Building a GA from Design Principles for Learning Bayesian Networks

Steven van Dijk, Dirk Thierens, and Linda C. van der Gaag

Universiteit Utrecht, Institute of Information and Computing Sciences, Decision Support Systems, PO Box 80.089, 3508 TB Utrecht, The Netherlands
{steven, dirk, linda}@cs.uu.nl

Abstract. Recent developments in GA theory have given rise to a number of design principles that serve to guide the construction of selecto-recombinative GAs from which good performance can be expected. In this paper, we demonstrate their application to the design of a GA for a well-known hard problem in machine learning: the construction of a Bayesian network from data. We show that the resulting GA is able to efficiently and reliably find good solutions. Comparisons against state-of-the-art learning algorithms, moreover, are favorable.

1 Introduction

Recent developments in GA theory [5, 18, 12, 6, 20] have yielded in-depth insight in the search behavior of selecto-recombinative GAs. This insight stresses the importance of concepts such as linkage, mixing, and disruption. The use of a selecto-recombinative GA for solving a search problem is appropriate if the linkage of the problem can be assessed with reasonable confidence. The linkage then determines the location of the building blocks for good solutions for the problem. It further allows for the design of a crossover operator that mixes well and indicates where disruption can occur, thereby making counter measures easier to take. We use these recent insights to define various design principles to guide the construction of GAs from which we can expect good performance.

To demonstrate the effectiveness of our GA design principles, we apply them to the problem of learning Bayesian networks (BNs) from data. BNs [13] are probabilistic graphical models that capture a joint probability distribution by explicitly modeling the independences between the statistical variables involved in a directed acyclic graph (DAG). The strengths of the relationships between the variables are quantified by probability tables. Figure 1 depicts a hypothetical example of a BN. While Bayesian networks have proven their value as a robust framework for reasoning with uncertainty in a range of applications, their construction can be a daunting task. Especially when having to rely upon extensive collaboration of domain experts to build the graphical structure and to elicit the required probabilities, handcrafting a network is hard and very time consuming. If databases are available that store information about the statistical variables of importance, however, these data can be exploited to construct a Bayesian

E. Cantú-Paz et al. (Eds.): GECCO 2003, LNCS 2723, pp. 886–897, 2003.

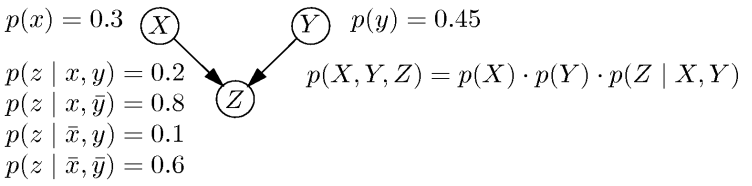

Fig. 1. Bayesian network with three variables ($X = \{x, \bar{x}\}$, $Y = \{y, \bar{y}\}$ and $Z = \{z, \bar{z}\}$).

network automatically. The learned network can then be further improved with the help of domain experts.

The BN learning problem essentially is an optimization problem, where a Bayesian network has to be found that best represents the probability distribution that has generated the data in a given database. The use of maximum-likelihood estimates allows us to take the frequencies in the data for a network's probability tables. The learning problem thereby reduces to the problem of searching the optimum in the space of all DAGs. A trade-off has to be made between the structural complexity of a network and the accuracy with which it describes the data, since complex networks tend to suffer from over-fitting and make the running time of inference algorithms prohibitively large. An oft-used measure that serves to balance accuracy and complexity is the MDL measure, based on the principle of minimal description length from information theory [15,9]. In this paper we solve the BN learning problem for databases with complete cases by searching for a DAG that, after completion to a network with maximum-likelihood estimates, minimizes the MDL score.

The learning problem is a suitable problem to exemplify our GA design principles with since the graphical nature of a BN allows for an easy assessment of the structure of the problem. A learned network should connect two nodes if the corresponding variables are dependent. Even though the reverse need not be true, strong dependences found in the data indicate which pairs of nodes are important for the search. This observation translates into an assessment of the linkage between the genes in an appropriately-chosen representation, and therefore of the location of the building blocks.

