CS 3100 – Models of Computation – Fall 2010
Notes for the remainder of CS 3100

1 Notes on Boolean Reasoning

Introduction

Please refer to notes26/minDFA-Diagonalization-BDDs.pdf online. These are excerpts from my book, including discussions on cardinalities, on DFA minimization, the notion of ultimate periodicity, and then the main topic—Boolean functions and BDDs. We will discuss the last topic (Boolean functions and BDDs) now.

Boolean Functions

Boolean Functions can be described via truth-tables. For example, the `and` function is described as follows:

in1	in0	out
0	0	0
0	1	0
1	0	0
1	1	1

A truth-table for an N input Boolean function is guaranteed to have 2^N rows. Therefore, truth-tables are impractical for Boolean functions with many inputs. For example, the truth-table for an equality comparator comparing whether two bytes are equal will be a 16-input function (2 bytes). This truth-table will have 65,536 rows—somewhat like this:

b7	b6	b5	b4	b3	b2	b1	b0	a7	a6	a5	a4	a3	a2	a1	a0	f
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0
...																
0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	1
0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0
...																
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Boolean functions are very important in computer science. Every aspect of computer design and programming involves Boolean functions. Clearly, therefore, we must have *compact representations for Boolean functions.* This is what brings us to the topic of BDDs. Read Section 11.2 of my book along with these notes.

Who invented BDDs?

Binary Decision Diagrams are more properly called *Reduced Ordered Binary Decision Diagrams.* We will simply call them “BDD” for singular and “BDDs” for plural. There wasn’t a single apple fall on someone’s head that

caused BDDs to be invented in an instant. Rather, the ideas behind BDDs were incubated by many over the 1970s and 1980s. It was, however, Prof. Randal Bryant of the Carnegie-Mellon University who popularized BDDs and essentially re-introduced them to computer engineering and computer science. Bryant's contributions were several: (i) showing how canonicity results from fixing a variable order,[1](ii) numerous applications in CAD for VLSI.

Example BDDs for a comparator

We shall use the `DDcal` tool (written by Prof. Somenzi's team at UC Boulder) to build and manipulate BDDs. Here is a simple example of usage of DDcal:

```
. ssh -Y yourname@lab3-1.eng.utah.edu
. Provide your password
. Create a file called f_good_comp.ddc with contents from --begin to --end
  Do not put --begin and --end in the file!!
--begin
var = b7*a7*b6*a6*b5*a5*b4*a4*b3*a3*b2*a2*b1*a1*b0*a0
f = (b7 == a7) * (b6 == a6) * (b5 == a5) * (b4 == a4) * (b3 == a3) * (b2 == a2) \
   * (b1 == a1) * (b0 == a0)
dot f_good_comp.dot
[f]
--end
. Type ~cs3100/DDcal
. Press the Load button
. Select the above file (browse thru the directories)
. Press the Accept button
. You should see a figure similar to the following:
. Press the Save button and save your drawing as a postscript
. Read the README file and the format of the examples within
  the directory ~cs3100/DDcalExs
```

Note that the BDD in Figure 1 manages to represent this function rather nicely and compactly!
Here is how to read this figure:

- Ignore all numbers within ovals (they are internal numbers for nodes).
- Look at level It names a variable say `b7`.
- The solid (blue) edge emanating from the oval is to be read "suppose this variable is a 1." If in doubt, type `[var]` in the Statement window below to see the BDD for the function `var`. We defined this to force a variable order.
- See where that edge leads to. In our case, it leads to a node `a7`.
- Now look at the dotted edge leading out of node `a7`. It says "suppose this variable is a 0."
- At this point we see that the dotted edge leads to the rectangle at the bottom, saying the function is a 0.
- If you considered the solid edge out of `a7`, it would lead to the `b6` node. This means that the status of the function now depends on `b6` and so on.
- So any path from the root (top-most node) to a "1" spells out an input combination when this function is a 1.

[1]Bryant's proof of canonicity did not use the Myhill-Nerode theorem or the view that BDDs are minimal DFAs. This was later discovered. A good exposition of this idea appears in one of Clarke and Minea's early papers.

