CS 161: Artificial Intelligence
Lecture Notes

Henry Genus

Spring 2020

Contents

1 Introduction

Artificial Intelligence is actually one of the oldest fields of computer science.
It seeks to understand and build intelligent agents that can:

1. Perceive
2. Understand
3. Predict
4. Manipulate
5. Learn

These broad goals are shared with philosophy, psychology, neuroscience, etc.
BUT we build rational systems, where they seek to understand human ones.

Each specific goal, however, is shared with more computational disciplines:

1. Perception? Linguistics
2. Learning? Statistics
3. Understanding/Prediction? Mathematical Logic

The three eras of AI:

1. Knowledge Representation & Reasoning
   (a) (model): logic
   (b) CS-focused
2. Machine Learning
   (a) (model & functions): probability
   (b) Statistics and Math-focused
3. Neural Networks
   (a) (functions): neural networks
   (b) Engineering-focused

Our initial goals were proposed by Alan Turing; he thus developed **The Turing Test:**
≡ "can an AI convince a human it is human?" This requires:

1. natural language processing
2. knowledge representation
3. automated reasoning
4. machine learning

This avoids physical interaction, so robotics and vision are unneeded.
Since we tend to avoid monolithic approaches, this test is not applied nearly as often in the modern day.

Overview of an AI agent:

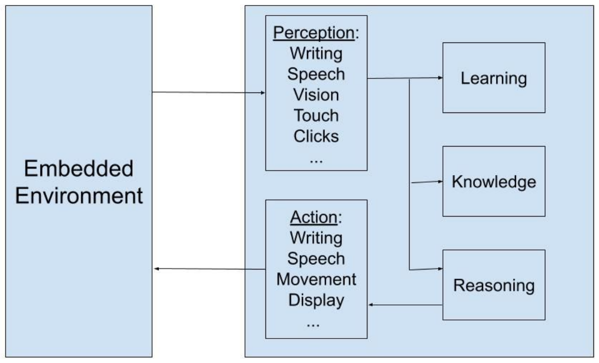

Embedded Environment
Perception: Writing Speech Vision Touch Clicks ...
Action: Writing Speech Movement Display ...
Learning
Knowledge
Reasoning

Figure 1: a course-grain view of an AI agent

How can we acquire knowledge?

1. from experts
2. by conversion from other forms of knowledge
3. from experience

How can we categorize knowledge?

1. Uncertain Knowledge (Opinion)
   - (a) Belief Networks
   - (b) Fuzzy Logic
2. Factual Knowledge (Facts)
   - (a) Propositional Logic
   - (b) First Order Logic (if A then B)

Why do we say A caused B?

1. Needed for explanation
2. Allows us to predict the future
3. Suggest ways to control future events
4. Moral responsibility
5. Legal liability

How can we formalize the reasoning process?

1. Deduction: What is implied by a knowledge base?
2. Belief Revision: What beliefs to prioritize?
3. Causality: What is the cause of an event?

Consider natural language processing:
It involves the following processes:

1. Understand Natural Language
2. Generate Natural Language
3. Text Summarization
4. Machine Translation
5. Speech Transcription

It has the following complications:

1. Syntactic Ambiguity: "They are cooking apples"
2. Semantic Ambiguity: "She ran to the bank"
3. Pragmatic Ambiguity: "Can you open the door?"

We use the following approaches:

1. Classical:
   - (a) Provide the system with rich knowledge
   - (b) Perform the above three types of analysis
   - (c) Disambiguate using knowledge and reasoning
2. Modern:
   - (a) Rely on corpus (collection of data)
   - (b) Disambiguate using machine learning

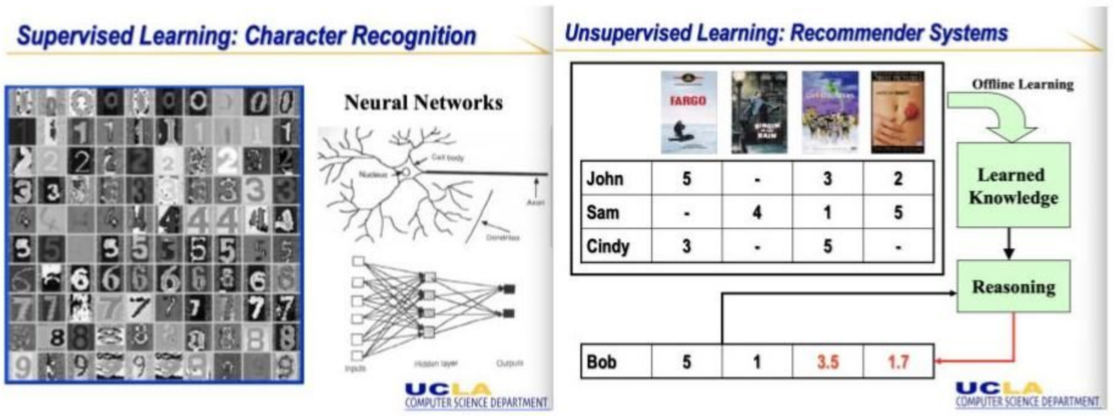
But what is machine learning?
$\equiv$ Using experiences and observations to improve future actions
Characteristics:

1. What aspects of performance to be improved?
   We must determine and consider:

   (a) Irrelevant aspects of the world
   (b) How the world evolves
   (c) What are desirable/undesirable situations

2. What feedback is available?
   (a) Supervised: give observations and actions they should lead to
   (b) Unsupervised: give observations, machine finds patterns
   (c) Reinforcement: give positive/negative feedback on actions

3. How to represent output?
   (a) Logical Knowledge
   (b) Probabilistic Knowledge/Bayesian Networks
   (c) Neural Networks


Supervised Learning: Character Recognition
Neural Networks
Nucleus
Cell body
Axon
Dendrites
Inputs
Hidden layer
Outputs
UCLA
COMPUTER SCIENCE DEPARTMENT
Unsupervised Learning: Recommender Systems
FARGO
Offline Learning
Learned Knowledge
Reasoning
UCLA
COMPUTER SCIENCE DEPARTMENT

John	5	-	3	2
Sam	-	4	1	5
Cindy	3	-	5	-

Bob	5	1	3.5	1.7

Figure 2: An example deep learning process

In implementing AI, an important part is **planning**: $\equiv$ finding a sequence of actions that will achieve a goal.

We must consider:

1. Input:
   (a) Actions (preconditions, effects)
   (b) Initial & Goal States
   (c) Knowledge of world (physics)

2. Output:
   (a) Conditional/Contingency Plan
   (b) Sensorless (Conformant) Approach
   (c) Hybrid

The largest application in the public conscience is **robotics**; Physical agents that perform tasks by manipulating the world. These are equipped with:

1. Effectors: legs, wheels, joints, grippers
2. Sensors: cameras, ultrasound, gyroscopes, accelerometers

Common categories:

1. Manipulators (robot arms): assembly lines, ISS
2. Mobile robots: unmanned probes/vehicles
3. Mobile robots with manipulators: humanoid

2 LISP

- is one of the oldest languages (1958)
- is the second high level language developed (behind FORTRAN (1957))
- was introduced for symbolic manipulation
- is a functional language
- has a uniform and simple syntax

Upon starting the LISP IDE, the user sees a listener (>).
The user then gives a LISP expression of the form:

```
(op arg1 ... argn) ; where op is a function name or special operator
```

Types of expressions:

1. Numeric Expressions
2. Symbolic Expressions
3. Boolean Expressions

Other Material:

1. Branching
2. Function Definition

and that is LISP!
Lisp uses prefix notation for operations:

```
; Mixing of types in an expression is allowed:
> (+ 2.7 10)
> 12. 7
; There is no limit on the argument count
> (+ 21 35 12 7)
> 75
; Expressions can be recursive
> (+ (* 3 51) (- 10 6))
> 19
```

We introduce two special operators:

1. The quote operator gets an expression without evaluation.
   We use this since sometimes lists represent data rather than expressions.

   ```
   > (quote (+ 3 1))
   > (+ 3 1)
   ; this can also be written "> '(+ 3 1)"
   ```

2. The setq operator assigns a value to a variable.

   ```
   > (setq x 3)
   > 3
   > (setq y (+ 1 3))
   > y
   > 4
   ; we can use this to bind an expression to a variable
   > (setq z '(+ 1 3))
   > (+ 1 3)
   ; expressions in expressions are evaluated
   > (setq z y)
   > 4
   ```

More generally, a LISP expression is either

1. an atom (number, symbol, string, etc)
2. a list (>= 0 expressions)

A common way to use lists/expression is to represent data; we then need:

1. accessors (car, cdr) = (first, rest)
2. constructors (cons, list)

The accessors car and cdr behave as follows:

```
> (setq x '(a b c d)) // this would be an error without the quote
> (a b c d)
; car (or first) gives the first item in a list
> (car x)
> a
; cdr (or rest) gives the rest of the list
> (cdr x)
> (b c d)
; how would I print the second element?
> (car (cdr x)) // shorthand: (cadr x)
> b
; we can recurse indefinitely here
> (caddr x)
> c
```

The empty list is denoted “NIL” and evaluates to false.
NIL functions expectedly with car/cdr

```
> (car NIL)
> NIL
> (cdr NIL)
> NIL
> (cdr '(c))
> NIL
```

The constructors list and cons behave as follows:

```
; list joins the following elements
> (list 1 2 3)
> (1 2 3)
; cons joins the two parameters, such that arg1=car \& arg2=cdr
> (cons 'a '(b c))
> (a b c)
> (cons (+ 1 2) '(b c))
> (3 b c)
> (cons 'a NIL)
> (a)
> (cons '(a b) '(c d))
> ((a b) c d)
> (cons 1 (cond 2 NIL))
> (1 2)
```

Boolean expressions use NIL for false and t for true

```
> (> 3 1)
> t
> (< 3 1)
> NIL
; listp evaluates type of element
> (not (listp 3))
> t
> (listp '(a b))
> t
; there is an =NIL operator 'null'
> (null (cdr '(2)))
> t
; note that it uses the most recent true value for return
> (OR NIL 3)
> 3
> (OR NIL (cdr '(c)) 7)
> 7
```

We can evaluate equality in one of three ways:

- '=' compares integers
- 'equal' compares elements (or direct values)
- 'eqL' compares underlying pointers

Now onto the big stuff......

Functions

```
> (defun name (parameters)
      operations))
```

A very simple function to start with is the “square” function

```
> (defun square (x)
      (* x x))
> (square 3)
> 9
```

We can use the “cond” keyword to emulate a switch

```
> (defun odd (x)
      (cond ((= x 0) NIL)
        ((= x 1) t)
        (t (odd (- x 2)))))
```

In Lisp, we tend to write functions that fall into one of three types:

1. numeric (as above)
2. list
   (a) use accessors
   (b) use constructors

A quick note on numeric functions:

```
; we bind variables with the keyword "let
> (defun foo (x y)
      (let ((a (+ x y))
        (b (* x y)))
        (/ a b)))
; this binding is local to the scope of the operation
> (setq x 5)
> (+ (let ((x 3)) (+ x (* x 10))) x)
> 38
; binding is done in parallel
> (setq x 2)
> (let ((a (+ x -1))
      (b (+ a 3)}
; GIVES AN ERROR: let* binds in series \& would allow the above:
> (let ((x 3) (y (+ x 2))) (x + y)
> 12
> (let* ((x 3) (y (+ x 2))) (x + y)
> 15
```

We consider some accessor functions:

1. compute the sum of all the elements of a list:

   ```
   > (defun sumlist (L)
   (cond ((null L) 0)
     (t (+ (first L) (sumlist (rest L)}
   ; a curly brace denotes "close all paren"
   > (sumlist ‘(1 2 3))
   > 6
   ```

2. determine whether an element is a member of a list:

```
> (defun member (x L)
      (cond ((null t) NIL)
        ((equal x (first L)) t)
        (t (member x (rest L)}
> (member NIL ‘(a b))
> NIL
> (member ‘(a b) ‘(1 7 (a b) d))
> t
```

3. return the last element in a list

```
> (defun last (L)
      (cond ((null (rest L)) (first L))
        (t (last (rest L)}
```

We consider some constructor functions:

1. remove one occurrence of an element from a list

```
> (defun remove (elm list)
      (cond ((null list) NIL)
        ((equal elm (first list)) (rest list)
        (t (cons (first list) (remove elm (rest list)}
```

2. append a list to another

```
> (defun append (L1 L2)
      (cond ((null (last L2)) L2)
        (t (cons (first L2) (append L1 (rest L2)}
```

3. reverse a list

```
> (defun reverse (L)
      (cond ((null L) NIL)
        ((null (rest L) (first L))
        (t (append (reverse (rest L) (first L)}
```

3 Problem Solving as Search

Search problems work as follows:

$$\begin{tikzcd}
\boxed{\text{Search Problem}} \arrow[r] & \boxed{\text{Search Engine}} \arrow[r] & \text{Solution} \\
& \boxed{\text{Search Strategy}} \arrow[u] &
\end{tikzcd}$$

Solving a search problem consists of answering the following two questions:

1. How do you format a problem as a search problem?
2. What search strategy do you use (uninformed/blind or informed/heuristic)?

