Lecture Notes on Binary Decision Diagrams

15-122: Principles of Imperative Computation
Frank Pfenning

Lecture 19
October 28, 2010

1 Introduction

In this lecture we revisit the important computational thinking principle *programs-as-data*. We also reinforce the observation that *asymptotic complexity isn't everything*. The particular problem we will be considering is how to autograde functions on integers, something that is actually done in a subsequent course 15-213 *Computer Systems*. The autograder in that class uses the technology we develop in this and the next lecture.

2 Boolean Functions

Binary decision diagrams (BDDs) and their refinements are data structures for representing boolean functions, that is, functions that take booleans as inputs and produce a boolean as output. Let us briefly analyze the structure of boolean functions before we go into representing them. There are two boolean values, in C0 represented as `true` and `false`. In this lecture we will use to the bit values 1 and 0 to represent booleans, for the sake of brevity and following the usual convention in the study of boolean functions.

For a function with n boolean arguments, there are 2^n different possible inputs. The output is a single boolean, where for each input we can independently chose the output to be 0 or 1. This means we have $\underbrace{2 * 2 * \cdots * 2}_{2^n} = 2^{2^n}$ different functions.

Another way to think of a function of n boolean arguments is a function taking a single (unsigned) integer in the range from 0 to $2^n - 1$. Such a

boolean function then represents a set of n-bit integers: a result of 0 means the integer is not in the set, and 1 means the integer is in the set. Since there are 2^k different subsets of a set with k elements, we once again obtain 2^{2^n} different n-ary boolean functions.

Functions from fixed-precision integers to integers can be reduced to boolean functions. For example, a function taking one 32-bit integer as an argument and returning a 32-bit integer can be represented as 32 boolean functions

$$\begin{array}{l} \mathit{bool}\ f_0(\mathit{bool}\ x_0, \ldots, \mathit{bool}\ x_{31}) \\ \ldots \\ \mathit{bool}\ f_{31}(\mathit{bool}\ x_0, \ldots, \mathit{bool}\ x_{31}) \end{array}$$

each of which returns one bit of the output integer.

For the purpose of autograding, we would like to provide a reference implementation as a boolean function (perhaps converted from an integer function) and then check whether the student-supplied function is equal to the reference function. In other words, we have to be able to compare boolean functions for equality. If they don't agree, we would like to be able to give a counterexample.

One way to do this would be to apply both functions to all the possible inputs and compare the outputs. If they agree on all inputs they must be equal, if not we print the input on which they disagree as a counterexample to correctness. But even for a single function from `int` to `bool` this would be infeasible, with about $2^{32} = 4G$ (about 4 billion) different inputs to try. This kind of "brute-force" approach treats "functions as functions" in the sense that we just apply the functions to inputs and observe the outputs.

An alternative is to explicitly represent boolean functions in some form of data structure and compute with them *symbolically*, that is, treat "functions as data". We discuss a succession of such symbolic representations in the next few sections.

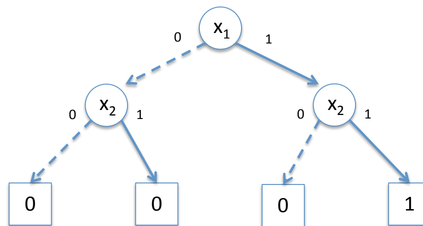
3 Binary Decision Trees

Binary decision trees are very similar to binary tries. Assume we have boolean variables $x_1, \ldots, x_n$ making up the input to a function. At the root node we test one of the variables, say, x_1, and we have two subtrees, one for the case where $x_1 = 0$ and one where $x_1 = 1$. Each of the two subtrees is now testing another variable, each with another two subtrees, and so on. At the leaves we have either 0 or 1, which is the output of the function on the inputs that constitute the path from the root to the leaf.