Our contributions are the following. We show how to solve an important search problem from GA design principles. We further describe a carefully designed GA for the learning problem that compares favorably against current state-of-the-art learning algorithms. We present results on two real networks in addition to the commonly used Alarm network. The remainder of the paper is organized as follows. In Section 2, we discuss previous work on the BN learning problem. We continue in Section 3 with a discussion of recent developments in GA theory and with a statement of our design principles. In Section 4, we describe our GA for the learning problem. We present results and comparisons in Section 5. We conclude and discuss future work in Section 6.

2 Related Work

Most state-of-the-art algorithms for learning BNs from complete data can be classified as taking one of two approaches: the use of an (in)dependence test such as Pearson χ^2 or mutual information [16,4], and the use of a quality measure such as MDL [2,7,9]. With both approaches, encouraging results have been reported. Both approaches, however, also have their disadvantages.

In the first approach, a statistical test is employed for examining whether or not two variables are (in)dependent given some conditioning set of variables. The *order* of the test is the size of the conditioning set used. By starting with zero-order tests and selectively growing the conditioning set, in theory, all relevant (in)dependences can be extracted from the data and the network can be exactly recovered. In practice, however, the test quickly becomes unreliable for higher orders. For these higher orders, the number of data available for the test decreases exponentially with the order. For example, for binary variables and a database of 1000 cases (a realistic size), a sixth-order independence test would be based, on average, on approximately $1000/2^6 \approx 15$ cases from the database. As a result, the test can be inaccurate, which can affect the quality of the learned network.

In the second approach, an information measure is used for assessing the quality of candidate DAGs. This approach suffers from the size of the search space of DAGs. To efficiently traverse this huge space, often a greedy search algorithm is used to focus attention on the most promising regions. Other algorithms explicitly constrain the search space by assuming a topological ordering on the nodes of candidate DAGs. With both types of algorithm, the optimal DAG may be pruned from the space that is effectively searched.

Larrañaga et al. [11] have proposed a genetic algorithm based upon the latter approach. In their GA, a DAG is represented by a connectivity matrix that is stored as a string (the concatenation of its rows). Recombination is implemented as one-point crossover on these strings; mutation is implemented as random bit-flipping. After children have been generated, the resulting graphs are rendered acyclic by randomly deleting arcs that are part of a cycle. In addition, for nodes with a number of (immediate) predecessors that exceeds a predefined constant, the best subset is chosen that is small enough. In related work, Larrañaga et al. [10] experimented with a GA that searches for an ordering that is passed on to K2, a greedy search algorithm. They concluded that the results were comparable to those of their previous GA. The use of different crossover operators in a GA for the learning problem has recently been explored by Cotta and Muruzàbal [3], who concluded that adaptive operators gave the best results.

Recently, Wong et al. [22] proposed the "hybrid evolutionary programming" (HEP) algorithm that combines the use of independence tests with a quality-based search. The search space of DAGs is constrained in the sense that each possible DAG only connects two nodes if they show a strong dependence in the available data. The algorithm evolves a population of DAGs to find a solution that minimizes the MDL score. A new population $Pop(t+1)$ is constructed as follows. First, half the members from $Pop(t)$ are copied and put in an intermediate population. The members from the other half are "merged" with

the members from the previous population $Pop(t-1)$ that were *not* selected. Merging is similar to crossover, except that it selectively uses those parts of the parents that give the largest improvement of the MDL score of the child. The child is then put in the intermediate population. To each individual in this intermediate population, four different mutation operators are applied. The resulting population is then added to $Pop(t)$. Selection consists of ranking the individuals and putting the top half in $Pop(t+1)$.

As mentioned before, the HEP algorithm constrains the search space of DAGs by only allowing edges that connect nodes with a strong dependence between them. More specifically, the algorithm allows an edge between two nodes if there exists a zero-order dependence between them, and no first-order order *in*dependences. The (in)dependence test employed makes use of a threshold that indicates the required level of significance. Wong et al. argue that it is better to avoid manually setting the threshold. Instead, they allow the search algorithm to evolve the threshold and constrain the solutions to match their individual threshold. Initially, a threshold is chosen at random for each individual in the population. A new individual receives the mutated threshold of its parent.

Wong et al. compared an earlier version of their algorithm against the GA of Larrañaga et al. [11], and found that it was faster and produced networks of better MDL scores. We will compare our GA against HEP in Section 5.

