Intro to AI

An agent (or AI program) gets to interact with an environment. It can perceive the environment through sensors and it can affect its state through its actuators.

The big question is the function that maps sensors to actuators, which is called the control policy for the agent

This class is about how does an agent make decisions that it can carry out with its actuators, based on past sensor data

Those decisions take place many, many times, and the loop environment Feedback → sensors → agent decision → actuator interaction with environment and so on is called perception action cycle

Terminology

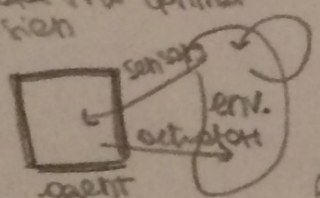
Fully vs partially observable

↑

At any point in time the environment is completely sufficient to make the optimal decision

↑

You need memory on the side of the agent to make the best decision



state ← For many environments it's convenient to assume that the environment has some sort of internal state

An environment is Fully observable if the sensors can always see the entire state of the environment and partially observable if the sensors can only see a fraction of the state but remembering past measurements gives us additional information of the state that is not readily observable right now

Deterministic vs stochastic

↑

Agent's action uniquely determine the outcome

↑

The outcome of an action can't be predicted

Discrete vs continuous

↑

Finitely many action choices and Finitely many things you can sense

↑

The space of possible actions or things that you could sense may be infinite

Benign vs adversarial

↑

The environment might be random (stochastic) but it has no objective on its own that would contradict the own objective

↑

Your opponent is really out there to get you. It's much harder to find good actions in that case

opponent deceives you and counteracts what you're trying to achieve ✓

AI and uncertainty

"what to do when you don't know what to do?"

Reasons for uncertainty may be
- sensor limit
- adversaries
- stochastic environment
- laziness
- ignorance

AI deals with uncertainty and manages it in decision making

Problem solving

theory and techniques of building agents that can plan ahead to solve problems.

In particular problems where the complexity comes from the idea that there are many states.

eg navigation problem where there are many choices to start with, and the complexity comes from picking the right choice now, and the right choice at the next intersection and so on

In contrast to problems where the complexity comes from partial observability. We can't see the result of our action and the actions themselves are not known

Definition of a problem
- Initial state
- actions(state) → {actions} (this is problem specific)
- result(state, action) → new state
- goaltest(state) → true or false
- path cost(path) → cost

sequence of state/action transitions

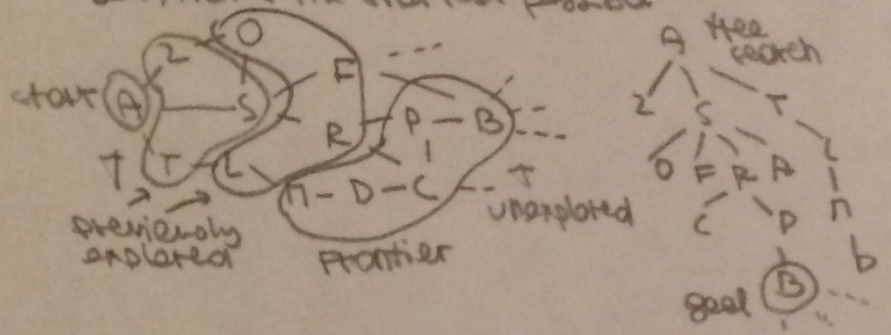
The farthest paths that have been explored we call the frontier

...

Function tree_search(problem):
  Frontier: {[initial]}
  loop:
    if Frontier is empty: return FAIL
    * path = remove_choice(Frontier)
      s = path.end
      if s is a goal: return path
      for a in actions(s): ← actions from s, expanding path
        add [path + a → Result(s, a)] to Frontier

Tree search is a family of algorithms with only difference in the choice * is which we do look at first

BFS (or shortest first search)
Always chooses from the Frontier, one of the paths that hasn't been considered yet, that's the shortest possible



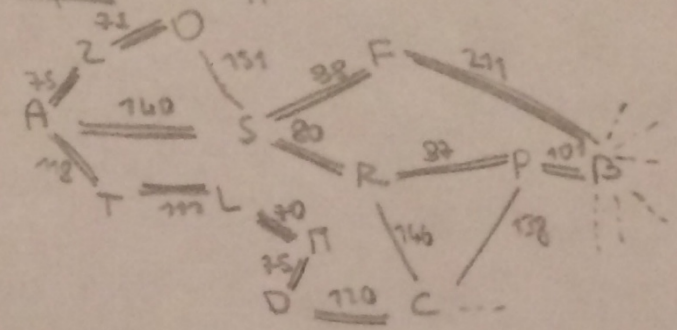
We keep track of explored states so that if we find a duplicate when expanding the Frontier we don't add it to our shortest path

Function graph_search(problem) ← tree_search w explored!
  Frontier: {[initial]}
  explored: {}
  loop:
    if Frontier is empty: return FAIL
    path = remove_choice(Frontier)
    s = path.end
    add s to explored ← termintes when goal is removed from the frontier, we can make it stop as soon as a goal is added TO the Frontier through
    if s is a goal: return path
    for a in actions:
      add [path + a → Result(s, a)] to Frontier
      unless Result(s, a) in Frontier + explored

However BFS finds the path with the lowest number of steps. If our actions have a cost, we will want to use a different strategy

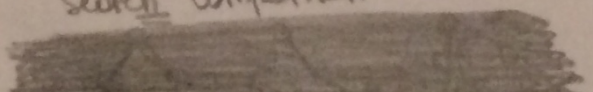
Uniform cost search (cheapest-first)
We pick the path with the lowest total cost and expand it each time by removing it from the Frontier and adding it unexplored neighbours

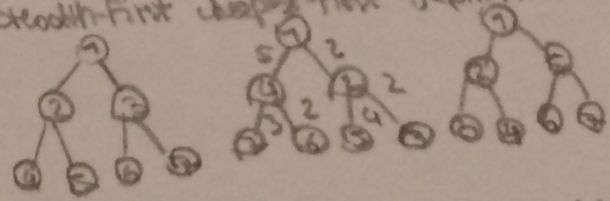
We only return when we pop a path with a goal state, off the Frontier. Not when the goal is first added, like in BFS



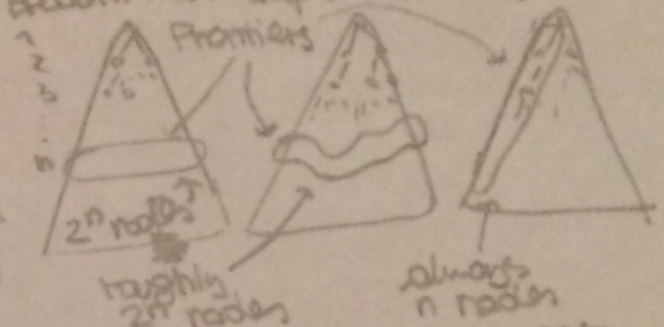
If a path is expanded and reaches an already explored node but with a higher cost, we'll drop it

Search comparison

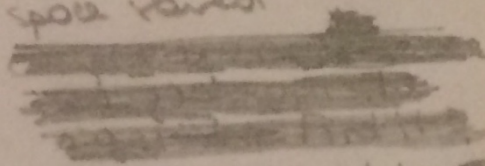




depth-first is the only one which is not optimal. Meaning it doesn't necessarily find the shortest path

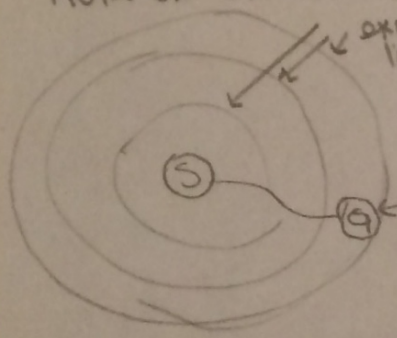


If we also keep track of the explored set, then we don't get that much savings, but without it depth-first has a huge advantage in terms of space saved



If the tree is infinite and the goal is at a finite level, depth-first and cheapest-first will find it, so they are complete. However depth-first will keep following an infinite path, if it finds one

More on uniform cost

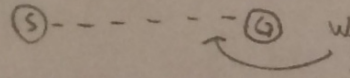


expands like this

the search wasn't directed in any way towards the goal, just expanding everywhere out in the space

with small spaces that's fine, but with larger spaces we won't get to the goal fast enough

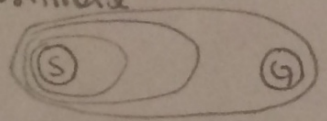
We want to add [redacted] more knowledge to find the goal faster. In search, proven to be most useful is an estimate of the distance S→G



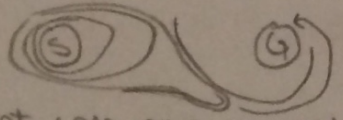
We'll take an estimate the straight line distance S→G

Greedy best-first search

It expands first the path that's closest to the goal according to the estimate



Gets us immediately towards the goal, but that won't be always the case if the are obstacles along the way



← [redacted] Not the best path

In that case, once it reaches the barrier, it will try to increase along a path that's getting closer and closer to the goal

We'd like [redacted] TO explore a small number OF nodes (greedy search) but still guarantee to find a shortest path (uniform cost search)

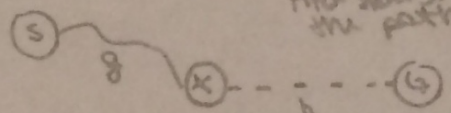
[redacted]

A* search

always expand the path with minimum F

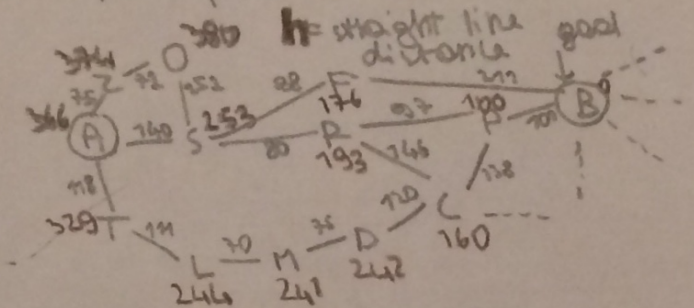
$F = g + h$ ← h(path) is the estimated distance to the goal (same as h(s) where s is the final state of the path)

↙ g(path) is the path cost

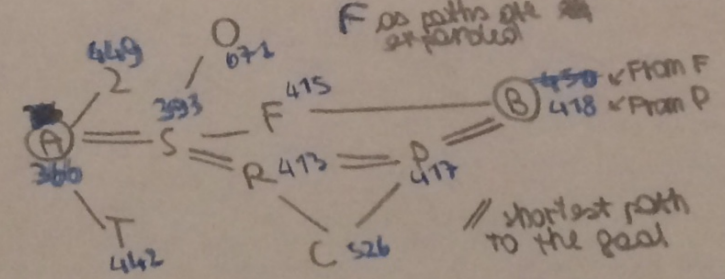


minimizing g helps us keep the path short and minimizing h helps us keep focused on finding the goal

Finds the shortest path while expanding the minimum number of paths possible



F as paths are expanded



// shortest path to the goal

[redacted] A* Finds the shortest path if h(s) ≤ true cost [redacted] is h never overestimate

When A* ends, it returns a path p with estimated cost c which equals the actual cost.

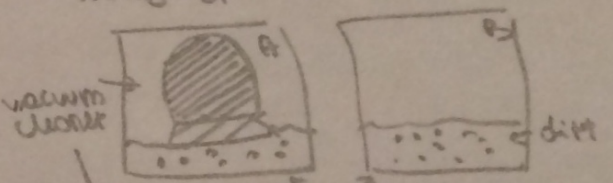
All paths on the Frontier have an estimated cost that's greater than c as the Frontier is explored in cheapest-first order

IF h is optimistic, then the estimated cost is less than the true cost. So p has an estimated cost that's less than the true cost, of any of path in the frontier, and paths that go beyond the frontier, must have a cost that's greater than that, as the step cost is always 0 or more.

So p must be the minimal cost path

Other state spaces

problem solving through search can [redacted] deal with [redacted] many types of state spaces



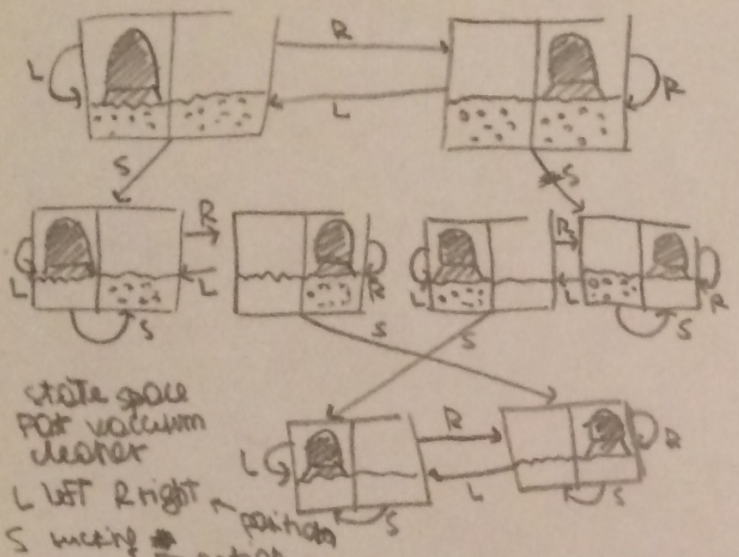
It can be in either position, but each position can have additional properties. In that case each position can either have dirt in it or not have dirt in it

$2 \times 2 \times 2$ ← possible states

↑ two states (A and B)

↑ state A can be dirty or not

↑ state B can be dirty or not



state space for vacuum cleaner

L left R right S suck = action

say the cleaner also had
- power on/off/sleep
- camera on/off
- brush height 1/2/3/4/5
- positions 10

We'd have $3 \times 2 \times 5 \times 2^{10} \times 10$
(power, camera, brush height, dirt (for each of the 10 positions), robot's position)

A fairly trivial problem, like this one, works out to a large number of state space. That's why we need efficient search algorithms

Sliding blocks puzzle

starting state, all the positions are messed up

goal state, numbers are in order (left to right, top to bottom)

heuristics for this:
- h1 = # misplaced blocks
- h2 = sum (distance of blocks) distance = # blocks to move to get to the right position

h1 is admissible because every tile that's in the wrong position must be moved at least once to get into the right position, so h1 never overestimates

h2 is also admissible because any tile in the wrong position can be moved closer to the correct position no faster than one space per move. But h2 is always ≥ h1, so with the exception of breaking ties, A* search using h2 will always expand fewer paths than one using h1

We're trying to build an AI that can solve problem like this all on its own. We might complain that in order for the search algorithm to work we had to provide it with a heuristic function. We might think that coming up with a good heuristic function is really where all the intelligence is, so a problem solver using it really isn't intelligent at all

description
h1 { a problem can have A → B } h2
if (B is adjacent to B) and (B is blank) ← not considered

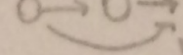
Both our heuristic can be derived from a simple description of the problem

h = max (h1, h2) ← another way to come up with good heuristic

The only problem with combining multiple heuristics is that there is some cost to compute the heuristic

This is called generating a relaxed problem. We've taken the original problem, where it's hard to move squares around, and made it easier by relaxing one of the constraints

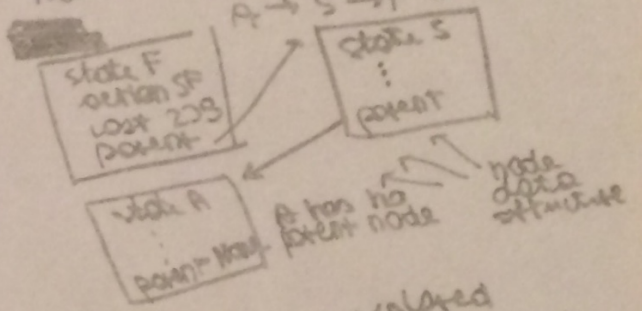
In the state space, we're adding new operators that traverse the state in new ways and makes the problem easier (thus never overestimates)



Problems with search
All solutions we've found consist of a fixed sequence of actions.

