Readme for ACE v3.0*

Mark Chavira and **Adnan Darwiche**
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095
ace@cs.ucla.edu

August 12, 2015

1 Preliminaries

ACE is a package that compiles a Bayesian network into an Arithmetic Circuit (AC) and then uses the AC to answer multiple queries with respect to the network. ACE's approach to probabilistic inference has four chief advantages,

- ACE makes highly effective use of certain types of parametric structure (especially determinism) in the network in addition to topological structure, to make online inference more efficient.
- ACE pushes much of the work involved in performing repeated inference to an offline phase, which runs just once. Online inference, which can be run any number of times, incurs very little overhead.
- Each time online inference is run, ACE can compute the answers to many queries simultaneously.
- Each time online inference is run, there is very little variance in the time required. Hence, the approach used by ACE may work well in the context of real–time requirements.

The program primarily supports the .hugin/.net network format. However, the program should also work with the following formats: .erg, .ergo, .dne, .dsc, .dsl, .xbif, and .xdsl. ACE runs on Windows, Linux, or OS X and requires Java runtime 1.8 or later. The normal interface to ACE is through the command line, which is what this document describes. However, ACE also includes a lightweight evaluator (including source for porting to other languages) that allows Java programs to answer queries once compilation is complete. Moreover, ACE can be applied to ground instances of relational Bayesian networks directly from within the PRIMULA [9] tool.

2 Installation

The installation incorporates some code produced at the Decision Systems Laboratory at the University of Pittsburgh [8] to read certain network formats. To install:

1. Place the files in a directory on your hard drive.
2. Add the directory to your PATH environment variable.
3. Edit compile (or compile.bat) and evaluate (or evaluate.bat) to allocate about 85% of physical RAM to the program. For example, if you have 2GB of RAM, then change -Xmx512M to -Xmx1700M. *This step is very important, as performance can be significantly worse if insufficient memory is allocated to the program.*

Throughout this document, we refer to evidence files in the .inst format. This format is described in Appendix B.

3 Overview of using the program

From the command line, move into a directory containing a network file. Suppose the network is foo.net. Compile the network as follows:

```
compile foo.net
```

`compile` reports some information to standard out and stores the compiled AC into the files foo.net.lmap and foo.net.ac. To evaluate the AC, execute the following command:

```
evaluate foo.net foo.inst
```

where foo.inst specifies the evidence (see Appendix B for the format of this file). `evaluate` reads the files foo.net, foo.inst, foo.net.lmap, and foo.net.ac, displays some results to standard out, and writes the probability of evidence and a posterior marginal for each network variable to a file named foo.net.marginals. One may compile an AC once and evaluate it many times using different evidence. Moreover, one may answer queries with respect to multiple evidence sets in the same invocation of `evaluate`, which avoids reading files multiple times. To do so, list multiple .inst files instead of one:

```
evaluate foo.net foo1.inst foo2.inst foo3.inst
```

In this case, `evaluate` will compute probability of evidence and posteriors for each evidence file and write all of the information to the marginals file. Times reported to the terminal will be the sum of the times for all evidence sets. If no evidence file is specified, `evaluate` will run as if a single, empty evidence file were specified.

4 Compiling with evidence

An AC is usually compiled without evidence and is therefore capable of answering queries with respect to *any* evidence. However, if one is willing to commit to specific evidence $\mathbf{e}$, then one can compile *with* $\mathbf{e}$. The advantage is that compiling with evidence is usually faster, generates smaller ACs, and improves online

inference time. The disadvantage is that the generated AC is good only for answering queries where the evidence is a superset of **E**. However, there are many applications where such an AC is precisely what is needed. See [2] for examples. To compile with evidence, place the evidence in a file, say foo.inst, and pass it to the compiler using the -e option as follows:

```
compile -e foo.inst foo.net
```

When running `evaluate` with an AC that was compiled with evidence, the effective evidence is the union of the compiled evidence and the evidence passed to `evaluate`.

5 Compilation Methods

Each time `compile` is invoked, it uses one of two algorithms as the basis for compilation. First, if an elimination order can be generated for the network having sufficiently small width, then tabular variable elimination will be used as the basis (this algorithm is similar to the one discussed in [5], but uses tables to represent factors rather than ADDs). If width is large, then logical model counting will be used as the basis. Tabular variable elimination is typically efficient when width is small but cannot handle networks when the width is larger. Logical model counting, on the other hand, incurs more overhead than tabular variable elimination, but can handle many networks having larger treewidth. Both tabular variable elimination and logical model counting produce ACs that exploit local structure, leading to efficient online inference. When logical model counting is invoked, it proceeds by encoding the Bayesian network into CNF, simplifying the CNF, compiling the CNF into d–DNNF, and then extracting the AC from the compiled d–DNNF. A dtree over the CNF clauses drives the compile step.

