



THE CLASSIC WORK
EXTENDED AND REFINED

The Art of Computer Programming

VOLUME 4A

Combinatorial Algorithms
Part 1

DONALD E. KNUTH

This page intentionally left blank

For example, $f(0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, x_{25}) = x_{25}$. This function of 25 variables has $Z(f) = 6233$ nodes—which isn't bad, since it represents 5757 words.

Of course we've studied many other ways to represent 5757 words, in Chapter 6. The ZDD approach is no match for binary trees or tries or hash tables, when we merely want to do simple searches. But with ZDDs we can also retrieve data that is only partially specified, or data that is only supposed to match a key approximately; many complex queries can be handled with ease.

Furthermore, we don't need to worry very much about having lots of variables when ZDDs are being used. Instead of working with the 25 variables x_j considered above, we can also represent those five-letter words as a sparse function $F(a_1, \dots, z_1, a_2, \dots, z_2, \dots, a_5, \dots, z_5)$ that has $26 \times 5 = 130$ variables, where variable a_2 (for example) controls whether the second letter is 'a'. To indicate that **crazy** is a word, we make F true when $c_1 = r_2 = a_3 = z_4 = y_5 = 1$ and all other variables are 0. Equivalently, we consider F to be a family consisting of the 5757 subsets $\{w_1, h_2, i_3, c_4, h_5\}$, $\{t_1, h_2, e_3, r_4, e_5\}$, etc. With these 130 variables the ZDD size $Z(F)$ turns out to be only 5020 instead of 6233.

Incidentally, $B(F)$ is 46,189—more than nine times as large as $Z(F)$. But $B(f)/Z(f)$ is only $8870/6233 \approx 1.4$ in the 25-variable case. The ZDD world is different from the BDD world in many ways, in spite of having similar algorithms and a similar theory.

One consequence of this difference is a need for new primitive operations by which complex families of subsets can readily be constructed from elementary families. Notice that the simple subset $\{f_1, u_2, n_3, n_4, y_5\}$ is actually an extremely long-winded Boolean function:

$$\bar{a}_1 \wedge \cdots \wedge \bar{e}_1 \wedge f_1 \wedge \bar{g}_1 \wedge \cdots \wedge \bar{t}_2 \wedge u_2 \wedge \bar{v}_2 \wedge \cdots \wedge \bar{x}_5 \wedge y_5 \wedge \bar{z}_5, \quad (131)$$

a minterm of 130 Boolean variables. Exercise 203 discusses an important *family algebra*, by which that subset is expressed more naturally as ' $f_1 \sqcup u_2 \sqcup n_3 \sqcup n_4 \sqcup y_5$ '. With family algebra we can readily describe and compute many interesting collections of words and word fragments (see exercise 222).

ZDDs to represent simple paths. An important connection between arbitrary directed, acyclic graphs (dags) and a special class of ZDDs is illustrated in Fig. 28. When every source vertex of the dag has out-degree 1 and every sink vertex has in-degree 1, the ZDD for all oriented paths from a source to a sink has essentially the same "shape" as the original dag. The variables in this ZDD are the *arcs* of the dag, in a suitable topological order. (See exercise 224.)

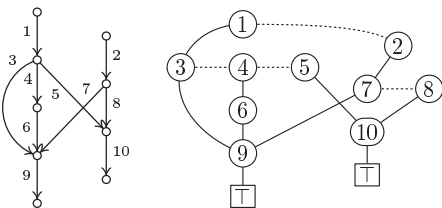


Fig. 28. A dag, and the ZDD for its source-to-sink paths. Arcs of the dag correspond to vertices of the ZDD. All branches to \perp have been omitted from this ZDD diagram in order to show the structural similarities more clearly.

Every Hamiltonian path in this graph must clearly either start or end at Augusta, Maine (ME). Suppose we start in Sacramento, California (CA). Proceeding as above, we can find a ZDD that characterizes all paths from CA to ME; this ZDD turns out to have only 7850 nodes, and it quickly tells us that exactly 437,525,772,584 simple paths from CA to ME are possible. In fact, the generating function by number of edges turns out to be

$$4z^{11} + 124z^{12} + 1539z^{13} + \cdots + 33385461z^{46} + 2707075z^{47}; \tag{134}$$

so the longest such paths are Hamiltonian, and there are exactly 2,707,075 of them. Furthermore, exercise 227 shows how to construct a smaller ZDD, of size 4726, which describes just the Hamiltonian paths from CA to ME.