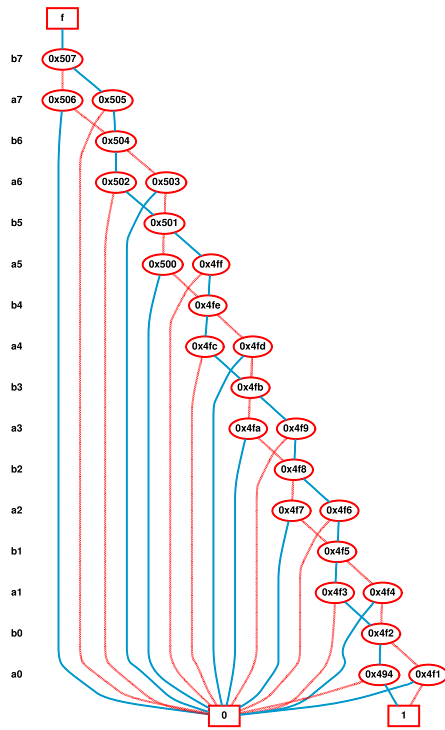


Figure 1: BDD for good variable declaration order

As described in my book's Chapter 11.2 (especially around $L_{interleaved}$ and $L_{noninterleaved}$), these BDDs are representing minimal DFA for languages similar to $L_{interleaved}$ and $L_{noninterleaved}$) described in the book. Notice that we can "confuse" the DFA by presenting all the `a` bits upfront and then presenting the `b` bits. This gives us an exponentially big DFA. A BDD for the comparator with this poor variable order is now shown.

The bad variable order

OK it is possible to have a huge BDD if you are not careful. If you change the `var` declaration to what is below, the BED also becomes gigantic. The idea is to tweak the `var` declaration so as to reach the quickest decisions.

```
. Exit out of DDcal (to "forget" previous variable order)
. Create a file f_bad_comp.ddc
var= b7*b6*b5*b4*b3*b2*b1*b0*a7*a6*a5*a4*a3*a2*a1*a0
. All other lines are as before
. Re-invoke DDcal and load f_bad_comp.ddc
. You will get a huge BDD now. If you hit "reorder" -- Presto! it will
  reorder the variables to reduce the size of the BDD dramatically!
```

You can read about the 4-bit case from my book.

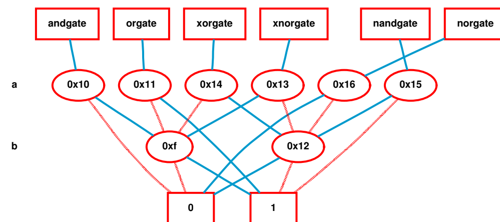


Figure 2: Six common gates

BDDs for 2-input gates

OK so now that you have a feel for BDDs, let's look at the BDDs for many of the Boolean functions we have studied. Hopefully the BED script plus the pictures will tell you the story. See Figure 2.

```
var = a*b
andgate = a*b
orgate = a+b
xorgate = (a == b)'
xnorgate = (a == b)
nandgate = (a * b)'
norgate = (a + b)'
[andgate orgate xorgate xnorgate nandgate norgate]
```

Uses of BDDs for Functional Equivalence Verification

We will use BDDs for functional equivalence verification. Suppose someone claims that four NANDs connected in this clever way realizes the XOR function:

```
var=a*b
xor = (a*b' + a'*b)
p = (a * b)'
q = (a * p)'
r = (b * p)'
f = (q * r)'
check = (f == xor)
[f xor check]
```

How do we verify this claim? Don't use "exhaustive testing;" instead, use BDDs!

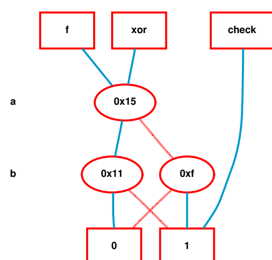


Figure 3: "Curious XOR" and a BDD-based Check

Since the BDD in Figure 3 is that of XOR, we can conclude that this is an XOR gate!

If we had two big fat circuits and wanted to compare them, see whether we get the "1" BDD when we assert the equality between them (Figure 3). A "1" BDD is the *true* function. If the circuits aren't the same, we will get a non-1 BDD.

2 NP-Completeness

NP-completeness

Let us define the $O()$ complexity of DTM algorithms. Given a DTM M running some algorithm A on input x of length N, if M's run takes $O(f(N))$ steps, we can say that algorithm A has time complexity $O(f(N))$. Space complexity can be similarly measured.

Turing machines are *robust* models for algorithms. They model the complexity of algorithms within a polynomial factor with respect to all other machine types! That is, modern digital computers, Turing machines, and all deterministic TM variants are in an equivalence class of complexity in the sense that a deterministic algorithm expressed in any one machine form can be mechanically translated into a deterministic algorithm in another form with only a polynomial factor of difference in the $O()$ complexity. This is fine for our purposes as elaborated now.

In the study of asymptotic worst-case complexity, and especially in our pursuit of the existence of polynomial time algorithms for problems, a polynomial factor difference is perfectly acceptable. It is only when the complexity jumps by an exponential factor that we worry. Luckily, all TM variants are within a polynomial factor of each other in time complexity.

Non-deterministic Algorithms

Non-deterministic algorithms are algorithms slated for "execution" on NDTMs. This approach may seem bizzare, but allows us to obtain many uncanny insights. Define the non-deterministic complexity of an algorithm by the number of execution steps taken by the most favorable ("shortest") run of an NDTM. Examples:

- An ND algorithm for Boolean satisfiability: Write down a satisfying assignment for the N variables in question, each time selecting either a 0 or a 1. Check if this selection satisfies the formula.

