Carmen: A software tool for probabilistic graphical models

Manuel Arias and Francisco J. Díez
Dept. Inteligencia Artificial. UNED
Juan del Rosal, 16. 28040 Madrid

Abstract
Carmen is a software tool, currently under development, for probabilistic graphical models. The intention of making it useful for different research groups and also reliable for building real-world applications implies that the system must be robust, efficient, scalable, and extensible. For this reason, we have tried to closely adhere to the principles of software engineering, with special emphasis on documenting the source code. This paper describes the main features of Carmen and the results of a preliminary rough analysis of its performance.

1 Introduction
Graphical probabilistic models, such as Bayesian networks and influence diagrams, are a powerful tool for uncertain reasoning in real-world problems (Pearl, 1988). Several tools, both commercial and open source, have been developed in the last years. However, none of the latter packages has been very widely used, perhaps because the lack of adherence to the principles of software engineering has made them difficult to maintain and unreliable for building real-world applications. The aim of building Carmen is to offer the scientific community a public software tool that can be used by different researchers for implementing and testing new algorithms and can also serve as a tool for building and deploying real-world applications. In our opinion, this requires to strictly follow those principles. We have taken profit of our experience in the development of previous tools, mainly the Elvira program (Elvira Consortium, 2002), but we have written the code from scratch with a careful redesign of the whole system.

In this paper we describe the objectives of this project, the principles that we have tried to follow, and an overview of the structure of packages and classes of Carmen. Finally, we offer a rough comparison of the performance of Carmen with that of other well-known tools.

2 Overview

2.1 Non-functional requirements
As a consequence of the above-stated goals, the desired properties of Carmen are as follows:

Open source. This will allow new programmers to easily add new algorithms, and reciprocally, Carmen will benefit of contributions from different individuals and groups.

Multiplatform. It is desirable that Carmen can run on any platform: Linux, Windows, etc.

Robust. This is a requirement for deploying real-world applications.

Efficient. This is also a requisite of a tool for building-real world applications and for testing time-consuming algorithms.

Scalable. The basic algorithms (for instance, deciding whether two nodes are connected) must be able to deal with networks of hundreds or thousands of nodes.

Extensible. Carmen should be able to fit new kinds of probabilistic graphical models, new types of inference, etc.

2.2 Methodology

2.2.1 Analysis and design principles
When designing Carmen, we have tried to comply with the following principles (see (Eckel, 2003; Shalloway and Trott, 2004) for a more detailed explanation):
Consistency. “Use the same criteria everywhere.”

Simplicity. “Make common things easy, and rare things possible.” Simple code is easier to understand, easier to modify, and consequently, more robust.

Substraction. Closely related to the former, it states that “a design is finished when you can not take anything else away”

Design from context. The context must be analyzed first, in order to get a general vision of the problem, before coming down to the details.

Design for change. This is a consequence of the requirement of extensibility. Given that it is impossible to know in advance which new features will be needed in future, the system must be ready to accommodate new functionalities with no modification of the existing code or with only minor modifications.

Encapsulate variation. A class should not be concerned with how other classes are implemented, which subclasses derive from it, how its objects are instantiated, etc.

Strong cohesion and low coupling. Strong cohesion means that the contents of an object must be closely related; for instance, an object should not be responsible for several unrelated tasks. Low coupling means that dependence among classes must be as weak as possible.

Liskov’s principle. A class deriving from a base class should support all of the behavior of its base class.

We have also applied the CVA approach (Commonality and Variability Analysis), which consists in first analyzing the commonalities among the entities of the domain—for instance, among different clustering algorithms—in order to create abstractions—an abstract class for all those algorithms—and then identifying variations and encapsulating them in subclasses (Shalloway and Trott, 2004).

These principles may seem obvious, but all programmers know well that they are often violated in practice.

The application of these principles is made easier by the application of design patterns (Eckel, 2003; Gamma et al., 2005). A pattern is a solution to a problem that appears frequently, and can be applied in several contexts. Several authors have collected and analyzed some patterns used in different situations. According to (Shalloway and Trott, 2004), patterns must be understood as guidelines for the analysis of problems rather than closed prescriptions: “If you understand the principles of patterns and work on a project where a pattern that is unknown to you applies, you are likely to derive it yourself”.

