Speeding up the learning of equivalence classes of Bayesian network structures

Daly, Ronan; Aitken, Stuart; Shen, Qiang

Publication date:
2006

Citation for published version (APA):
Daly, R., Aitken, S., & Shen, Q. (2006). *Speeding up the learning of equivalence classes of Bayesian network structures*. 34-39.

SPEEDING UP THE LEARNING OF EQUIVALENCE CLASSES OF BAYESIAN NETWORK STRUCTURES

Rónán Daly
School of Informatics
University of Edinburgh
Edinburgh EH8 9LE, UK
Ronan.Daly@ed.ac.uk

Qiang Shen
Department of Computer Science
University of Wales, Aberystwyth
Aberystwyth SY23 3DB, UK
qqs@aber.ac.uk

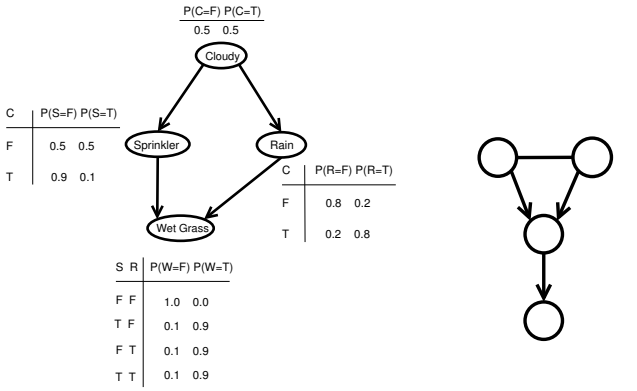
Stuart Aitken
School of Informatics
University of Edinburgh
Edinburgh EH8 9LE, UK
stuart@aiai.ed.ac.uk

ABSTRACT

For some time, learning Bayesian networks has been both feasible and useful in many problems domains. Recently research has been done on learning equivalence classes of Bayesian networks, i.e. structures that capture all of the graphical information of a group of Bayesian networks, in order to increase learning speed and quality. However learning speed still remains quite slow, especially on problems with many variables. This work aims to describe a method to speed up algorithm learning speed. A brief overview of learning Bayesian networks is given. A method is then given, so that tests of whether a particular move is valid can be cached. Finally, experiments are conducted, which show that applying this caching method produces a marked increase in learning speed.

KEY WORDS

Bayesian networks, belief networks, machine learning, probability, data mining, optimization.



(a) Example Bayesian network (b) Example PDAG

Figure 1. Example network and PDAG

1 Introduction

The task of learning Bayesian networks from data has, in a relatively short amount of time, become a mainstream application in the process of knowledge discovery and model building [1, 2]. The reasons for this are many.

For one, the model built by the process has an intuitive feel — this is because a Bayesian network consists of a directed acyclic graph (DAG), with conditional probability tables annotating each node. Each node in the graph represents a variable of interest in the problem domain and the arcs can (with some caveats) be seen to represent causal relations between these variables — the nature of these causal relations is governed by conditional probability tables associated with each node/variable. An example Bayesian network is shown in Figure 1.

Another reason for the usefulness of Bayesian networks is that aside from the visual attractiveness of the model, the underlying theory is quite well understood and has a solid foundation. A Bayesian network can be seen as a factorisation of a joint probability distribution, with the conditional probability distributions at each node making up the factors and the graph structure making up their method of combination. Because of this equivalence, the network can answer any probabilistic question put to it, regarding the variables modelled.

Finally, the popularity of Bayesian networks has been increased by the accessibility of methods to query the model and learn both the structure and parameters of the network. It has been shown that inference in Bayesian networks is NP-Complete [3, 4], but approximate methods have been found to perform this operation in an acceptable amount of time. Learning the structure of Bayesian networks is also NP-Complete [5], but here too, heuristic methods have been found to render this operation tractable.

It is with the latter remark that this paper concerns itself, that is, the learning of the structure of a Bayesian network from a sample of data. There are generally three different methods used in this task. The first finds conditional independencies in the data and then uses these conditional independencies to produce the structure [6]. The second uses dynamic programming and optionally, clustering, to construct a DAG [7, 8]. The third method — which we will be dealing with — defines a search on the space of Bayesian networks, using a scoring function defined by the implementer, that says relatively how good the network is.