That works fine when the domain is
- fully observable
- known
- discrete
- deterministic
- static

Note on implementation



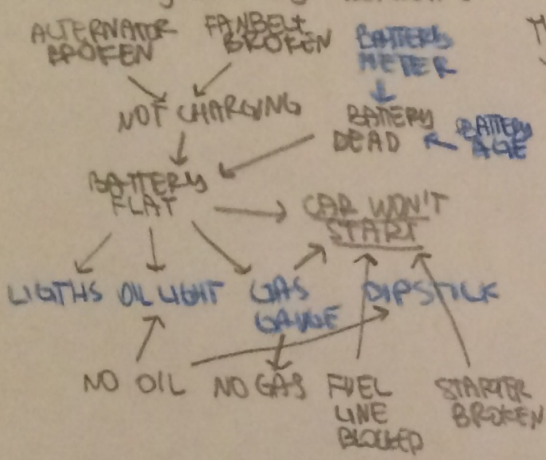
Frontier
keeps track items and adding in new ones
priority queue!

membership test
set!

explored
add new members and check for membership
set!
other data structures needed

Most efficient search implementation use both priority queues and sets!

Probability and Bayes networks



There are many different ways to explain that the car won't start

Diagnose the problem which may increase or decrease your belief that the battery, for example, may cause the failure

Other inspections. All of these can influence your measurements

A Bayes network exists on observable variables, [redacted] to hidden causes.
It is composed of nodes, that correspond to events that we might or might not know, typically called random variables.
These nodes are linked by arcs that suggest a child is influenced by its parent in a probabilistic way (an older battery has a higher chance of causing the battery to be dead)

[redacted]

We can observe things like the oil lights and the lights and the battery meter and compute probabilities of hypothesis like the alternator is broken, or the fan belt is broken, or the battery is dead

Probabilities

Used to express uncertainty, and the management of uncertainty is key to many things in AI

Coin Flips
$P(H) = 0.5 \quad P(T) = 0.5$
$P(H) = 0.25 \quad P(T) = 0.75$

$P(H,H,H) = 0.125 \quad P(H)$
$P(H) \cdot P(H) \cdot P(H)$ since they are independent events

The coin has to come up either heads or tails, therefore the total probability of both coming up is 1

x_i = result of ith coin flip $\quad x_i = \{H, T\}$
$P(x_1 = x_2 = x_3 = x_4) = 0.125 \quad P_i(H) = 0.5 \ \forall i$
$= P(H,H,H,H) + P(T,T,T,T)$

$P(\{X_1, X_2, X_3, X_4\} \text{ contains } \geq 3 \ H) = 0.3125$

H H H H	0.625	$0.625 \cdot 5$
H H H T	⋮	
H H T H		
H T H H		
T H H H		

summary
$P(A) = p \Rightarrow P(\neg A) = 1 - p$
$X \perp Y: \quad P(X) \cdot P(Y) = P(X, Y)$

Dependence
$P(X_2 = H) = 0.5 \rightarrow$ H: probability of $X_2 = H$, given that $X_1 = H$: $P(X_2 = H \mid X_1 = H) = 0.9$
pick coin based on X_1's outcome $\rightarrow$ T: $P(X_2 = T \mid X_2 = T) = 0.8$

$P(X_2 = H) = P(X_2 = H \mid X_1 = H) \cdot P(X_1 = H) + P(X_2 = H \mid X_1 = T) \cdot P(X_1 = T)$
$= 0.9 \cdot \frac{1}{2} + (1 - 0.8) = \frac{7}{2} = 0.35$

$P(Y) = \sum_i P(Y \mid X = i) \cdot P(X = i)$ — total probability

$P(\neg X \mid Y) = 1 - P(X \mid Y)$
~~$P(X \mid \neg Y) = 1 - P(X \mid Y)$~~ ← WRONG! You can never negate your condition and assume these values total up to 1

$P(D_2) \quad P(D_2 = \text{sunny}) = 0.9$
$P(D_2 = \text{sunny} \mid D_2 = \text{sunny}) = 0.8$
$P(D_2 = \text{rainy} \mid D_2 = \text{sunny}) = 0.2$
$P(D_2 = \text{sunny} \mid D_2 = \text{rainy}) = 0.6$
$P(D_2 = \text{rainy} \mid D_2 = \text{sunny}) = 0.4$

$P(D_2 = \text{sunny}) = P(D_2 = \text{sunny} \mid D_1 = \text{sunny}) \cdot P(D_1 = \text{sunny}) + P(D_2 = \text{sunny} \mid D_1 = \text{rainy}) \cdot P(D_1 = \text{rainy}) = 0.78$

suppose the same dynamics applied from D_2 to D_3
$\hookrightarrow P(D_3 = \text{sunny}) = P(D_2 = \text{sunny}) \cdot P(D_3 = \text{sunny} \mid D_2 = \text{sunny}) + P(D_2 = \text{rainy}) \cdot P(D_3 = \text{sunny} \mid D_2 = \text{rainy}) = 0.756$

cancer
$P(C) = 0.01$
$P(\neg C) = 0.99$

$P(+ \mid C) = 0.9$
$P(- \mid C) = 0.1$
$P(+ \mid \neg C) = 0.2$
$P(- \mid \neg C) = 0.8$

joint probabilities
$P(+, C) = P(C) \cdot P(+ \mid C) = 0.009$
$P(-, C) = P(C) \cdot P(- \mid C) = 0.001$
$P(+, \neg C) = P(\neg C) \cdot P(+ \mid \neg C) = 0.198$
$P(-, \neg C) = P(\neg C) \cdot P(- \mid \neg C) = 0.792$
prob of − and ¬C

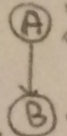
$P(C \mid +) = \frac{P(+, C)}{P(+, C) + P(+, \neg C)} = 0.043$
↑ normalize

Bayes rule (next) →

Bayes rule

$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$ — likelihood, prior; posterior; P(B) marginal likelihood → expanded with total probability

Bayes network



A not observable — P(A) [redacted]

B observable — P(B|A), P(B|¬A) — A causes B to be positive or not (B test)

[redacted]

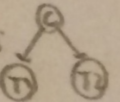
[redacted] diagnostic reasoning → P(A|B) [redacted]
(inverse of causal reasoning) → P(A|¬B) [redacted]
→ P(B|A) [redacted]

$P'(A|B) = P(B|A) \cdot P(A)$ ← Bayes rule non normalized
↑ not real probability
[redacted]
$P'(\neg A|B) = P(B|\neg A) \cdot$ [redacted] $P(\neg A)$

$P(A|B) = \eta P'(A|B)$
$P(\neg A|B) = \eta P'(\neg A|B)$

normalizer $\eta = (P'(A|B) + P'(\neg A|B))^{-1}$

e.g.



$P(C) = 0.01$
$P(+|C) = 0.9$
$P(-|\neg C) = 0.8$

$P(C) \cdot P(+|C) \cdot P(+|C)$ [redacted]

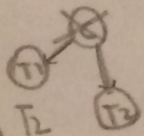
$P(C|++) = 0.1698$ → $\frac{P(C) \cdot P(+|C) \cdot P(+|C)}{P(C) \cdot P(+|C) \cdot P(+|C) + P(\neg C) \cdot P(+|\neg C) \cdot P(+|\neg C)}$

	prior	+	+	P'	P(C|++)
C	0.01	0.9	0.9	0.0081	0.1698
¬C	0.99	[redacted] 0.2	0.9	0.0396	0.8301
				0.0477	

P(C|+-)

	prior	+	−	P'	P(C|+-)
C	0.01	0.9	0.1	0.0009	0.0056
¬C	0.99	0.2	0.8	0.1584	0.9943
				1583	

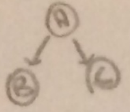
conditionally independent



$P(T_2|C, T_1) = P(T_2|C)$

Given the value of C, then T_2 would be independent of T_1

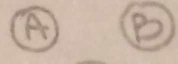
conditionally because it holds true only if we know C



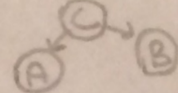
Given A, B⊥C ← independent
written as B⊥C|A

$P(T_2 = + | T_1 = +) = 0.2301$
↓ $P(+_2|+_1, C) \cdot P(C|+_1) +$
$P(+_2|+_1, \neg C) \cdot P(\neg C|+_1) =$
↑ same as total probability but everything is conditioned on $+_1$
$= P(+_2|C) \cdot P(C|+_1) +$
$P(+_2|\neg C) \cdot P(\neg C|+_1)$

A⊥B

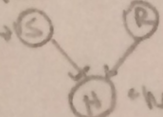


A⊥B|C



=

Different types of Bayes network



$P(S) = 0.7$
$P(R) = 0.01$

$P(H|S,R) = 1$
$P(H|\neg S,R) = 0.9$
$P(H|S,\neg R) = 0.7$
$P(H|\neg S,\neg R) = 0.1$

$P(R|S) = P(R)$
↑ both S and R impact H, but since I don't know anything about H, there's no way S might implicate R.
S and R are independent S⊥R

Explaining away. If we know H, then the cause of S can explain away happiness
If I also know S, it becomes less likely R is true [redacted] to be the

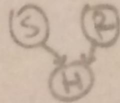
The S is given ↘

$P(R|H,S) = \frac{P(H|R,S) \cdot P(R|S)}{P(H|S)}$
$= \frac{P(H|R,S) \cdot P(R)}{P(H|R,S)\, P(R) + P(H|\neg R,S) \cdot P(\neg R)} = 0.0142$

$P(R|H,\neg S) = \frac{P(H|R,\neg S) \cdot P(R|\neg S)}{P(H|\neg S)}$ ← P(R)
$= 0.0833$ — $P(H|\neg S,R) \cdot P(R) + P(H|\neg S, \neg R) \cdot P(\neg R)$

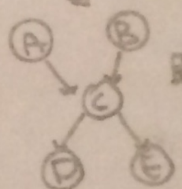
↗ ¬S made much more likely that the cause of H is related to R

S and R are independent, but H adds a dependence between S and R



$P(R|H,S) = 0.0142$
$P(R|S) = 0.01$
$P(R|H,\neg S) = 0.0833$
✓ in the absence of information about H, R is unaffected by knowledge of S
R⊥S

[redacted] Independence does not imply conditional independence



P(A) · P(B) no incoming arcs
[redacted] P(C|A,B)
P(D|C) P(E|C)
O → 2^n variables

Bayes networks define probability distributions over graphs of random variables

The Bayes network is defined by probability distributions that are inherent to each individual node

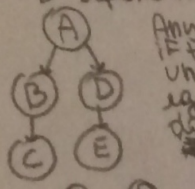
The joint probability represented by a Bayes network is the product of various Bayes network probabilities defined over individual nodes, where each node's probability is only conditioned on the incoming arcs

$P(A,B,C,D,E) = P(A) \cdot P(B) \cdot P(C|A,B) \cdot P(D|C) \cdot P(E|C)$

The joint distribution over 5 variables requires $2^5 - 1$ probability values

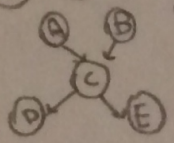
The Bayes network here requires only 10 such values. That makes it scale significantly better to large networks than the combinatorial approach!

D-separation



Any two variables are independent if they are not linked by just unknown variables

eg IF we know B, then everything downstream of B becomes independent of anything upstream of B

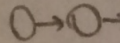


Explain away effect

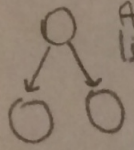
IF we know C to be true, the knowledge of A will affect what we believe about B. IF we know that A is true, we will discount cause B. IF we know that A is False, we will increase our belief for the cause B

D-separation (or reachability)

Active triplets

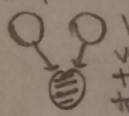


Initial and Final variable are dependent, if all variables are unknown



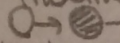
Any structure like this renders the left and right variables dependent

Explain away case

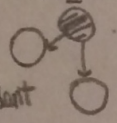


IF the center variable is known, the variables on top become dependent

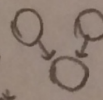
Inactive triplets



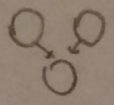
If the center variable is known (so is behind the conditioning bar), the the first and last variables become independent



... unless the center variable is known, in which case left and right variable become independent

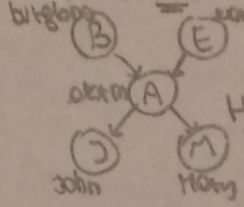


In contrast with a case where all variables are unknown. Here the top variables are independent



IF a direct successor of a conversion variable is known, then we get knowledge of the conversion variable, which goes back to the previous case

Probabilistic Inference



The output for each of the query variables, is going to be a joint probability distribution over those variables

We call this, the posterior distribution given the evidence

$P(Q_1, Q_2 \dots | E_1 = e_1, E_2 = e_2)$ That is

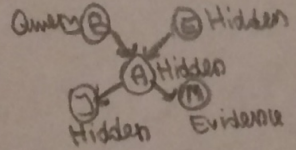
Make likely explanations over all the possible values for the query variables, which combination of values has the highest probability

$\arg\max_{q_1, q_2} P(Q_1 = q_1, Q_2 = q_2 \dots | E_1 = e_1, \dots)$

With Bayes network we're not restricted to go only in the causal direction, which giving evidence variables (the root of the tree) and asking for query vars (nodes at the bottom).

