Techniques for Logical Deduction

Approaches to Deduction

Asterisks indicate primary features of approach

Truth Tables

* easy to understand
exhaustive listing of all cases (doesn't work for infinite domains)
* brute force, little thinking
exponential (2^n) in number of variables

Natural Deduction

relatively easy to follow, hard to understand
flexible input form
* many one-directional inference rules
requires insight and cleverness
stored intermediate facts grow exponentially

Resolution

hard to understand
standardized input (CNF), grows exponentially
* single one-directional inference rule (good for algorithms)
stored intermediate facts grow exponentially

Algebraic Logic

easy to understand
flexible input form
* few bidirectional simplification rules
requires some insight
stored intermediate facts not used

Matrix Logic

relatively easy to understand
* every object is an operator
* standard matrix addition and multiplication
brute force
exponential (effectively the same as truth tables)

Boundary Logic (void-based reasoning)

hard to understand
flexible input form (any logical form)
* few easy to apply rules
requires little thinking
* facts shrink instead of growing

Logic Gates

When numbers are expressed in binary, addition can be expressed in terms of logical gates.

	32	16	8	4	2	1	
							powers of 2
13			·	·		·	summand
+22		·		·	·		summand
=35	·				·	·	sum

Rules of combination:

A	B	sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1
		XOR	AND

N-variable Boolean Functions

Different Boolean functions have different binary values associated with each combination of values of variables. For N variables, there are 2^N combinations of values (all the rows in a truth table). For the 2^N combinations, there are two ways to assign a truth value, resulting in (2^2^N) Boolean functions of N variables.

Number of variables	*Number of functions*	
0	2	2^2^0
1	4	2^2^1
2	16	2^2^2
3	256	2^2^3
4	65536	2^2^4
5	very large	2^2^5

These functions can be arranged at the nodes of an N-dimensional hypercube, which is also a *binary, complemented, distributed lattice*. Here are the listing for 0,1, and 2 variables:

0 variables

	functions	
	0	1
function names	True	False

1 variable

a	*functions*			
0	0	0	1	1
1	0	1	0	1
function names	False	a	~a	True

2 variables

a	*b*	*functions*															
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
names		F	&	nif	a	nfi	b	xor	v	nor	=	~b	fi	~a	if	nand	T

`nif = ~(a -> b)`   `fi = (b -> a)`   `nfi = ~(b -> a)`

Some of the 16 two-variable functions have common names, some have technical names from Electrical Engineering, and others are not named. Note that the above columns for the functions are the truth tables for that function.

Normal Forms

Any Boolean function can be expressed as a *conjunction of disjunctive clauses:*

CNF: Conjugate Normal Form

E.g.: `(A v B v ~C) & (A v C) & (B v ~C v D)`

Groupings of forms joined by OR are called **clauses**. Clauses are joined by AND.

CNF has minimal depth (2 layers deep) and
a maximal number of variable references (up to 2^n)

CNF is a normal form in that a specific Boolean function will have a single CNF form.

Any Boolean function can be expressed as a *nesting of implications:*

INF: Implicate Normal Form

E.g.: `((((A -> ~B) -> C) -> (D -> E)) -> ~G)`

INF has maximal nesting, or depth and a minimal number of variable references.

There are many INF forms for a given Boolean function (so it's not truly a normal form)

It is always possible to express a Boolean function
with only two occurrences of a selected variable.

Minimal Bases

It is possible to express many the Boolean functions in terms of other functions. The ***basis set*** is the set of functions which are taken to be non-decomposable.

Common Basis Sets:	`{and, or, not, T}`	
	`{not, if, T}`	
Small Basis Sets	`{nor, F}`	
	`{nand, T}`	
Minimal Basis Set:	`{nor}`	(this requires an innovative notation)

Resolution

Resolution expresses Boolean functions as sets of literals. This is a different way to express CNF. The disjunctive forms in each clause form a set with implicit disjunction. Each clause forms a different set.

Literals: atoms and negated atoms

Clauses: sets of literals joined by OR

The Resolution Rule

Let S1 and S2 be sets of clauses, and U be the set Union operator:

```
({a,b,...} U S1) & ({~a,b,...} U S2)  ==>  {b,...} U S1 U S2
```

E.g.: `{a,b,~c} & {~a,b,d}  ==>  {b,~c,d}` resolve on a

Termination

```
{a} & {~a}  ==>  { }  ==>  False

{a,~a}  ==>  True
```

Not complete

```
~{~a,~b} & { }  ==>  no action
```

Resolution and Natural Deduction

	Resolution	*Natural deduction*
***End Case*:**	`{a} & {~a} = { }`	`(a & ~a) = False`
***Modus Ponens*:**	`{a} & {~a,b} ==> {b}`	`a & (~a v b) ==> b` `a & (a -> b) ==> b`
***Chaining*:**	`{a,b} & {~a,c} ==> {b,c}`	`(a v b) & (~a v c) ==> (b v c)`

