**Ky-Cuong Huynh**



# CS 161 Notes

DECEMBER 25, 2015

Below are my notes from CS 161 during the Fall 2015 quarter at UCLA. I hope that current CS 161 students find them useful. NOTE: these notes are incomplete, as I took all notes by hand for Bayesian networks and onward. I do not have plans to transcribe them to here.

Open a pull request if you find any errors, and I'll gladly correct them. To read the notes offline, download this page's Markdown version and open it in Sublime Text, with the MarkdownEditing package installed.

Happy studying!

–

    - LISP
        - Lisp Expressions
        - setq, car, cdr
        - cons and list
        - Boolean expressions
- Lecture 2: 30 September 2015
    - Conditionals with cond
    - Functions
        - Numeric Examples
        - Symbolic Examples
        - Local Variables: let and let*
        - Equal vs =
- Discussion 1: 2 October 2015
    - Lisp Review
- Lecture 3: 5 October 2015
    - Reformulating into a Search Problem
        - Cannibals and Missionaries Problem
        - Three Functions to Rule Them All
        - 8-Queens Problem
    - Problem Solving
        - Search Strategies
            - Properties
                - Uninformed (blind) vs Informed (heuristic) Strategies
        - Breadth-first Search (BFS)
- Lecture 4: 7 October 2015
    - Depth-first Search
        - Properties
    - Depth-limited search
    - Iterative Deepening Depth-First Search
        - Properties

        - Lemma: number of nodes in a search tree
    - Repeated States
    - State space
- Discussion 2: 9 October 2015
    - Search Formulation
        - Search Tree
        - Search Engine Functions
        - Search Strategies
            - Properties
            - Breadth-first Search (BFS)
            - Depth-first Search (DFS)
            - Depth-limited search
            - Iterative Deepening Depth-First Search
- Lecture 5: 12 October 2015
    - Uniform Cost Search
        - Properties of UCS
    - Greedy Search
        - Properties
    - A* Search
        - Properties
        - Admissibility of h()
        - Proof of A* Optimality
        - 15-puzzle and A*
        - Performance Comparison
- Lecture 6: 14 October 2015
    - Effective Branching Factor
    - IDA*
    - Constraint Satisfaction Problems (CSPs)
        - Making a Search Tree for CSPs
        - The Path to Backtracking Search

        - Factors that Affect CSP Search
- Discussion 3: 16 October 2015
    - Uniform Cost Search
    - Greedy Search
    - A* Search
    - Constraint Satisfaction Problems
- Lecture 7: 19 October 2015
    - CSPs and Exploiting Problem Structure
        - Forward Checking
        - Arc Consistency
        - Constraint Graphs in Polynomial Time
        - Solving Non-Trees with Trees
    - Incomplete Methods
        - Local Search
            - Random Restarts/Hill Climbing
            - Properties
- Lecture 8: 21 October 2015
    - Hill Climbing
    - Simulated Annealling
    - Two-player Games
    - Tic-Tac-Toe
    - Minimax
        - Chess and Minimax
        - Evaluation Functions
        - Alpha-Beta Pruning
- Lecture 9: 26 October 2015
    - Knowledge Representation and Reasoning
    - Propositional Logic (Boolean logic)
    - Syntax
    - CNF and DNF

        - Horn CNF (Horn clauses)
        - Example Sentences
    - Semantics
- Lecture 10: 28 October 2015
    - Entails and Meanings
    - Implication, Validity, Consistency, and Equivalence
        - Valid Sentence
        - Inconsistent Sentence
        - Equivalent Sentences
        - Implication
    - Truth Table Queries
    - Inference
    - Refutation Theorem
    - Resolution
        - Unit Resolution
- Discussion 5: 30 October 2015
    - Syntax
        - Normal Forms
    - Semantics
        - Validity, Consistency, Equivalence, Implication
        - Proving Sentences
    - Inference
        - Inference Rules
        - Exhaustive Resolution
        - Converting to CNF
- Lecture 11: 2 November 2015
    - Converting Propositional Sentences to CNF
        - CSPs and Consistency
        - WALKSAT
    - First-Order Logic (Predicate Calculus)

        - Syntax
        - Quantification
- Lecture 12: 9 November 2015
    - Converting English to First-Order Logic (FOL)
        - Contrapositive
    - Inference Engines
        - Binding Lists
        - Prolog Example
    - Workflow for FOL on Digital Logic
        - Decide vocabulary
        - Encode general domain knowledge
        - Encode problem-specific information
        - Diagnosis
    - FOL Inference
        - Resolution in FOL
- Discussion 6: 13 November 2015
    - Syntax
    - Equivalents
    - Knowledge Engineering
        - Construct a Vocabulary
        - Encode General Knowledge
        - Encode Problem-Specific Knowledge
    - Unification
        - Unique General Unifier
- Lecture 13: 16 November 2015
    - Converting to CNF for FOL; Skolemization
    - Resolution on FOL KB
    - Reduction to Propositional Logic (Propositionalization)
        - Semi-Decidability
    - Probabilistic Reasoning

- Lecture 14: 18 November 2015
  - Probabilistic Beliefs
  - Belief Revision
    - Bayes Conditioning/Rule
- Lecture 15: 23 November 2015
  - Probability Rules
    - Chain Rule
    - Bayes' Rule
  - Steps to Find a Probability
  - Bayesian Networks

# Lecture 1: 28 September 2015

## Introduction

"Monolithic approach unnecesary for usefulness": no need for magical Star Trek computer that does everything. Modern AI only does particular things: perception, learning, prediction, etc.

AI does overlap with other fields, but we're interested in building rather than understanding. And we want rational behaviors rather than human-like ones.

An example of how AI used to be is the Turing test. We wanted human behaviors at the time. To pass it needs a lot: NLP, learning, etc.

The architecture of intelligent agents:

Embedding environment => Perception & Action (not covered here) => Learning + Knowledge + Reasoning (what we will focus

on).

## Knowledge Representation

- How do we represent knowledge?
  - Need facts and beliefs (uncertain knowledge)
- What knowledge is relevant?

- How do we acquire knowledge?
  - Experts
  - Converting from elsewhere
  - Learning

Facts example: Blah people have jobs. Logic puzzle. Who has which jobs? We can use propositional and first-order logic to solve.

Beliefs example: blah probabilities, what's the chance of blah? We'll use belief networks, fuzzy logic, etc.

## Reasoning

Questions:

- How do we formalize reasoning: deduction, revising beliefs upon encountering contradictions, determining causality, etc. We can do deduction, but we have not solved belief revision or causality. We'll discuss only deduction.

- How do so efficiently in time, space, footprint, etc.?

Example of belief revision: got gold rings, later stained by sulfuric acid, and so they couldn't have been gold.

For causality: when do we say A caused B? The answer often depends on our purpose: explanation, liablity, predictions, etc.

With KR & R (knowledge representation and reasoning), we can do:

- Medical diagnosis
- Automated theorem proving
- Credit card fraud
- Formal verification of hardware: working multiplier, interlock on microwave, etc. We take our knowledge of the design and automatedly reason on it to verify.

## Natural Language Understanding

Applications:

- Understand NL to let us interface easily
- Generate NL to have nice responses
- Summarize texts for us
- Machine translate languages; Cold War-era push for this.

Difficulties: we need context, but we have syntactic and pragmatic ambiguity. We're not sure who's who ("They ran to them"), and what we can do ("Go pick that up").

Classically, we'd tackle this by breaking it down via sentence structure, and apply grammars to trees, etc. We have not figured this out. Nowadays, we use languages corpuses and just apply machine learning (probablistic reasoning). We have lots of data on mapping sentences in one language to another. This is Google Translate, whose components are nearly orthogonal to the languages at play. What's new? Our standards for success

have changed too. We want just the gist, not national intelligence.

Also worth a try: Eliza the Chatbot. It just used clever templates. That was done decades ago.

## Machine Learning

What helps us: we have lots and lots of data.

The fundamental paradigm shift: we ask for votes from people, data, etc., rather than trying to do everything from first principles.

It can be:

- Supervised: we train the machine on observations and their linked actions
    - Example: character recognition. Safety features on modern cards: self-stopping to avoid collisions, lane guides, etc.
- Unsupervised: find patterns in the data for us
    - Recommender systems on Amazon, Netflix, etc. We can take movies and user ratings and predict ratings for other users.
- Reinforcement: we train the machine by saying yes/no to its observations

Another area worth checking out: planning, which is used for things like autonomous spacecraft, etc. This is useful for communications that are laggy, non-existent, etc.

And of course robotics. Examples: DARPA Grand Challenge.

Here at UCLA: Darwiche on KR & R, Korf on search, Yuille on vision, Sha on ML, etc.

// Break time!

# LISP

Today and Wednesday will cover LISP.

## Lisp Expressions

Simplest expressions in Lisp:

- Numeric
- Symbolic
- Boolean
- Conditional
- Function definitons

We'll blast through these today, then write programs like hell on Wednesday. Lisp does have an REPL interpreter.

General form of a Lisp expression:

```
(operator arg1, ..., argN)
```

Operator is an atom or a function (built-in or user-defined) or a special operator (with side effects, counter to functional paradigm).

We evaluate left-to-right:

```
(+ 137 349)
;; => 486
```

```
;; 137, 349 are atoms
```

**Atoms** are numbers, symbols, strings, etc. We call them “literals” in other languages.

```
(- 1000 334)
(+ (* 3 5 1) (- 10 6))
```

Everything is a list, and we nest all the time. Parentheses all the fucking time, of course.

Next comes the `quote` operator:

```
(quote (+ 3 1))
; => (+ 3 1)
```

We get back a data list, and this is very, very common. Shortcut:

```
'(+ 3 1)
```

This lets us create data structures and manipulate them directly.

## setq, car, cdr

To bind a value to a symbol, we use `setq`:

```
(setq x 3)
; x is now bound to the value 3
```

New result:

> x 3

We can get fancier:

```
(setq x (+ 3 1))
```

> x 4

> y ; => unbound symbol error

`setq` is barred from use in our programs. It binds things globally, which is bad, bad, bad. Another example:

```
(setq x '(+ 3 1))
```

> x (+ 3 1)

`x` is not holding a string, but an actual Lisp expression.

Now, given `x`, we can get its head and remaining elements via `first` and `rest`.

```
(setq x '(a b c d))
(car x) ; > a
(cdr x) ; > (b c d)
```

Sometimes `first` is called `car` instead. `cdr` is rest.

To get `c`:

```
(car (cdr (cdr x)))

;; OR, stupid abbreviation:
(caddr x)
```

To get `b`:

```
(setq x '(a (b c) d e)
```

```
(caadr x)
```

To get `c`:

```
(cadadr x)
; Need a last car, as we get a list (c) back
,
; but we want just c
```

Note:

```
(cdr (a b)) ;; => (b), a list
(car (cdr '(a b))) ;; => b, a symbol
```

Next up, NIL, which represents no value or an empty list.

```
(car ()) ;; => NIL
(car NIL) ;; => NIL
```

## cons and list

As car and cdr can tear apart a list to any element, we can construct any list by putting together a first element and the rest.

NOTE: It only takes two arguments.

// McCarthy didn’t like to type, the bastard. Who thinks // something like

To make `(1 2)` from scratch:

```
(cons 1 (cons 2 NIL))
```

Easier:

```
(cons 1 '(2))
```

Note the quote! Otherwise, it'll try to evaluate 2 as a function: "Not a function name". Be careful of stray quotes:

```
(cons 1 '(2 NIL))

;; => (cons 1 '(2 NIL))
```

Edge case:

```
(cons NIL NIL)
;; => (NIL) or (())
```

As this is prepending an empty list in front of an empty list.

Another way to make a list (though avoided due to recursion):

```
(list arg1, ... argN)
```

With the list hold the arguments in the order they were passed in. Remember: if symbols/functions are undefined, they will toss an error unless we use quotes:

```
(list 'a 'b)
;; => (A B)
```

Another example (literals can be list'ed together easily):

```
(list 1 (list 1 2)) 3 4)
;; => (1 (1 2) 3 4)
```

If we are defined:

```
(setq a 7)
(setq b 8)
(list a b) ;; => (7 8)
```

## Boolean expressions

In Lisp, the only falsy values are `NIL` and `()`. The default "true" value is `T`. Goddamned laziness.

Predicates check if something is true:

```
(numberp ...) ;; True if arg is a number, sy
mbol, list, etc.
(symbolp ...) ;; True if a symbol
(listp   ...)
(null    ...) ;; If something is the empty l
ist
```

Another useful one: `equal`, which checks type and value.

`not` takes one arg, `or` and `and` take unlimited args.

`and` also short-cuts and leaves early if it's nil. If all is true, it return the last non-nil element. `or` does the opposite; the minute it finds something non-nil, it returns it, as that is true. That's very shortcutty, as everything non-nil is true.

# Lecture 2: 30 September 2015

Some Boolean practice:

```
(> 3 1) ;; T

(< 3 1) ;; NIL

(not (> 3 1)) ;; NIL

(not 3) ;; NIL

(not nil) ;; T
```

```
(and (+ 2 3) (cdr '(A)) (+ 1 3)) ;; NIL, as
(cdr '(A)) == NIL
```

## Conditionals with cond

Executes the doThis statements for the first non-nil booleanCase. If all booleanCases are nil, then the whole `cond` returns nil.

```
(cond

  ( (booleanCase1) doThis1 doThis2 )
  ( (booleanCase2) doThis1 doThis2 )
)


;; Notice how we return an int or list.
;; In fact, we're returning an atom, a numbe
r or a symbol
;; Functions return Lisp expressions in gene
ral
;; It's not a strictly typed language, and t
his makes
;; it easy to have programs write programs
(let ((x 5)) )
(cond
  ( (= x 0) 0)
  ( (< x 0) 'neg)
  ; How we do default cases
  (T 'pos)
)
```

## Functions

```
(defun functionName (param1 param2 param3)
  ;; Code
)
```

```
(defun square (x)
   (* x x)

   ;; Value of last expression is returned
)
```

Example:

```
(square 2) ;; => 4
```

Defun is special with a side-effect: it creates a new function.

```
(defun abs (x)
    (cond
        ;; Unary negation operator exists
        ;; Wanna get a positive if negative
        ;; Could also do (* x -1)
        ( (< x 0) (- x) )

        ;; T is the default/else case; don't
 make
        ;; it first; it becomes an always ot
herwise
        ;; Could check that x actually is a
number, etc.
        ( T x)
    )
)
```

What happens if we define something twice? The later definition overrides the first one.

Side effects == state changes, printing something out, setting a global, etc.

```
(defun sum (x y)
    (+ x y)
)
```

Numeric, then symbolic expressions in functions follow. All recursive!

- Numeric expressions
- Symbolic expressions (selecting/getting information)
- Symbolic expressions with construction

## Numeric Examples

Factorial:

```
(defun fact (n)
    (cond
        ((= n 0) 1)
        ;; Not base case, so recursively do
it
        (t (* n (fact (- n 1))))
    )
)
```

Exponent:

```
// Assume power >= 0 for now
// O(P) with this implementation
(defun exp (base power)
    (cond
        ((= power 0) 1)
        ((* base (exp B (- power 1))))
    )
)
```

Faster way is to divide and conquer into two halves, for even and odd cases:

even: B^P = (B^(P/2))^2 odd: B^P = B * B^(P-1)

This is logarithmic in P.

But we still need local variables! Thus far, we technically only knew about globals.

```
(defun exp (b p)
    (cond
        ((zerop p) 1)
        ;; We now need a local variable to h
old each half
        ;; If we repeat the exp twice, it's
linear again
        ;; Or we could use the square functi
on, or cleverly exponentiate again
        ;; to raise to power 2
        ((evenp p) (square (exp b (/ p 2))))

        ;; These parantheses are probably mi
smatched
        (t (* b (exp b (- p 1))))
    )
)
```

## Symbolic Examples

Given a list of numbers (of any length), find their sum:

Example usages:

```
(sum-list '(1 2 3)) ;; => 6

(defun sum-list (L)
    ;; Base case is 0, as that's the additiv
e identity
    ;; Compare to products, where multiplica
tive identity is 1
    (cond
        ;; Empty list?
        ((null L) 0)
        ;; Otherwise, add the first element
to the sum of the rest
```

```
        (t (+ (car L) (sum-list (cdr L)))
    )
)
```

// I'll certainly practice recursion like hell with Lisp

Next, check if a given element is a member of a list.

```
;; This actually is a defined function for L
isp, so add ?
;; True if x \in L
(defun member? (x L)
    (cond
        ;; Don't forget the empty list base-
case!
        ;; This goes first, as otherwise nil
 matches with nil,
        ;; and the function returns T, which
 is incorrect
        ;; for nil being in nil, lol
        ((null L) NIL)

        ;; Equal to first element?
        ((equal x (first L)) t)

        ;; Else, try the rest
        (t (member? x (rest L)))
    )
)
```

Finally, return the last element of a list, assuming `(length L) >= 0`:

```
(defun last! (L)
    (cond
        ;; Avoid length, which is O(N); this
 checks if single element
        ;; quickly and easily. It returns th
e first elem if only one elem
```

```
        ((null (rest L)) (first L))

        ;; Else, recursively, cut the list d
own to size 1
        (t (last (rest L)))
    )

)
```

By the way, `member` in Lisp doesn't return T or nil. It returns a sublist.

Another example, finding the nth element. Assume n >=0, and within length of L. :

```
(defun nth (L n)
    (cond
        ;; Tis first elem? Give it back
        ((= n 0) (first L))

        ;; Else, chop down the list and shif
t the index
        (t (nth (rest L) (- n 1)))
    )
)
```

Now, let's remove an element from a list. Given a list and a specified element, give back that list without that element (all instances of it)

```
(defun remove (x L)
    (cond
        ;; Empty list gives back empty list
        ((null L) NIL)

    ;   ;; Eliminate this first element from
 what we examine
        ;; This takes care of the rest of th
```

```
e list
        ((equal x (first L)) (remove x (rest
 L)))

        ;; Imaginge x = b
        ;; L = (a b c)
        ;; Then want a + (b c)
        ;; We need to construct a list of th
at
        ;; first to-be-kept element with a s
anitized
        ;; version of the rest of the list
        (t (cons (first L) (remove x (rest L
))))
    )
)
```

Now, let's append lists together. Given two lists, append L2 to L1 (this is actually built-in).

Example: `(append '(a b) '(1 2)) ;; => (a b 1 2)`

```
(defun append (L1 L2)
    ;; We recurse on L1 to size against L2
    (cond
        ((null L1) L2)
        (t (cons (first L1) (append (rest L1
) L2)))
    )
)
```

Why can't we simply do cons of (a b) and (1 2).

Cons is meant for an atom and a list to get back a clean list.

Cons'ing list with list is a list with three elements:

((a b) 1 2)

We don’t want that sublist.

Visually:

(a b c) (1 2)

is the same as cons a with (append ‘(b c) ‘(1 2))

Try reverse for practice.

## Local Variables: let and let*

Example:

Nested functions are possible, but not with defun.

```
(defun math (x y)
    (let (
        (a (+ x y))
        (b (* x y))
        )
    )

    ;; Local vars ready, do something now
    (- a b)
)
```

`let` defines local variables. We a have a list of bindings:

```
(let ( ;; Start binding list
    (var1 (valueExpr1))
    (var2 (valueExpr2))
    )
)
```

But usually, we use `let*` instead, as `let` does the bindings in parallel. `let*` does bindings in sequence.

An example of global and local scope mixed together:

```
(setq x 5)
(+
    (let (
        (x 3)
        )
    ;; The inner x is 3 => 33
    (+ x (* x 10))
    )

    x ;; This outer x is 5 => 33 + 5 = 38
      ;; As there's that outer addition
)
```

Another example:

```
(setq x 2) ;; Globally, x bound to 2
(let (
    (x 3)
    (y (+ x 2))) ;; Parallel bind, so this x
 is bound to 2, from global
                 ;; as we're doing the local
 binding simultaneously,
                 ;; and haven't seen it yet

    (* x y)
)

(let* (
    (x 3)
    (y (+ x 2))) ;; Sequential bind, so this
 x is bound to 3

    (* x y)
)
```

**Equal vs =**

- (= …) will compare value. This is faster.
- (equal …) will compare value and type
- (eql …) is another function, which is different.

# Discussion 1: 2 October 2015

Hang Qi

hangqi@cs.ucla.edu

Tues/Thurs 1:30-2:30 PM

3rd year PhD student. He also attends every lecture with us.

No break in section.

General workflow:

1. Write source into .lsp file
2. (load “filename.lsp”) into interpreter and then try out function calls

## Lisp Review

We have either atoms or lists. Atoms are symbols or numbers:

- Numbers: 1, 0. 5
- Symbols: x a sum (i.e., non-numeric types)

Lists are lists of atoms or lists.

Numeric operators use prefix notation and include arithmetic and Boolean comparators.

Selectors operate on lists and are ones like first/car, rest/cdr, etc.

Constructors are cons and list.