We could have J and M be the evidence variables, and B and E be the query variables. Or we could have any other combination

eg Many calls to report the alarm is going off and we want to know whether there has been a burglary



Enumeration

$P(+b|+j,+m)$ burglar alarm occurred, John called, Mary called

$= P(+b,+j,+m) / P(+j,+m)$

$= \Sigma_e \Sigma_a P(+b,+j,+m,e,a)$ enumerating all possible values of the hidden variables

rewrite this expression in terms of the CPTs in the Bayes network

$= \Sigma_e \Sigma_a P(+b)P(e)P(a|+b,e)P(+j|a)P(+m|a)$

IF we call this $f(a,e)$, then $+$ becomes $f(+a,+e) + f(+a,\neg e) + f(\neg a,+e) + f(\neg a,\neg e)$

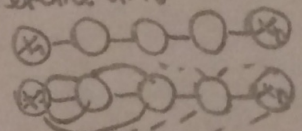
Speeding up enumeration

Have less variables in the inner loop

$= P(+b) \Sigma_e P(e) \Sigma_a P(a|+b,e) P(+j|a) P(+m|a)$

Maximize independence

The structure of a Bayes network determine how efficient it is to inference on it



$O(n)$

$O(2^n)$ IF all n variables are boolean

complete, every node points to every other node

Bayes networks tend to be more compact when written in the causal direction (From causes to effects)

Conditional prob

$P(Q|E) = \frac{P(Q,E)}{P(E)}$

Notation $P(+a) = 1 - P(\neg a)$ prob that a is true

Variable Elimination

in many networks operates much faster than inference by enumeration

raining (R) → (T) traffic → (L) late

$P(+l) = \sum_{+r} \sum_{+t} P(r)\,P(t|r)\,P(+l|t)$

↑ enumeration

- Joining Factors

  $P(R)$

  $P(T|R) \rightarrow P(R,T) = P(R) \cdot P(T|R)$

$P(R,T)$ (R,T) → (L) $P(L|T)$

- Elimination

$P(+t) = P(+r,+t) + P(\bar{r},+t)$

$P(-t) = P(+r,-t) + P(\bar{r},-t)$

T → L

- Join Factors again

$P(+t,+l) = P(+t) \cdot P(+t|+l)$

$P(+t,-l) = P(+t) \cdot P(+t|-l)$

$P(-t,+l) = P(-t) \cdot P(-t|+l)$

$P(-t,-l) = P(-t) \cdot P(-t|-l)$ (TL)

- Elimination again

$P(+l) = P(+t,+l) + P(\bar{t},+l)$

$P(-l) = P(+t,-l) + P(-t,-l)$

If we make a good choice of the order in which we apply these operations, then variable elimination can be much more efficient than just doing the whole enumeration

Approximate Inference

Sampling

1d	2d	
h	h	...
h	t	...
t	h	...
t	t	...

counts, must be enough that we could estimate the joint probability distribution

By sampling we can come up with an approximate value for the joint probability distribution, as opposed to exact inference, where the computation may be very complex

IF we don't know what the conditional probabilities tables are, but we can simulate the process then we can still proceed with sampling whereas we couldn't with exact inference

To [redacted] a conditional probability, we reject all samples that do not [redacted] match the condition (rejection sampling)

This procedure, like the previous one, is consistent. With an infinite number of samples, it computes the true joint probabilities distribution

IF the evidence is unlikely, you'll end up rejecting a lot of samples in rejection sampling

Likelihood weighting, generates samples so we can keep every one

B → A

$P(B,+a)$

Fix the evidence variable

(A will always be positive)

$-b,+a$

$-b,+a$

$+b,+a$

... then we get the samples we want

The resulting set of samples is inconsistent though

we can fix that by assigning a probability to each sample and weighting them correctly

C, R, S, W

$P(+c) = 0.5$

$P(+s|+c) = 0.1$

$P(+s|-c) = 0.5$

$P(+r|+c) = 0.8$

$P(+r|-c) = 0.2$

$P(+w|+s,+r) = 0.99$

$P(R|+s,+w)$

$0.2 \cdot 0.99$ $+c,+s,+r,+w$

$P(C|+s,+r)$, since we see the evidence when we generate a sample that has that evidence we get good ones based on that evidence. But C wasn't and so it will be generating values at random, without looking at R and S, and most of the time it will generate values that don't go well with the evidence with low probability associated with them

Gibbs sampling takes all the evidence into account and not just the upstream evidence

It uses Markov Chain Monte Carlo MCMC

The idea is that we resample just one variable at a time, conditioned on all the others

- Initialize variables to random values, keeping the evidence values fixed

  $+c$ $+s$ $-r$ $-w$

- at each iteration we select one non-evidence variable and resample it based on all the other variables

  $+c$ $+s$ $-r$ $-w$

  ↑

  $+c$ $-s$ $-r$ $-w$ ← new sample

  .....

- [redacted] repeat that again with another variable

+c -s -r -w
↑
+c -s -r -w

- and repeat [redacted]

In MCMC samples are dependent on each other.
This technique is still consistent!

Machine learning

Used to find Bayes networks (or other models) based on data

This unit is about supervised learning

The vast majority of work in the field falls into supervised learning

For each training example...

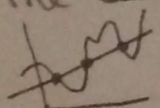
[redacted] Feature vector
$x_1, x_2, x_3 \dots x_n \rightarrow y$ ← Feature label

data
$\vdots$
$x_{m1}, x_{m2}, x_{m3} \dots x_{mN} \rightarrow y_m$
(x_m)

What we want to find out: $f(x_m) = y_m$ ← produce target signal as close as possible
↑ for any [redacted] feature vector
tolerate certain amount of error

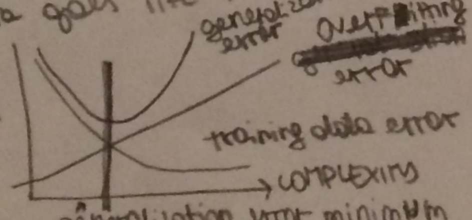
Occam's razor

Everything else being equal, [redacted] choose the [redacted] less complex hypothesis



FIT ⟷ LOW COMPLEXITY

The more complex the hypothesis you allow, the more you can just fit your data. However in reality your [redacted] generalization error on unknown data goes like this:



If you don't find the model that minimizes the training data error but instead push back on complexity, your algorithm will perform better

Spam detection

email → spam / ham
??

[redacted]

spam	ham
offer is secret	play sports today
click secret link	[redacted] went play sports
secret sports link	secret sports event
	sport is today
	sport [redacted] costs money

p(spam) = 3/8

Maximum Likelihood

SSSHHHHH $p(S) = \pi$

$p(y_i) = \begin{cases} \pi & \text{if } y_i = S \\ 1-\pi & \text{if } y_i = H \end{cases}$

11100000   if we write the data like this, then...

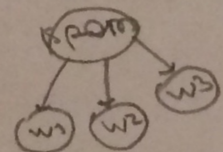
$p(y_i) = \pi^{y_i} \cdot (1-\pi)^{1-y_i}$
this falls when y_i is 0; this falls when y_i is 1

assuming independence → $p(\text{data}) = \prod_{i=1}^{8} p(y_i) = \pi^{\text{count}(y_i=1)} \cdot (1-\pi)^{\text{count}(y_i=0)}$

the data likelihood maximizing number of the probability is indeed the empirical count ([redacted] 3/8 from previous example)

P("secret" | spam) = 1/3
P("secret" | ham) = 1/15

Bayes network where the parameters are estimated using supervised learning by a maximum likelihood estimator, based on training



[redacted] P(spam | "sports") = 1/6

$\frac{[\text{redacted}]\ P(\text{"sports"} \mid \text{spam}) \cdot P(\text{spam})}{P(\text{"sports"} \mid \text{spam}) \cdot P(\text{spam}) + P(\text{"sports"} \mid \text{ham}) \cdot P(\text{ham})}$

Bag of words

representation of a document that just counts the frequency of words

eg Hello I will say hello

dict "hello" I will say
2 1 1 1

different dict "hello" goodbye
2 0

In most cases we make sure that all the words found in messages are actually included in the dictionary

[redacted]

P(spam | m) = 25/26   m = "secret is secret"

$\frac{\frac{3}{8} \cdot \frac{1}{3} \cdot \frac{1}{3} \cdot \frac{1}{3}}{\frac{3}{8} \cdot \frac{1}{3} \cdot \frac{1}{3} \cdot \frac{1}{3} + \frac{5}{8} \cdot \frac{1}{15} \cdot \frac{1}{15} \cdot \frac{1}{15}}$

P(spam | m) = 0   m = "today is secret" [redacted]

"today" doesn't appear in spam

[redacted] $\frac{\frac{3}{8} \cdot 0 \cdot \frac{1}{[\text{redacted}]} \cdot \frac{1}{9}}{\frac{3}{8} \cdot 0 \cdot \frac{2}{9} \cdot \frac{1}{9} \dots}$ ← we're clearly overfitting

Laplace smoothing

ML $p(x) = \frac{\text{count}(x)}{N}$
↑ seen so far

LS (k) $p(x) = \frac{\text{count}(x) + k}{N + k|x|}$

k = 1

messages	spam	
1	1	P(spam) = 2/3
10	6	P(spam) = [redacted]
100	60	[redacted]

P(spam) = 61/102 → $\frac{60+1}{100+2}$ ← k

k: [redacted] classes

The more samples we get, the more LS will get closer to ML [redacted]

$k=1$

$P(spam)=\frac{2}{5}$ $\quad P(\text{"today"}|spam)=\frac{0+1}{9+12}=\frac{1}{21}$ ← k · dict size

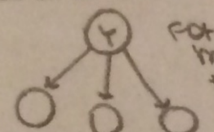
$P(ham)=\frac{3}{5}$ $\quad P(\text{"today"}|ham)=\frac{2+1}{15+12}=\frac{1}{9}$

$P(spam|m)$ m = "today is secret"

$\frac{\frac{2}{5}\cdot\frac{1}{21}\cdot\frac{2}{21}\cdot\frac{4}{21}}{\_\_\_\_ + \frac{3}{5}\cdot\frac{3}{27}\cdot\frac{2}{27}\cdot\frac{2}{27}}$

summary

x_k features of training examples → y^k labels (spam or ham)



For those features we made a model describing the conditional probability of each feature

ML is to fit those primers in LS & the Bayes network

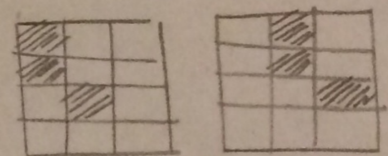
Bayes rule, to take any training examples and figure out what the ham probability was (Y)

This is called a generative model, in that conditional probabilities all aim to maximize the probability of individual features as if those describe the physical world

—

Digit recognition

input vector = pixel values



Those will 100% different

There are different solutions, but a common one is to use smoothing in a different way from the way discussed before

Input smoothing, instead of just counting 1 pixel values count, you could mix it with counts of the neighboring pixel values

If all pixels are slightly shifted, we'll get about the same statistics as the pixel itself

Overfitting prevention

Occam's razor, suggests there's a tradeoff between how well we can fit the data and how smooth our learning algorithm is

In LS we select a value k to make our statistical counts smoother

How to choose k

Cross validation (assumes there's plenty of training examples)

TRAINING DATA		
TRAIN	CV	TEST
80%	10%	10%

TEST ← verify the performance and report that performance

TRAIN ↑ Find all parameters (eg Bayes network probabilities)

CV → Find the optimal k. Train for different values of k, observe how well the training model performs on CV data and maximize over all k's to get the best performance

By keeping the data separate from the beginning, and train on the TRAIN data, you use CV to find how good data is doing on k to fine tune it. And then finally only once you use the test data, you get a fair answer to the question "how well will your model perform on future data?"

Supervised learning

classification, target labels (or classes) are discrete ($Y_i = \{0,1\}$)

regression, predict a continuous quantity ($Y_i \in [0,1]$, $Y_i \in \mathbb{R}$)

Linear regression

$$\begin{bmatrix} x_{11}\ x_{12}\ x_{13}\ x_{14} \dots x_{1n} \to Y_1 \\ x_{m1}\ x_{m2}\ x_{m3}\ x_{m4} \dots x_{mn} \to Y_m \end{bmatrix}$$

data ← Y_i are continuous here

what we look for $F(x)=Y$ (x ← vector)

$F(x)=w_1x+w_0$ ← if x is 1-dimensional (ie n=1)

$F(x)=w\cdot x+w_0$ (← vectors)

$\text{loss}=\sum_j (Y_j - w_1 x_j - w_0)^2$

residual error obtained after fitting the linear function as good as possible

minimizing of loss can be written as

$w^* = \arg\min_w L$ ← solution to the regression problem; ← all possible vectors w

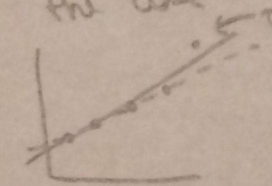
Minimizing quadratic loss

$$w_1 = \frac{m\sum x_i y_i - \sum x_i y_i}{m\sum x_i^2 - (\sum x_i)^2}$$

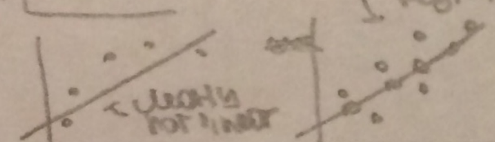
$$w_0 = \frac{1}{m}\sum Y_i - \frac{w_1}{m}\sum x_i$$

=

Linear regression works well if data is approximately linear. But there's many examples where that's not the case

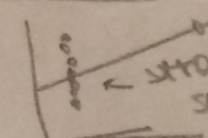


← outliers penalize a lot because we're minimizing quadratic error



← clearly not linear

good linear regression



← strong vertical spread

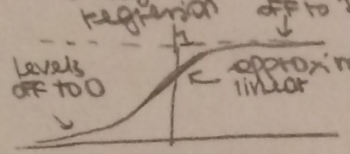
...

Another problem with linear regression is that as you go to infinity in the x-space, your y also becomes infinity. In some problems that is not a plausible model (eg weather)

For such problems there is a model called logistic regression

$F(x)$ → linear function

$z = \frac{1}{1+e^{-F(x)}}$ → output of logistic regression



$F(x) \uparrow \infty \Rightarrow z \uparrow 1$

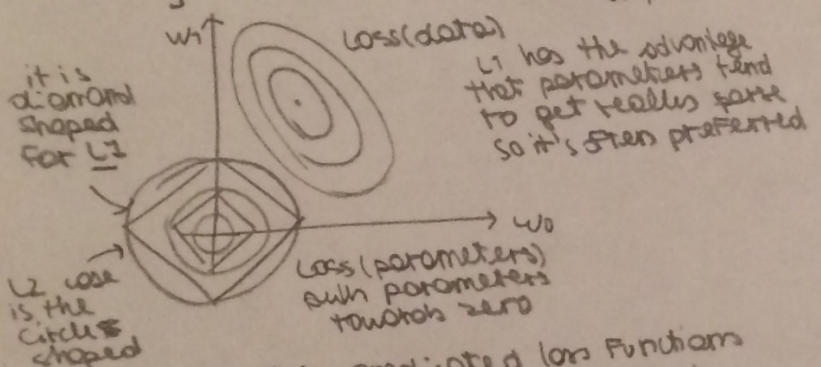
$F(x) \downarrow -\infty \Rightarrow z \downarrow 0$

Regularization

Loss = Loss(data) + Loss(parameters)

(as seen before) (complexity control term)

$\sum_j (Y_j - W_1 X_j - W_0)^2 + \sum_i |W_i|^p$ $p=1$ or $p=2$



L1 has the advantage that parameters tend to get reduced some so it's often preferred

it is diamond shaped for L1

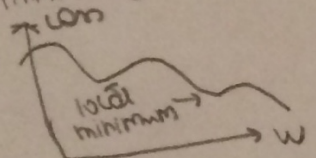
L2 case is the circle shaped

Minimizing more complicated loss functions

gradient descent

start with initial guess w_0 and then iteratively...

