Artificial Intelligence

Spring 2019

1 Problem Solving: Searching

Definition. A **problem (domain)** is a 5-tuple $(S, A, \textsc{Actions}, \textsc{Result}, c)$ where

- S is a set, whose elements are called *states*,
- A is a set, whose elements are called *actions*,
- $\textsc{Actions}$: $S \to \mathcal{P}(A)$ is a function which maps a state to a set of *applicable* actions,
- $\textsc{Result}$: $S \times A \to S$ is a function maps a state s to a new state s' which is the result of performing action a, and
- $c : S \times A \times S \to \mathbb{R}$ is a *cost function*.

An **instance** of a problem is a 3-tuple (P, I, G) where P is a problem, $I \in S$ is an *initial state*, and $G \subseteq S$ is a set of *goal states*. A **solution** to a problem is a tuple $(a_1, \ldots, a_n) \in A^n$ such that $\textsc{Result}(\ldots \textsc{Result}(\textsc{Result}(I, a_1)), a_2) \ldots) \in G$.

The **state space** of a problem instance is the set of states reachable from the initial state by following a sequence of actions. The state space is implicitly defined by the states, actions, and *transition model* ($\textsc{Actions}$, $\textsc{Result}$).

Example. The 8-puzzle.

- $S = \{M_\sigma\}$, $M_\sigma = (m_{ij})$, $0 \le i, j \le 2$, where $m_{ij} = \sigma(3i + j)$ and $\sigma \in S_9$. The position of the 9 represents the location of the blank tile.
- $A = [8]$. Corresponds to switching the location of tile $i \in A$ with the black square.
- Let $\text{loc}(M, i) = (x, y)$ such that $M_{xy} = i$. Then $a \in \textsc{Actions}(M)$ if and only if $|x_a - x_9| + |y_a - y_9| = 1$ where $\text{loc}(M, a) = (x_a, y_a)$ and $\text{loc}(M, 9) = (x_9, y_9)$ (i.e. tile i is adjacent to the blank).
- $\textsc{Result}(M_\sigma, a) = M_{\sigma'}$ where $\sigma' = \sigma \cdot (a, 9)$.
- $c(M, a, M') = 1$ for all $(M, a, M') \in S \times A \times S$.
- $I \in S$
- $G = \{M_{(1,2,\ldots,9)}\}$, for example.

The state space implicitly defines a directed graph, called a *search tree*, whose vertices are states and whose edge set $E = \{(s_i, s_j, a) : s_j = \textsc{Result}(s_i, a)\}$. Solutions to problems are discovered using a graph/tree search algorithm. We call the set of unexplored vertices in the search tree, the *frontier.*

```
initialize frontier with $I$
while frontier is not empty:
    remove $s$ from frontier #crucial step
    if $s \in G$:
        return solution($s$)
    else:
        add successors($s$) to frontier
return null
```

1.1 Search Strategies

1.1.1 Uninformed

Let b be the branching factor (maximum number of successors of any node), d be the depth of the shallowest node, and m the length of the longest path in the state space. We assume step costs are non-negative and $b < \infty$.

- BFS
  - Expand shallowest nodes in frontier. Use FIFO queue to store frontier.
  - Goal test can be applied when node is generated. Discard paths to explored states or states in the frontier.
  - *Bidirectional* search starts a BFS from initial and goal state. Unfortunately, it is often difficult to find predecessors of a given node.
- Uniform-cost Search
  - Expand node with lowest path cost. Use minimum priority queue.
  - Only apply goal test when node is expanded.
- DFS
  - Expand deepest node first. Use LIFO queue.
  - *Backtracking* search only generates 1 successor at a time. Uses only $O(m)$ space.
  - *Depth-limited* search only explores paths of length at most k. Combines DFS and BFS. *Iterative deepening* (IDS) increments k until a solution is found. It generates a total of $d(b)(d-1)(b^2) + \ldots + 1(b^d) = O(b^d)$ nodes. It is the ideal (uninformed) search method for large search spaces with unknown solution depth.
  - *Iterative lengthening* incrementally increases path cost limits (combine DFS and uniform-cost).