This paper will seek to establish a method to increase the learning speed of a search algorithm. This will be done by caching certain data, necessary to establish the validity of a move in the search space of a learning algorithm. This

is in order to minimise the effort in recomputing data. Also, the conditions under which the cache entries are valid will be examined. To this end, the rest of this paper will be structured in the following fashion.

Firstly, there will be a more in depth study of the problem of searching for an optimum Bayesian network, in both the space of Bayesian networks themselves and of equivalence classes of Bayesian networks. Then, a new method of caching validity tests will be shown, along with the conditions under which the cached results are valid, invalid or out of date. Next, results of tests against uncached algorithms will be discussed and finally, any conclusions and possible future directions will be stated.

2 Learning Bayesian Networks

To start out this section, some definitions and notation are introduced.

A graph $\mathcal{G}$ is given as a pair (V, E), where $V = \{v_1, \ldots, v_n\}$ is the set of vertices or nodes in the graph and E is the set of edges or arcs between the nodes in V. A directed graph is a graph where all the edges have an associated direction from one node to another. A directed acyclic graph or DAG, is a directed graph, without any cycles, i.e. it is not possible to return to a node in the graph by following the direction of the arcs.

A Bayesian network on a set of variables $V = \{v_1, \ldots, v_n\}$ is a pair $(\mathcal{G}, \Theta)$, where $\mathcal{G} = (V, E)$ is a DAG and $\Theta = \{\theta_1, \ldots, \theta_n\}$ is a set of conditional probability distributions, where each θ_i is associated with each v_i.

In learning a Bayesian network from data, both the structure $\mathcal{G}$ and parameters Θ must be learned, normally separately. In the case of complete multinomial data, the problem of learning the parameters is easy, with a simple closed form formula for Θ [9]. However, in the case of learning the structure, no such formula exists and other methods are needed. In fact, learning the structure is an NP-Hard problem and consequently enumeration and test of all network structures is not likely to succeed. In fact with just ten variables there are roughly 10^{18} possible DAGs, which leaves heuristic search methods through the space of different structures as possibly the only tractable solution.

In order to create a space in which to search through, three components are needed. Firstly all the possible solutions must be identified as the set of states in the space. Secondly a representation mechanism for each state is needed. Finally a set of operators must be given, in order to move from state to state in the space.

Traditionally, in searching for a Bayesian network structure, the set of states was the set of possible Bayesian network structures, the representation was a DAG and the set of operators were various small local changes to a DAG, e.g. adding, removing or reversing an arc. Successful application of the operators was also dependent on the changed graph being a DAG, i.e. that no cycle was formed in applying the operator. In keeping with the terminology used by Chickering this space shall be called B-space [10].

Once the search space has been defined, two other pieces are needed to complete the search algorithm, a scoring function which evaluates the "goodness of fit" of a structure with a set of data and a search procedure that decides which operator to apply, normally using the scoring function to see how good a particular operator application might be. An example search procedure is greedy search, that at every stage applies the operator that produces the best change in the structure, according to the scoring function. As for the scoring function, various formulæ have been found to see how well a DAG fits a data sample, e.g. by giving the relative posterior probability [11], or using a large-sample approximation such as the Bayesian information criterion [9].

3 Searching in the Space of Equivalence Classes

According to many scoring criteria, there are DAGs that are equivalent to one another, in the sense that they will produce the same score as each other. Looking at this in more depth, it is found that these equivalent DAGs produce the same set of independence constraints as each other, even though the structures are different. Independence constraints show how a set of variables are influenced or dependent on another set of variables, given a certain third set of variables. These constraints can be checked by analysing the DAG for certain structures. It turns out, according to a theorem by Verma and Pearl, that two DAGs are equivalent iff they have the same skeletons and the same v-structures [12]. By skeleton, it is meant the undirected graph that results in undirecting all edges in a DAG and by v-structure, (sometimes referred to as a morality) it is meant a head-to-head meeting of two arcs, where the tails of the arcs are not joined. From this notion of equivalence, a class of DAGs that are equivalent to each other can be defined, notated here as $Class(\mathcal{G})$.