$w_{i+1} \leftarrow w_i - \alpha \nabla_w L(w_i)$



$\frac{\partial L}{\partial w_1} = -2 \sum_j (Y_j - W_1 X_j - W_0) X_j$ w_1^*

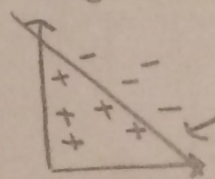
$\frac{\partial L}{\partial w_0} = -2 \sum_j (Y_j - W_1 X_j - W_0)$ w_0^*

$w_1^m \leftarrow w_1^{m-2} - \alpha \frac{\partial L}{\partial w_1}(w_1^{m-1})$

$w_0^m \leftarrow w_0^{m-2} - \alpha \frac{\partial L}{\partial w_0}(w_0^{m-1})$

Perception algorithm

very early model of a neuron



1 if $W_1 X + W_0 \geq 0$ } $F(x)$

0 if $W_1 X + W_0 < 0$ }

linear function

perceptron only converges if the data is linearly separable

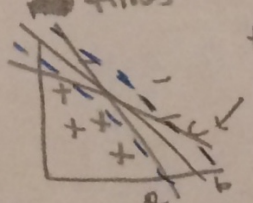
it is iterative like grade descent

start with random guess for w_1 and w_0

$w_i^m \leftarrow w_i^{m-1} + \alpha (Y_j - F(X_j))$

learning weight, error

this is an online learning rule, meaning we process one data at the time and we might go through the data many times



there may be many linear separators. b however seems to be at the same distance from all training examples

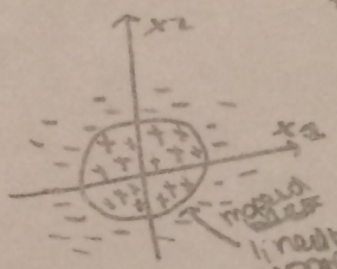
// this region is called the margin

the margin of a linear separator is the distance of it to the closest training example

Maximum margin algorithms

- support vector machines
- boosting

SVMs derives a linear separator that maximizes the margin. By doing so it obtains guarantees over perceptron, which only picks a linear separator without considering the margin.

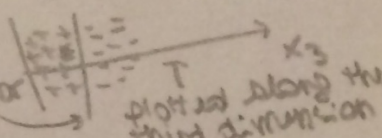


kernel trick

x_1 x_2 x_3

$x_3 = \sqrt{(x_1)^2 + (x_2)^2}$

makes things linearly separable!



this trick can be applied to any linear learning algorithm

Linear methods: summary

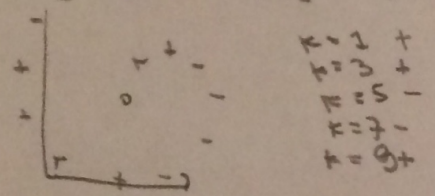
- regression vs classification
- exact solution vs iterative solution
- smoothing
- non-linear problems

K nearest neighbors

parametric
has constant number of parameters (like prob. or weights) it is independent of training set size

non-parametric
the number of parameters can grow (a lot)

- Learning: memorize all data
- label new example by
  - Find K nearest neighbors
  - return majority class label



K is a smoothing parameter, it makes the function less scattered

We can, once again, use cross validation to find the optimal K

problems of KNN → solved these for O(log n) search
- very large datasets
- very large feature spaces

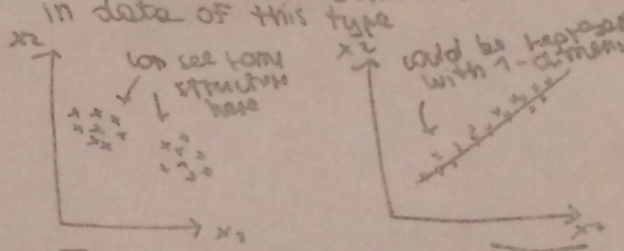
Nearest neighbors works really well for small input spaces (3 or 4 dimensions) but very poorly if our input space is 20-25 or maybe 100 dimensions

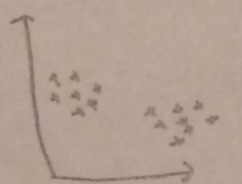
Unsupervised Learning

$$\text{data} \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1N} \\ x_{21} & x_{22} & \dots & x_{2N} \\ \vdots & & & \\ x_{M1} & x_{M2} & \dots & x_{MN} \end{bmatrix}$$

In unsupervised learning we're not given target labels

The task is to find structure in data of this type





Assume the given data is IID, which means identically distributed and independently drawn from the same distribution

Density estimation: recover the density of probability distribution that generated the data
- clustering
- dimensionality reduction

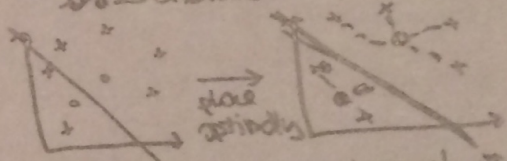
Clustering

Most basic form of unsupervised learning
- k-means
- expectation maximization

k-means

k-means estimates for a given K, say K=2, the best centers of clusters representing those data points. Those are found iteratively by the following algorithm:
- Guess cluster centers at random
- Assign to each cluster center the most likely corresponding data points. This is done by minimizing Euclidian distance.

Each cluster center represents half of the space and the line separating these spaces is the equidistant line (also called Voronoi graph).
- Given this correspondence between data points and cluster centers, find the optimal cluster centers that correspond to the points associated with the cluster center



- Go back and reassign cluster centers with the shifted (after finding the optimal cluster centers) Voronoi diagram
- When the point association doesn't change, we have converged

k-means

Initially: select k cluster centers at random

repeat

Correspond data points to nearest cluster center

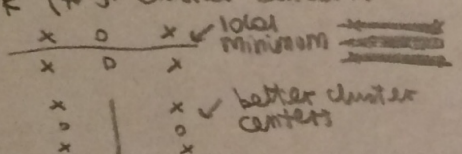
Update cluster center by the mean of the corresponding data points

Empty cluster centers: restart at random

until no change

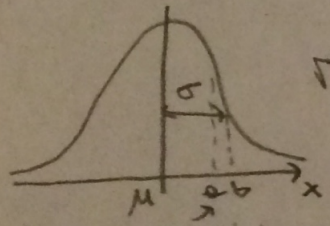
Problems with k-means

- need to know k (# of cluster centers)
- local minimum
- high dimensionality
- lack of math basis



Expectation maximization (k-means generalized)

Gaussian, normal distribution



$$\frac{1}{\sqrt{2\pi\sigma}} \exp\left(-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}\right)$$

exp becomes 0 at $x=\mu$, where it peaks

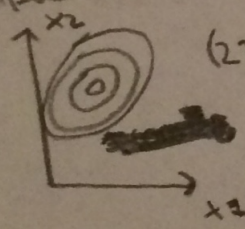
normalizer to make sure that the area underneath sums up to 1

... and goes to 0 exponentially fast for $+\infty$ or $-\infty$

the space is infinite so any individual value x has probability 0.

You can make an interval (a and b), and the area underneath this function is the total probability that and experiment will land up between a and b

Multivariate Gaussian (more than one input variable)



$$(2\pi)^{-\frac{N}{2}} |\Sigma|^{-\frac{1}{2}} \exp\left(-\frac{1}{2}(x-\mu)^T \Sigma^{-1} (x-\mu)\right)$$

generalization of the one dimensional case

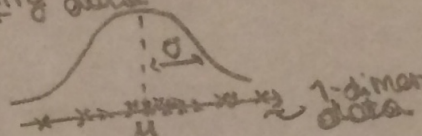
Gaussian learning

Given data points, find the best Gaussian fitting the data

Parameters of a Gaussian: μ, σ

$$f(x|\mu,\sigma^2) = \frac{1}{\sqrt{2\pi\sigma}} \exp\frac{(x-\mu)^2}{2\sigma^2}$$

Fitting data



$\mu = \frac{1}{n}\sum_{j=1}^{n} x_j$ ← the average

n data points

$\sigma^2 = \frac{1}{n}\sum_{j=1}^{n}(x_j-\mu)^2$ ← average deviation

$x_1 \dots x_n$ ← data

$P(x_1 \dots x_n | \mu, \sigma^2)$

for one gaussian model $= \pi_i P(x_i|\mu,\sigma^2)$

the best μ and σ^2 estimates maximize this expression, for a given dataset $x_1 \dots x_n$

$$= \left(\frac{1}{2\pi\sigma^2}\right)^{\frac{n}{2}} \exp -\frac{\sum_i (x_i-\mu)^2}{2\sigma^2}$$

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i-\mu)^2$$

Maximum likelihood estimate

Expectation Maximization

- We have again k randomly chosen cluster centers, say k=2.
- In the correspondence step, [illegible] data points are attracted to a cluster center in proportion to the posterior likelihood.
- In the adjustement step (or the maximization step, the data center correspond to all data points in different strengths. As a result the cluster centers tend not to move as far as in k-means
- Repeat until it converges, like in k-means. However all the correspondences are still alive. Which means that is not a 0,1 correspondence but a soft correspondence

Expectation maximization

$$P(x) = \sum_{i=1}^{k} P(C=i)\, P(x \mid C=i)$$

$P(C=i)$: π_i — $P(x \mid C=i)$: μ_i, Σ_i

← k cluster centers, each has a generic gaussian attached

E-step: assume we know π_i, μ_i and Σ_i

prob that the jth data point corresponds to the ith cluster center

$$e_{ij} = \pi_i \cdot (2\pi)^{-\frac{M}{2}} |\Sigma|^{-\frac{1}{2}} \exp^{-\frac{1}{2}(x_j-\mu_i)^T \Sigma_i^{-1} (x_j-\mu_i)}$$

Figure out π_i, μ_i and Σ_i

M-step:

$$\pi_i \leftarrow \sum_j e_{ij} / M$$

$$\mu_i \leftarrow \sum_j e_{ij} / \sum_j e_{ij}$$

$$\Sigma_i \leftarrow \sum_j e_{ij} (x_j-\mu_i)^T (x_j-\mu_i) / \sum_j e_{ij}$$

Choosing k

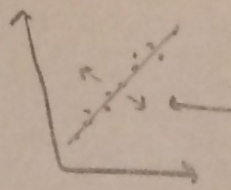
- Periodically evaluate which data is poorly covered by the existing mixture
- Generate new cluster centers at random near unexplained points
- Run the algorithm to see whether the existence of these clusters is still justified ← justification test

minimize: $-\sum_j \log p(x_j \mid \sigma, \Sigma, K) + \text{cost} \cdot K$

- Remove unnecessary clusters and create new random ones to be checked again

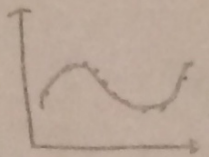
Highly recommended trick for any implementation of expectation maximization as it overcomes local minimum problems to some extent

Dimensionality reduction



1-dimension is really needed

the orthogonal dimensions almost no information



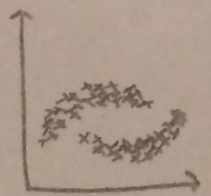
1-dimension here as well. The projection is non-linear though

Linear dimensionality reduction

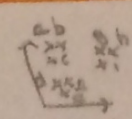
Given data points, find a linear subspace to which to project the data

1. Fit a gaussian
2. Calculate eigenvalues and eigenvectors
3. Pick eigenvectors whose eigenvalues are the largest
4. Project data onto your chosen eigenvectors

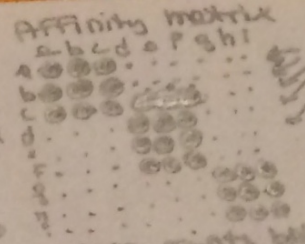
Spectral clustering (affinity-based clustering)



EM and k-means will fail in finding these clusters as the clusters are not defined by a center but defined by affinity (by the presence of nearby points)

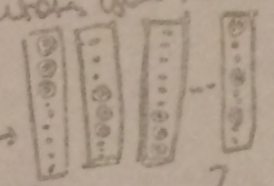


matrix where each data point gets plotted relative to each other data point



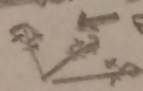
Principle component analysis

to represent the affinity matrix by the most dominant vectors you find in it



By taking the eigenvectors with the largest eigenvalues you reduce the dimensionality of the underlying data space, which is the # of eigenvalues.

If you represent each data vector using the # eigenvectors # before



places the points apart by covariance clustering in that space

Representation with logic

Propositional logic!

B (burglary), E (earthquake), A (alarm), M (Mary calls), J (John calls)

True False

$(E \lor B) \Rightarrow A$ — ∨ = or, ⇒ = implies

$A \Rightarrow (J \land M)$ — ∧ = and

$J \Leftrightarrow M$ — if and only if

$J \Leftrightarrow \neg M$ — ¬ = not

model → {B: True, E: False...}

Truth tables list all possibilities for the propositional symbols

P	Q	¬P	P∧Q	P∨Q	P→Q	P⇔Q
False	False	true	False	False	true	true
False	true	true	False	true	true	False
true	False	False	False	true	false	False
true	[redacted] true	False	true	true	true	true

A valid sentence is always true in every possible model –

A satisfiable sentence is only true in some models –

An unsatisfiable is False for all models

Limitations

UNCERTAINTY OBJECTS SHORTCUTS

[redacted] First order logic!

	world	beliefs
First-order logic	relations, objects, functions	T | F | ?
Propositional logic	Facts	T | F | ?
Probability theory	Facts [redacted]	[0...1]

atomic (problem solving) — a representation of the state is just an individual state, with no pieces inside of it — (A) (B) ← states

Factored — A representation of a individual state [redacted] is factored into several variables — B E A M J ← could be boolean or other types

structured — A state is not just a set of [redacted] values for variables but can include relationships between objects — what we get in First-order logic!

[redacted]

A_1 [redacted] B_3 C_3 D_2 [redacted] ← model

constants: {A, B, C, D, 1, 2, 3}

functions: [redacted]

NUMBER OF {A→1, B→3, C→3, D→2}

relations: LEFT OF {[A,B], [B,C], [C,D]}

RIGHT OF {[B,A], [C,B], [D,C]}

VOWEL {[A]}

RAINY {} {[]}

Syntax!

