

The C2D Compiler User Manual*

Adnan Darwiche

Computer Science Department
University of California
Los Angeles, CA 90095
darwiche@cs.ucla.edu

July 24, 2005

Abstract

This manual describes the C2D compiler (version 2.20): a software system for converting propositional theories in conjunctive normal form (CNF) into equivalent theories in deterministic, decomposable negation normal form (d-DNNF).

1 Negation Normal Form

Deterministic, Decomposable Negation Normal Form, d-DNNF, is a tractable logical form, which permits some generally intractable logical queries to be computed in time polynomial in the form size [2, 3, 4, 5]. These queries include clausal entailment; model counting; model minimization based on model cardinality; model enumeration; and probabilistic equivalence testing. Most notably, d-DNNF is a strict superset of, and more succinct than, OBDDs [1], which are popular in supporting various AI applications, including diagnosis and planning.

A negation normal form (NNF) is a rooted directed acyclic graph in which each leaf node is labeled with a literal, *true* or *false*, and each internal node is labeled with a conjunction \wedge or disjunction \vee . Figure 1 depicts an example. For any node n in an NNF graph, $vars(n)$ denotes all propositional variables that appear in the subgraph rooted at n , and $\Delta(n)$ denotes the formula represented by n and its descendants. A number of properties can be stated on NNF graphs:

- Decomposability holds when $vars(n_i) \cap vars(n_j) = \emptyset$ for any two children n_i and n_j of an and-node n . The NNF in Figure 1 is decomposable.
- Determinism holds when $\Delta(n_i) \wedge \Delta(n_j)$ is logically inconsistent for any two children n_i and n_j of an or-node n . The NNF in Figure 1 is deterministic.
- Smoothness holds when $vars(n_i) = vars(n_j)$ for any two children n_i and n_j of an or-node n . The NNF in Figure 1 is *not* smooth (the or-node with index 12 violates smoothness).

Satisfiability and clausal entailment can be decided in linear time for decomposable negation normal form (DNNF). Moreover, its models can be enumerated in output polynomial time, and any subset of its variables can be forgotten (existentially quantified) in linear time [2, 6]. Deterministic, decomposable negation normal form (d-DNNF) is even more tractable as far as queries are concerned, as we can count its models given any variable instantiation in polytime [3, 6].

*The C2D compiler is licensed only for non-commercial, research and educational use.

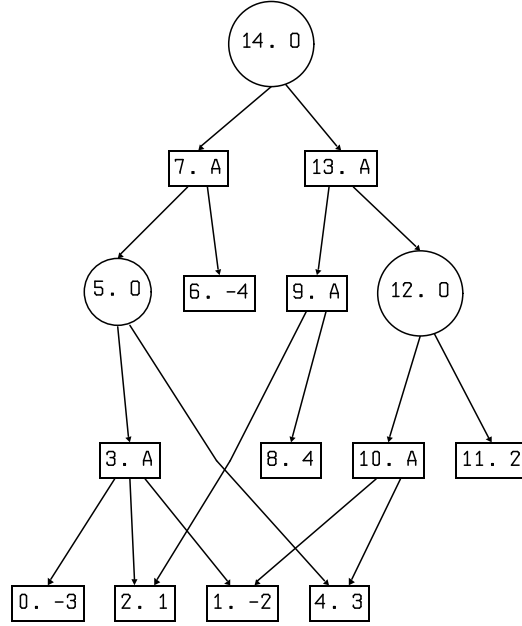


Figure 1: A negation normal form (NNF) graph. Variables are indexed starting at 1 (ranging from 1 to 4). Each NNF node is labeled with an index starting at 0 (ranging from 0 to 14). Or-nodes are labeled with “O”, and-and-nodes are labeled with “A”, and leaf nodes are labeled with literals (i for positive literals and $-i$ for negative literals, where i is a variable index).

2 The compilation process

This section provides an overview of the CNF to d-DNNF compilation process, as described in [5]. The brief description is needed to interpret some of the compiler options discussed in the following section. For a more comprehensive description though, the reader needs to consult [5].

The first step in compiling a CNF is to construct a decomposition tree (dtree), which is a binary tree whose leaves are tagged with the CNF clauses [4, 5]. Figure 2 depicts a dtree for a CNF which contains 4 clauses.¹ Each internal dtree node n has a set of variables associated with it, called the *separator*, which is simply the variables that are shared by the left and right subtrees of node n .

