UNM CS 427 AI Notes

by Luke Hanks

Search Problems (2-5)

Search problems involve no adversaries and no uncertainty. They rely on the accuracy of the model.

World state contains *everything* about the world. **Search state** contains *only* that which is needed to solve the search problem.

A **search graph** has all search states as nodes and transition actions as directed edges between them. A **search tree** has the start state as the root. Every node has all of its successors(neighbors in the search graph) as its children in the search tree. In addition to the search state nodes also encapsulate the plan (list of actions) that lead there. The number of nodes in a search tree $= 1 + b + b^2 + ... + b^m = O(b^m)$ where b is branching factor and m is max depth. If the search graph has cycles, then states repeat in the search tree and m is infinite.

General Alg: Initially the root is the only node in the fringe. Take a node out of the fringe. Test if its state satisfies the goal. If it does, then return its plan. If it does not, then add it to the closed set (set of expanded nodes) and expand it by putting its successor nodes in the fringe (except those that are in the closed set). Repeat. Variations (DFS, BFS, A*, ect.) are defined in how states are picked out of the fringe.

Depth First Search (DFS): Fringe is a LIFO queue. Takes time $O(b^m)$. Fringe only has siblings on path to root, so fringe size is $O(bm)$. Remember that m is often infinite. DFS doesn't necessarily find the optimal solution and may take forever. Fix this with iterative deepening.

Breadth First Search (BFS): Fringe is a FIFO queue. Takes time $O(b^s)$ where s is the shallowest solution. The fringe size is also $O(b^s)$.

Uniform Cost Search (UCS): BFS, but considers the cost of state transitions instead of just length of plan. For a fringe UCS uses a priority queue prioritizing lowest total cost of plans.

Greedy Search: UCS, but prioritizes with a heuristic (estimate of how close a state is to a goal state) instead of the total cost of the plan. Heuristics vary based on the search problem.

A* Search: Uses the sum of the total cost of the plan (backward cost $g(n)$) and the heuristic (forward cost $h(n)$). Let $h^*(n)$ be the true cost to a nearest goal node. **Permissible heuristics** are optimistic in that $h(n) \leq h^*(n)$. The closer $h(n)$ is to $h^*(n)$ the fewer nodes will need to be expanded. $h(n)$ should not be too expensive to compute. Good heuristics can often be found by solving a relaxed version of the search problem. **Consistent heuristics** don't drop between states more than the true cost between those states in that $\text{cost}(A \text{ to } C) \geq h(A) - h(C)$. Consistency is only needed when there are cycles in the search graph.

Constraint Satisfaction Problems (CSPs)

There is a set of variables that need values assigned to them from their domains (sets of possible values) in such a way that they don't violate a set of constraints.

Backtracking Search: Assign values to a variable until you find an assignment that doesn't immediately violate a constraint. Then recurs onto the next variable. Once you've gone trough all possible assignments for this variable, return failure.

Forward checking is done by keeping track of each unassigned variable's domain and filtering it down with each assignment so that nothing in its domain would conflict with the current assignment. If a domain becomes empty, then the current assignment is invalid and move on to the next one.

TODO: Learn arc consistency and k-consistency

Assignment ordering affects runtime. Usually best to prioritize variables with smaller domains (minimum remaining values). Usually best to prioritize values that have the smallest effect on the domains other variables (least constraining value).

It helps a lot if you can find independent sub-problems.

If the CSP has a tree structure then it can be solved in linear time. TODO: Find out why tree CSPs are easy.

Iterative Algs for CSPs start with a quickly generated (usually random) set of assignments that don't necessarily satisfy the constraints and then tries to improve the assignments until they do satisfy the constraints.

Adversarial Search (6-7)

Limit ourselves to turn based zero sum games with only two players and mutual perfect information.

Minimax Search

Complexity like DFS. Time is $O(b^m)$. Space is $O(bm)$. b and m are too large in most games. Limit the depth of search (m). When you reach the limit evaluate the utility of that game sate. Not perfect decisions, but more practical complexity. We can also prune the search tree. Keep track of values α and β.