// Practice from Forney

```
;; [Purpose]
;;   Returns a list that is the reverse of i
ts inputted from
;; [Inputs]
;;   L (list) to reverse
;; [Outputs]
;;   (list) L reversed

;; () => ()
;; (1) => (1)
;; (1 2) => (2 1)
;; (1 2 3) => (3 2 1)
```

```
(defun reverse-list (L)
    ; Recursion for everything!
    cond (
        ; Base case: empty list. Just return
 NIL.
        ( (null L) NIL )
        ; Otherwise, append the first elemen
t to the end of the reversed list
        ; append expects to append the eleme
nts in the second arg (a list)
        ; to the list that is the first arg,
 and returns the merged list
        ; BUT: append expects list arguments
, so we need to  make a list of
        ; the element returned by (first), o
therwise, it tries to eval. first L
        ( T (append (reverse-list (rest L))
(list first L)) )
    )
)
```

Another one!

```
;; [Purpose]
;;   Replaces the nth element of the given l
ist L
;;   with the given list R
;; [Inputs]
;;   L (list) to replace element within
;;   n (int) index to replace in L
;;   R (atom / list) replace L[n] with R
;; [Outputs]
;;   (list) L with replacement R
(defun replace-element (L n R)
    (cond
        ; Base case: empty list, so return N
IL
        ( (null L) NIL )

        ; n is 0, so we reached the right sp
ot for replacement
        ; So return R and the rest of the li
st, so as to cut
        ; out the replaced element
        ( (= n 0) (cons R (rest L)) )

        ; Otherwise, L is non-empty, and n i
s non-zero,
        ; so we're not at the right spot yet
, and
        ; need to return the first element p
lus the rest
        ; of the list, with the right elemen
t replaced.
        ; We also decrement n in the process
. This gets
        ; us the elements before and after t
he replaced one.
        ( T (cons (first L) (replace-element
 (rest L (- n 1) R))) )
```

```
    )
)

;; [Purpose]
;;   Returns the value of n raised
;;   to integer power pow
;; [Inputs]
;;   n (number): number to raise to power po
w
;;   pow (integer): power to raise n to
;; [Output]
;;   (number) n ^ pow
(defun power (n pow)
    (cond
        ; Base case: power is zero, so retur
n 1, an identity
        ( (= pow 0) 1 )

        ; Positive n recursive case: return
n * n^(pow - 1)
        ( (> pow 0) (* n (power n (- pow 1))
) )

        ; Negative n recursive case: return
1/n * n^(pow - 1)
        ; n is negative, but we want 1/n^pow
, not 1/n^(-pow).
        ( (< pow 0) (/ 1 (power n (* pow -1)
)) )
    )
)

;; [Purpose]
;;  Returns a list with every element but th
e last one in the input list
;; [Input]
;;  A list L
;; [Output]
;; A list with elements 0 to n - 2 of input
L
```

```
(defun butlast (L)
    (cond
        ; Base case: empty list
        ((null L) NIL)

        ; Check if we only have 1 element. A
lso return NIL.
        ((null (rest L)) NIL)

        ; Recursive case, we have 2 or more
elements
        ; so combine the first with the rest
, except fro the last one
        ; Recursion! In the 1 element case,
the last one, we return NIL
        (T (cons (first L) (butlast (last L)
) )
    )
)
```

```
; Reverse the list L
(defun reverse (L)
    (cond
        ; Base case: empty list
        ( (null L) NIL)

        ; Recursive case: add reverse of res
t to first elem
        ( T (append (reverse (rest L)) (list
 (first l)) )
    )
)
```

# Lecture 3: 5 October 2015

# Reformulating into a Search Problem

Couple of weeks on this topic, one of the most central and broadly applicable things we'll learn.

We take a problem, formulate it into a search problem, and then feed it into a search engine.

Problem => Search problem => Search engine => Solution ^ | Search strategies

We do this because search is a problem that we can represent in the same way all the time, no matter which original problem was reduced to it. Knowing this, we can come up with different ways of solving search, which will solve any problem we can reformulate in terms of search.

We'll handle that first transition to search, and some common strategies today.

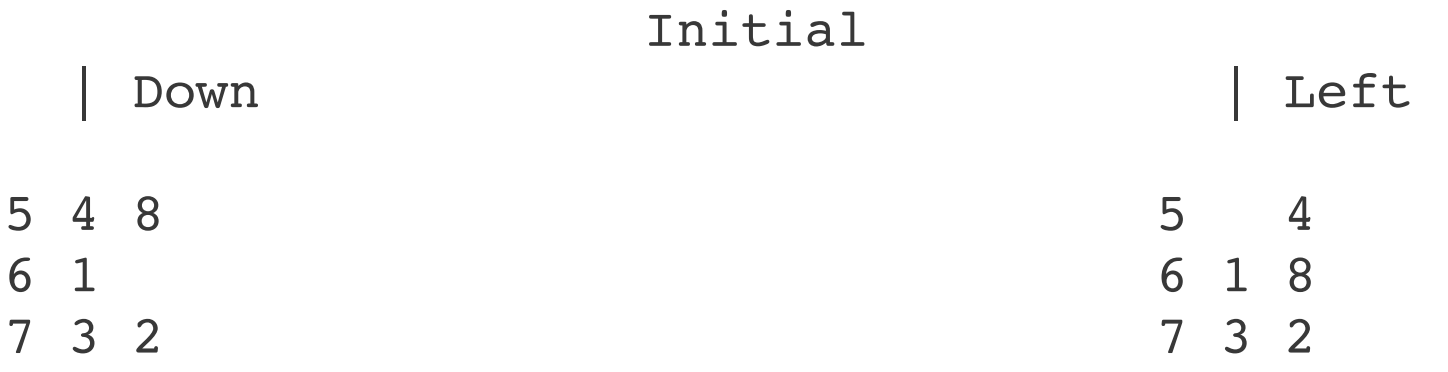
First (toy) problem, the 8-puzzle. Slide pieces around within a puzzle to get a goal configuration in the least moves possible.

```
5  4           1  2  3
6  1  8   =>   8     4
7  3  2        7  6  5
```

Notice that at each point (a **state**), we can move (an **action**) in one of 4 ways to get to another point (state). Don't think about moving each of the pieces. Instead, just move the blank, which swaps position with the number on the new spot it takes up. We can move the blank in four different ways:

- Left
- Right
- Up
- Down

Sometimes, we have fewer choices, as above (only down or left)




. . . . . .

That state we want to reach is known as a **goal state** or a **final state**. Notice that if do list out the whole **search tree**, we would eventually find the goal state. We have reduced our puzzble solving to finding a path of minimum length within this tree. The tree encodes our possible actions from each state. One of those states is our goal. Our solution is the path we found from the initial state (root node) to the final state.

It’s also possible that there can be more than one winning state.

What makes these search trees harder? If there are too many choices.

The **branching factor b** records the number of children per

node, the number of actions possible at each state. Note that it's an upper bound, as the number of children may vary by node, with itself bounded by the number of possible, distinct actions. We can get a tighter branching factor by showing that no more than a subset of certain size is possible at each state.

The depth of the search tree records the magnitude of the number of possible states, but we're interested in the **depth d of the solution**, the length of the shortest path from the initial state (root node) to the goal state (some node below it). We'll actually encode the complexity of our problem in terms of these factors.

The 15-puzzle has a few trillion possible states, but can now be solved in a few milliseconds.

Chess' average branching factor is 36 (bad).

## Cannibals and Missionaries Problem

Three cannibals and three missionnaries. with one boat. Want to get them all to the other side.

We should never have more missionaries than cannibals on either side.

The boat can transport:

- 1 M
- 1 C
- 1 M, 1 C
- 2 M
- 2 C

It cannot be empty moving from one side to the other.

// This is so colonial!

Note that our current conception ignores games, scores, players, etc. Here, we have a yes/no on whether we reached the goal state. Book has a chapter for that.

Initial state:

```
|      [ ]  MMM
|           CCC
```

Final state:

| [] MMM |
| --- |
| CCC |

Again, we want to minimize the number of moves made. Also, we're assuming for the moment that each action has the same cost. Later, we can try to minimize that (each cost would be an assigned weight to each edge).

Which actions are **applicable** for our first state?

All of them actually.

CM:

```
CM [ ]     |    MM
           |    CC
```

CC:

```
CC []     |   MMM
          |   C
```

MM:

```
MM []     |   M
          |   CCC
```

C:

```
C []      |   MMM
          |   CC
```

M:

```
M []      |   MM
          |   CCC
```

But now, we have to send the book back, whlie still obeying the same constraints, and eventually getting everyone to the other side.

The above MM and M actions lead to **invalid states**!

*Even when an action is applicable, it may lead to an invalid state.*

That is, even if an action seems possible at first, it may lead to an invalid state. Therefore, our successor function should check for both the validity of our move as well as of the next state.

This problem can actually be solved in 11 steps.

## Three Functions to Rule Them All

The **initial state function** returns the initial state.

The **final state test function** (yes/no) tells us if a state is final.

It’s a predicate

The **sucessor function**, given a state, returns the valid next states, the children of a state. This abstracts out all the details of the problem, which separates the search reformulation from the search strategies.

We separate problem formulation (recasting it as a search problem) from problem solving (all computation, the search process). This specifies the search tree, which we’ll later build and search through. This abstracts out the search engine implementation details (problem solving) from the problem formulation. We can feed our functions into our engine without issue to solve any problem that we can reformulate into a search problem. We distinguish between reasoning and representation.

Another simpler example (4-puzzle):

```
  1       1  2
3  2   =>    3
```

Suppose we were given a search engine, and told to plug-in these three functions. All in Lisp of course. Let’s represent each state as a list.

```
(#M at-boat-location #C at-boat-location t/n
il)
    => true is right bank, nil is left bank
```

This gives us:

(1 1 NIL) CM:

```
CM [ ]    |   MM
          |   CC
```

(0 2 NIL) CC:

```
CC  [ ]     |   MMM
            |   C
```

MM:

```
MM  [ ]     |   M
            |   CCC
```

(0 1 NIL) C:

```
C  [ ]      |   MMM
            |   CC
```

M:

```
M  [ ]      |   MM
            |   CCC
```

Example successor function implementation:

```
(successors '(3 3 t))
```

Gives back a list of possible next states:

```
( (0 1 NIL) (1 1 NIL) (0 2 NIL) )
```

## 8-Queens Problem

Place queens on the board such that none of them are attacking each other. We'll do 4x4 for simplicity.

Invalid placement:

```
0  q  0  0
0  0  0  q
0  0  q  0
q  0  0  0
```

Given a magical search engine, we need only to make this into a search problem and give the engine a search tree spec. This is exemplary of a whole class of problems.

Initial state, empty board, with each action being placing a a single queen onto the board:

```
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
```

This has 16 possible children. 15 for each of the next children, etc.

Notice that we know the height beforehand now. We have a height of 4, as we only place 4 queens in total. 8 for 8 queens, etc. Later, when we know our possible search strategies, we can decide among them based on whether our search tree is finite. In fact, this tree is of fixed height.

Notice that our final-state-checker is now more complicated, and that we made it a predicate because in general there can be more than one final state. This problem is all about finding a final state, because we don’t know what it looks like. It’s no longer simply comparing, but verifying.

Notice also that our solution is known to be at depth 4.

These are **constraint satisfaction problems** (CSPs), which are a whole class of problems.

Antoher problem:

```
Given forty + ten + ten = sixty
```

Find arithmetic substitutions for each letter, e.g., 29786 + 850 + 850 = 31486. The numerical result matches the cipher results.

More examples of CSPs: university class scheduling, GPS navigation with shortest path, routing in computer networks, airline route planning, planning manufacturing robot arm paths, etc.

## Problem Solving

Given an initial state, and then we've built out part of the search tree, and we haven't seen the final state yet.

The leaves of our tree are the **frontier** or **fringe** are our departure points for where to go next.

Central questions:

- Which node do I pick to **expand** next? Expand means we pick a node check if it's the final state, and if not, then we **generate** its children thorugh the successor function which become part of the frontier.

Expand and generate now mean very specific things. Example question: number the nodes in the order in which they will be by a particular strategy.

Expand:

1. Check if goal state
2. If not, generate children

Generate:

1. Call the successor function
2. Add the returned children to the search tree

Frontier: not yet tested for final state, where we could go next.

Could we have a case where a returned solution is not optimal, meaning the best possible solution? Yes, depending on the search strategy. Some guarantee optimality, others do not.

## Search Strategies

### Properties

1. Optimality

Is the search strategy guaranteed to return an optimal solution? This is different from the most efficient (in time or space) solution.

1. Completeness

Is the search strategy guaranteed to find the solution if one exists? Or does it have the possiblity of going off infinitely and never stumbling upon the solution?

1. Time complexity

How long will the strategy take?

1. Space complexity

How much storage will the strategy take?

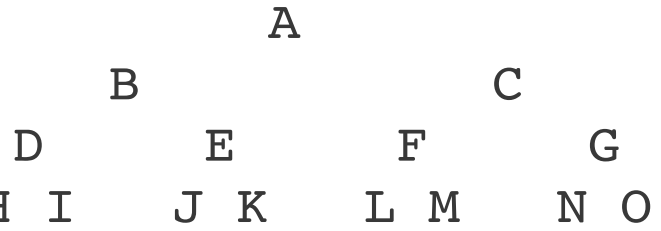
#### Uninformed (blind) vs Informed (heuristic) Strategies

These are the two broad categories of strategies.

Uninformed/blind strategies need only the three functions we defined earlier. The heuristic strategies take a fourth input, to be discussed later. This fourth input can vary in quality/usefulness, and the more effort we put into it, the more efficient we can make our strategy.

## Breadth-first Search (BFS)

Suppose the complete search tree is:

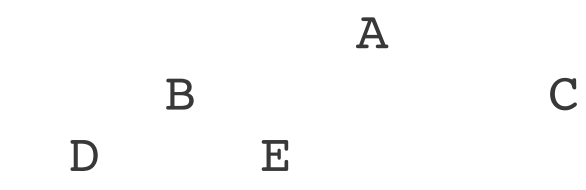



Goal state is O.

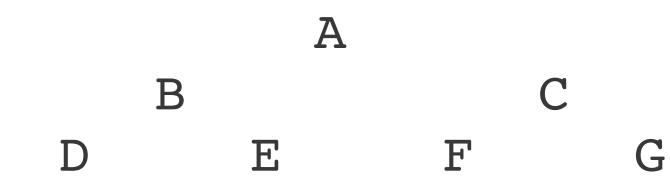
Our first strategy: expand the shallowest node.

But we begin with just:

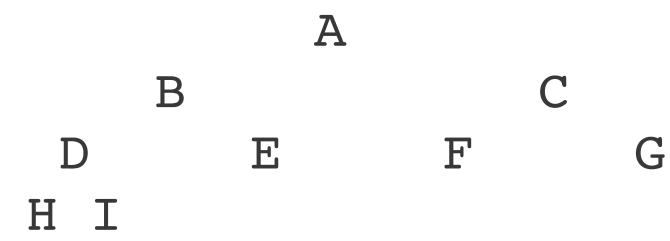

Becomes:




We only have one choice here: C, as it's the shallowest.

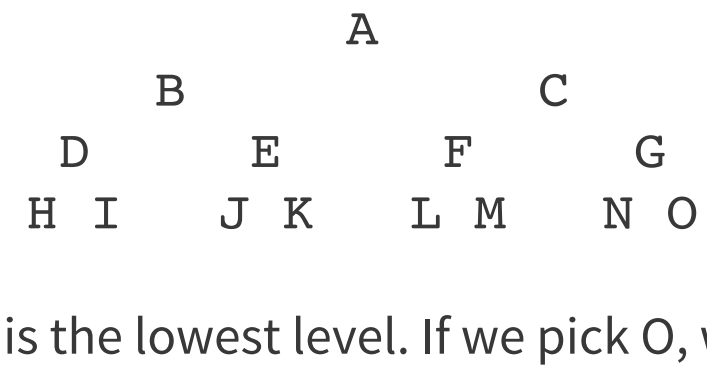



Now we could pick any of D, E, F, G. Suppose we pick D:




Then, we'd have to pick E, F, or G for expansion.

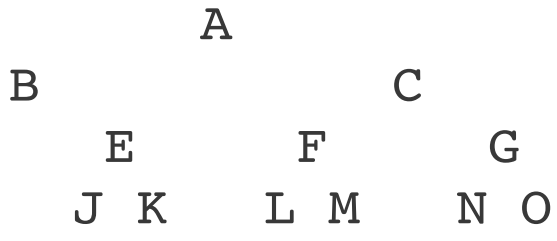
Eventually we get to:




Our fringe is the lowest level. If we pick O, we're done. But if we pick H, then we:

1. Is H the goal? No.
2. Okay, expand H. Oh wait, no children exist for H.
3. All right, then eliminate H from our search tree.

Suppose we pick I next. Then we eliminate I similarly, which means we also eliminate D.




We can number the order in which nodes are expanded:

1: A 2: B 3: C 4: D 5: E 6: F . . . 15: O

This happens to go left-right down the levels.

Is this strategy complete? Yes, as it visits all the nodes in increasing levels. We see nodes at depth 0, then 1, then 2, etc. This algorithm expands nodes by order of their depth. We do all nodes for d = 0, then 1, then 2, etc.

This also guarantees optimality, meaning we'll find the shortest solution, as the nodes being leftmost rightmost on the same depth is irrelevant, as the optimal solution cares only about depth.

Later we'll relate the number of nodes to b and d.

Number of nodes N is:

```
N(b, d) = b^0 + b^1 + b^2 + ... + b^d

        = b^(d+1) - 1 / b 01

        = O(b^d)
```

Book has a table of numbers to show how much space BFS takes up.

100 bytes per node. Depth 8, 1000 nodes per second, => 31 hours and 11 GB. Depth 10: 128 days and 1 TB. Depth 12: 35 years, and 111 TB.

This is the problem with exponential complexity.

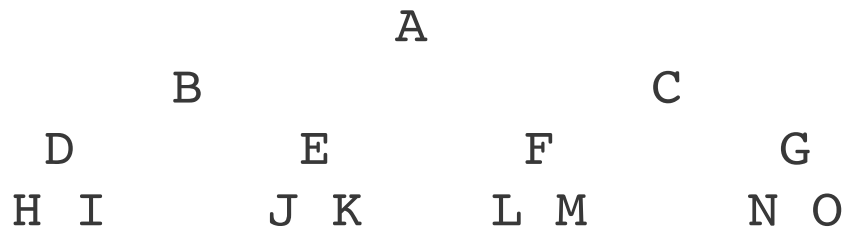
Then, later, we'll see DFS, which trades the first two properties to get better space complexity. Then, finally, we'll find a way to get

the best of both worlds.

# Lecture 4: 7 October 2015

// Heuristic search on Monday, which is good, // as I'm not prepared for that yet.

Again, our tree:




A search strategy answers the question:

> Which node should we expand next?

Once we expand a node, we check if it's the goal state, and then generate its valid children. These new children will join the fringe/frontier. The terminology here needs to be very precise for exams.

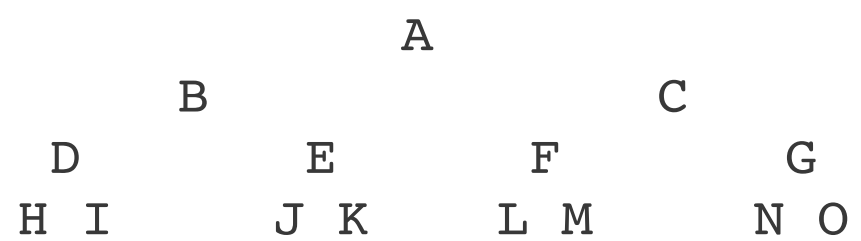
Why do we check goal state during expansion and not during generation (i.e., while generating them)? In principle, it's totally possible, but we risk losing optimality for many contexts. In general, checking goal state during expansion is usually better for optimality.

Recall: BFS is complete, optimal, and O(b^d) for space and time complexity.

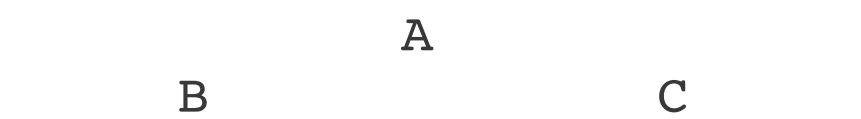
Why is it optimal? Because it always expands the shallowest levels first, which means it finds the most optimal (with fewest steps) solution first.

## Depth-first Search

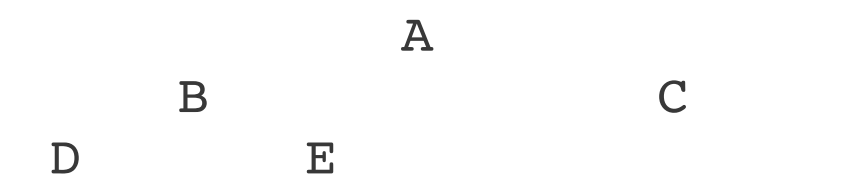
The opposite strategy: expand the deepest node.



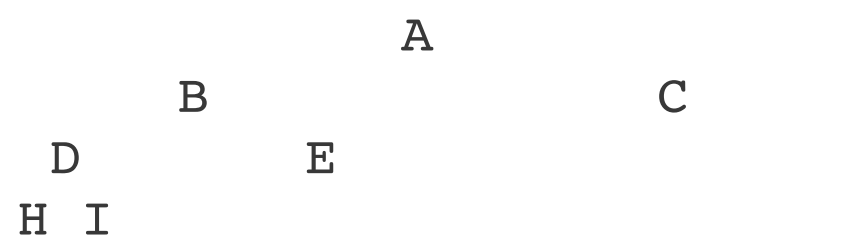

Begin with:



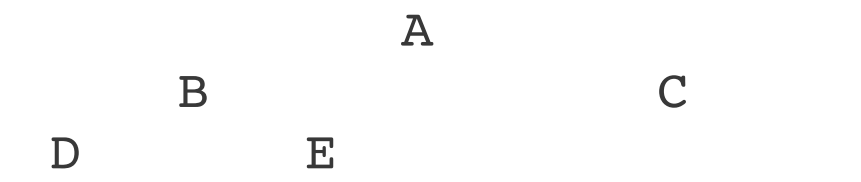

Expanding B:




We now choose D, which gets us:




We pick H, which is not a goal, but has no children, so we eliminate it from the tree.



```
          I
```

The same thing happens with I, which also causes D's loss:

```
              A
        B           C
            E
```

Next comes E:

```
              A
        B           C
            E
          J K
```

15 steps later, we end up with:

```
A
C
G
O
```

Notice how with BFS we had the whole tree in memory at one point, but that doesn't happen with DFS.

