stopping criterion, stopping parameter, pruning strategy, pruning parameter, mv split, mv distribution, mv classification*). There are 15 types of split criteria, 2 possible split types, 4 types of missing-value strategies during split computation, 7 types of missing-value strategies during training data distribution, and 3 types of missing-value strategies during classification. Hence, there are $15 \times 2 \times 4 \times 7 \times 3 = 2{,}520$ possible different algorithms.

Now, let us analyse the combination of stopping criteria and their parameters. There is the possibility of splitting until class homogeneity is achieved, and no parameter is needed (thus, 1 possible algorithm). There are 9 possible parameters when the tree is grown until a maximum depth, and 20 when reaching a minimum number of instances. Furthermore, there are 10 possible parameter values when reaching a minimum percentage of instances and 7 when reaching an accuracy threshold. Hence, there are $1 + 9 + 20 + 10 + 7 = 47$ possible algorithms just by varying the stopping criteria component.

Next, let us analyse the combination of pruning methods and their parameters. REP pruning parameter may take up to 5 different values, whereas PEP pruning may take up to 4. MEP can take up to 101 values, and EBP up to 50. Finally, CCP takes up to 4 values for its first parameter and up to 5 values for its second. Therefore, there are $5 + 4 + 101 + (4 \times 5) + 50 = 180$ possible algorithms by just varying the pruning component.

If we combine all the previously mentioned values, HEAD-DT currently searches in the space of $2{,}520 \times 47 \times 180 = 21{,}319{,}200$ algorithms. Now, just for the sake of argument, suppose a single decision-tree induction algorithm takes about 10 s to produce a decision tree for a given (small) data set for which we want the best possible algorithm. If we were to try all possible algorithms in a brute-force approach, we would take 59,220 h to find out the best possible configuration for that

data set. That means ≈2,467 days or 6.75 years just to find out the best decision-tree algorithm for a single (small) data set. HEAD-DT would take, in the worst case, 100,000 s—10,000 individuals (100 individuals per generation, 100 generations) times 10 s. Thus HEAD-DT would take about 1,666 min (27.7 h) to compute the (near-)optimal algorithm for that same data set, i.e., it is ≈2,138 times faster than the brute-force approach. In practice, this number is much smaller considering that individuals are not re-evaluated if not changed, and HEAD-DT implements reproduction and elitism.

Of course there are no theoretic guarantees that the (near-)optimal algorithm found by HEAD-DT within these 27.7 h is going to be the same global optimal algorithm provided by the brute-force approach after practically 7 years of computation, but its use is justified by the time saved during the process.

4.6 Related Work

The literature in EAs for decision-tree induction is very rich (see, for instance, [3]). However, the research community is still concerned with the evolution of decision trees for particular data sets instead of evolving full decision-tree induction algorithms.

To the best of our knowledge, no work to date attempts to automatically design full decision-tree induction algorithms. The most related approach to the one presented in this book is HHDT (Hyper-Heuristic Decision Tree) [38]. It proposes an EA for evolving heuristic rules in order to determine the best splitting criterion to be used in non-terminal nodes. It is based on the degree of entropy of the data set attributes. For instance, it evolves rules such as *IF* ($x\ \% \geq high$) and ($y\ \% < low$) *THEN use heuristic A*, where x and y are percentages ranging within [0, 100], and high and *low* are threshold entropy values. This rule has the following interpretation: if x % of the attributes have entropy values greater or equal than threshold *high*, and if y % of the attributes have entropy values below threshold *low*, then make use of heuristic *A* for choosing the attribute that splits the current node. Whilst HHDT is a first step to automate the design of decision-tree induction algorithms, it evolves a single component of the algorithm (the choice of splitting criterion), and thus should be further extended for being able to generate full decision-tree induction algorithms, which is the case of HEAD-DT.

Another slightly related approach is the one presented by Delibasic et al. [13]. The authors propose a framework for combining decision-tree components, and test 80 different combination of design components on 15 benchmark data sets. This approach is not a hyper-heuristic, since it does not present an heuristic to choose among different heuristics. It simply selects a fixed number of component combinations and test them all against traditional decision-tree algorithms (C4.5, CART, ID3

and CHAID). We believe that employing EAs to evolve decision-tree algorithms is a more robust strategy, since we can search for solutions in a much larger search space (21 million possible algorithms in HEAD-DT, against 80 different algorithms in the work of Delibasic et al. [13]).