1.1.2 Informed

An *evaluation function* $f(n)$ is a cost estimate of the cost of the cheapest solution from the initial state to a goal state, through node n. A *heuristic function* $h(n)$ estimates the cheapest cost of a path from node n to a goal state. We say h is admissible if it never overestimates the actual

	BFS	DFS (tree)	DFS (graph)	Uniform-cost	Depth-limited	IDS	Bidirectional
Complete	Yes	No	Yes	Yes	No	Yes	Yes
Optimal	Yes†	No	No	Yes	No	Yes†	Yes†
Time	$O(b^d)$	$O(b^m)$		$O(b^{1+\lfloor \frac{C^*}{\epsilon} \rfloor})$	$O(b^k)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(bm)$		$O(b^{1+\lfloor \frac{C^*}{\epsilon} \rfloor})$	$O(bk)$	$O(bd)$	$O(b^{d/2})$

Table 1: †: if cost function is non-decreasing in path length. *: if all step costs are at least $\epsilon > 0$. Tree version does not maintain explore set.

cheapest cost. We say h is *consistent* or *monotonic* if $h(n) \leq c(n, a, n') + h(n')$ for all successors n' of n. Note consistency implies admissibility.

- *Greedy best-first search.* Uses $f(n) = h(n)$. Expands node on frontier with cheapest $f(n)$. Is incomplete and non-optimal. Its runtime is highly dependent on the chosen heuristic.
- *A* search.* Uses $f(n) = g(n) + h(n)$ (recall $g(n)$ is the true cost of a path from initial node to node n).

Theorem 1.1

Tree-search A* is optimal if $h(n)$ is admissible. Graph-search A^* is optimal if $h(n)$ is consistent.

Proof. Suppose h is consistent and n' is a successor of n. So $g(n') = c(n, a, n') + g(n)$ for some $a \in A$. Then

$$f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n),$$

so f is non-decreasing along any path. Moreover, whenever A* selects a node for expansion, the optimal path to that node has been found. Suppose not, then there exists a node n' in the frontier on the optimal path from I to n. Then n' has a lower cost than n and would have been expanded first, a contradiction. ∎

Let C^* be the cost of the optimal solution. It follows that A* never expands a node with $f(n) > C^*$. Hence A* is complete if all step-costs are strictly positive. All nodes with $f(n) < C^*$ will be expanded. In this sense, A* is optimally efficient, no other algorithm is guaranteed to expand fewer nodes. Note: As with uninformed search, A* often runs out of memory before time costs become significant.

- Iterative-deepening A*. Uses f-costs as cutoff. Ideal for unit step-costs.
- MA* (memory-bounded A*) and SMA* (simplified MA*). SMA* works like A* till memory is full then drops the oldest leaf node with the largest f-cost, before backing up the value to its parent.

1.1.3 Generating Heuristics

- One way to generate a heuristic is to relax the restrictions on a problem (this adds more edges to the state space). The true cost of an optimal solution to the relaxed problem will always be an admissible (and consistent) heuristic for the original. For example, in the 8-puzzle, allowing a tile to move to any adjacent square yield the heuristic $h_2 = \sum_{\text{tiles } t}$ distance of t from actual position.

- If $h_1, \ldots, h_m$ are admissible heuristics, so is $h(n) = \max\{h_1(n), \ldots, h_m(n)\}$.
- One can use pattern databases to store the exact cost of solving subproblems of a given problem from a given position (cost of subproblem is always less than entire so the corresponding heuristic is admissible).

2 Adversarial Search

Definition. A **game** is a 6-tuple $(S_0, P, A, \textsc{Result}, T, U)$ where

- S_0 is the initial state.
- $P(s)$ is the player whose turn it is to move in state s.
- $A(s)$ is the set of applicable actions in state s.
- $\textsc{Result}$ is defined as in section 1.
- $T(s)$ is true if and only if s is a terminal state.
- $U(s, p)$ returns the utility of player p in terminal state s.

A **zero-sum** game is a game in which the total payoff to all players in any terminal state is a fixed constant. A **strategy** of a player is a function which assigns, to every state in which it is that player's turn, an applicable action.