We could repeat this experiment for each of the states in place of California. (Well, the starting point had better be outside of New England, if we are going to get past New York, which is an articulation point of this graph.) For example, there are 483,194 Hamiltonian paths from NJ to ME. But exercise 228 shows how to construct a *single* ZDD of size 28808 for the family of all Hamiltonian paths from ME to *any* other final state—of which there are 68,656,026. The answer to Bryant’s problem now pops out immediately, via Algorithm B. (The reader may like to try finding a minimum route by hand, before turning to exercise 230 and discovering the absolutely optimum answer.)



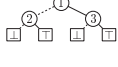
***ZDDs and prime implicants.** Finally, let’s look at an instructive application in which BDDs and ZDDs are both used simultaneously.

According to Theorem 7.1.1Q, every monotone Boolean function f has a unique shortest two-level representation as an OR of ANDs, called its “disjunctive prime form”—the disjunction of all of its prime implicants. The prime implicants correspond to the minimal points where $f(x) = 1$, namely the binary vectors x for which we have $f(x') = 1$ and $x' \subseteq x$ if and only if $x' = x$. If

$$f(x_1, x_2, x_3) = x_1 \vee (x_2 \wedge x_3), \tag{135}$$

for example, the prime implicants of f are x_1 and $x_2 \wedge x_3$, while the minimal solutions are $x_1x_2x_3 = 100$ and 011 . These minimal solutions can also be expressed conveniently as e_1 and $e_2 \sqcup e_3$, using family algebra (see exercise 203).

In general, $x_{i_1} \wedge \cdots \wedge x_{i_s}$ is a prime implicant of a monotone function f if and only if $e_{i_1} \sqcup \cdots \sqcup e_{i_s}$ is a minimal solution of f . Thus we can consider f ’s prime implicants $\text{PI}(f)$ to be its family of minimal solutions. Notice, however, that $x_{i_1} \wedge \cdots \wedge x_{i_s} \subseteq x_{j_1} \wedge \cdots \wedge x_{j_t}$ if and only if $e_{i_1} \sqcup \cdots \sqcup e_{i_s} \supseteq e_{j_1} \sqcup \cdots \sqcup e_{j_t}$; so it’s confusing to say that one prime implicant “contains” another. Instead, we say that the shorter one “absorbs” the longer one.

A curious phenomenon shows up in example (135): The diagram  is not only the BDD for f , it’s also the ZDD for $\text{PI}(f)$! Similarly, Fig. 21 at the beginning of this section illustrates not only the BDD for $\langle x_1x_2x_3 \rangle$ but also the ZDD for $\text{PI}(\langle x_1x_2x_3 \rangle)$. On the other hand, let $g = (x_1 \wedge x_3) \vee x_2$. Then the BDD for g is  but the ZDD for $\text{PI}(g)$ is . What’s going on here?

The key to resolving this mystery lies in the recursive structure on which BDDs and ZDDs are based. Every Boolean function can be represented as

$$f(x_1, \dots, x_n) = (\bar{x}_1? f_0: f_1) = (\bar{x}_1 \wedge f_0) \vee (x_1 \wedge f_1), \quad (136)$$

where f_c is the value of f when x_1 is replaced by c . When f is monotone we also have $f = f_0 \vee (x_1 \wedge f_1)$, because $f_0 \subseteq f_1$. If $f_0 \neq f_1$, the BDD for f is obtained by creating a node $\textcircled{1}$ whose LO and HI branches point to the BDDs for f_0 and f_1 . Similarly, it's not difficult to see that the prime implicants of f are

$$\text{PI}(f) = \text{PI}(f_0) \cup (e_1 \sqcup (\text{PI}(f_1) \setminus \text{PI}(f_0))). \quad (137)$$

(See exercise 253.) This is the recursion that defines the ZDD for $\text{PI}(f)$, when we add the termination conditions for constant functions: The ZDDs for $\text{PI}(0)$ and $\text{PI}(1)$ are $\textcircled{\perp}$ and $\textcircled{\top}$.