Search problems have 3 major parts:

1. Initial State
2. Final/Goal State
3. Actions/Operators

The problem takes the form:

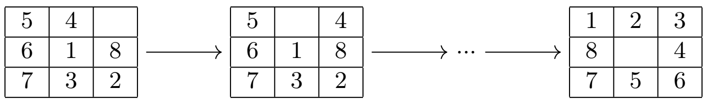
$$\begin{tikzcd}
S_0 \arrow[r, "a_0"] & S_1 \arrow[r, "a_1"] & \cdots \arrow[r, "a_{f-1}"] & S_f
\end{tikzcd}$$

with the solution

$$[a0, a1, \ldots, a_{n-1}]$$

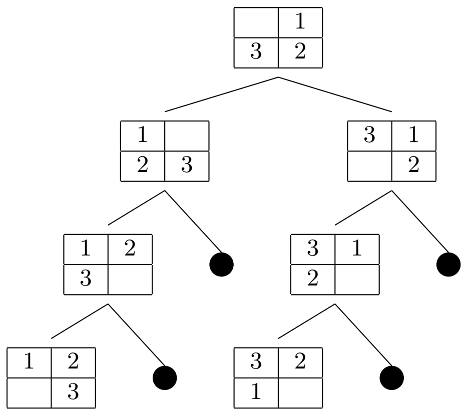
We will introduce a few "toy problems" to learn to answer these:

8 PUZZLE



Let's work through this; it is tempting to say we can move any tile in one of 4 directions,
BUT we need to simplify — that is too many choices.
Instead, we think about moving the empty space... now we have 4 choices max!
Let's build a search tree for a 2x2 version of the problem:

- a path is a sequence of choices
- each action has a cost (all 1 in this example)
- min cost of this example is 3
- since the tree is infinite, other paths work
- dots represent dead-end repeated paths



When we perform a search like this, we can see that the tree growth exponentially with height.
What determines the complexity of a search problem?

1. branching factor (average choices)
2. number of possible states
3. depth of solution

We were able to just walk through the last problem because it is so short;
let's consider a more complicated one:

MISSIONARIES AND CANNIBALS
We have three missionaries and three cannibals with a boat on one side of a river.
We wish to ferry everyone across, but the boat can only ferry up to two people at once.
We cannot let there be more cannibals than missionaries in any location.
The third statement in this problem is called a constraint; any state violating this constraint is invalid.

We will lay out the solution to this problem using LISP:

```
; We must first answer the question: How do I represent a state?
> (M C B)
; M & C represent people on boat side, B is bool for (boat on right)
let initial-state = (3 3 t)
let final-state = (3 3 NIL)
; we define a final-function that checks for final-state equivalence
> (fin-fcn '(3 3 t))
> NIL
> (fin-fcn '(3 3 NIL))
> t
; How do I change states? We define a successor function:
> (succ-fn '(3 3 t))
> ((0 1 NIL)
    (1 1 NIL)
    (0 2 NIL))
; as we can see, only three of the five choices are valid
```

We must therefore pass three things to the function:

1. the initial state
2. the final-state function
3. the successor function

We can now move into a slightly more abstract level of problem:

Constraint-Satisfaction Problems (CSP)

These problems have no known solution on start, BUT we know the finality of the search tree!

N QUEENS PROBLEM
We wish to place N queens on an NxN chess board such that none can capture any other.
We use the following definitions:

- STATE – partially-filled board
- INITIAL STATE – empty board
- FINALITY FUNCTION – none of the queens are on the same row, column, or diagonal

There are many other forms of problem that can be solved by a blind search:

- ROUTE FINDING
  - geographic navigation
  - computer network routing
  - automated travel advisory systems
- TRAVELING SALESMAN
  - street-cleaning
  - mail-delivery

- VLSI LAYOUT (on chips)
  - cell layout (group components into calls)
  - channel routing (between cells)
- ROBOT NAVIGATION
  - continuous navigation on flat plane
  - navigation of limbs and components
- SCHEDULING
  - automatic assembly sequencing
  - protein design

4 Blind Search Strategies

TERMINOLOGY:

- A **node** in a graph represents state, parent, action, and path-cost.
- **Expanding** a node involved **generating** children & a goal test.
- The **fringe/frontier** is the set of reachable nodes yet to be expanded.
- We define a **strategy** as the criteria for choosing the next node to expand.

To decide on a search strategy, we need some criteria to evaluate them.
We have four major criteria:

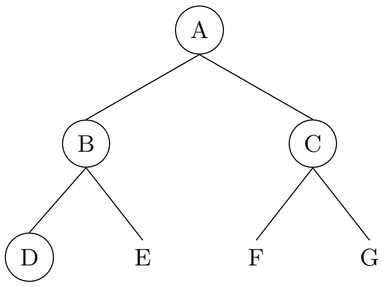
1. **completeness**: does it find a solution (if one exists)?
2. **optimality**: does it always find the best solution?
3. **time complexity**: number of worst case nodes expanded
4. **space complexity**: maximum number of nodes stored in memory

What do we use to measure these?

- b – max branching factor
- d – depth of least cost solution
- m – max depth of trees

Let's consider our two basic search algorithms:

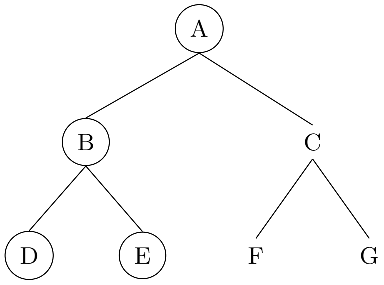
Breadth-First Search



Queue:
A
BC
CDE
DEFG
EFGHI...

1. Complete? Yes
2. Optimal? Yes
3. Time? $O(b^d)$
4. Space? $O(b^d)$

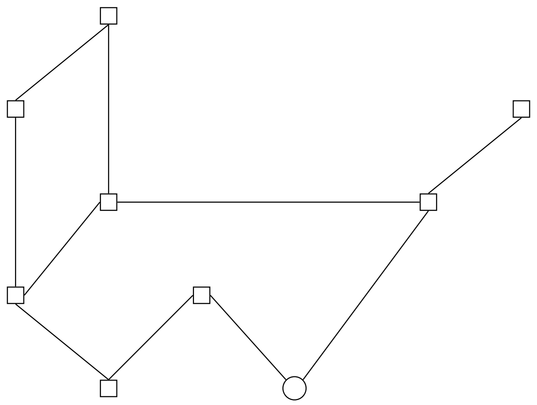
Depth-First Search



Stack:
A
CB
CED
CE...

1. Complete? No
2. Optimal? No
3. Time? $O(b^d)$
4. Space? $O(bm)$

We can't seem to improve the time complexity, but can we improve by blending BFS & DFS?
Consider a navigation problem:



We can see there are only 9 locations, so we can limit our path length!
We can use this to define a new algorithm:

Limited-Depth Search

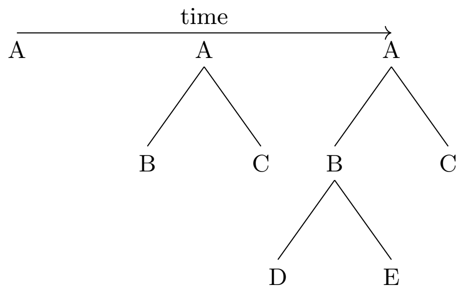
w/ length cutoff L, using priority queue (early = high priority)

1. complete? No
2. optimal? No
3. time? O(b^L)
4. space? O(bL)

We can extend this iteratively, increasing the length each round, for:

Iterative Deepening

where L denotes the iterative length.



1. Complete? No
2. Optimal? No
3. Time? $O(b^l)$
4. Space? $O(bl)$

But shouldn't we expect the time complexity to increase from limited-depth search?

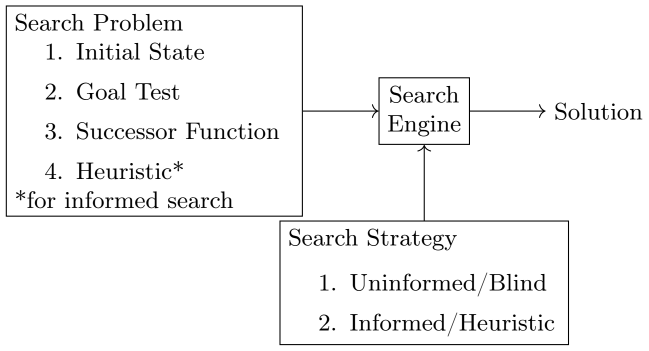
$$t = b^0(\frac{b}{b-1}) + b^1(\frac{b}{b-1}) + ... + b^d(\frac{b}{b-1})$$

$$\begin{aligned}\text{Constant} &= (b=2) \rightarrow 2 \\ &= (b=3) \rightarrow 1.5 \\ &= (b=10) \rightarrow 1.1\end{aligned}$$

Therefore the constant approaches 1!

5 Heuristic Search

We can now expand on our earlier search problem model:



We will begin our discussion of heuristic searches by discussing best-first search.
This is uninformed, but will lead us naturally into heuristics.

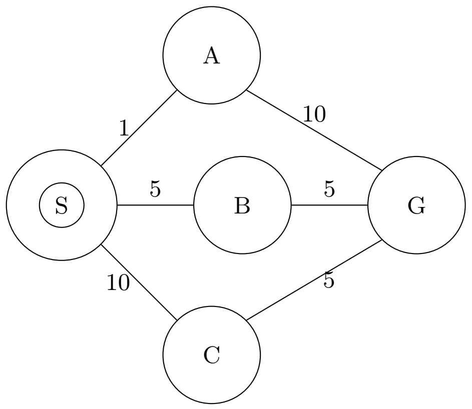
Uniform-Cost Search (UCS)

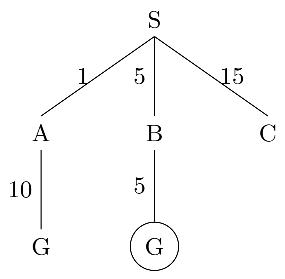
- uninformed generalization of BFS to weighted graphs
- expands based on contours of uniform cost and shape
- we goal check on expansion and not generation or our solution may be suboptimal
- we expand according to the function g(n) := cost of actions from initial state to node n

Properties:

1. complete? YES
2. optimal? YES
3. time? $O(b^{ceil(C*/E)})$
4. space? $O(b^{ceil(C*/E)})$

where E = minimal action cost & C* = optimal solution cost





Unfortunately, this algorithm tends to wander a bit; we want a smarter algorithm!

Greedy Search

- a modified UFS based on an estimate of the distance to the goal state h(n)
- h is an **admissible** function $\equiv h(G) = 0$
- h is a **consistent** function $\equiv h(n) \leq$ actual cost
  This is stricter than the admissible requirement, but is hard to avoid in practice.

Properties:

1. complete? NO (for example: A —1— B —2— C)
2. optimal? NO
3. time? $O(b^m)$
4. space? $O(b^m)$

Let's compare our two algorithms:
UCS: g(n): actual cost to get to n from initial state

- optimal
- conservative
- slow

Greedy: h(n) estimate of cost to get to final state from n

- non-optimal
- aggressive
- fast

Neither of these has all the properties we want; can we get the best properties of both?
It turns out YES: we just have to add them directly!
f(n) = g(n) + h(n) estimates the total cost & is admissible (provided h(n) is admissible)
This leads us to a very important algorithm:

A^* Search

- UCS based on f(n) = g(n) + h(n)
- forms contours which slim approaching goal
- prunes nodes outside contours
- we can prove that this algorithm is optimal:

Let's discuss the choice of heuristic: can we find a good heuristic for every problem?
h(n) = 0 works for every problem, but this makes the algorithm UCS!
Therefore, we want the maximum <u>admissible</u> heuristic for a given problem.

We will now consider a concrete example: 8 PUZZLE
We have two heuristics we would like to consider:
$h_1(n)$ = # pieces out of place
$h_2(n)$ = Manhattan Distance ≡ horizontal + vertical distance
$h_2(n) \leq h_1(n)$ for all n, so we say h_2 dominates h_1
Clearly h_2 is the better choice, but how much better can it really be?

We use two properties to evaluate search $A*$ search algorithms

1. effective branching factor (b^*) := avg branches per node
2. node count $(N) = 1 + (b^*)^0 + (b^*)^2 + \ldots + (b^*)^d$

d	IDS	$A^*(h_1)$	$A^*(h_2)$
4	N = 112; $b^* = 2.35$	N = 13; $b^* = 1.48$	N = 12; $b^* = 1.45$
6	N = 680; $b^* = 2.87$	N = 20; $b^* = 1.34$	N = 18; $b^* = 1.30$
8	N = 6384; $b^* = 2.73$	N = 39; $b^* = 1.33$	N = 25; b^* 1.24
10	N = 47,127	N = 93	N = 39
12	N = 3,644,035	N = 227	N = 73

Clearly, A^* is much more efficient, and h2 is much more efficient than h1;
how do we formally evaluate this?
time cost = $O(b^{\triangle})$, for absolute error $\Delta = h(n) - h^*$
$\implies$ hypothetical min $O(h^{\epsilon})$, for fractional error $\epsilon = (h * -h)/h^*$ is the

6 Constraint Satisfaction

Constraint Satisfaction Problems are a subclass of problems where

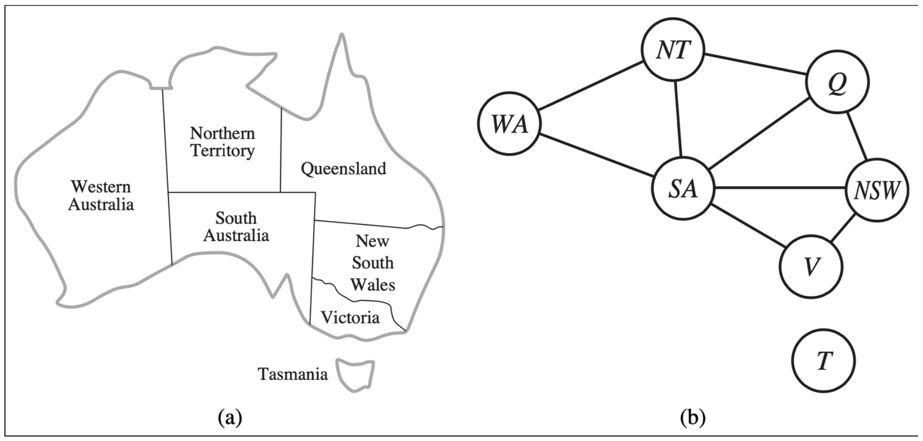
- the path to the solution is not relevant
- the state space is sufficiently large

We solve these problems by breaking up the state space into smaller atomic elements.