For example, consider conjunction ("and"). In C0 we could write

```
bool and(bool x1, bool x2) { return x1 && x2; }
```

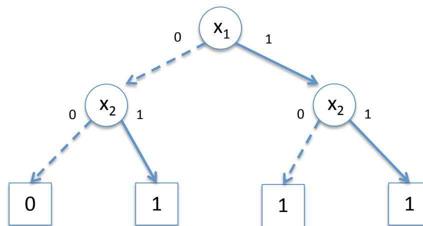
In mathematical notation, we usually write $x_1 \wedge x_2$. In the diagram below for $x_1 \wedge x_2$, we indicate the 0 (left) branch with a dashed line and the 1 (right) branch with a solid line. This convention will make it easier to draw diagrams without a lot of intersecting lines.



As another example, consider disjunction ("or"). In C0 we would write

```
bool or(bool x1, bool x2) { return x1 || x2; }
```

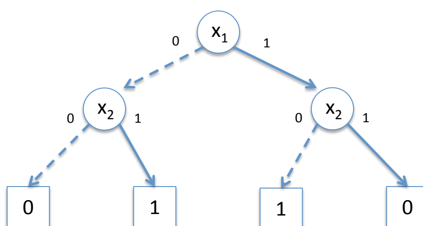
In mathematical notation, we usually write $x_1 \vee x_2$.



Just one more example, exclusive or, which is mathematically often written as $x_1 \oplus x_2$. We cannot write `x1 ^ x2` because the exclusive-or operator in C0 is an operation on arguments of type `int` not `bool`. Nevertheless, it is easy to define.

```
bool xor(bool x1, bool x2) { return (x1 && !x2) || (!x1 && x2); }
```

As a binary decision tree, this would be



Binary decisions trees have some nice properties, but also some less pleasant ones. The biggest problem is their size. A binary decision tree of n variables will have $2^n - 1$ decision nodes, plus 2^n links at the lowest level, pointing to the return values 0 and 1.

A nice property is *canonicity*: if we test variables in a fixed order $x_1, \ldots, x_n$, then the binary decision tree is unique. We could therefore test the equivalence of two boolean functions by comparing their binary decision trees for equality. However, if implemented in a naive way this equality test is exponential in the number of variables, because that is the size of the two trees we have to compare.

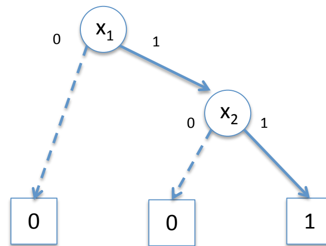
A note on terminology: if the order of the variables we test is fixed, we refer to decision trees as *ordered*. Ordered binary decision trees are isomorphic to binary tries storing booleans at the leaves. Also, we identify the subtrees (and later subdiagrams) of a node labeled by x as the *low* (for $x = 0$) and *high* (for $x = 1$) successors.

We would like to make the representation of boolean functions more compact than decision trees while preserving canonicity to make it easy to compare functions for equality. Recall that, among other operations, comparison of equality is what we need for autograding purposes.

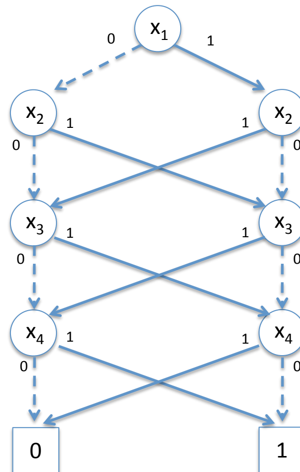
4 Binary Decision Diagrams

Binary decision diagrams (BDDs) differ from binary decision trees in two ways. First, they allow redundant test of boolean variables to be omitted. For example, in the tree for $x_1 \wedge x_2$, both branches in the test of x_2 on the left lead to 0, so there really is no need to test x_2 at all. We can simplify this

to the following BDD.



The second improvement is to allow sharing of identical subtrees. The following is a BDD for the function that returns 1 if the inputs have odd parity (that it, an odd number of 1's) and 0 if the inputs have even parity (that is, an even number of 1's). Rather than 15 nodes and 2 (or 16) leaves, it only has 7 nodes and 2 leaves, exploiting a substantial amount of sharing.