6 Noisymax

When using logical model counting as a basis for compilation, the approach that Ace uses can be easily adapted to exploit the structure inherent in many non–tabular factor types. Ace includes such support for the noisy–max factor type. Noisy–max is a generalization of the noisy–or model and is supported in the Genie [8] network format (files ending with ".xdsl"). Noisy–max can sometimes be a superior representation for modeling certain types of relationships. Moreover, there is potential to capitalize on the additional structure inherent in a noisy–max factor during inference. When using logical model counting as a basis for compilation, Ace will automatically encode any noisy–max factor in a manner that can be exploited by the compilation algorithm. In some cases, this special encoding leads to very large gains in efficiency, both offline and online, especially when evidence is specified. See [2] for more information about this type of encoding, where it was combined with evidence to allow Ace to deal efficiently with diagnostic networks having treewidth in excess of 500, even when classical evidence techniques could not reduce treewidth.

7 Source Code for an Ace Evaluator

Included with the package is Java source code that implements an Ace evaluator. Once a compilation has been obtained, this evaluator can be used by a Java program to read in the AC and answer many queries with respect to the AC. A key feature of this evaluator is the small amount of code needed to implement it. This code may also be used as a basis for implementing an evaluator in other languages. Documentation

for this evaluator is included in javadocs and in comments found in the source code itself. Source code is located in aceEvalSrc, a compiled version in aceEval.jar, and javadocs in aceEvalDocs.

8 Usage for `compile`

`compile` supports compilation based on tabular variable elimination and on logical model counting. By default, `compile` will utilize tabular variable elimination when possible, and use logical model counting only when necessary. Each time `compile` is invoked, one of four encoding methods is selected and one of three methods of generating a dtree is selected. The examples thus far have used the default encoding and dtree methods. The user may optionally override these defaults. Although encoding and dtree methods are always selected, they will affect the compilation only when logical model counting is used as a basis. The user may optionally specify an evidence file. Following is complete program usage:

```
compile
    [-version]
    [-retainFiles]
    [-encodeOnly]
    [-noEclause]
    [-forceC2d | -forceTabular]
    [-d02 | -sbk05 | -cd05 | -cd06]
    [-dtBnMinfill | -dtClauseMinfill | -dtHypergraph <count>]
    [-e <evidenceFile>]
    <networkFile>
```

Following is a description of each option:

- -version displays a version string and terminates the program ignoring all other arguments.
- -retainFiles retains the files used to compile (e.g., the CNF file); normally these files are deleted. This option is redundant when -noCompile is specified. This option has no effect when tabular compilation is invoked.
- -encodeOnly suppresses compilation (but still encodes), retains files necessary to compile, and prints out the command that would have been executed to compile. -retainFiles is redundant in this case.
- -noEclause in the generated encoding, replaces eclauses with regular clauses. An eclause is a special kind of clause that the c2d compiler (which provides the compilation engine for ACE) understands as meaning *exactly one* instead of *at least one*. Eclauses result in a non–standard CNF, and this option allows ACE–produced CNF encodings to be used with other tools. This option has no effect when tabular compilation is invoked.
- -forceC2d forces compilation using logical model counting techniques, suppressing tabular compilation.
- -forceTabular forces compilation using tabular compilation, suppressing logical model counting compilation.
- -d02 specifies that the program encode the network into CNF using the method defined in [7]. This encoding is the original encoding developed for compiling ACs by compiling CNFs. Some enhancements have been added to the original encoding. This encoding has largely been replaced by -cd06, but remains an option here. This option has no effect when tabular compilation is invoked.