A constraint satisfaction consists of a few parts:

1. variables $X_1, ...X_n$
2. domain $D_1, ..., D_n$ of values for each X to be assigned
3. constraints $X_i D_i$ | consistent assignments

Consider a state map of Australia:



We wish to color the map in 3 colors such that adjacent states differ in color.
{X} = {WA, NT, NSW, V, SA, T}
D = {R, G, B}
WA != ST...
We can visualize this problem using a constraint graph, since constraints are binary.
This is as opposed to unary constraints (single element rules).
The Rules:

1. each variable forms a node
2. each constraint forms an edge

Constraint Satisfaction Problems seem specific, but the algorithms end up very generalizable.
There are two categories of problems

1. **Hard CSP** ≡ no violation of constraints allowed
2. **Soft CSP** ≡ some constraints are preference constraints and not required.
   (These are also called constraint optimization problems)

There is one major form of problem where contests are held for who can solve it the best:

Satisfiability (SAT)

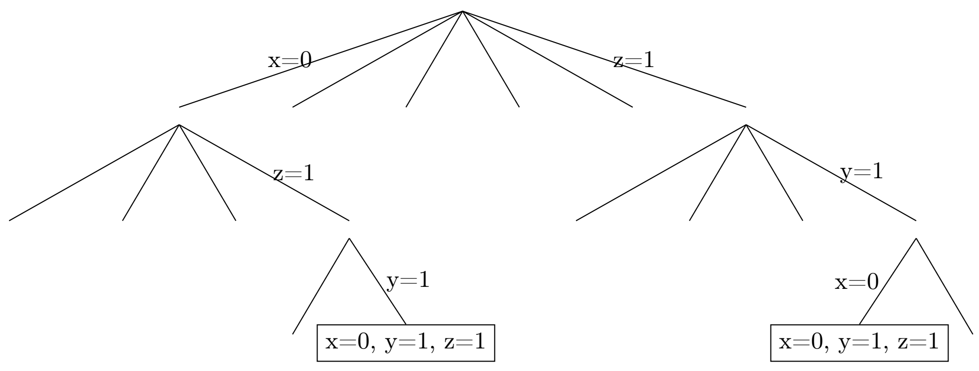
Variables: $X_1, X_2, ..., X_n$
Values: $1, 0$
Constraints: $\{X_1 \vee \neg X_2 \vee X_4, X_2 \vee \neg X_3 \vee X_5, ...\}$

This was the first and is now the prototypical NP-Complete problem!
Complete:

- the "hardest" problem in a given complexity class
- if you can provide a solution, you can solve all problems in the class

Formulating CSP as Search

Let's walk through an example tree w/ var $= x, y, z$ & $D = 0, 1$:



This is terrible and would be unfeasible; luckily we can note there are only 8 complete states!
$\implies$ If we let nodes be variables and edges values with DFS, we get time $O(d^n)$ & space $O(dn)$.

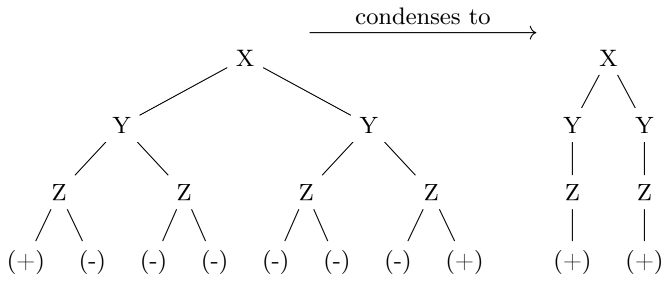
A straight BFS may not be optimal, however, as we may violate constraints early;
thus we backtrack, failure testing at intermediate nodes.

We can improve the efficiency of backtracking by answering a few questions:

1. Which variable do we assign next?
2. Which value should we try next?
3. Can we detect failure early?
4. Can we take advantage of the problem structure?

Variable & Value Ordering

Consider the following problem:
Variables: x, y, z
Values: $0, 1$
Constraints: $x = y, y = z$



Note that if we use the tree to the left, we must search the entire tree.
If we reorder the variables, however, we can prune the tree and use the right tree.

When it comes to ordering,
Variable ordering changes branch count.
Value ordering changes branch order.
We can choose variables and values based on a few models:

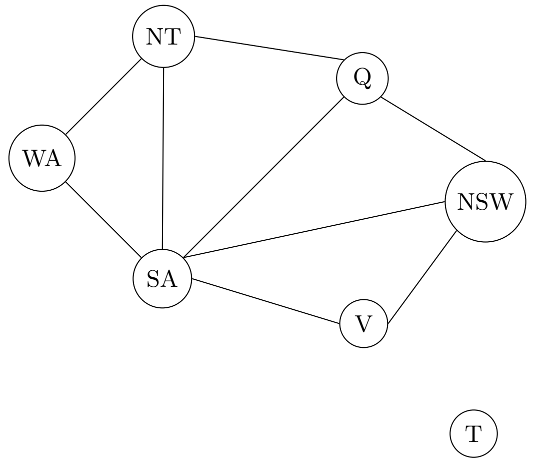
1. **Most Constrained Variable** $\equiv$ choose the variable with the minimum remaining values
2. **Most Constraining Variable** $\equiv$ choose the variable that places the most constraints
3. **Least Constraining Value** $\equiv$ choose the value that places the least restrictions

We choose this model heuristically via direct time comparison.

This leads us toward our methods of early failure detection
Forward Checking – less effective but more efficient
Arc Consistency – more effective but less efficient

Forward Checking

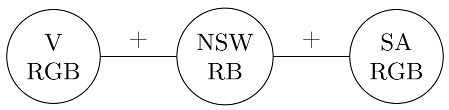
≡ keep track of the consistent values unassigned variable; if a variable has no values left, then failure!



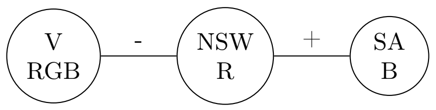
WA	NT	Q	NSW	V	SA	T
(R)GB	~~RG~~B	R(G)B	R~~GB~~	RG(B)	~~RGB~~	RGB

Arc Consistency

≡ make all arcs consistent; if we can, then assign arbitrarily left to right, else fail.
An arc is **consistent** if for all values of x, there exists a corresponding value of y;
we can therefore propagate constraints across the arc from the right.



This arc is **consistent**



This arc is **inconsistent**

ripple effect ≡ check an arc when a value is removed.
Single ripple effect: $O(d)$
A binary CSP arc count: $O(n^2)$
Checking consistency of an arc: $O(n^2)$
$\Longrightarrow\ T = O(n^2 d^3)$ for a binary CSP; that is not bad!

Symmetry

Disconnection

Consider the AU coloring problem;
we know Tanzania is disconnected, so we can assign it a value arbitrarily.
As a general rule, we can solve disconnected components of a search problem separately.

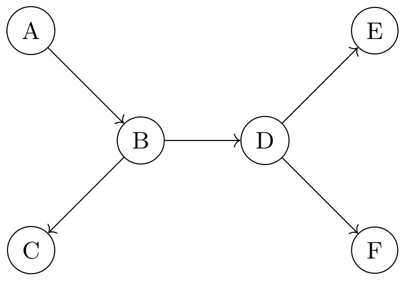
Consider a search problem with n=80 & d=2 @ 10 million nodes per second;
the time complexity without division is $2^{8}0$ or 4 billion years!
breaking the problem into 4 disconnected reduces the time cost to $4 * 2^{2}0 = 0.4$ seconds!

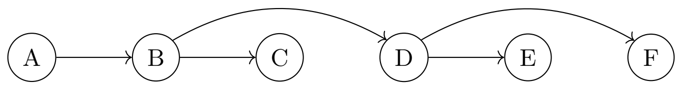
Trees

Tree structured CSPs are especially simple to solve.
Algorithm:

1. Orient the tree
2. Make arcs consistent
3. Assign values

We call the resulting structure a **backtrack tree**.
Since the arcs are consistent, we can assign any permissible value!





As an addendum, we can introduce symmetry via a constraint (such as $x < y < z$);
this greatly constricts the problem size, BUT deleting all symmetry is NP-Hard!

7 Two Player Games

A few things set two-player games apart from standard search problems:

- Opponents — the next stage is unpredictable, so we need a strategy
- Time Limits — search time is limited, so a good, fast result is better than a perfect slow one

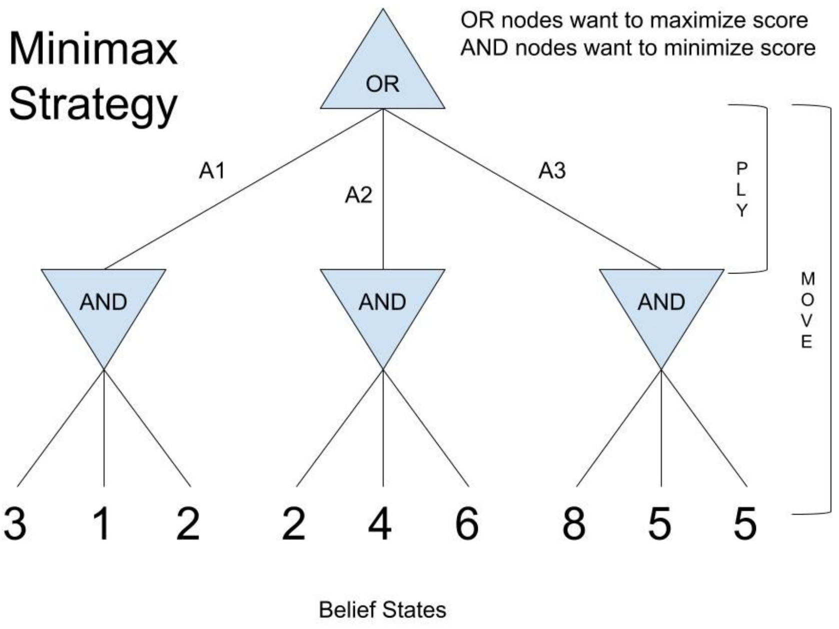
Formulating 2-Player Games As Search

Consider Chess; we have two major issues to address:
The effective branching factor is roughly 35, so the tree is huge.
We don't know the moves the opponent will make.

What we need is a **conditional plan** (equivalent to a strategy).
This solution will thus include:

1. a goal state at every leaf
2. an action at each of our turns
3. a path for each of the foe's choice

Minimax



We only need an algorithm which chooses from {A1, A2, A3}, since we can reapply each turn.
We assume that the opponent is perfect, since if they aren't we will only do better.
If choose A_1 — then our opponent will choose 3, thus we want to choose the best worst case.

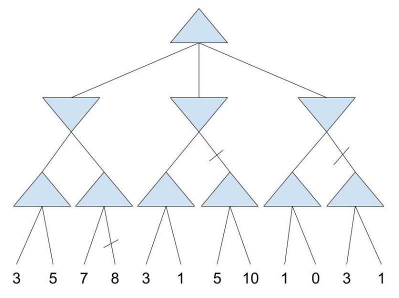
We consider this approach bottom-up, since we compute scores starting at the bottom.
How do we build the tree? DFS.
Recall space = O(bm) & time = O(b^m)

Consider chess: it has a branching factor of roughly 35.
The time limit is often 100 seconds.
We can generate 10^4 nodes/second.
$\implies$ choices = $log_3 5(10^6) = 4$, so our approach is 2 PLY. Is that good?
4-ply look ahead — human novice
8-ply look ahead — human master
12-ply look ahead — Kasparov, Deep Blue
Thus our method is not nearly sufficient.

We need to "cut m" via an **evaluation function**
≡ a function which attempts to approximate the minimal node value.
Requirements for the function:

1. agrees with the utility function on terminal nodes
2. is efficient to compute
3. is accurate in premise

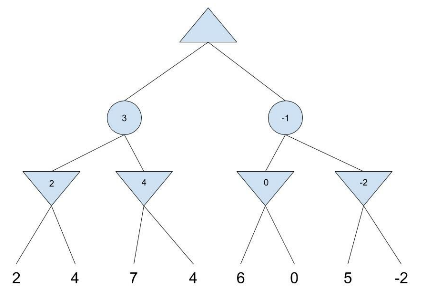
Note that this may lead us to cut off paths at different steps.