In the worst case, the size of a BDD will still exponential in the number of variables, but in many practically occuring cases it will be much smaller. For example, independently of the number of variables, the constant function returning 0 can be represented by just a single node, as can the constant function returning 1.

5 Reduced Ordered Binary Decision Diagrams

Reduced ordered decision diagrams (ROBDDs) are based on a fixed ordering of the variables and have the additional property of being *reduced*. This means:

Irredundancy: The low and high successors of every node are distinct.

Uniqueness: There are no two distinct nodes testing the same variable with the same successors.

ROBBDs recover the important *canonicity* property: for a fixed variable ordering, each boolean function has a canonical (unique) representation as an ROBDD. This means we can compare boolean functions by constructing their ROBDDs and checking if they are equal.

ROBDDs have at most two leaf nodes, labeled 0 and 1. We sometimes draw them multiple times to avoid complex edges leading to them.

6 Constructing ROBDDs

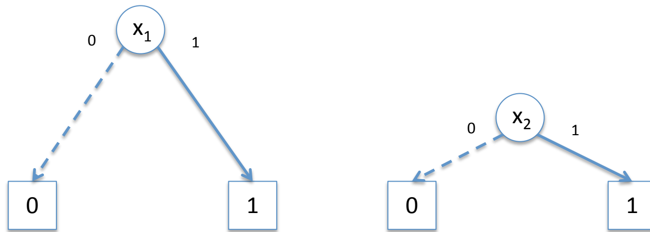
The naive way to construct an ROBDD is to construct a binary decision tree and then incrementally eliminate redundancies and identify identical subtrees. Unfortunately, this would *always* take exponential time, because that's how long it takes to construct the original decision tree.

Another way is to follow the structure of the boolean expression that defines the function. To see how this works we have to see how to implement each of the boolean functions that make up expressions as operations on ROBDDs. As a running example, we use

$$(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$$

To construct an ROBDD for this expression, we start on the left, constructing an ROBDD for $x_1 \vee x_2$. We have already seen the result, but here we decompose it further into the two variables x_1 and x_2. The ROBDDs for

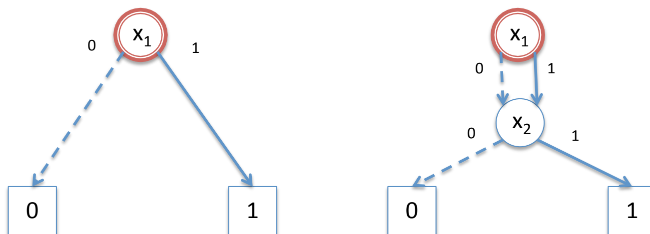
these are the following:



We have already drawn them in a way that is suggestive of the variable ordering (x_1 before x_2).

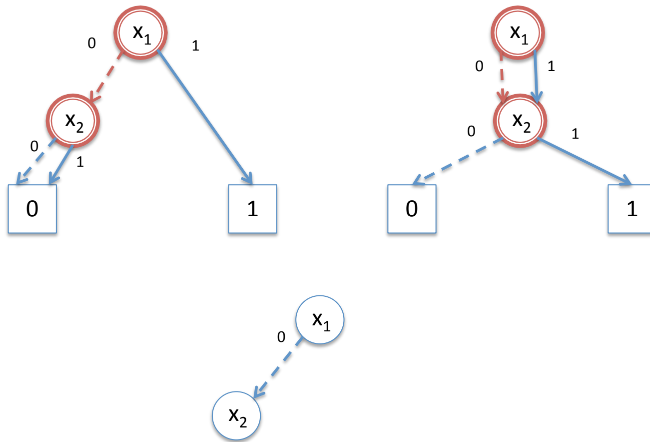
Applying any boolean operation to two ROBDDs with the same variable order means to start at the root and follow parallel paths to the leaves. Once we arrive at the leaves, we apply the given boolean operation to the boolean constants 0 and 1 to form the result for that particular path. We (implicitly) expand variables that are not being tested to see what is would mean to descend to the low or high successor.