Alpha-Beta Pruning

- $\alpha =$ best already explored option along path to the root for maximizer
- $\beta =$ best already explored option along path to the root for minimizer
- Every time a node is expanded it will get its initial α and β from it's parent. The root starts with $\alpha = -\infty$ and $\beta = +\infty$.
- When expanding a max node, we adjust its α up to the values of the children. If we come across a child with value $\geq \beta$, then we set this max node to that value. Otherwise, set this max node to the highest value of its children.
- When expanding a min node, we adjust its β down to the values of the children. If we come across a child with value $\leq \alpha$, then we set this min node to that value. Otherwise, set this min node to the lowest value of its children.

Expectimax Search

Minimax, but were the min nodes are expected utility nodes. Alpha-beta pruning doesn't apply to expecti nodes.

$$\text{Expected Utility} = \sum_{\text{outcomes}} P(\text{outcome}) \cdot \text{Utility}(\text{outcome})$$

Axioms of rationality

> A rational agent should choose the action that maximizes its expected utility, given its knowledge.

^ This! The main reason I'm interested in AI is because I want to be more rational myself. AI is in part a study of rationality.

Lottery notation: $[P_1, U_1; P_2, U_2; ...P_n, U_n;]$ such that $\sum_{i=1}^{n} P_n = 1$

Axioms:

- Orderability: $(A \succ B) \vee (B \succ A) \vee (A \sim B)$
- Transitivity: $(A \succ B) \wedge (B \succ C) \Rightarrow (A \succ C)$
- Continuity: $A \succ B \succ C \Rightarrow \exists p [p, A; 1 - p, C] \sim B$
- Substitutability: $A \sim B \Rightarrow [p, A; 1 - p, C] \sim [p, B; 1 - p, C]$
- Monotonicity: $A \succ B \Rightarrow (p \geq q \Leftrightarrow [p, A; 1 - p, B] \succ [q, A; 1 - q, B])$

Markov Decision Processes (8-11)

Markov indicates that past and future states are independent given the current state.

Like a search problem, but searching for a policy instead of a plan. A policy is a mapping of states to actions, $\pi : S \rightarrow A$. This is good for potentially never ending games. An explicit policy makes a reflex agent.

- Set of states $s \in S$
  - Start state s_0
  - Terminal state (sometimes)
- Set of actions $a \in A$
- q-states (s, a)
  - Intermediary state when the agent has committed to an action a from s, but the new state s' is still uncertain.
- Transitions (s, a, s')
- Transition function $T(s, a, s') = P(s' \mid s, a)$
  - Prob that a from s leads to s'
  - AKA model or dynamics
- Reward function $R(s, a, s')$
- Discount rewards over time by multiplying by γ^t where t is the number of time steps ahead.
  - With rewards and penalties sooner is worth more consideration than later, all else equal.
  - Discounting helps the policy converge.
- $V^*(s) =$ expected utility starting in s and acting optimally, aka value
  - $V^*(s) = \max_a Q^*(s, a)$
  - Substitute the definition of $Q^*(s, a)$ to get the Bellmen Equation.
  - $V_k(s) =$ optimal value of s if the game ends in k more time steps

- $Q^*(s, a)$ = expected utility starting out having taken action a from s and thereafter acting optimally, aka q-value
  - $Q^*(s, a) = \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V^*(s')$
- $\pi^*(s)$ = optimal action from state s (π^* is what is searched for)
- Value Iteration
  - $V_0(s) = 0$
  - $V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V_k(s')]$
  - Complexity of each iteration is $O(|S|^2 \cdot |A|)$
  - $\lim_{k \to \infty} V_k(s) = V^*(s)$
  - Policies converge long before values do which means value iteration tends to over do it.
- Policy Evaluation
  - When we have a fixed policy we can calculate values (under that policy) fast.
  - $V^\pi(s)$ = expected total discounted utility starting in s and following π
    - $V^\pi(s) = \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V^\pi(s')]$
  - Can be computed with simplified value iteration, $V_k^\pi(s)$
    - Complexity $O(|S|^2)$
    - The computation can be done by a linear system solver.
- Policy Extraction
  - Given a mapping of states to values $V^*(s)$, find the appropriate policy.
    - $\pi^*(s) = \arg\max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V^*(s')]$
  - Given a mapping of state-action pairs to q-values $Q^*(s, a)$, find the appropriate policy.
    - $\pi^*(s) = \arg\max_a Q^*(s, a)$
    - This is trivial.
- Policy Iteration
  - Start with a random policy π_0. Perform policy evaluation on it. From the values of that evaluation extract a new policy π_{i+1}. Repeat until the policy converges (which is guaranteed).
  - $V_0^{\pi_i}(s) = 0$
  - $V_{k+1}^{\pi_i}(s) \leftarrow \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V_k^{\pi_i}(s')]$
  - $\pi_{i+1}(s) = \arg\max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V^{\pi_i}(s')]$
  - Usually faster than value iteration.
  - This is great when there are many actions and/or the maximizing actions rarely change during value iteration rounds.