Rather than maintaining depth information, we can use a stack or queue to maintain the fringe itself. We can simply push the children on the stack for DFS and pop off to exp. With a queue, we end up with BFS, a level-order traversal, which has them on the opposite side.

1: A 2: B 3: D 4: H 5: I 6: E 7: J 8: K 9: C 10: F 11: L 12: M 13: G 14: N 15: O

BFS and DFS are expected to be automatic in our head. We'll be asked to mark expansion order on exams.

## Properties

1. Completeness

If the search tree is infinite, DFS is not complete. It could potentially go down a doomed branch for forever and ever. BFS though goes level-by-level, which allows it to eventually find a solution.

Though DFS would be complete if it's a finite tree. We're currently assuming d of the optimal could be infinite, but not b.

1. Optimality

If multiple solutions exist, and one is shallower and on the right side of the tree, then we'd find the deeper, non-optimal one first. DFS is not optimal, even though it might deliver the optimal solution sometimes.

1. Space complexity

Is this O(d)? No! We obviously could go much deeper than the optimal solution. Rather, we're bounded by the depth *of the tree* m. But what about the children at each node? We have to store the frontier, the siblings of each frontier node.

$O(bm)$

We're now linear in space.

1. Time complexity

We could potentially go down every wrong path possible before backtracking, traversing down and up, left to right. We end up

visiting every node (if finite), so exponential in time:

`O(b^m)`

But notice that BFS is O(b^d), with d <= m.

Interestingly, DFS is used more often in practice, since it's easier to deal with time complexity (we just wait longer) than with space complexity (not easy to add on more memory). If the search tree is complete (which gives completeness and possibly optimality too), then it's not too bad at all. How might we get completeness even with an infinite tree?

If we knew the optimal solution's cost, we'd know how many levels to go deep. Or if the solutions are dense (many in the treee), wrong turns aren't too costly.

But Darwiche is looking for something else. Hint via picture.

He drew a graph with start and end nodes. Dijkstra's?

But what if we have an upper bound on the cost of a solution?

"This problem can be solved in less than 33 steps, at the least."

So how do we use this to get O(bm) while gaining completeness? Simple: we cut the search depth at 33.

## Depth-limited search

A new parameter: L, the depth of search

Our new time complexity: O(b^L).

DLS:

- Complete
- Not optimal (could have a solution shallower than L)
- Space complexity of O(bL)
- Time complexity of O(b^L)

When might this happen? This graph, where if we have N nodes, we should be able to do it in at most N - 1 steps.

Parameters so far:

- b: branching factor
- d: depth of optimal solution
- m: depth of search tree
- L: depth of search

## Iterative Deepening Depth-First Search

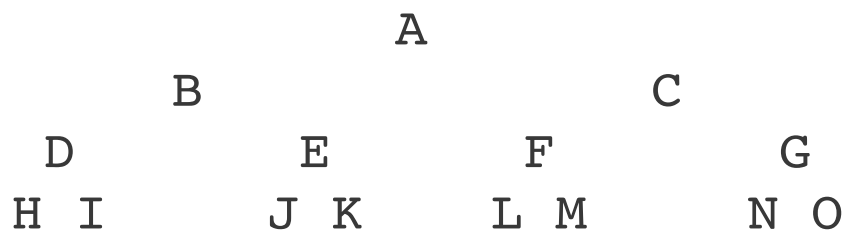
Suppose we don't know an upper bound on the solution's cost. We can't easily do DLS, and thus DFS or BFS. We're still blind, and we're avoiding the use of memory for now.

What if we did DLS, but guessed L? We'll make a finite number of guesses.

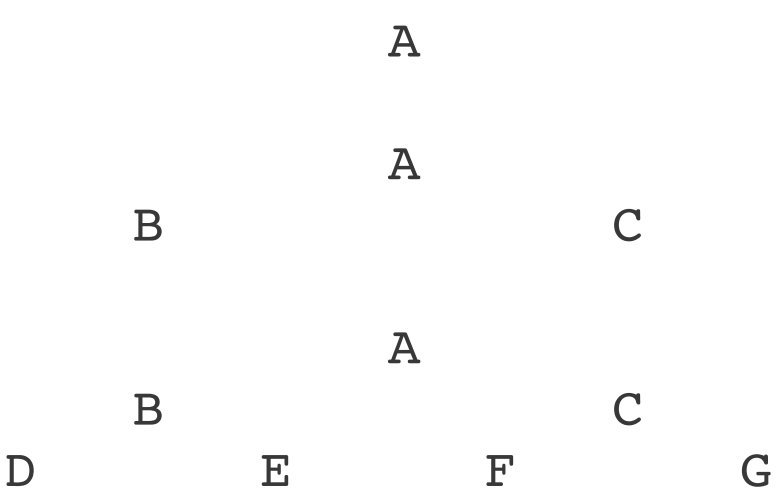
Suppose we try a random L and find a solution. What next? Try a shallower L. If we don't find a solution, what next? Try a deeper L.

But this isn't a systematic process. But what if we do L = 0, 1, 2, … m? We know that if we find a solution at any point, it's optimal, as we're diving to a set depth each time. Because we go

down the levels, we achieve that characteristic critical for optimality, as we always find the shallowest solutions first.




We're doing DFS on A, then ABC, then ABCDEFG, etc.




1: A (level 0) 2: A (level 1) 3: B 4: C 5: A 6: B 7: E 8: E 9: C 10: F 11: G 12: A

We're constantly revisiting, yet we do get completeness and optimality. Now how much complexity did we sacrifice?

But it's still linear space, as it's iterations of DLS, which is linear in space. It's a constant multiple of DLS.

## Properties

- Complete
- Optimal
- Space: O(bl)
- Time:

We don't go deeper than d, as that's the optimal solution.

Let’s add:

Iteration 1: 1 node 2: 3 nodes 3:

Recall, a tree with branching factor b has b/(b-1) nodes,

with this many steps:

```
b^d (b/(b-1))
```

For DFS.

Now, with iterative deepening search, we’re increasing d:

```
b^0(b/(b-1) + b^1(b/(b-1)) + ... + b^d (b/(b
-1))
```

Pull out b/(b-1):

```
(b/(b-1)) [ b^0 + b^1 + ... + b^d]
```

From last time’s derivation:

```
= (b/(b-1)) [ b/(b-1) * b^d]
```

So it’s DFS’s time complexity times (b/b-1).

But this factor is small, just 2 for b = 2, meaning we basically get O(b^d).

Ratios:

b = 2: 2 b = 3: 1.5 b = 10: 1.11 b -> \infty: 1

So even though we revisit, it turns out to be not be that bad. Why? Because the last level of a tree growing exponentially dominates the node count. It’s that last b^d factor that matters

the most over the long run.

IDDFS wins the blind search contest, taking completeness, optimality, linear space complexity (Korf invented it), and basically the same exponential time complexity.

### Lemma: number of nodes in a search tree

Number of nodes for branching factor b and depth d:

N(b, d) = 1 + b + b^2 + … + b^d

Multiple by b on both sides:

b + b^2 + b^3 + … + b^(d+1)

Then subtract

b(N(b,d)) - N(b,d) = b^(d+1) - 1

Rearrange to get:

N(b,d) = b^(d+1) - 1 / d - 1

Approximate out that the - 1 to get:

b^(d+1) / d - 1

And again apprximate out a b^1 to get:

(b/(b-1)) b^d

## Repeated States

Simplest way: check if the children we’re generating for a state

are repeats of our parent. Then, undo the generation. I.e., the boats problem has the option of sending that same person back across.

Another way: check for repeats along the path from a state back to the root.

The most involved way: check for no repeats everywhere. To do this, we use a closed list, where we put every expanded node onto a list. But we can't keep simply say no to expansion upon seeing a node again, as its children may contain a better solution. Without careful bookkeeping of optimal solutions, we lose optimality.

Even without checking for repeated states, we didn't lose the properties of our blind searches, but in practice, we would try to account for repeated states.

Of course, keeping a closed list means another space sacrifice. The fringe is sometimes called the open list (open for expansion). This is no longer tree search, but graph search. Take a grad class if interested.

## State space

Besides our search tree, which encodes our states and possible actions from each, we can also have the idea of a state space.

It's a graph, one node per distinct state. Then, edges are drawn from one state to another if a valid action exists to go from one to the other.

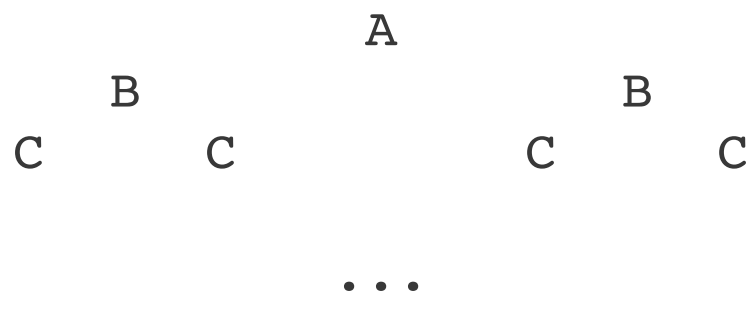
Yet, even if a state space is finite, its search tree may be infinite.

Example state space:

```
A => B => C => D
```

Where there are two actions from A to B, B to C, etc.

Search tree:




Despite a state space liner in its number of nodes, we have a search tree exponential in its number of nodes.

Next homework will have us implementing these searches and solving one of the problems via defining its search formulation functions.

Next time: heuristic/informed searches.

# Discussion 2: 9 October 2015

## Search Formulation

Overview: we re-formulate our problem into a **search problem** that can be plugged into a **search engine** for solutions. This engine can use different **search strategies**.

Usually, we transform the problem into a set of states, and we

especially care about the **initial state** and **goal state**. We then identify possible **actions** that take us from one state to another.

Example: 4-queens problem. Places 4 queens such that none are attacking each other. Here, an intermediate state is a partially-placed board, i.e., only some of the 4 queens have been placed.

Action here: place a queen on a tile. But not all states are **valid**, i.e., allowed by the rules governing our problem. Some states are invalid, which constraints which actions we can take. Actions themselves can be **applicable**, i.e., usable in a particular state (won't go into a wall), but may still lead to an invalid state (queens left attacking).

## Search Tree

Each node is a state and each branch an action. The root node is the initial state, and a goal state is some other node in the tree. A path through the tree shows a series of actions and states, hopefully leading to the goal state.

The leaf nodes are called the **frontier** or **fringe**, the outer edge of our current search, not yet **expanded** for children **generation**.

We say that we *expand our search frontier* when we pick a frontier node and:

1. Check if it is the goal state. Done if so.
2. If not, generate its valid children, the legal possible next states.

## Search Engine Functions

The three functions that need to be given to a search engine to formulate a search problem:

- initial_state(): return the initial state
- is_final_state(): check if a given state is the final state
- successors(): generate the valid children states for a given state

## Search Strategies

Now that we have our search tree, we need systematic methods for expanding our search frontier. This week, we learned the category of blind/uninformed searches.

Keep in mind that node generation order is different from node expansion order. We generate children, but may pick a different one along the frontier for expansion.

### Properties

1. Completeness: is a solution guaranteed (if one exists)?
2. Optimality: is the returned solution the best one?
3. Time complexity
4. Space complexity

#### Breadth-first Search (BFS)

Expand the shallowest node along the frontier.

Complete, optimal, O(b^d) for space and time

#### Depth-first Search (DFS)

Expand the deepest node along the frontier.

Not complete for infinite trees, not optimal (it could go past a

shallower solution without realizing it), O(b^m) for time complexity, O(bm) for space complexity.

Recall:

- b: branching factor, the number of children for each noden
- d: depth of the optimal solution
- m: depth of the tree (d <= m)

**Depth-limited search**

To avoid the completeness issue of DFS, suppose we knew an upper bound on the depth or cost (whichg gives us depth indirectly from knowing the cost of actions) of a solution?

Then, we can limit our search depth L to that bound.

Now, we're complete for solutions with d <= L. Not optimal though, as we could have shallower solutions we don't know about. Now, our space complexity is O(bL) and our time complexity O(b^L).

**Iterative Deepening Depth-First Search**

We perform DLS for L = 0, 1, 2, …, which lets us hit shallower solutions first. We can see this as performing DFS on progressively bigger trees. We revisit nodes, but that turns out to be a constant factor of DLS' time complexity. We no longer know or assume d.

Now, we're complete and optimal, with time complexity O(b^d) and space complexity O(bd).

# Lecture 5: 12 October 2015

Today, we learn about heuristic searches! They require that know more about our problem though.

Wednesday will cover CSPs, and Monday will go over game-playing. Then we'll move on to knowledge representation and reasoning.

From before, we said that BFS can be extended with a cost to each edge.

See the map example from book for the big graph. See picture (add link here). Map corresponds to

## Uniform Cost Search

Small graph example:

```
          [A]

     1              10

[S]  5    [B]      5    [G]

     15             5

          [C]
```

Starting at S (start) and ending at G (goal), the cost of a path is the sum of the weights of all the edges taken.

*Suppose we take the lowest-cost path each time.*

If a frontier expansion doens't work out, we go back to the last one. It's a queue-based traversal again.

S -1-> A -10-> G (11)

Another path:

S -5-> B -5-> G (10)

Next, we have the frontier G (11) and G (10). Expand G(10), which is the goal, so we stop, which must also be optimal.

Solution: A -140-> S -80-> RV -97-> P -101-> B (total: 418)

We're assuming we have the frontier stored and sorted by weight. Recall that last time we avoided checking goal state during generation, as we could lose optimality.

That's seen here, as we generate G after S->A, even though S->B->G has a lower cost. By checking as part of expansion instead, we can check the whole new frontier.

We assume everything has a positive edge cost. Zeros and negatives complicate things up. We're still not worrying about repeated state detection yet.

Compare with our human search: we wouldn't take clearly suboptimal pahths.

UCS is optimal, but not efficient time and space-wise

## Properties of UCS

This is a blind search method. It only looks at the past. How much did it cost me so far? It scores the total history so far. While we get the optimal solution, it may do a lot of unnecessary work.

1. Complete? Yes.
2. Optimal? Yes.
3. Time complexity? Exponential, because BFS is a special case of UCS (where every cost is 1 and we had O(b^d)). Here, we're not sure what depth we have to go to as it depends on the depth of the optimal solution plus the minimum cost of the edges. c* is the cost of the optimal solution. Epsilon is the minimum cost of an edge. Details not too important.
4. Space complexity? Also exponential.

## Greedy Search

This is a heuristic search method.

The left-side column of numbers is the straight-line distance. This is the distance as the crow flies. New search strategy: expand the node with the shortest **straight-line distance** to the goal. We assume we have a table of straight-line distances from each node to the goal node. This gives us the solution

A -140-> S -99-> F -211-> B (total: 450)

But this is not an optimal solution, though it was much more efficient.

That said, we assumed here that we knew the straight-line distances. That's specific to our problem, but the idea is not.

Suppose `g(n)` is the cost from the initial state to a node `n` in the search tree. We can get this from traversing and adding up costs.

Define `h(n)` as our heuristic: estimate of cost from `n` to `goal`.

It’s usually lower than the real cost. This was our straight-line distance infromation.

### Properties

1. Complete? No. It may oscillate between a dead-end and an initial state for example. N -87-> I -92-> V -…-> F. But it goes to N, then back to I, then back to N, etc. This doesn’t happen to UCS because we add up the costs for a state in the search tree. Build the tree and add up, and we see that’d we’d pursue V soon after trying N first.
2. Optimal? No.
3. Time complexity? Depends on our estimate h(n). Could be as bad as any other strategy if h() is a crappy estimate.
4. Space complexity? Same dependence on h().

We’ll formalize efficiency analysis based on h(). Note that h() is also often called the **heuristic** of a search strategy.

Greedy is the contrast of UCS. It looks at how much closer a prospective node gets us to the final node, ignoring its cost. It looks at the future.

## A* Search

So the obvious solution is to combine the two. Add together g(n) and h(n), and minimize this `f(n) = g(n) + h(n)`. g(n) comes from the problem. h(n) is a given estimate specific to a problem. We’ll prove that if h() satisfies a property, we’ll get everything we want (A* requires that).

Cost = paid so far + expect to pay.

Notice that A* will avoid suboptimal solutions and always pick the better one, even if it sees a possible other solution along the way, ASSUMING h() is good enough (to be formalized).

## Properties

1. Complete? Yes.
2. Optimal? Yes.
3. Time complexity? Depends on h(n)
4. Space complexity? Exponential

## Admissibility of h()

We want our h() estimate to not over-estimate. For example, h(goal) should be 0, and should never exceed the actual optimal cost.

```
h(n) <= optimal cost for going from n to a g
oal
```

If this bound exists, then h() is said to be **admissible**, and our algorithm becomes the optimality king A*. If we do overestimate (and are not admissible), then we risk skipping over the optimal solution(s), because we'll think that it costs more than it actually does.

But how hard is it to get such an h()? But wait, we could just make h(n) = 0 for all n. However, this means f(n) = h(n) + g(n) = 0 + g(n) = g(n). This means "A" *becomes UCS, with all its aimless wandering. The higher h(n) is, as long as we don't overestimate, the better the performance of A*. It may vary from node to node, affecting performance.

A* is super useful! It picks the next node to expand based on the

lowest f(n).

## Proof of A* Optimality

To prove that A* will bypass a G2 to get to a G, we'll examine the case where there is already a goal node revealed and some other node exists that would actually eventually lead to the optimal goal node (with optimal cost):

```
f(G2) = g(G2) + h(G2) = g(G2) > c*
```

Where c* is the cost of the optimal solution.

And in general (by assumption of admissibility of h()):

```
f(n) = g(n) + h(n) <= c*
```

And therefore:

```
f(n) < f(G2)
```

And since A* picks based on f(n), it would pick the other node, not the already-revealed goal node, leading to an optimal solution.

Is straight-line distance admissible? Yes, because that's the shortest distance possible between two points. Any other path is the same or greater in length.

## 15-puzzle and A*

What should our heuristic be? It should, given a board, spit back an estimate of the number of moves required to get to the goal state.

One heuristic: the number of tiles out-of-place. Will this overestimate? No. It's actually a nice lower-bound, as everything out-of-place will eventually have to move to another place. Every misplaced tile is assumed to require at least one move to position winningly.

Another: look at how many horizontal and vertical moves are required for each tile, and then add up all of these. This takes care of the oversimplification that each misplaced tile only takes one move to correct.

```
5  4                    1  2  3
6  7  8      =>        8     4
7  3  2                 7  6  5
```

h1: 7 h2: 4 + 2 + 2 + 2 + 0 + 3 + 2 = 18

h2 dominates h1, and is admissible, as again, we have this minimum number of moves. This is the **Manhattan distance** heuristic, which comes up with problems on a grid.

## Performance Comparison

We'll quantify in terms of number of nodes expanded, not time, which is implementation-dependent, rather than an intrinsically better algo.

d IDDFS A*(h1)* A(h2) b* (IDDFS) b*(h1)* b(h2)

2 10 6 6 2.87 1.48 1.45 4 112 12 12 2.80 1.34 1.30 6 680 18 18 2.79 1.38 1.22 8 6384 39 25 . . . 10 47127 93 39 . . . 12 364404 227 73 . . . 14 3473941 538 113 . . . 16 * . . . . . 18 * . . . . . * . . . . . 20 * 7276 676 . 1.47 1.27 24 * 39135 1641 . 1.48 1.26

# Lecture 6: 14 October 2015

// Only about half of the class is here today, // due to the career fair.

## Effective Branching Factor

Suppose we know that the depth of our eventually found optimal solution was d = 10, and N = 47127 nodes expanded. Suppose that the tree has uniform cost and branching factor b, *what's b*?

Well, N = b^*0* + *b*^1 + … + b*^d

Solve for b. *Interestingly, b* is relatively constant across depths. It gives us a way to quantitatively compare search strategies. b* is basically what the branch factor is, if it were uniform for a certain number of nodes and solution depth. Through it, we get a measure of how much we branch off from each state, and thus how many possibilities we have to explore before we can find the solution.

NOTE: There is research on automatically generating heuristics. One way: relaxations, where we drop some constraints, solve that version, and then find a lower bound to our wanted solutions.

What if h1 and h2 do better than each other for particular applications?

h2 > h1 sometimes, and h1 > h2 sometimes.

Simple, take h(n) = max[h1(n), h2(n)]. Take the largest one at each point. As each was admissible, so too must the larger one of them for any state, and thus this new h(n) is admissible.

But what about the cost of computing h(n)? g(n) is just summing as we go. h(n) though could be anything. It needs to be one that can be computed efficiently. For example, the 15-puzzle heuristic had us computing how many tiles were out of place, etc. Some heuristics can be tabulated (and thus later looked-up efficiently) beforehand, but not all (as there may be too possibilities; even 15-puzzle runs into this).

NOTE: An advanced idea called **pattern databases** preserves our ability ot look things up by tabulating only a subset of the possibilities. Here, for example, we would tabulate only part of the board (say 4 tiles at each state). From this partial table, we can still get useful heuristics.

What would be a perfect heuristic? One that gives the exact cost. Use that with A, *and it just zooms toward the solution. How do we get that? We could give our current state to UCS to get h(n) exactly, and then A* would proceed directly to the solution, but each step would take forever. Again, we see the trade-off between efficiency and optimality.

## IDA*

Also, there is a combination of IDDFS and A*. Do DFS for a solution up to cost blah, then a higher one, and so on. This gets us optimality (as we find the lowest-cost solution first), but the tree traversal may now look weird (less uniform). To get the next

layer, we can peek ahead.

