state space size:
- Fundamental counting principle: if there are n variable objects in a given world that can take on values $x_1, x_2, \dots, x_n$, the total # of states is $\prod x_i$.
- eg: pacman can be in 120 distinct (x,y) pos. PM can go NSEW. 2 Ghosts; can be in 12 diff posns. 30 food pellets (can be eaten/not) size: $120 \cdot 4 \cdot 12^2 \cdot 2^{30}$

completeness: if a soln to a search prob exists, is the strategy guaranteed to return a soln given ∞ resources?

optimality: is a strategy guaranteed to find the lowest cost path to a goal state?

branching factor (b): increase in # of nodes on the frontier each time a frontier node is dequeued & replaced w its children is O(b). At depth k in the search tree, ∃ $O(b^k)$ nodes.

max depth (m) · optimal path cost (C^*)

shallowest soln depth (s). · ε (minimal cost btwn 2 nodes in ss graph)

· heuristics - take in a state as an input & output a estimate. two main properties

① Admissability (optimistic):

$0 \le \underbrace{h(n)}_{\text{heuristic}} \le \underbrace{h^*(n)}_{\text{true cost}}$

↳ inadmissable heuristics break optimality by trapping good plans on the fringe

② Consistency: must under-estimate the cost/weight of each edge in a graph } ∀ nodes A, $h(A) - h(C) \le cost(A, C)$

· g(n) - backwards cost computed by UCS

· h(n) - heuristic value fcn or estimated forward cost, used by greedy search

· f(n) - fcn representing total cost, used by A*: $f(n) = h(n) + g(n)$

consistency ⇒ admissability

· dominance: $h_1 \ge h_2$ if ∀n $h_1(n) \ge h_2(n)$

· $h(n) = \max(h_1(n), h_2(n))$ of 2 admissable heuristics is admissable & dominates both

· idea: from each state, locally move towards states w/ higher objective value until a max (hopefully global) is reached.

· state space: ~~complete~~ set of complete configurations

· stochastic hill climbing - selects action randomly from possible uphill moves

· random sideways move hill climbing - allows moves that don't strictly increase obj.

Propositional Logic (PL)

· PL written in sentences composed of propositional symbols (PS)

· model - assignment of T/F to all PS

· for N symbols, 2^N models

· valid sentence - True in all models

· satisfiable - True in ≥1 model

· unsatisfiable - False in all models

· CNF (Conjunctive Normal Form) $(P_1 \vee \dots \vee P_i) \wedge \dots \wedge (P_j \vee \dots \vee P_N)$

Converting to CNF

$A \Leftrightarrow (B \vee C)$

① Eliminate ⇔

$(A \Rightarrow (B \vee C)) \wedge ((B \vee C) \Rightarrow A)$

② Eliminate ⇒

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

③ ¬'s must only appear on literals

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

④ Distribute

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

· entailment (⊨)

↳ A entails B (A ⊨ B) if in all sentences where model A is true, model B is as well

→ if A ⊨ B, the models of A are a subset of the models of B.

· branching factor: # of actions available from a state (aka largest # of successors from any state)

↳ b = 9 for throttle = {-1, 0, 1} steering = {-1, 0, 1}

· a proof is a demonstration of entailment btwn α & β

↳ sound algorithm: everything it claims to prove is entailed

↳ complete algorithm: everything that is entailed can be proved

2 ways to show entailment:

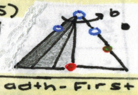
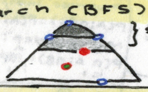
① (A ⊨ B iff A ⇒ B is valid)

↳ direct proof

② (A ⊨ B iff A ∧ ¬B is unsatisfiable)