In our example we start at x_1 on the left and a hypothetical x_1 on the right.



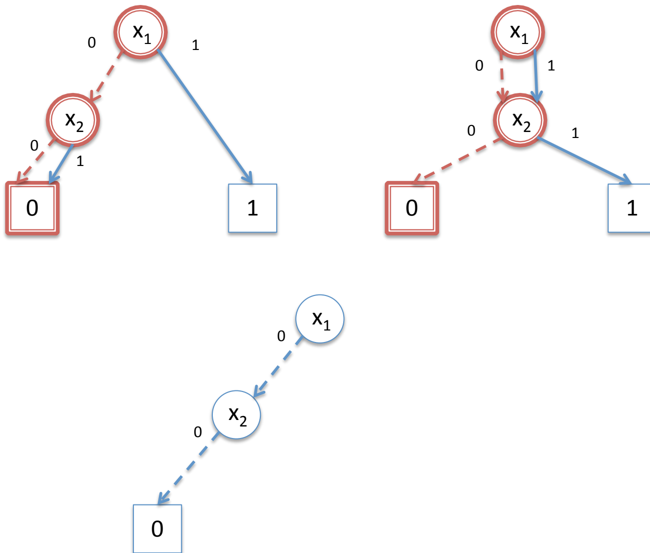
If we follow the low successor on the left and right, we end up at a hypo-

thetical x_2 on the left.

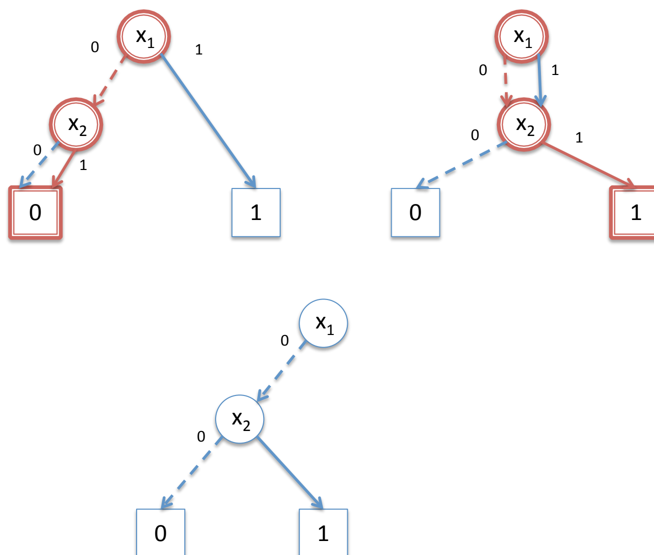


We show the partially constructed ROBDD on the bottom.

At this point, we follow the 0-successor in both given BDDs. Both lead us to 0, so we have to calculate $0 \vee 0 = 0$ and put 0 as the result.

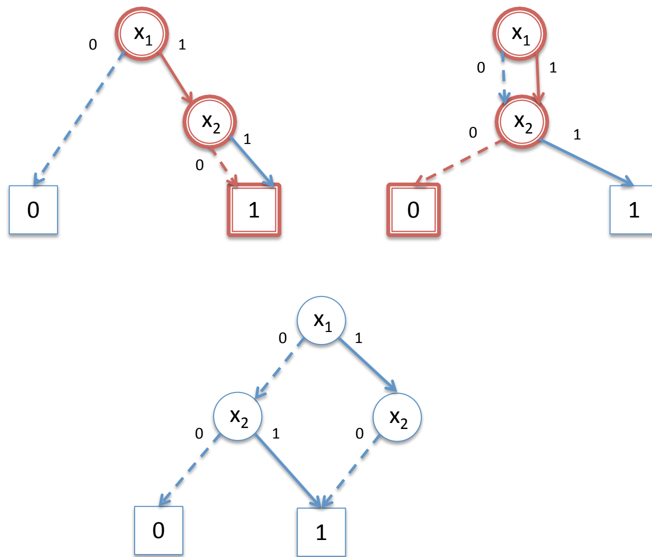


Now we go back up one level to x_2 and follow the 1-successor, which leads us to 0 on the left diagram and 1 in the right diagram. Since $0 \vee 1 = 1$, we output 1 in this case.