- An ND algorithm for the existence of a k-clique in a graph: Write down k node instances. Check if they are connected pairwise.

If an ND algorithm can run in *non-deterministic polynomial time* (NP-time), we say that the problem (language) is in the family NP. Note that NP-time is not the same as P-time. In the former, we assume that "guesses" (perfect non-deterministic execution path selection) comes for free, and ascribe a cost to this path.

P versus NP

While the existence of ND algorithms looks like "cheating," in the study of "hard" problems, one has to be lucky to find even ND algorithms! However, for a large class of hard problems, we know that ND algorithms exist. However, we do not know whether such ND algorithms also have equivalent DTM algorithms that run in polynomial time. This is the famous P versus NP question.

Settling the P versus NP question

This question is still open. All we can do is this: identify an equivalence class of problems in NP such that solving one of them using a P-time algorithm immediately gives us a P-time algorithm for all other problems in this equivalence class. Such an equivalence class has been identified, and is called NP-complete. This includes thousands of important real-world problems such as bin-packing, the Traveling Salesperson Problem, Clique identification, register coloring in compilers, etc.

An NP-complete problem x can be defined in two ways:

- Problem x is in NP (has an NP-time algorithm)
- Every other problem in NP has a P-time mapping reduction to x

Thus, if x can be solved in P-time, the whole NP class can be solved in P-time.

A more convenient definition for NPC is:

- Problem x is in NP
- For some problem y in NPC, there is a P-time mapping reduction from y to x.

3CNF Sat

The first NPC problem identified was 3CNF satisfiability. The problem instance is as follows: The given Boolean formula is a conjunctive normal form (CNF) formula with three literals per clause. An example is

```
(a+b+c)*
(a+b'+c')*
(a'+b'+c)*
(a+b'+c)*
```

Another example is

```
(a+b+c)*
(a+b'+a')*
(p'+b'+r)*
(u+t'+a)*
```

Each line above is called a *clause*.

Note that in any unsat 3CNF sat formula, for every assignment, there must be one clause that is TTT (all true) and another that is FFF (all false).

This fact will hopefully allow you to precisely understand the proposed mapping reduction reduction from 3SAT to k-Clique.

NP-completeness and BDDs

Hopefully the study of Boolean reasoning through BDDs helped you understand Boolean satisfiability a little bit. If it were possible to build (in polynomial time) polynomially sized BDDs for every Boolean formula, then 3SAT would be P-time solvable. As things stand, 3SAT is not polynomially solvable.

3 Solving Puzzles using BDDs

Let us gain an appreciation of how BDDs help us manipulate sentences in mathematical logic. Here is a puzzle by Lewis Carroll. **You are required to model and solve it using DDcal, as part of Assignment 9a.** There will be another part (9b) that will be given soon.
From the premises

1. Babies are illogical;
2. Nobody is despised who can manage a crocodile;
3. Illogical persons are despised.

Conclude that *Babies cannot manage crocodiles.*

Here is another puzzle by Lewis Carroll. I've shown below how to encode and solve it. From the premises