Reinforcement Learning (9-11)

Offline planning is using a sufficient model to build (in your head) plans or policies before actually acting (everything we've done previously).

Online learning is taking actions to learn what kinds of actions maximize utility and from that building a policy.

We no longer know $T(s, a)$ or $R(s, a, s')$.

- Exploration: you have to try unknown actions to get information
- Exploitation: eventually, you have to use what you know
- Regret: even if you learn intelligently, you make mistakes
- Sampling: because of chance, you have to try things repeatedly

Model-Based Learning

1. Build MDP model based on experience.
    - Count outcomes s' for each (s, a)
    - Normalize to give an estimate of $\hat{T}(s, a, s')$
    - Discover each $\hat{R}(s, a, s')$ when we experience (s, a, s')
2. Build a policy based on the learned MDP model (w/ something like value iteration).

Model-Free Learning

TODO: Figure out how to describe Model-Free Learning.

- **Passive Reinforcement Learning**
    - Get a fixed policy. Execute it and learn state values on the way via direct evaluation.
    - **Direct evaluation**: Follow π. Every time you visit a state record what the sum of discounted rewards turned out to be. Average those samples.
        - Problem is values are learned in isolation.
    - **Sample-Based Policy Evaluation**: Take samples of outcomes s' (by doing the action) and average
        - $sample_n = R(s, \pi(s), s'_n) + \gamma V^\pi_k(s'_n)$
        - $V^\pi_{k+1}(s) \to \frac{1}{n} \sum_i sample_i$
        - Problem is we can't keep returning to the same state to take the same action over and over.
- **Temporal Difference Learning**
    - Update $V(s)$ each time we experience a transition (s, a, s', r)
    - Likely outcomes s' will contribute updates more often.
    - Learning values: Evaluate fixed policy with running average.
        - $sample = R(s, \pi(s), s') + \gamma V^\pi(s')$
        - $V^\pi(s) \leftarrow (1 - \alpha) V^\pi(s) + \alpha(sample) = V^\pi(s) + \alpha(sample - V^\pi(s))$

        - α is the learning rate.
        - Problems is we don't know how to iterate to a better policy π'
- **Active Reinforcement Learning** (off-policy learning, Q-Learning)
    - Q-Value Iteration
        - $Q_0(s) = 0$
        - $Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma \max_{a'} Q_k(s',a')]$
        - $\lim_{k \to \infty} Q_k(s) = Q^*(s)$
        - Problem is we don't know T and R
    - Sample-based Q-value iteration
        - experience (s, a, r, s')
        - $sample = r + \gamma \max_{a'} Q(s', a')$
        - $Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(sample) = Q(s,a) + \alpha(sample - Q(s,a))$
    - How you choose your actions
        - With a probability of ϵ deviate from the current policy and instead take a random action.
        - Alternatively use some exploration function f
            - say $f(u,n) = u + k/n$
            - $sample = r + \gamma \max_{a'} f(Q(s',a'), N(s',a'))$
    - Regret is the difference between total utility while learning and the total utility that would have been gained if you had been following the optimal policy.
- **Approximate Q-Learning**
    - Too many states to learn about them individually.
    - Learn about q-state features instead of the q-states themselves.
    - Boil q-states down to a feature vector $\vec{f}(s,a)$.
    - Multiply the feature vector with a weight vector $\vec{w}$ to approximate the q-value.
    - $Q(s,a) = \sum_{i=1}^{n} w_i f_i(s,a) = \vec{w} \cdot \vec{f}(s,a)$
    - experience (s, a, r, s')
    - $\text{difference} = [r + \gamma \max_{a'} Q(s',a')] - Q(s,a)$
    - Exact Q update: $Q(s,a) \leftarrow Q(s,a) + \alpha[\text{difference}]$
    - approximate Q update: $w_i \leftarrow w_i + \alpha[\text{difference}] f_i(s,a)$