Sentences

VOWEL(A)

[redacted]

2 = 2 LEFTOF(A,B)

Terms

A, B, 2 ← constants

x, y ← variables

NUMBER OF (A) (return to 1)

OPERATORS:

∧ ∨ ¬ ⇒ ⇔ ()

QUANTIFIERS:

∀x (for all) ∃y (exists)

$\forall x$ VOWEL(x) ⇒ NUMBER OF (x) = 1

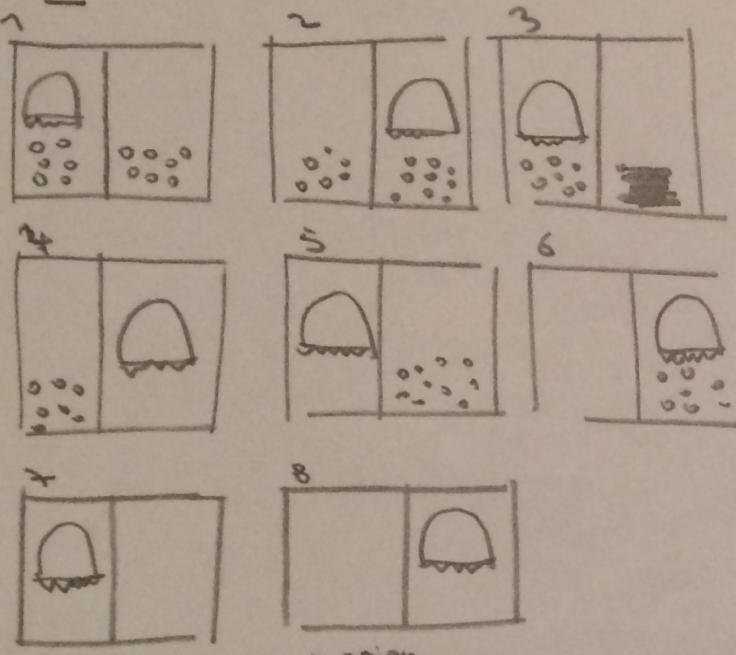
$\exists x$ NUMBER OF (x) = 3

Note, if the quantifier is omitted, you can just assume it means ∀ [redacted] (shortcut)

$\forall x$ tends to go with a conditional

$\exists x$ tends to not go with a conditional

Vacuum world



loc → A, B ← right location
V ← vacuum

dirt → D1, D2

Loc ← true if on location

Vacuum ← = vacuum

Dirt ← = dirt

At(o,l) ← an 'o' object at location 'l'

At(V, A) ← vacuum is at location A ← there's no dirt at any location

$\forall d \forall l \; Dirt(d) \wedge Loc(l) \Rightarrow \neg At(d,l)$

$\exists l \exists d \; Dirt(d) \wedge Loc(l) \wedge At(V,l) \wedge At(d,l)$

↑ vacuum is in a location with dirt

First order means relations are on objects but not on relations (that would be called higher order)

$\forall_2 \; Transitive(R) \Leftrightarrow (\forall a,b,c \; R(a,b) \wedge R(b,c) \Leftrightarrow R(a,c))$

↑ higher order logic statement. Invalid in First order logic

Planning!

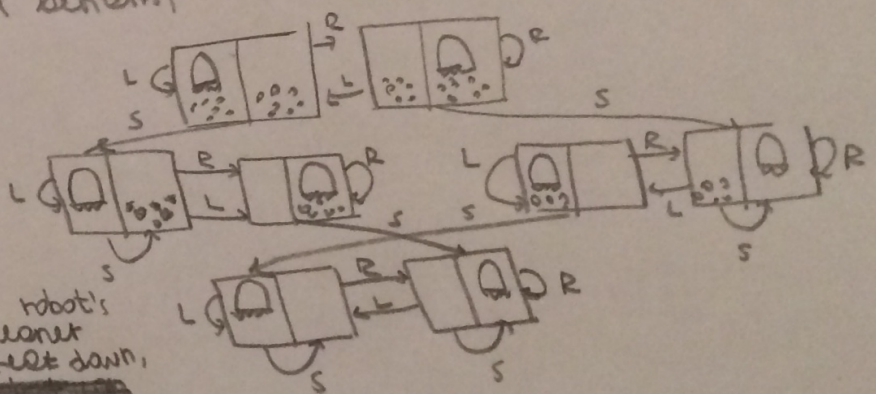
Planning and execution must be interleaved if the environment is

- stochastic
- multiagent
- partial observability

In addition to these, we can also have difficulties because of lack of knowledge on our own part (eg model of the world)

- unknown
- ~~hierarchical~~ hierarchical

(eg in path finding the actions are not going from A to B but turning steering wheel, press pedal and other low level actions)

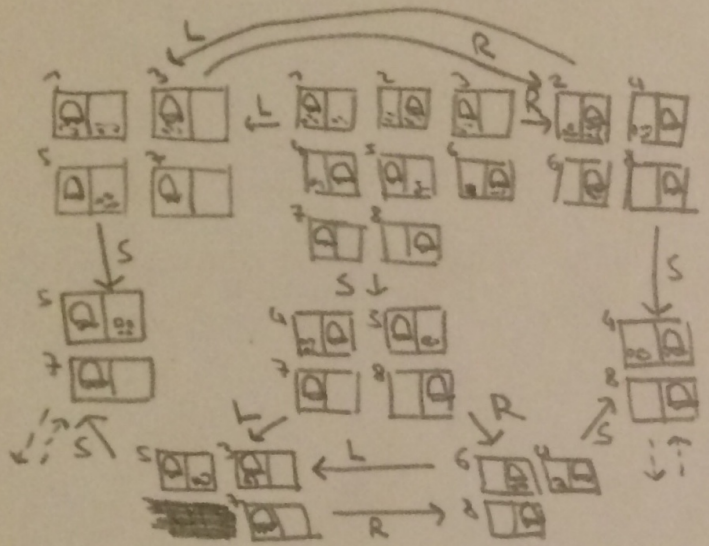


Suppose our robot's vacuum cleaner sensors break down,

and so the robot can no longer perceive either which location it is in or whether there's any dirt

→ Then the agent could be in any one of these states above. We believe that we're in one of all these states, and now when we execute an action we're going to get to another belief state

This is the belief state space for the sensorless vacuum problem



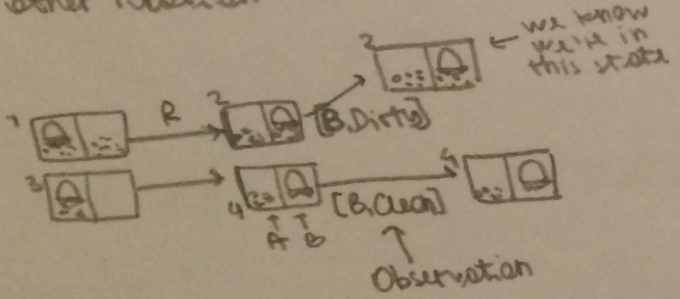
By executing actions we gain knowledge about the world

Going left or right are not inverses of each other in the belief state

It is possible to form a plan that reaches a goal without ever observing the world. Plans like that are called conformant plans

—

Suppose our vacuum can see what's going on in the current location (whether there's dirt or not) but it can't see anything about the other location

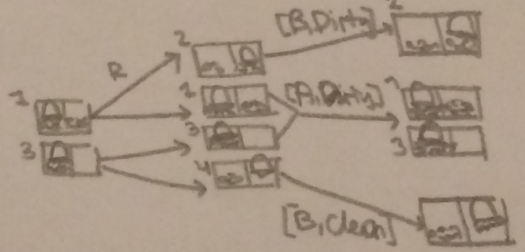


In a deterministic world, each state within a belief state, maps directly into one other one (that's what we mean by deterministic). That means the size of the belief state will either stay the same or it might decrease if two of the actions sort of accidentally end up bringing you to the same place.

The observation works kind of the opposite way. When we observe the world we're taking the current belief state and partitioning it up into pieces

—

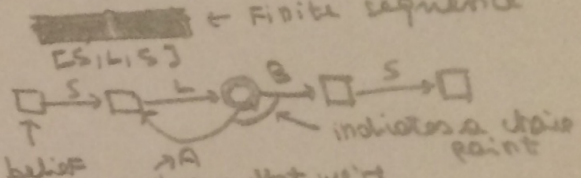
Let's consider a robot that has slippery wheels, so that sometimes when you make a movement (a left or right action), the wheels slip and you stay in the same location, and sometimes they work and you arrive where you expected to go



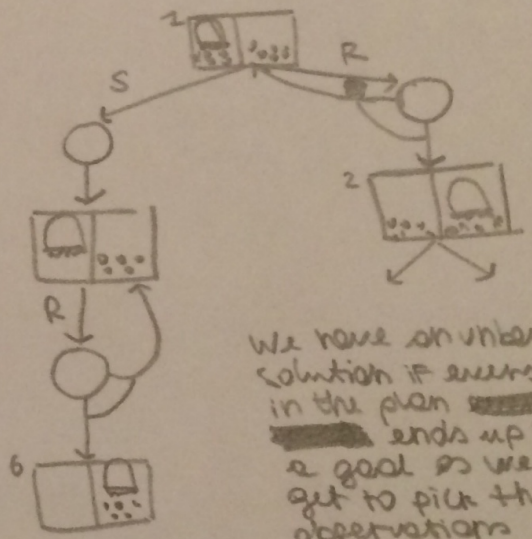
The result of actions will often result in a belief state that's larger than it was before. That is, the action will increase uncertainty because we don't know what the result of the action is going to be.

But in terms of the observations, the same thing holds as in the deterministic world (this is stochastic), the observation partitions the belief state into smaller states

In the stochastic world, no finite plan is guaranteed to always work



A path to the goal can be found through search, just as we did in problem solving



We can guarantee a bounded solution if we have no loops in our plan

Mathematical Notation

$S' = \text{Result}(S, a)$

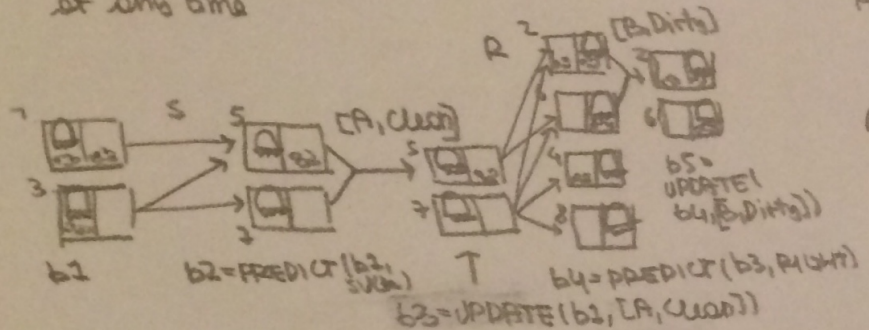
$[A, S, F]\ \text{Result}(\text{Result}(A, A \to S), S \to F) \in \text{Goals}$

$b' = \text{Update}(\text{Predict}(b, a), o)$

stochastic environments — predict what our action will do — update based on observation o

---

Kindergarten world because there are little roaches walking around who can deposit dirt in any location at any time



Classical Planning

representation language for dealing with states, actions and plans, and it's also an approach for dealing with the problem of complexity by factoring the world into variables

STATE SPACE: ← Boolean (2^n) ← all possible assignments to n boolean variables
e.g. vacuum world: DirtA, DirtB, VacA

WORLD STATE: complete assignment, assign true or false to each of the variables

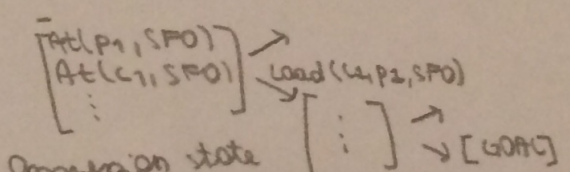
BELIEF STATE: some variables are assigned others are not → partial assignments, complete assignments, arbitrary formula T ... in boolean logic

deterministic, fully observable domain

Action(Fly(p, x, y)
PRECOND: Plane(p) ∧ Airport(x) ∧ Airport(y) ∧ At(p, x)
EFFECTS: ¬At(p, x) ∧ At(p, y))

Init(At(C1, SFO) ∧ At(C2, JFK) ∧ At(P1, SFO) ∧ At(P2, JFK) ∧ Cargo(C1) ∧ Cargo(C2) ∧ Plane(P1) ∧ Plane(P2) ∧ Airport(JFK) ∧ Airport(SFO))
Goal(At(C1, JFK) ∧ At(C2, SFO))
Action(Load(c, p, a),
PRECOND: At(c, a) ∧ At(p, a) ∧ Cargo(c) ∧ Plane(p) ∧ Airport(a)
EFFECT: ¬At(c, a) ∧ In(c, p))
Action(Unload(c, p, a),
PRECOND: In(c, p) ∧ At(p, a) ∧ Cargo(c) ∧ Plane(p) ∧ Airport(a)
EFFECT: At(c, a) ∧ ¬In(c, p))
Action(Fly(p, From, To),
PRECOND: At(p, From) ∧ Plane(p) ∧ Airport(From) ∧ Airport(To)
EFFECT: ¬At(p, From) ∧ At(p, To))

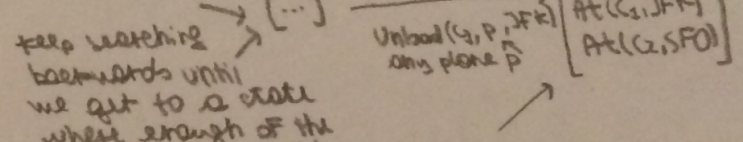
Progression search



Progression state space search

Search through the space of exact states

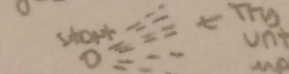
Regression search
... in which we start a the goal

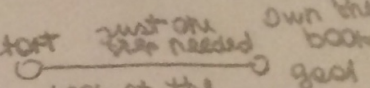


Regression vs Progression

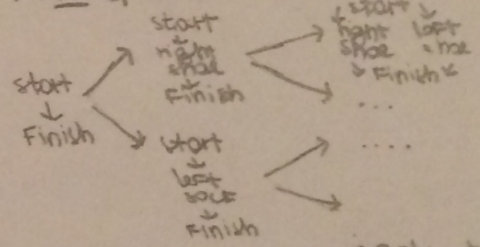
Action(Buy(b)
PRE: ISBN(b)
EFF: Own(b))

Goal(Own(0136042597))





Plan space search



Popular in the 1980's, but it's faded from popularity. Right now, the most popular approach has to do with forward search.
The advantage of forward search seems to be that we can come up with very good heuristics. So we can do heuristic search.
It seems to be easier to come up with good heuristics since forward search deals with concrete plan states

Sliding puzzle example

Action(Slide(t,a,b)
PRE: On(t,a) ∧ Tile(t) ∧ Blank(b) ∧ Adj(a,b)
EFF: On(t,b) ∧ Blank(a) ∧ ¬On(t,a) ∧ ¬Blank(b))

with this kind of formal representation we can automatically come up with good heuristics

IF we come up with a relaxed problem by going in and throwing out some of the prerequisites (thus making the problem strictly easier), then you get a new heuristic

Because we have our actions encoded in this logical form, a program can edit that and come up with good heuristics

—

Situation calculus

Moving all the cargo from airport A to airport B, regardless of how many pieces of cargo there are.

