Outline

- Boolean Circuits
- Boolean Satisfiability (SAT)
- Binary Decision Diagrams (BDD's)

Sources

Enderton: 1.6.


N. J. Cutland. *Computability*. 

Review

Last week

- Propositional Logic: Semantics
- Satisfiability and Tautologies
- Propositional Connectives and Boolean Functions
- Compactness
**Boolean Gates**

Consider an electrical device having $n$ inputs and one output. Assume that to each input we apply a signal that is either $\mathbf{T}$ or $\mathbf{F}$, and that this uniquely determines whether the output is $\mathbf{T}$ or $\mathbf{F}$.

The behavior of such a device is described by a Boolean function:

$$F(X_1, \ldots, X_n) = \text{the output signal given the input signals } X_1, \ldots, X_n.$$  

We call such a device a **Boolean gate**.

The most common Boolean gates are **AND**, **OR**, and **NOT** gates.

---

**Boolean Circuits**

The inputs and outputs of Boolean gates can be connected together to form a **combinational Boolean circuit**.

![Combinational Boolean Circuit](image)

A combinational Boolean circuit corresponds to a **directed acyclic graph** (DAG) whose leaves are inputs and each of whose nodes is labeled with the name of a Boolean gate. One or more of the nodes may be identified as outputs.

A common question with Boolean circuits is whether it is possible to set an output to true (e.g. when the output represents an error signal).

---

**Sharing Sub-Expressions**

The formula $$(D \land (A \land B)) \lor ((A \land B) \land \neg C)$$

This formula highlights an inefficiency in the logic representation as compared with the circuit representation: the formula $A \land B$ appears twice. For larger circuits, this kind of redundancy can result in an exponential blow-up in the size of the corresponding formula.

We can overcome this inefficiency by replacing the redundant sub-expression with a new place-holder variable. We then conjoin a new formula which says that the new variable is equivalent to the replaced expression:

$$((D \land E) \lor (E \land \neg C)) \land (E \leftrightarrow (A \land B))$$

Note that the new formula is not tautologically equivalent to the original formula (why?).

But it is equisatisfiable (i.e. the original formula is satisfiable iff the new formula is satisfiable). Since we are only concerned with the satisfiability of the formula, this is sufficient.
Converting to CNF

This same idea is behind a simple algorithm for converting any propositional formula (or an associated Boolean circuit) into an equisatisfiable formula in conjunctive normal form (CNF) in linear time and space. We will view the formula or circuit as a DAG.

1. Label each non-leaf node of the DAG with a new propositional variable.
2. Construct a conjunction of disjunctive clauses which relate the inputs of that node to its output (the new propositional variable)
3. The conjunction of all of these clauses together with a single clause consisting of the variable for the root node is satisfiable if the original formula is satisfiable.

Converting to CNF: Example

\[(A \land B) \leftrightarrow E\]
\[(((A \land B) \rightarrow E) \land (E \rightarrow (A \land B)))\]
\[((\neg (A \land B) \lor E) \land (\neg E \lor (A \land B))\]
\[((-A \lor \neg B \lor E) \land (-E \lor A) \land (-E \lor B)\]
\[(-A \lor \neg B \lor E) \land (-E \lor A) \land (-E \lor B)\]
\[(-D \lor \neg E \lor G) \land (-G \lor D) \land (-G \lor E)\]
\[((-E \lor \neg F \lor H) \land (-H \lor E) \land (-H \lor F)\]
\[(G \lor H \lor \neg I) \land (I \lor \neg G) \land (I \lor \neg H)\]
\[(I)\]

Standard Representation

Each variable is represented by a positive integer. A negative integer refers to the negation of the variable. Clauses are given as sequences of integers separated by spaces. A 0 terminates the clause.

\[(A' + B' + E)(E' + A)(E' + B)\]
\[(C' + F)(F' + C)\]
\[(D' + E' + G)(G' + D)(G' + E)\]
\[(E' + F' + H)(H' + E)(H' + F)\]
\[(G + H + I')(I + G')(I + H')\]
\[(I)\]
**Boolean Satisfiability (SAT)**

We have seen that there is a natural correspondence between checking Boolean circuits and satisfiability of propositional formulas.