Let's say that a Boolean function f is *sweet* if it is monotone and if the ZDD for $\text{PI}(f)$ is exactly the same as the BDD for f . Constant functions are clearly sweet. And nonconstant sweetness is easily characterized:

Theorem S. *A Boolean function that depends on x_1 is sweet if and only if its prime implicants are $P \cup (e_1 \sqcup Q)$, where P and Q are sweet and independent of x_1 , and every member of P is absorbed by some member of Q .*

Proof. See exercise 246. (To say that “ P and Q are sweet” means that they each are families of prime implicants that define a sweet Boolean function.) ■

Corollary S. *The connectedness function of any graph is sweet.*

Proof. The prime implicants of the connectedness function f are the spanning trees of the graph. Every spanning tree that does not include arc x_1 has at least one subforest that will be spanning when arc x_1 is added to it. Furthermore, all subfunctions of f are the connectedness functions of smaller graphs. ■

Thus, for example, the BDD in Fig. 22, which defines all 431 of the connected subgraphs of $P_3 \square P_3$, also is the ZDD that defines all 192 of its spanning trees.

Whether f is sweet or not, we can use (137) to compute the ZDD for $\text{PI}(f)$ whenever f is monotone. When we do this we can actually let the BDD nodes and the ZDD nodes *coexist* in the same big base of data: Two nodes with identical (V, LO, HI) fields might as well appear only once in memory, even though they might have completely different meanings in different contexts. We use one routine to synthesize $f \wedge \bar{g}$ when f and g point to BDDs, and another routine to form $f \setminus g$ when f and g point to ZDDs; no trouble will arise if these routines happen to share nodes, as long as the variables aren't being reordered. (Of course the cache memos must distinguish BDD facts from ZDD facts when we do this.)

For example, exercise 7.1.1–67 defines an interesting class of self-dual functions called the Y functions, and the BDD for Y_{12} (which is a function of 91 variables) has 748,416 nodes. This function has 2,178,889,774 prime implicants; yet $Z(\text{PI}(Y_{12}))$ is only 217,388. (We can find this ZDD with a computational cost of about 13 gigamems and 660 megabytes.)

A brief history. The seeds of binary decision diagrams were implicitly planted by Claude Shannon [*Trans. Amer. Inst. Electrical Engineers* **57** (1938), 713–723], in his illustrations of relay-contact networks. Section 4 of that paper showed that any symmetric Boolean function of n variables has a BDD with at most $\binom{n+1}{2}$ branch nodes. Shannon preferred to work with Boolean algebra; but C. Y. Lee, in *Bell System Tech. J.* **38** (1959), 985–999, pointed out several advantages of what he called “binary-decision programs,” because any n -variable function could be evaluated by executing at most n branch instructions in such a program.

S. Akers coined the name “binary decision diagrams” and pursued the ideas further in *IEEE Trans.* **C-27** (1978), 509–516. He showed how to obtain a BDD from a truth table by working bottom-up, or from algebraic subfunctions by working top-down. He explained how to count the paths from a root to $\boxed{\top}$ or $\boxed{\perp}$, and observed that these paths partition the n -cube into disjoint subcubes.

Meanwhile a very similar model of Boolean computation arose in theoretical studies of automata. For example, A. Cobham [*FOCS* **7** (1966), 78–87] related the minimum sizes of branching programs for a sequence of functions $f_n(x_1, \dots, x_n)$ to the space complexity of nonuniform Turing machines that compute this sequence. More significantly, S. Fortune, J. Hopcroft, and E. M. Schmidt [*Lecture Notes in Comp. Sci.* **62** (1978), 227–240] considered “free B -schemes,” now known as FBDDs, in which no Boolean variable is tested twice on any path (see exercise 35). Among other results, they gave a polynomial-time algorithm to test whether $f = g$, given FBDDs for f and g , provided that at least one of those FBDDs is ordered consistently as in a BDD. The theory of finite-state automata, which has intimate connections to BDD structure, was also being developed; thus several researchers worked on problems that are equivalent to analyzing the size, $B(f)$, for various functions f . (See exercise 261.)