You can express the notion of "All" in First order logic

Actions: objects Fly(p,x,y)
situations: objects ← correspond to paths
S0 S'=Result(S,a) ← action
↑ initial situation
Poss(a,s) ← action a possible in state s
SomePlaneAt(s) ⇒ Poss(a,s)

Plane(p,s) ∧ Airport(x,s) ∧ Airport(y,s) ∧ At(p,x,s) ⇒ possibility axiom Poss(Fly(p,x,y),s) for the action Fly

← these types of predicates that can vary from one situation to another are called Fluents

↙ Thus always refer to a specific situation s that we always put last in the predicate

successor state axioms are used to describe what happens in the state that's a successor of executing an action

∀a,s Poss(a,s) ⇒ (Fluent true ⇔ a made it true ∨ a didn't undo it)

Assume ∀ for each variable without a quantifier

↳ Poss(a,s) ⇒ In(c,p,Result(s,a)) ⇔ (a = Load(c,p,x) ∨ (In(c,p,s) ∧ a ≠ Unload(c,p,x)))

≡
≡
so At(p,JFK,s0)
∀c Cargo(c) ⇒ At(c,JFK,s0)

goal ∃s ∀c Cargo(c) At(c,SFO,s) ← move all cargo from JFK to SFO

The advantage of situation calculus is that we have the full power of First order logic

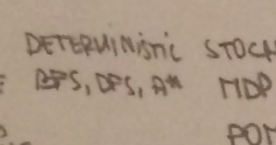
Planning under uncertainty

- things discussed in this class so far
- Today's class puts together PLANNING and UNCERTAINTY using Markov decision problems and partial observable Markov decision processes (MDPs and POMDPs)

FULLY OBSERVABLE	DETERMINISTIC	STOCHASTIC
	DFS, DPS, A*	MDP
PARTIALLY OBSERVABLE		POMDP

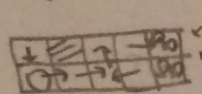
Markov decision processes

states $S_1 \dots S_N$
actions a_1 or a_2
state transition matrix $T(s, a, s') = P(s' | a, s)$
reward $R(s)$ for each state s



The Finite state machine becomes Markov if the outcomes are somewhat random*

MDP gridworld

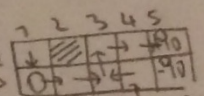


when those states are reached, the task is done
80% ← trying to move forward
10% 10%

A policy assigns actions to any state

$\pi(s) \rightarrow A$

The planning problem becomes finding the optimal policy



reward function to denote the cost of an action
$R(s) = \begin{cases} +10 & \text{as} \\ -10 & \text{b5} \\ -3 & \text{other states} \end{cases}$

incentive to shorten the final action sequence

80% 10% 10%

By choosing that direction we never risk to fall into the −10 cell, so that move is preferred to the one that will get us closer to the goal

expectation
$E\left[\sum_{t=0}^{\infty} \gamma^t R_t\right] \rightarrow \max$

↑ sum of all future rewards (t is time)

γ discount factor = 0.9 ; discounts future rewards relative to immediate rewards (alternative to the negative reward in the reward function (−3 above))

Value function

$V^{\pi}(s) = E_{\pi}\left[\sum_t \gamma^t R_t \mid s_0 = s\right]$

For any possible policy that assigns actions to the non-absorbing states you can now simulate the agent and compute what is the average reward that is being received until you finally hit a goal state

Value iteration

recursively calculate the value function so that we get the optimal value function.

$V(s) \leftarrow \left[\max_a \gamma \sum_{s'} P(s' | s, a) V(s')\right] + R(s)$

(fixed-up)

- For any given state
- maximize over all possible actions
- possible successor state that we look up in the existing table

$V(s) \leftarrow R(s)$ for terminal states

This process of updates converges as Bellman has shown a while ago

When it converges, we have a Bellman equality

policy $\pi(s) = \underset{a}{\operatorname{argmax}} \sum_{s'} P(s' | s, a) V(s)$

- For any state and any action a
- picks the action a that maximizes the value function

summary

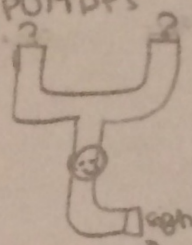
Fully observable	$S_1 \dots S_N$ $a_1 \dots a_K$
stochastic	$P(s' \mid a, s)$
reward	$R(s)$
objective	$E \sum \gamma^t R^t \rightarrow \max$

↑ expected

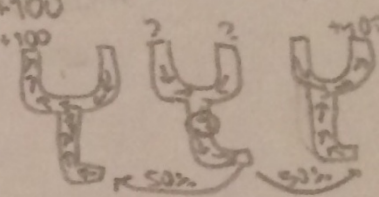
value iteration $V(s) \leftarrow$ assign a value to each state

converges $\pi = \operatorname{argmax} \dots\dots$

POMDPs



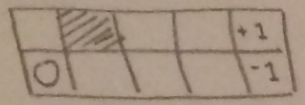
tells the agent which entrance gives +100



belief space

In POMDPs some actions might be information-gathering actions whereas others might be goal driven actions

Reinforcement Learning



What if we don't know where the +1 and -1 rewards are? A reinforcement learning agent can learn to explore the territory, find where the rewards are, and then learn an optimal policy

Forms of learning

supervised $(x_1, y_1) \ldots (x_n, y_n)$ $y = F(x)$

unsupervised $x_1 \ldots x_n$ $P(X = x)$ ← probability distribution

reinforcement s, a, s, a, s, a $\pi(s)$ ↑ r = rewards

MDP review

$s \in S$ ← set of states s_0 ← initial state

$a \in Actions(s)$ ← set of actions

$P(s' | s, a)$ ← probability distribution as the world is stochastic
= transition function
$T(s, a, s')$

$R(s, a, s')$ ← reward function

$R(s')$ ← we don't care how you got to this state

$\pi(s)$ ← policy to solve an MDP

$\sum_t \gamma^t R(s_t, \pi(s_t), s_{t+1})$ ← the optimal policy maximizes that result

↑ The sum might be infinite so γ^t makes it bounded by counting future rewards less than current rewards

$U(s) = \operatorname{argmax}_a \sum_{s'} P(s' | s, a) U(s')$ ← expected value

↑ utility of any state s, is the maximum over all actions we can take in s of the expected value of taking that action

What if you don't know R and/or P? Then you can't solve the MDP

With reinforcement learning, you can learn R and P by interacting with the world. Or you can learn substitutes that will tell you as much as you know to that you'll never actually have to compute with R and P

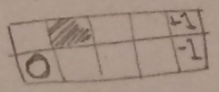
What you learn depends on what you already know and what you want to do

agent	know	learn	use
utility-based agent	P	$R \to U$	U
Q-learning agent		$Q(s, a)$	Q
reflex agent		$\pi(s)$	π

Passive RL, the agent has fixed policy that executes and learn about R or P while executing the fixed policy

Active RL, we change the policy as we go. It also gives you the possibility to explore (actions just for the purpose of learning to be able to do better in the future

Temporal difference learning (TD)



↑ passive

π
$U(s)$
$N(s)$ ← number of times s was visited

- Run the policy
- When it gets to a terminal state, start it over again from the start
- Keep track of how many times each state is visited
- Update the utilities and we'll get a better and better estimate for the utility

if s' is new then $U[s'] \leftarrow r'$
if s is not null then
  increment $N_s[s]$
  $U[s] \leftarrow U[s] + \alpha(N_s[s])(r + \gamma U[s'] - U[s])$

↑ learning rate, if we are sure we want to make a big step, but if we've seen this state a lot & times we're pretty confident and want to make a small step

It converges to the correct utility for the policy π

passive learning algorithms might have one of these problems
- long convergence
- limited by policy
- missing states
- poor estimate

All are corrected because passive learning sticks to the same policy throughout

(greedy ← active)

It uses the same TD learning algorithm we've seen before, but after the change to the utilities we recompute the optimal policy

If the initial policy was flawed, the greedy algorithm would tend to move away from the initial policy towards a better policy

Utility estimates could be off if
- we haven't sampled enough (low # $N(s)$ for state s)
- we could get a bad utility if our policy was off

In the pseudocode above

$U[s] = +R \quad N_s < \ell \; ?$
$U(s)$ ← exploration threshold
↑ when a state has been visited ℓ times

exploratory agent does much better than the passive and greedy agents, converges much faster to much better results!

$\pi^*(s) = \underset{a \in A(s)}{\operatorname{argmax}} \sum_{s'} P(s'|s,a)\, U(s')$

Say we've learned our utility and we can use our policy to act in the world. Although we need the transition model if it is not given, to be able to apply our policy even though we know the utilities

Q Learning, we don't learn U directly and we don't need the transition model. What we learn is a direct mapping "Q" from states and actions to utilities

$$\pi^*(s) = \underset{a \in A(s)}{\operatorname{argmax}} \sum_{s'} Q(s,a)$$

We start with a table of Q values, each state in the table is divided into different actions all starting with Q utility of 0 and are updated as we go

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R(s) + \gamma\, Q(s',a') - Q(s,a))$$

← update formula

—

A state can be represented by a collection of Features instead by exhaustively listing everything that's true in the state

$s = [F_1, F_2, F_3, F_4 \dots F_n]$

Then our utility of a state would be

$$Q(s,a) = \sum_i w_i F_i$$

With this formulation similar states have the same value

weights, our task then is to learn good values for these weights, now input is each feature

$w_i \leftarrow \dots$

it is also updated like $Q(s,a)$

HMMs and Filters

↑ Hidden Markov Models

Are used to analyze or predict time series

$S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow \dots$

$\downarrow \quad \downarrow \quad \downarrow$

$Z_1 \quad Z_2 \quad Z_3 \dots$

↑ each state also emits a measurement. Rather than being able to observe the state itself what you get to see are measurements (that's why it's called 'hidden')

← Bayes network. sequence of states that evolve over time and each state depends only on the previous state. Markov chain

Stationary Distribution

$P(A_t) = P(A_{t-1}) \leftarrow X$

$P(A_t|A_{t-1})P(A_{t-1}) + P(A_t|B_{t-1}) \cdot P(B_{t-1})$

$x = 0.5 \cdot x + 1 \cdot (1-x)$

$x = -0.5x + 1$

$1.5x = 1 \rightarrow \frac{2}{3}$ $\quad \frac{1}{3}$

0.5 G A ⇄ B (0.5 / 0.5)

↑ chance of A in the stationary distribution $P(A_\infty)$

↑ chance of B $P(B_\infty)$

Finding transition probabilities

1 G R ⇄ S 0.5 (1 / 0.5)

R S S S R S R

↑ observations

MAXIMUM LIKELIHOOD

$P(R_0) = 1$

$P(S|R) = 1$

$P(R|R) = 0$

$P(S|S) = 0.5$

$P(R|S) = 0.5$

Laplacian smoothing

GR ⇄ S

$k=1$

$P(R_0) = \frac{1}{3}$ [redacted]

RSSSS

↑ observation

$1\begin{cases} P(S|S) = \text{[redacted]} \frac{4}{5} \leftarrow k \\ P(R|S) = \frac{0+1}{3+2} \end{cases}$

MLE ↑ # outcomes

$1\begin{cases} P(S|R) = \frac{2}{3} \\ P(R|R) = \frac{1}{3} \end{cases}$

Happy Grumpy problem

The get modification to go to HMMs is that the status R and S are hidden

0.6 ↺ R ⇄ S ↻ 0.8 (R → S: 0.4, S → R: 0.2)

$P(R_0) = 0.5$
$P(S_0) = 0.5$

H_1 ← observe day 1 to be H

← posterior probability

$P(R_1|H_1) = \frac{P(H_1|R_1)P(R_1)}{P(H_1)}$ ← Bayes rule
= 0.229

$P(H_1|R_1) \cdot P(R_1) + P(H_1|S_1)P(S_1)$ $\quad P(R_1|R_0)P(R_0) + P(R_1|S_0)P(S_0)$

HMM equations

$x_1 \to x_2 \to x_3 \to x_4$
$\downarrow \quad \downarrow \quad \downarrow \quad \downarrow$
$z_1 \quad z_2 \quad z_3 \quad z_4$

Given state x_i, all the states before, [redacted] after it and z_i are conditionally independent

$x_1 \downarrow z_1$ $\quad P(x_1|z_1) = \frac{P(z_1|x_1)P(x_1)}{P(z_1)}$ [redacted] $\propto P(z_1|x_1)P(x_1)$ ← remember to normalize

$P(z_1)$ ← doesn't depend [redacted] on x_1 therefore...

$P(x_{i+1}) = \sum_{x_i} P(x_i) P(x_{i+1}|x_i)$ ← sum over all previous x_i's

For these equations you need

$P(x_0)$ initial state distribution

$P(x_{i+1}|x_i)$ ← state distribution

$P(z_i|x_i)$ ← measurement distribution

Particle Filters

The topic is robot localization. the robot has sensor data.

The [redacted] task is to figure out where the robot is

Particle Filters represent the space by a collection of points, or particles. Each [redacted] one is an hypothesis where the robot might be.

A point is a concrete value with XY location and direction where the robot might be headed.

The sum or set of all these vectors together form the belief space

The density of these guesses represent the posterior probability of being [redacted] at a certain location [redacted]

Each particle is a representation of a possible state, and the more consistent the particle with the measurement, the more [redacted] likely [redacted] it represents the actual robot location given the map of the environment [redacted]

```
particle_filter(S, U, Z)    ← particles with weights; U control; Z measurement vector
    S' = Ø   η = 0
    for i = 1 ... n          ← # particles
        sample j ~ {w} with replacement      ← weight of j
        x' ~ P(x' | U, S_j)                  ← new particle set controlled
        w' = P(Z | x')
        η = η + w'
        S' = S' ∪ {<x', w'>}
    end
    for i = 1 ... n
        w_i = (1/η) w_i
```

They only put computation resources in places with high probability!

Problems

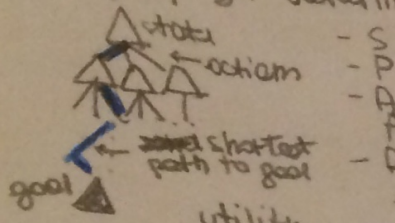
- don't work in high-dimensional spaces
- Problems with degenerative cases

work well in many applications

Games

they form a well defined subset of the real world and are self contained with rules that we can understand. So they're not as messy as, say driving a car.