Although we have tried to made a thorough analysis of each part of Carmen, we are aware that it is impossible to have a design analysis that will never change. Additionally, we often realize that the implementation can be made more clear, robust, or efficient. For this reason, we have made extensive use of refactoring (Fowler, 2005). This task has been facilitated by the integrated development environment (IDE) Eclipse,¹ which explicitly includes several refactoring options.

2.2.2 Programming language

The development language for Carmen is Java, mainly in order to allow it to run on different platforms. We have used the latest version, 1.5, which offers new interesting syntactical features, such as template classes (typed collections, such as ArrayList<CertainClass>) and enhanced loops, which significantly improve the iteration on lists.

We have defined a coding style for Carmen that is an extension of Sun’s Code Conventions.²

2.2.3 Documentation

There are two kinds of documentation for the Carmen project: internal and external. Internal documentation consists of comments included in the Java files, and works in combination with the Javadoc utility developed by Sun,³ which generates a set of HTML pages with many cross-references. We devoted a significant effort to provide thorough and clear explanations, but

³http://java.sun.com/j2se/javadoc/.
we have also followed the criterium of not documenting the obvious: in fact, a careful choice of the names of methods and variables is by itself a way of documenting the code.

Sun has proposed a set of tags for documenting the Java code, such as \texttt{@author}, \texttt{@param}, \texttt{@return}, etc. We have extended this collection with new tags especially intended for guaranteeing the correctness of the code. For instance, \texttt{@precondition} indicates a condition that must be fulfilled before invoking a certain method; \texttt{@postcondition} is a logical condition fulfilled after the execution of a certain method; \texttt{@paramCondition} indicates a property that the parameters must satisfy; \texttt{@invariant} refers to a logical property always satisfies by the objects of a certain class; \texttt{@frozen} means that an attribute is set by the constructor and will not be modified afterwards; and \texttt{@sideEffect} refers to secondary effects of the the execution of a method. These tags may be useful for the verification of the source code by human programmers and in the future might be adapted to be used by automatic verification tools.

Additionally, we are generating external documents, such as the current paper, which offer a general overview of Carmen and contain several UML diagrams, mainly of types class, object, sequence, and components.

\textbf{2.2.4 Testing}

Following the recommendation of software engineering, we have built a test suite for each class in Carmen with JUnit.\footnote{www.junit.org.} After introducing a modification in Carmen, we run the battery of tests in order to detect possible bugs in the program.

\textbf{3 Packages and classes in Carmen}

In this section we describe the main packages and classes in which Carmen is structured.

\textbf{3.1 Package graphs}

Mathematically a graph is a pair \(G = (\mathcal{N}, \mathcal{E})\) where \(\mathcal{N}\) is a set of elements called \textit{nodes}, and \(\mathcal{E}\) is a set of \textit{links}. It is usual to define a link as a pair of nodes \((n_1, n_2)\); a pair is said to be undirected if both \((n_1, n_2)\) and \((n_2, n_1)\) are in \(\mathcal{E}\). However, from a computational point of view it is more efficient to define a link as element of \(\mathcal{N} \times \mathcal{N} \times \{\text{directed}, \text{undirected}\}\). A graph is directed if all its links are directed, and undirected if all its links are undirected.

In Carmen we have constructed a package, called graph, aimed at dealing with different kinds of graphs, especially probabilistic graphical models (Bayesian network, influence diagram, Markov network, Markov decision model, chain graph...) and those related with them: cluster tree (or rather, cluster forest), decision tree, probability tree, etc. In the future other kinds of graphs might be implemented based on this package, such as Petri network, neural network, finite automaton, UML diagram, etc.

The main class in this package is Graph, which has two subclasses: LabelledGraph and UnlabelledGraph (see Figure 1). In the former, each arc has the form \((n_1, n_2, \text{boolean}, \text{label})\), where boolean indicates whether the link is directed and the label, which can be any object, allows to have several links between \(n_1\) and \(n_2\) of the same type (directed or undirected), as required, for instance, for finite automata. All the graphs that we have used so far in Carmen are unlabelled.

The neighbors of a node are represented by three adjacency lists: parents, siblings, and children. (The siblings of \(n\) are the nodes connected to \(n\) by undirected links.) This is a way of implicitly representing the links of a graph. However, when a link that has some properties to be stored (for instance, the parameters of a link in a canonical model, such as the noisy OR) the link must be explicitly represented as an object, while in other cases, such as the specification of a probability tree, it would be cumbersome to explicitly represent the links. For his reason, Carmen allows that the links of a graph are either explicit or implicit. This is an example of how we have tried to combine generability and efficiency.
Figure 1: Analysis model for the main classes in the graph package.