The compiler uses an exhaustive version of the DPLL procedure to decompose every dtree node, by ensuring that the separator variables for that node are either instantiated or no longer shared between the left and right subtrees. When that happens, the compiler will recurse on the left and right subtrees, compiling each independently and combining the results. The compiler also uses sophisticated caching techniques to avoid compiling the same sub-CNF multiple times, whenever possible.

Note that a separator variable v for dtree node n may already be instantiated by the time we reach node n in the recursive decomposition process, for two reasons:

- Variable v may belong to the separator of an ancestor m of n and may therefore have been instantiated when decomposing m ; or
- Variable v may have been set by unit resolution, which is a standard part of DPLL.

¹If the CNF has n clauses, the dtree will contain n leaf nodes, and $n - 1$ internal nodes.

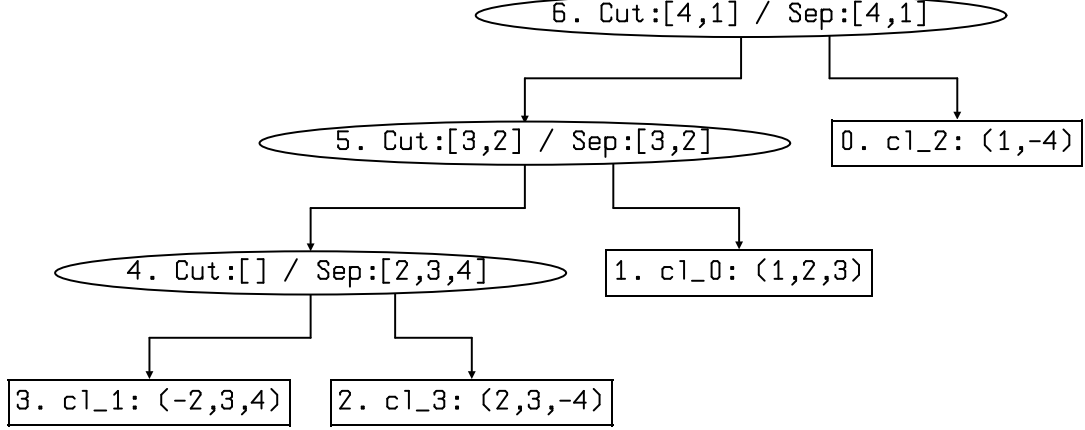


Figure 2: A dtree for a CNF with four clauses. Each dtree node is labeled with an index starting at 0 (ranging from 0 to 6). Leaf nodes are tagged with the CNF clauses, which are also indexed starting at 0 (ranging from 0 to 3). Each internal node is tagged with a *cutset* (*Cut*) and a *separator* (*Sep*). A clause is represented by a list of literals.

Consider the dtree in Figure 2 for an example. Suppose that we start at the root node by setting Variable 4 to true. Using unit resolution and Clause cl_2 , we can conclude that variable 1 must be true. The left and right subtrees of the root are now *decomposed* since all its separator variables are instantiated, so the compiler can now recurse on each independently.

A separator variable may also no longer be shared between the left and right subtrees if other variables are set, as such setting may lead to subsuming clauses. Suppose for example that we set Variable 4 to false in the dtree of Figure 2. Clause cl_2 will then become subsumed and the left and right subtrees of the root are now decomposed. Therefore, there is no need to instantiate Variable 1 in this case.

When arriving at node n , we may want to insist that we instantiate every free variable in the separator for node n , even though it is no longer shared between the left and right subtrees. This level of control is possible through a compiler option to be discussed in Section 3. There are also several methods for constructing a dtree, which can be chosen using the compiler options discussed in Section 3.

A dtree can be generated using one of two methods. The first method converts the CNF into a hypergraph and then uses hypergraph decomposition to recursively decompose the CNF into balanced components. The C2D compiler provides various options for controlling balance, which can be done using fixed balance factors or random ones. The second method for generating a dtree is by inducing one from a variable elimination order. The C2D compiler provides three methods for generating elimination orders, which are then converted into dtrees. We finally note that generating dtrees using hypergraph partitioning is based on using the hmetis program <http://www-users.cs.umn.edu/~karypis/metis/hmetis/>, which uses randomization. Hence, the compiler may generate different dtrees and different NNFs across different runs.