3 Design Considerations

During the previous decade, GA theory has progressed to the point where GAs giving predictable results can be designed. A descriptive model of the selecto-recombinative GA has emerged from the literature [5, 18, 12, 6, 20] in which the representation of solutions is divided into several partitions[1]. A schema of highest fitness in a partition is called a building block. The proportion of building blocks of a certain partition in the population should converge to 1.0. The propagation of building blocks to this end is modeled by competitions between strings and statistical decision making. By mixing on the boundaries of the partitions, the building blocks are assembled into a close-to-optimal solution. In other words, a good solution matches the building block of each partition. The model is applicable especially to problems that involve an additively decomposable fitness function. In the work by Harik et al. [6], more specifically, partitions are separable, that is, any gene is part of a single partition. The partitions themselves then correspond to the subfunctions into which the fitness function can be decomposed. The model was tested on deceptive subfunctions, and proved to be quite accurate. Later it was shown that the model could also predict results for more realistic problems that involve overlapping partitions, such as the map-labeling

[1] A *chromosome* is a string over an alphabet A. A *schema* is a string over $A \cup \{\#\}$, which is matched by a chromosome that has the same symbols in the same positions, except for the #'s. A *partition* is a string over $\{\#, f\}$ that is matched by schemata that have an element from A at positions where the partition has an f.

problem [20], provided that the deviations from the underlying assumptions are not too severe.

The insights yielded by the model of the selecto-recombinative GA can be formulated into the following design principles [21]:

- Use an additively decomposable fitness function.
- Ensure a good building-block supply, either in the initial population or during the run.
- Ensure that the proportions of building blocks increase.
- Ensure good mixing of building blocks.
- Minimize disruption of building blocks.

The first principle allows for a representation of solutions with a direct mapping of the fitness subfunctions to genes. Ensuring a good building-block supply is vital since it is impossible to construct a good solution without building blocks. Building blocks will be scattered throughout the initial population, or will be injected during the run by the genetic operators. Proper selection of building blocks is necessary to ensure their proportions within the population grow. During the run of the algorithm, all building blocks have to be assembled on the same individual, which makes good mixing of building blocks vital. Since mixing tends to be disruptive, however, care needs to be taken that not too many building blocks are lost. Central to most of these principles is the assessment of the linkage, which is the first design issue to be addressed. We next exemplify the various principles by designing a GA for the BN learning problem.

4 The GA

We recall that the learning problem involves searching for a DAG. For the representation of DAGs, we use a list of genes (corresponding with connections between nodes) of fixed length. Each gene can be set to three different alleles corresponding with the two possible arc directions and the absence of an arc. Within this representation, we need to group genes into partitions in such a way that the building blocks of the partitions match a close-to-optimal solution. Recall that each node corresponds with a statistical variable. The arcs to and from that node correspond with dependences that are reflected in the data. Therefore, genes are linked (part of the same partition) if they involve the same node. The linkage of an efficient GA involves partitions whose size is bounded by a small constant. Therefore, we use only connections that correspond with plausible dependences. This effectively constrains the search to plausible DAGs.

The whole algorithm now consists of two phases. First, we construct from the database an undirected graph, called the *skeleton graph*, that represents the search space for the GA. The graph contains a node for each statistical variable present in the database. The edges of the skeleton graph are found by performing conditional dependence tests on the data: nodes are connected by edges if their corresponding variables are found to be dependent. In the next phase, the GA will turn the skeleton into a DAG by evolving a population of DAGs that use

the skeleton as a template. We observe that during the run of the GA, the graphical structures yielded need to be acyclic. Our GA ensures this property by applying a "repair" operator as soon as cycles are introduced during initialization and crossover. This repair operator breaks cycles in a manner that is not too disruptive. In addition, no node can have more than a predefined number of direct predecessors. This constraint is enforced because Bayesian networks with a large number of predecessors are impractical, since performing probabilistic inference on such networks is computationally prohibitive.

4.1 Generation of the Skeleton Graph

We construct from the data a skeleton graph to reduce the search space for the GA. The skeleton graph is constructed from the "important" edges for each node. An edge is deemed important if zero- and first-order tests show that the corresponding variables are dependent. We use just zero- and first-order conditional dependence tests, since higher-order tests quickly become unreliable. If we now consider the Bayesian network with the optimal (lowest) MDL score given the data, then, ideally, the skeleton graph is the underlying undirected graph of the DAG of the network. The skeleton graph, however, can include spurious edges when a higher-order independence test would have been required to remove a connection between two nodes. In our experiments, we found that skeleton graphs were produced that could contain approximately twice as many edges as the original network from which the data was sampled.