BUT, unlike in IDDFS where the multiple iterations didn't hurt us (because the last layer doubled the number of nodes, which dominated the number of preceding nodes, and which in the end gave us a constant factor difference in time complexity), we don't know how many layers of cost (called **contours**) (contours group the states by the cost to reach them) we have here. It depends on the search tree. Shortest-path turns out to have too many contours, but the n-puzzle problem is solved using this.

## Constraint Satisfaction Problems (CSPs)

Some examples:

- Map/Graph Coloring (on board: map of Australia, with provinces): Give every province a color, under the condition that no two adjacent states can have the same color. This is an NP-complete problem.
- n-queens problem: Place queens on an n x n board such that none are attacking each other.
- Satisfiability problem (SAT problem): Find a set of variable assignments such that all the expressions are satisfied. Also NP-complete. SAT is special, because it was the first problem to be shown NP-complete, and everything else in NP can be reduced to it. So that let's pick one problem and try to solve that, then just reduce everything else into SAT. There are many SAT solvers producd over decades of research into the problem. this lets us take NP- complete problems and reduce them to SAT, then throw it at a solver to get a pretty good result, without having to invest a lot of effort into producing a specialized solution to a problem

that is by definition a very hard one.

- The problem where we assign numbers to letters, and then have numeric words like one and one giving two, with the numbers also adding up correctly. (I forget the name for this problem).

We can reformulate this for search:

- A state is a partial assignment of values to variables
- Initial state: no state has anything assigned, i.e., the empty assignment
- Final state: all variables assigned, with all constraints satisfied

Notice that we can't easily find the goal state, but we can easily check if one is the goal state.

Each one of these problems has a number of variables, and we can solve them by making assignments to them such that the constraints are met.

In the map coloring probem, the variables are the provinces, and their values are their colors. The constraint is that no two adjacent provinces can have the same color.

In the n-queens problem, the variables are the rows, and the values are the column of the placed queen. Some math needed to check for attacking, which is the constraint. Another formulation flips the rows and columns. Or: every queen is a variable, and the value is their location. We can even make every queen go into a particular column/row, and then assign the rest (bypassing need for (row, column) checking)

## Making a Search Tree for CSPs

CSPs: variables, values, and constraints. That's all we need to make them into a search problem. Suppose we have n Boolean variables that need to be assigned values. How many possibilities? 2n for the first level. Now, the next level has n - 1 variables, as we assigned to one variable (choosing a state to proceed to means giving it a value; moving forward means making more assignments). Then, we have 2(n - 1) possibilities for the second level, all the way to 2(1). Adding up, we have 2(n + (n-1) + (n-2) + …) children, which is n!*2^n. That's horrendously bad.

But notice that we're constantly repeating states as part of expansion. Why? Because different branches just have us assigning values to variables in different orders. We can eliminate this by *imposing an assignment order*.

Suppose we have { x, y, z } that take on { t, f }. We'll make it so that we always assign x, then y, then z.

```
          x = 0          x = 1
y = 0       y = 1    y= 0       y= 1 z = ...
```

Follow the left branch of the above tree: given that we assigned 0 to x, what do we assign to y, and then z?

Now, we have 2^n children, the number of possible assignments exactly, no more, no less (recall that a bit-vector of n bits can take 2^n possible distinct values, as each has a choice of 0 or 1). This is the standard search tree for CSPs, not the previous factorial disaster.

## The Path to Backtracking Search

So what search strategy should we use?

Let's think about when certain strategies are applicable. We use IDDFS if the depth of our solution is not known and we want linear space complexity. We use LDS if depth is known or if the tree is infinite, to avoid infinite loops. Neither apply here though, because the tree is finite. We're limited by n, as we have 2^n nodes. Not only that, all of the solutions are at the very last level, after we've assigned something to everything. At that last level, it's easy to check if a node is a solution. What about A*? Well, cost is irrelevant here. Because here the number of "moves" we made doesn't apply. In every case, we're making assignments to n variables. Thus, A* (and greedy) is eliminated.

So, we use DFS, as optimality is not an issue (all solutions are at the same depth) We also have completeness as it's a finite tree, we get the DFS bonus of linear space, and every strategy has exponential time here, so whatever. UCS is usable as well, but it's exponential in space, which is not practical for a large- enough number of nodes.

But we don't usually blindly go all the way to the bottom, and then check if our constraints are met. Rather, we can check for contradictions with just a partial assignment, and then avoid going further down wrong paths. We **backtrack** to an earlier state to continue diving.

## Factors that Affect CSP Search

1. Variable assignment order. Just choosing a different order can

make a huge difference. It may be faster to check against our constraints when variables are assigned in one order vs. another. The sooner we make partial assignments that will bump up against constraints, the sooner we can prune off more of the tree: . . . vs . . . . . NOTE: There is another idea called **constraint propagation**, where we check the implications of our constraints, e.g., x = y, y = z implies x = z. This is not possible in general, as we can go on forever exploring implications, killing our efficiency.

An example: The assignment order x,z,y vs x,y,z to satisfy x = y, y = z. x,y,z turns out to be faster.

In general, we do not know how to order variables, but there are rules of thumb for specific problems.

One heuristic: *choose the most constrained variable*. This makes sense, as it's easier to run into contradictions with it.

Another: *choose the most constraining variable*. Choosing that variable eliminates possibilities for other variables. Example: pick the province bordering the most other ones.

1. Value ordering. We can affect to which side DFS goes first, since most implementations go left-to-right given multiple choices at the same depth. This permutes the tree, rather than pruning it, as we're merely changing the visitation order rather than eliminating possible states. Suppose under x = 0, there is 1 solution at the bottom level and x = 1 has 4 solutions there. We see here that the density of solutions beneath a value choice can differ, so we can make more heuristics to decide the best values to assign (and thus check) first (recall that we're gonna have a

fixed set of possible values that our variables can take on)

NOTE: Here, “heuristic” is an overloaded term, used in its general sense: a rule of thumb. It may not always work, but it works probabilistically, etc. Again, no general solution for determining variable ordering.

1. Detecting failures early (reasoning)
    - Foward-checking
    - Arc consistency

ASIDE: How does reducing to SAT allow us to beat specialized work on a particular problem? It doesn’t, BUT, there has been so much work on SAT that we get better results (sooner!) by throwing a SAT solver at a problem rather than investing decades into the problem specifically. If we have the resources, then we can beat the SAT solvers, but otherwise, just use SAT. The new paradigm: in other computational complexity classes, we again build a solver for a problem within a class, and then reduce the other problems to that. This lets us skip the arduous labor of building problem-specific solutions.

Darwiche: see beyondnp.org for more on this new paradigm.

Next time: finish up this, then game-playing, then a brand-new area of AI (knowledge representation and reasoning).

# Discussion 3: 16 October 2015

## Uniform Cost Search

Draw the graph corresponding to this adjacency list, where parantheses indicate the cost from LHS node to RHS node:

S: A (1), B (4) A: B (2) C (5), G (12) B: C (2) C: G (3)

This is a directed graph.

Initial state: S. We expand it:

- Goal? Nope
- Generate: We now have A and B as our frontier

Since it's cost of 1 to A, and 4 to B, we choose to expand A next.

Search strategies answer the question: which node should be expanded next?

g(n): cost from initial state to state n. It's our sole selection criteria for UCS.

Next, we expand A, which gives B, C, G.

To get their g(n), we add up the cost from S->A (1), then A->B (2), which means B has g(n) = 3.

Similarly, C has g(n) = 6, G has 13, and C has 6. When costs are the same, we choose arbitrarily (usually left-to-right).

So we expand B, and then we get C with g(n) = 5, and then examining our frontier again, we'll eventually expand a goal state (of which multiple may appear, indicating multiple paths possible to the goal).

Properties:

1. Complete? Yes.
2. Optimal? Yes.
3. Exponential time and space complexity

## Greedy Search

// Re-run based on same graph as above

Greedy search chooses the next node to expand based on a heuristic h(n), which is an estimate on the cost from a state n to the goal state. It chooses purely based on this outside information, and ignores any cost associated with a particular path.

h(n) for goal is 0, as we right on the goal then.

Properties:

1. Complete? No, it can oscillate in a dead-end.
2. Optimal? No, it ignores costs.
3. Time and space complexity depends on h()

## A* Search

A* gets the best of both worlds by combining UCS and Greedy. We choose our next node by picking the smallest f(n) = g(n) + h(n).

g(n): cost incurred so far to reach this state h(n): estimated cost to reach goal state

Properties:

1. Complete? Yes, as it considers g(n) as well as h(n), preventing dead-ends.
2. Optimal? Yes, assuming h(n) is admissible (does not overestimate, i.e., h(n) <= actual optimal cost always), preventing it from skipping over optimal solutions
3. Time complexity: depends on h(n). Exponential in worst-case, polynomial in best.
4. Space complexity: can be exponential in worst-case.

If we have two different heuristics that work well for different graph examples, just take their max to get the best heuristic at all times. h3(n) = max[h1(n), h2(n)]

## Constraint Satisfaction Problems

A general class of problems has variables that take on values limited to particular domains, subject to constraints.

Specific example: the n-queens problem, where each queen is a variable, with its coordinate values limited to its assigned row, and all of them constrained to not attack each other.

To formulate this for search:

- Initial state: empty assignment (no variable holds a value)
- Goal state: all variables are assigned, respecting constraints
- Action: assign a variable

Convert CSPs into this form, and we can use search to tackle them.

Things that affect the effectiveness of our search for CSPs:

- Order of variables for assignment
- Order of values in the tree
- Failure detection

# Lecture 7: 19 October 2015

## CSPs and Exploiting Problem Structure

Review: we formaluate CSPs as a search problem by treating states as a partial assigment of values to variables, subject to a set of constraints. The goal state is a full set of assignments that is valid under the constraints.

To make this better, we saw that order of assignment and values could get us to solutions faster. Some variable and value ordering heuristics: the one that has the most constraints, the one that constrains the most other variables, etc.

// Darwich gave no example of value ordering heuristics

We also saw early failure analysis, where we check as we go, detecting contradictions of our constraints before a full assignment. That gave us backtracking search, where we go back to a parent state to try a different path. Darwiche is saying we can do early failure detection during expansion or generation. If the latter doesn't affect correctness, then generation is better, as we never put it on the search tree.

Now, we want to improve our solution-finding speed even more, by exploiting problem structure.

Two new techniques: forward searching and arc consistency.

## Forward Checking

Again, a map of Australia and its provinces, labeled S1 through S6. Each variable can take on the value RED, GREEN, or BLUE (RGB).

The idea: each time we make an assignment, we go back to our list of values, and check if that logically eliminated possible values for another variable. We put a cross or some other marker to denote this. If at any point any variable loses all of its possible values, then we have an incorrect assignment.

```
R    G    B S1        x    x S2  x S3  x    x
    x   <-- Invalid assignment. We backed ou
rselves into a corner. S4 S5 S6       x    x
```

Here, ‘x’ indicates an eliminated value. Each adjacent province must have a different color, so that eliminates the choice of the color that we just assigned to its neighbor. Notice we also lose the other choices of color for a particular province, as it can only take on one color.

Notice that despite this clear dead-end, we have not violated our constraint yet (no adjacent provinces may have the same color). Forward-checking has allowed us to see ahead. We back-track earlier and higher up in the tree. The book gives some numbers for comparing these combinations of these techniques. There are other optimization methods.

## Arc Consistency

Every time we have a constraint between two variables, we

imagine/store a directional arc between them. It may be a bidirectional constraint, so two arcs then.

x -> y <-

We say we have arc consistency if and only if:

> For every value of x there must exist a compatible value for y.

“Compatible” means not violating the constraints. We’re assuming here that it’s constant-time to check that. If we index that well, that’s doable.

Example, x can take on G or B, Y can take on R or B.

x=G => y=R

x=B => y=R

But if Y is already constrained to not be able to take on R, then it’s inconsistent, as it violates the definition for when x=B.

To make arcs consistent, we can take out one of the values for one of the nodes.

Initially, every variable has all of its possible values. Then, we pick a value for a variable. When we do so, we’ve eliminated some possible values for other variables. We check the incoming arcs and try to make them consistent again.

Example: We assign y=R. We eliminate R as a possible value for x

to make the arc from x to y consistent. If at any point we lose all possible values for a variable, we know we have an invalid solution. Backtrack.

S1 <- S2 -> S4 |__|___| | S3

Arc consistency is just fancily keeping track of the relationships between variables and then noticing dead-ends. It's smarter than the mechanical tabulation of forward checking.

Here, all arcs are bidirectional, at least for map coloring, since the adjacent coloring constraints applies on both sides of a border.

n = total number of variables d = total number of values a variable can take on. Size of *domain*.

Number of arcs depends on number of constraints? Worst case: every pair of variables is constraints, n^2 arcs.

We'll have: n^2 arcs d^2 checking steps per arc

O(n^2 * d^2) complexity assumes we process each arc once. However, each arc may actually be processed more than once. Let's try to bound how many checks may happen. Simple: the number of variables:

O(n^2 * d^2 * d) = O(n^2 d^3) This is n^2 time (d^3 is a constant).

Detecting and fixing arc consistency is what matters here, not the complexity.

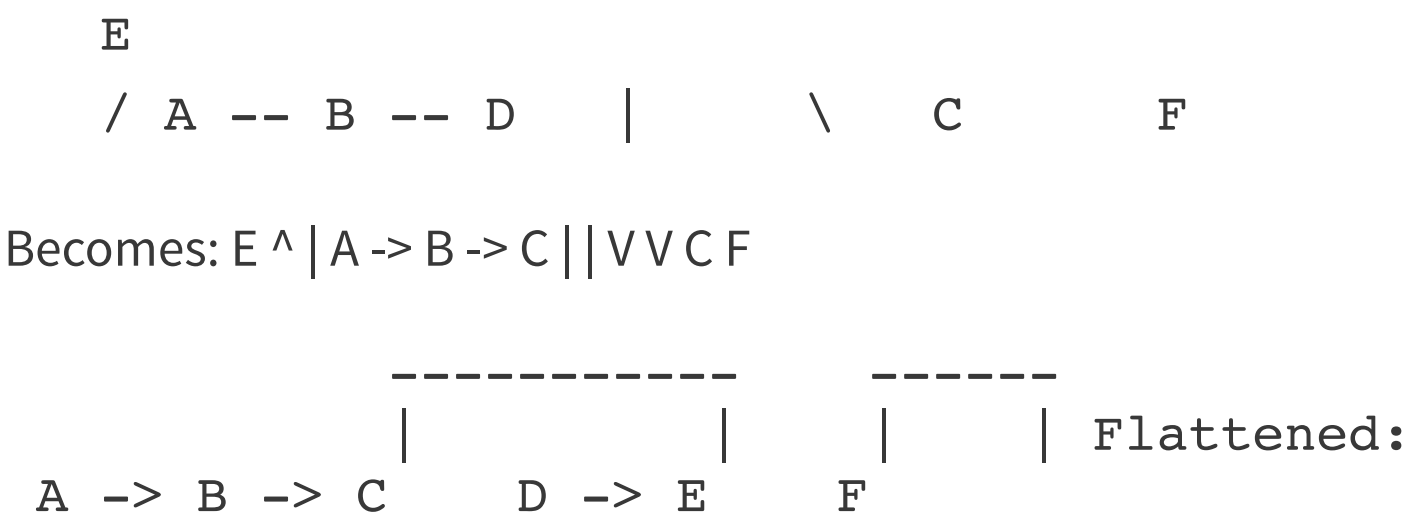
## Constraint Graphs in Polynomial Time

Now let's make a **constraint graph**, which is undirected in its edges. We make an edge between two states if they constrain each other.

*If the graph turns out to be a tree* (no cycles), it's solvable in polynomial time, without need for backtracking search. It's no longer an NP-complete problem. CSPs are in general hard, but this subclass is doable in p-time.

Method:

1. Pick any node in the tree
2. Look at the outgoing edges, and give directions to them, pushing outward. This is "directing an undirected graph".




Now, we work backwards, checking arc consistency and eliminating values to fix consistency. O(n d^2), with every arc consistent in the end. Now, sweep forward, so pick a value for A. Then, since the arcs are consistent, there must also be a compatible for B, C, etc.

Overall complexity: O(n d^2). d^2 is a constant, so this is linear.

// Who here has done 180? *All of us raise our hands*. // Stella: "It's a pre-req.""

A lot of problems suddenly become solvable in polynomial time when we abstract them into graphs that turn out to be trees.

BUT we must have a tree-structured CSP problem. But how might we use tree-CSP solver box to tackle problems that do not result in trees?

### Solving Non-Trees with Trees

What if we move nodes such that it becomes a tree? There's the cut-set problem, usually where we try to choose the fewest number of nodes such that it becomes a tree. We call the tree-box multiple times, with all possible values for each variable. There is a cut-set algorithm that can do this.

We apply tree-box to the cut-set for every possible variable value?

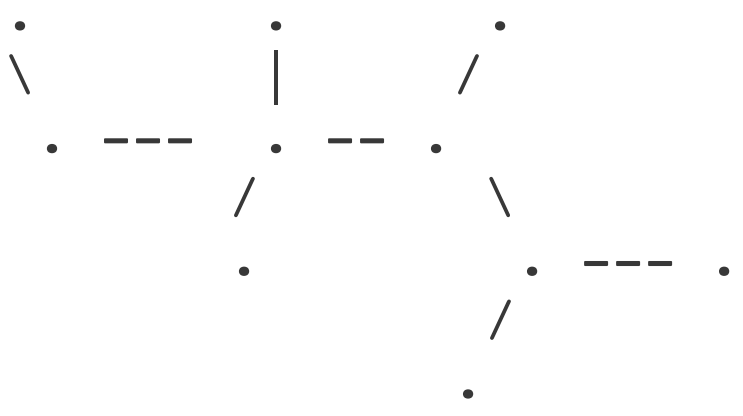
The cut-set is a set of nodes. Graphs differ in how many nodes need to be removed for them to be made into a tree. It is okay for trees to be disconnected, i.e., some nodes can be loners. There may be multiple cut-sets of the same size, and they can be prioritized, but too advanced for us; heuristics can be invented for them. More impactful: color choice order, e.g., which colors do we try first?

## Incomplete Methods

We'll end up with three ways to solve CSPs: backtracking search (which is exhaustive, and thus exponential), systematic early failure detection (forward checking or arc consistency), and now incomplete methods (local search).

## Local Search

Local search: pick an assignment. Is this your card? No, let me modify it. How about now? No, let's try again. The search space is no longer like a tree, but rather, it picks among the next possible modifications, giving a graph like this:



Concrete example: 4 queens problem. Invalid assignment? Move a queen within a column, and try again.

x x x x => x q x x => x q x x x q x q x x x q x x x q q x q x q x q x x x
x x x x x x x x x x q x

Unlike our previous algorithms, we might return a "solution" even if violates constraints, because we may have a problem where there is no solution, and we want the solution with as few violations as possible. We have a scoring system. Here, for example, we may count the number of possible queen-on-queen attacks, and we want the assignment with the lowest such count.

We need a scoring function that that can take a state and generate the score. We need a neighbor generation function.

1. Start anywhere
2. Look at neighors and pick one with the best score

However, we may get stuck in parts, end up with suboptimal solutions, etc.

So why use this? It becomes very useful when augmented. And notice that we need only store one state in memory.

**Random Restarts/Hill Climbing**

But what if we're waiting for like 10 hours. Here's one way to counteract that:

1. Run for a set number of expansions.
2. If nothing after that many, randomly restart.

This is hill climbing, where we try to get to the top of the highest peak. But if we're dropping in flat lands, we won't find the peaks. Where we begin matters, thus the random restart to a new spot. We find a local maximum rather than the global one, etc. Local search stops when we it finds a node better than the ones around it. Often, hill climbing restarts randomly again anyway, as it knows that it may not have found that best solution.

NOTE: Local/Incomplete are just that: incomplete. No guarantee against infinite self-looping or completeness in finding a solution, and yet they can become very efficinet.

Variations of local search:

- Hill climbing
- Simulated annealing

Where they differ: instead of getting and scoring neighbors, we might flip a coin between neighors, and we go there if it's better than current state, but if it's worse, we may still want to go there,

but we're only more likely to do that if it's early on in our search (simulated annealing). All of them do something beyond choosing the best-score neighbor. Empirically, they can outperform vanilla local search, even though the only differ slightly.

Again, we're answering the question: where do we go next?

Some strategies turn out to be exhaustive (based on how the probabilities work out, i.e., no zero probability of visit for any node in graph), but no guarantee on time.

### Properties

- Constant space (scales like hell)
- Need to pick right variation of local search for a problem (need to experiment) to solve it quickly

# Lecture 8: 21 October 2015

// Midterm two weeks from today, closed-book, closed-notes. Delayed to 11/4, // unlike original 11/2.

## Hill Climbing

Look at neighbors, and go to the one with the best score, if it improves on our current score. If no better neighbor exists, then stop, and declare a local maxima.

A variation: what if the best neighbor next to us has the same score as us? Now, we have the choice to move or stay. If we do move, it's called a **side-move**. In n-queens, allowing side-moves

can improve local search success from ~14% to ~90%.

## Simulated Annealling

current is current state

schedule(t) is a function with exponential decrease curve, like temperature over time, where the annealling name comes from. The temperature affects the probability that we'll move to a randomly selected neighbor; it goes down exponentially over time. Little t is time

```
for t = 1 to \infty do
    T = schedule(t) // Get current temperatu
re
    if T = 0
        // Very end boundary, when we conver
ged, and hit end of search
        return current


    next = a randomly selected neighbor of c
urrent
    delta = value(next) - value(current) //
Difference between current and next

    // If better, go there
    if delta > 0
        current = next
    // Even if not better, go there sometime
s probabilistically
    else
        // The more negative delta is, the l
ess likely
        // we are to go to a bad number
        current = next with probabilty e^(de
lta/T)
```

```
// Same as (1/e^(|delta|/T))
```

Randomization allows us to avoid getting stuck, where deterministic local search can get us stuck in a place. Adding in the probablistic moves sort of makes everything possible to get unstuck.