We can skip any value lower than the min!
This is called **alpha-beta pruning**
(alpha for max, beta for minimum).
Our behavior is dependent on the type of node:
min node — examine children lowest-first
max node — examine children largest-first
The gain depends on value order.
$\implies$ with a-b pruning, $T = O(b^{m/2})$.
This is up to twice as deep!
This is the best case, the average is $O(b^{3m/4})$.

Stochastic Search

This strategy cannot work for games of chance since we cannot predict our choices.
We can instead use probabilities to weight the values.



Our choice of cutoff is very important:
A depth cutoff in a search must be **quiescent**
≡ unlikely to drastically change values.
A bad cutoff may fall victim to the **horizon effect**
≡ when negatives are pushed just past the cutoff.
We address this by logging clearly great moves.
Time complexity: $O(b^m n^m)$ for n distinct events.

Additional Game Strategies

At the beginning or the end of a game, it may be more practical to do a table lookup.
≡ create a policy (a mapping from every single state to its optimal reachable state).
We can then do a **retrograde** (≡ bottom up) search from an ideal to choose our action.

Probabilistic & Perfect Information

1. define a chance node which holds the expected value of its children
2. **Monte Carlo Simulation** ≡ estimate values by win percentage

Probabilistic & Imperfect Information

1. **averaging over clairvoyance** ≡ solving over all possible "rolls"
   This has the negative tendency to avoid moves that gain the player information
2. introduce randomness for an equilibrium solution

8 Knowledge Representation

In the beginning of AI, there was a debate between symbolic and numeric representations.
Symbolic representation is now standard, though numeric representation is used in Bayesian Networks.

Once knowledge representation was decided, acquisition needed to be decided.
We can model knowledge, whether procedurally or declaratively.
We can learn knowledge from information in the world.
The former approach is often used in AI, whereas the latter is used in ML.
This is changing however; progress in modern day requires more broad knowledge.

We will be discussing the **classical model of logic**

$$\begin{tikzcd}
\boxed{\text{Statements in Logic}} \arrow[r] & \boxed{\text{Logical Deduction}} \arrow[r] & \text{Conclusions} \\
 & \text{Observations} \arrow[u] &
\end{tikzcd}$$

Consider the example problem of the WumpusWorld:

1. The grid contains a goal, pits, and a Wumpus
2. Cells adjacent to pits are breezy
3. Cells adjacent to the Wumpus smell
4. The Wumpus and pits kill you

P		
Breeze	OK	
A	Stench	W

The logic depends on two components:

1. Prior Knowledge (Rule/Problem Structure)
2. Observation (Learning/Breeze and Stench)

This is a canonical **logical** approach; ≡ if something is deduced, it MUST be true.
It is more complicated than many animals can do!

Propositional Logic

There are two major components to propositional logic:

1. **Syntax** (≡ Grammar)
2. **Semantics** (≡ Meaning)

Even if the syntax doesn't match, semantics can tell if two statements are equal

Syntax

A syntax consists of three elements:

1. boolean variables $(X_1, ..., X_N)$
2. logical connectives
   (a) $\wedge \equiv$ AND
   (b) $\vee \equiv$ OR
   (c) $\neg \equiv$ NOT
   (d) $\implies \equiv$ IMPLIES
   (e) $\iff \equiv$ IFF
3. Sentences
   (a) if S is a sentence, then $\neg S$ is a sentence.
   (b) if $S_1 \& S_2$ are sentences, then each of the following are compound sentences:
      - $S_1 \wedge S_2$
      - $S_1 \vee S_2$
      - $S_1 \implies S_2$
      - $S_1 \iff S_2$

We use a few terms to refer to elements of a syntax:

- $X, \neg X$ are called **literals**

- X is the **positive literal**
- $\neg X$ is the **negative literal**

- $X \wedge Y$ is called a **conjunction**
  - X, Y are called the **conjuncts**
- $X \vee Y$ is called a **disjunction**
  - X, Y are called the **disjuncts**
- $X \implies Y$ is called a **premise**
  - X is called the **premise antecedent**
  - Y is called the **implicant**
- $\neg Y \implies X$ is called the **contrapositive** (in relation to the premise)

We have a few logical rules to work with sentences:

1. $a \implies B = \neg a \vee B$
2. $\neg(a \wedge B) = \neg a \vee \neg B$
3. $\neg(a \vee B) = \neg a \wedge \neg B$

Let's try to capture the following state with logic:

	B_1	
B_2	P	B_3
	B_4	

Consider the example problem of the WumpusWorld:
We can represent this in (at least) two ways:

1. $P \implies (B_1 \vee B_2 \vee B_3 \vee B_4)$
2. $P \implies (B1 \wedge B2 \wedge B3 \wedge B4)$

The latter is **stronger**, since it is more specific.
The former is true, but doesn't model all we know.
This is a common problem when writing AI.

Normal Forms

Normal forms can be categorized in one of two ways:

1. a grammar is **practical/restricted** if there are logics it cannot represent.
2. a grammar is **universal** if it can represent any theoretical logic.

The normal forms we will cover are:

1. **Conjunctive Normal Form***
   $[(A \wedge \neg B) \vee (A \wedge \neg D \wedge E) \vee ...]$
   We call a disjunction of literals a **clause**.
   Therefore, this form is a **conjunction of clauses**.
2. **Disjunctive Normal Form***
   $[(A \vee \neg B) \wedge (A \vee \neg D \vee E) \vee ...]$
   We call a conjunction of literals a **term**.
   Therefore, this form is a **disjunction of terms**.
3. **Horn Form**
   $[(A \vee \neg B \vee \neg C) \vee (\neg A \vee \neg B \vee \neg C)]$
   A **horn clause** is a clause with at most 1 positive literal.
   A clause $A \vee \neg B \vee \neg C$ can be written as $B \wedge C \implies A$
   $B \wedge C$ is called the **body**; A is called the **head**.
4. **Negation Normal Form***
   $[(A \wedge \neg D \wedge E) \vee (A \vee \neg C \vee D) \wedge (A \wedge \neg B)]$
   Allows either clauses or terms, but only allows ($\neg$) on variables, not sentences.
   Are often represented as circuits.
   Is tractable iff all entries to all AND gates share no variables.
   This subset of NNFs are called **Decomposable (DNNF)**.

(* is used to represent grammars that are universal)
Fun fact: SAT on a CNF is hard but SAT on a DNF is easy.
Horn Form makes SAT linear, therefore Horn Form is **tractable**.

We can do these SAT problems in two ways:

2. **Forward Chaining** (data driven) $\equiv$ iterate over the clauses; if the body is true, infer the head
3. **Backward Chaining** (Goal Driven) $\equiv$ recursively attempt to prove the goal to the head's body

In the ideal case, backward chaining is much faster than even linear!

Semantics

Semantics are concerned with the notion of **worlds**.
$\equiv$ a set containing a fixed truth value for all variables/preposition symbols
We would like to know if a sentence holds in a given world.
If a holds in world w, we say $w \models a$ (equivalent to "world w contains a").

Consider the following example:

	E	A	B
W_1	T	T	T
W_2	T	T	F
W_3	T	F	T
W_4	T	F	F
W_5	F	T	T
W_6	F	T	F
W_7	F	F	T
W_8	F	F	F

Which worlds do the following hold in?

- E? $\{1, 2, 3, 4\}$
- $\neg E$? $\{5, 6, 7, 8\}$
- $\neg B$? $\{3, 4, 7, 8\}$
- $\neg B \wedge \neg E$? $\{7, 8\}$
- A? $\{1, 3, 5, 7\}$
- $((\neg E \wedge \neg B) \vee A)$? $\{1, 3, 5, 7, 8\}$
- $((E \vee B) \implies A)$? $\{1, 3, 5, 7, 8\}$

Therefore,

- $W_1 \vee W_3 \vee W_5 \vee W_7 \vee W_8 \models (E \vee B) \implies A$
- $W_2 \vee W_4 \vee W_6 \not\models (E \vee B) \implies A$

We can thus derive that for sets of worlds S and T, if $S \models a$ && $T \models b$, then

1. $S \cap T \models (a \wedge b)$
2. $S \cup T \models (a \vee b)$
3. $S/T \models (a \wedge \neg b)$

If we thus assert the property $(a \wedge b)$, we can prune the set of worlds down to $S \cap T$.
If we get to one world, we know all; if we get to none, we have an inconsistency.
We can use these properties to build a pretty effective SAT Solver.

We denote the **meaning/model** of $\alpha M(\alpha)$ and define it by $M(\alpha) = \{w | w \models \alpha\}$.
Thus we should be able to translate the following easily:

1. α is equivalent to $\beta \rightarrow M(\alpha) = M(\beta)$
2. α is contradictory $\rightarrow M(\alpha) = \{\}$
3. α is a **tautology**/is valid $\rightarrow M(\alpha) = \{w\}$
4. α & β are mutually exclusive $\rightarrow M(\alpha) \cap M(\beta) = \{\}$
5. α implies β ($\alpha \implies \beta$) $\rightarrow M(\alpha) \subset M(\beta)$

9 Classical Propositional Logic

We will attempt to systemically answer the following question:
 Does Δ imply α?

As well as the related questions:
 Is Δ equivalent to α?
 Is Δ satisfiable?
 Is Δ valid?

We will discuss four major methods:

1. Enumerating Models
2. Inference Rules
3. Search (CSP/SAT)
4. NNF Circuits

(1) and (2) are deprecated but (2) is fundamental to (3) and (4).
This lecture will discuss the first two, and the next two will be discussed in the next.

Inference By Enumerating Models

Suppose our knowledge base $\Delta = \{A, A \vee B \implies C\}$.
Let $\alpha = C$; we can then ask: Does Δ imply α?

1. convert our knowledge base to CNF
 $\Delta = A \wedge (A \vee B \implies C) = A \wedge (\neg(A \vee B) \vee C)$
2. build a truth table for the KB
3. use the models to solve:
 $M(\Delta) = \{W1, W3\}$
 $M(\alpha) = \{W1, W3, W5, W7\}$
4. If $M(\Delta) \subseteq M(\alpha)$, then $\Delta \models \alpha$

	A	B	C	Δ	α
W_1	T	T	T	T	T
W_2	T	T	F	F	F
W_3	T	F	T	T	T
W_4	T	F	F	F	F
W_5	F	T	T	F	T
W_6	F	T	F	F	F
W_7	F	F	T	F	T
W_8	F	F	F	F	F

Complexity: $O(2^n)$, n = # variables
(it can be used up to 25 variables!)
We can just as easily use this to solve the other questions.

Many logical systems demonstrate **monotonicity**.
 $\equiv$ as the information increases, the set of entailed sentences can only grow.
For these problems, enumeration can cause the set to explode;
 we need a model that can ignore irrelevant propositions.

Inference By Resolution

This method starts from the **Deduction Theorem**: Any question of implication can be converted into a SAT question.

If we wish to solve a SAT question, we can attempt to prove by contradiction. Say we want to prove that Δ implies α:

1. assume Δ & $\neg\alpha$
2. show that this causes a contradiction

To solve questions of satisfiability, we will use the concept of **inference rules**.
An inference rule is of the form $\frac{P_1, P_2}{P_3}$.
 This says "if P1 and P2, then P3".

To apply proof by contradiction to questions of satisfiability, we:

1. convert the implication to normal form
2. apply inference rules until we either terminate or have a contradiction

If we can use rule R for a given derivation, we use the symbol $\vdash_R$.
If a rule will derive α from Δ any time $\Delta \not\models \alpha$, we call the rule **complete**.
 Modus Ponens $\frac{A, A \implies B}{B}$ is not complete.
If a rule's denominator is always true provided the numerator, we call the rule **sound**.
 $\frac{A, (A \vee B)}{(A \wedge B)}$ is not sound.

We would assume a sound and complete rule would be needed for proofs,
BUT our primary rule is actually not complete:

$$\text{Resolution:} \quad \frac{A \vee B, \neg B \vee C}{A \vee C}$$

This may be counterintuitive, but we can see it holds:
$A \vee B = \neg A \implies B$
$\neg B \vee C = B \implies C$
$\neg A \implies B \ \&\& \ B \implies C = \neg A \implies C = A \vee C$
This is not complete, but it is **refutation complete** when applied to a CNF.
$\equiv$ given any CNF with a contradiction, it will derive the contradiction.

The Algorithm: We are asked the question "Does Δ imply α?"
This is equivalent to "Is $\Delta \wedge \neg\alpha$ a contradiction?".
We therefore apply resolution until either:

1. a contradiction occurs, in which case Δ implies α
2. no more resolution can be applied, in which case it does not

Example 1:
$\Delta : ((A \vee \neg B) \implies C) \wedge (C \implies D \vee \neg E) \wedge (E \vee D)$
$\alpha : A \implies D$
Does Δ imply α? $\rightarrow$ Is $\Delta \wedge \neg\alpha$ unsatisfiable?
The empty set cannot be true, so Δ implies α
Example 2:
$\Delta : ((A \wedge B) \implies C) \wedge A \wedge (C \implies D)$
$\alpha : C$
If we apply resolution, the algorithm will terminate; therefore, Δ does not imply α.

1. $\neg A \vee C$
2. $B \vee C$
3. $\neg C \vee D$
4. $E \vee D$
5. A
6. $\neg D$
7. $C < 0, 4 >$
8. $D \vee \neg E < 2, 6 >$
9. $E < 3, 5 >$
10. $D < 7, 8 >$
11. $\Phi < 5, 9 >$

This algorithm depends on a CNF, but how do we get one?