Single player deterministic



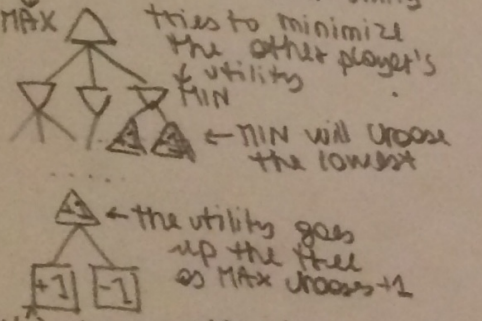
- S set of states containing s_0
- P set of players
- Actions(s,p) allowable actions for player p in state s
- Result(s,a) → s' result of applying action a in state s
- terminal(s) → bool is it the end of the game?
- U(s,p) → int, player p score

utility

deterministic, two player, zero-sum games

the sum of the utilities of the two players is 0

tries to maximize its own utility



U(s)=+1, the utility is only defined at terminal states

minimax

routine to compute the value for a given state s, therefore allowing us to make the best possible move

```
if s is □ : U(s)
if s is △ : maxValue(s)
if s is ▽ : minValue(s)

def maxValue(s):
    m = -∞
    for (a,s') in successors(s):
        v = value(s')
        m = max(m,v)
    return m
```

Complexity

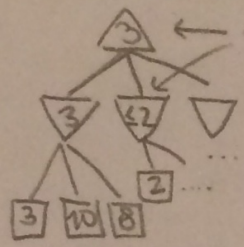
$O(b^m)$ time

$O(bm)$ space

we only look at a path at a time in a depth-first search order, so we have b nodes at each level for m levels

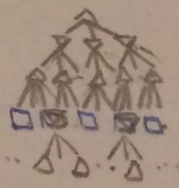
For a game like chess with average m=30 and b=40 it takes TOO long

reduce b (branching factor)



'MIN' will choose 2 or a smaller value if present, and MAX above will choose 3. There is no reason to expand the other nodes and their subtrees

reduce m (depth)



cut off the search at a certain depth and pretending they're all terminals there

The value is estimated using an evaluation function which given a state s returns an estimate of the final value for that state

We get evaluation from playing the game before, seeing similar situations and figuring out what their value is

```
def value(s, depth, α, β):
    if CUTOFF(s, depth): Eval(s)
    if s is □ : U(s)
    if s is △ : maxValue(s, depth, α, β)
    if s is ▽ : minValue(s, depth, α, β)

α best value for MAX
β best value for MIN

value(s0, 0, -∞, +∞) initial call

def maxValue(s, depth, α, β):
    v = -∞
    for a,s' in successors(s):
        v = max(v, value(s', depth+1, α, β))
        if v ≥ β : return v
        α = max(α, v)
    return v
```

reduce b

tree → graph

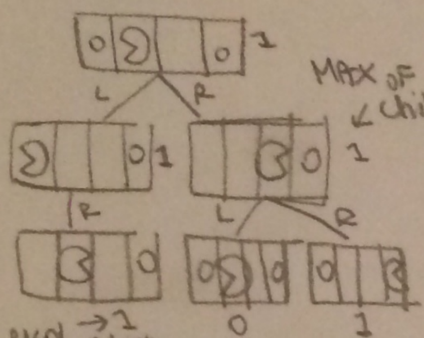
transforming the tree to a graph can be useful in many games

eg chess

- opening books
- closing book
- killer-move heuristic

using the evaluation function is imperfect in that it is just an estimate of the true value of the tree

[redacted]

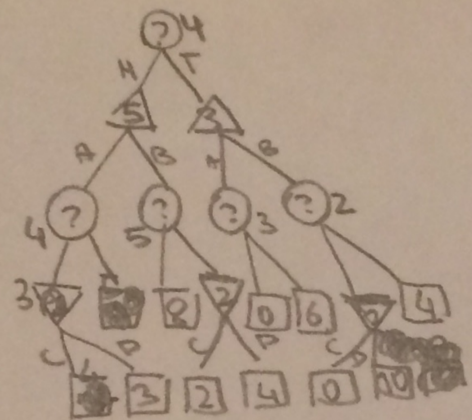


evaluation function counting dots eaten

At the starting state, L and R are equally good, but choosing R will lead [redacted] to a similar tree in the next [redacted] move in which pacman would choose L and go back

This is called the horizon effect ← since by cutting off search we specify an horizon beyond which the agent can't see

↘ If the evaluation function is imperfect we can make these mistakes



Chance

we want to move to stochastic games like backgammon or other games that introduce dice or other sorts of random action

```
def value (s, depth, α, β):
    if [redacted] CUTOFF (s, depth): Eval(s)
    if s is □: V(s)
    if s is △: maxValue(s, depth, α, β)
    if s is ▽: minValue(s, depth, α, β)
    if s is ⓘ: expValue(s, depth, α, β)
```

take the expected value of all the possible results

Game Theory

The study of finding an optimal policy when that policy [redacted] can depend on [redacted] the opponent's policy and vice versa

Prisoner's dilemma
there are two criminals, Alice and Bob, that have a working relationship and they're both caught at the scene of a crime. But the police don't quite have [redacted] enough evidence to put them away so they offer each independently a deal saying "If you testify against your [redacted] cohort, we'll give a better deal and a reduced sentence time. Alice and Bob both understand what's going on

	A: testify	A: refuse
B testify	A=-5 B=-5	A=-10 B=0
B refuse	A=0 B=-10	A=-1 B=-1

Dominant strategy:
A dominant strategy is one for which [redacted] a player does better than any other strategy, no matter what the other player does

For both, testifying is a dominant strategy

Pareto optimal outcome
The pareto optimal outcome is one where there's no other outcome that all players would prefer

so a pareto optimal solution is one where nobody can unilaterally switch their strategies to improve themselves, without making someone else worse off

A=-5, B=-5	A=-10, B=0 ✓
A=0, B=-10 ✓	A=-1, B=-1 ✓

✓ pareto [redacted] optimal

Equilibrium
No player can [redacted] benefit from switching to a different strategy, assuming that the other player stays the same

John Nash proved that every game has [redacted] [redacted] at least one equilibrium point

A=-5, B=-5 is an equilibrium point

Game console game

	A: blu	A: dvd
B blu	A=+9, B=+9	A=-4, B=-1
B dvd	[illegible]	A=+5, B=+5

A: blu, A: dvd ← console manufacturer
B blu, B dvd ↑ game manufacturer
o equilibrium

There's no dominant strategy
A=+9, B=+9 is pareto [redacted] optimal

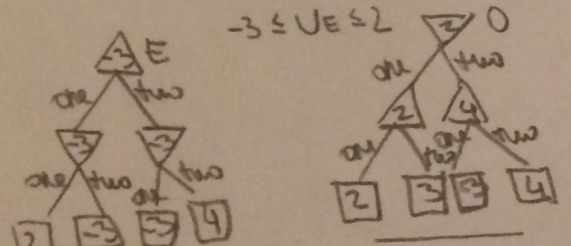
[redacted]

two players, [redacted] and [redacted], simultaneously show either one or two fingers, and if the total number of fingers is even, then E wins that many dollars from O, otherwise the other player wins that many dollars if the total number of fingers is odd from E (zero-sum game)

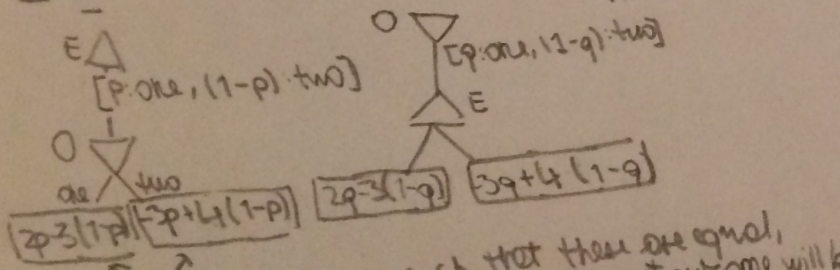
	O: one	O: two
E: one	E=+2	E=-3
E: two	E=-3	E=+4

There's no single move, that's the best strategy for either player, but there is what's called a mixed strategy

A single strategy of always playing either one or the other is called a pure strategy, and a mixed strategy is when you have a probability distribution over the possible moves



Mixed strategy



E wants to choose p such that these are equal, to avoid O knowing what the next outcome will be (and q should be 7/12

IF they both use that strategy then no matter what the opponent played, UE is -1/12

Mechanism design

Someone is going to be running a game that players are going to be participating in, and we want to design the rules of the game such that we get a high expected utility for the people that run the game, the players that play the game, and for the public at large

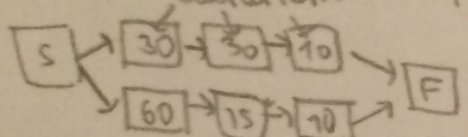
eg advertising game

Advanced Planning

- time
- resources
- active perception
- hierarchical plans

scheduling

- will look at the problem of scheduling a series of tasks, each of which has duration



- scheduling is a process of figuring out a schedule under which we specified the times under which each of these tasks starts such that we can finish as soon as possible

For every task we define the earliest start time ES and the latest possible start time LS for which it's possible to complete the task network in the shortest possible total amount of time

$ES(S) = 0$

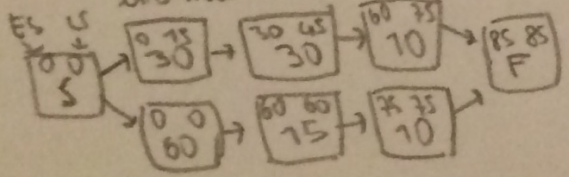
$ES(B) = \max_{A \to B} ES(A) + Duration(A)$

(any state B, all its predecessors)

$LS(F) = ES(F)$

$LS(A) = \min_{B \to A} LS(B) - Duration(A)$

(any now A)



Resources, are things that are used somewhat in the plan

resources could be handled in the language of classical planning

Action(Inspect(n_1, n_2, n_3, n_4, n_5, b_1, b_2, b_3, b_4, b_5),

PRE: Fastened(n_1, b_1) ... Fastened(n_5, b_5),

EFF: Inspected)

Action(Fasten(n, b)

PRE: Nut(n) ∧ Bolt(b)

EFF: Fastened(n, b) ∧ ¬Nut(n) ∧ ¬Bolt(b))

Init(Nut(N_1) ... Nut(N_4), Bolt(B_2) ... Bolt(B_5))

Goal(Inspected)

we can't achieve the goal (not missing) and we consider $4! \times 5!$ paths before we discover that (this) this all combinations of nuts and bolts)

The idea is to make nuts and bolts identical rather than not each one to be distinct

Extended language of classical planning to handle resources

Action(Inspect($n_1 \ldots n_5, b_1 \ldots b_5$), PRE: Fastened($n_1, b_1$) ... Fastened($n_5, b_5$) EFF: Inspected) USE: Inspector(1)

Action(Fasten(n, b), CONSUME: Nuts(1), Bolts(1) ← consuming resources, they're gone once used EFF: Fastened(n, b))

Resources(Nuts(5), Bolts(4), Inspectors(2))

↑ these are resources and how many of each there are

- uses a resource "inspector" while the action is going on, but once the action is completed, the inspector is available for use elsewhere

Hierarchical Planning

"close the abstraction gap"

When planning, it's difficult to deal with things at a low level. We'd rather deal with more abstract plans

Hierarchical task network

Rather than having a plan be a sequence of individual steps, we can talk about higher order steps, where the individual steps can correspond to multiple steps

↑
This idea is called refinement planning

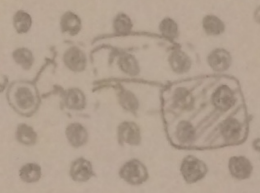
```
Refinement ( go(Home, SFO),
  STEPS : [Drive(Home, SFOLongTermParking),
      Shuttle(SFOLongTermParking, SFO)])

Refinement ( go(Home, SFO),
  STEPS:[ Taxi(Home, SFO)] )


Refinement( Navigate([a,b],[x,y])
  PRECOND: a=x ∧ b=y
  STEPS : [] )
Refinement( Navigate([a,b],[x,y])
  PRECOND: Connected([a,b],[a-1,b])
  STEPS [Left, Navigate([a-1,b],[x,y])])
Refinement( Navigate([a,b],[x,y]) =
  PRECOND: Connected([a,b][a+1,b])
  STEPS: [Right, Navigate([a+1,b],[x,y])])
```

A HTN is FA for every part, at least one refinement achieves the goal

Sometimes we can solve a HTN problem without going all the way down to the concrete steps

Reachable states

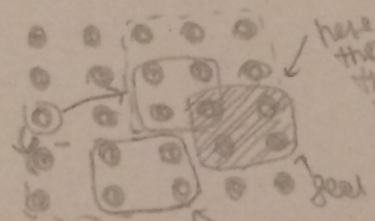
each concrete state within it is a possible refinement.



once a state in a [illegible] reaches a goal, we can follow the arrows backwards to get our refinement

Formulations can be difficult to specify exactly what states can be reachable by an abstract action, because HTN refinements are complicated.

We can go with the notion of an approximate set of reachable states



here it is certain that we get to the goal, however

if the goal was within an upper bound only, we'd be uncertain

lower and upper bound on the states that are reachable

lower bound is one of the things that we know we can get to and the upper is things that we might be able to get to but we're not quite sure

Conformant plan

Extend classical planning to allow active perception to deal with partial observability

```
Init(Object(Table)^Object(Chair)^Can(C1)^
  Can(C2)^InView(Table))
Goal(Color(Chair, c)^Color(Table, c))
Action(RemoveLid(can),
  PRECOND: Can(can)
  EFFECT: Open(can))
Action(Paint(x, can),
  PRECOND: Object(x)^Can(can)^Color(can, c)^
      Open(can)
  EFFECT: Color(x, c))
Percept(Color(x, c),
  PRECOND: Object(x)^InView(x))
Percept(Color(can, c)
  PRECOND: Can(can)^InView(can)^Open(can))
Action(LookAt(x),
  PRECOND: InView(y)^(x≠y)
  EFFECT: InView(x)^¬InView(y))
```

Percept schemas (other than Action schemas). We can perceive the color of something if it's an object and within our field of view.

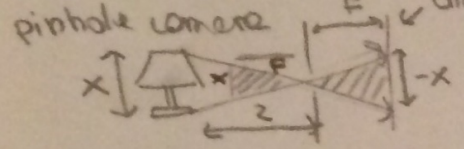
New variable c is introduced. If the PRECOND is true, then you'll learn the value of c

You can already see the robot can without knowing the color to achieve the goal. But perception allows us to come up with better plans by using conditionals (are they already the same color?) is the color of either chair or table available so that we color just the first object?)