3 Invoking the compiler

The simplest way to invoke the compiler is as follows:

```
c2d -in foo.cnf
```

where *foo.cnf* is a file specifying a CNF (see Appendix A for the syntax of this file). The compiler will then try to compile the CNF, and if successful, will write out the compiled d-DNNF into a file *foo.cnf.nnf* (see Appendix C for the syntax of this file). The compiler can be passed a number of directives, which we describe below:

-in *fname*: specifies the CNF file name. See Appendix A for the syntax of file *fname*. This is a required directive.

-reduce: reduces the compiled NNF by applying some post-processing techniques. Reduction will usually generate a smaller NNF, without changing its logical content, so it should almost always be used.

-dt_method *n*: specifies the method for constructing the dtree:

n = 0: uses hypergraph partitioning for constructing a dtree, with a random balance factor when generating each decomposition.

n = 1: uses hypergraph partitioning for constructing a dtree, with fixed balance factor when generating each decomposition in the dtree (can be set using the **-ubfs** option described below).

n = 2: uses the natural elimination order $(1, 2, 3, \dots, n)$ for constructing the dtree.

n = 3: uses the reverse natural elimination order $(n, n - 1, \dots, 1)$ for constructing the dtree.

n = 4: uses a minfill elimination order for constructing the dtree.

The default is *n* = 0.

-ubfs *s e*, where $s \leq e$ and each of *s* and *e* must be between 1 and 49 inclusive. This directive is only meaningful with **-dt_method 1**, as it specifies the balance factors to be used when generating the dtree using hypergraph partitioning. The directive requests all balance factors ranging from *s* to *e* to be used. For example, if **-ubfs 20 23** is specified, the compiler will generate dtrees using the following balance factors: 20, 21, 22 and 23. The default is *s* = 25, *e* = 25. A lower balance factor will generate a more balanced dtree. Usually balance factors between 20 and 30 will work best.

-dt_count *n*: This is only meaningful with **-dt_method 0** and **-dt_method 1** as it specifies the number of randomized dtrees to be generated. If **-dt_method 0** is specified, a total of *n* dtrees will be generated and the best is chosen. If **-dt_method 1** is specified, then a total of *n* dtrees will be generated for each balance factor—the **-ubfs *s e*** directive can be used to specify which balance factors to try. The default is *n* = 25, but this can take too much time for large CNFs (those with thousands of variables). Hence, a smaller value may need to be chosen if the dtree construction time is too large (at least when compared to compilation time).

-dt_in *fname*: requests that the dtree be loaded from a file, *fname*. See Appendix B for the syntax of file *fname*.

-dt_out: requests that the generated dtree be saved to a file *fname.dtree*, where *fname* is the CNF file name specified by the **-in** directive.

-smooth: smoothes the compiled NNF.

-smooth_all: smoothes the compiled NNF, while ensuring that every variable declared in the CNF appears in the resulting NNF (even irrelevant ones).

-minimize: minimizes the compiled NNF; that is, will generate an NNF whose models are exactly the minimal cardinality models of the given CNF. The cardinality of a model is the number of variables set to true in the model. The resulting NNF may not be equivalent to the given CNF. This option produces an NNF which is also smoothed and includes all variables.

-count: counts the models of given CNF. It is usually more efficient to specify **-smooth_all** when counting, as that would allow using long integers instead of rationals (which would be needed if the NNF is not smoothed).

-in_memory: suppresses the saving of compiled NNF to file. This is typically used with the **-count** and **-smooth_all** options, and can be lot faster. It may exhaust memory though, so should be avoided in that case.

-exist *fname*: will existentially quantify the variables declared in file *fname* out of the resulting NNF. Note that the resulting NNF is only guaranteed to be decomposable in that case (determinism may no longer hold). See Appendix D for the syntax of file *fname*.

-cache_size *n*: specifies the maximum amount of memory *n*, in Megabytes, which will be allocated to the cache. The default is $n = 512$ Megabytes. When the cache fills up, the compiler will start dropping cache entries based on a scheme that factors in the order of insertion into the cache, and the cache hits for different cache entries. If the available memory is not big enough to accommodate *n* Megabytes, then paging will take place and this may slow the compiler considerably. It may be usually better to limit the cache size in order to avoid paging since, in many cases, dropping cache entries would not affect the compiler's performance as much as paging would. One should try to set *n* to the maximum value possible without leading to paging. The default value will work for many problems, so this compiler option should only be used on very demanding CNFs, if one does not have sufficient memory to accommodate the default value.