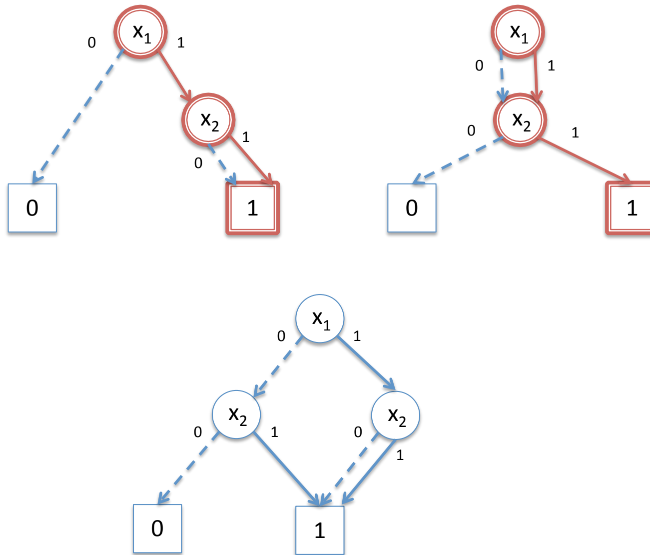


Next we return to x_1 and consider the case that $x_1 = 1$ and then $x_2 = 0$. In this case we compute $1 \vee 0 = 1$ and we have to output 1 for this branch. This time, the node for x_2 in the left diagram is only a conceptual device guiding our algorithm; in an actual implementation we would probably

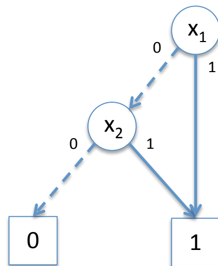
not construct it explicitly.



Finally, we follow the 1-successor from both x_2 nodes and again arrive at 1.



As we now return back up the tree we note both low and high successors of the right occurrence of x_2 are identical, and we eliminate the node, just returning a reference to the constant 1 instead. We obtain the already familiar diagram for $x_1 \vee x_2$. This should be no surprise, since ROBDDs are canonical, it must in fact be the same as we had before.



Here is a summary of the algorithm for applying a boolean operation to two binary decision diagrams. We traverse both diagrams in parallel, always following 0-links or 1-links in both given diagrams. When a variable

is tested in one diagram but not the other, we proceed as if it were present and had identical high and low successors. When we reach the leaves 0 or 1 on both sides, we apply the boolean operation to these constants and return the appropriate constant.

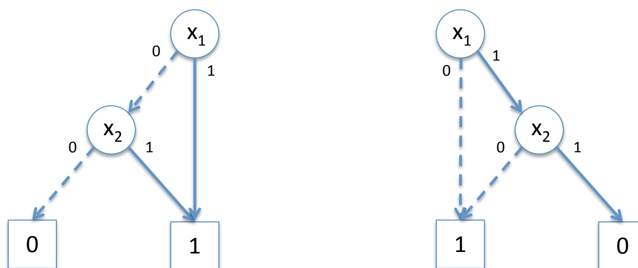
As we return, we avoid redundancy in two ways:

1. If the call on low and high successors return the same node, do not construct a new node but just return the single one we already obtained up the tree. This avoids redundant tests.

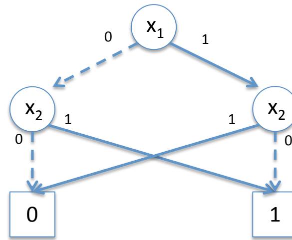
2. If we would be about to construct a node that is already in the resulting diagram somewhere else (that is, has the same variable label and the same successor nodes), do not make a new copy but just return the node we already have.