### 3.2 Package probNetwork

The package `probNetwork` contains the basic data structures for representing probabilistic graphical models, and its subpackages contain additional classes for reading and saving networks, inference, etc. Some of its basic classes are shown in Figure 2. Every `probNetwork` contains a list of `probNodes` (probabilistic nodes), implemented by three `linkedHashMap`s, for three kinds of nodes: chance, decision, and utility. Each `probNode` has a link to a variable. This allows a variable to make part of different probabilistic models, playing different roles (for instance, it may be associated to a decision node in an influence diagram and to a chance node in a Bayesian network), and also be part of a potential or a cluster in a Hugin forest. Given that a variable may have a significant number of properties, having links to variables from different data structures can save a significant amount of memory.

A `probNode` also contains a list of potentials. In the case of a Bayesian network, each `probNode` has one potential, which represents the conditional probability of its variable given its parents, while a `probNode` in a Markov network may have one, several, or none potentials, each defined on the variable represented by this node. If a potential is defined on several variables, the graph of the network must contain a link for each pair of such variables.

### 3.2.1 Package io

We have built parsers for reading network in the Elvira format, and some classes to write networks in Elvira and Hugin formats.\(^5\) The `io` package also includes a class for writing messages on log files.

### 3.3 Package inference

The package `inference` contains inference methods for computing the posterior probabilities of Bayesian networks, including queries of several variables. Each algorithm is implemented as a class, such as `VariableElimination` and `Clustering-Propagation`; all of them inherit from `Evidence-Propagation`. Each computation of probability is an instance of this class. Its main methods are `individualProbabilities` and `joinProbability`. The former receives a list of `variablesOfInterest` and an `evidenceCase`, which consists of a list of findings, and returns a list of potentials, `ArrayList<Potentials>`. A finding is a pair of a variable and the value taken on by it. Each potential in the list returned gives the posterior probability of one of the variables of interest. The method `joinProbability` receives a list of variables of interest, called `query`, and an `evidenceCase`, and returns a single potential.

#### 3.3.1 Variable elimination

The variable elimination algorithm operates on a Markov network. When eliminating a variable, the algorithm first retrieves all the potentials defined on that variable \(V\), which, because of the properties of `probNetwork`, must be stored in the node \(n\) representing that variable or in its neighbor nodes. Then that node \(n\) and its links are deleted, and the neighbors of \(n\) are married.

i.e., a link is drawn between each pair of them, if necessary. The potentials are multiplied, $V$ is marginalized out, and the resulting potential is stored in one of the neighbors of $n$. The variable to be eliminated is selected by an elimination heuristic (see below). After eliminating all the variables that are neither evidence nor part of the query, all the potentials must be multiplied together, thus returning the join probability of the query variables. The computation of the individual probabilities proceeds by creating a query for each variable of interest invoking joinProbability on each query.

When variable elimination is invoked on a Bayesian network, this is first transformed into a Markov network by removing the directions of links and marrying the parents of each node, and in order to accelerate the propagation of evidence, this algorithm prunes the network by removing barren nodes (i.e., nodes that are not part of the evidence nor ancestors of any evidence node) and also the nodes that are $d$-separated from the variables of interest given the evidence (Pearl, 1988).

### 3.3.2 Clustering algorithms

The class ClusterPropagation is abstract. It currently has one concrete subclass, HuginPropagation, and we will soon add two other concrete subclasses: LazyPropagation and ShenoyShafer-Propagation. Every subclass of ClusterPropagation has a method createForest, which invokes the constructor of (a subclass of) ClusterForest. A cluster forest is a graph consisting of one or several directed trees (see Figure 2); for instance, the method createForest of HuginPropagation invokes the constructor of HuginForest, in which every node represents a clique of variables and the separator with its parent node. The process is illustrated in the UML sequence diagram in Figure 3.

The propagation of evidence operates on the clusterForest in two phases: propagation of evidence towards the root of each tree, and distribution of evidence from the root nodes downwards.