↳ proof by contradiction

Uninformed Search Summary

Search	Frontier	Completeness	Optimality	Time	Space
Depth-First Search (DFS)  1 node, b nodes, b^2 nodes, b^m nodes	Stack; remove deepest node & replace it w/ its children (children now the new deepest nodes) (LIFO)	tree search - no; graph search: ↳ yes if finite (no cycles) ↳ no if infinite	No. Finds leftmost solution.	$O(b^m)$	$O(bm)$ ↳ bc it maintains b nodes at each of the m depths of the frontier
Breadth-First Search (BFS)  } s tiers	Queue; FIFO	yes; if ∃ soln, the depth s of shallowest node must be finite. BFS will search this depth	No, bc it doesn't take costs into consideration when determining node to replace on frontier. Only guaranteed optimal if ∀ costs = 1, which reduces to UCS.	$O(b^s)$	$O(b^s)$
UCS (Uniform Cost Search) selects lowest cost frontier node for expansion	Heap-based priority queue. Priority for node v is the backwards cost (path from start node to v). · orders by path cost (forward cost)	yes. same reason as BFS. Note: despite the fact that UCS is complete & optimal, it's slow bc it explores in all directions, (explores increasing cost contours)	yes if all edge costs are non-neg.	$O(b^{C^*/\varepsilon})$ $\frac{C^*}{\varepsilon}$ tiers  ≤1 ≤2 ≤3	$O(b^{C^*/\varepsilon})$
Iterative deepening: 'merges' DFS's space advantage w/ BFS's time advantage.	Stack	yes (same as BFS)	No. same as BFS.	$O(b^s)$	$O(bs)$

· graph DFS guaranteed to return a ~~best~~ path passing thru all possible grid locations if one exists ↳ not tree DFS

↳ same w/ BFS (both graph & tree) traverse whole ~~1st~~

Informed Search Summary

Search	Heuristic	Complete	Optimal	
Greedy: selects lowest heuristic value for expansion	estimate distance to nearest goal for each state	No	No, esp w/ bad heuristic	orders by goal proximity (backwards cost)
A*: selects lowest total cost	$f(n) = h(n) + g(n)$	yes	Graph ↳ yes if heuristic is consistent ↳ UCS optimal (h=0 consistent); Tree ↳ yes if heuristic is admissable ↳ UCS is special case (h=0)	

· can prove entailment using 3 rules of inference

① Modus ponens: if our KB contains A & A ⇒ B, we can infer B

② And-elimination: if our KB contains (A ∧ B) we can infer A and can infer B

③ Resolution: if our KB contains A & B we can infer (A ∧ B)

· entailment algorithms

① Model Checking

N symbols → 2^N models to check

⇒ $O(2^N)$ in propositional logic

⇒ $O(\infty)$ in first order logic

② DPLL SAT Solver

· recursive depth 1st enumeration of models w/ some extra features:

1) early termination - stop if all clauses are satisfied or

· $(A \vee B) \wedge (A \vee \neg C)$ is satisfied by A = True

· any clause is falsified

· $(A \vee B) \wedge (A \vee \neg C)$ is falsified by {A = F, B = F}

2) pure literals - if all occurrences of a symbol in as-of-yet unsatisfied clauses have the same sign, give the symbol that value

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

↳ A is pure & (+), so set it to T.

3) unit clauses - if a clause left w/ a single literal, set symbol to satisfy clause.

if A = F in $(A \vee B) \wedge (A \vee \neg C)$ we have $B \wedge \neg C$, so set C = F, B = T

↳ satisfying these can lead to further propogation/new unit clauses

③ Forward Chaining

↳ useful for special case when our KB only has literals & implications, bc then we can prove entailment in O(N) time (N = size of KB).