Because of this apparent redundancy in the space of DAGs, attempts have been made to conduct the search for Bayesian network structures in the space of equivalence classes of DAGs [10, 13, 14]. The search set of this space is the set of equivalence classes of DAGs. To represent the states of this set, a different type of structure is used, known as a partially directed acyclic graph (PDAG). A PDAG (an example of which is shown in Figure 1) is a graph that contains both undirected and directed edges and that contains no directed cycles and will be notated herein as $\mathcal{P}$. Again, the equivalence class of DAGs corresponding to a PDAG is denoted as $Class(\mathcal{P})$, with a DAG $\mathcal{G} \in Class(\mathcal{P})$ iff $\mathcal{G}$ and $\mathcal{P}$ have the same skeleton and same set of v-structures. Related to this is the idea of a *consistent extension*. If a DAG $\mathcal{G}$ has the same skeleton and the same set of directed edges as a PDAG $\mathcal{P}$ then it is said that $\mathcal{G}$ is a consistent extension of $\mathcal{P}$. Not all PDAGs have a DAG that is a consistent extension of itself. If a consistent extension exists, then it is said that the PDAG *admits* a consistent extension. Only PDAGs that admit a consistent extension can be used to represent an equivalence class of DAGs and hence a Bayesian network.

Directed edges in a PDAG can be either compelled, or made to be directed that way, whilst others are reversible, in that they could be undirected and the PDAG would still rep-

resent the same equivalence class. From this idea, we can define a completed PDAG (CPDAG), where every undirected edge is reversible in the equivalence class and every directed edge is compelled in the equivalence class. We shall denote a CPDAG as $\mathcal{P}^{\mathcal{C}}$. It can be shown that there is a one-to-one mapping between a CPDAG $\mathcal{P}^{\mathcal{C}}$ and $Class(\mathcal{P}^{\mathcal{C}})$.

With this representation of equivalence classes of Bayesian network structures and a set of operators that modify the PDAGs which represent them (e.g. Insert an undirected arc, Insert a directed arc etc.), a search procedure can proceed. But one might ask why go to the bother of this type of search. Firstly, an equivalence class can represent many different DAGs in a single structure. Search in the space of DAGs often moves between states with the same equivalence class and so, in a sense, is wasted effort. This also affects the connectivity of the search space, in that the ability to move to a particular neighbouring equivalence class can be constrained by the particular representation given by a DAG.

There is also the problem given by the prior probability used in the scoring function in that whilst searching through the space of DAGs, certain equivalence classes can be over represented by this prior because there are many more DAGs contained in the class.

These concerns have motivated researchers, with the results that recent implementations of algorithms that search through the space of equivalence classes have produced results that show a marked improvement in execution time and a small improvement in learning accuracy, depending on the type of data set [13, 15].

3.1 Techniques for Searching through Equivalence Classes

Note that here we refer to a *move* as an application of an operator to a particular state in the search space.

To be able to conduct a search through the space of equivalence classes, a method must be able to find out whether a particular move is valid and if valid, how good that move is. These tasks are relatively easy whilst searching through the space of DAGs — a check whether a move is valid is equivalent to a check whether a move keeps a DAG acyclic. The goodness of such a move is found out by using the scoring function, but rather then scoring each neighbouring DAG in the search space, the decomposability of most scoring criterion can be taken advantage of, with the result that only nodes whose parent sets have changed need to be scored.

However, this task of checking move validity and move score is not as easy in the space of equivalence classes. For one, instead of just checking for cycles, checks also have to be made so that unintended v-structures are not created in a consistent extension of a PDAG. Scoring a move also creates difficulties, as it is hard to know what extension and hence what changes in parent sets of nodes will occur, without actually performing this extension. Also, a local change in a PDAG *might* make a non-local change in a corresponding extension and so force unnecessary applications of the score function.

These problems were voiced as concerns by Chickering [10]. In this paper he performs validity checking of moves by trying to obtain a consistent extension of the resulting PDAG — if none exists then the move is not valid. Scoring the move was achieved by scoring the changed nodes in the consistent extension given. These methods were very generic, but resulted in a significant slowdown in algorithm execution, compared to search in the space of DAGs.

To alleviate this problem, authors proposed improvements that would allow move validity and move score to be computed without needing to obtain a consistent extension of the PDAG [14, 13]. This was done by defining an explicit set of operators, with each operator having a validity test and corresponding score change function, that could be calculated on the PDAG. These changes resulted in a speedup of the execution time of the algorithm, with the result that search in the space of equivalence classes of Bayesian networks became competitive with search in the space of Bayesian networks.