-nnf_block_size *n*: specifies the size *n*, in Megabytes, of memory blocks used to store the NNF. The default is $n = 64$ Megabytes. This is meant to avoid incremental memory allocation which would be slower.² The compiler will start by allocating *n* Megabytes to store the NNF. When this memory is consumed, it will allocated another *n* Megabytes, and so on. If *n* Megabytes is not available, the compiler will reset *n* to $n/2$ and try again. It will repeat this process until it is able to allocate the necessary memory; otherwise, it will declare an "out of memory" error.

-force *fname*: will ensure that variables declared in *fname* are instantiated in their dtree nodes. That is, if variable *v* in *fname* appears in the separator for dtree node *n*, then *v* will be instantiated when processing node *n* even if *v* is no longer shared between the left and right subtrees of node *n*. See Appendix D for the syntax of file *fname*.

²The operating system will incur some memory overhead which will be needed to support operations like incremental freeing, etc.

-keep_trivial_cls: will keep trivial clauses of the form $x \vee \neg x$ when loading the CNF. Keeping these clauses provides control over the dtrees generated from elimination orders. Other types of trivial clauses will always be removed (those of length > 2 and containing a literal and its negation).

-simplify: simplifies the CNF by running unit and binary resolution, removing satisfied clauses, reducing clauses with resolved literals, and finally writing out a file that contains the simplified CNF. If *fname* is the CNF file name, then *fname_simplified* will be the name of file containing the simplified CNF. This option invokes the compiler as a preprocessor, as it does not lead to any compilation. All other directives (except -in) will be ignored in this case.

-simplify_s: similar to -simplify except that it also removes clauses that are subsumed by other clauses.

-check_ entailment: checks that the generated NNF is indeed decomposable and that it implies every clause in the CNF. This is used for verifying the correctness of generated NNF. The implication test is not compatible with the -exist option.

-visualize: saves files *cnf.vcg*, *nnf.vcg* and *dtree.vcg*, which can be viewed by the graph layout tool at <http://www.aisee.com/>. These files contain visualization of the given CNF, the NNF and dtree constructed for that CNF. If -smooth, -smooth_all, -minimize or -reduce is specified, a corresponding *smoothed_nnf.vcg*, *minimized_nnf.vcg*, or *reduced_nnf.vcg* will be written.

-silent: suppresses the printing of compiler information, such as progress information, CNF and NNF statistics.

4 Interpreting the compiler output

If the -silent option is not specified, the compiler will print various information before it starts compiling, during the compilation process, and after the compilation is completed. Most of the output is self explanatory, yet we provide a sample output next with an explanation of some of the not so obvious elements.

```
c2d -in c1908.cnf -dt_count 50 -smooth_all -count -cache_size 10 -nnf _block_size 50
```

```
c2d compiler version 2.20
```

```
Copyright (c) Automated Reasoning Group, UCLA 2004-2005
```

```
Licensed only for non-commercial, research and educational use
```

```
Loaded cnf: 751 vars 2053 clauses (0 eclauses)
```

```
0 unit clauses, 1612 binary clauses, max clause size: 9
```

```
Generating dtree...(hgr 2053->1776): ***.....
```

```
.....
```

```
Sum Cluster=42, cutset=29, context=36, separator=29 done.
```

```
Max Cluster=40, Cutset=28, Context=35, Separator=28, Height=20
```

```
Compiling...1.3.5.6.8.9.12.13.14.15.16.17.18.19.20.21. [50 MBs]24.28.31.35.
```

```
42.51.55.59.63.67.70.74.76.79.81.84.87.89.92.94.97.done.
```

```
Cache memory: 8.8 MB / Cache count: 80886
```

```

NNF memory: 83.7 MB
Learned clauses: 6897
0.9% of nodes, and 0.9% of edges are dead.
Saving 2961219 nodes and 8604724 edges...done.
Compile Time: 93.815s / Pre-Processing: 16.884s / Post-Processing: 9.093s

Loading 2961219 nnf nodes and 8604724 edges...done.
Smoothing 2961219 nodes, 8604724 edges...2980254 nodes, 8642800 edges / 5.689s
Counting...8589934592 models / 2.864s
Saving 2980254 nnf nodes and 8642800 edges...done.