· FC iterates through every statement where the premise (left side) is known to be true, adding it to the conclusion

④ Backward Chaining

Local Search

Search	Algorithm	Complete	Optimal
Hill Climbing / Steepest Ascent ~~mutated~~	· start wherever. ↳ loop: move to best neighbor state. ↳ stop if no neighbors better than curren	w/o random restarts? No. w/ random restarts? yes	
Simulated Annealing ↳ combines random walk w/ hill climbing	· ↑T ⇒ allow more "bad" moves · gradually reduce T		
Local Beam Search ~~↳ K iterations of~~	· create K iterations of local search algo, random initialized · for each iteration, gen all successors & choose best K to survive.		
Genetic Algorithm	· resample K individuals at each step (selection) weighted by fitness fcn. · combine by pairwise crossover operators, plus mutation (for variety)		

Reinforcement Learning

· offline planning - agents have full knowledge of transition & reward fcns (eg, solving an MDP).

· online planning - no knowledge of T/R so have to try exploration (receiving feedback for actions it performs) & uses RL to estimate a policy for exploitation (reward maximization)

· 2 types of RL:

① model-based learning - attempts to estimate transition & reward fcns, then uses those estimates to solve the MDP

② model-free learning: attempts to estimate Q-values of states directly

First-Order Logic (FOL)

· FOL uses objects as its basic components

· each object represented as a constant symbol, each relationship by a predicate symbol, & each fcn by a fcn symbol.

· terms - ~~refer to~~ logical exps that refer to an obj.

↳ simplest = constant symbols

· atomic sentences: relationships btwn objects

Brother(John, Richard)

· quantifiers ∀, ∃:

$\forall x \neg P \equiv \neg \exists x P$ $\qquad \neg(P \vee Q) \equiv \neg P \wedge \neg Q$

$\neg \forall x P \equiv \exists x \neg P$ $\qquad \neg(P \wedge Q) \equiv \neg P \vee \neg Q$

$\forall x P \equiv \neg \exists x \neg P$ $\qquad P \wedge Q \equiv \neg(\neg P \vee \neg Q)$

$\exists x P \equiv \neg \forall x \neg P$ $\qquad P \vee Q \equiv \neg(\neg P \wedge \neg Q)$

· formulating inference using FOL (i.e. KB ⊨ q)

① Propositionalization: translate the problem from FOL to PL so we can use our other PL methods on it

② Lifted inference: directly do inference with FOL

Model-Based Learning

$T(s,a,s') = \dfrac{\#\text{ times } s,a \to s'}{\#\text{ times we did } s,a}$

$R(s,a,s') = \dfrac{\sum \text{rewards from } s,a \to s'}{\#\text{ times we did } s,a \to s'}$

$Q_{k+1}(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_k(s')]$

↳ $V_k(s') = \max_a Q_k(s',a)$

Direct Evaluation

$\hat{V}^\pi(s) = \dfrac{\sum_{\text{episodes}} \text{utility starting from state } s}{\#\text{ episodes } s \text{ appeared in}}$

Approximate Q-learning

$diff = [R(s,a,s') + \gamma \max_{a'} Q(s',a')] - Q(s,a)$

$w_i \leftarrow w_i + \alpha \cdot diff \cdot f_i$

Temporal Difference Learning

uses exponential moving average

↳ sample $= R(s, \pi(s), s') + \gamma V^\pi(s')$

update:

$V^\pi(s) = (1-\alpha) V^\pi(s) + \alpha \cdot sample$

... learning rate

Q-learning

Q-value samples:

sample $= R(s,a,s') + \gamma \max_{a'} Q(s',a')$

$Q(s,a) \leftarrow (1-\alpha) Q(s,a) + \alpha \cdot sample$

Approx. Q-learning

· allows us to generalize learning experiences to many similar situations by using feature-based representation of the states (repr. each state as a feature vector).

1) Init all weights to 0 ($\vec{w}=0$)

2) For each training sample w/features $f(x)$ & true class label $y^* \in \{-1,+1\}$, do:

i) classify sample using current weights. Let $\vec{y}$ be the class predicted by curr $\vec{w}$:

$$y = \text{classify}(x) = \begin{cases} +1 & \text{if } h_w(x) = w^T f(x) > 0 \\ -1 & \text{o/w} \end{cases}$$

ii) Compare y to y^*

a) if $y = y^*$: do nothing

b) if $y \neq y^*$:

$\vec{w} \leftarrow \vec{w} + y^* f(x)$

Note:
Case 1: Misclassified as $(-)$
$w \leftarrow w + f(x)$
Case 2: Misclassified as $(+)$
$w \leftarrow w - f(x)$

If you went through every training sample w/o having to update your weights, then terminate. o/w repeat ②.