It turns out that Boolean satisfiability or SAT is widely useful for a variety of problems.

SAT was the first problem ever shown to be $\mathcal{NP}$-complete:

S. A. Cook. The Complexity of Theorem Proving Procedures.  

This means that:

- Unless $\mathcal{P} = \mathcal{NP}$, we will never find a polynomial algorithm to solve SAT.
- If we can nonetheless improve algorithms for SAT, there are many other problems that could benefit.

---

**Worst Case Upper Bounds for SAT**

A weakly exponential upper bound is a bound of the form $p(n)c^n$ where $c < 2$ is a constant, $n$ is the number of variables, and $p$ is a polynomial. A $k$-SAT solver solves SAT instances in which no clause has length greater than $k$. Some interesting best-known bounds are as follows.

- General SAT: $p(n)2^n$
- $k$-SAT: $p(n)(2 - \frac{2}{k+1})^n$
- 3-SAT: $p(n)1.481^n$
- 3-SAT formula with exactly one satisfying assignment: $p(n)1.308^n$

---

**SAT in Practice**

How hard is SAT in practice?

A lot of work has gone into building SAT solvers that work well in practice.

Sharad Malik put together a nice history of SAT solvers.

---

**What is the state-of-the-art?**


**SAT 2004 Competition**

- 55 solvers
- 3 benchmark categories
  - Industrial
  - Crafted
  - Random

**Best solvers**

- Industrial: zChaff 2004
- Crafted: March-eq
- Random: AdaptNovelty
Solving General Search Problems with SAT

Modeling

- Define a finite set of possibilities called states.
- Model states using (vectors of) propositional variables.
- Use propositional formulas to describe legal and illegal states.

Solving

- Construct a propositional formula describing the desired state.
- Translate the formula into an equisatisfiable CNF formula.
- If the formula is satisfiable, the satisfying assignment gives the desired state.
- If the formula is not satisfiable, the desired state does not exist.

Example

- Define a finite set of possibilities called states.
  For this problem, each possible coloring is a state. There are $3^{\binom{E}{2}}$ possible states.
- Model states using (vectors of) propositional variables.
  A simple encoding uses two propositional variables for each edge. Since there are 4 possible combinations of values of two variables, this gives us a state space of $4^{\binom{E}{2}}$, which is larger than we need, but keeps the encoding simple.
- Use propositional formulas to describe legal and illegal states.
  Since the color of each edge is modeled with 2 variables, there are 4 possible colors. We can write a set of formulas which disallow the fourth color. For example, if $e_1$ and $e_2$ are the variables for edge $e$, we simply require $\neg(e_1 \land e_2)$.

Example

Recall that a graph consists of a set $V$ of vertices and a set $E$ of edges, where each edge is an unordered pair of distinct vertices.

A complete graph on $n$ vertices is a graph with $|V| = n$ such that $E$ contains all possible pairs of vertices.

How many edges are in a complete graph? \( \frac{n(n-1)}{2} \)

Problems involving graph coloring are important in both theoretical and applied computer science.

Suppose we wish to color each edge of a complete graph without creating any triangles in which all the edges have the same color.

What is the largest complete graph for which this is possible? The answer depends on the number of colors we are allowed to use.