Probability Review (12-13)

- $P(A \cap B) = P(A)P(B \mid A) = P(B)P(A \mid B)$
- If A and B are disjoint, then $P(A \cap B) = 0$
- $P(A \cup B) = P(A) + P(B) - P(A \cap B)$

- A joint distribution of n variables with domain sizes d will have d^n rows.
- Marginal Distribution: sum rows over some variable(s) to eliminate the variables from the joint distribution.
- $P(a \mid b) = \frac{P(a,b)}{P(b)}$
- Inference: Select rows consistent with all of the evidence $e_1 ... e_k$ from the joint distribution $P(Q, H_1 ... H_r, E_1 ... E_k)$ and from those sum over the hidden variables $H_1 ... H_r$. What's left is $P(Q \mid e_1 ... e_k)$.
- **Product** Rule: $P(y)P(x \mid y) = P(x, y) \Leftrightarrow P(x \mid y) = \frac{P(x,y)}{P(y)}$
  - Marginal * Conditional = Joint
- **Chain** Rule: $P(A, B) = P(A \mid B)P(B)$
  - $P(x_1, x_2, ...x_n) = \prod_{i=1}^{n} P(x_i \mid x_1, ...x_{i-1})$
  - $n!$ different ways to apply the chain rule to a joint distribution of n variables because you can go through the variables in any order.
- **Bayes'** Rule: $P(A \mid B) = \frac{P(B|A)P(A)}{P(B)}$
  - $P(A \mid B) \propto_A P(B \mid A)P(A)$
- **Independence**: $X \perp Y$
  - $\Leftrightarrow \forall x, y : P(x, y) = P(x)P(y)$
  - $\Leftrightarrow \forall x, y : P(x \mid y) = P(x)$
- **Conditional independence**: $X \perp Y \mid Z$
  - $\Leftrightarrow \forall x, y, z : P(x, y|z) = P(x \mid z)P(y \mid z)$
  - $\Leftrightarrow \forall x, y, z : P(x \mid z, y) = P(x \mid z)$

Hidden Markov Models (HMM) (13-15)

Forward Algorithm

Belief before considering evidence: $B'(X_{t+1}) = P(X_{t+1} \mid e_{1:t}) = \sum_{x_t} B(x_t)P(X_{t+1} \mid x_t)$

In the binary values case (0 and 1).

$$B'(X_{t+1} = 0) = B(X_t = 0)P(X_{t+1} = 0 \mid X_t = 0) + B(X_t = 1)P(X_{t+1} = 0 \mid X_t = 1)$$

$$B'(X_{t+1} = 1) = B(X_t = 0)P(X_{t+1} = 1 \mid X_t = 0) + B(X_t = 1)P(X_{t+1} = 1 \mid X_t = 1)$$

Belief after considering evidence: $B(X_{t+1}) = P(X_{t+1} \mid e_{1:t+1}) \propto_X B'(X_{t+1})P(e_{t+1} \mid X_{t+1})$

Particle Filtering

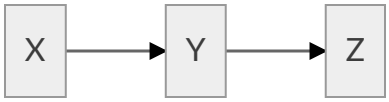
- Sometimes the domain of X is too large for the forward alg like when it's continuous.
- Keep track of a set of states x called particles.
- Elapse time: $x' = \text{sample}(P(X' \mid x))$
- Weight particles based on evidence: $w(x) = P(e \mid x)$
- Resample: Get new samples from a new distribution made by multiplying the probability of each state with a particle by that particle's weight.