To construct the skeleton graph, we start with the empty graph and find for each node the nodes that should be connected to it. An edge is added between two nodes if their corresponding variables are dependent. The neighbors of a node X are found by building a candidate list of neighbors. This list contains all nodes Y that show a dependence on X through a zero-order dependence test. It is possible that two nodes (variables) are dependent without having to be connected, such as the nodes X and Y in Figure 2. These nodes become independent given Z. We therefore try to remove each node Y from the candidate list of neighbors for X by testing for a dependence on X given any other node Z in the list. If such a test shows that X and Y become independent given Z, then Y is removed from X's candidate list. For each node Y in the final list of neighbors of X we then add an edge between X and Y in the skeleton graph. The total number of dependence tests to be performed when constructing the skeleton graph is bounded by $O(n^2)$, where n denotes the number of statistical variables (and therefore nodes). In our experiments, up to 45 percent of the total running time was spent on calculating the skeleton graph.

We used the χ^2-test for determining whether or not two statistical variables are dependent given some conditioning set of variables. This test uses a threshold indicating the largest p-value that would cause it to reject the null-hypothesis that the variables are independent. A large threshold allows weak dependences to be found, but may also result in reporting incorrect dependences. A small threshold makes the test more selective. In practice, we found that the zero-order dependences from the original network could be recovered with a threshold

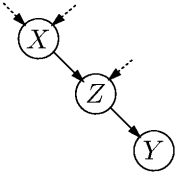

Fig. 2. Variables X and Y are dependent, yet independent given Z.

very close to zero. In fact, the p-values of the conditional tests rise significantly for nodes that are not true neighbors. In our experiments, therefore, we set the threshold as low as 0.005.

4.2 The GA

In the design of our GA for the BN learning problem, we applied the design principles mentioned before. The requirement of using an additively decomposable fitness function is satisfied by using the MDL measure. This measure is derived from information theory and states that the *description length* of a BN and a database is the sum of the size of the network, and the size of the database after it has been compressed using the network. The best network given a database then is the one yielding the smallest description length. We note that a network that is more complex can represent the underlying distribution of the data better and compress it to a smaller size. Complex networks, however, require a larger encoding to specify their arcs and tables. The MDL measure balances these two issues. It is to be minimized and has the following decomposed form [9]:

$$MDL(D, B) = \sum_{i=1}^{n} MDL_i(D, \pi(i)) \,,$$

where D denotes the database, B is the candidate Bayesian network, n is the number of nodes or variables, and $\pi(i)$ equals the set of predecessors of node i. The measure thus decomposes into local terms per node.

Having chosen an additively decomposable fitness function, we now outline the GA, indicating how the other design principles are satisfied:

Encoding: a string of genes, each of which corresponds with an edge in the skeleton graph, and can be set in one of the following states (the alleles): directedFrom, directedTo, none.

Initialization: an assignment of one of the three alleles is made to each gene at random. In other words, an edge is either removed or turned into an arc. All incoming arcs of nodes that thus end up with too many predecessors (recall that no node can have more than a given number of predecessors) are traversed in random order. An arc is reversed and if it then still causes an infeasible solution, it is removed. Finally, the solution is made acyclic with the operator MakeAcyclic.

Selection scheme: for selection and replacement, the incremental elitist recombination scheme [17] is used.

Mutation: no mutation in the traditional sense is used. Traditionally, random mutation is used to (re)generate building blocks during the run. In our GA, we ensure a good building-block supply by choosing an appropriately sized population and as a side-effect of the genetic operators (recombination, Repair and MakeAcyclic).

Recombination: to guarantee good mixing of building blocks, recombination is performed by crossover on the level of partitions. Through the crossover operator, we try to transfer schemata from the relevant partitions intact from parent to child. Since partitions overlap, this may still cause possible building blocks to be disrupted. Disruption is dealt with after crossover. For the crossover operator, a node is picked at random. The genes corresponding with the edges to its neighbors in the skeleton graph are added to the selection. The operator continues picking nodes until half the number of genes are selected. The first child now inherits the selection of genes from the first parent and its complement from the second one. The other child is constructed by taking the selection of genes from the second parent and its complement from the first one. The genetic operator Repair is applied to the edges in both children that are part of a possibly disrupted building block. Finally, MakeAcyclic is applied to both children.