What if you are only allowed one color? Answer: $n = 2$

What if the number of colors is 2? Answer: $n = 5$

What if the number of colors is 3? This is a job for SAT

Example

- Construct a propositional formula describing the desired state.
  The desired state is one in which there are no triangles of the same color. For each triangle made up of edges $e, f, g$, we require:
  \[ \neg((e_1 \leftrightarrow f_1 \leftrightarrow g_1) \land (e_2 \leftrightarrow f_2 \leftrightarrow g_2)) \].
- Translate the formula into an equisatisfiable CNF formula.
  This can be done using the CNF conversion algorithm we described earlier.
- If the formula is satisfiable, the satisfying assignment gives the desired state.
  An actual coloring can be constructed by looking at the values of each variable given by the satisfying assignment.
- If the formula is not satisfiable, the desired state does not exist.
  If the formula can be shown to be unsatisfiable, this is essentially a proof that there is no coloring.

What if the number of colors is 3? Answer: $n = 16$

These and similar questions are studied in Ramsey theory.
Boolean Functions

Recall our definition of Boolean functions.

For $k \geq 0$, a $k$-place Boolean function is a function from $\{F, T\}^k$ to $\{F, T\}$. A Boolean function is anything which is a $k$-place Boolean function for some $k$.

Boolean functions can be represented by propositional formulas. However, as we saw earlier, the representation is not always efficient.

Binary Decision Diagrams are an efficient data structure for representing and performing operations on Boolean functions.

Boolean Function Notation

Assume all functions are $n$-place Boolean functions on variables $x_1, \ldots, x_n$.

Identity: $x_i$

Negation: $\overline{f}$

Conjunction: $f \cdot g$

Disjunction: $f + g$

Definitions

Let $f$ be an $n$-place Boolean function.

A restriction or cofactor of $f$ is formed by replacing one of its arguments by a constant:

$$f|_{x_i=b}(x_1, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, b, x_{i+1}, \ldots, x_n).$$

The Shannon expansion of a function around variable $x_i$ is given by

$$f = x_i \cdot f|_{x_i=1} + \overline{x_i} \cdot f|_{x_i=0}.$$

The function resulting when some argument $x_i$ of function $f$ is replaced by function $g$ is called a composition of $f$ and $g$, and is denoted $f|_{x_i=g}$:

$$f|_{x_i=g}(x_1, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, g(x_1, \ldots, x_n), x_{i+1}, \ldots, x_n).$$

Some functions may not depend on all their arguments. The dependency set of a function $f$, denoted $I_f$, contains those arguments on which the function depends:

$$I_f = \{i \mid f|_{x_i=0} \neq f|_{x_i=1}\}.$$
Binary Decision Trees

A binary decision tree $T$ defines a Boolean function $f_v$ for each vertex $v$ in the tree, defined as follows

- If $v$ is a terminal vertex, then
  - If $\text{value}(v) = 1$, then $f_v = 1$.
  - If $\text{value}(v) = 0$, then $f_v = 0$.
- If $v$ is a nonterminal vertex and $\text{var}(v) = x_i$, then
  
  \[ f_v(x_1, \ldots, x_n) = x_i \cdot f_{\text{low}(v)}(x_1, \ldots, x_n) + \overline{x_i} \cdot f_{\text{high}(v)}(x_1, \ldots, x_n). \]

The Boolean function defined by $T$ is the function $f_{\text{root}(T)}$ where root$(T)$ denotes the root vertex of $T$.

Example

A binary decision tree for the two-bit comparator, given by the formula

\[ f(x_1, x_2, x_3, x_4) = (x_1 \leftrightarrow x_3) \land (x_2 \leftrightarrow x_4), \]

is shown below (left is low, right is high).

Truth Assignments and Binary Decision Trees

To find the value of the function associated with the tree for a given truth assignment, simply traverse the tree from the root as follows.

- If $\text{var}(v)$ is assigned 0, move to low$(v)$.
- If $\text{var}(v)$ is assigned 1, move to high$(v)$.

The value that labels the terminal vertex is the value of the function for this assignment.

Truth Assignments and Binary Decision Trees

What is $f(1, 0, 1, 0)$, where

\[ f(x_1, x_2, x_3, x_4) = (x_1 \leftrightarrow x_3) \land (x_2 \leftrightarrow x_4)? \]

The path leads to a terminal vertex labeled with 1, so $f(1, 0, 1, 0) = 1$. 
A More Concise Representation

Binary decision trees do not provide a very concise representation for Boolean functions.