Multiclass Perceptron Algorithm

$\vec{w} = \vec{0}$

For each training sample

i) Predict ~~[illegible]~~ w/current weights:

$y = \operatorname{argmax}_y w_y f(x)$

ii) Compare y to y^*:

a) if $y = y^*$: do nothing

b) if $y \neq y^*$: lower score of wrong answer & raise score of correct answer:

$\omega_y = \omega_y - f(x)$

$\omega_{y^*} = \omega_{y^*} + f(x)$

Perceptron Properties

Seperability: true if some parameters get the training set perfectly correct

Convergence: if training is seperable, perceptron will eventually converge (binary case)

Mistake bound: max # of mistakes (binary case) related to margin/degree of seperability: mistakes $< K/\delta^2$

If a perceptron misclassifies data, it might still misclassify after the weight update

Perceptron Probs

Noise: if data not seperable, weights might thrash ⇒ avg weight vectors over time (avg'd perceptron) can help

Mediocre generalization: finds "barely" seperating soln

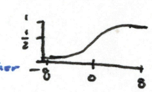
Overtraining: test/held-out accuracy usually rises then falls

Probabilistic Perceptron Decisions

(instead of deterministic)

If $z = w \cdot f(x)$ (output of perceptron) very positive → prob of class +1 should approach 1 & vice versa

Sigmoid Fcn: $\phi(z) = \dfrac{1}{1+e^{-z}}$ — probability that the class is +1, according to the classifier



Softmax Fcn:

$\sigma(z)_i = (e^{z_i}) / (\sum_{j=1}^{K} e^{z_j})$

Hyperbolic tangent:

$g(z) = \dfrac{e^z - e^{-z}}{e^z + e^{-z}}$

ReLU: $\max(0,x)$

rectified linear ~~[illegible]~~ unit

Optimization

Gradient Ascent:

randomly initialize $\vec{\omega}$
while $\vec{\omega}$ not converged:
$\vec{\omega} = \vec{\omega} + \alpha \nabla_\omega f(\omega)$
end

α: learning rate

Gradient descent: $\vec{\omega} = \vec{\omega} - \alpha \nabla_\omega f(\omega)$

Stochastic gradient descent: only use 1 data point per iteration to compute ∇. ~~[illegible]~~ data point randomly sampled from data

Mini-batch gradient descent: uses a batch size of m data pts. at each iteration to compute ∇ ~~[illegible]~~

e.g, for least-sq./linear regression

$\text{Loss}(h_w) = \frac{1}{2} \| \vec{y} - X\vec{w} \|_2^2$

$\nabla_w \text{Loss} = -X^T y + X^T X \vec{w}$

$\vec{w} \leftarrow \vec{w} - \alpha(-X^T \vec{y} + X^T X \vec{w})$ (for LS-gradient descent)

· allows us to turn linear combination of input features into a probability using logistic fcn:

$h_w(\vec{x}) = \dfrac{1}{1+e^{-w^T x}}$

· single layer NN w/softmax activation is the same as logistic regression

Neural Networks

· many probs require non-linear solns

· NN's: multi-layer perceptrons that are more expressive

· Universal Fcn Approx. Thm: 2-layer NN w/sufficient # of neurons can approximate any continuous fcn

Accuracy

· accuracy of binary perceptron after making n predictions:

$$l^{acc}(\vec{w}) = \frac{1}{n} \sum_{i=1}^{n} (\text{sgn}(w \cdot f(x_i)) == y_i)$$

indicator fcn; weight vector; fcn that derives feature vector; data point i; actual class label of x_i