### 3.3.3 Package eliminationHeuristic

Elimination algorithms are used by both variable elimination and clustering methods. The main method in the abstract class EliminationHeuristic is getVariableToDelete, which returns the variable proposed for elimination. Currently Carmen offers three heuristics, implemented as subclasses of EliminationHeuristic: SimpleElimination, which chooses the node having fewer neighbors, CanoMoralElimination (Cano and Moral, 1995), and FileElimination. The latter is not properly a heuristic method, because it reads the list of variables from a file. It is used for forcing Carmen to use a certain elimination order, for instance, the same order

---

6This is an application of the creational design pattern Factory Method (Gamma et al., 2005), because the creation of the cluster forest, although defined in the class ClusterForest, is deferred to its subclasses.
used by other software tool for the sake of comparison.
When Carmen will be released it will offer the possibility of dynamically loading all the elimination heuristics available on a certain directory. This way it will be possible to add new elimination methods without modifying the source code existing so far.\footnote{This is an application of the \textit{Strategy} design pattern (Gamma et al., 2005).}

\subsection*{3.4 Package editSupport}

The package \textit{editSupport} has two purposes: to allow undo/redo operations on \textit{ProbNets} and to inform the listeners, i.e., the objects interested in the changes performed on a \textit{probNet}. Each modification, such as adding or removing a node or a link or modifying a potential, is performed by first creating an instance of \textit{PNEdit} and then passing this object to an instance of \textit{PNESupport} (where PNE stands for “probabilistic network edit”), which informs the listeners, executes the “edit”, and tracks it in a pushdown list to be able to undo it if necessary.

\subsubsection*{3.4.1 Edits}

As mentioned above, operations on probabilistic networks are defined as instances of \textit{PNEdit}, which is a Java interface that derives from Swing’s \textit{UndoableEdit}. It has only one method, \textit{doEdit}, which executes the action by invoking the corresponding method in \textit{ProbNetwork}. Two abstract classes implement the \textit{PNEdit} interface: \textit{SimplePNEdit} and \textit{CompoundPNEdit}. The latter represents a complex change composed of several \textit{simplePNEdit}s. For example, an instance of \textit{CompoundRemoveNodeEdit} will consists of a set of actions, such as one instance of \textit{ReplacePotentialsEdit}, several \textit{RemoveLinkEdit}s, and finally one \textit{SimpleRemoveNodeEdit}, which removes a node isolated from the rest of the network.

The idea of encapsulating a request as an object of type \textit{Edit} is an application of the behavioral pattern \textit{Command} (Gamma et al., 2005).

\subsubsection*{3.4.2 Edit listeners}

There are several objects that may be interested in the changes performed on a \textit{probNet}. An obvious example is a graphical user interface that has to reflect the changes performed on a graph by an inference algorithm, such as the elimination of a node or the reversal of an arc. In this case, the GUI should add itself to the list of listeners of the \textit{PNESupport} instance of that network.

A different use of \textit{PNESupport} listeners is the implementation of restrictions. For instance, the restriction that a Bayesian network can not have cycles may be implemented as an instance
of NoCyclesRestriction, which will veto the addition of a link \( n_1 \rightarrow n_2 \) if the network contains a path \( n_2 \rightarrow n_1 \). In fact, PNESupport performs three steps: First, it sends the message editWillHappen to all the listeners; if some of the listeners vetoes the edit (by throwing an exception), the process stops. Second, it executes the edit, as explained above. And third, sends the message editHappened to all the listeners, just in case they need to update some representation, as it was the case for the GUI in the previous example.

Another example of the use of PNESupport listeners is that we have a method that builds the clique tree (more exactly, the clique forest) at the same time as it eliminates the nodes and triangulates the graph. When the elimination heuristic proposes to eliminate node \( n \), our method marries all its neighbors outside the clique, \( S \). This clique may be partitioned in two sets: \( S \) and \( C \setminus S \), where \( S \) (separator) is the subset of nodes having neighbors outside the clique. Clearly, \( n \in C \setminus S \). If \( C \setminus S \) contains other nodes, these can be removed at the same time as \( n \), but the elimination heuristic must be informed in order to update its internal representation of the graph (which in general it will record, at least, a list of the non-eliminated nodes and the number of neighbors that each node has). For this reason, the elimination heuristic must add itself as a listener of the PNESupport instance.

We have implemented PNESupport as a subclass of Swing’s UndoableEditSupport. The fact that the edition of a probNet is not done by directly invoking the methods of this class, but by means of a mediator class PNESupport resembles some standard patterns, such as Facade, Proxy, and Mediator, although it does not completely match any of them.

4 Performance of Carmen

As a first approach to assessing the performance of Carmen, we have compared it with some well-known software tools, such as Elvira (version 0.16), GeNIE (v. 2.0), Hugin (v. 5.6), and Netica (v. 3.14). We have built some test networks with a double requirement: small number of nodes and states (to make the network tractable by the demo versions of some programs) and big clusters in the clique tree, to make the measurement of time more reliable.