- -sbk05 specifies that the program encode the network into CNF using the method defined in [10]. This encoding is the original encoding proposed for performing Bayesian inference using the model counter Cachet [1]. It typically performs at least as well as -d02. Some enhancements have been added to the original encoding. This option has no effect when tabular compilation is invoked.
- -cd05 specifies that the program encode the network into CNF using the method described in [3]. This encoding was the first to utilize types of local structure other than determinism, most notably equal parameters within a CPT. It typically performs at least as well as -d02 and -sbk05. This encoding has largely been replaced by -cd06, but remains an option here. This option has no effect when tabular compilation is invoked.
- -cd06 specifies that the program encode the network into CNF according to the method defined in [4]. This encoding uses all the advantages of -cd05 while in addition using structured resolution to increase the decomposability of the CNF. It typicall performs at least as well as -d02, -sbk05, and -cd05. -cd06 is the default encoding. This option has no effect when tabular compilation is invoked.
- -dtBnMinfill specifies that the program generate the clause dtree according to the method defined in [3]. A dtree is first generated for the Bayesian network using minfill. Each leaf in this dtree corresponds to one of the network tables T and is replaced with another dtree over the clauses that T generated. This option is the default dtree method. This option involves randomization and so may cause results to differ from one run to the next. This option has no effect when tabular compilation is invoked.
- -dtClauseMinfill specifies that the program generate the clause dtree by applying minfill directly to the generated clauses. This method was used in a few cases in [3]. This option has no effect when tabular compilation is invoked.
- -dtHypergraph <count> specifies that the program generate the clause dtree using hypergraph partitioning. <count> specifies the number of random dtrees to use. A good number for many networks is 25, although this number requires too much time on large networks. In [6], the large size of the networks lead to the use of 3 for <count>. This option involves randomization and so may cause results to differ from one run to the next. This option has no effect when tabular compilation is invoked.
- -e <evidenceFile> specifies that compilation occur with the evidence in <evidenceFile>, which is in the .inst format.
- <networkFile> the .net/.hugin file containing the network.

A detailed description of encoding techniques and clause dtrees is beyond the scope of this document. See papers referenced above for more information. There are four encoding techniques. Usually, -cd06, which is the default, performs best. However, one may wish to try them all to see which causes ACE to compile fastest and which produces the smallest AC. Likewise, there are three dtree generation techniques. Usually, -dtBnMinfill, which is the default, performs best. Again, one may wish to try all of them.

9 Usage for `evaluate`

`evaluate` accepts a network and a list of evidence files in the .inst format, reads the AC from the .lmap and .ac files produced by `compile`, performs inference, and outputs results. Usage for the program follows:

```
evaluate

    [-version]
    <networkFile>
```

```
<evidenceFile>*
```

Following is a description of each option:

- -version displays a version string and terminates the program ignoring all other arguments.
- <networkFile> the .net/.hugin file containing the network.
- <evidenceFile> an evidence file in the .inst format. `evaluate` will union this evidence with any evidence specified during compilation. More than one evidence file may be specified, in which case `evaluate` iterates once for each file. If no evidence files are specified, `evaluate` will compute priors.

10 Troubleshooting

In this section we address common problems encountered when running ACE.

Running out of memory: The `compile` and `evaluate` commands are implemented as scripts, which invoke a Java program with 512MB of memory. If Java runs out of memory while executing `compile` or `evaluate`, you should adjust the amount of memory the script allocates. To do so, find the line in the script that begins "java -Xms8M -Xmx512M" and change the 512 to about 85% of the number megabytes of physical RAM in your machine.

Trouble loading networks: Some of the file formats ACE reads support extensions to standard Bayesian networks. For example, the .net/.hugin format supports influence diagrams. ACE does not support such extensions, and an error will result from attempting to load them. The linux version of Ace utilizes a library that sometimes has trouble reading networks that use Windows–style line terminators. If you are having trouble loading a network, try replacing Windows–style line terminators with Unix–style line terminators.

11 Options used in published results

This section lists the options used for some of the experiments in the publications referenced in this document. Compilations were performed on a machine with 2GB of RAM. Note that -dtHypergraph and -dtBnMinfill involve randomization and so may produce results that differ from one run to the next.

All networks from reference [6]:

- Edit evaluate.bat to allocate 1200 megabytes of memory.
- Compile using: `compile -noTabular -d02 -dtHypergraph 3 foo.net`
- Evaluate using: `evaluate foo.net foo.inst`

Munin1-4 from reference [3]:

- Edit evaluate.bat to allocate 1200 megabytes of memory.

- Compile using: `compile -noTabular -cd05 -dtClauseMinfill foo.net`
- Evaluate using: `evaluate foo.net foo.inst`

Other networks from reference [3]:

- Edit evaluate.bat to allocate 1200 megabytes of memory.
- Compile using: `compile -noTabular -cd05 -dtBnMinfill foo.net`
- Evaluate using: `evaluate foo.net foo.inst`

Networks from reference [4]:

- Edit evaluate.bat to allocate 1200 megabytes of memory.
- Compile using: `compile -noTabular -cd06 -dtBnMinfill foo.net`
- Evaluate using: `evaluate foo.net foo.inst`

A Release Notes

This section lists the major changes that have been incorporated through the various versions of Ace.