There is typically a lot of redundancy in such trees.

In the previous example, there are eight subtrees with roots labeled by $x_4$, but only three are distinct.

This observation leads to a natural improvement: merge isomorphic subtrees.

The result is a directed acyclic graph (DAG), called a binary decision diagram (BDD).

Note that the function represented is unchanged.

Ordered Binary Decision Diagrams

An ordered binary decision diagram (OBDD) has the additional property that for some ordering $\prec$ of the variables $x_1, \ldots, x_n$, $\text{var}(v) \prec \text{var}(\text{low}(v))$ and $\text{var}(v) \prec \text{var}(\text{high}(v))$ for each vertex $v$.

In his original paper, Bryant called these function graphs.

Our comparator example is an OBDD which uses the variable ordering: $x_1 \prec x_3 \prec x_2 \prec x_4$.

Reduced Binary Decision Diagrams

The representation can be made even more concise by eliminating vertices $v$ for which $\text{low}(v) = \text{high}(v)$. A BDD which contains no such vertices is called reduced.

Reduced Ordered Binary Decision Diagrams (ROBDD's) have become the data structure of choice for representing Boolean functions, and are now the most common type of BDD.

The primary advantage of ROBDD's is that they are canonical.

Theorem

For any $n$-place Boolean function $f$, there is a unique ROBDD (on $n$ variables) denoting $f$ and any other OBDD denoting $f$ contains more vertices.

Proof

By induction on the size of $f$. 

Example

After merging isomorphic subtrees, the example looks like this.

```
  2
  /\n /  \
2 2
/ \ / \ 
2 2 3 3
1 1 0 0
```

**Canonicity of ROBDD’s**

Given any OBDD, an equivalent ROBDD can be computed in linear time by applying a procedure called *Reduce*.

The fact that ROBDD’s are canonical make several important Boolean function operations trivial:

- Two Boolean functions are equivalent if they have isomorphic ROBDD’s.
- Satisfiability can be determined by simply checking if the ROBDD has a terminal labeled with 1.
- A tautology is represented by the ROBDD with a single vertex labeled 1.

**Example**

The ROBDD for the comparator example is:

![ROBDD Diagram]

From now on, when we refer to BDD’s, we mean ROBDD’s.

Note that the size of a BDD depends very much on the variable ordering.

**Variable Ordering**

In general, finding an optimal ordering is known to be $\mathcal{NP}$-complete.

There are Boolean functions that have exponential size BDD’s for any variable ordering (multiplier).

However, heuristics have been developed for finding a good variable ordering when such an ordering exists.

Heuristics try to group *related* variables together.

For example, when converting a circuit to a BDD, the variables in a sub-circuit are related because together they determine the output of that sub-circuit.

Thus, these variables should usually be grouped together.

This can be done by placing the variables in the order in which they are encountered during a depth-first traversal of the circuit.

**Dynamic Variable Ordering**

A technique called *dynamic reordering* can be useful if no obvious ordering heuristic applies.

When this technique is used, the BDD package internally tries a variety of reorderings and keeps the best one.

Uses various techniques to try to find minimum BDD sizes without getting stuck in a local minimum.
Logical Operations on BDD’s

We begin with the operation of restricting some argument $x_i$ of the Boolean function $f$ to a constant value $b$.

Recall the definition of the restriction or cofactor of $f$:

$$f|_{x_i=b}(x_1, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, b, x_{i+1}, \ldots, x_n).$$

If $f$ is represented by a BDD, the BDD for the restriction $f|_{x_i=b}$ is computed by a depth-first traversal of the BDD.

For any vertex $v$ which has a pointer to a vertex $w$ such that $\text{var}(w) = x_i$, we replace the pointer by low($w$) if $b$ is 0 and high($w$) if $b$ is 1.