Bayes' Networks (16-19)

- Network is a directed, acyclic graph (DAG)
- Nodes in the graph are random variables.
- Edges are parent child relations.
- Each node has a conditional distribution $P(node \mid parents(node))$ associated with it, usually represented as a conditional probability table (CPT).
- Bayes' nets implicitly encode a full joint distribution $P(x_1, x_2, ...x_n) = \prod_{i=1}^{n} P(x_i \mid parents(X_i))$.
- Number of entries in a full joint distribution is 2^N while a Bayes' net only has $O(N2^{k+1})$ where N is the number of variables and k is the max number of parents each node has in the Bayes' net.

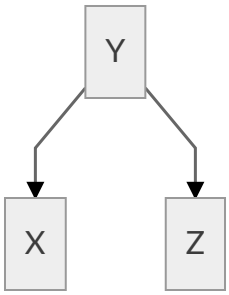
D-Separation

Consider these triples.



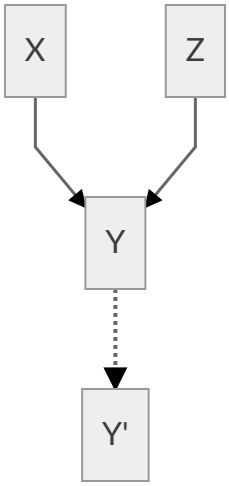
Causal Chain

- $\boxed{X} \rightarrow \boxed{Y} \rightarrow \boxed{Z}$
- $X \not\perp Z$
- $X \perp Z \mid Y$



Common Cause

- $\boxed{X} \leftarrow \boxed{Y} \rightarrow \boxed{Z}$
- $X \not\perp Z$
- $X \perp Z \mid Y$



Common Effect

- $\boxed{X} \rightarrow \boxed{Y} \leftarrow \boxed{Z}$
- $X \perp Z$
- $X \not\perp Z \mid Y$
- $X \not\perp Z \mid Y'$ where Y' is any descendent of Y

In the above triples if Y and Y' are given or not given such that X and Z become independent, then the triple is considered inactive. Triples that are not inactive are active. An undirected path that contains any inactive triple is inactive (active otherwise).

Query: $X_i \perp X_j \mid X_{k_1}, ..., X_{k_n}$?

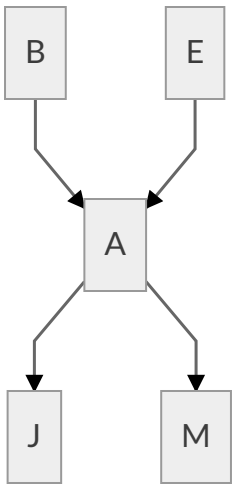
- Check all undirected paths between X_i and X_j
  - If one or more active, then independence not guaranteed.
  - If all inactive, then independence is guaranteed.

If the set of conditional independences of Bayes' net A is a subset of the set of conditional independences of Bayes' net B, then all distributions in A can be encoded in the structure of B.

Inference

- Posterior Probability: $P(Q \mid E_1 = e_1, ...E_k = e_k)$
- Most likely explanation: $\arg\max_q P(Q = q \mid E_1 = e_1, ...E_k = e_k)$

Enumeration



$$P(B \mid +j, +m) \propto_B P(B, +j, +m)$$

$$= \sum_{e,a} P(B, e, a, +j, +m)$$

$$= \sum_{e,a} P(B)P(e)P(a \mid B, e)P(+j \mid a)P(+m \mid a)$$

Variable Elimination

Factors:

1. Joint distribution: $P(X, Y)$ sums to 1.
2. Selected joint: $P(x, Y)$ sums to $P(x)$.

3. Single conditional: $P(Y \mid x)$ sums to 1.
4. Family of conditionals: $P(Y \mid X)$ sums to $|Y|$
5. Specified family: $P(y \mid X)$ sums inconsistently.