Fitness function: the MDL measure is used to compute the fitness of solutions. It is minimized to find the best solution.

Stop criterion: the GA is halted when the difference between the average fitness in the population and the fitness of the best individual is smaller than a certain threshold (we used 0.001), or when the last 10% of the recombinations performed did not improve the average score by 1.

We describe the Repair and MakeAcyclic operators in more detail.

The Repair operator is used to handle the disruption of possible building blocks after crossover. We consider a node in one of the parents before recombination. The genes that correspond with the edges in the skeleton graph connected to that node can store a building block, that is, a combination of alleles for those genes that matches a close-to-optimal solution. If the node is picked by the crossover operator, these genes are transfered together to a child and no disruption occurs. If, on the other hand, the child inherits alleles for these genes from both parents, a building block may be disrupted. We can repair the building block by returning the genes inherited from one of the parents to their former setting. Since we don't know which settings are building blocks, we rely on the MDL score to find the best change. For our implementation, we make the simplifying assumption, for reasons of efficiency, that we can retrieve the original building block by changing genes one at a time. Moreover, to avoid changing a child too much, we try to find a minimal set of edges to be changed. The results obtained with the simplifying assumption are quite satisfactory.

The Repair operator now proceeds as follows. For every node, it investigates the genes that correspond with the edges connected to that node in the

skeleton graph. Suppose that the first parent donates x alleles and the second parent donates y alleles to a child. If $x < y$, the operator puts the x (indices to the) edges in a set S, otherwise the y edges. It then continues with subsequent nodes, disregarding edges already in S. For example, if the second node that is investigated has seven incident edges, of which two are already in S and three are inherited from the first parent, the remaining two edges are put in S. After all nodes have been visited, each edge in S is investigated for each child. If the allele for the gene corresponding with the edge can be changed to another allele with an improvement of the MDL score, the best change is chosen. In a typical run of the GA, for a skeleton of 95 edges, on average eight edges are changed in a child in the early iterations of the run. This number gradually drops to zero as the population converges.

The function MakeAcyclic operator is used to render a graph acyclic. It first finds all cycles in the graph by performing a recursive descent from each node of a spanning tree. Next, it finds the change for any edge that is part of a cycle that deteriorates the MDL score the least. Note that any change will break at least one cycle. This is repeated until all cycles have been resolved. If a change caused a new cycle or causes an arc to point to a node with the maximum number of predecessors, the operator reverts the change and marks it as forbidden. The process is then repeated (but forbidden states are avoided) until all cycles have been broken. Note that termination is guaranteed since a change to a deleted edge will break a cycle but not cause new conflicts.

It is worthwhile to point out that the running time of the GA can be dramatically optimized by caching the local score for a node with a certain predecessor set. Since the calculation of the MDL score is computationally intensive and has to be done by any MDL-based learning algorithm, the use of a population-based search method gives a relatively minor overhead. For example, a typical run with a population size of 150 for a database of 10000 cases took 14 minutes; a run with a population size of 1000—a more than six-fold increase—took only 35. In our experiments, the GA (using an incremental selection scheme) performed on average 2385 recombinations until convergence, corresponding with about 32 generations in a generational scheme.

5 Results

To study the performance of the resulting GA, we conducted experiments on various databases that were sampled from three real-life Bayesian networks by means of logic sampling [8]. The Alarm network [1] (37 nodes, 46 arcs) was originally built to help anesthetists monitor their patients, and is widely used for evaluating the performance of BN learning algorithms. The Oesoca network [19] (42 nodes, 59 arcs) was developed at Utrecht University, in collaboration with The Netherlands Cancer Institute, to aid gastroenterologists in staging oesophageal cancer and predicting the outcome of different treatment alternatives. The VSD network [14] (38 nodes, 52 arcs) also comes from the medical domain and was developed for treatment planning for a congenital heart disease.