4 A Caching Method for Increasing Learning speed

Whilst the execution time of searching for equivalence classes of Bayesian networks has decreased, it still remainsquite high for problem instances with many variables. This is especially so if the search algorithm needs multiple traversals through the search space. Upon analysing the execution of a typical greedy search, it was found that much of the time was spent computing validity tests for the various operators. However, the results of many of these tests are still valid after most moves in the space. This led to the realisation that these results could be cached, so as to be used in the next iteration of the search. For this to be useful, a method would be needed that could say when certain cached tests became invalid and when new entries could be added into the cache. In this paper, such a method is proposed. A high level description is as follows.

Each state of the search space is represented as a CPDAG. A particular move might change this into a general PDAG. This needs to be transformed back into a CPDAG for the next iteration. In general, due to this transformation, there can be an arbitrary number of differences between the two CPDAGs, which can be represented by a set of primitive changes of the type of arcs in the CPDAGs. There are seven such primitive changes Insert Undirected, Delete Undirected, Insert Directed, Delete Directed, Reverse Directed, Direct Arc and Undirect Arc. If the effect of each primitive change on the validity of all moves can be ascertained, there will be a method to update cached values.

One such way to find the effect of changes is to look at what makes a particular move valid in the first place. For the purposes of this paper, the set of six operators proposed by Chickering will be analysed [13]. Take, for example, the InsertU operator that inserts an undirected link between x and y. For this move to be valid, two tests must be passed. Firstly, every undirected path from x to y must contain a node in $N_{x,y}$, that is, the intersection of the undirected neighbours of x and y. Secondly, both x and y must

Op	Action
IU_{xy}	Check := $\Xi_x\,(\Xi_y, x) \cup (\Xi_x, y)$
DU_{xy}	Check := $(\Xi_y, x) \cup (\Xi_x, y)$
ID_{xy}	Check := (y, N_x)
DD_{xy}	Check := (y, N_x)
RD_{xy}	Check := $(y, N_x) \cup (x, N_y)$
DA_{xy}	Check := $(y, N_x) \cup (y, \Xi_x)$
	Invalid := (Ξ_y, x)
UA_{xy}	Invalid := (y, N_x)
	Check := $(\Xi_y, x) \cup (\Xi_x, y)$

Table 1. $\Omega_{x,y}$ is a clique

Op	Action
IU_{xy}	—
DU_{xy}	—
ID_{xy}	Invalid := $\{y, V \backslash \Xi_x\}$
	Check := $\{y, \Xi_x \setminus y\}$
DD_{xy}	Invalid := $\{y, \Xi_x\}$
	Check := $\{y, (V \backslash \Xi_x) \backslash y\}$
RD_{xy}	Invalid := $\{y, \Xi_x \cup x\}$
	Check := $\{y, V \backslash (\Xi_x \cup x)\}$
	Invalid := $\{x, V \backslash \Xi_y\}$
	Check := $\{x, \Xi_y \backslash x\}$
DA_{xy}	Invalid := $\{y, V \backslash y\}$
	Check := $\{y, \Xi_x \backslash y\}$
UA_{xy}	Invalid := $\{y, \Xi_x\}$
	Check := $\{y, V \backslash \Xi_x\}$

Table 2. $\Pi_x = \Pi_y$

Op	Action
IU_{xy}	Check := $x \cup y \cup N_{x,y}$
DU_{xy}	Check := $x \cup y \cup N_{x,y}$
ID_{xy}	Check := y
DD_{xy}	Check := y
RD_{xy}	Check := $x \cup y$
DA_{xy}	Check := $x \cup y$
UA_{xy}	Check := $x \cup y$

Table 3. N_y is a clique

have the same parent sets. Looking at the first test we can see that inserting an undirected link $w - z$ in the graph might affect whether the first test is true or not. In fact it might only affect it if x has an undirected path to w and y has an undirected path to z. With observations such as these, we can limit the number of tests that we need to do at each iteration of the algorithm. Specifically, for each *primitive operator*, we can find the affect it has on each validity test in terms of three sets of parameters to the validity tests — whether the test is valid, whether it is invalid or whether it needs to be checked. We can then take the parameters from each of the validity tests of an operator, couple this with whether a move's score might have changed and find out whether a move needs to be added, deleted or updated in the cache. A list of the effects on each of the validity tests by each of the primitive operators will now be given, after a short word on notation.