```

The above output results from compiling a CNF that describes a circuit known as `c1908`. The first part of the output prints statistics on the size of given CNF. The second part prints information about the dtree generation process. In this case, the default generation process is used, which appeals to hypergraph decomposition with randomly chosen balance factors. The number of dtrees generated is 50, which is the default, and the best dtree generated is chosen. Either a “*” or a “.” is printed for each generated dtree, where a “*” indicates the generation of a dtree that improves on the best dtree so far, and a “.” indicates the generation of a dtree that did not survive the selection process. Some quality measures are then printed on the selected dtree.

During the compilation process, the compiler prints progress information in terms of percentages of the completed work. These are educated percentages, but they tend to be quite helpful though in general. The compiler will also print a “cache filled” message when the cache is filled, indicating that cache entries will now start being dropped. This message will be helpful in deciding whether to increase the cache size using the `-cache_size` option. In particular, if the cache is filled early in the compilation process, then one may want to increase the cache size. Finally, the compiler will also indicate when it has allocated a new memory block for storing the NNF by printing “[. MBs]” messages as necessary.

When the compilation process finishes, the compiler prints some statistics, including the number of learned clauses, in addition to memory and timing information. The *preprocessing time* includes CNF loading, dtree generation, and construction of various data structures. The *postprocessing time* includes freeing these data structures, and saving the NNF to file. The *compile time* is purely the time needed to apply the exhaustive DPLL procedure, after having set up all the necessary data structures. Finally, the results of smoothing and counting are shown with their corresponding timings.

Note that the C2D compiler will save the original NNF to file, free all memory, then load the NNF again to perform post processing operations like smoothing. This is meant to make the best out of available memory.

A Syntax of CNF files

A CNF file follows the DIMACS format, which is used by most SAT solvers (with one notable extension discussed at the end of the section). It contains the following header declaration, which must appear before any clause declaration:

$$p \text{ cnf } n \ m$$

where n is the number of CNF variables, and m is the number of CNF clauses.

Variable indices start from 1, hence, $1, 2, \dots, n$ is the set of all variable indices. Each clause in the CNF is specified on a separate line, according to the following format:

$$[-]v_1 [-]v_2 \dots [-]v_k 0$$

where each v_i is a variable index, and $[-]$ refers to an optional $-$ sign. If the $-$ sign does not appear, then we have a positive literal. If the $-$ appears, then we have a negative literal. The 0 is used to terminate a clause.

The CNF file may also contain comment lines that have the following format:

$$c \text{ comment line}$$

Comments lines can appear anywhere in the CNF file.

The following is an example CNF file, which corresponds to the NNF in Figure 1 and dtree in Figure 2:

```
p cnf 4 4
1 2 3 0
-2 3 4 0
1 -4 0
2 3 -4 0
```

A.1 E-clauses

The CNF file may also declare special clauses, known as *E-clauses*. These also have the standard form:

$$[-]v_1 [-]v_2 \dots [-]v_k 0$$

but they have a stronger semantics. Specifically, an E-clause declares that *exactly* one literal in the clause must be true. E-clauses are quite useful when encoding domains with multi-valued variables. Suppose, for example, that we have a planning domain in which a block B can be at one of three possible locations. We would introduce three Boolean variables in this particular case, say, x, y, z , to represent the presence of block B at these different locations. We would then assert the E-clause $x \vee y \vee z$ to indicate that the block must be at exactly one of these locations.

An E-clause can be specified using regular clauses, but it will require $O(k^2)$ such clauses. One of these clauses is

$$[-]v_1 [-]v_2 \dots [-]v_k 0$$

and the others are binary clauses, which ensure the exclusion constraint: if one literal is true, then all other literals in the clause must be false.

E-clauses must be specified last in the CNF file. Their count is declared using the following line:

$$\text{eclauses } e$$

where e is the number of eclauses. This declaration must appear after the header declaration and before any clause declaration.

B Syntax of dtree files

A dtree file is associated with a particular CNF file. A dtree file is specified by a header line that declares the number of nodes in the dtree, followed by a set of lines each specifying a node in the dtree. If the dtree file is associated with a CNF file that contains m clauses, the dtree file must then specify a dtree with $2m - 1$ nodes.³

Nodes are listed in the dtree file in topological order, with children listed before their parents; the last node listed will then be the dtree root. Moreover, nodes are implicitly indexed, with the first node specified having index 0, the second node specified having index 1, and so on.