We compared our GA against a rather straightforward hillclimber (HC), for a baseline performance. The hillclimber starts with the empty graph and considers pairs of nodes that are connected in the skeleton graph. In each step, it finds the change (remove, insert, or flip an arc) that improves the MDL score the most. This is continued until no progress is made, upon which MakeAcyclic is called to produce the result. We also compared our GA against the original implementation of the HEP algorithm by Wong et al. In our experiments, we used databases of 1000 and 10000 cases. The GA and the HEP algorithm were run ten times. All algorithms allowed a maximum of five predecessors per node. The GA used a population size of 150. The HEP algorithm used the default settings as suggested by Wong et al. The results are presented in Table 1.

Table 1. Results of the experiments. Below the name of the database, the MDL score of the original network is shown. In each table, the first column denotes the algorithm, the second gives the average MDL score with standard deviation, and the third column shows the best MDL score found in the ten runs.

	algo	avg±sd	best
Alarm-1000 (17834.3)	GA	16757.4±12.7	16747.1
	HEP	16591.9±40.4	16567.2
	HC	17387.3	
Oesoca-1000 (26070.0)	GA	24065.0±0.0	24065.0
	HEP	23800.3±72.8	23748.5
	HC	24676.6	
VSD-1000 (25790.6)	GA	23186.3±0.0	23186.3
	HEP	24641.7±52.8	24551.5
	HC	23318.8	

	algo	avg±sd	best
Alarm-10000 (139266.9)	GA	139240.4±89.7	139097.5
	HEP	140196.6±590.6	139441.7
	HC	140946.4	
Oesoca-10000 (213765.7)	GA	213089.8±0.0	213089.8
	HEP	213478.8±1096.1	212715.1
	HC	215192.8	
VSD-10000 (207647.4)	GA	206296.1±15.56	206251.2
	HEP	220160.1±958.3	219423.3
	HC	209101.0	

From the table we observe that both the GA and HEP yield networks with MDL scores that are close to the score of the original network. In fact, for all networks and associated databases, the GA found an average score that was better than the score of the original network. For all but two databases (Alarm-10000 and VSD-10000), the same observation holds for HEP. It may seem surprising that the original network is not the network of the best MDL score. The difference originates from the fact that the database is a finite sample, subject to sampling error, that does not reflect all the dependences from the original network accurately. The probability distribution observed in the data, therefore, is likely to differ from the distribution captured by the original network.

A striking difference in the results yielded by the two algorithms is that the GA shows a much smaller standard deviation than HEP. The GA can thus be seen as the more reliable of the two algorithms as it is more likely to always give results of similar quality. We observe, however, that for half the number of databases (Alarm-1000, Oesoca-1000 and Oesoca-10000) HEP's best result is slightly better than the GA's. We further observe that the hillclimber also

performs quite well, thereby giving an indication of the benefit from the skeleton graph. The average MDL score yielded by the GA is always better than that by the HC, whereas HEP's best solution can be worse than the result from the HC.

The GA and the HEP algorithm yielded their results fairly quickly, within a maximum of 20 minutes per run. For the databases of 1000 cases, the longest running time was four minutes. HEP was generally faster than the GA, with a maximal difference in running time by a factor eight. It is important to realize, however, that for the BN learning problem, the time spent is not a critical factor. Once a network is learned, it is either used directly for daily problem solving, or inspected and improved manually. The time spent on learning the network, therefore, is well within reasonable bounds for both algorithms.

6 Conclusions

Building upon recent developments in GA theory we discussed various principles for GA design. We demonstrated how to apply these design principles, rather straightforwardly, to derive a GA for learning Bayesian networks from data that performs at least comparably to the state-of-the-art algorithms. We feel that the robustness of the resulting GA is a confirmation of the feasibility of our approach. For future work, we would like to put the model from which our design principles were derived on a more formal footing. Specifically, it is worth investigating how overlap between partitions influences the predictions of the model.

Acknowledgments. This research was supported by the Netherlands Organisation for Scientific Research (NWO). We would like to thank Wong, Lee and Leung for graciously providing us with an implementation of their HEP algorithm.