Op	Action
IU_{xy}	$C_x^{N\Xi}, C_y^N, C_x^N, C_y^{N\Xi}$ in $\mathcal{P}' \setminus (N_{x,y} \cup x - y)$
	Invalid := $\left(C_x^{N\Xi} \backslash C_y^{N\Xi}, C_y^N \setminus y\right) \cup \left(C_x^{N\Xi} \backslash \Xi_x, y\right)$
	Invalid := $\left(C_y^{N\Xi} \backslash C_x^{N\Xi}, C_x^N \setminus x\right) \cup \left(C_y^{N\Xi} \backslash \Xi_y, x\right)$
	Check := $(\Xi_x, y) \cup (\Xi_y, x)$
	Check := $\left(C_x^{N\Xi} \cap C_y^{N\Xi}, C_y^N\right) \cup \left(C_y^{N\Xi} \cap C_x^{N\Xi}, C_x^N\right)$
DU_{xy}	$C_x^{N\Xi}, C_y^{N\Xi}, C_x^N, C_y^N$ in $\mathcal{P}'$
	Valid := $\left(C_x^{N\Xi} \setminus C_y^{N\Xi}, C_y^N\right)$
	Valid := $\left(C_y^{N\Xi} \setminus C_x^{N\Xi}, C_x^N\right)$
	C_x, C_y, C_x^N, C_y^N in $\mathcal{P}' \setminus N_{x,y}$
	Check := $\left(\left(\bigcap_{i \in N_{x,y}} \Xi_i\right) \cap C_x^{\Xi}, \left(\bigcap_{i \in N_{x,y}} N_i\right) \cap C_y\right)$
	Check := $\left(\left(\bigcap_{i \in N_{x,y}} \Xi_i\right) \cap C_y^{\Xi}, \left(\bigcap_{i \in N_{x,y}} N_i\right) \cap C_x\right)$
ID_{xy}	$C_y^{\Xi}, C_x^{N\Pi}$ in $\mathcal{P}'$
	Invalid := $\left(C_y^{\Xi} \setminus y, C_x^{N\Pi}\right) \cup \left(y, C_x^{N\Pi} \backslash N_x\right)$
	Check := (y, N_x)
DD_{xy}	$C_y^{N\Xi}, C_x^{N\Xi}, C_x^{N\Pi}$ in $\mathcal{P}'$
	Check := $\left(C_y^{N\Xi} \backslash C_x^{N\Xi}, C_x^{N\Pi}\right)$
RD_{xy}	$C_x^{N\Xi}, C_y^{N\Xi}, C_x^{N\Pi}, C_y^{N\Pi}$ in $\mathcal{P}'$
	Invalid := $\left(C_x^{\Xi} \setminus x, C_y^{N\Pi}\right) \cup \left(x, C_y^{N\Pi} \backslash N_y\right)$
	Check := $\left(C_y^{N\Xi} \backslash C_x^{N\Xi}, C_x^{N\Pi}\right)$
DA_{xy}	$C_x^{N\Xi}, C_y^{N\Pi}$ in $\mathcal{P}'$
	Check := $\left(C_x^{N\Xi} \backslash C_y^{N\Xi}, C_y^{N\Pi}\right)$
	Check := (y, N_x)
UA_{xy}	$C_x^{N\Xi}, C_y^{N\Xi} C_y$ in $\mathcal{P}' \setminus (N_{x,y} \cup x - y)$
	Invalid := $\left(C_x^{N\Xi} \backslash C_y^{N\Xi}, C_y^N\right) \cup \left(C_x^{N\Xi} \backslash Ch_x, y\right)$
	Check := (Ch_y, x)
	Check := $\left(C_x^{N\Xi} \cap C_y^{N\Xi}, C_y\right)$

Table 4. Every semi-directed path from y to x contains a node in $\Omega_{x,y}$

Much of Chickering's notation is reused and hence N_x is the set of undirected neighbours of x and Π_x is the set of parents of x. We also add Ξ_x as the set of children of x. $N_{x,y}$ is $N_x \cap N_y$ and $\Omega_{x,y}$ is $\Pi_x \cap N_y$. In order to specify nodes that are removed from a particular primitive operator, we use $C_x^{\Sigma}$where Σ is a combination of N, Π and Ξ. This stands for the set of nodes that are reachable from x, including x, by a combination of the given arc types. For example, $C_x^{N\Xi}$ is the set of nodes reachable from x by following undirected arcs and directed arcs away from the current node. As a shorthand C_x is equivalent to C_x^N. Continuing on, $\mathcal{P}$ is the previous PDAG and $\mathcal{P}'$ is the