The header line in the dtree file has the form:

dtree n

which specifies that the dtree has n nodes.

A line which specifies an internal dtree node has the form:

I l r

where l is the left-child index and r is the right-child index of the specified node.

A line which specifies a leaf dtree node has the following form:

L i

where i is the index of CNF clause corresponding to the leaf node. Note that clause indices start at 0, which corresponds to the first clause listed in the corresponding CNF file.

The following dtree file specifies the dtree in Figure 2:

```
dtree 7
L 2
L 0
L 3
L 1
I 3 2
I 4 1
I 5 0
```

C Syntax of NNF files

The NNF file starts with the following declaration:

nnf v e n

where v is the number of nodes in the NNF, e is the number of edges in the NNF, and n is the number of variables over which the NNF is defined. The number of variables n is the same as that declared in the corresponding CNF file, and may be larger than the actual number of variables that appear in the NNF. This happens when some of the variables appearing in the CNF are irrelevant:

³Recall that the leaf nodes in a dtree must be in one-to-one correspondence with the CNF clauses. Hence, the dtree must have m leaves in this case. Moreover, a binary tree with m leaves must have $m - 1$ internal nodes; hence, the $2m - 1$ count.

their values are not constrained by the CNF clauses. Note also that the variables are indexed in the NNF in the same way they are indexed in corresponding CNF.

After the above declaration, each following line in the NNF file will specify an NNF node, which can be an and-node, an or-node, or a leaf-node (corresponding to a literal). The nodes are listed in the file in topological order, with children appearing before their parents in the NNF graph. The nodes are also implicitly indexed, so the first node listed in the file has index 0, the second node listed has index 1 and so on. The last node listed in the NNF file is therefore the root of the NNF and has index $v - 1$.

An and-node is specified in the NNF file as follows:

$$A \ c \ i_1 \ i_2 \ \dots \ i_c$$

where c is the number of children of the and-node and i_1, \dots, i_c are the indices of these children. Note that an and-node with $c = 0$ (no children) represents *true*.

An or-node is specified in the NNF file as follows:

$$O \ j \ c \ i_1 \ i_2 \ \dots \ i_c$$

where c is the number of children of the or-node and i_1, \dots, i_c are the indices of these children. If $j = 0$, then it should be ignored. If $j > 0$, then j represents a variable index, c must be equal to 2, and the two children of the or-node are guaranteed to *conflict* on variable j ; that is, one of them must imply j and the other must imply $-j$. Note that an or-node with $c = 0$ (no children) represents *false*.

A leaf-node is specified in the NNF file as follows:

$$L \ [-]j$$

where $[-]j$ is the literal corresponding to the leaf node. The file will contain at most one leaf node in the file for each literal.

The following NNF file specifies the NNF in Figure 1:

```
nnf 15 17 4
L -3
L -2
L 1
A 3 2 1 0
L 3
O 3 2 4 3
L -4
A 2 6 5
L 4
A 2 2 8
A 2 1 4
L 2
O 2 2 11 10
A 2 12 9
O 4 2 13 7
```

It is recommended to only load NNF files that have been saved by the compiler since the syntax of an NNF file, as expected by the compiler, is quite strict, as far as white space, end of lines, etc.

D Syntax of variables files

A file which specifies a set of variables contains a single line which has the following format:

$$n \ i_1 \ i_2 \ \dots \ i_n$$

where n is the number of variables specified in the file, and i_1, \dots, i_n are the corresponding variable indices. Recall that variables indices start from 1 as dictated by the CNF file format given in Appendix A.

References

- [1] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35:677–691, 1986.
- [2] Adnan Darwiche. Decomposable negation normal form. *Journal of the ACM*, 48(4):608–647, 2001.
- [3] Adnan Darwiche. On the tractability of counting theory models and its application to belief revision and truth maintenance. *Journal of Applied Non-Classical Logics*, 11(1-2):11–34, 2001.
- [4] Adnan Darwiche. A compiler for deterministic, decomposable negation normal form. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI)*, pages 627–634, Menlo Park, California, 2002. AAAI Press.
- [5] Adnan Darwiche. New advances in compiling CNF to decomposable negational normal form. In *Proceedings of European Conference on Artificial Intelligence*, 2004.
- [6] Adnan Darwiche and Pierre Marquis. A knowlege compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.