References

1. I. A. Beinlich, H. J. Suermondt, R. M. Chavez, and G. F. Cooper. The Alarm monitoring system: a case study with two probabilistic inference techniques for belief networks. In J. Hunter et al., editors, *Proceedings of the Second Conference on Artificial Intelligence in Medicine*, pages 247–256. Springer, 1989.
2. G. F. Cooper and E. Herskovits. A Bayesian method for the induction of probabilistic networks from data. *Machine Learning*, 9:309–347, 1992.
3. C. Cotta and J. Muruzábal. Towards more efficient evolutionary induction of Bayesian networks. In J.-J. M. Guervós et al., editors, *Lecture Notes in Computer Science, Volume 2439: Proceedings of the Parallel Problem Solving from Nature VII Conference*, pages 730–739. Springer-Verlag, 2002.
4. L. M. de Campos and J. F. Huete. A new approach for learning belief networks using independence criteria. *International Journal of Approximate Reasoning*, 24(1):11–37, 2000.
5. D. E. Goldberg, K. Deb, and J. H. Clark. Genetic algorithms, noise, and the sizing of populations. *Complex Systems*, 6(4):333–362, 1992.

6. G. Harik, E. Cantú-Paz, D. E. Goldberg, and B. L. Miller. The gambler's ruin problem, genetic algorithms, and the sizing of populations. *Evolutionary Computation*, 7(3):231–253, 1999.
7. D. Heckerman, D. Geiger, and D. M. Chickering. Learning Bayesian networks: The combination of knowledge and statistical data. *Machine Learning*, 20(3):197–243, 1995.
8. M. Henrion. Propagating uncertainty in Bayesian networks by probabilistic logic sampling. In J. F. Lemmer and L. N. Kanal, editors, *Proceedings of the Second Conference on Uncertainty in Artificial Intelligence*, pages 149–163. Elsevier, 1988.
9. W. Lam and F. Bacchus. Using causal information and local measures to learn Bayesian networks. In D. Heckerman and A. Mamdani, editors, *Proceedings of the Ninth Conference on Uncertainty in Artificial Intelligence*, pages 243–250. Morgan-Kaufmann, 1993.
10. P. Larrañaga, C. M. H. Kuijpers, R. H. Murga, and Y. Yurramendi. Learning Bayesian network structures by searching for best ordering with genetic algorithm. *IEEE Transactions on Systems, Man, and Cybernetics*, 26(4):487–493, 1996.
11. P. Larrañaga, M. Poza, Y. Yurramendi, R. Murga, and C. Kuijpers. Structure learning of Bayesian networks by genetic algorithms: A performance analysis of control parameters. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(9):912–926, 1996.
12. B. L. Miller and D. E. Goldberg. Genetic algorithms, selection schemes, and the varying effects of noise. *Evolutionary Computation*, 4(2):113–131, 1996.
13. J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.
14. N. Peek and J. Ottenkamp. Developing a decision-theoretic network for a congenital heart disease. In E. Keravnou et al., editors, *Proceedings of the Sixth European Conference on Artificial Intelligence in Medicine*, pages 157–168. Springer, 1997.
15. J. J. Rissanen. Modelling by shortest data description. *Automatica*, 14:465–471, 1978.
16. P. Spirtes and C. Glymour. An algorithm for fast recovery of sparse causal graphs. *Social Science Computer Review*, 9(1):62–73, 1991.
17. D. Thierens. Selection schemes, elitist recombination, and selection intensity. In T. Bäck, editor, *Proceedings of the Seventh International Conference on Genetic Algorithms and their Applications*, pages 152–159. Morgan-Kaufmann, 1997.
18. D. Thierens and D. E. Goldberg. Mixing in genetic algorithms. In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms and their Applications*, pages 38–45. Morgan-Kaufmann, 1993.
19. L. C. van der Gaag, S. Renooij, C. Witteman, B. Aleman, and B. Taal. Probabilities for a probabilistic network: A case-study in oesophageal cancer. *Artificial Intelligence in Medicine*, 25(2):123–148, 2002.
20. S. van Dijk, D. Thierens, and M. de Berg. Scalability and efficiency of genetic algorithms for geometrical applications. In M. Schoenauer et al., editors, *Lecture Notes in Computer Science, Volume 1917: Proceedings of the Parallel Problem Solving from Nature VI Conference*, pages 683–692. Springer-Verlag, 2000.
21. S. van Dijk, D. Thierens, and M. de Berg. On the design and analysis of competent GAs. Technical Report TR-2002-15, Utrecht University, 2002.
22. M. L. Wong, S. Y. Lee, and K. S. Leung. A hybrid data mining approach to discover Bayesian networks using evolutionary programming. In W. B. Langdon et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference*. Morgan-Kaufmann, 2002.