· softmax fcn defines probability of classifying $x^{(i)}$ to class j as:

$$\sigma(x_i)_j = \frac{e^{f(x_i)^T w_j}}{\sum_{l=1}^{n} e^{f(x_i)^T w_l}} = \mathbb{P}[y_i = j \mid f(x_i); w)$$

· likelihood of a set of weights:

$$l(\vec{\omega}) = \prod_{i=1}^{n} P(y_i \mid f(x_i); w)$$

· log-likelihood (want to maximize this quantity by finding $\vec{w}^*$):

$$\log l(\vec{\omega}) = \log \prod_{i=1}^{n} P(y_i \mid f(x_i); w) = \sum_{i=1}^{n} \log P(y_i \mid f(x_i); w)$$

↳ maximize by computing the ∇.

$$\nabla_w l(w) = \left[\frac{\partial ll(w)}{\partial w_1} \ \cdots \ \frac{\partial ll(w)}{\partial w_n} \right]$$

Backpropogation

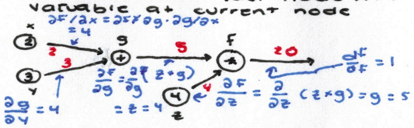
· allows you to efficiently compute gradients for each param. in the NN.

· represents NN as a dependency graph of operators & operands (computational graph)

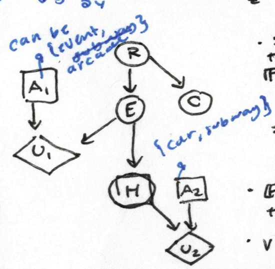
Backpropogation Algorithm:

1) Forward pass: compute output of node

2) Backwards pass: partial derivative of last node wrt variable at current node



$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial y} = 4$



· say we eliminated R & C to find $\mathbb{P}[E=\text{true}]$. Find $\mathbb{P}[E=\text{true} \mid H=\text{true}]$

$$\mathbb{P}[E=\text{true} \mid H=\text{true}] = \frac{P[H=\text{true} \mid E=\text{true}]\,\mathbb{P}[E=\text{true}]}{P[H=\text{true}]}$$

$$= \frac{\mathbb{P}[H=\text{true} \mid E=\text{true}]\,\mathbb{P}[E=\text{true}]}{\mathbb{P}[H=\text{true} \mid E=\text{true}]\,\mathbb{P}[E=\text{true}] + \mathbb{P}[H=\text{true} \mid E=\text{False}]\,\mathbb{P}[E=\text{False}]}$$

· $EU[\text{event, subway} \mid H=\text{true}] = \mathbb{P}[E=\text{true} \mid H=\text{true}]\,U_1(\text{event}, E=\text{true}) + \mathbb{P}[E=\text{False} \mid H=\text{true}]\,U_1(\text{event}, E=\text{False}) + U_2(\text{subway}, H=\text{true})$

· $VPI(H) = P[H]\,MEU(H) + \mathbb{P}[\neg H]\,MEU[\neg H] - MEU[\emptyset]$

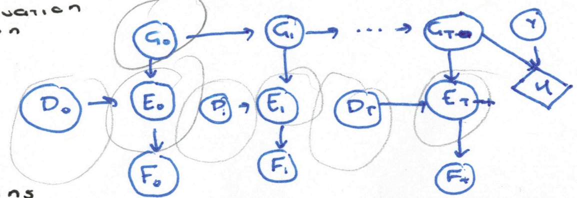
· $MEU(R=\text{true}) = \max_{a_1,a_2} \sum_e P(e \mid r)[U_1(e,a_1) + \sum_h P(h \mid e)\,U_2(h,a_2)]$

· what are all possible individual variables X s.t. we can decompose $MEU(X=x) = MEU_1(X=x) + MEU_2(X=x)$? R, C, E, H (by linearity of E)

· Let X be ~~any~~ a boolean variable in any decision network. The following conditions are jointly sufficient to show that $VPI(X) > 0$: $P(x)$ is not