1. Get rid of all connections, but for $\{\vee \wedge \neg\}$
   - $A \iff B \rightarrow (A \implies B) \wedge (B \implies A)$
   - $A \implies B \rightarrow \neg A \vee B$
2. Use deMorgan's Law to push negatives inward
   - $\neg(A \wedge B) \rightarrow \neg A \vee \neg B$
   - $\neg(A \vee B) \rightarrow \neg A \wedge \neg B$
3. Distribute $\vee$ over $\wedge$
   - $(A \wedge B) \vee C \rightarrow (A \vee C) \wedge (B \vee C)$

Example: $A \iff (B \vee C)$

1. $\rightarrow (A \implies (B \vee C)) \wedge ((B \vee C) \implies A) \rightarrow (\neg A \vee B \vee C) \wedge (\neg(B \vee C) \vee A)$
2. $\rightarrow (A \vee B \vee C) \wedge ((\neg B \wedge \neg C) \vee A)$
3. $\rightarrow (\neg A \vee B \vee C) \wedge ((\neg B \vee A) \wedge (\neg C \vee A)$

Coincidentally, this is an NNF.

10 Modern Propositional Logic

Reducing Queries to SAT

Consider the sentence $\Delta = (A \vee B \vee \neg C) \wedge (\neg A \vee C) \wedge (A \wedge C \wedge \neg D)$
We treat each clause as a constraint, and solve as a CSP
$\rightarrow$ if $w = \{A = T, B = F, C = T, D = F\}$, then $w \models \Delta$

Thus we can see why SAT is the prototypical NP-Complete problem.
We will now discuss various methods to solve SAT problems.

Sat Solvers

There are two ways to solve SAT problems that are considered standard:

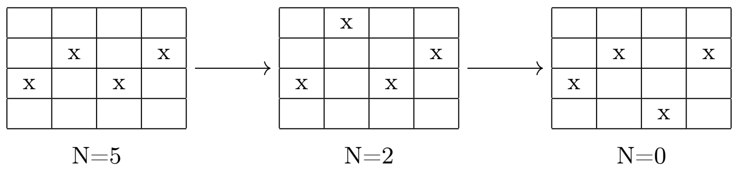
1. Backtrack Search (DFS + Failure Detection)
2. Local Search (Very Fast, Incomplete Search)

In the context of SAT, backtrack search is called DPLL (initials of the 4 creators).
There are a few tools commonly used to make DPLL faster.

1. uses a degree heuristic to determine value ordering
2. can learn from the past via conflict clause learning
3. can perform component analysis to break problems down
4. utilizes random restarts
5. utilizes unit resolution (resolution with a single term clause — linear & incomplete)

Local Search is much simpler and faster, but incomplete.

1. Guess a truth assignment
2. check whether the rule holds.
   YES → DONE NO → try again



It turns out that the N queens problem is simple for local search;
  This is because the solutions are densely distributed.

Solutions are densely distributed when a problem is **under-constrained**.
Solutions are sparsely distributed when a problem is **over-constrained**.
The hardest problems tend to be at the threshold of the two.

1. Hill Climbing

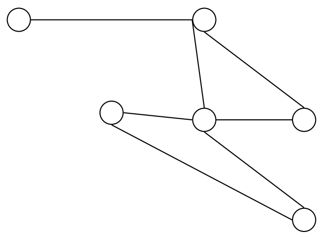
The difficulty of these algorithms comes in the way we determine what truth assignment to use.
There are two standard approaches to this: We can imagine the complete assignments in a graph.

Hill-climbing is choosing the next node by heuristic.
We have two major issues:

- local extrema
  solution: utilize random restarts
- side-moves solution: consecutive side-move limits

Our algorithm therefore takes the form of an embedded loop.
There is no guarantee this visits every world; we have two choices to deal with this:

1. allow the algorithm to complete — complete but can loop infinitely
2. terminate after a time limit — incomplete


neighborhood structure

2. Stochastic methods

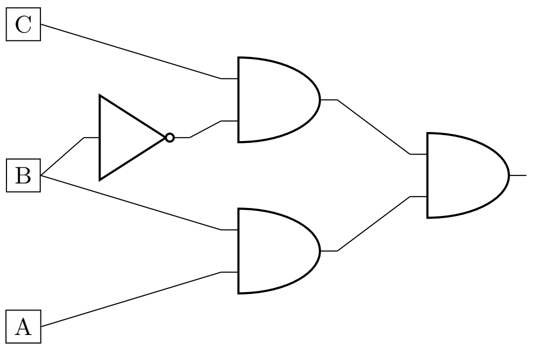
Stochastic ≡ probabilistic
A common approach is **simulated annealing**

```
While not at goal:
    pick a neighbor randomly
    if it is better, go there
    else, there is a % chance based on how much worse it is and the depth of the neighbor
End
```

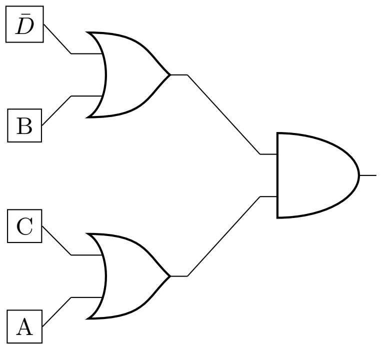
By exploring less when deeper, we can avoid local extrema and have a complete algorithm
(provided we allow the algorithm to run for infinite time).

Compiling Sentences To Tractable Circuits

Circuits are more compact than formulas, since circuits can reuse components!



DNF $(A \wedge \neg B) \vee (B \wedge C)$



CNF $(A \vee C) \wedge (B \vee \neg D)$

It therefore makes sense to use them to express complicated logic.

Tractable NNF Circuits

NNF circuits are not inherently tractable, but they are given certain restrictions.

If the sub-circuits of all AND gates in a circuit share no variables, we say it is **decomposable**
This property allows us to solve components of the circuit separately.
an AND gate is satisfiable iff all its sub-circuits are
an OR gate is satisfiable iff one of its sub-circuits are
If the circuit is decomposable, The circuit is tractable for SAT.

If the inputs to an OR gate are mutually exclusive, we say it is **deterministic**.
If a circuit is decomposable and deterministic, it is a d-DNNF circuit.
d-DNNF circuits are tractable for #SAT ("sharp-SAT").
#SAT counts the number of satisfying assignments in linear time.

If all sub-circuits of an OR gate share all variables, the gate is said to be **smooth**.
These can be easier to work with, but are logically equivalent to d-DNNFs.

Complexity:

- SAT is NP complete ≡ it is not thought to be solvable in polynomial time
- #SAT is #P complete ≡it involves counting the number of acceptable NP solutions
- Maj-SAT is PP complete ≡ a probabilistic algorithm is run a set # of times for a polynomial time

While these methods are increasingly comple, they all can repeat information over time or space.
We address a solution to this issue in the next lecture.

11 First Order Logic Representation

When we discussed propositional logic, we ran into a roadblock representing repeating data.
Consider the Wumpus World:
  Propositional: Each square needs a similar set of variables to represent all possible states
  First Order: $\forall$r Pit(r) $\implies$ [$\forall$ s Adjacent(r, s) $\implies$ Breezy(s)]
First order logic is thus much more expressive and succinct; Our main challenge will be computational.

Consider the representation of a world:
  Propositional Logic:

- Variables {A, ...}
- Values {T, F}

First Order Logic:

- Objects with Properties
- Relationships Between Objects
- Functions Mapping Objects to Objects

How would we represent each of the following in first-order logic?

1. One plus one equals two
    - Objects: One, One plus one, two
    - Properties: None
    - Relations: equals
    - Functions: plus
2. Squares adjacent to the Wumpus are smelly
    - Objects: Squares, Squares adjacent to the Wumpus, Wumpus
    - Properties: Smelly
    - Relations: adjacent
    - Functions: None

Syntax

The syntax of first order logic is much nicer than that of propositional logic

- constants — uppercase words that represent objects
  Examples include Z, Jack, UCLA, etc
- predicates — lowercase words that represent relations
  Examples include adjacent(), at(), etc
- property — single argument predicate
- equality — a key subset of predicates
- functions — lowercase words that give a value for each input
  Examples include leftLeg(), father(), etc

These are domain specific and form our "vocabulary".
Our domain-independent vocabulary is as follows:

- variables: x, y, z
- connectives: $\vee \wedge \neg \implies \iff$
- quantifiers: $\forall$ $\exists$

We can use all of these to define atomic sentences.
These are of the form: predicate (Term1, . . . , TermN)
  Term $\equiv$ a constant, variable, or function
  Ground term $\equiv$ term with no variables

The new operators in first-order logic are called **quantifiers**. Quantification comes in two forms

1. **UNIVERSAL QUANTIFICATION**
   FORM: $\forall$ variables sentence
   ex. $\forall$ x at(x, UCLA) $\implies$ smart(x) is a predicate – at(x, UCLA) is a relation – smart(x) is a property
   This forms a conjunction of the instantiations of the predicate, and often appears with '$\implies$'
   $\rightarrow$ [at(John, UCLA $\implies$ smart(John)] $\wedge$ [at(fatherOf(John), UCLA) $\implies$ smart(fatherOf(John))]

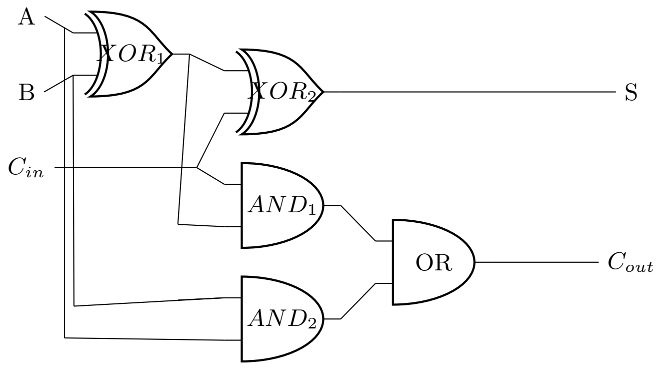
2. **EXISTENTIAL QUANTIFICATION**
   FORM: ∃ variables statement
   ex ∃ x at(x, UCLA) ∧ tall(x)
   This forms a disjunction of the instantiations of the predicate, and often appears with '∧'
   → [at(John, UCLA) ∧ tall(John)] ∨ [at(fatherOf(John), UCLA) ∧ tall(fatherOf(John))]

Quantification is not always commutative:
$\exists\, x\, \exists\, y = \exists\, y\, \exists\, x$
$\forall\, x\, \forall\, y = \forall\, y\, \forall\, x$
$\forall\, x\, \exists\, y! = \exists y\, \forall\, x$
Why? consider:
$\forall\, x\, \exists$ y loves(y, x) means "everyone in the world has at least one person who loves them"
$\exists\, y\, \forall$ x loves(y, x) means "there is at least one person who loves everyone in the world"
We can, however, use one operator to simulate the other:
¬∀ x likes (x, IceCream) = $\exists x \neg$likes (x, IceCream)

Asserting a number of unifications tends to be a bit trickier
"Spot has two sisters"
→ $\exists\, x\, \exists$ y sister(x, spot) ∧ sister(y, spot) ∧ x != y
"Spot has exactly two sisters"
→ We use the above statement, plus ∀ z sister(z, spot) $\implies$ ((z = x) ∨ (z = y))
Which can also be written as ¬(∃ z sister(z, spot) ∧ ((z = x) ∨ (z = y)))
Some people make this cleaner by using to represent "exists a unique"
∃! x king(x) = [∃ x king(x)] ∧ [∀ y king(y) $\implies$ (x = y)]
(this is an error in actuality, since the x on the left is out of scope)

We can see it is critical to develop well formed formulas with no free variables.

Consider the 1-bit adder:
We want to derive the output given the input.
This is hard to do with circuits, but this is easy with first order logic.



Our vocabulary consists of:
domain:
constants: AND, OR, NOT, XOR, 0, 1
functions: type(g), signal(i, o), in(g), out(g)
predicates: connected(g_1, g_2)
instance (the specific layout of this circuit):
constants: $XOR_1, XOR_2, AND_1, AND_2, OR_1$
We then define our knowledge base:
Domain:
∀ t1, t2 connected(t1, t2) $\implies$ (signal(t1) = signal(t2))
∀ t1, t2 connected(t1, t2) $\iff$ connected(t2, t1)
∀ g type(g) = OR $\implies$ [signal(out(1, g)) = 1 $\iff$ ∃ n signal(in(n, g)) = 1]
(similar rules for other gates are omitted for the sake of brevity)
(the most general part follows)
∀ t signal(t) = 1 ∨ signal(t) = 0
¬1 = 0
Instance:
type(XOR_1) = XOR, type(XOR_2) = XOR, ...
connected(out(1, XOR_1), in(2, AND_2)), ...

This is all we need to begin queries!

$\exists i_1, i_2, i_3$ signal(in(1, adder)) = 1
= $i_1 \wedge$ signal(in(2, adder))
= $i_2 \wedge$ signal(in(3, adder))
= i_3 $\wedge$ signal(out(1, adder))
= 0 $\wedge$ signal(out(2, adder))
$\rightarrow$ {(1, 1, 0), (1, 0 1), (0, 1, 1)}

We could expand this to check if a circuit is functioning & to diagnose errors with:
ok(g) — represents whether the circuit is ok
stuck(g) — represents whether a gate is always off and stuck on 0
We may even want to include wires if that is what we're testing.
All of this is called Knowledge Engineering.