Op	Action
IU_{xy}	$C_x^{N\Xi}, C_y^{N\Xi}$ in $\mathcal{P}' \setminus (N_{x,y} \cup x - y)$
	Check := $\left(C_x^{N\Xi}, C_y^{N\Xi}\right) \cup \left(C_y^{N\Xi}, C_x^{N\Xi}\right)$
DU_{xy}	$C_x^{N\Xi}, C_y^{N}, C_x^{N}, C_y^{N\Xi}$ in $\mathcal{P}' \setminus N_{x,y}$
	Check := $\left(C_x^{N\Xi}, C_y^{N}\right) \cup \left(C_y^{N\Xi}, C_x^{N}\right)$
ID_{xy}	$C_x^{N\Pi}, C_y^{\Xi}$ in new graph
	Check := $\left(C_y^{\Xi}, C_x^{N\Pi}\right)$
	Check := (Π_y, y)
DD_{xy}	$C_x^{N\Pi}, C_y^{\Xi}$ in $\mathcal{P}'$
	Check := $\left(C_x^{N\Pi}, C_y^{\Xi}\right)$
RD_{xy}	$C_x^{\Xi}, C_y^{\Xi}, C_x^{N\Pi}, C_y^{N\Pi}$ in $\mathcal{P}'$
	Check := $\left(C_x^{\Xi}, C_y^{N\Pi}\right) \cup \left(C_y^{\Xi} - C_x^{N\Pi}\right)$
	Check := $(\Pi_x, x) \cup (x, \Xi_x) \cup \left(y, C_x^{\Xi}\right)$
DA_{xy}	$C_x^{N\Xi}, C_y^{N}$ in $\mathcal{P} \setminus x - y$
	Invalid := $\left(C_y^{N}, C_x^{N\Xi}\right)$
	Check := $(\Pi_x \cup x, y) \cup (y, \Xi_y)$
UA_{xy}	$C_y^{N}, C_x^{N\Pi}$ in $\mathcal{P}'$
	Check := $\left(C_y^{N}, C_x^{N\Pi}\right)$

Table 5. Every semi-directed path from x to y that does not include the edge $x \rightarrow y$ contains a node in $\Omega_{y,x} \cup N_y$

Op	Action
IU_{xy}	Check := $\{x, N_{x,y}\} \cup \{y, N_{x,y}\} \cup \{x, y\}$
DU_{xy}	Check := $\{x, N_{x,y}\} \cup \{y, N_{x,y}\} \cup \{x, y\}$
ID_{xy}	—
DD_{xy}	—
RD_{xy}	—
DA_{xy}	Check := $\{x, N_{x,y}\} \cup \{y, N_{x,y}\} \cup \{x, y\}$
UA_{xy}	Check := $\{x, N_{x,y}\} \cup \{y, N_{x,y}\} \cup \{x, y\}$

Table 6. $N_{x,y}$ is a clique

current PDAG. When we say $\mathcal{P} \setminus (N_{x,y} \cup x - y)$ we mean the PDAG $\mathcal{P}$ less the nodes in $N_{x,y}$ and the arc $x - y$.

A list of the effects of each primitive operator on each validity test are given in Tables 1, 2, 3, 4, 5, 6, 7 and 8. The notation $\{X, Y\}$ means the set of unordered pairs given by each pair of the elements in X and Y. The notation (X, Y) is similar, but for ordered pairs instead of unordered. The set of *invalid* entries show the parameters for which a particular test could not succeed. Therefore any operator relying on this test could delete the relevant entries in a cache. The set of *valid* entries show the parameters for which that particular test is valid (though perhaps not other relevant tests). Assuming that the other tests are valid, the cache entry can remain, though perhaps with a changed score. Finally the set of *check* entries show parameters that will have to be rechecked to see if they are valid or not.