---

· if we discard all ... independence assumptions of the naive Bayes model, we need $2^{\#\text{variables}} - 1$ parameters



· M×N grid w/o walls/obstacles

· Pacman trying to figure out location of ghosts at timestep T (G_T)

· 3 sensor variables: D_t, E_t, F_t

· guess Y is pacman's guess of G_T

· Utility $U(Y, G_T)$ predetermined fcn that's larger when Y closer to G_T

(a) Pacman doesn't observe sensor values & doesn't know any of the CPTs in the DBN. $U(y, g_T) = \begin{cases} 100 & y = g_T \\ -1 & \text{o/w} \end{cases}$. uniform distribution & ghost moves randomly for a long time. What's MEU?

$$MEU(\emptyset) = \mathbb{P}[Y=G_T]\,U[Y=G_T] + \mathbb{P}[Y \neq G_T]\,U[Y \neq G_T]$$

$$= \frac{1}{MN} \cdot 100 + \frac{MN-1}{MN} \cdot (-1)$$

(b) Assume Pacman knows ~~everything~~ all conditional distributions in the DBN. What maximizes his expected utility when no evidence is available?

$$Y = \operatorname{argmax}_{g_T} \sum_{g_{T-1}} \mathbb{P}(g_T \mid g_{T-1})\,\mathbb{P}(g_{T-1})$$

(c) Pacman can choose to uncover a sensor value for all t to help max utility.

(i) Which of the following is the strongest valid statement about VPIs for sensor variables?

$VPI(E) \geq VPI(F) \geq VPI(D)$

(ii) Pacman chooses F. What maximizes $\mathbb{E}[\text{utility}]$?

$$Y = \operatorname{argmax}_{g_T} \sum_{e_T, d_T} P(f_T \mid e_T) P(d_T) P(e_T \mid d_T, g_T) P(g_T \mid f_{0:T-1})$$

(iii) Pacman chooses E. Using fcns for the belief distribution & general utility fcn, what's the expression for $EU(Y \mid e_{0:T})$?

$$EU(Y \mid e_{0:T}) = \sum_{g_T} B(G_T = g_T)\,U(Y, g_T)$$

(iv) Minimum set of variables to maximize expected utility: $G_0, E_{1:T}, D_{1:T}$ (Markov blanket of all ghost variables → would contribute a positive VPI)

... are linear classifiers & can represent any fcn, incl. nonlinear ones)

Loss

· want to predict Q-values as close to $q((x,y),a)$ as possible

1) MSE loss — works bc it defines squared difference btwn the 2 values we want to be similar

2) Indicator — (zero-one loss) won't work bc it doesn't give reasonable metric for how close our estimates are & doesn't yield a gradient

3) Cross-entropy loss isn't suitable since we aren't doing classification

CSPs (Constraint Satisfaction ...)

- type of identification problem where we have to see if a state is a goal state or not
- defined by 3 factors:
  1) Variables: $(X_1, \ldots, X_N)$ that can take on a single value from a defined set of values
  2) Domain: set $\{x_1, \ldots, x_d\}$ representing all possible values a variable can take on
  3) Constraints: define restrictions on values of variables
- CSPs are NP-Hard
- given problem w/ N vars of domain size $O(d)$, for each var, there are $O(d^N)$ possible assignments
- types of constraints
  1) Unary - involve a single variable
     ↳ not repr. in constraint graphs, just used to trim domain
  2) Binary - involve 2 variables
  3) Higher-order - involve ≥2 variables

- every discrete, finite CSP can be represented as a SAT problem & vice versa, & a correct repr. of a discrete, finite CSP has exactly same # satisfying args.