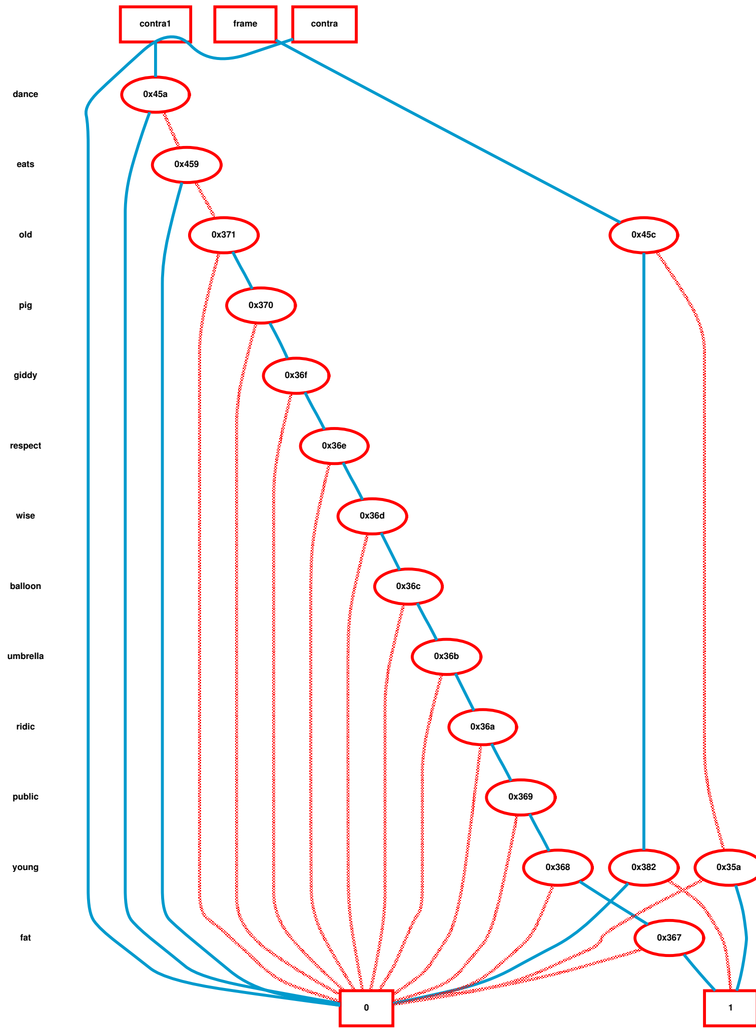
1. All who neither dance on tight ropes nor eat penny-buns are old.
2. Pigs, that are liable to giddiness, are treated with respect.
3. A wise balloonist takes an umbrella with him.
4. No one ought to lunch in public who looks ridiculous and eats penny-buns.
5. Young creatures, who go up in balloons, are liable to giddiness.
6. Fat creatures, who look ridiculous, may lunch in public, provided that they do not dance on tight ropes.
7. No wise creatures dance on tight ropes, if liable to giddiness.
8. A pig looks ridiculous carrying an umbrella.
9. All who do not dance on tight ropes and who are treated with respect are fat.

Show that *no wise young pigs go up in balloons.*

```
# A puzzle by Lewis Carroll :
#
#  From the premises
#
#  All who neither dance on tight ropes nor eat penny-buns are old.
#  Pigs, that are liable to giddiness, are treated with respect.
#  A wise balloonist takes an umbrella with him.
#  No one ought to lunch in public who looks ridiculous and eats penny-buns.
#  Young creatures, who go up in balloons, are liable to giddiness.
#  Fat creatures, who look ridiculous, may lunch in
#      public, provided that they do not dance on tight ropes.
#  No wise creatures dance on tight ropes, if liable to giddiness.
#  A pig looks ridiculous carrying an umbrella.
#  All who do not dance on tight ropes and who are treated
#      with respect are fat.
#
# Show that "no wise young pigs go up in balloons."

#  First specify the desired variable ordering. DDcal can later reorder
var = dance*eats*old*pig*giddy*respect*wise*balloon*umbrella*ridic*\
      public*young*fat

#  All who neither dance on tight ropes nor eat penny-buns are old.
A1 = (dance' * eats') => old

#  Pigs, that are liable to giddiness, are treated with respect.
A2 = (pig * giddy) => respect

#  A wise balloonist takes an umbrella with him.
A3 = (wise * balloon) => umbrella

#  No one ought to lunch in public who looks ridiculous and eats penny-buns.
A4 = (ridic * eats) => public'

#  Young creatures, who go up in balloons, are liable to giddiness.
A5 = (young * balloon) => giddy

#  Fat creatures, who look ridiculous, may lunch in
#      public, provided that they do not dance on tight ropes.
A6 = (fat * ridic * dance') => public

#  No wise creatures dance on tight ropes, if liable to giddiness.
A7 = (wise * giddy) => dance'

#  A pig looks ridiculous carrying an umbrella.
A8 = (pig * umbrella) => ridic

#  All who do not dance on tight ropes and who are treated
#      with respect are fat.
A9 = (dance' * respect) => fat

# Show that "no wise young pigs go up in balloons."
proofGoal = wise * young * pig => balloon'

# Negate proof-goal and add it in
contra1 = A1 * A2 * A3 * A4 * A5 * A6 * A7 * A8 * A9 * \
           proofGoal'
```

```
# Unfortunately contra1 did not emerge to be false. Why? Forgot this!
frame = (young == old')

# Add this frame condition. Now we get "false". Thus proved by contradiction.

contra = contra1 * frame
[contra1 frame contra]
```

We pretty much encoded each English assertion and put it in. This is conjoined with the negated `proofGoal`. Notice how the "frame axiom" is needed to obtain a correct proof by contradiction (the tool "does not know that" old means not young!) This is why `contra1` is conjoined with `frame`. Figure 4 shows that we indeed have a successful proof by contradiction—showing that `proofGoal` is proven!


contra1
frame
contra
dance
eats
old
pig
giddy
respect
wise
balloon
umbrella
ridic
public
young
fat
0x45a
0x459
0x371
0x370
0x36f
0x36e
0x36d
0x36c
0x36b
0x36a
0x369
0x368
0x45c
0x382
0x35a
0x367
0
1