What we did before this is also called **exhaustive search** or **systematic search**. Hitting everything possible, systematically exploring everything, etc. Notice also that local search has no memory of the past. There is a non-zero probability of visiting every state, so they're complete as time goes to infinity. Some variations keep limited memory, taboo search.

## Two-player Games

Think of any board game, etc.

Sometimes a distinction is made between games that are deterministic or have an element of change, and whether players have perfect or imperfect information

Deterministic: state of game can be determined from information on initial state and subsequent player moves.

Deterministic & perfect information: chess, go, othello

Chance & perfect information: Monopoly, backgammon

Chance & imperfect information: poker, scrabble, bridge, thermonuclear war

Deterministic & imperfect information: Battleship

Chess is too complex for us to start with, even though it drove a lot of research here.

## Tic-Tac-Toe

Intitial state: empty board

X = player 1 O = player 2

We could make a search tree. The first set of children: all of the possible initial places for the X. Then, following that, the possible moves for O.

In the end: not an even bottom level, as we can win early, sometimes we tie with a full board, etc.

---

| | | . . . . X . etc. . X . . . . . . . . . \ \ . . O \ . X . . . . . . . . X . . . O

This is a **game tree**. Let scoring system be:

+1: X wins -1: O wins 0: tie

This is for the leaves, the end-game states.

Now what do we do with the tree? Well, let's try to find a winning sequence of moves. Problem: a path to a winning state is no longer a solution. It's just a trace of a possible playing-out of the game, as the opponent may take different moves; it's a possible outcome.

So what's a solution now? What's the output of our game-winnning box? Simple: "Tell me oh Delphi, what move should I make next, based on the current state of the game?" They're an

advisor sitting next to us the whole time. It's no longer a one-shot wonder.

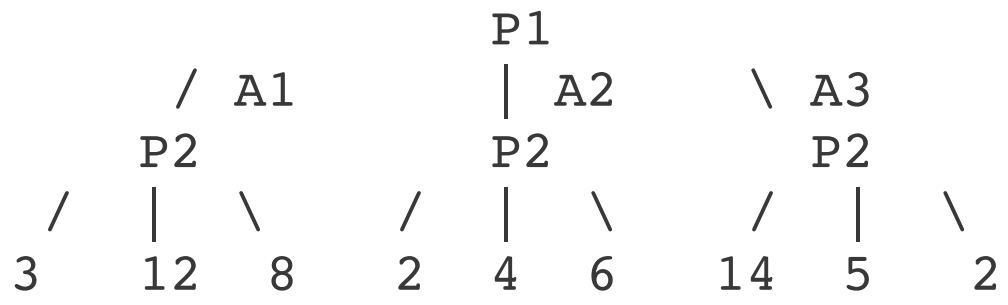
Alternatively, give all of the information to a box, and then it gives us back a **policy** or a **conditional plan**, which is a sequence of actions that tell us what do first, then how to respond to our opponents' moves, etc. That is a one-shot wonder, a master plan for winning or tying in every case.

Back to the advisor box. This algorithm looks at a game tree, and tells us which child of the root to go with next. Given that, we can use it to play (and win!).

No need for domain-specific knowledge thus far.

## Minimax

Abstraction: This is a 1 move (or 2 ply tree).




The numbers are the scores with respect to player 1, so they're trying maximize them, whilst player 2 is trying to minimize them.

One "move" is after both players have executed actions, i.e., turns. Each ply is an action by one player.

A ply is a single action:

Move = 2 ply Ply = 1/2 move

So what action A_i should player 1 choose? The scores are with respect to player 1, so higher scores are better outcomes for player 1.

Some ways we could choose:

- Choose action that leads to highest possible score
- Choose action that leads to highest average end-scores
- Choose action that hedges against opponents' moves, assuming that they choose the best possible one for themselves?

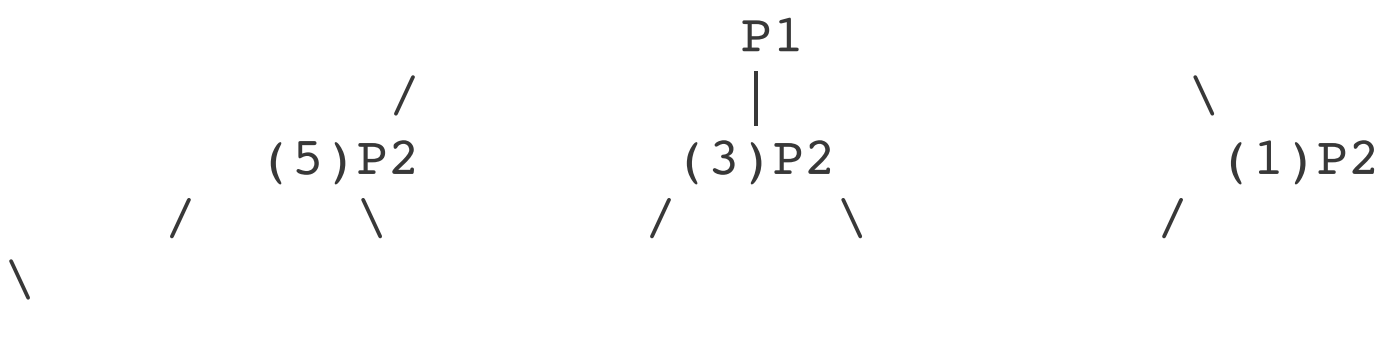
Key insight: we need to assume some sort of **model** about our opponent and their behavior. Is the opponent a **perfect player**, meaning they make the possible best choice for themselves? We often tackle games assuming this. An opponent could also be **effectively random**, picking their moves at random.

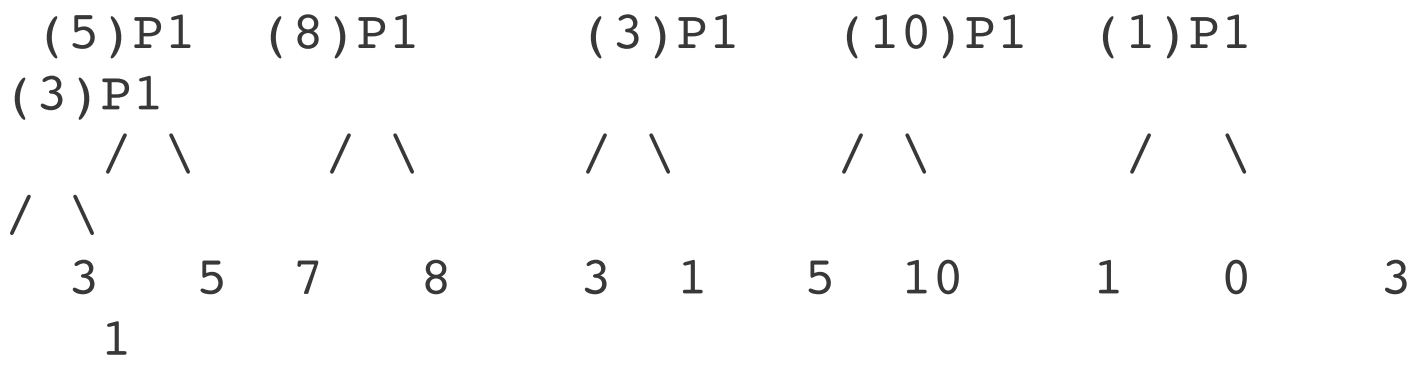
So how shall we choose?

Well, look at the minimum score outcome for each action? Why? Because, if we choose an action that has that outcome, our perfect opponent will work to make sure that minimum score is the outcome.

MIDTERM question: there will be a question like this. Now what?

Now, let's see a deeper game tree example:






We can label the lower level with the max of the outcomes, as player 1 is maximizing, as they have control during their turn. Among those labels, player 2 above it will choose the minimum of those labels, as they have control during their turn. Player 1 tries to maximize their score, player tries to minimize it. Hella clever. It seems like we're playing backward, but rather it's just that player 2 can think ahead and realize what outcomes are possible, and during their turn, they try to minimize player 1's score.

This assumes a perfect opponent (if they're dumb or random, this won't work well). Actual implementation: use depth-first seach, going down and computing scores in linear space, rather than exponential space for exhaustive tabulation. We go down the leaves, then compute score for parent.

How does the DFS process work? We go down a series of actions by the players, and track the score. Once the game ends, that's the value for our leaf node. Once a parent's nodes are filled out, we can label it with either the max of its children (for a P1 turn) or the min of its children (for a P2 turn).

Note: we still need to examine the whole tree before we can give a recommendation for which of the children of the root we should choose. Note that the labels on those root children tell us the end outcome for P1, as P2 will be minimizing. Thus, here the

action we should take is the leftmost one, as it can lead to an outcome of 5, the best among the root children's outcomes.

## Chess and Minimax

Assumptions:

- 100 second time-limit to get advice back
- 10,000 nodes examinable per second
- => 1 million nodes examinable within those 100 seconds

Average branching factor b in chess is b = ~35

That means with b = 35, and b^m space, we can only look at depth = 4. That's how much look-ahead we get: 4 plies ahead.

b^m = 10^6 => m = 4 = 4 plies, the depth of the game tree (root = current board state)

This is equivalent to a human amateur. An expert can do ~8 plies ahead, i.e., 4 turns (4 pairs of player moves) ahead.

What about IBM's Deep Blue? Each level of look-ahead makes a huge a difference. Kasparov and Deep Blue could do 12 plies (or 6 pairs of actions).

First, let's make this work for any time-limit, i.e., derive a depth limit, and we know we can only look at so-many levels within some set time, but we still want an answer back based on that incomplete look-ahead. But without the whole tree, we can't get the exact numbers for that bottom level, as that's what let us figure out a label for each action. However, with chess, there are too many possible actions at any one point.

## Evaluation Functions

Simple: we can approximate numbers for them, and then propagate upward to get min-max scores. Think about how humans heuristically evaluate a given chess board for a player. Looking at the pieces, their positions, etc. we can figure out how well the game is going. We call something that can do this an **evaluation function**, which evaluates the nodes at that last level of the game tree reachable by our min-max algo within the timeout.

Sometimes we know a timeout a priori, but if we may get a "Time's up!" signal at any point, then we can do IDDFS such that we always have an answer ready. We can successively increase depth as long as we haven't been interrupted yet.

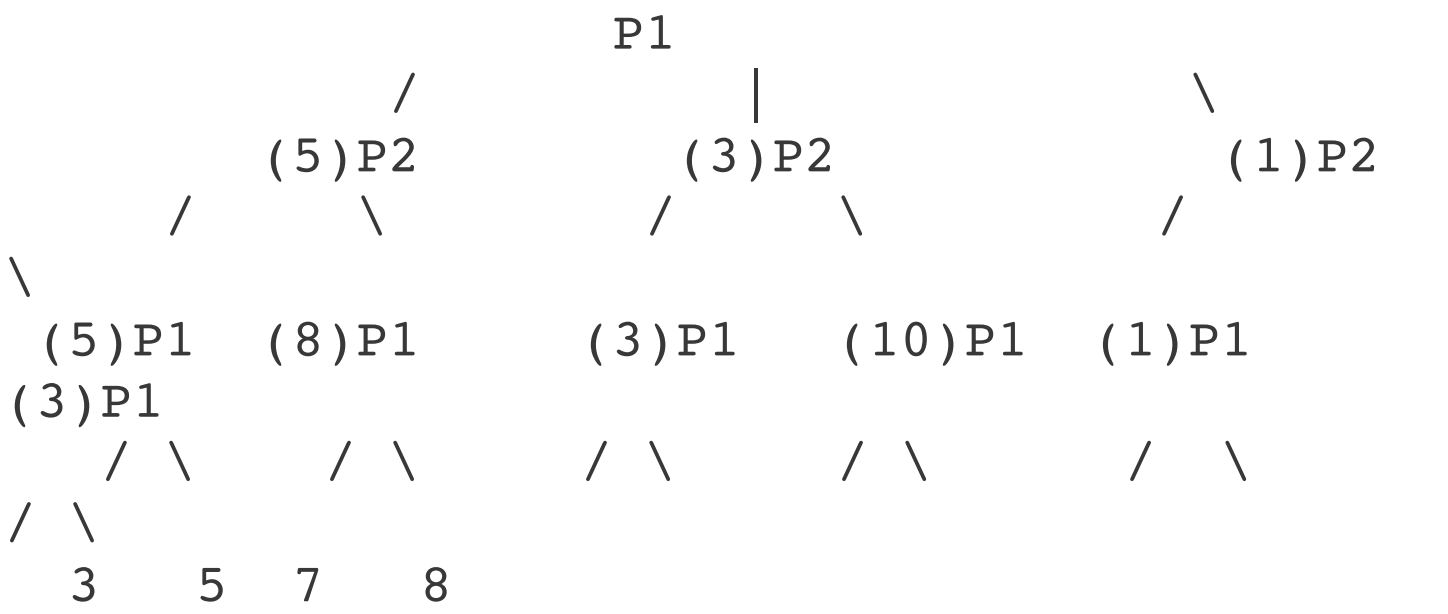
The evaluation function is domain-specific; it has to be. Problem: some things look great up to that point, but one level deeper, and suddenly we realize it's actually a terrible action. Countering that is strategies that don't expand actions uniformly; some will be investigated more deeply than others.

In summary, evaluation functions give back an approximate number for the deepest level of a game tree, which we then propagate back up to get labels that let us recommend actions. Since crucial information may be in the level just beyond, we can investgiate some nodes more deeply than others, keeping us in budget time/node-wise, but gaining better results. If we have an explicit time-out, we can pre-compute to what level we should go, based on depth and branching factor. If we don't know, then do IDDFS, and keep an answer on which action to choose next ready at all times.

## Alpha-Beta Pruning

Wihtout necessarily exhaustively searching the game tree, we can get double the look-ahead depth. Imagine we're in the middle of the DFS traversal, have computed 3 for one parent of a set of leaves, then go down to another leaf of another parent. But we find an outcome score of 2, so we don't expand that player 2 move any more, as we already know that up against a perfect opponent, we can still snatch an end-score of 3, so why investigate a set of moves that we know will lead to a worse outcome?

This is **pruning** the game tree for us, letting us spend that million node budget more wisely.




Going down, keep in mind that player 2 minimizes, but player 1 maximizes.

GO BOTTOM UP to figure out the sequence of moves that players would make, i.e., compute the scores. If we go down a path and we know that min has the possibility of lowering our score, we won't bothering exploring that further.

Here, for example, we prune that second root child after finding

that outcome score of 3 on the bottom level, which is worse than our already-found outcome of 5.

We need two variables to implement this. Joseph McCarthy came up with alpha and beta to track our numbers. Thus, it's called **alpha-beta pruning**.

NOTE: ordering of nodes affects the outcome of the alpha-beat pruning. If we get lucky, we can examine b^(m/2) nodes, effectively doubling our examinable depth within the same amount of time, as we prune so many nodes.

On average, it's 3/4 of the tree, i.e., b^(3/4 m). Minimax is not usually used without alpha-beta pruning.

In summary:

- Need a heuristic evaluation function
- Use IDDFS to compute node scores bottom-up
- Need alpha-beta pruning

Is this still possible for games with chance? In principle, yes, but then we take an average over the possible games, but looking ahead is kinda pointless after a certain number of steps ahead, as they become less likely due to chance.

# Lecture 9: 26 October 2015

## Knowledge Representation and Reasoning

Our set-up:

```
Knowledge Base (KB)
         |
         V Query -> Reasoning -> Answer
```

Two main approaches: reason from logic; reason from probabilities. Nowadays, we use both together.

From Forney: The KB is our *background knowledge*, a set of sentences that are *axioms* about what is true in this world. From these, we can **infer** more terms/sentences. This seems to imply to me that even if our initial set of knowledge isn't enough to answer a query, what we infer in the process can be added on, letting us answer it in the end. That is, even if a query isn't immediately answered by an axiom, we can apply the axioms to winnow down our possible worlds, and then check if our query applies to all of them.

Logic was the first historically; probability is the recent breakthrough.

## Propositional Logic (Boolean logic)

There is more than one kind of logic: there is propositional logic, as well as first-order logic. We'll do both.

Example: logic puzzles from elementary, where we take givens, write it into a knowledge base (called knowledge acquisition or knowledge engineering), then solve.

Another: a grid where an agent moves around looking for gold. But some of the cells have a pit and others with some monster

W.

In neighboring cells, we can detect the presence of these threats: a breeze for pits; a stench for the monster. These are all around the threats.

Our question at any point: is it safe to go to a neighboring cell?

This is fucking minesweeper.

P? . . B P? . A S W

B = breeze S = stench W = monster P = pit A = agent (where we started)

Notice if we are where the S is, then we've eliminated the possibility of a pit in the cell above it, as there was no breeze, but there was a stench, so there must be a monster to the right.

## Syntax

Syntax is a set of grammar rules that tells us what constitutes a valid statement in some logic.

Semantics explains how to interpret logical sentences.

**Propositional/boolean variables** (aka atoms) are the variables P_1, P_2, …, P_n, which are either true or false.

Inductive definition:

1. Every P_i is a sentence.
2. If alpha is a sentence in propositional logic, then so is NOT alpha.

3. If alpha and beta are sentences in propositional logic, then so are alpha AND beta and alpha OR beta. The AND is known as “conjunction”, and we’re “conjoining” alpha and beta. The OR is known as “disjunction”, and and we’re “disjoining” alpha and beta. All of these are known as **connectives**. Example: NOT(P_3 OR P_17) AND (P_5 OR P_16).

Using these connectives, we can define all the others. Memorize this: alpha IMPLIES beta == NOT(alpha) OR BETA

alpha EQUIV beta == (alpha IMPLIES beta) AND (beta IMPLIES alpha)

## CNF and DNF

While we can make sentences of arbitrary complexity, sometimes we restrict our statements for computational reasons, etc.

These are restricted statements are called **fragments**. These are subsets of propositional logic.

**Conjunctive Normal Form (CNF)** looks like this:

```
(A OR NOT(B)) AND (B OR NOT(C) OR NOT(D)) AN
D (E OR NOT(A))
```

Each of the **clauses** connected by the ANDs are disjunctions of atoms or negated atoms. Atoms or negated atoms are known as **literals**. Clauses are therefore disjunctions of literals. We then conjoin all of the clauses. We sometimes call the true form the *positive literal*, and the negated form the *negative literal*.

Tersely: CNF is a conjunction of clauses; a conjunction of disjunctions of literals. If seen as a circuit, then max-depth 2.

**Disjunctive Normal Form (DNF)** looks like this:

```
(A AND B) OR (A AND NOT(C)) OR (NOT(B) AND N
OT(C))
```

Now, we have **terms**, which are conjunctions of literals. DNF then is a disjunction of terms.

We knew these before as product of sums (CNFs) and sums of products (DNF). Those were often written as products (sometimes with dots), with overlines for negation, and with pluses for conjunction.

Suppose we have a solver box for anything in CNF form.

“Do you just accept this restriction?”

“Yeah!”

Systematic conversion turns out to be solvable. But the big question is: is all knowledge representable in CNF? Otherwise, it’s pointless. Formally: does every sentence in regular propositional logic have a corresponding form in CNF? That is, is CNF a **universal representation**, one that is general to represent anything. It turns out that CNF is one, as is DNF. We’ll learn systematic conversion later.

## Horn CNF (Horn clauses)

A subset of CNF, with additional restriction on clauses.

A **Horn clause** is a clause that contains at most 1 positive literal. Note that the rest could be negative, that there could be no positive literals, etc.

(A OR NOT(B)) is (A OR B OR NOT(C)) is not (NOT(A) OR NOT(B)) is

If we restrict our input to be in Horn, then SAT can be solved in poly time. But it turns out that Horn is not universal. The fragment of propositional logic that is Horn logic is considered tractable, i.e., solvable, as compared to intractable.

## Example Sentences

Example grid:

```
    B
B   P   B
    B
```

What's the logical relationship between pits and breezes?

Aside: is KR & R broadly applicable? Sure, to automated verification of programs, etc.

The above grid means that there is a variable P or B to say whether there is a pit or a breeze there, not that there is. In general, P_ij or B_ij for each square.

Example:

// If there is a breeze, then there is a pit // in one of the adjacent squares B => P_1 OR P_2 OR P_3 OR P_4

// True, but useless P_1 OR P_2 OR P_3 OR P_4 => B

// What I think we need P => B_1 AND B_2 AND B_3 AND B_4

But have we captured all knowledge? That's hard to tell. In practice, we get back an "IDK" from an engine, because it was missing info.

## Semantics

Semantics is the meaning of some text. For us, we get some sentences, and then we query:

- Does (blah) imply (blah-blah)? Example: given what we know about the grid, is it safe to go to square blah?

Semantics let us determine when the answer should be yes/no.

- Another one: are alpha and beta equivalent? Example:

NOT(NOT(A) OR B) =?= (A OR NOT(B)) // Yes, De Morgan's

A => NOT(B) =?= B => NOT(A) // Yes, contraposition

- Another: is alpha consistent?

And so on.

| | |
|---|---|
| Example: [ | ], a two-cell grid with either a vacuum or dirt or both in each. |

LV : left vacuum RV : right vacuum LD : left dirt RD : right dirt

16 possible combinations. Each one of these is called a **world**, one way that the world could be.

| | |
|---|---|
| World 1: [vac, dirt | dirt] // Triangle vac, circle dirt in diagrams |

LV = T RV = F LD = T RD = T

We could write this out as a truth table too.