Solving CSPs

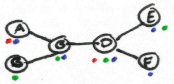
- Backtracking search
  → optimization on DFS, where it also:
  1) fixes an ordering for variables & selects values for variables in that order
  2) only selects values that don't conflict w/ any prev values. If no values exist, backtrack to prev variable & change its value.
- Filtering: can we detect inevitable failure early? i.e. keep track of domains for unassigned variables and cross off bad options
  - Forward checking: whenever a value is assigned to X_i, prune the domains of the unassigned variables that share a constraint w/ X_i that would violate that constraint
  - arc consistency: an arc $X \to Y$ is consistent iff ~~...~~ $\forall x$ in the tail, $\exists y$ in the head that could be assigned w/o violating a constraint
    ↳ limitations: ~~...~~ after enforcing AC, can leave 0, 1, or >1 solns left
    + still runs in backtracking search

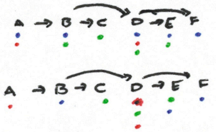
Orderings for CSPs

1) MRV: Minimum Remaining Values
   ↳ choose most constrained variable next
2) LCV: Least constraining value
   ↳ ~~...~~ choose the value that prunes the fewest domains of remaining unassigned variables

CSP Structure

- has $O(n|d|^n)$ leaves in search tree
- general runtime is $O(d^N)$ but can be reduced to $O(d^2n)$ (linear in # of ~~...~~ variables) by doing the following:
  1) Pick arbitrary node in the constraint graph to serve as a root node
  2) Convert the undirected graph edges to edges that point away from the root. Linearize/topologically sort the graph s.t. all edges point rightwards
  3) Perform a backwards pass of arc consistency from $i = N$ to $i = 2$ for all arcs $Parent(X_i) \to X_i$
  4) Perform a forwards assignment, assigning each X_i a value consistent





Cutset Conditioning

- find the smallest subset of variables in a graph s.t. their removal results in a tree (a cutset for the graph)
  ↳ leaves us w/ tree w (n-c) variables
    ⇒ solvable in $O((n-c)d^2)$
    ⇒ runtime of cutset conditioning on a general CSP is $O(d^c(n-c)d^2)$
    ↳ may still need to backtrack d^c times.

K-Consistency

- 1-consistency (node consistency): each single node's domain has a value that meets its unary constraints
- 2-consistency (arc consistency): for each pair of nodes, any consistent assignment can be extended to another
- K-consistency: for each K nodes, any consistent assignment to K-1 can be extended to the K^{th} node

MDPs (Markov Decision Processes)

- models used to solve nondeterministic search problems
- some properties:
  - γ: discount factor
  - $T(s,a,s')$: transition function - probability that taking action "a" from state "s" results in "s'"
    ↳ $P(s' \mid s, a)$
  - $R(s,a,s')$: Reward function -
  - $Q(s,a)$: Action states
- finite horizons - defines lifetime for agents before they get terminated
- discount factors - model exponential decay of rewards over time, so instead of maximizing additive utility:
  $U([s_0, a_0, s_1, a_1, \ldots]) = R(s_0, a_0, s_1) + \cdots + R(s_n, a_n, s_{n+1})$
  we maximize discounted utility:
  $U([s_0, a_0, s_1, a_1, \ldots]) = R(s_0, a_0, s_1) + \gamma R(s_1, a_1, s_2) + \gamma^2 R + \cdots$
  $= \sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1})$
  guaranteed to be finite valued as long as $|\gamma| < 1$:
  $\sum_{t=0}^{\infty} \gamma^t R_{max} = \frac{R_{max}}{1-\gamma}$
- Markovian/memorylessness encoded in transition function: $T(s,a,s') = P(s' \mid a, s)$

Solving MDPs

- want to find optimal policy $\pi^*: S \to A$, a function mapping each state $s \in S$ to action $a \in A$. An explicit policy $\pi(s)$ defines a ~~...~~ reflex agent; ~~...~~ given a state s, an agent implementing π will select $a = \pi(s)$.
- $U^*(s)$ or $V^*(s)$: optimal value of a state s; expected value of the utility of an optimally-behaving agent that starts in s will receive.
- $Q^*(s,a)$: optimal value of a Q-state