12 First Order Logic Inference

To do computation, we must first convert from first-order logic to propositional logic.
In general, we do replacement, written as: {x/y} — replace x with y.
We must handle instantiation uniquely, however:

1. UNIVERSAL INSTANTIATION
   (∀ variable) (sentence) subst(variable/ground-term, sentence)
   ex: for constants {John, Richard}:
   $\forall$ x King(x) $\land$ Greedy(x) $\implies$ Evil(x)
   $\rightarrow$ King(John) $\land$ Greedy(John) $\implies$ Evil(John)
   $\rightarrow$ King(Richard) $\land$ Greedy(Richard) $\implies$ Evil(Richard)
   $\rightarrow$ King(Father(John)) $\land$ Greedy(Father(John)) $\implies$ Evil(Father(John))
   While this creates ground sentences, we can see that it could run infinitely
2. EXISTENTIAL INSTANTIATION
   (∃ variable) (sentence) subst(variable/new-constant, sentence)
   ex: for constants{C}
   $\exists$ x Crown(x) $\land$ On-Head(x, John)
   $\rightarrow$ Crown(C) $\land$ On-Head(C, John)

Without any functions, this term count is finite, but with functions, it could be infinite.
We can see that in these cases:

1. $\forall$ level 1 = $\forall$ level 2
2. SAT($\exists$ level 1) $\iff$ SAT($\exists$ level 2)

This has a major result:
Theorem (Herbrand 1980)
if a sentence is entailed by a first order logic knowledge base,
then it is entailed by a finite subset of that propositional knowledge base.
Application:

```
for n = 0 to $\infty$ do
    create a propositional knowledge base by instantiating with depth-n terms
    see if the sentence is entailed by this knowledge base
```

We have a problem, however:
this only terminates if the sentence is entailed!
$\implies$ inference is considered **semi-decidable** (Church/Turing 1936)

We can use resolution to prove implications as with propositional logic.
We must first define the idea of **unification**.
A **unifier** is a set of variable assignments that make two statements equivalent.
Examples:

1. Knows(John, x) & Knows(John, John), {x/John} $\rightarrow$ Knows(John, John)
2. Knows(John, x) & Knows(y, OJ), {x/OJ, y/John} $\rightarrow$ Knows(John, OJ)
3. Knows(John, x) & Knows(y, Mother(y)), {x/Mother(John), y/John} $\rightarrow$ Knows(John, Mother(John)
4. Knows(John, x) & Knows(x, OJ), NO SOLUTION

We can improve unification by indexing facts/predicate indexing
$\equiv$ we cache highly used facts — very useful for many predicate, few clause logics.
If we store by predicate and first argument, then we can lookup by either.
This formed a subsumption lattice where children are formed by substitution on a higher.
In this lattice, the highest common descendent of two nodes comes from their MGU.
Since this is $O(2^n)$, it requires a small n; thankfully, most AI problems meet this.

First order resolution goes as in the following example:
(($\forall$ x Rich(x) $\implies$ Unhappy(x)), Rich(Ken))
{$\neg$ Rich(x) $\lor$ Unhappy(x), Rich(Ken)}
$\neg$Rich(x) $\lor$ $\frac{\text{Unhappy}(x),\text{Rich(Ken)}}{\text{Unhappy}(Ken)}$ with {x/Ken}

We use this to walk through the following proof:
Statements:

- Jack owns a dog.
- Every dog owner is an animal lover
- No animal lover kills an animal
- Either Jack or curiosity killed the cat

- Cats are animals

Query: Did curiosity kill the cat?
First Order Knowledge Base:

- Owns(Jack, Dog)
- ∀ x (*exists* y Owns(x, y) ∧ Dog(y)) $\implies$ ALover(x)
- ∀ x ALover(x) $\implies$ (∀ y Animal(y) $\implies$ ¬kills(x, y))
- Kills(Jack, Tuna) ∨ Kills(Curiosity, Tuna)
- Cat(Tuna)
- ∀ x Cat(x) $\implies$ Animal(x)

CNF Knowledge Base:

1. Dog(D)
2. Owns(Jack, D)
3. ¬Owns(x, y) ∨ ¬Dog(y) ∨ ALover(x)
4. ¬ALover(x) ∨ ¬Animal(y) ∨ ¬Kills(x, y)
5. Kills(Jack, Tuna) ∨ Kills(Curiosity, Tuna)
6. Cat(Tuna)
7. ¬Cat(x) ∨ Animal(x)
8. Query — ¬Kills(Curiosity, Tuna)
   ————————————————————————————-
9. ¬Owns(x, D) ∨ ALover(x) by <1, 5>: y/D
10. ALover(Jack) by <2, 9>: x/Jack
11. Animal(Tuna) by <7, 8>: x/Tuna
12. ¬ALover(x) ∨ ¬Kills(x, Tuna) by <4, 11>: x/Tuna
13. ¬Kills(Jack, Tuna) by <10, 12>: x/Jack
14. Kills(Jack, Tuna) by <8, 5>
15. CONTRADICTION by <12, 13>

To perform resolution, we need a CNF; we convert to a CNF much the same as with propositional: Consider: ∀ x [∀ y Animal(y) $\implies$ Loves(x, y)] $\implies$ [∃ y Loves(y, x)]
Step 1: eliminate $\implies$ and $\iff$
∀ x [¬∀ y ¬Animal(y) ∨ Loves(x, y)] ∨ [∃ y Loves(y, x)]
Step 2: move negation inward
∀ x [∃ y Animal(y) ∧ ¬Loves(x, y)] ∨ [∃ y Loves(y, x)]
Step 3: standardize variables
∀ x [∃ y Animal(y) ∧ ¬Loves(x, y)] ∨ [∃ z Loves(z, x)]
Step 4: "skolem"-ize
∀ x [Animal(F(x)) ∧ ¬Loves(x, F(x))] ∨ [Loves(G(x), x)]
Step 5: drop universal quantifiers
[Animal(F(x)) ∧ ¬Loves(x, F(x))] ∨ [Loves(G(x), x)]
Step 6: distribute
[Animal(F(x)) ∨ Loves(G(x), x)] ∨ [¬Loves(x, F(x)) ∨ Loves(G(x), x)]

We have a special case when the knowledge base contains only **definite clauses**.
a definite clause contains exactly one positive term (ex ¬A ∨ ¬B ∨ C)
we can thus convert these to an if-then rule (ex $A \wedge B \implies C$)

We will show two special algorithms for dealing with definite clause knowledge bases.
We will do this with the following example:

1. American(x) ∧ Weapon(x) ∧ Sells(x, y, z) ∧ Hostile(z) $\implies$ Criminal(x)
2. ∃ x Owns(Nono, x) ∧ Missile(x)
   Owns(Nono, M1) ∧ Missile(M1)
3. Missile(x) ∧ Owns(Nono, x) $\implies$ sells(West, x, Nono)
4. Missile(x) $\implies$ Weapon(x)
5. Enemy(x, America) $\implies$ Hostile(x)
6. American(West)
7. Enemy(Nono, America)

Notice that to be operated on, all clauses must be universally quantified

Forward Chaining

![]($(https://paper-attachments.dropbox.com/s_37BC3BDDFB4E93FABD6C94C74D020C3CD8B25F78236483B019E726793$

If there are no function symbols and all clauses are definite, the KB is a **data log**
    These are especially conducive to representing databases.
If we preprocess as rules come in, we can handle many facts and even model a brain in real time.

Backward Chaining

![]($(https://paper-attachments.dropbox.com/s_37BC3BDDFB4E93FABD6C94C74D020C3CD8B25F78236483B019E726793$

This is often used with improvements for logic programming (as in prolog)
The system often caches subgoals (like Missile(y)) for efficiency's sake

13 Probabilistic Reasoning

Let's motivate our study:

Early AI was almost entirely logic-based, but in the 70s, a crisis emerged.
Classical logic is **monotonic**, but human reasoning is not
($\equiv$ things that are true remain true with the introduction of new information)
$((\Delta \models \alpha) \implies (\Delta \vee \beta \models \alpha))$

Consider the following example:
Say you are told that Tweety is a bird (Δ)
Say you are asked "Does Tweety fly?" You would say yes (α)
But now say I tell you Tweety is a penguin (Δ)
Now say you are asked "Does Tweety fly?" You would say no ($\neg\alpha$)
If we were to build this with first-order logic:
Δ : $\forall$ x bird(x) $\implies$ flies (x)
good: $\Delta\wedge$ bird(Tweety) $\models$ flies(Tweety)
bad: $\Delta\wedge$ bird(Tweety) $\wedge\neg$flies(Tweety) (CONTRADICTION)
We may thus conclude our information Δ was bad and do
Δ : $\forall$ x bird(x) $\wedge$ abnormal(x) $\implies$ flies (x)
good: $\Delta\wedge$ bird(Tweety) $\models$ flies(Tweety)
bad: $\Delta \wedge \neg$lies(Tweety) $\models_?$ flies(Tweety)
We therefore must assume abnormal is false unless we hear otherwise.
This is not classical logic! It involves assumptions!
Another abnormal logic type:
Δ : Quaker(x) $\wedge\neg$ab(x) $\implies$ Pacifist(x)
Republican(x) $\wedge\neg$ab(x) $\implies$ $\neg$Pacifist(x)
But Nixon is a Quaker Republican, he can't both be and not be a pacifist!
Resolving this turns out to be complicated, and we will not address it; instead we introduce a new model:

Belief Revision

We introduce the idea of degrees of belief in [0, 1]
We will use the same example as with logic:

W	E	B	A	Pr(w)
1	1	1	1	0.0190
2	1	1	0	0.0010
3	1	0	1	0.0560
4	1	0	0	0.0240
5	0	1	1	0.1620
6	0	1	0	0.0180
7	0	0	1	0.0072
8	0	0	0	0.7128

Instead of ruling worlds in or out by entailment, we represent the chance of each world occurring.
f: sentences $\rightarrow$ degree of certainty; $Pr(\alpha) = \sum Pr(w)$ for $w \models \alpha$
Therefore

- Pr(E) = 0.1
- $Pf(\neg E) = 1 - Pr(E) = 0.9$
- Pr(B) = 0.12
- $Pr(B \vee \neg E) = 1 - Pr(\neg B \wedge E) = 0.92$

We have a few properties that are required for probability to work:

1. $0 \leq Pr(\alpha) \leq 1$
2. α is inconsistent $\iff Pr(\alpha) = 0$
3. α is valid $\iff Pr(\alpha) = 1$
4. $Pr(\alpha) + Pr(\neg\alpha) = 1$
5. $Pr(\alpha \vee B) = Pr(A) + Pr(B) - Pr(\alpha \wedge B)$
6. If α & β are mutually exclusive, $Pr(\alpha \vee \beta) = Pr(\alpha) + Pr(\beta)$

Now how can we change beliefs with the introduction of new information?
Clearly, we must zero out any worlds that are invalid
P($\alpha|\beta$) =

$$\left\{ \begin{array}{ll} 0 & \text{if } w \models \neg\beta \\ Pr(\alpha)/Pr(\beta) & \text{if } w \models \beta \end{array} \right\}$$

We can thus get **Bayes Condition**

$$Pr(\alpha|\beta) = \frac{Pr(\alpha \wedge \beta)}{Pr(\beta)}$$

We apply this to the earlier example:
Pr(B) = 0.2, Pr(B|A) = 0.741
Pr(E) = 0.1, Pr(E|A) = 0.307

Pr(B) = 0.2, Pr(B|E) = 0.2
Pr(E) = 0.1, Pr(E|B) = 0.1

We say two variables are **independent** iff P(E) = P(E|B) && P(B) = Pr(B|E)
Therefore we can see that B and E are independent in this system of belief.

How does this gel when introducing a third?
Pr(B|A,E) = 0.253 < Pr(B|A)
Pr(B|A,¬E) = 0.957 > Pr(B|A)
While inapplicable this graph, Pr finds A **conditionally independent** iff $P(\alpha|B \wedge C) = P(\alpha|C)$.
This is equivalent to saying "B gives C".

Thus using these properties I can define a model of the world.
We have a few equations that can calculate all we need to know:

1. **Chain Rule** $\equiv Pr(e_1 \wedge e_2 \wedge ... \wedge e_n) = Pr(e_1|e_2 \wedge e_3 \wedge ... \wedge e_n) + ... + Pr(en)$
2. **Case Analysis** $\equiv Pr(\alpha) = \sum Pr(\alpha \wedge \beta_i) = \sum Pr(\alpha|\beta_i)Pr(\alpha)$ where $\{\beta_i\}$
3. **Bayes' Rule** $\equiv Pr(\alpha|\beta) = \frac{Pr(\alpha)}{Pr(\beta)}Pr(\beta|\alpha)$

This last rule is just a rework of Bayes' Condition, so why did he get credit for another rule? The rule turns out to be incredibly useful; consider:
Say a patient gets a positive test for a disease (D)
Say we know P(D) = 1/1000
Say we know the false positive rate P(T|¬D) = 2%
Say we know the false negative rate P(D|¬T) =5%
We can use this to estimate the odds of the person having the disease!
P(D|T) = (P(D)/P(T)) P(T+D) = (P(T|D)P(D))/(P(T∧¬B)P(¬D) * (1-P(D|¬T))P(D)) = 4.5%

We have thus build a probability calculus on top of probability logic:
To move on, we need to consider the fact that a variable can have multiple values
We can't use our previous rules; this leads us to **equality** $\equiv (x = grn) \vee (x = blue) \implies P(y = large)$

14 Bayesian Networks

The tool we will use to work with probabilistic reasoning is called a **Bayesian Network**. This consists of two major components:

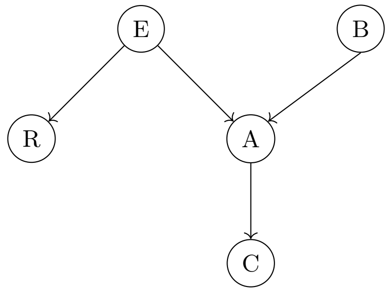
1. Directed Acyclic Graph shows casualty
2. Numbers/Nodes represent probability

Notation:

- Variable — X
- Value — x
- Variable Probability Distribution — Pr(x) = Pr(X=x)
- Set of Variables — X
- Instantiation — x
- World Probability Distribution — Pr(X) = truth table

Remember that if $Pr(\alpha|\beta, \gamma) = Pr(\alpha|\gamma)$, then α & β are independent given γ
We can apply the same idea to sets of variables:
X and Y are independent given Z if Pr(x|y, z) = Pr(x|z)
We write this as I(X, Z, Y)

Consider the following:



Logic

The alarm is triggered by burglary or earthquake.
A radio report is caused by an earthquake.
A call from the neighbor may come post-alarm.