Finally, the work of Pappa and Freitas [30] proposes a grammar-based genetic programming approach (GGP) for evolving full rule induction algorithms. The results showed that GGP could generate rule induction algorithms different from those already proposed in the literature, and with competitive predictive performance.

4.7 Chapter Remarks

In this chapter, we presented HEAD-DT, a hyper-heuristic evolutionary algorithm that automatically designs top-down decision-tree induction algorithms. The latter have been manually improved for the last 40 years, resulting in a large number of approaches for each of their design components. Since the human manual approach for testing all possible modifications in the design components of decision-tree algorithms would be unfeasible, we believe the evolutionary search of HEAD-DT constitutes a robust and efficient solution for the problem.

HEAD-DT evolves individuals encoded as integer vectors (linear genome). Each gene in the vector is an index to a design component or the value of its corresponding parameter. Individuals are decoded by associating each integer to a design component, and by mapping values ranging within [0, 100] to values in the correct range according to the specified component. The initial population of 100 individuals evolve for 100 generations, in which individuals are chosen by a pairwise tournament selection strategy to participate of mutually-exclusive genetic operators such as uniform crossover, random uniform gene mutation, and reproduction.

HEAD-DT may operate under two distinct frameworks: (i) evolving a decision-tree induction algorithm tailored to one specific data set; and (ii) evolving a decision-tree induction algorithm from multiple data sets. In the first framework, the goal is to generate a decision-tree algorithm that excels at a single data set (both meta-training and meta-test sets comprise data from the same data set). In the second framework, there are several distinct objectives that can be achieved, like generating a decision-tree algorithm tailored to a particular application domain (say gene expression data sets or financial data sets), or generating a decision-tree algorithm that is robust across several different data sets (a good "all-around" algorithm).

Regardless of the framework being employed, HEAD-DT is capable of searching in a space of more than 21 million algorithms. In the next chapter, we present several experiments for evaluating HEAD-DT's performance under the two proposed frameworks. Moreover, we comment on the cost-effectiveness of automated algorithm design in contrast to the manual design, and we show that the genetic search performed by HEAD-DT is significantly better than a random search in the space of 21 million decision-tree induction algorithms.