Bellman Equations:

$$U^*(s) = \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma U^*(s')]$$

$$Q^*(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma U^*(s')]$$

$$U^*(s) = \max_a Q^*(s,a)$$

Value Iteration

- time-limited value for a state s with a time-limit of K timesteps ($U_K(s)$) represents the maximum expected utility attainable from s given that the MDP terminates in K timesteps. (eg. depth-K expectimax)
- ~~...~~ idea: VI is a DP algo that uses an iteratively longer time to compute time-limited values until convergence
- algorithm:
  ① Initialize, $\forall s \in S \quad U_0(s) = 0$
  ② Repeat update rule until convergence: $\forall s \in S$
  $U_{K+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma U_K(s')]$

- $O(|S||A|)$

Policy Extraction

- if you're in a state s, you should take the action a which yields the max expected utility
  $\forall s \in S \quad \pi^*(s) = \arg\max_a Q^*(s,a)$
  $\pi^*(s) = \arg\max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma U^*(s')]$

Q-Value Iteration

$$Q_{K+1}(s,a) \leftarrow \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma \max_{a'} Q_K(s',a')]$$

Policy Iteration

- algorithm:
  ① Define an initial policy. Can be arbitrary but converges faster the closer the initial policy is to optimal policy.
  ② Repeat the following until convergence:
  1) Policy Evaluation: compute $U^\pi(s)$ $\forall s \in S$, until convergence
     $U^\pi(s) = \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma U^\pi(s')]$
     $U^{\pi_i}_{K+1}(s) \leftarrow \sum_{s'} T(s, \pi_i(s), s')[R(s, \pi_i(s), s') + \gamma U^{\pi_i}_K(s')]$
  2) Policy improvement: for fixed values, get a better policy using policy extraction:
     $\pi_{i+1}(s) = \arg\max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma U^{\pi_i}(s')]$
     ↳ if $\pi_{i+1} = \pi_i$, we have converged ⇒ $\pi_i = \pi^*$

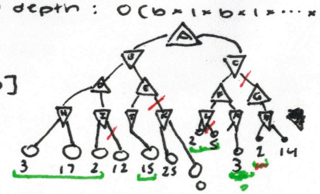
MDP Recap

- Value iteration: used to compute optimal values of states by iterative updates until convergence
- Policy evaluation: compute values for a particular policy
- Policy extraction: turn your values into a policy
- Policy iteration: compute optimal values for a particular policy

Alpha-Beta Pruning

branching factor b, depth d

- Minimax → $O(b \times b \times \cdots \times b) = O(b^d)$
  ↳ also worst case α-β
- best case:
  → odd depth: $O(b \times 1 \times b \times 1 \times \cdots \times b)$
  → even depth: $O(b \times 1 \times b \times 1 \times \cdots \times b \times 1) = O(b^{d/2}) = O(\sqrt{b^d})$



Approx. Q-learning

Linear value fcns:

$V(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + f_n(s) w_n = \vec{w} \cdot \vec{f}(s)$

$Q(s,a) = w_1 f_1(s,a) + \cdots + w_n f_n(s,a) = \vec{w} \cdot \vec{f}(s,a)$

$difference = [R(s,a,s') + \gamma \max_{a'} Q(s',a')] - Q(s,a)$

$w_i \leftarrow w_i + \alpha \cdot difference \cdot f_i(s,a)$

exact Q-learning:

$Q(s,a) = Q(s,a) + \alpha \, difference$

ε-greedy

- explore randomly w.p. ε
- exploit w.p. (1-ε)
- exploration fcns:
  $Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha[R(s,a,s') + \gamma \max_{a'} f(s',a')]$
  where f is an exploration fcn.
- common choice for $f(s,a) = Q(s,a) + \frac{K}{N(s,a)}$
  K: predetermined value
  $N(s,a)$ = # times Q-state (s,a) was visited