Independence

R & C are independent given A — I(R, A, C)
E & B are independent — I(E, ϕ, B)

This can be much more efficient than a table.
The independence and parameters uniquely identify the graph.

We introduce a few terms:

- Parents(V) — the nodes pointing through V via an edge
  Parents(A) = {E, B} & Parents(R) = {E}
- Descendants(V) — the nodes reachable by V
  Descendants(E) = {R, A, C} & Descendants(B) = {A, C} & Descendants(X) = {}
- Non-Descendants(V) — the nodes unreachable from V, excluding the self & parents
  Non-descendants(A) = {R} & Non-descendants(E) = {B}

We call these independence statements the **Marcovion Assumptions**
≡ all the defined independence statements of a Bayesian Network
These are of the form I(V, Parents(V), Non-Descendants(V))
Marcov(G) = { I(C, A, BE), I(R, E, ABC), I(A, BE, R), I(B, ϕ, ER), I(E, ϕ, B) }

We can parametrize the structure by assigning weight to the connections.

![](https : //paper − attachments.dropbox.com/$s_1 3F2D47E7D3DCA10BE34C69E62E331D2224F9C5F91BEA8639D242F2792$

Note that $Pr(\neg d, b, \neg c) = \theta_{\{}\neg d|b, \neg c\}$

Consider an arbitrary row abcde

$Pr(abcde) = \theta_a \ \theta_{b|a} \ \theta_{c|a} \ \theta_{d|bc} \ \theta_{e|c} = 0.7$

$Pr(ab\neg cd\neg e) = \theta_a \ \theta b|a \ \theta_{\neg c|a} \ \theta_{d|b\neg c} \ \theta_{\neg e|c} = 0.216$

$Pr(\neg(abcde)) = \theta_{\neg a} \ \theta_{\neg b|\neg a} \ \theta_{\neg c|\neg a} \ \theta_{\neg d|\neg b \neg c} \ \theta_{\neg e|\neg c} = 0.09$

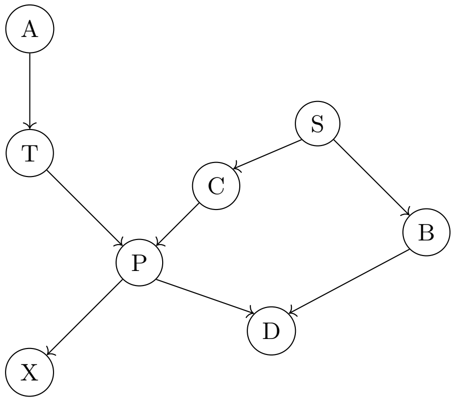
We can evaluate the independence of any two nodes are independent given a third by

Deseperation

This uses an expansion on Markov's much more efficient than probability theory.
If setting Z blocks paths x→y, x & y are d-separated given Z
There are three ways for a node to be connected in a path, treated as follows:

- sequential $\equiv \rightarrow w \rightarrow$ is blocked iff w $\in$ **Z**
- divergent $\equiv \leftarrow w \rightarrow$ is blocked iff w $\in$ **Z**
- convergent $\equiv \rightarrow w \leftarrow$ is blocked iff w & Decendants(w) $\notin$ **Z**

We can clarify by giving an example:



dsep(C, S, B)?
→ open(BSC)? false.
→ open(CPDV)?
−→ open(SPD)? true.
−→ open(PDS)? false.
$\implies$ dsep(C, S, B) = true.

15 Modeling and Inference

![$CausalGraphandConditionalProbabilityTables$]($https://paper-attachments.dropbox.com/s_5F1FEB98332EEC99C328935$

Variables:
C — condition
T's — tests to detect the condition
S — sex of the patient

We first compute the marginal distributions to discuss probabilities.
These come in two major forms:

![]($https://paper-attachments.dropbox.com/s_5F1FEB98332EEC99C328935F7AB0E929534BB33EF2AB134DF14A85072A$

We then use this information to find the **Most Probable Explanation**:
We assume the final consequence (A) and search for the most probable query
Setting A moves us from 32 → 8 states;
result: {C=no, S=fe, T1=-ve, T2=-ve} = 47%

This is not applicable in every situation; we generalize into the **Maximum a Posteriori Hypothesis**:
this is more complex and less efficient, but more often applicable
we find a subset of variables & find the MPE
(ex. S=C | A → {C=no, S=male} $|approx$ 49.3%

When we seek to make inferences, algorithms fall into two major categories:

1. variable elimination
2. conditioning

In both situations, complexity is tied to the topology of the Bayesian Network
the actual property is **tree width** — it is roughly analogous to connectivity
MPH = $O(nd^w)$ given n=#var, d=#val, w=width
We will not define tree width, as it is complex, but we observe a few special cases:
Trees have width 1 (one path from any node to node)
Poly-trees (>1 parent ok) have width = the maximum parent count of any node
These are both singly connected networks; multiply connected leads to a DAG

![]($https://paper-attachments.dropbox.com/s_5F1FEB98332EEC99C328935F7AB0E929534BB33EF2AB134DF14A85072A$

Our first method of performing inference is

Weighted Model Counting

Consider the statement $\Delta = (A \vee B) \wedge \neg C$
This has 3 variables and suggests 8 worlds
Given an nd-DNNF circuit, we can solve in O(N)!
Consider the example on the right; it is trivial! WMC = 0.04 + 0.10 + 0.00 = 0.14
We therefore only need a compiler which would transform Δ →sd-DNNF
Unfortunately, in the general case this is intractable
There are tractable subsets of this, though:

![]($https://paper-attachments.dropbox.com/s_5F1FEB98332EEC99C328935F7AB0E929534BB33EF2AB134DF14A85072A$

We can use probability is our weights! → $WMC(\Delta \wedge \alpha) = Pr(\alpha)$
So we need to convert Δ →boolean circuit.
We can see that Δ holds in $w_1, w_4, \& w_7$
We thus say $W(A) = W(\neg A) = ... = W(\neg C) = 1$
We then let $W(P_i) = \theta_i \& W(\neg P_i) = 0,$ so $W(A) = W(P_1)W(P_4)W(P_7)$

Thus we can solve probabilistic reasoning by symbol manipulations!
Properties of the algorithm:

1. there are many possible representations
2. it is not sensitive to tree width

3. it is not only applicable to Bayesian Networks

We can see that the size(Bayes) = size(CPT); though size(Bayes) = O(nd^{k+1}), size(joint-table) = O(D^n)!
This shows that Bayesian Networks are much more space efficient!

The process of modeling logic as a Bayesian Network has 3 steps:

1. define variables & values
2. define edges
3. specify CPT

Variables will then be labeled either query or evidence variables depending on the query.
This is a common approach used in early spam filters and Google ad Rephil.

![]$(https://paper-attachments.dropbox.com/s_5F1FEB98332EEC99C328935F7AB0E929534BB33EF2AB134DF14A85072A$

Consider the following statements

- The cold causes a sore throat/chill
- The flu causes a sore throat/chill/fever/body ache
- Tonsillitis can show itself in fever/body ache

This network forms a bipartite graph.
We could have used one multivariable disease node.
We use this to give probabilities of a given condition.

If we have complete information, we can model a system as a Bayesian Network.
For incomplete information, however, we must use Expectation Maximization.
For example, suppose the following environment:

![]$(https://paper-attachments.dropbox.com/s_5F1FEB98332EEC99C328935F7AB0E929534BB33EF2AB134DF14A85072A$

What is the marginal $(P|\neg S, \neg B, \neg U)$? – 10.21%!
WHAT?? why is that so high?
It turns out this is because of the false negative rate of the scanning test.
We want a false negative rate of below 5%.
We can address this in one of three ways:

1. get a better scanning test — a false (-) for S of 4.63% meets our requirement
2. lower the success of the procedure — 75.59% would meet our requirement
3. increase P(L|P) — 99.67% meets our requirement

(b) and (c) turn out to be either impractical or in-economical, so our standard approach is to pay!

16 Bayesian Learning

We wish to learn both the parameters and structure of a Bayesian Network.

Parameters

Consider the disease model from the previous lecture:

![]($https : //paper - attachments.dropbox.com/s_E 29353D8DE6A7F32069419A77A11E9F7A8BD49E718AC081FD5F77701FB$

Cases 1 and 3 have incomplete data, whereas 2 had **complete**.
If every example is complete, the data set is called complete.
If complete, the maximum likelihood parameters are unique.
We find the likelihood of a parameter set like so:

![]($https : //paper - attachments.dropbox.com/s_E 29353D8DE6A7F32069419A77A11E9F7A8BD49E718AC081FD5F77701FB$

Param $\{S_1\} \to BN_1 \to Pr_1(.)$
Param $\{S_2\} \to BN_2 \to Pr_2(.)$
.
.
.
We pick the set of parameters that maximizes the probability,
so score $S_i = \prod Pr_i(e_i)$ is the likelihood of parameters

Parameter Estimation With Complete Data

![]($https : //paper - attachments.dropbox.com/s_E 29353D8DE6A7F32069419A77A11E9F7A8BD49E718AC081FD5F77701FB$

The large table is our empirical distribution .
This is formed by collapsing our distribution.
$\theta_{\{}\neg S|H\} = \frac{Pr(\neg s,h)}{Pr(h)} = \frac{5}{6}$.
This is our maximum likelihood parameter estimate!

Parameter Estimation With Incomplete Data

This uses an iterative algorithm called **expected maximum**.
Say we wish to fill the row <h, s, ?>.
We start with an arbitrary guess at the CPT.
CPT$_1$ $\to$ BN$_1$ $\to Pr_1(.)$.
We must choose from {TFT, TFF}, so we assign probabilities according to the probability function
x = Pr1(e|h, ¬s)
y = Pr(¬e|h, ¬s)
We then take the assignment with the greater probability, yielding
CPT$_2$ $\to$ BN$_2$ $\to Pr_2(.)$.

We only needed to surmise one term here, but we may need to iterate.
The probabilities Pr1(.) & Pr2(.) are guaranteed to be non-increasing, so this converges.
This is thus a form of local search algorithm, so we know we may have to run repeatedly.
Note: this is the thinking behind the original algorithm, but it isn't how it is performed in practice.

Structure

Consider the following three structures:

![]($https : //paper - attachments.dropbox.com/s_E 29353D8DE6A7F32069419A77A11E9F7A8BD49E718AC081FD5F77701FB$

What do I need to optimize to choose between these three structures?
We have learned many of these algorithms, so we won't discuss in detail, but:

1. local search methods (approximate methods)
   $\to$ we transform the structure looking for a better score
   We must thus specify movement within the neighborhood structure: $\to$ legal operations: add/remove/reverse edge

2. systemic search methods (exact methods)
   → A* is a good example

Why can't we just use maximum likelihood? We face **overfitting**.
Say we are using likelihood to compare; then C > B > A.
We will thus end up with a complete DAG, no matter what.
Thus we need a model that balances structure complexity with likelihood.
Why? Consider:

$![](https://paper-attachments.dropbox.com/s_E29353D8DE6A7F32069419A77A11E9F7A8BD49E718AC081FD5F77701FB$

The data is clearly a linear fit, but if we estimate to the fourth degree, we get bad data!
There is no fixed answer to the problem; we clearly need some function of the form f(likelihood) - g(complexity).
A common one is called MDK, but we need not discuss details.