Some sentences, which are our knowledge base:

- Beta1: At least one vacuum: LV OR RV
- Beta2: Not two vacuums: NOT(LV AND RV)
- Beta3: Dirt and vacuum cannot co-exist in the same cell: LV => NOT(LD) RV => NOT(RD)
- Beta4: Left vacuum exists: LV
- Beta5: No dirt in left: NOT(LD)

Then notice that the first two imply exactly one vacuum. The next two say that vacuums and dirt cannot coexist in the same cell. The fourth says the vacuum is in the left cell. The fifth says there is no dirt in the left cell.

It turns out that the fifth is implied by the first four.

In the beginning, we know nothing, and all of the worlds are possible for our problem domain. With each sentence, we go back to the picture and knock down possible worlds. Knowledge will correspond to a subset of the possible set. Syntax will be our sentences; semantics will be our worlds.

Initially: all possible states there Then, beta1 is brought in: there must be at least one vacuum in either cell. We toss out the ones

that have no vacuum in them.

Now, beta2: cannot have vacuums in both cells. Notice that we don't have to care about the interactions between the assertions; we just apply them one at a time.

Beta3: Vacuum and dirt cannot coexist; tosses those that have both.

Beta4: vacuum in left cell => tosses those with vacuum in right cell

The **meaning** (semantics) of our knowledge base is the two worlds:

| | | |
|---|---|---|
| [vac | dirt], [vac | ] |

Now, do our previous statements imply beta5 (no dirt in left cell)? We check that for every world left over after we apply each sentence in our knowledge base. Here, it is, and so our KB does imply beta5.

We can check if a sentence is true in a particular world. How? Plug-in the values as fixed by that world into that sentence alpha.

Then we define the **meaning** of a sentence delta as the set of worlds in which that sentence is true, where W is the set of all (possible) worlds.

$$M(delta) = \{ W : W_i \models delta\}$$

| | |
|---|---|
| Where W_i | = delta means “delta is true in W_i” |

| | |
|---|---|
| World | = sentence |

Now, we know that

```
delta implies alpha
```

if and only if

```
M(delta) subseteq M(alpha)
```

If delta is true, then so is alpha, as the worlds where delta is true is a subset of alpha. The reverse would be incorrect, as then there’d be some delta where it was true, but alpha was false, which would violate the implication we wanted to show.

If we write a function that takes a sentence and gives back the set of all worlds where that is true. Then, we do this again for another sentence, and check if the sets are subsets.

{W_5, W_7} \subseteq {W_5, W_6, W_7, …}

“Yes, it’s a guess, but it has to be justified at some point.”

Evaluating a sentence is linear (go through and substitute). How many variables can we do? 2^N possible for N variables. 2^10 ~= 1000, 2^20 ~= 1 mill, 2^30 ~= 1 bill, about the current limit depending on techniques. ~25 with our current approach.

Using this subset-checking approach, we have to generate them all first.

# Lecture 10: 28 October 2015

Today: finish semantics, do inference (truth tables, rules/resolution, CSP).

Next homework due Monday, with homework due Monday because it's relevant to midterm (which covers up to end of today). There is an optional project later.

## Entails and Meanings

From before: we had variables, and the worlds are all the possible fixed combinations of them. We then used this symbol: w_i |= alpha to mean that a sentence alpha is true in world w_i, meaning that w_i evaluates to true when we plug-in the variables values in w_i.

Example: W_2 has (A, B, C) = (T, T, F) and alpha = (NOT(A) OR B) => (NOT(T) OR T) = (F OR T) = T

We say:

- "alpha true in w_i"
- "w_i satisfies alpha"
- "w_i entails alpha"
- "w_i is a model of alpha"

| | |
|---|---|
| w_i | = alpha is read as “w_i **entails** alpha” |

We say that the **meaning** of alpha is the set of all worlds in which alpha is true; the set of all worlds that entails alpha. When a world satifies a sentence, it is called a **model** of that sentence.

```
M(alpha) = { w_i IN W : w_i |= alpha }
```

Recall that W is the set of all possible worlds for a set of atoms.

Example:

```
M(NOT(A) OR B) = { w_1, w_2, w_5, w_6, w_7,
w_8 }
```

Where:

w_1 = (T, T, T) w_2 = (T, T, F) w_5 = (F, T, T) w_6 = (F, T, F)

Recall that this sentence is the same as A => B. This means that “IF A is true, then B must be true.”

w_3 = (T, F, T) fails because A is true here, but B is false

But w_5 has A = false, which technically doesn’t apply, as A is false here. “IF A is *true*”, which it’s not here. We don’t contradict that assertion that a true A requires a true B.

This is a *material implication*. Huge debate over whether A => B captures all of natural language conditionals.

## Implication, Validity, Consistency, and

# Equivalence

Semantics is the process of interpreting the meaning of a sentence.

Defining meaning, and all these things will let us interpret sentences.

## Valid Sentence

Example: complicated sentence to decide if a sentence is valid, i.e., always true. Tautology: A OR NOT(A). Translated to our set-based thinking: our sentence alpha is true in all worlds. There is no counterexample; we successfully stated a fact.

A sentence alpha is **valid** if the meaning of alpha is the set of all possible worlds:

```
M(alpha) = W
```

Or:

```
w_i |= alpha FOR ALL w_i IN W
```

## Inconsistent Sentence

A sentence being inconsistent means it disagrees/contradicts itself. Our sentence alpha is not true in any world.

```
M(alpha) = nullset
```

Or:

```
w_i NOT|= alpha FOR ALL w_i IN W
```

Example:

```
(A => B) AND (A => NOT(B)) AND A
```

The first two means that A must be false, but since A is true in the second, we'll always have false.

## Equivalent Sentences

Two sentences are equivalent if they have the same meaning. In other worlds, they share the same set of worlds in which they are true. Two sentences alpha, beta are equivalent if:

```
M(alpha) = M(beta)
```

Example: alpha = (A => B) ^ (A => NOT(B)); B = NOT(A)

Practically, our system is unaffected if we swap one for the other.

## Implication

Beta is true whenever alpha is true. In terms of worlds:

```
M(alpha) \subseteq M(beta)
```

| | | |
|---|---|---|
| If w_i | = alpha then w_i | = beta. |

Whenever alpha is true, since it's a subset, beta is also true.

This is kind of weird, as the smaller the set of worlds, the more we know, as we've eliminated other possibilities. Example: someone troubleshooting causes of failures. Who knows more? The one with fewer possibilities. Thus, alpha is a "stronger" statement than beta. The ultimate state of knowledge: only one world. But if we have empty, then it's inconsistent, as technically

everything follows from it, as the nullset is a subset of everything. That's why dividing by 0 allows us to "prove" that everything non-equal is suddenly equal, etc.

## Truth Table Queries

Example:

Knowledge base set of sentences, called delta:

- A
- A OR B => C

Our alpha = C

Our query? Does delta |= alpha? Does delta entail alpha? When we do so, we treat the set of all KB sentences as an conjoined single statement, i.e., A AND (A OR B => C) |=? C

This is exhaustive. We list out A, B, C and all of their possible values. Then, we compute M(delta), M(alpha), and show that M(delta) \subseteq M(alpha).

A B C A OR B A OR B => C A AND (A OR B=>C) C

The last two are delta and alpha. We can split up expressions, or compute them in-place for each. In the end, we get truth tables for delta and alpha. Then delta => alpha, because M(delta) is a subset of M(alpha).

The middle two intermediate calculations are not necessary on a test. They just let us carry over expression results.

MIDTERM QUESTION: we may be asked to use any one of these three methods

Actual computation doesn't use this representation, but this is a nice way to prove the correctness of algorithms.

## Inference

Mathematicians are not like us computer scientists. How do we actually get it done?

Example: if we know that P is true, and that P => Q, then Q is also true.

This is an inference rule, known as **modus ponens**, apparently often taught in philosophy classes.

Another rule: If P is true, then P OR Q is true. This is called **OR-introduction**.

Logic writes this with a line:

```
   P
-------
P OR Q
```

Basically, these rules let us add the therefore-statement to our knowledge base. If P is in our KB, then we can add in P OR Q.

Example:

These are our givens.

1. A

2. (A OR B) => C

Prove C, and our steps look like this:

1. A OR B is true, by or-introduction on 1 (A)
2. C, by modus ponens on 2 & 3

But then the question comes up whether or not a set of inference rules is sufficiently powerful or **complete** to prove every implication that semantics lets us check via truth table.
Example: we only have modus ponens above, and try to prove the same thing. Not possible, even though the truth tabled showed that C must be true.

“What is a complete set of inference rules?” Completeness comes up alongside efficiency.

Another question: how did we know to use or-introduction to choose B as the RHS of our or-statement? We knew that that would let us prove C eventually, but an algorithm won’t know that. What keeps the algo from wondering around and trying different statements until it gets it? This is apparently known as a control problem.

There is a rule called **resolution** that is complete for a specific case, just by itself.

| | |
|---|---|
| Syntax note: | = alpha means alpha is a tautology, i.e., alpha is always valid. |

# Refutation Theorem

Before resolution, first observe that:

```
alpha => beta
```

means:

```
M(alpha) \subseteq M(beta)
```

or:

```
alpha |= beta.
```

We can even write this a different way:

```
M(alpha) INTERSECT COMPLEMENT(M(beta)) = nul
lset
```

(this is equivalent to alpha implies beta)

```
M(alpha AND NOT(beta)) = nullset
```

(this is equivalent to the previous nullset statement)

In other words:

```
alpha => beta
```

if and only if

```
alpha AND NOT(beta) inconsistent
```

This is **proof by contradiction**. Assume alpha, assume not beta, but contradiction, so alpha true must mean that beta is also true, i.e., alpha implies beta. This is also known as the **Refutation Theorem**.

Example:

```
A AND (A OR B => C) implies C
```

iff

```
[A AND (A OR B =>C)] AND NOT(C) inconsistent
```

Notice that this is a constraint satisfaction problem. We have variables A, B, C; domain {true, false}; and three constraints:

- A
- A OR B => C
- NOT(C) // C is false

No solution, so this means that this statement is inconsistent, so by the Refutation Theorem, A AND (A OR B => C) implies C. We can do CSP with backtracking search or local search. We can even reduce this problem to SAT and toss this to a SAT solver. Note that SAT solvers do their early failure detection via a special case of resolution. Thus, search is augmented by the reasoning of inference rules.

## Resolution

Suppose we had :

- NOT(alpha) => beta
- beta => gamma

Well, notice that NOT(beta) OR gamma means

```
beta => gamma
```

Then by chaining:

- NOT(alpha) => gamma
- alpha OR gamma

Rsolution is not complete, but if we convert something to CNFs, then negate the statement we want to prove, and run resolution on it, then we can look for contradictions, which would prove that implication by contradiction.

Example:

1. A OR NOT(B) => C
2. C => D OR NOT(E)
3. E OR D

Does A => D?

Converted to CNF (procedure to be learned later):

1a. NOT(A) OR C 1b. B OR C

1. NOT(C) OR D OR NOT(E)
2. E OR D

Recall that the actual CNF is 1a AND 1b AND 2 AND 3

We want to negate what we want to show:

```
NOT(A => D)
NOT(NOT(A) OR D)
A AND NOT(D)
```

Is this a CNF? Actually, yes, because we can see these as two **unit-clauses**, clauses with a single literal.

```
(A) AND (NOT(D))
```

Adding this to our knowledge base:

1a. NOT(A) OR C 1b. B OR C

1. NOT(C) OR D OR NOT(E)
2. E OR D
3. A
4. NOT(D)

So let's use resolution and try to show that this is inconsistent

Recall resolution:

```
alpha OR beta, NOT(beta) OR gamma
----
alpha OR gamma
```

We drop the shared variable of beta. We need a beta and a not-beta between two sentences.

Resolving on 3 and 5, we get:

1. E

Resolving on 1a and 4:

1. C

Resolving on 2 and 6:

1. NOT(C) OR D

Notice that we're eliminating shared variables, as the combination of two sentences let us conclude something

simpler.

Resolving on 5 and 8:

1. NOT(C)

Which contradicts 7, so, by contradiction, A => D follows from givens 1-3.

### Unit Resolution

Special case of resolution:

```
alpha OR beta, NOT(B)
----
alpha
```

If we restrict ourselves to unit resolution, we lose completeness even for CNF. In general, resolution is exponential in the worst case, but unit resolution can be done in linear time for generating all implications from an initial knowledge base. Some KBs (which are not universal) are such that unit resolution is complete.

# Discussion 5: 30 October 2015

// Last part of today will mention how to convert to CNF for HW 5

## Syntax

Variables are Boolean variables taking on {true, false}

Literals are variables or their negated form. s

Sentences are literals or connectives of literals:

alpha, alpha OR beta, alpha AND beta, NOT(alpha), alpha => beta

Some equivalances:

```
alpha => beta <=> NOT(alpha) OR beta <=> NOT
(alpha) => NOT(beta)[contrapositive]

NOT(alpha AND beta) <=> NOT(alpha) OR NOT(be
ta)

NOT(alpha OR beta) <=> NOT(alpha) AND NOT(be
ta)

NOT(NOT(alpha)) <=> alpha
```

Where a <=> b means (a => b) AND (b => a)

## Normal Forms

We have CNF (conjunctions of clauses, which are disjunctions of literals).

```
(alpha OR beta) AND (gamma) AND (NOT(alpha)
OR gamma)
```

DNF is disjunctions of terms, which are conjunctions of literals:

```
alpha OR (alpha AND gamma) OR (alpha AND NOT
(beta))
```

Horn is a subset of CNF, where each clause has at most one positive literal (meaning a non-negated one)

```
NOT(alpha) AND gamma AND (alpha OR NOT(beta)
)
```

# Semantics

This is concerned with the meaning of sentences. What is the meaning of a sentence alpha? The set of all worlds where that sentence is true.

```
M(alpha) = { w_i IN W : w_i |= alpha }
```

Example: two variables A and B. 2^2 = 4 possible worlds. In general, N variables give 2^N possible worlds, as we can fix each variables as true or false in that world configuration.

```
    A   B   (A AND B)

w_0 T T T

w_1 T F F

w_2 F T F

w_3 F F F
```

Then for this set of all possible words W = { w_0, …, w_3 }:

```
M(A) = { w_0, w_1 }

M(B) = { w_0, w_2 }

M(A AND B) = { w_0 }
```

Let’s figure out some relationships among these sets:

```
M(A AND B) = M(A) INTERSECT M(B) // Where bo
th are true

M(A OR B) = M(A) UNION M(B) // Where one or
the other is true
```

```
M(NOT(A)) = COMPLEMENT(M(A)) = W - M(A)
```

## Validity, Consistency, Equivalence, Implication

From these definitions, we can further define these four concepts.

**Valid**: the sentence holds in every world (there is no counterexample):

```
M(alpha) = W
```

Note that true is always true in every world:

```
M(true) = W
```

Alpha is valid if the meaning of alpha is the set of all possible worlds. It is true in every possible world.

**Inconsistent**: alpha is inconsistent if its meaning is the nullset; there is no world in which is true. We've made some fail assertion.

**Equivalent**: alpha and beta are equivalent if their meanings are identical sets: they are true in all the same worlds:

```
M(alpha) = M(beta)
```

**Implication**: alpha implies beta if the meaning of alpha is a subset of beta. Whenever alpha is true, so is beta.

```
M(alpha) \subseteq M(beta)
```

## Proving Sentences

Suppose we want to show equivalence between:

`NOT(A OR B)`

And:

`NOT(A) AND NOT(B)`

Want to show that:

`M(NOT(A OR B)) = M(NOT(A) AND NOT(B))`

We can't apply De Morgan's directly, but we want to show that A => B and B => A via a bidirectional subset relationship on the truth tables. We can do the truth table in a steady fashion, left-to-right.

```
    A    B    (A OR B)          NOT(A OR B)
  NOT(A) AND NOT(B)

w_0 T T T F F

w_1 T F T F F

w_2 F T T F F

w_3 F F F T T
```

We get:

`M(...) = { w_3 } = { w_3 } = M(...)`

Notice that this fulfills the definition of set equality. Two sets are equal if and only if they are subsets of each other.

## Inference

First thing we need is a **knowledge base**, a conjunction of

sentences, things that we hold to all be (simultaneously) true as axioms.

A query is another sentence, an assertion we want to test.

| | |
|---|---|
| Does KB | = beta? That is, does KB imply/entail beta? |

Again, do a truth table, and show that M(KB) \subseteq M(beta)

## Inference Rules

There are some rules that let us derive additional truths from our knowledge base..

**Modus ponens**: If alpha, and alpha => beta, then beta is also true.

**Or-introduction**: If alpha, then alpha OR beta, where is beta is anything of our choice.

**Resolution**: If alpha OR beta, NOT beta OR gamma, then alpha OR gamma

And we have the idea of **completeness** of inference rule sets. Can our set of rules find any and all implications of a knowledgebase?

Any one of these rules by itself is not complete.

Example:

KB: P

Sentence beta to be proven: P OR Q

But can’t prove using just resolution; need or-introduction here.

**Refutation**: KB => B iff KB AND NOT(B) is inconsistent. This is proof by contradiction.

Same example as above.

1. P

2. NOT(B) = NOT(P) AND NOT(Q) // Applying De Morgan’s

We can split this, as they’re independent of each other. This is a conjunction of sentences, and our KB is a conjunction of sentences as well, so we can split it up and absorb them individually into our KB.

1. NOT(P)

2. NOT(Q)

| | |
|---|---|
| But 1 contradicts 2, so KB | = B, by Refutation Theorem. |

Example:

KB:

1. NOT(A) OR B
2. NOT(B) OR C
3. A

Try to prove: beta = C

**Refutation + resolution is complete**. MP, OI, and RES by themselves are not complete.

1. NOT(C) // Getting the negation of our claim for refutation
2. NOT(A) OR C // By resolution on 1 and 2
3. C // By resolution on 3 and 5
4. 4 and 6 contradict each other
5.

| | |
|---|---|
| KB | = beta // By Refutation Theorem, as KB AND NOT(B) inconsistent |

## Exhaustive Resolution

Example:

KB:

1. NOT(A) OR NOT(B) OR C
2. A
3. NOT(C) OR D

Prove beta = C

Getting NOT(beta):

1. NOT(C)
2. NOT(A) OR NOT(B) OR D // Resolution on 1, 3 // (eliminate shared var and disjoin rest)
3. NOT(B) OR D // Resolution on 2, 5

4. NOT(A) OR NOT(B) // Resolution on 1, 4
5. NOT(B) // Resolution on 2, 7
6. NOT(B) OR C // Resolution on 1, 2
7. NOT(B) // Resolution on 4, 10

IF we have exhausted all possible resolutions, then we must conclude that *the knowledgebase cannot imply beta*, but only AFTER we have EXHAUSTED ALL POSSIBLE RESOLUTIONS.

## Converting to CNF

Three rules:

1. Get rid of all connectives that are not AND, OR, NOT

2. Push negations inwards:

NOT(alpha AND beta) => NOT(alpha) OR NOT(beta) NOT(alpha Or beta) => NOT(alpha) AND NOT(beta)

3. Distribute OR's over AND's:

(alpha AND beta) OR gamma => (alpha OR gamma) AND (beta OR gamma)

KB:

- (A AND B) => C // Need to convert
- A // Already in CNF
- C => D

Converting the first:

```
(A AND B) => C  // Convert implication to ba
sal connectives
```

```
=> [NOT(A AND B)] OR C
=> [NOT(A) OR NOT(B)] OR C // Conjunction no
w; 'tis a clause; no distribution needed her
e, by coincidence
```

Converting the second:

```
C => D
=> NOT(C) OR D // Which is a clause
```

And now our CNF is the conjunction of all these clauses, our converted knowledgebase:

1. [NOT(A) OR NOT(B)] OR C // Parantheses not needed; equivalent here
2. A
3. NOT(C) OR D

Another example, with KB:

1. (A OR NOT(B)) => C
2. C => (D OR NOT(E))
3. E OR D // Already good
4. A => D // Easily becomes NOT(A) OR D

Converting 1:

NOT(A OR NOT(B)) OR C => (NOT(A) AND B) OR C // Distributing happens to give us two clauses => (NOT(A) OR C) AND (B OR C)

Final KB:

1. E OR D
2. NOT(A) OR D

3. (NOT(A) OR C) AND (B OR C)
4. NOT(C) OR (D OR NOT(E)) // From 2

# Lecture 11: 2 November 2015

// Justine thanked me for handing her that old 161 midterm! // She did it over the weekend, and said it was actually rather straightforward.

// Midterm will take up the whole period. Closed book, closed notes. // Questions we'll get: search trees and number according to expansion order // for different algorithms; minim

- Number nodes for expansion order
- Minimax and alpha-beta pruning
- Formulate into search from a word problem
- On logic, we'll need the truth table and inference rules

methods (2/8)

- Typically one question on writing a small Lisp function

// Expect about 8 question, 10-15 min per. No suprises: everything is // something we've seen before. Study!

"There's a high-pitched noise. That's not him, that's you!"

## Converting Propositional Sentences to CNF

Remember: resolution is only complete when used with refutation.

Example: delta = A OR NOT(B) alpha = A OR NOT(B) OR C

| | |
|---|---|
| ANd delta | = alpha (entails/implies), i.e., M(delta) \subseteq M(alpha) |

But using resolution does not let us show this, as we don't have a positive B here.

Using refutation, negate the first and conjoin (delta AND NOT(alpha) inconsistent iff delta entails alpha)

```
NOT(A OR NOT(B) OR C)
NOT(A) AND B AND NOT(C)
```

We can list and get NOT(B), contradicting B.