How to:

1. **Initialize**: Delete all entries in all factors that are inconsistent with evidence.
2. Pick a hidden variable (probably the one in the least factors).
3. **Join**: Get all factors that include the joining variable. Build joint factor by multiplying consistent entries across factors.
4. **Marginalize**: Sum entries in the joint factor that differ only by the marginalizing variable.
5. **Repeat** steps 2 through 3 until you have eliminated all hidden variables.
6. Join any remaining factors.
7. **Normalize**: Divide each entry in your joint table by the sum of all the entries.

Use the above to eliminate hidden variables. Eliminate variables in the most factors.

Sampling

Can be used to approximate inference or to learn probabilities.

TODO: Explain sampling from a single distribution (too trivial for now).

Prior Sampling

1. Find an ordering of variables that is consistent with the Bayes' net (i.e. parents always come before children).
2. Set each variable X in the sample by going along the ordering and sampling a x from $P(X \mid \text{Parents}(X))$.
3. Return the filled out sample.

Rejection Sampling

If you know the queries ahead of time then you might be able to speed things up.

- You only have to sample as far in the ordering as the lowest variable from your queries.
- If you get a sample that's inconsistent with evidence in all the queries, you can reject it and start over.

Likelihood Weighting

1. Give every sample has an initial weight $w = 1$.
2. Instead of rejecting samples when they contradict the evidence, just force them to match the evidence and then multiply the sample's weight by $P(e \mid \text{Parents}(E))$.
3. When you're trying to extract the distribution from the sample set, count the samples according to their weights.

Gibbs Sampling

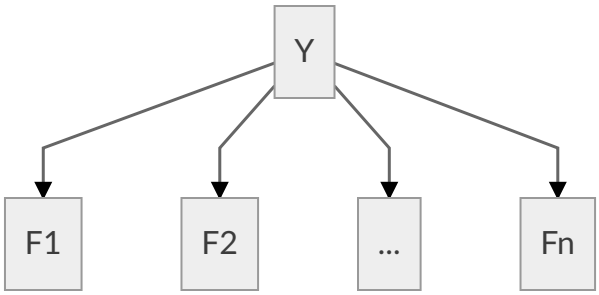
1. Instantiate a purely random sample as a starting place, but make it consistent with evidence.
2. Update one non-evidence variable (selected at random) by sampling it conditioned on all the other variables.
3. Repeat for a long time.
4. Return the final version of the sample.

For step 2:

$$P(X \mid e_1, ..., e_n) = \frac{P(X, e_1, ..., e_n)}{P(e_1, ..., e_n)} \propto_X \prod \text{CPTs with } X$$

Machine Learning

Naive Bayes



$$\boxed{F_1} \leftarrow \boxed{Y} \rightarrow \boxed{F_n}$$

$$P(Y \mid F_1, ...F_n) \propto_Y P(Y, F_1, ...F_n) = P(Y) \prod_{i=1}^{n} P(F_i \mid Y)$$

Parameter Estimation

Maximum Likelihood

$P_{ML}(x) = \frac{\text{count}(x)}{\text{total samples}}$

Laplace Smoothing

Pretend you saw every outcome k more than you actually did.

$P_{LAP,k}(x) = \frac{c(x)+k}{\sum_x [c(x)+k]} = \frac{c(x)+k}{N+k|X|}$

$P_{LAP,k}(x \mid y) = \frac{c(x,y)+k}{c(y)+k|X|}$

Perceptron

$y = \arg\max_y w_y \cdot f(x)$

If wrong, subtract $f(x)$ from weight of wrong label and add $f(x)$ to weight of right label.

Mira

Before making adjustments to weight vectors, multiply $f(x)$ by τ to get near minimal adjustment that still fixes the issue.

$\tau = \frac{(w'_y - w'_{y*}) \cdot f + 1}{2f \cdot f}$

Crazy gradient decent tricks

- Function to computation graph.
- Top numbers are evaluations of the function along.
- If node is a function of x, plug top left number into derivative of node function and multiply by the bottom right number.
- If node is just a + sign then push the bottom numbers through.
- If the node is a * sign then multiply the bottom right number by both of the top left numbers and then swap their positions.