Model-Oriented Vs. Query-Oriented Learning

This can also be referred to as (unsupervised vs supervised) or (unlabeled vs labeled).
We have done model learning now; we move on to query based.
A model-based approach might give us the following:

$![](https://paper-attachments.dropbox.com/s_E29353D8DE6A7F32069419A77A11E9F7A8BD49E718AC081FD5F77701FB$

Say we promise only to infer upward:
Then we don't 'model' the system and instead prepare for a specific 'query'.
Labeling of correct responses is done by humans, and this is done to 'train'.
If we want to talk about supervised learning, we need to introduce

Arithmetic Circuits

$![](https://paper-attachments.dropbox.com/s_E29353D8DE6A7F32069419A77A11E9F7A8BD49E718AC081FD5F77701FB$

The (+) symbols represent OR gates
The (*) symbols represent AND gates
The θ are the parameters
The λ are the evidence

When given a query, we can perform weighted model counting on the equivalent arithmetic circuit.
If A=True, $\lambda_A = 1$; if A=False, $\lambda_{\neg A} = 1$; if unknown, both are 1
Thus we can evaluate this in linear time;u nfortunately converting to an arithmetic circuit is $O(nd^w)$

BUT what do we do if we don't have the parameters? We use the labeled data:

$![](https://paper-attachments.dropbox.com/s_E29353D8DE6A7F32069419A77A11E9F7A8BD49E718AC081FD5F77701FB$

Cross Entropy gives us a measure of the disagreement between the two.
Therefore, we often seek to minimize it across a structure.
We generally use it to perform gradient descent (nth dimensional hill climbing).
Consider the example of recognizing shapes:

$![](https://paper-attachments.dropbox.com/s_E29353D8DE6A7F32069419A77A11E9F7A8BD49E718AC081FD5F77701FB$

Though pixel is functional in general, we allow it to be probabilistic to permit noise.
It will thus be inferred to be very close to 0/1.
A lot of heights will be zeroed as well depending on the column.
We can thus see that we have a lot of **background knowledge**   The simplest case of this is $A \iff B$, for which we could simply substitute and reduce cases.

17 Decision Trees and Random Forests

These two models are often used in **classifiers**.
 ≡ a machine learning system that makes decisions on input.
  the inputs are called characteristics or instance.
  the output is called a decision.
We can construct them from Bayesian Networks.

We must first introduce the idea of **entropy**.

$$\text{ENT}(X) = -\sum_X Pr(x)log_2(Pr(x))$$

We can show it with the following data table:

$![](https://paper - attachments.dropbox.com/s_6 6C470E465947B218F12E6833ACF54B222DBA6F153DEEC1F4A1A4D0690$

The form we tend to utilize is **CONDITIONAL PROBABILITY**.
If we have ENT(X) and we learn that Y=y, we have

$$E(X|y) = -\sum_X Pr(x|y)log_2(Pr(x|y))$$

Alternatively, if we plan to observe Y but do not yet know the value

$$E(X|Y) = -\sum_Y Pr(y)\text{ENT}(X|y)$$

It also turns out that information can never increase average entropy, ie

$$\text{ENT}(X|Y) \leq \text{ENT}(X)$$

Note that this specifies average; the entropy of a single value may increase:

$![](https://paper - attachments.dropbox.com/s_6 6C470E465947B218F12E6833ACF54B222DBA6F153DEEC1F4A1A4D0690$

These are used to build classifiers by supervised learning of labeled data.
Our CPT thus effectively functions as our model.

We will now use the notion of a decision tree/random forest to solve a problem.
We will use the following data and corresponding tree:

$![We have 12 labeled variables](https://paper - attachments.dropbox.com/s_6 6C470E465947B218F12E6833ACF54B222DBA6F$

This model is called **interpretable** because it is easy to read, as opposed to a neural network.
Classifying a variable is as easy as parsing the tree!
 Consider X_{12} — we can just walk; this probability happens to match, but it won't be in general.

$![](https://paper - attachments.dropbox.com/s_6 6C470E465947B218F12E6833ACF54B222DBA6F153DEEC1F4A1A4D0690$

The depth of the decision tree is a sign of its complexity.
Splitting is as easy as making a choice.
Nodes represent attributes.
Leaves represent decisions.

We can equivalently build:
 this has 4 attributes rather than the 10 from above
 this is much shallower, and thus simpler

The algorithm itself is very simple; we just split repeatedly as if tracing the tree.

This assumes a black box for choosing variables, but developing one is not hard
How do we choose which attribute to split on at a given depth?
We can define the algorithm by looking at the following state:

$![](https://paper - attachments.dropbox.com/s_6 6C470E465947B218F12E6833ACF54B222DBA6F153DEEC1F4A1A4D0690$

We thus use conditional entropy as a score to determine our next split.

The algorithm is as follows:

![]($https://paper-attachments.dropbox.com/s_66C470E465947B218F12E6833ACF54B222DBA6F153DEEC1F4A1A4D0690$

How do we evaluate an algorithm? We use **cross-validation**.
$\equiv$ split the dataset into 80/20 training/testing data & repeat to find average score.

This can be generalized one more time to a **random forest**.
We build a series of trees and majority vote to determine the output.
We call this type of method an ensemble learning method.

Suppose we have a dataset of 5 values; we may bootstrap data sets by random choice to get:

![]($https://paper-attachments.dropbox.com/s_66C470E465947B218F12E6833ACF54B222DBA6F153DEEC1F4A1A4D0690$

The count of numbers chosen will be a parameter.
We can test the power using the out of bag examples.

![]($https://paper-attachments.dropbox.com/s_66C470E465947B218F12E6833ACF54B222DBA6F153DEEC1F4A1A4D0690$

A specific subset of these are called naive:

![]($https://paper-attachments.dropbox.com/s_66C470E465947B218F12E6833ACF54B222DBA6F153DEEC1F4A1A4D0690$

Traditionally, we want AI to be easily explainable.
Consider the following example:

![]($https://paper-attachments.dropbox.com/s_66C470E465947B218F12E6833ACF54B222DBA6F153DEEC1F4A1A4D0690$

Say we are asked C|{S=1, G=0, F=1, M=1}.
We might say "yes, because F & M!", as S & G are not used.
This is called a **PI-explanation**.
In actuality, we can make a tractable circuit from this data, which is much power powerful.
This, however, is not as interpretable!
In the current day, Random Forests < Bayesian Classifiers < Neural Networks.

18 Neural Networks

These have been around for approximately 50 years, but have had many ups and downs.
They have had a large impact in image analysis and sequence to sequence translation.
Neural networks are a model-free approach to problem solving based on function fitting.
Traditional approaches are model-based and work via representation and reasoning.

The basic unit of a neural network is the **NEURON**.
These map a set of input activations to an output activation.

![](https://paper-attachments.dropbox.com/s_6E2D9A912A4932DB33218BCF0120FD3DBC4521D2822A1AA28C3A628B53

a — activations
w — weights
b — bias
g(.) — activation function
ai = g($[\sum w_{ij} a_j] + b)$

Activation Functions come in the following forms (generally):

![](https://paper-attachments.dropbox.com/s_6E2D9A912A4932DB33218BCF0120FD3DBC4521D2822A1AA28C3A628B53

A connected set of these forms a neural network!
Modern neural networks also use non-neuron elements, but they are application specific.

We could learn the structure, but many situations have heuristically derived templates.
The specific forms of neural networks we consider are **Feed-Forward Neural Networks**.

![](https://paper-attachments.dropbox.com/s_6E2D9A912A4932DB33218BCF0120FD3DBC4521D2822A1AA28C3A628B53

Greater depth requires more training, sp multi layer NN's are computation-heavy.
They used to be nearly impossible to train, but deep-learning has changed this.
Neural networks are universal function approximators, since they map in → out.
Are they expressive enough?
YES; by the above, they can express every function up to a constant error.

Pure-Step Neural Networks

Basic gates can be formed by networks consisting only of neurons with step functions.

We can form an AND gate from a single neuron:
![](https://paper-attachments.dropbox.com/s_6E2D9A912A4932DB33218BCF0120FD3DBC4521D2822A1AA28C3A628B53

We can also form an OR gate from a single neuron:
![](https://paper-attachments.dropbox.com/s_6E2D9A912A4932DB33218BCF0120FD3DBC4521D2822A1AA28C3A628B53

We cannot, however, form an XOR gate from a single neuron.
This is because an XOR gate is not **linearly separable**.
![](https://paper-attachments.dropbox.com/s_6E2D9A912A4932DB33218BCF0120FD3DBC4521D2822A1AA28C3A628B53

Our neuron functions can only represent linearly separable functions.
Networks, however, can represent much more complex representations.

![](https://paper-attachments.dropbox.com/s_6E2D9A912A4932DB33218BCF0120FD3DBC4521D2822A1AA28C3A628B53

This neural network, as can be seen, represents a complex non-linear function.
We train the neural network by setting the weights to match labeled data.
If I fix the values of a_1 and a_2, $a_5 = f_1(w_{13}, ..., w_{45}) = f_2(w_{13}, ..., w_{45})$.
Thus this is just a complex case of function fitting!
Modern neural networks can have billions of parameters, so this can grow very complex.

Training neural networks is an exercise in optimization.

Our optimization criteria is called the **loss function**.
This has a couple common forms:

- cross entropy
- mean-squared error $= \frac{1}{N} \sum (\text{NN}(L_i) - L_i)^2$.
  This is of the form f(w1, . . . , wk), so we can optimize by weight!

We do the actual optimization via gradient descent.
Tensor flow, RUST, and the ADAM optimizer can do this for us.
We can stochastically break the data to increase the accuracy of the fit.

![]($https : //paper - attachments.dropbox.com/s_6E2D9A912A4932DB33218BCF0120FD3DBC4521D2822A1AA28C3A628B53$

We call a single iteration of the gradient-descent algorithm an **epoch**.
How do we know which epoch to stop on?

1. set a limit on epoch count
2. stop when epoch stops changing much (patience)

We could also divide the data into batches to run optimizations in parallel.
We thus perform an iteration on all batches per epoch.
This has two major advantages:

1. parallelism
2. randomization (stochastic gradient descent)

A batching approach can run extremely fast on GPU's rather than CPU's!

19 Neural Network Applications

There are two fundamental sub-networks within Convolutional Neural Networks (CNN).

- convolution — local detection of features/patterns
- aggregation/abstraction

A CNN alternates between the two structures.

Diagramming neurons can get complex and does not scale well; we use matrices instead.

We define the parameters of a subnetwork of a neural network as follows:

1. f — the filter size
2. n_c — the number of input channels (matrices)
3. s — the step size; how far to move the filter between iterations
4. p — the padding; extra area around the input edge added to emphasize edge features

Convolution

Consider the example of analyzing a greyscale image; it may look like:

$![convolutionw/f = 3, nc = 36, s = 1, p = 0](https://paper-attachments.dropbox.com/s_A A78EEC73035B10A6424111F8CC1$

We pass the filter over each fxf submatrix of the input, performing a column dot product.
We then apply an ReLU function before entering into the output matrix.
The values in the filter are chosen via gradient descent.
We can notice that each of the hyperparameters affect the size of the output.
We choose these parameters heuristically, as many good models already exist.
The output is 4x4x1 in this example, but we could do the following transformations.

- f++ $\rightarrow$ 3x3x1
- n_c++ $\rightarrow$ 4x4x2
- s++ $\rightarrow$ 2x2x1 (in this case some of the input data is ignored)
- p++ $\rightarrow$ 6x6x1

We can model a convolution with increased parameters like so:

$![complexconvolutionw/f = 3, nc = 3, s = 1, p = 0](https://paper-attachments.dropbox.com/s_A A78EEC73035B10A6424111$

This behavior may bee seen in a color (rgb) image seeking to detect two features

Max-Pooling

An example may take the following form:

$![max-poolw/f = 2, nc = 1, s = 2, p = 0](https://paper-attachments.dropbox.com/s_A A78EEC73035B10A6424111F8CC18$

The output is determined by the maximum of the values in the region it represents.
This effectively functions to aggregate data values to the most likely representation.
Padding is almost always zero, and there are no weights.

These two operations are applied alternately to form a CNN:

$![](https://paper-attachments.dropbox.com/s_A A78EEC73035B10A6424111F8CC18FF4AA14B9B2A8F73FF013C6765837$

As we move right, we tend to increase filter count and thus n_c, but decrease f.
We can thus think of this operation as stretching and squishing volume in alternation.
The number of weights tends to be controlled and independent of the data.
We evaluate these with **tensors** $\equiv$ parallelize-able multidimensional arrays.

The Future of Neural Networks

Neural networks are intriguing because they allow us to do new things in a pretty simple way.
BUT people encounter issues:

1. data hunger
   this is self explanatory — neural networks require lots of labeled data to train
2. brittleness/lack of robustness
   small, seemingly arbitrary changes to data can cause large output changes
   this leads to adversarial attacks which seek to exploit brittleness
3. difficulty of explanation
   neural networks are unable to effectively explain why a certain output is correct
   this stems from the fact that NN are model-free and focus on function fitting

Here is an example of an integrated model developed in Dr. Darwiche's lab: The data below compares the correctness of three approaches to image recognition

1. Bayesian Network
2. Bayesian Network + Background Knowledge
3. Neural Network

![]$(https://paper-attachments.dropbox.com/s_AA78EEC73035B10A6424111F8CC18FF4AA14B9B2A8F73FF013C6765837$

We observe that approach (b) beats the neural network handedly in low data situations.
Even in high data situations, approach (b) tends to beat out the neural network.

The below data compares the approaches when trained with different data than tested with.
Noise is either horizontal or vertical pairs, and the models see only one type in each.

![]$(https://paper-attachments.dropbox.com/s_AA78EEC73035B10A6424111F8CC18FF4AA14B9B2A8F73FF013C6765837$

In this situation, approach (b) is relatively unaffected, but the neural network falls apart! This is analogous to a student solving problems by applying formulae.
If a query represents something not well represented in the training data, an NN fails.
Though neural networks do very well with prototypical situations, this is impractical.
These type of brittleness errors are common in applications like language translators.
This is okay in those situations, but for higher risk thinks like self-driving cars, it is not.
Thus we can see that AI demands the integration of model-based and -free approaches.