A solution has been the definition of networks containing \( m \times n \) nodes \( X_{i,j} \), \( m \) nodes \( R_i \), and \( n \) nodes \( C_j \). Each node \( R_i \) is a child of all the \( X_{i,j} \)'s (the \( i \)-th row) and each \( C_j \) is a child of all the \( X_{i,j} \)'s (the \( j \)-th column), with \( 0 \leq i \leq m - 1 \) and \( 0 \leq j \leq n - 1 \). The size of the largest clique is roughly \( O(m \times n) \), which means that the complexity of a network grows extremely fast with the number of nodes.

We have made some experiments with a \( 6 \times 6 \) network by introducing evidence on all the \( R \) and \( C \) nodes. Both GeNIE and Netica needed around 3–4 seconds to compile the network and 1.5 seconds to propagate evidence, i.e., to compute the posterior probabilities of all the \( X \) nodes. Hugin needed around 10 seconds to compile the network and the same amount of time to propagate evidence; these times were the same for the four triangulation algorithms offered by that version of Hugin. Carmen, in turn, needed 0.42 seconds to compile the network and 16.0 to propagate evidence. Given that GeNIE and Netica are all implemented in C or C++, it is not surprising that they are around 10 times faster than Carmen, which is implemented in Java.

Elvira ran out of memory for the \( 6 \times 6 \) network; when both tools were compared on a \( 5 \times 5 \) network, Carmen was over 100 times faster than Elvira.

The fact that in our experiments Hugin was slower than GeNIE and Netica might due to the fact that we used an old version of that program, which seems to be based on non-efficient triangulations. In fact, when analyzing Hugin’s log files, we saw that the biggest clique for that network contained 24 variables, while Carmen,

---

8We would have preferred to use the name PNEManager rather than PNESupport, but we have decided to be coherent with Sun’s Swing terminology (http://java.sun.com/products/jfc), in which EditManager refers to a different concept.

which used the heuristic method by Cano and Moral (Cano and Moral, 1995), built a tree whose biggest clique contains only 22 variables. When we forced Carmen to use the same elimination ordering as Hugin and, consequently, to propagate evidence on a tree containing the same cliques, Carmen needed 50.4 seconds, i.e., it was five times slower than Hugin with the same triangulation and it was three times slower than Carmen itself with the Cano-Moral tree.

However, the fact that Carmen is slower when propagating evidence can be compensated by the fact that it is around 7 to 10 times faster when compiling the network. This means that when the barren nodes (i.e., nodes that are not variables of interest nor parents of evidence nodes) can be removed, which in general speeds up significantly the propagation of evidence: the time spent in compiling the pruned network is negligible compared to the time save in the phase of inference.

In any case, we insist, these are only very preliminary results. It is necessary to perform further experiments with different kinds of networks, such as those in the Bayesian Network Repository,\textsuperscript{10} and with other public software tools, such as JavaBayes, SamIam, and others.

5 Conclusions

Carmen is a software tool, currently under development, for probabilistic graphical models. The intention of making it useful for different research groups and also reliable for building real-world applications implies that the system must be robust, efficient, scalable, and extensible. For this reason, we have tried to closely adhere to the principles of software engineering. A particular effort has been devoted to clearly documenting the source code, by means of tools such as Javadoc and UML, not only to facilitate the work of the programmers that will use Carmen, but also as a requisite to make the software robust to future additions and changes. We have also developed an extensive battery of tests (in JUnit) for checking the stability of Carmen under new modifications.

Currently Carmen only does inference on Bayesian networks, but in the near future it will be able to deal with influence diagrams and some Markov decision processes. Future work also includes performing a more detailed comparison of Carmen with other software packages, both commercial and open source. It will take into account the benchmarking networks and the results of the software competition that will take place during the UAI Conference at the MIT (Cambridge, MA) in July 2006. However, the preliminary evaluation of Carmen’s performance seems to indicate that it is efficient enough (when compared to commercial tools) to be used in real-world applications.

The release of an alpha version of Carmen, including the Java source code and extensive documentation, is scheduled for September 2006.

References


E. Gamma, R. Helm, R. Johnson, and J. Vlissides. 2005. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Boston.


\textsuperscript{10}www.cs.huji.ac.il/labs/compbio/Repository.