Back to conversion; let's convert this:

```
B  <=> (P OR Q)
```

1. Get rid of all connectives except for AND, OR, NOT

[B => (P OR Q)] AND [(P OR Q) => B]

[NOT(B) OR P OR Q] AND [NOT(P OR Q) OR B]

2. Push negations next to variables, i.e., get rid of negations of whole sentences. Apply De Morgan's law: NOT(…) = NOT(.) …

[NOT(B) OR P OR Q] AND [(NOT(P) AND NOT(Q)) OR B]

3. Apply distributive law:

(alpha AND beta) OR gamma <=> (alpha OR gamma) AND (beta

OR gamma)

[NOT(B) OR P OR Q] AND [(NOT(P) OR B) AND (NOT(Q) OR B)]

We now have CNF:

```
(NOT(B) OR P OR Q) AND (NOT(P) OR B) AND (NO
T(Q) OR B)
```

This last step may have to be done twice.

MEMORIZE:

```
A AND B AND C => D
```

Gives CNF:

```
NOT(A) OR NOT(B) OR NOT(C) OR D
```

Negate the the LHS and disjoin all, with non-negated D. This is often called a clause because it converts so straightforwardly.

Example:

```
B OR NOT(C) => E
```

Gives:

```
NOT(B) OR C OR E
```

Another:

```
NOT(A) AND C AND NOT(D) => NOT(F)
```

Gives:

```
A OR NOT(C) OR D OR NOT(F)
```

Okay, another example, where we apply all possible resolutions, but find no contradiction, which means delta does not imply alpha.

Delta, our KB:

```
A AND B => C
A
C => D
```

Alpha, our sentence to prove: C

| Query: Does delta | = alpha? |
| --- | --- |

Working through the proof with resolution:

1. NOT(A) OR NOT(B) OR C
2. A
3. NOT(C) OR D
4. NOT(C)
5. NOT(B) OR C
6. NOT(B) OR D
7. NOT(B)
8. NOT(A) OR NOT(B)
9. NOT(B)
10. NOT(A) OR NOT(B) OR D
11. NOT(B) OR D

Don’t worry too much about showing that we have bene exhaustive (there are formal methods to show that). In the general case, complexity can actually be quite bad. We use

something called *subsumption*, where we toss out everything that does not have B, as NOT(B) is stronger than something like NOT(B) OR C.

Adding the negation of alpha and not getting a contradiction shows that delta does not imply alpha.

## CSPs and Consistency

Recall that we found invalid assignments in CSPs with backtracking, which was augmented by arc consistency and early failure detection.

NOTE: back-tracking is COMPLETE. It will come back with a solution, or come back with “no solution exists”.

The other method: local search, which is NOT complete, but can be very fast for some problems. What will it look like for this? Make a random full assignment, and check constraint satisfication. If not, then check and compare against neighbor full assignments. It could wander around forever. That move to neighbors might mean changing one variable value, etc.

In general, we now have two main methods for checking if a knowledgebase delta implies alpha. Recall that delta |= alpha iff delta AND NOT(alpha) is inconsistent. We can check this:

1. Using resolution on delta AND NOT(alpha)
2. Using CSPs with backtracking search or local search.

## WALKSAT

WALKSAT is one answer to where to go next when we’re not yet

at a goal assignment. Pick a violated clause. Then, with probability p, choose a var whose flipping maximizes number of satisfied clauses. With probability 1 - p, choose a random var and flip it.

Why have this random flipping at all? It's randomization that helps prevent us from getting stuck. So long as p is not 0 or 1, then this method will find a solution with probability 1 as the number of iterations approaches infinity. That randomization is important!

As for that picked violated clause, it is apparently picked at random.

## First-Order Logic (Predicate Calculus)

This is the logic that lets us do quantification (if for all, then there exists). Recall the pits and monsters grid. When listing breezes around a pit, we had to have a variable for each breezy grid.

Some basic concepts:

- Object
- Relation
- Property
- Function

Consider the object of a cell, pit, etc.

An example relation: adjacency.

An example property: breezy or not breezy.

An example function: up/down/left/right of a cell. up(cell) = cell above it.

The syntax of first-order logic is more intricate.

Let's translate some English:

`one plus one equals three`

Objects: one, three Properties: Relation: equals Function: plus

A function takes one or more objects and maps them to another object. A relation states a relationship between existing objects. No properties here.

`squares neighboring the wumpus are smelly`

Objects: squares, wumpus Properties: smelly Relations: neighboring Functions:

Some more basics of first-order:

- Constants: John, 2, UCLA
- Predicates: neighor(), brothers(), >
- Functions: cell_above(), plus()
- Variables: x, y, z, a b, c
- Connectives: AND, OR, NOT, =>
- Equality: =
- Quantifiers: FOR ALL, THERE EXISTS

The first three are more like variables in propositional logic, which may exist sometimes; the remaining are always there.

Designating objects can be done in more than one way:

- By name, i.e., constants
- By variables
- By functions (3 is 1 + 2; the cell above this one)

Properties can be seen as a special case of relations. Predicates cover both altogether. Equality is like a built-in predicate that's just so useful. It is the logic of math. Early AI looked down on propositional logic as too trivial, but has since returned to it.

## Syntax

The simplest possible sentence in first-order logic has this form:

```
Predicate(term1, term2, ..., termN)
```

Where:

```
term = function(term1, ..., termM)
     or constant
     or variable
```

This predicate of terms (objects) is called the **atomic sentence**. Compare this to the simplest possible sentence "A" in propositional logic.

IMPORTANT: we have terms, which are objects, which are not two-valued like the Boolean nature of propositional variables.

Example:

```
Friends(John, spouseOf(Mark))
```

Connectives are just NOT(sentence), S1 AND S2, etc.

FINAL PROBLEM: Get an English sentence, and write it in first-

order logic. This is easier than the other way around.

## Quantification

We say that a variable x mentioned in a sentence S is called **free**.

But if we say:

```
FOR ALL x, S (holds)
```

Then we say x is **universally quantified**.

Example:

```
FOR ALL x At(x, Berkeley) => Smart(x)
```

“Everybody at Berkeley is smart.”

WE MUST QUANTIFY variables for it to be a valid first-order sentence. We can see this as making a sentence for and with each possible value of x, and then conjoining all of them together. In this way, FOR ALL can be seen as a big AND.

If we have

```
THERE EXISTS x, (such that) S
```

then we say x is **existentially quantified**:

```
THERE EXISTS x AT(x, Stanford) AND Smart(x)
```

Next, if we have:

```
FOR ALL x, FOR ALL y ....
```

Can we flip the quantification order? Yup! Same thing for:

```
THERE EXISTS x, THERE EXISTS y
```

But what about:

```
THERE EXISTS x, FOR ALL y
```

Concrete example:

```
THERE EXISTS x, FOR ALL y loves(x,y) // Some
one loves everyone
```

Flipped:

```
FOR ALL y, THERE EXISTS x loves(x, y) // Eve
ryone is loved by someone
```

Is this the same? There are consistent with each other, but we want to check equivalence.

NOTE: loves(x,y) is not a symmetric relation. The order of the predicate matters.

Concrete example: John, Suzanne, Sally

John loves himself, Suzanne, Sally. This satifies both.

But what about: everyone loves themselves but no one else? Then this fulfills the second, but not the first.

IN GENERAL, we cannot flip existence and for-all.

Note that there is **duality** between for-all and there-exists:

```
FOR ALL x likes(x, IceCream)
```

```
NOT THERE EXISTS x NOT(Likes(x, IceCream))
```

Everyone likes ice cream. There does not exist someone who does not like ice cream. See them in modular components.

Another example:

```
THERE EXISTS x likes(x, Broccoli)
```

```
NOT FOR ALL x NOT(likes(x, Broccoli))
```

Someone like broccoli, so it is not true that everyone dislikes broccoli.

# Lecture 12: 9 November 2015

Recall from last time:

The simplest possible sentence in first-order logic has this form:

```
Predicate(term1, term2, ..., termN)
```

Where:

```
term = function(term1, ..., termM)
     or constant
     or variable
```

In general, terms designate objects. Note that functions are unique mappings.

Recall also:

```
THERE EXISTS x, FOR ALL y Loves(x,y) // Ther
e is someone who loves everyone
```

Flipped:

```
FOR ALL y, THERE EXISTS x Loves(x, y) // Eve
ryone is loved by someone
```

Note that Loves(x,y) is interpretable as x Loves() y. And that the x in the second sentence not necessarily unique. Each y may have a different x who loves them.

Example love mapping:

A->A A->B A->C

For which both sentences hold.

But consider:

A->A B->B C->C

Then, only the second sentence holds, and not the first.

There is a formalization of worlds and semantics for first-order logic, but we won't go into details for it, as it's more complicated than a set of truth assignments. We call that analogous concept an *interpretation*.

## Converting English to First-Order Logic (FOL)

Express the sentence: "mother is female parent"

We first need to define a **vocabulary** composed of constants, functions, and predicates. Here, for example, we need to define what it means to be a mother.

---

```
FOR ALL x, y Mother(x,y) <=> Female(x) AND P
arent(x, y)
```

Constants: N/A Functions: N/A Predicates: Mother(), Female(), Parent()

---

Next example, defining siblings:

```
FOR ALL x, y Sibling(x,y) <=> Sibling(y,x)
```

If we forget to define commutativity and other pieces of knowledge like this on relations, then our knowledge base will not give us all the conclusions expect.

The sentence: “Tuna has two sisters”

// Who the hell names their kid “Tuna”? // Oh, it’s apparently a cat.

We could begin by defining what it means to be sisters:

```
Sister(x, y) <=> ...
```

Or we could also just write it straight, but we’ll need to return to this afterward to define Sister():

```
    THERE EXISTS y,z Sister(Tuna, y) ^ Siste
r(Tuna, y) ^ Sister(Tuna, z) ^ NOT(y=z)
```

More formally, we might write that last predicate as NOT(=(y,z)).

Note that this really only states that Tuna has at least two sisters, rather than exactly two.

---

Another sentence: “There is exactly one king.”

### Contrapositive

Contrapositive: If A => B, then NOT(B) => NOT(A)

First version (there’s at least one king but all the others are not kings):

```
THERE EXISTS x, King(x) AND [FOR ALL y NOT(x
=y) => NOT(King(y))]
```

Contrapositive version (if there is another king, it’s x):

```
THERE EXISTS x, King(x) AND [FOR ALL y King(
y) => x=y ]
```

---

How about this to English:

```
FOR ALL x [Cat(x) v (THERE EXISTS x Brother(
Rich, x))]
```

For all x, x is a cat OR x is a brother of Rich (a constant). But remember that without that first FOR ALL, the x would be “free”, and the sentence would not be a syntactically correct FOL sentence. It would not be **well-formed**.

## Inference Engines

We’ll feed axiomatic knowledge and a query into an inference engine. We’ll get yes/no for whether a query sentence holds, but also a **binding list**.

```
KB    ->  [ Inference ] -> yes/no
Query ->  [ Engine    ] -> binding list
```

From 131: Prolog does this (a logic programming language)

Our example knowledge base:

```
FOR ALL x, Cat(x) => Mammal(x)
Cat(Tuna)
Cat(Spot)
```

Our query:

```
THERE EXIST x Mammal(x)
```

Is there a mammal?

## Binding Lists

Yes, of course, but we also get back a binding list that looks like this:

```
{ x|spot, x|tuna }
```

Indeed, there is x such that Mammal(x), and in fact, *here are the bindings of x that fulfull this*.

Let's look at append() in Prolog, and exhumatize (?) it.

Constant: EL (the empty list) Function: list(a,b) // Construct a list from its two args Predicate: Append(a,b,c) // Append b to a and get that result in c

Defining Append via FOL:

```
// Appending EL to y gives us y
FOR ALL y Append(EL, y, y)

// If z is the result of appending x and y,
then this must also be true
```

```
FOR ALL a,x,y,z Append(x, y, z) => Append(Li
st(a, x), y, List(a,z))
```

Now, querying the above knowledge base about Append, we ask:

```
THERE EXISTS c Append(List(1, EL), List(2,EL
), c)
```

Then we get back “yes” and:

```
{ c|List(1, List(2,EL)) }
```

A more interesting query:

```
THERE EXISTS a,b ; Append(a,b, List(1,List(2
, EL)))
```

Does there exist a, b such that appending b to a gives us the list made by (List(1, List(2, EL)))? Yes, there are three values that will do so.

## Prolog Example

```
// In Prolog, everything is implicitly unive
rsally quantified
// We assume/read this as "For all y, ..."

// [] is the empty list
append([], y, y)

// This is cons notation: [a|x]
append([a|x], y, [a|z]) := append(x, y, z)

// Prolog reverses the if-then order
// See Vlad for questions on this synatx, as
 he's taken 131
```

Next-up: automated reasoning about digital logic, diagnosing problems with inputs vs outputs, etc. This would be the theory behind crap like Xilinx.

# Workflow for FOL on Digital Logic

Our steps:

1. Decide our vocabulary
2. Encode general domain knowledge (how the different logic gates work, etc.)
3. Encode problem-specific knowledge (our particular module/circuit)
4. Pose queries (test)
5. Debug (ugh)

For this circuit:

// TODO: Update with link to picture

1 —> XOR[x_1] —–> XOR[x_2] —> 1 2 —> 2

3

## Decide vocabulary

The constants label the gate instances, as well as their type.

- Constants: 0, 1, X_1, X_2, A_1, A_2, O_1, C_1, AND, OR, NOT

Type(a) tells us what kind of gate something is: AND, OR, etc.
Signal(a) tells us the value of the signal on a wire.

- Functions: In(a,b), Out(a,b), Type(a), Signal(a)

- Predicates: Connected(a,b)

In practice, we come up with some initial vocabulary formulation, then come back and change things; this job was called *knowledge engineering*.

## Encode general domain knowledge

If two terminals are connected, they must have the same signal

1. FOR ALL T_1, T_2 ; Connected(T_1, T_2) => Signal(T_1) = Signal(T_2)

All terminals have a value of 0 or 1

1. FOR ALL T, Signal(T) = 1 OR Signal(T) = 0

Connectedness is commutative:

1. FOR ALL T_1, T_2 ; Connected(T_1, T_2) = Connected(T_2, T_1)

Now the axiom that describes the behavior of OR gates. Why two parameters? This is from the book, with labeled outputs 1 and 2.

1. FOR ALL g, (Type(g) = OR) => [ Signal(In(1,g)) = 1 OR signal(In(2,g) = 1) => Signal(Out(1,g)) = 1 ]

But this only describes OR-gate behavior for when we have a 1. What about when we have both 0-value inputs? We can handle this by making this an iff:

1. FOR ALL g, (Type(g) = OR) => [ Signal(In(1,g)) = 1 OR signal(In(2,g) = 1) <=> Signal(Out(1,g)) = 1 ]

Then, we'd do this for the remaining gates, etc.

### Encode problem-specific information

Now the second part of our knowledge base, where we capture the information conveyed by that circuit diagram. We declare the gates' types and then define their connections:

1. Type(X_1) = XOR
2. Type(O_1) = OR
3. Connected(Out(1,X_1), In(1,X_2)) …
4. Connected(In(3,C_1), In(1, A_2))

This will encode our adder into our KB. Note that this encoding into KB from schematic could be automated. We can even take in a truth table for an arbitrary logic gate definition and spit out an FOL axiom for it.

This is extensible to any system described in terms of **function blocks**, where we just need a table that defines a particular function block.

### Diagnosis

Suppose we feed input for our module, and ask what went wrong, e.g., when we should have gotten 1 for an output. To define the idea of a bad gate/module, we define OK(), Bad(), etc. This is saved for the grad-level AI work.k

## FOL Inference

Suppose we want to make the two sentences on each line indentical.

| Substitute { x | Jane }: |
|---|

`Knows(John, x) Knows(John, Jane)`

| Substitute { y | John, x | OJ }: |
|---|---|---|

`Knows(John, x) Knows(y, OJ)`

| Substitute { y | John, x = Mother(John) }: |
|---|---|

`Knows(John, x) Knows(y, Mother(y))`

Not possible, referred to as a not **unifiable**.

`Knows(John, x) Knows(x, OJ)`

These substitutions are called **unifiers** (see Vlad for Prolog examples).

FINAL PROBLEM: “Find the most general unifier”

---

// Capitals are constants; lowercase is variables

Another example, unifiable or not:

`P(A, y, z)  P(x, B, z)`

Simple (we bind x to A, etc.):

```
{ x|A, y|B }
```

A less general unifier:

```
{ x|A, y|B, z|C }
```

We make commitments we don't have for z's value.

## Resolution in FOL

Our KB has (where Rich is a predicate [wealthy]):

```
NOT(Rich(x)) OR Unhappy(x) // We are either
not rich or unhappy
Rich(Ken) // Ken is rich
```

But this is not easily cancellable, as x != Ken. We have to unify these before applying classic resolution.

Our unifier:

```
{ x|Ken }
```

Then we have:

```
NOT(Rich(Ken)) OR Unhappy(Ken)
Rich(Ken)
```

And we conclude Unhappy(Ken), because we do this AFTER substituting.

In the logic notation:

```
A OR B', NOT(B) OR C
--------------------
[A OR C]G  // Darwich uses theta here, meani
```

```
ng we substitute in G
```

Assuming:

```
B' G = B G
```

Next time will bring examples of this.

# Discussion 6: 13 November 2015

On node expansion: if goal, STOP.

Non-trivial condition on u and f(u) from the midterm: monotonically increasing function f(u)

Or: f(u) = c * u (c > 0) f(u) = u + c

## Syntax

Objects are constants or variables (which take on a value later):

```
apple, fruit, x
```

Functions return objects:

```
best_friend(x)
add(x,y)
```

Predicates return true/false:

```
is_best_friend()
```

Connectives are logical connectives:

```
AND, OR, NOT, =>, <=>
```

Quantifiers specify the scope of our sentence:

```
ALL X is_man(x) => is_mortal(x)
```

The atomic sentence has the form:

```
Predicate(term, term, ..., term)
```

Where each term is a constant, variable or function. Example:

```
is_best_friend(John, father_of(John))
```

Connectives can be applied to sentences as well.

*Grounding sentence*: tells us something about a predicate:

```
ALL x, P(x)
```

Equivalent to:

```
P(A) AND P(B) AND P(C)
```

Consider also:

```
EXISTS x, P(x)
```

Which is equivalent to:

```
P(A) OR P(B) AND P(C)
```

Where A, B, C, etc. are the possible values of x.

## Equivalents

ALL x, ALL y, A == ALL y, ALL x, A

EXISTS x, EXISTS y, A == EXISTS y, EXISTS x, A

ALL x, A == NOT(EXISTS x) NOT(A)

EXISTS x, A == NOT(ALL x) NOT(A)

But note that:

```
EXISTS x, ALL y, A != ALL y, EXISTS x, A
```

# Knowledge Engineering

Construct a knowledgebase (systematically).

Example:

1. Lucy is a professor
2. All pprofesosrs are people.
3. John is the dean.
4. Deans are professors.
5. All professors consider the dean a friend or don’t know him.
6. Everyone is a friend of someone.
7. People only criticize people that are not their friend.
8. Lucy criticized John.

Query: Is John a friend of Lucy’s?

## Construct a Vocabulary

First step: construct a vocabulary that lets us discuss the problem.

Predicates: is_prof(x), is_person(x), is_dean(x), is_friend_of(x, y), knows(x, y), has_criticized(x, y)

Functions: none needed here.

Constants: Lucy, John

## Encode General Knowledge

Notice how general knowledge uses the quantifies.

Statement 2:

```
ALL x, is_prof(x) => is_person(x)
```

Statement 3:

```
ALL x, is_dean(x) => is_prof(x)
```

Statement 5:

```
ALL x, ALL y; is_prof(x) AND is_dean(y)
    => is_friend_of(y, x) OR NOT(knows(x,y))
```

Statement 6:

```
ALL x, EXISTS y is_friend_of(x, y)
```

Statement 7:

```
ALL x, ALL y; criticize(x, y) AND is_person(
x) AND is_person(y)
        => NOT(is_friend_of(y, x))
```

## Encode Problem-Specific Knowledge

Statement 1:

```
is_prof(Lucy)
```

Statement 3:

```
is_dean(John)
```

Statement 8:

```
criticized(Lucy, John)
```

And our query:

```
is_friend_of(John, Lucy)
```

Another one: “Is the dean a friend of himself?”

```
ALL x, is_dean(x) => is_friend_of(x, x)
```

(No constant for dean, so asking if someone is a dean, are they also a friend of themselves, and we want to know if this applies to all people).

Recall that our knowledge engine still takes in a knowledge base and query, but gives back true/fase as well as a binding list, all the values of each variable that fulfill the query.

Because of this binding list return, we can ask questions like “Who are the friends of Lucy?”

```
EXISTS x, is_friend_of(x, Lucy)
```

We’d get back the values of x that fulfill the predicate “x is a friend of Lucy”, alongside a value of true. Or, we’d get back false and the empty set.

Example binding list: { x/Fred, x/Franklin }

Next week will cover inference and deriving new knowledge from our existing knowledge based.

# Unification

If we have two atomic sentences P, Q, unification is finding the substitution that makes P, Q identical.

Example:

`Knows(John, x)`

`Knows(y, mother_of(y))`

We make the bindings:

`{ y/John, x/mother_of(John) }`

Also correct, since y/John:

`{ y/John, x/mother_of(y) }`

Another example:

```
P(A, y, z)
P(x, B, z)
```

Binding 1:

```
// Leave z as variable
{ x/A, y/B }
```

Binding 2:

```
// z bound to A for both sentences; could bi
nd to any value
{ x/A, y/B, z/A }
```

But we often want the most general quantifier, meaning the one that makes the fewest restrictions on the values of variables.

Notice how binding 2 imposes a value on z, whereas binding 1 does not.

### Unique General Unifier

Important: the most general unifier (MGU) is unique (for every unifiable pair of expressions)

# Lecture 13: 16 November 2015

// Unbelievably cold and windy today. It's biting // and requires multiple layers to combat.

## Converting to CNF for FOL; Skolemization

In propositional logic (PL), resolution is guaranteed to terminate, either at a contradiction or showing consistency.