After this transformation, Reduce is applied to ensure that the result is canonical.

Logical Operations

Bryant gives a uniform algorithm called Apply for computing all 16 binary operations.

Let $\odot$ be an arbitrary binary operation, and let $f$ and $f'$ be two Boolean functions. To compute $f \odot f'$:

1. If root($f$) and root($f'$) are both terminal vertices, then $f \odot f' = \text{value(root($f$))} \odot \text{value(root($f'$))}$.

2. If var(root($f$)) = var(root($f'$)), then use the Shannon expansion. Let $x = \text{var(root($f$))} = \text{var(root($f'$))}$:

$$f \odot f' = x \cdot (f|_{z=1} \odot f'|_{z=1}) + \overline{x} \cdot (f|_{z=0} \odot f'|_{z=0}).$$

Notice that this effectively breaks the problem into two subproblems which are solved recursively.

The root of the resulting BDD will be a vertex $v$ labeled by $x$.

The first part of this expression computes high($v$), and the second part of the expression computes low($v$).

Logical Operations

All 16 binary propositional connectives can be implemented efficiently on Boolean functions that are represented as BDD’s.

In fact, the complexity of these operations is linear in the size of the argument BDDs.

The key idea for efficient implementation of these operations is the Shannon expansion:

$$f = x_i \cdot f|_{x_i=1} + \overline{x}_i \cdot f|_{x_i=0}.$$
Logical Operations

By using dynamic programming, it is possible to make the algorithm polynomial.

- A hash table is used to record all previously computed subproblems.
- Before any recursive call, the table is checked to see if the subproblem has been solved.
- If it has, the result is obtained from the table; otherwise, the recursive call is performed.
- The result must be reduced to ensure that it is in canonical form.

BDD Extensions

A single DAG can be used to represent a collection of Boolean functions:

- The same variable ordering is used for all of the functions.
- All identical subgraphs are merged.
- Two functions are identical iff they have the same root.
- Checking equivalence can be done in constant time.

Another useful extension adds labels to the edges in the DAG to denote Boolean negation. This makes it unnecessary to use different subgraphs for a formula and its negation.

How does this extension affect canonicity?

BDD’s and Finite Automata

BDD’s can also be viewed as deterministic finite automata.

An $n$-argument Boolean function can be identified with the set of strings in $\{0, 1\}^n$ that evaluate to 1.

This is a finite language. Finite languages are regular. Hence, there is a minimal DFA that accepts the language.

The DFA provides a canonical form for the original Boolean function.

Logical operations on Boolean functions correspond to standard constructions from automata theory.

Representing Finite Relations

BDD’s are extremely useful for obtaining concise representations of relations over finite domains.

If $R$ is an $n$-ary relation over $\{0, 1\}$, then $R$ can be represented by the BDD for its characteristic function:

$$f_R(x_1, \ldots, x_n) = 1 \text{ iff } R(x_1, \ldots, x_n).$$

Representing Relations

If $R$ is an $n$-ary relation over the domain $D$, where $D$ has $2^m$ elements for some $m > 1$.

To represent $R$ as a BDD, we encode elements of $D$ using a bijection

\[ \phi : \{0, 1\}^m \rightarrow D \]

that maps each Boolean vector of length $m$ to an element of $D$.

We construct a new Boolean relation $R'$ of arity $m \times n$ according to the following rule:

\[ R'(\vec{x}_1, \ldots, \vec{x}_n) = R(\phi(\vec{x}_1), \ldots, \phi(\vec{x}_n)), \]

where $\vec{x}_i$ is a vector of $m$ Boolean variables which encodes the variable $x_i$ that takes values in $D$.

$R$ can now be represented as the BDD for the characteristic function $f_{R'}$ of $R'$.

A common application of this technique is to use a BDD to represent a set of elements of $D$ (since sets can be viewed as unary relations).