These two guarantee irreduncancy and canonicity, and the result will be a valid ROBDD.

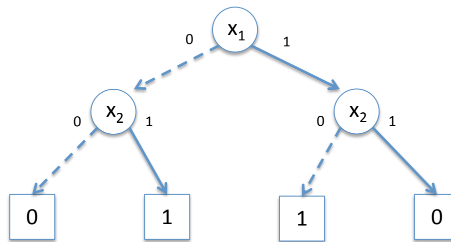
Returning to our original example, we wanted to construct an ROBDD for $(x_1 \lor x_2) \land (\neg x_1 \lor \neg x_2)$. From an ROBDD for some expression e we can construct the negation but just swapping 0 with 1 at the leaves. Doing this, and then applying our algorithm again for a disjunction, yields the following two trees, for $x_1 \lor x_2$ on the left, and $\neg x_1 \lor \neg x_2$ on the right. You are invited to test your understanding of the algorithm for combining ROBDDs by applying to two these to diagrams, construction their conjunction.



If you do this correctly, you should arrive at the following.



Looking back at the binary decision tree for exclusive-or,



we see that they are the same once we have identified the leaf nodes. This is a proof that

$$x_1 \oplus x_2 \equiv ((x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2))$$

that is, the expression on the right implements exclusive-or using disjunction, conjunction, and negation.

7 Validity and Satisfiability

We say that a boolean expression is *valid* if it is equivalent to `true`. That is, no matter which truth values we assign to its variables, the value of the expression is always 1. Or, it is the body of a constant boolean function always returning `true`.

Using the ROBDD technology, it is quite easy to check whether a boolean expression is valid: we construct its ROBDD and see if the result is *equal* to

1. This exploits the canonicity of ROBDDs in an interesting and essential way.

We say that a boolean expression is *satisfiable* if there is some assignment of truth values to its variables such that its value is `true`. For a boolean function, this would mean it there is some input on which the function returns `true`. Again, we can use ROBDDs to decide if a boolean expression is satisfiable. We construct an ROBDD and test whether it is the constant 0. If yes, then it is not satisfiable. Otherwise, there must be some combination of tests on variables that yields 1, and the expression is therefore satisfiable.

In order to construct a satisfying assignment (assuming on exists), we can just traverse the BDD starting at the root, following arbitrary links that do not go to 0. Eventually we must reach 1, having recorded the choices we made on the way as the satisfying assignment.

In order to compute the *number* of satisfying assignments, we can just recursively compute the number of satisfying assignments for each subtree and add them together. Care must be taken that when we come to a missing test (because it is redundant) we think of it as a test with two identical successors. This just means we multiply the number of solutions for the successor node by 2.

These operations will be important when we consider applications validity and satisfiability in the next lecture.

Exercises

Exercise 1 *Construct a 3-variable ROBDD which has maximal size. How many distinct nodes does it have?*

Exercise 2 *Demonstrate by example that ordering we chose for variable can have a significant impact on the size of the ROBDD.*

Exercise 3 *In the autograding application, we can compare ROBDDs for equality in order to be ensured of correctness, because the representation of boolean functions is canonical. Explain how we can obtain a* counterexample *in case the ROBDD constructed from the student's function is* not *identical to the one constructed from the reference implementation.*

Exercise 4 *Many functions on integers have a restricted domain. For example, under a two's-complement representation the integer logarithm would be defined only for positive numbers. For other inputs, the result is irrelevant, so a student function should be considered correct if and only if it agrees with the reference implementation for every integer in a specified domain (rather than on all inputs).*

Explain how to modify the autograder sketched in this lecture to account for domain specifications. Consider a domain as given by a boolean expression which returns 1 *for an element in the domain and* 0 *for an element not in the domain of a function. This is just like specifying a precise precondition.*