5 Experimental Results

Experiments on two datasets were performed in order to ascertain whether a measurable improvement was gained by implemented the caching system. For the first experiment,

Op	Action
IU_{xy}	C_x, C_y in $\mathcal{P}' \setminus (N_{x,y} \cup x - y)$
	Check := $\{C_x \setminus x, C_y \setminus y\} \cup \{x, C_y \setminus N_y\} \cup \{y, C_x \setminus N_x\}$
	Check := $\{x, N_y\} \cup \{y, N_x\}$
DU_{xy}	C_x, C_y in $\mathcal{P}'$
	Valid := $\{C_y \setminus C_x, C_x \setminus C_y\}$
	C_x, C_y in $\mathcal{P}' \setminus N_{x,y}$
	Check := $\left\{\left(\bigcap_{i \in N_{x,y}} N_i\right) \cap C_x, \left(\bigcap_{i \in N_{x,y}} N_i\right) \cap C_y\right\}$
ID_{xy}	—
DD_{xy}	—
RD_{xy}	—
DA_{xy}	C_x, C_y in $\mathcal{P} \setminus x - y$
	Valid := $\{C_x, C_y\}$
UA_{xy}	C_x, C_y in $\mathcal{P}$
	Invalid := $\{C_x \setminus x, C_y \setminus y\} \cup \{x, C_y \setminus N_y\} \cup \{y, C_x \setminus N_x\}$
	Check := $\{x, N_y\} \cup \{y, N_x\}$

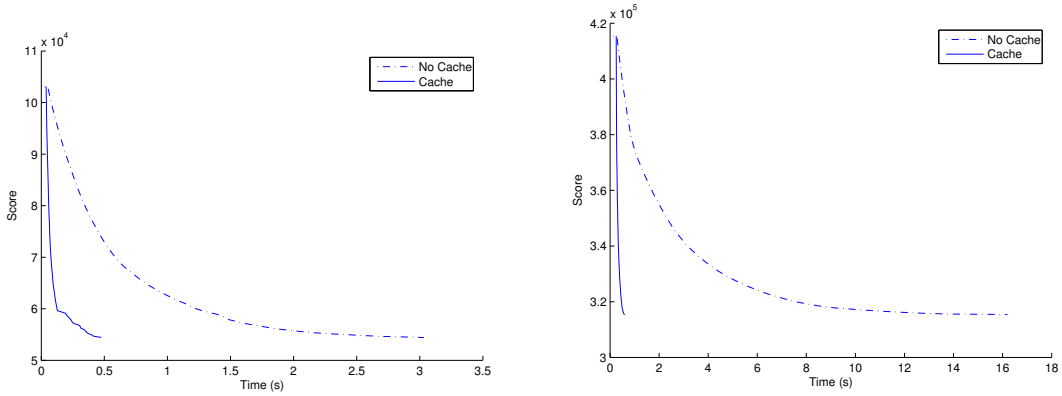
Table 7. Every undirected path from x to y contains a node in $N_{x,y}$

Op	Action
IU_{xy}	—
DU_{xy}	—
ID_{xy}	Valid := $(y, V \setminus \Xi_x)$
	Check := $(y, \Xi_x \setminus y)$
DD_{xy}	Add := (y, Ξ_x)
	Check := $(y, V \setminus (\Xi_x \cup x))$
RD_{xy}	Add := (y, Ξ_x)
	Check := $(y, V \setminus (\Xi_x \cup x))$
	Add := $(x, V \setminus (\Xi_y \cup y))$
	Check := $(x, \Xi_y \setminus x)$
DA_{xy}	Check := $(y, V \setminus \Xi_x)$
	Check := $(y, \Xi_x \setminus y)$
UA_{xy}	Add := (y, Ξ_x)
	Check := $(y, V \setminus (\Xi_x \cup x))$

Table 8. $\Pi_x \neq \Pi_y$

5000 instances were sampled from the ALARM Bayesian network, which contains 37 variables [16]. The second experiment used 5000 samples from the Insurance Company Benchmark (COIL 2000) which contains 86 variables. For both experiments, the time taken for a particular score was averaged over 100 runs, for both the cached and uncached algorithms. Plots of the results can be seen in Figure 2.