Lattices

A lattice is a *directed graph* with links representing an ordering relation. Lattices can have a maximal and a minimal element

```
( )   maximal
 |
( )
 |                > greater than
( )
 |
( )   minimal
```

A ***partial ordering*** uses the ordering relation *greater-than-or-equal-to*.

```
    ( )         maximal
   /   \
(1)     (2)                  1 = 2
   \   /
    ( )         minimal
```

Hasse Diagrams (aka lattices)

A *set* and an *ordering relation* {S,>}, such that

- each object is a *vertex*
- if `(a > b)`, then a is *higher than* `b`.
- if there is no c such that `(a > c > b)`, then a *is connected to* `b`.

```
(a)     (b)     no maximal element
   \   /
    (c)         (a>c) & (b>c) & (c>d)
     |
    (d)         minimal
```

Matrix Logic

By arranging the truth table of a Boolean function in a matrix form, the rules of logic can be converted into the rules of matrix algebra. The general format is:

```
           B
         T   F
      T  .   .
    A
      F  .   .
```

Some examples:

```
A & B      A v B      A = B      A->B        A          ~A          T

 1 0        1 1        1 0        1 0        1 1        0 0        1 1
 0 0        1 0        0 1        1 1        0 0        1 1        1 1
```

Each Boolean matrix is an ***operator***. That is, in this formulation, there are no objects. When using binary operations, matrix addition is `xor`; matrix multiplication is `and`.

```
a + b = c                    a * b = c

0 + 0 = 0                    0 * 0 = 0
0 + 1 = 1                    0 * 1 = 0
1 + 0 = 1                    1 * 0 = 0
1 + 1 = 0                    1 * 1 = 1

       xor                          and
```

Note that these relations are the same ones that apply to computational addition.

As well, some matrix combinations result in matrices which are not Boolean functions. This then extends Boolean operations into generally unexplored territory, *imaginary Boolean operations*. Some examples of translating between operators:

```
a + ~a = T                    1 1  +  0 0  =  1 1
                              0 0     1 1     1 1

a * ~a = a                    1 1  *  0 0  =  1 1
                              0 0     1 1     0 0

xor + and = or                0 1  +  1 0  =  1 1
                              1 0     0 0     1 0

nor^2 = nor                   0 0  *  0 0  =  0 0
                              0 1     0 1     0 1

xor^2 = equal                 0 1  *  0 1  =  1 0
  (square-root of equal)      1 0     1 0     0 1

and + or = ?                  1 0  +  1 1  =  2 1
                              0 0     1 0     1 0
```

Boolean Cubes

A Boolean function can be expressed in terms of a collection of vertices of a hypercube (*this is not the same use as the lattice hypercube*). The set of all Boolean functions of N variables is defined by all the possible collections (the power set) of vertices (called **cubes**).

Each cube is the *conjunction of unique literals*, one from each variable. The whole is formed by the disjunction of all cubes.

Examples:

1 variable

```
a   ~a
·---·
```

All possible combinations:

void	= F	no cubes
a		single cube
~a		single cube
a v ~a	= T	both cubes

2 variables

```
 a&b   a&~b
   ·---·
   |   |
   ·---·
~a&b   ~a&~b
```

All possible combinations:

void	= F	no cubes
a&b	= and	single cubes
a&~b	= nif	
~a&b	= nfi	
~a&~b	= nor	
a&b v a&~b	= a	two cubes
a&b v ~a&b	= b	
a&b v ~a&~b	= equal	
a&~b v ~a&b	= xor	
a&~b v ~a&~b	= ~b	
~a&b v ~a&~b	= ~a	
a&b v a&~b v ~a&b	= or	three cubes
a&b v a&~b v ~a&~b	= fi	
a&b v ~a&b v ~a&~b	= if	
a&~b v ~a&b v ~a&~b	= nand	
a&b v a&~b v ~a&b v ~a&~b	= T	all cubes

Boolean Cube Operations

Cubes can be used for computation, either symbolically or physically.

function = set of cubes

not function = set of cubes not in function

f or g = overlay the cubes of f and the cubes of g

f and g = intersect the cubes of f and the cubes of g

Perspective as an Operator

By removing the orientation of a Boolean cube (or a Boolean lattice), varieties of Boolean function collapse into the same form. For example, all single cube functions are the same (i.e. composed of one cube) with orientation is ignored. Another example, expressed in matrix notation:

```
1 1  =  0 1  =  0 0  =  1 0
0 0     0 1     1 1     1 0

 a       ~b      ~a      b
```

These four functions are the same when the matrix is free to rotate.