All of this work was conceptual, not implemented in computer programs, although programmers had found good uses for binary tries and Patricia trees — which are similar to BDDs except that they are trees instead of dags (see Section 6.3). But then Randal E. Bryant discovered that binary decision diagrams are significantly important in practice when they are required to be both *reduced* and *ordered*. His introduction to the subject [*IEEE Trans.* **C-35** (1986), 677–691] became for many years the most cited paper in all of computer science, because it revolutionized the data structures used to represent Boolean functions.

In his paper, Bryant pointed out that the BDD for any function is essentially unique under his conventions, and that most of the functions encountered in practice had BDDs of reasonable size. He presented efficient algorithms to synthesize the BDDs for $f \wedge g$ and $f \oplus g$, etc., from the BDDs for f and g . He also showed how to compute the lexicographically least x such that $f(x) = 1$, etc.

Lee, Akers, and Bryant all noted that many functions can profitably co-exist in a BDD base, sharing their common subfunctions. A high-performance “package” for BDD base operations, developed by K. S. Brace, R. L. Rudell, and R. E. Bryant [*ACM/IEEE Design Automation Conf.* **27** (1990), 40–45], has strongly influenced all subsequent implementations of BDD toolkits. Bryant summarized the early uses of BDDs in *Computing Surveys* **24** (1992), 293–318.

Shin-ichi Minato introduced ZDDs in 1993, as noted above, to improve performance in combinatorial work. He gave a retrospective account of early ZDD applications in *Software Tools for Technology Transfer* **3** (2001), 156–170.

The use of Boolean methods in graph theory was pioneered by K. Maghouth [*Comptes Rendus Acad. Sci.* **248** (Paris, 1959), 3522–3523], who showed how to express the maximal independent sets and the minimal dominating sets of any graph or digraph as the prime implicants of a monotone function. Then R. Fortet [*Cahiers du Centre d'Etudes de Recherche Operationelle* **1,4** (1959), 5–36] considered Boolean approaches to a variety of other problems; for example, he introduced the idea of 4-coloring a graph by assigning two Boolean variables to each vertex, as we have done in (73). P. Camion, in that same journal [**2** (1960), 234–289], transformed integer programming problems into equivalent problems in Boolean algebra, hoping to resolve them via techniques of symbolic logic. This work was extended by others, notably P. L. Hammer and S. Rudeanu, whose book *Boolean Methods in Operations Research* (Springer, 1968) summarized the ideas. Unfortunately, however, their approach foundered, because no good techniques for Boolean calculation were available at the time. The proponents of Boolean methods had to wait until the advent of BDDs before the general Boolean programming problem (7) could be resolved, thanks to Algorithm B. The special case of Algorithm B in which all weights are nonnegative was introduced by B. Lin and F. Somenzi [*International Conf. Computer-Aided Design CAD-90* (IEEE, 1990), 88–91]. S. Minato [*Formal Methods in System Design* **10** (1997), 221–242] developed software that automatically converts linear inequalities between integer variables into BDDs that can be manipulated conveniently, somewhat as the researchers of the 1960s had hoped would be possible.

The classic problem of finding a minimum size DNF for a given function also became spectacularly simpler when BDD methods became understood. The latest techniques for that problem are beyond the scope of this book, but Olivier Coudert has given an excellent overview in *Integration* **17** (1994), 97–140.

A fine book by Ingo Wegener, *Branching Programs and Binary Decision Diagrams* (SIAM, 2000), surveys the vast literature of the subject, develops the mathematical foundations carefully, and discusses many ways in which the basic ideas have been generalized and extended.

Caveat. We've seen dozens of examples in which the use of BDDs and/or ZDDs has made it possible to solve a wide variety of combinatorial problems with amazing efficiency, and the exercises below contain dozens of additional examples where such methods shine. But BDD and ZDD structures are by no means a panacea; they're only two of the weapons in our arsenal. They apply chiefly to problems that have more solutions than can readily be examined one by one, problems whose solutions have a local structure that allows our algorithms to deal with only relatively few subproblems at a time. In later sections of *The Art of Computer Programming* we shall be studying additional techniques by which other kinds of combinatorial problems can be tamed.