In first-order logic (FOL), resolution is NOT guaranteed to terminate. It can end in contradiction, but it may not.

NOTE: new conventions, following :

- FOR ALL: A
- THERE EXISTS: E
- AND: AND
- OR: OR
- NOT: ~

Example for CNF conversion:

```
A x [A y, Animal(y) => Loves(x,y)] => [E y,
Loves(y, x)]
```

“Everyone who loves all animals is loved by someone”

// This is dumb notation. Should use z instead of y again // (which is not an animal second time around)

Break it down into pieces for translation.

Okay, steps for converting to CNF:

1. Eliminate =>, <=>

(A x) [~[(A y) ~Animal(y) OR Loves(x,y)]] OR [E y, Loves(y,x)]

Note where the negation goes:

```
A x, P => Q
```

Becomes:

```
A x, ~P OR Q
```

1. Move negation (~) inwards.

Recall the rules for this (analogous to De Morgan’s):

- ~(A x, P) => (E x, ~P)
- ~(E x, P) => (A x, ~P)

Moving inward, applying de Morgan’s as well:

```
A x [E y, Animal(y) AND ~Loves(x,y)] OR [E y
, Loves (y,x)]
```

1. Standardize variables (each quantifier gets a different

variable)

A x [E y, Animal(y) AND ~Loves(x,y)] OR [E z, Loves(z,x)]

2. Skolemization (gets rid of existential quantification, as we eventually get rid of all quantifiers, assuming the result is universally quantified)>

Example:

```
(A x, E y) Loves(y,x)
```

Everyone has someone who loves them. But the the particular y who loves x is rather unique to x. So, we define a function F(x) that gives back the person that loves x, as that particular y is rather unique to x, almost a property of it.

```
A x, Loves(F(x), x)
```

Skolem showed that transforming a KB this way led to an equally satisfiable KB (only satisfiable if the original was, and vice versa).

NOTE: F(x) must be unique to the knowledge base, or else we're setting up links that may have unintended side effects.

Note that skolemization can depend on more than one variable (F(x,z)) or none at all (when we just have a constant), i.e., F(x) = c for any x.

What about something like:

```
E x, King(x)
```

Make up a **Skolem constant** that is unique to the KB:

```
King(G)
```

Back to our original example, replacing with functions and constants to Skolemize our sentence:

```
A x [Animal(F(x)) AND ~Loves(x, F(x))] OR [L
oves(G(x), x)]
```

And applying the rule that (A AND B) OR C => (A OR C) AND (B OR C):

```
A x [Animal(F(x)) OR Loves(G(x), x)] AND [~L
oves(x, F(x)) OR Loves(G(x), x)]
```

1. Drop universal quantifiers, as they're assumed for Skolemized sentences.

[Animal(F(x)) OR Loves(G(x), x)] AND [~Loves(x, F(x)) OR Loves(G(x), x)]

Notice that now have CNF: (P OR Q) AND (~R OR Q). Our literals are now predicates, functions, etc. instead of simple constant atoms.

## Resolution on FOL KB

English story:

1. Jack owns a dog:

E x Dog(x) AND Owns(Jack,x)

2. Every dog owner is an animal-lover.

A x [E y, Owns(x,y) AND Dog(y)] => Alover(x)

3. No animal-lover kills an animal.

A x Alover(x) => [A y, Animal(y) -> ~Kills(x,y)]

4. Either Jack or curiosity killed the cat Tuna.

Kills(Jack, Tuna) OR Kills(Curiosity, Tuna) Cat(Tuna) A x, Cat(x) => Animal(x)

That's our knowledge base delta. Now is alpha = Kills(Curiosity, Tuna) implied by delta? Does delta |= alpha? Did curiosity kill the cat?

Resolution with this added to our KB: delta AND ~alpha (which is in CNF). Recall that this is how we did this with proof by contradiction.

Conversion to CNF:

(1) requires just Skolemization, here introducing a constant:

```
Dog(D)
Owns(Jack, D) // Recall that KB's are conjoi
ned terms, so we can split them
```

(2) becomes:

```
~Owns(x,y) OR ~Dog(y) OR Alover(x)
```

(3) becomes:

```
~Alover(x) OR ~Animal(y) OR ~Kills(x,y)
```

(4) is already a clause:

```
Kills(Jack, Tuna) OR Kills(Curiosity, Tuna)
```

(4b) is already a unit-clause:

```
Cat(Tuna)
```

(4c) becomes:

```
~Cat(x) OR Animal(x)
```

Our final KB in CNF form:

1. Dog(D)
2. Owns(Jack, D)
3. ~Owns(x,y) OR ~Dog(y)
4. OR Alover(x)
5. ~Alover(x) OR ~Animal(y) OR ~Kills(x,y)
6. Kills(Jack, Tuna) OR Kills(Curiosity, Tuna)
7. Cat(Tuna)
8. ~Cat(x) OR Animal(x)
9. ~Kills(Curiosity, Tuna) // Our negated query

Just like PL, we do resolution on terms that share B and ~B (literal and its unification), but now we may need to do unification to match.

Look at (1) and (3). Dog(D) and ~Dog(y) don’t quite mirror, unless we bind/unify y to D: theta = { y/D }. Remember the result comes AFTER we substitute in D, meaning that we only apply resolution after all of the variables have been replaced according to the unification:

1. Dog(D)
2. Owns(Jack, D)
3. ~Owns(x,y) OR ~Dog(y) OR Alover(x)

4. ~Alover(x) OR ~Animal(y) OR ~Kills(x,y)
5. Kills(Jack, Tuna) OR Kills(Curiosity, Tuna)
6. Cat(Tuna)
7. ~Cat(x) OR Animal(x)
8. ~Kills(Curiosity, Tuna) // Our negated query
9. ~Owns(x,D) OR Alover(x) // 1,3 where theta = { y/D }
10. Alover(Jack) // 2,3 where theta = { x/Jack, y/D }
11. Animal(Tuna) // 6,7 where theta = { x/Tuna }
12. ~Animal(y) OR ~kills(Jack, y) // 4,9 where theta = { x/Jack }
13. ~Kills(Jack, Tuna) // 10,11
14. Kills(Curiosity, Tuna) // 5,12 => CONTRADICTION!

## Reduction to Propositional Logic (Propositionalization)

By converting to PL, we can turn it into a CSP and throw it at a SAT solver. This is cheating to avoid the complexity/sophistication of FOL.

We want to take something that is FOL and convert it to PL.

Example:

```
A x, King (x) AND Greedy(x) => Evil(x)
```

And suppose we only have a PL reasoning engine. We can't have functions, variables, quantifiers, etc.

But what we can do is generate instances over the whole domain of x, as that gives us Boolean constants.

```
King(P1) AND Greedy(P1) => Evil(P1)
...
```

```
King(P5) AND Greedy(P5) => Evil(P5)
```

Now, for each one, we need only substitute a propositional variable for each term, e.g., x1 = King(P5), which is now true/false.

```
x1 AND y1 => z1
...
x5 AND y5 => z5
```

Problem: notice how we could end up with a very large PL KB despite only a few variables. Sometimes, this approach actually wins out. We can also become more sophisticated: we can filter what to add to our KB or not, based on our query.

But we're not done propositionalizing just yet: we haven't converted functions, e.g., Father(x,y) means we'll need King(Father(P1)) AND … Now, instead of a linear relationship of possible variable values to number of sentences in our KB, we could possibly have infinitely many sentences, as we can introduce functions of arbitrary complexity and interconnectedness. No longer 5 values to 5 sentences; we could have those 5, then accounting for father generates many more.

## Semi-Decidability

Turing/Church theorem result: does delta |= alpha? We can convert to PL and reason, but to avoid the infinity, we limit how much we nest (0 is no function applications, 1 is one function application, 2 has function nesting, etc.).

Then, if we limit our testing, then we will generate a finite PL KB. If no entailment at that level of nesting, then add another level, and keep going. If alpha does follow, then we will terminate at

some nesting level. But if alpha does not, we could go on infinitely.

If it follows, we can tell you; if it does not follow, we cannot tell you. Sometimes, FOL work is done without function symbols altogether, and/or with a finite domain (to avoid infinitely many values).

Datalog (in databases) is apparently related to this.

FOL courses also exist in graduate CS, philosophy, and math.

## Probabilistic Reasoning

~1979: McCarthy writes about how formal logic sucks at encoding common-sense knowledge.

Example problem:

```
Bird(x) => fly(x)
```

But if we try to write in exceptions, we contradict ourselves:

```
Bird(Tweety) AND ~Fly(Tweety)
```

So we try to define that a bird can be abnormal and therefore it doesn't fly. But that doesn't work either.

How about assuming something is true until we know otherwise? That births **non-monotonic logics**. They make assumptions based on existing knowledge, and can retract them with new knowledge. This is difficult mathematically.

Another example:

```
A x, Quaker(x) AND ~Abnormal(x) => Pacifist(
x)

A x, Republican(x) AND ~Abnormal(x) => ~Paci
fist(x)
```

By default, if we learn someone is a Quaker, we assume there are a pacifist, as we initially assume that they are not abnormal.

But what if we learn:

```
Quaker(Nixon)
Quaker(Republican)
```

The default conclusions clash: Pacifist and not Pacifist? The two normality assumptions can't both be true. We have no way of managing our (possibly conflicting) assumptions, which lead to **belief revision**.

We want to get to Bayesian networks, which will be built atop probability calculus.

# Lecture 14: 18 November 2015

Today: beliefs, belief change, independence/causality, properties of beliefs

## Probabilistic Beliefs

// We're back to propositional logic for now.

Example:

The earthquake or the burglary could set off the alarm:

```
Burglary OR Earthquake => Alarm
```

With the possible worlds being. This table is also called a **probability distribution** or **state of belief** or a **join distribution** specifically.

```
World   Earthquake  Burglary    Alarm   Prob
ability

W1          T           T           T
0.0190
W2          T           T           F
0.0010
W3          T           F           T
0.0560
W4          T           F           F
0.0240
W5          F           T           T
0.1620
W6          F           T           F
0.1800
W7          F           F           T
0.0072
W8          F           F           F
0.7128
```

Rather than just having different possible realities, we now assign probabilities to each, all of which should add to 1, and between 0 and 1.

Complaints when this idea was first introduced: where will the data for those probabilities come from? Constant fine-tuning of such constants by hand is stupid. Later, we'll learn Bayesian networks which will let us avoid having to write out such tables by hand.

// Darwiche wrote a book on Bayesian networks

The next step is to assign probabilities to a sentence. Example:

```
Pr(Earthquake OR Burglary)
```

```
alpha = Earthquake OR Burglary
```

Then, to get its probabilities, we collect the probabilities of worlds where alpha is true:

```
Pr(alpha) = SUM_{w_i |= alpha} Pr(w_i)
```

Example, probability of an earthquake:

```
Pr(E) = Pr(W1) + Pr(W2) + Pr(W3) + Pr(W4) =
0.1
```

Or no burglary:

```
Pr(~B) = 0.8
```

Of an alarm:

```
Pr(A) = 0.2442
```

We now have the ability to assign a probability to any sentence.

Note that for any sentence (derived from the sum adding up to 1 and assumptions on data within table):

```
0 <= Pr(A) <= 1
```

If a sentence is inconsistent, then:

```
Pr(A) = 0
```

If a sentence is valid (true in every world):

```
Pr(A) = 1
```

And for a sentence and its negation:

```
P(A) + Pr(~A) = 1
```

// Sleepy from sleeping late last night. Woke up earlier than expected too.

Given separate probabilities:

```
P(A OR B) = P(A) + P(B) - P(P AND B)
```

We cannot simply add, as there are worlds where both might be true, which leads to double-counting for their intersection.

We could state an inequality though:

```
P(A OR B) <= P(A) + P(B)
```

But we can get inequality if they have no overlap (**mutually exclusive**), meaning that (A AND B) would be inconsistent:

```
P(A AND B) = 0
P(A OR B) = P(A) + P(B)
```

Some more inequalities:

```
Pr(A AND B) <= Pr(A) + Pr(B)
```

If we go ahead and get the **marginal distribution**:

```
Veracity    Earthquake    Burglary    Alarm

   T            0.1          0.2       0.2442
   F            0.9          0.8       0.7558
```

Notice that probablities within each column add up to 1, as

expected. This table talks only about the probability of any one of these events.

Now, we define the idea of **entropy**, which defines how certain we are about our probabilities.

```
ENT(X) = - SUM_x [Pr(X=x) log_2(Pr(X=x))]
```

Capital letters like ‘X’ are variables; lowercase like ‘x’ are values.

Entropy of the above:

| Veracity | Earthquake | Burglary | Alarm |
|---|---|---|---|
| T | 0.1 | 0.2 | 0.2442 |
| F | 0.9 | 0.8 | 0.7558 |
| | 0.469 | 0.722 | 0.802 |

The more we’re sure about something being true or false, the lower the entropy. It has an upside-down parabola shape for the graph, with a peak at (0.5, 0.5), where we’re not sure at all if it should be true or false.

## Belief Revision

What if we get new information, often called **evidence**? For example, an alarm suddenly does trigger.

// Vlad reports that Berkeley’s AI class is superior. It’s not surprising. // I want the fucking semester system, dammit.

What if we’re told that the alarm has gone off? Then, the worlds where the alarm has not gone off contradict our new evidence. So, we would set their probabilities to 0.

We assume that evidence is **hard/certain evidence**, which is 100% true, and takes precedent over previous information. There is the idea of soft evidence (your deaf neighbor says the alarm went off).

```
World   Earthquake  Burglary    Alarm   P
        P(given alarm/~alarm)

W1          T           T           T
0.0190
W2          T           T           F
0.0010              0
W3          T           F           T
0.0560
W4          T           F           F
0.0240              0
W5          F           T           T
0.1620
W6          F           T           F
0.1800              0
W7          F           F           T
0.0072
W8          F           F           F
0.7128              0
```

The zeros would be flipped if we were told the alarm did NOT go off.

We want to add up to 1, but we should maintain pairwise proportionality. One way to do this: add up all of them, then divide them by that.

B is our evidence, alarm or ~alarm.

```
P(w_i | B) = {
    0 if w_i |= ~B // Contradicts our eviden
ce
```

```
    P(W_i)/P(B) if w_i |= B
}
```

This is our rule for updating beliefs.

## Bayes Conditioning/Rule

Play/derive with this, and we'll get Bayes' conditioning:

```
P(A|B) = P(A AND B) / P(B)
```

Some examples:

```
P(Burglary) = 0.2
P(Burglary|Alarm) = 0.741
P(Earthquake) = 0.1
P(Earthquake|Alarm) = 0.307
```

Patterns of plausible reasoning: if our systems don't match what we expect intuitively, we should rethink them. Example: given an alarm, and then certainty of no earthquakes, no burglaries, etc.

The big deal: it was shown that probabilistic reasoning fit these patterns.

Theorem: more information cannot increase entropy. See information theory.

Suppose that someone is told evidence such that they believe that burglaries are irrelevant to earthquakes, and vice versa:

```
P(Earth) = 0.1
P(Earth|Burg) = 0.1
P(Burg) = 0.2
P(Burg|Earth) = 0.2
```

Someone else could think differently. Then, we see that

earthquakes are **independent** of burglaries. Our belief does not change after learning something else:

```
P(A) = P(A|B)
```

A is independent of B. The probability of A did not change given B.

If that is so then, then we can rewrite P(A|B) from what we discovered earlier with Bayes' conditioning ( `P(A|B) = P(A AND B) / P(B)` )

```
P(A) = P(A AND B) / P(B)
```

```
P(A AND B) = P(A) * P(B)
```

The product is only true when they are independent, not in general. Note that independence one way implies the other way. This is an iff.

Stats classes get this wrong. This is the implication, not the origin. They often define indepedence this way, which means nothing. The origin is that P(A) = P(A|B).

Back to this example:

```
P(Burg) = 0.2
P(Burg|Earth) = 0.2
```

Then we compute:

```
// There was an alarm, so burglary chance higher
P(Burg|Alarm) = 0.741
```

But once we know there was an Alarm, Burglary and Earthquake

are no longer independent.

```
// There was an earthquake, so it's more lik
ely to be the cause of the alarm
P(Burg|Alarm AND Earth) = 0.253
```

Whether things are independent is **dynamic** and is based on current evidence. To account for this, we generalize independence:

```
P(A|C) = P(A|C,B)
```

A is independent of B, given C (our current evidence and background info).

More derviation work, and we'll get:

```
P(A AND B | C) = P(A|C) * P(B|C)
```

// Alpha for A, Beta for B, Gamma for C in board notes

// I am never doing theoretical CS as a focus area. It's simply too
// disconnected from reality and interesting work.

True or false:

```
P(A) = P(A AND B) + P(A AND ~B)
```

Where B is any sentence. This is always true: we're splitting A into two parts, those that intersect with B, and those that do not. Draw the Venn diagram. This is the law of total probability or something like that.

Further rewriting from our previous identity that `P(A AND B) = P(A|B)P(B)`:

```
P(A) = P(A|B)P(B) + P(A|~B)P(~B)
```

This is **case analysis**: to figure out P(A), we pick a B, see how things are when B is true, and weigh that when B is false.

Next time: Bayesian networks.

# Lecture 15: 23 November 2015

## Probability Rules

From last time (lowercase 'w' is a world w_i):

Definition of probabilitity:

```
P(A) = SUM_{w |= A} P(w)
```

Bayes' conditioning:

```
P(A|B) = P(A AND B) / P(B)
```

And when B_1, …, B_n are mutually exclusive to each other, and this list is exhaustive (there are no overlaps between B_i, and the whole list of them cover all of A):

```
P(A) = SUM_i^n P(A AND B_i)
     = SUM P(A | B_i) P(B_i)
```

This partitions A into parts among the B's. Even if we don't know P(A), we might know the probabilities of each of the B parts. This is also known as **case analysis** or the **law of total probability**:

```
[ B_1          B_3 ]
[    (B_4 (A)   )  ]
[ B_5              ]
```

(Here, A is composed of parts from the B regions around them)

Special case of this is when we only have two regions A and B in a Venn diagram:

```
(A()B)
```

We have:

```
P(A) = P(A AND B) + P(A AND NOT(B))
     = P(A|B)P(B) + P(A|NOT(B))P(NOT(B))
```

## Chain Rule

```
P(A_1 AND A_2 AND ... A_N) = P(A1|A_2, ..., A_n)P(A_2|A_3, ...)P(A_n)
```

This follows from Bayes' conditioning.

## Bayes' Rule

Bayes conditioning is not the same thing as Bayes' Rule:

```
P(A|B) = P(B|A)P(A) / P(B)
```

Example: A is having a disease. B is a symptom.

We can ask: P(A|B): given that I have this symptom, what's the probability I have this disease? This is cause given effect.

OR: P(B|A): given that someone has a disease, what's the probability that a patient has a symptom? This is effect given cause, and easier to find out in general.

Bayes' Rule is special in that we usually want the first (cause|effect), but we can find it out through the latter

(effect|cause), thanks to this rule.

Bayesian networks will automate what we'll try by hand.

The classic example, probability of a disease given a positive test result?

1/1000 people have this disease in general (chicken and egg for finding this out).

False positive rate: 2% False negative rate: 5%

## Steps to Find a Probability

Step 1. Define your variables:

```
T: test result  {positive, negative}
D: has disease {yes, no}
```

Or treat both as Booleans.

Step 2. translate our given information into formulas:

```
P(D = yes) = 1/1000     // General chance of
 disease

P(T = positive | D = no) = 2/100     // Fals
e positive

P(T = negative | D = yes) = 5/100
```

Step 3. Express the query in terms of our variables

```
P(D = yes | T = positive) = ?
```

So, let's use Bayes' Rule:

```
P(D = yes | T = positive)
    = P(T = positive | D = yes) P(D = yes) /
 P(T = positive)
```

But we have negative given yes; we want positive given yes. How do we get that? The complement! The conditioning part doesn't mess with P(A) + P(NOT(A)) = 1, so long as the condition is the same for both.

So we can get

```
P(T = positive | D = yes) = 1 - P(T = negati
ve | D = yes)
```

So, we have:

```
P(T = positive | D = yes) = 95/100
```

Now, let's finish filling the above Bayes rule:

```
P(D = yes | T = positive) = [(95/100)(1/1000
)] /
```

But where do we get P(T = positive) from? Notice that we can do a case analysis, where T = positive is considering with the condition of something else being true or false. Those conditional probabilties happen to be more available to us.

So we can figure this out as:

```
P(T = positive)
= P(T = positive | D = yes) P(D = yes) + P(T
 = positive | D = no) P(D = no)
```

We know these (P(D = no) = 1 - P(D = yes) = 1 - 1/1000), which gives us:

```
P(T = positive) = 2093/100,000
```

So finally, back in Bayes' Rule:

```
P(D = yes | T = positive)
    = P(T = positive | D = yes) P(D = yes) /
 P(T = positive)
    = (95/100)(1/1000) / (2093/100,000) = 4.
5%
```

So a test came back positive, and we get a 4.5% chance of disease? Why so low? In reality, notice that our initial chance of disease overall is 1/1000, so this is a big jump. Also, for now, we don't have a way of assessing the impact of our false positive/negative rates.

We can also just enumerate all probabilities:

```
D   T   P(blah)

y   +    0.95 * 1/1000
y   -    0.05 * 1/1000
n   +    0.02 * 999/1000
n   -    0.98 * 999/1000
```

From there, we can use the definition of probablity (sum up probabilities of models) or Bayes conditioning (as when we can multiply conditional by other) to compute any probability using just these two variables. All Bayesian networks do something like this to compute probabilities, but without constructing the whole table.

## Bayesian Networks

We will show that they are complete as well as consistent.

// See handwritten notes from this point onward.

Ad Astra Et Ultra