Computer Vision

Image Formation is the science of how images are created using cameras

pinhole camera

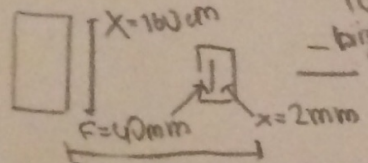


the ratio x to z is the same as x to F and this is called equal triangles

$$\frac{X}{Z} = \frac{x}{F} \qquad x = X \cdot \frac{F}{Z}$$

- the further an object, the smaller it appears
- the longer the F, the larger the projection
- big object, big image

perspective projection



$$Z = X \cdot \frac{F}{x} = 3200\,cm$$

the projected size of any object scales with distance

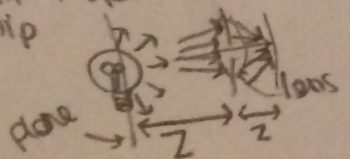
This law applies to x and y dimensions of the image taken

$$x = X \cdot \frac{F}{Z} \qquad y = Y \cdot \frac{F}{Z}$$

One consequence is that parallel lines seem to result in vanishing points

Lens

In a pinhole camera, very few rays of hit the chip. By using a lens, more rays will make it to the same point in the chip



In the drawn situation, everything is in focus on the plane where the lightbulb is. If you move the lightbulb, the resulting projections don't match up. Therefore you have to focus to make sure the distance between the image plane, the lens and the lightbulb are in tune

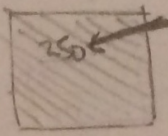
$$\frac{1}{F} = \frac{1}{Z} - \frac{1}{z}$$

Things we can do with images
- classify objects
- 3D reconstruction
- motion analysis

Object recognition

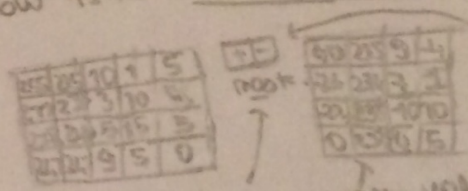
invariance, there's natural variations of the image that don't affect the nature of the object itself and you wish to be invariant to those natural variations
- scale
- illumination
- rotation
- deformation
- occlusion
- viewpoint

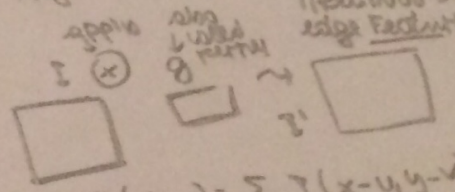
In computer vision we mostly use grayscale images, which are more robust to lighting variations than color



grayscale pixel 0 to 255 black white

One of the very basic things we can do is to extract features





$$I(x,y) = \sum_{u}\sum_{v} I(x-u, y-v)\cdot g(u,v)$$

↑ linear filter

Gradient image

$$I_x = I \circledast \begin{bmatrix} 1 & -1 \end{bmatrix}$$

$$I_y = I \circledast \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

$$E = \sqrt{(I_x)^2 + (I_y)^2}$$

↑ edges in any direction

Canny edge detector, in addition to the gradient magnitude, it thins areas and finds the local maxima, and it tries to connect them in a way that there's always just a single edge

Other masks

$$\begin{matrix}1 & 0 & -1\\ 2 & 0 & -2\\ 1 & 0 & -1\end{matrix} \qquad \begin{matrix}1 & 2 & 1\\ 0 & 0 & 0\\ -1 & -2 & -1\end{matrix}$$
← Sobel masks, there more of them

$$\begin{matrix}1 & 0 & -1\\ 1 & 0 & -1\\ 1 & 0 & -1\end{matrix} \qquad \begin{matrix}1 & 1 & 1\\ 0 & 0 & 0\\ -1 & -1 & -1\end{matrix}$$
✓ Prewitt masks, like the sobel but doesn't emphasize the center line

$$\begin{matrix}5 & -3 & -3\\ 5 & 0 & -3\\ 5 & -3 & -3\end{matrix} \qquad \begin{matrix}5 & 5 & 5\\ -3 & 0 & -3\\ -3 & -3 & -3\end{matrix}$$
← Kirsh masks

A gaussian kernel is a matrix whose value is maximum at the center and falls exponentially through the side

Applying a gaussian kernel blurs an image

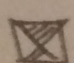
- down sampling, when down-sampling, blur the image first then pick each nth pixel. Blurring avoids funny effects
- noise reduction, pixel noise sometimes make it hard to compute things like gradients

$I \otimes F \otimes g$ (F ↑ gaussian kernel, g ← gradient kernel) $\qquad I \otimes (F \otimes g)$ ↑ convolution is associative

---

Harris corner detector

$\Sigma(I_x)^2 \to$ LARGE
$\Sigma(I_y)^2 \to$ LARGE } CORNER



$\begin{pmatrix}\Sigma I_x^2 & \Sigma I_x I_y\\ \Sigma I_x I_y & \Sigma I_y^2\end{pmatrix}$ we apply eigenvalue decomposition and we get two eigenvalues. If both are large we can say there's a corner

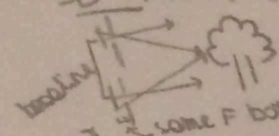
↖ de-rotate using eigenvalues

Modern feature detectors, extend Harris corners into much more advanced features
- localizable
- unique signatures

HOG
histogram-oriented gradients

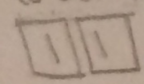
SIFT
scale invariant feature transform

3D vision, attempt to recover the full 3D information

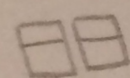


← attempt to recover depth!

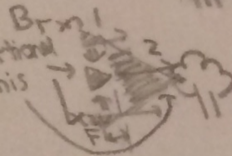
↑ stereo, screen objects and use the displacement (or parallax) of those two different projections to estimate the depth of the object



← it is easy to find displacement in vertical objects

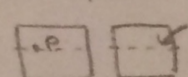


← aperture effect, we can't tell which parts correspond in both images



$z = \frac{FB}{x_2 - x_1}$

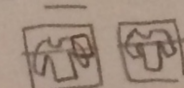
Correspondence



← P will appear on that line

to determine correspondence we match small image patches and/or match features

---



SSD minimization
□ → normalize ↘ ⊖ → $()^2$ → Σ → SSD
□ → normalize ↗ (pixels)

extract patch and compare with patches along the same horizontal axis and match with the patch with least sum of square difference error

Disparity map is the [illegible] of the best match. In the foreground the disparity is much larger than the background in most cases

---

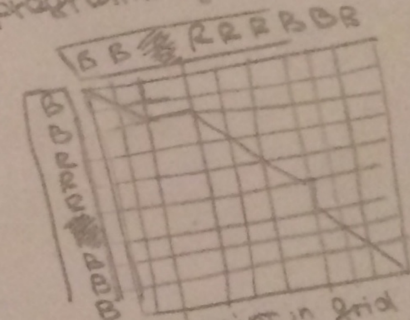
← occlusion

BB▒RRRBB
BBRRRR▒BBB

match the entire scan line to figure out the exact correspondence

cost → match { 0 / 20 ← bad match
↘ occlusion 10

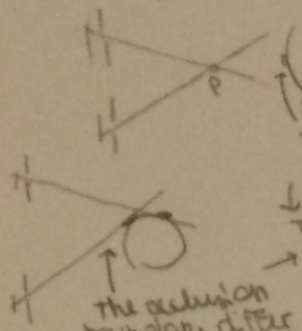
the best possible alignment is computed through dynamic programming



$v(i,j)$ ← point in grid

$v(i,j) = \max \begin{cases} match(i,j) + v(i-1, j-1) \leftarrow \text{diagonal} \\ occl + v(i-1, j) \\ occl + v(i, j-1) \end{cases}$

Correspondence Issues



Also in the case of reflective objects, we get really lousy depth estimates for the object

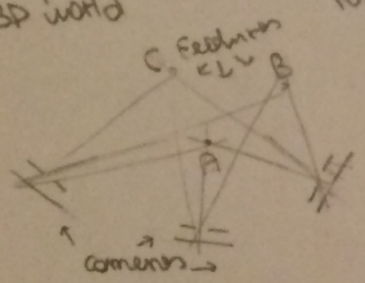
Intentionally adding texture to the scene makes finding correspondence much easier

Another way is to send a beam of light and measuring the time it takes to come back into the sensor (laser range finders) these are really accurate

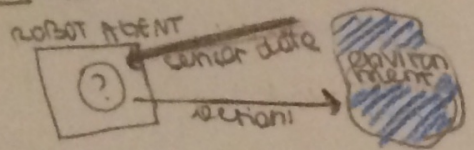
Structure from motion

3D world ↑ location of the camera

From positions of those projected features, recover the cameras and feature position

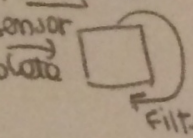


Robotics



robotics is
- continuous
- stochastic
- partially observable

Perception



← try to estimate it such that you can determine what to do next

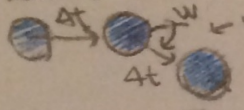
kinematic state: where it is but not how fast it is moving
dynamic state: includes the velocities

Particle Filter: For localization (Or monte carlo localization)

← particles, each has x, y and θ (direction)

predicted → $x' = x + v\Delta t \cos\theta$
→ $y' = y + v\Delta t \sin\theta$
→ $\theta' = \theta + w\Delta t$

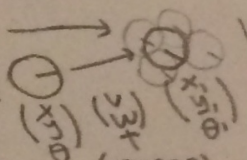
(Fixed velocity v) — turning velocity



← moves straight for Δt, then rotates and keeps going

---

slightly different predictions because of noise

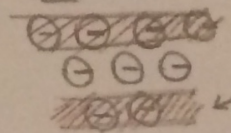


prediction step of a particle filter

In monte carlo localization we don't predict the exact actions but we add noise to v and w

---

So now we have a set of predictions from a single particle. The other important step is the measurement step



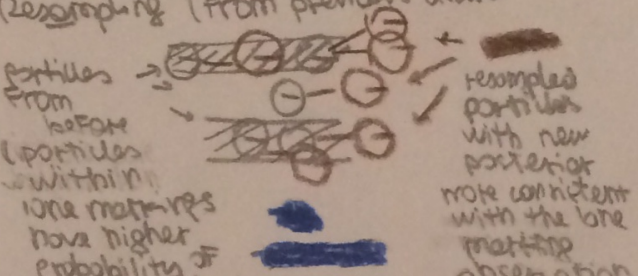
z = { bright / dark

← lane marking

(for the sake of simplicity) if the robot center is aligned with a lane marking it should see a bright spot and if it's on the road it should see a dark spot

P(bright | on lane marker) = 0.8
P(dark | off lane marker) = 0.9

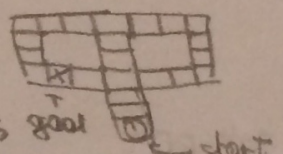
Resampling (from previous draw)



Repeat

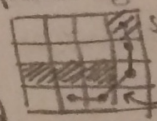
Planning

goal = 100
cost = -1
V(start) = 100 - 5 (distance goal from start)



Robotic path planning

- A state is discrete while the world is continuous



a car can't really do that

≡

Natural language processing

language models

C - A - B
- word-based
- probabilistic
- learned from data

S → NP VP
NP → Name → Sam
VP → slept

- tree based
- logical
- hand-coded

Bag of words

gets the idea across while losing all notion of sequence, thus making it easier to deal with

$P(w_1 w_2 w_3 \ldots w_n) = P(w_{1:n})$
$= \prod_i P(w_i | w_{1:i-1})$

Markov assumption, the effect of one variable on another will be local

$P(w_i|w_{1:i-1}) \approx P(w_i|w_{i-k:i-1})$ ← only go back k steps

$P(w_i|w_{i-1})$ ← for k=1

stationarity assumption, the probability of each variable is going to be the same

$P(w_i|w_{i-1}) = P(w_j|w_{j-1})$

Smoothing ← probabilistic models

---

n-gram models, if we're only looking at word sequences, what is that we're giving up and what are we getting?

letter models, are really good with unique words we haven't seen before, but they still hold properties of the language

The probability tables needed are much more compact than the probability tables for words

classification

- memorize common parts (Steve, Sam, Peter are usually names)
- Frequency of letters (long names with X or Z)

The gzip command can be useful for classification!

EN, DE, AZ ← samples of text in those languages

"This is a new piece of text to be classified" ← new

```
(echo `cat new EN|gzip|wc -c` EN;\
 echo `cat new DE|gzip|wc -c` DE;\
 echo `cat new AZ|gzip|wc -c` AZ)\
 |sort -n|head -1
```

gzip takes a file and looks for common subsequences and represents them in less than 1 byte

When adding a new file with the same subsequences, then that's going to be compressed nicely

Segmentation

given a sequence of language, figure out how to break it into words

This is for languages like Chinese, English has spaces and doesn't have this problem. But we have it in speech recognition

$s^* = \max P(w_{1:n}) = \max \prod_i P(w_i|w_{1:i-1})$
↑ best segmentation

$s^* \approx P(w_i)$ ← will turn out to be good enough and make learning easier

Given n characters, there $2^{(n-1)}$ ways to segment them

that's why the simplified assumption above is so helpful

$s^* = \arg\max_{s=f+r} P(f) \cdot P(s^*(r))$

↑ first word f and rest of the words r

Spelling correction

$c^* = \arg\max_c P(c|w)$ ← correction
↑ best possible correction

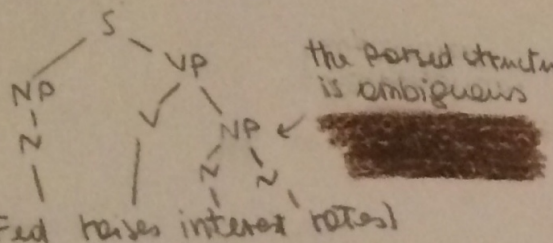
$= \arg\max_c P(w|c) \cdot P(c)$

$P(c)$ = from data counts
$P(w|c)$ = from spelling correction data ← hard to collect that data

Instead we deal with letter to letter errors. much easier to deal with easier data

- transposition
- replacement
- insertion/deletion

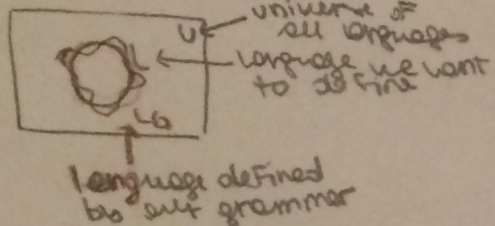
Sentence structure



P (Fed raises interest rates)

trees come from grammars

PCFG → S → NP VP
NP → N | D N | N N | N N N
VP → V NP | V |
N → interest | Fed | rates | raises
V → interest | raises | rates
D → the | a



natural languages are messy and cannot be easily defined

PCFG (probabilistic CFG) P(NP → N | lhs = NP) = .3

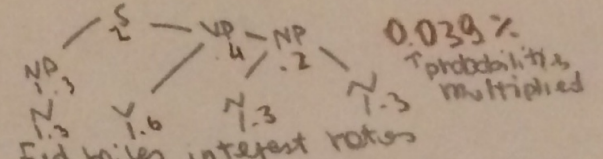
S → NP VP (1)
NP → N (.3)
| D N (.4)
| N N (.2)
| N N N (.1)

VP → V NP (.4)
| V (.4)
| V NP NP (.2)

N → interest (.3)
| Fed (.3)
| rates (.3)
| raises (.1)
...

V → rates (.3)
| raises (.6)
| interest (.1)

D → the (.5)
| a (.5)



Probabilities come from data. In that case however, parse trees must be created first.
"The penn tree bank" is a good resource of already created parse trees

(Lexicalized PCFG) LPCFG
we deal with specific words of the lhs (not categories)

P(VP → V NP NP | V = gave) = .25 ← transitive verb

TO create a tree we can use a top-down or bottom-up search

Machine translation

- word by word transfer, build a table of probabilities for translation of a word
- phrase level, consider multiple phrases at the time
- syntax with trees, probability of a tree in a language corresponding to a similar tree in another language
- meaning or semantics, try to understand what the sentence means and transfer to the corresponding meaning in another language

↑ Vaclaik pyramid