A.1 Release 1.0

- Initial release.

A.2 Release 1.0.1

- Added -retainFiles and -noCompile options to `compile`.

A.3 Release 1.1

- Added -resolve encoding option to `compile`.
- Made -resolve the default encoding method.

A.4 Release 1.2

- Adopted a new naming scheme for encoding options to `compile`: -iip changed to -d02 (Darwiche, 2002), -ii changed to -cd05 (Chavira, Darwiche, 2005), and -resolve changed to -cd06 (Chavira, Darwiche, 2006).

- Added -sbk05 (Sang, Beame, Kautz, 2005) encoding option to `compile`.
- Made -dtBnMinfill the default dtree method in `compile`.
- Simplification in `compile` has been reworked and is always run instead of being an option to `compile`. As a result, -s has been removed as an option to compiler.
- Simplification in `compile` is now compatible with -dtBnMinfill, making it compatible with all encodings and all dtree generation methods.
- Added interfaces necessary for using ACE from within the PRIMULA [9] tool.
- Added interfaces necessary for using Ace in the UAI 2006 Inference Evaluation.
- Added -noEclause option to `compile`, which allows generated CNFs to be used with tools that do not support eclauses.
- `compile` now outputs more information.
- Added special support for encoding a noisy–max factor in a way that makes its structure available to be exploited by the inference algorithm. This new encoding can work especially well in the presence of evidence.
- -dtBnMinfill is now capable of producing much higher quality dtrees, which take advantage of available evidence and any evidence that can be learned through simplification.
- The c2d compiler is now packaged with ACE.
- The installation process is much simpler.

A.5 Release 2.0

- Improved the format of the marginals file.
- Added tabular compilation.
- Added source code for the ACE evaluator.

A.6 Release 3.0

v3.0 primarily adds experimental features for special use-cases. It is largely the same as v2.0 for most users.

- Changed -noCompile switch to -encodeOnly and made it imply -noTabular.
- MPE added as an experimental feature that contains no documentation.
- Markov network compilation added as an experimental feature with no documentation.
- Elimination orders are now computed much faster on large networks
- Added UAI competition formats as an experimental feature with no documentation.

```
<?xml version="1.0" encoding="UTF-8"?>
<instantiation date="Jun 4, 2005 7:07:21 AM">
<inst id="A" value="true"/>
<inst id="B" value="false"/>
<inst id="C" value="true"/>
</instantiation>
```

Figure 1: An example .inst file.

B The .inst evidence file format

A .inst file specifies evidence for a Bayesian network using a very simple XML format. Figure 1 displays an example .inst file. The root element is the instantiation. Underneath the root are child inst elements. Each inst element identifies a network variable by its ID and associates a named value with the variable. All of the expected constraints apply: each variable must be a node in the Bayesian network; each variable should appear at most once; etc. The Samiam inference tool (available at http://reasoning.cs.ucla.edu/samiam) provides a convenient way to graphically edit Bayesian networks and .inst files.

References

[1] The cachet model counter. `http://www.cs.rochester.edu/users/faculty/kautz/Cachet/index.htm`.

[2] Mark Chavira, David Allen, and Adnan Darwiche. Exploiting evidence in probabilistic inference. In *Proceedings of the 21st Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 112–119, 2005.

[3] Mark Chavira and Adnan Darwiche. Compiling Bayesian networks with local structure. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1306–1312, 2005.

[4] Mark Chavira and Adnan Darwiche. Encoding CNFs to empower component analysis. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 61–74. Springer Berlin / Heidelberg, Lecture Notes in Computer Science, Volume 4121, 2006.

[5] Mark Chavira and Adnan Darwiche. Compiling Bayesian networks using variable elimination. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 2443–2449, 2007.

[6] Mark Chavira, Adnan Darwiche, and Manfred Jaeger. Compiling relational bayesian networks for exact inference. *International Journal of Approximate Reasoning*, 42:4–20, 2006.

[7] Adnan Darwiche. A logical approach to factoring belief networks. In *Proceedings of KR*, pages 409–420, 2002.

[8] The genie tool for bayesian networks and influence diagrams. `https://dslpitt.org/genie`.

[9] The primula tool for relational bayesian networks. `http://www.cs.aau.dk/~jaeger/Primula`.

[10] Tian Sang, Paul Beame, and Henry Kautz. Solving Bayesian networks by weighted model counting. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05)*, volume 1, pages 475–482. AAAI Press, 2005.