As can be seen the cached version of the algorithm performed better than the uncached version, especially with the larger COIL dataset. This is consistent with the hypothesis that caching results from validity checks speeds up algorithm execution. In fact, for large periods of the runtime, each step appears to run in roughly constant time. This is in contrast to the uncached version where the average step period appears to increase as the search continues. One plausible explanation for this behaviour comes from the fact that as the search progresses, the graph becomes increasingly more connected. This would cause the



(a) Alarm Dataset

(b) COIL Dataset

Figure 2. Score of an equivalence class as a function of the search time

"path" validity conditions (which would normally take time in $O(|S| + |E|)$) to increase in time needed. When the caching system is used, most of these conditions can be reused, with only a small amount of tests needing to be re-run, mostly independent of the number of variables in the graph.

6 Conclusions and Further Work

A method has been demonstrated that shows promise in speeding up the computation of learning equivalence classes of Bayesian networks by searching through a state space. This method would be especially useful in algorithms that take multiple passes through the state space. Results of experiments have shown that a measurable speed up was gained by caching the results of validity tests whilst looking through the state space.

There remains much work to be done in investigating the long term effect of this work. Firstly, the long lists of actions complicate the implementation and render it more opaque. If a method could be found to render the lists shorter or more comprehensible, this would be of great benefit. Secondly, formal proofs for each of the operators actions given are needed in order to be confident in the reliability of the method. Thirdly, a more extensive range of experiments needs to be run in order to check performance over many problem domains.

References

[1] D. Heckerman, A. Mamdani *et al.*, Real-world applications of bayesian networks, *Communications of the ACM*, 38(3), 1995, 24–26.

[2] N. Friedman, Inferring cellular networks using probabilistic graphical models, *Science*, 303(5679), 2004, 799–805.

[3] P. Dagum and M. Luby, Approximating probabilistic inference in Bayesian belief networks is NP-hard, *Artificial Intelligence*, 60(1), 1993, 141–154.

[4] S. E. Shimony, Finding maps for belief networks is NP-hard, *Artificial Intelligence*, 68(2), 1994, 399–410.

[5] D. M. Chickering, Learning Bayesian networks is NP-complete, in D. Fisher and H. Lenz (Eds.) *Learning from Data: Artificial Intelligence and Statistics V*, chap. 12 (Springer-Verlag, 1996), 121–130.

[6] P. Spirtes, C. Glymour *et al.*, *Causation, Prediction, and Search*, Adaptive Computation and Machine Learning, 2nd ed. (The MIT Press, 2000).

[7] S. Ott, S. Imoto *et al.*, Finding optimal models for small gene networks, in *Proceedings of the Ninth Pacific Symposium on Biocomputing* (World Scientific, 2004), 557–567.

[8] S. Ott and S. Miyano, Finding optimal gene networks using biological constraints, *Genome Informatics*, 14, 2003, 124–133.

[9] D. Heckerman, A tutorial on learning with Bayesian networks, Tech. Rep. MSR-TR-95-06, Microsoft Research, 1995.

[10] D. M. Chickering, Learning equivalence classes of Bayesian network structures, in F. Jensen and E. Horvitz (Eds.) *Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence* (San Francisco, California: Morgan Kaufmann, 1996), 150–157.

[11] D. Heckerman, D. Geiger *et al.*, Learning Bayesian networks: The combination of knowledge and statistical data, *Machine Learning*, 20(3), 1995, 197–243.

[12] T. Verma and J. Pearl, Equivalence and synthesis of causal models, in P. Bonissone, M. Henriona *et al.* (Eds.) *Proceedings of the 6th Annual Conference on Uncertainty in Artificial Intelligence* (New York: Elsevier, 1991), 255–268.

[13] D. M. Chickering, Learning equivalence classes of Bayesian-network structures, *Journal of Machine Learning Research*, 2, 2002, 445–498.

[14] P. Munteanu and M. Bendou, The EQ framework for learning equivalence classes of Bayesian networks, in *Proceedings of the 2001 IEEE International Conference on Data Mining* (Washington, DC, USA: IEEE Computer Society, 2001), 417–424.

[15] D. M. Chickering, Optimal structure identification with greedy search, *Journal of Machine Learning Research*, 3, 2002, 507–554.

[16] I. Beinlich, H. Suermondt *et al.*, The ALARM monitoring system: A case study with two probabilistic inference techniques for belief networks, in *Proceedings of the Second European Conference on Artificial Intelligence in Medicine* (1989), 247–256.