References

1. R.C. Barros, D.D. Ruiz, M.P. Basgalupp, Evolutionary model trees for handling continuous classes in machine learning. Inf. Sci. **181**, 954–971 (2011)
2. R.C. Barros et al., Towards the automatic design of decision tree induction algorithm, in *13th Annual Conference Companion on Genetic and Evolutionary Computation* (GECCO 2011). pp. 567–574 (2011)
3. R.C. Barros et al., A survey of evolutionary algorithms for decision-tree induction. IEEE Trans. Syst., Man, Cybern., Part C: Appl. Rev. **42**(3), 291–312 (2012)
4. L. Breiman et al., *Classification and Regression Trees* (Wadsworth, Belmont, 1984)
5. E. Burke, S. Petrovic, Recent research directions in automated timetabling. Eur. J. Oper. Res. **140**(2), 266–280 (2002)
6. E.K. Burke, G. Kendall, E. Soubeiga, A tabu-search hyperheuristic for timetabling and rostering. J. Heuristics **9**(6), 451–470 (2003)
7. E.K. Burke et al., A Classification of Hyper-heuristics Approaches, in *Handbook of Metaheuristics*, 2nd edn., International Series in Operations Research & Management Science, ed. by M. Gendreau, J.-Y. Potvin (Springer, Berlin, 2010), pp. 449–468
8. B. Cestnik, I. Bratko, *On Estimating Probabilities in Tree Pruning*, Machine learning-EWSL-91. Vol. 482. Lecture Notes in Computer Science (Springer, Berlin, 1991)
9. B. Chandra, P.P. Varghese, Moving towards efficient decision tree construction. Inf. Sci. **179**(8), 1059–1069 (2009)
10. B. Chandra, R. Kothari, P. Paul, A new node splitting measure for decision tree construction. Pattern Recognit. **43**(8), 2725–2731 (2010)
11. J. Ching, A. Wong, K. Chan, Class-dependent discretization for inductive learning from continuous and mixed-mode data. IEEE Trans. Pattern Anal. Mach. Intell. **17**(7), 641–651 (1995)
12. P. Clark, T. Niblett, The CN2 induction algorithm. Mach. Learn. **3**(4), 261–283 (1989)
13. B. Delibasic et al., Component-based decision trees for classification. Intell. Data Anal. **15**(5), 1–38 (2011)
14. F. Esposito, D. Malerba, G. Semeraro, A comparative analysis of methods for pruning decision trees. IEEE Trans. Pattern Anal. Mach. Intell. **19**(5), 476–491 (1997)
15. U. Fayyad, K. Irani, The attribute selection problem in decision tree generation, in *National Conference on Artificial Intelligence*. pp. 104–110 (1992)
16. A. Frank, A. Asuncion, UCI Machine Learning Repository (2010)
17. J.H. Friedman, A recursive partitioning decision rule for nonparametric classification. IEEE Trans. Comput. **100**(4), 404–408 (1977)
18. M. Gleser, M. Collen, Towards automated medical decisions. Comput. Biomed. Res. **5**(2), 180–189 (1972)
19. T. Ho, M. Basu, Complexity measures of supervised classification problems. IEEE Trans. Pattern Anal. Mach. Intell. **24**(3), 289–300 (2002)
20. T. Ho, M. Basu, M. Law, *Measures of Geometrical Complexity in Classification Problems*, Data Complexity in Pattern Recognition (Springer, London, 2006)
21. B. Jun et al., A new criterion in selection and discretization of attributes for the generation of decision trees. IEEE Trans. Pattern Anal. Mach. Intell. **19**(2), 1371–1375 (1997)
22. I. Kononenko, I. Bratko, E. Roskar, Experiments in automatic learning of medical diagnostic rules. Tech. rep. Ljubljana, Yugoslavia: Jozef Stefan Institute (1984)
23. W. Loh, Y. Shih, Split selection methods for classification trees. Stat. Sinica **7**, 815–840 (1997)
24. R.L. De Mántaras, *A Distance-Based Attribute Selection Measure for Decision Tree Induction*, Machine learning 6.1 (Kluwer, The Netherland, 1991). ISSN: 0885–6125
25. J. Martin, An exact probability metric for decision tree splitting and stopping. Mach. Learn. **28**(2), 257–291 (1997)
26. J. Mingers, Expert systems—rule induction with statistical data. J. Oper. Res. Soc. **38**, 39–47 (1987)

27. J. Mingers, An empirical comparison of selection measures for decision-tree induction. Mach. Learn. **3**(4), 319–342 (1989)
28. T. Niblett, I. Bratko, Learning decision rules in noisy domains, in *6th Annual Technical Conference on Research and Development in Expert Systems III*. pp. 25–34 (1986)
29. G.L. Pappa, Automatically Evolving Rule Induction Algorithms with Grammar-Based Genetic Programming. PhD thesis. University of Kent at Canterbury (2007)
30. G.L. Pappa, A.A. Freitas, *Automating the Design of Data Mining Algorithms: An Evolutionary Computation Approach* (Springer Publishing Company, Incorporated, 2009)
31. J.R. Quinlan, Induction of decision trees. Mach. Learn. **1**(1), 81–106 (1986)
32. J.R. Quinlan, Decision trees as probabilistic classifiers, in *4th International Workshop on Machine Learning* (1987)
33. J.R. Quinlan, Simplifying decision trees. Int. J. Man-Mach. Stud. **27**, 221–234 (1987)
34. J.R. Quinlan, Unknown attribute values in induction, in *6th International Workshop on Machine Learning*. pp. 164–168 (1989)
35. J. R. Quinlan, C4.5: programs for machine learning. San Francisco: Morgan Kaufmann (1993). ISBN: 1-55860-238-0
36. C.E. Shannon, A mathematical theory of communication. BELL Syst. Tech. J. **27**(1), 379–423, 625–56 (1948)
37. P.C. Taylor, B.W. Silverman, Block diagrams and splitting criteria for classification trees. Stat. Comput. **3**, 147–161 (1993)
38. A. Vella, D. Corne, C. Murphy, Hyper-heuristic decision tree induction, in *World Congress on Nature and Biologically Inspired Computing*, pp. 409–414 (2010)
39. I.H. Witten, E. Frank, Data mining: practical machine learning tools and techniques with java implementations. Morgan Kaufmann. ISBN: 1558605525 (1999)