2.1 Minimax

The minimax algorithm returns the value of the optimal action in state s.

$$\textsc{Minimax}(s) = \begin{cases} U(s) & \text{if } T(s) \\ \max_{a \in A(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } P(s) = \textsc{Max} \\ \min_{a \in A(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } P(s) = \textsc{Min} \end{cases} \tag{2.1}$$

2.1.1 Alpha Beta Pruning

The idea of $\alpha\beta$ pruning is to avoid unnecessary searches of branches in the game tree. For instance, if Max has a choice between actions which will transition to either state B, which he knows yields utility 3, or state C, in which Min has action choices yielding utility of 2, 4, and 10, respectively, then Max will know to go to state B after he explores the branch of the search tree which yields utility 2 (and without having to explore any other subtrees of state C) because then Min can do no worse than 2 (and Max no better than 2) in state C, and yet Max already has a better alternative.

```
$\alpha\beta$−Search($s$):
    $v \leftarrow$ max-val($s$, $-\infty$, $\infty$)
    return $a \in A(s)$ such that $val(a)$ = $v$

max−val($s$, $\alpha$, $\beta$):
    if $T(s)$ then return $U(s)$ #see defn of game
    $v \leftarrow -\infty$
    for $a \in A(s)$:
        $v \leftarrow \max(v$, min−val(RESULT($s$, $a$), $\alpha$, $\beta$))
        if $v \geq \beta$ then return $v$ #min player has a better move
        $\alpha \leftarrow \max(\alpha, v)$
    return $v$ #stored within the game tree at node s
```

min-val is defined similarly, but the roles of min/max and α/β are reversed. α represents the utility of the **highest** value choice found so far, *at any choice point along the search path, for* MAX. β represents the utility of the **lowest** value choice found so far, *at any choice point along the search path, for* MIN. Note: This means that α, β only get updated via information passed through v.

2.1.2 H-Minimax

Let $h : S \to \mathbb{R}_{\geq 0}$ be a heuristic function.

$$\text{H-MINIMAX}(s) = \begin{cases} h(s) & \text{if } \text{CUTOFF}(s) \\ \max_{a \in A(s)} \text{H-MINIMAX}(\text{RESULT}(s,a)) & \text{if } P(s) = \text{MAX} \\ \min_{a \in A(s)} \text{H-MINIMAX}(\text{RESULT}(s,a)) & \text{if } P(s) = \text{MIN} \end{cases} \tag{2.2}$$

The most simplistic choice for a CUTOFF function is a depth limit.

Cutoffs and heuristics? Finish 5.4.

2.2 Stochastic Games

Stochastic games involve an element of chance, e.g. dice, cards, etc. We introduce chance nodes into the game tree. Each chance node in the tree has a set of possible events $\Omega(s)$. Every $r \in \Omega(s)$ has an associated probability $P(r)$.

$$\mathbb{E}\text{-MINIMAX}(s) = \begin{cases} U(s) & \text{if } T(s) \\ \max_{a \in A(s)} \mathbb{E}\text{-MINIMAX}(\text{RESULT}(s,a)) & \text{if } P(s) = \text{MAX} \\ \min_{a \in A(s)} \mathbb{E}\text{-MINIMAX}(\text{RESULT}(s,a)) & \text{if } P(s) = \text{MIN} \\ \sum_{r \in \Omega(s)} P(r)\mathbb{E}\text{-MINIMAX}(\text{RESULT}(s,r)) & \text{if } P(s) = \text{CHANCE} \end{cases} \tag{2.3}$$

3 Local Search

Local search only maintains the current state, provided a massive savings on the memory costs of local search algorithms. However, they sacrifice the systematic approach of the previous algorithms, and hence often aren't complete, or optimal.

3.1 Hill Climbing

Each state in the state space is associated a number corresponding the the "goodness" of that solution, with goal states being the optimal states. Hill climbing greedily chooses the highest-valued neighbor of the current state until a peak is reached. This easily gets stuck at a local maxima or on plateau in the state space landscape.

Stochastic hill climbing randomly chooses amongst uphill moves (with a distribution correlated to the steepness of the move) in an attempt to avoid converging to non-optimal local maxima. **Random-restart hill climbing** randomly generates initial states and performs HC until a goal is found.

3.2 Simulated Annealing

Improved of HC by allowing some downhill moves. In particular, we choose a random successor s of our current node, if it improves the objective function s becomes the current node. Otherwise, s becomes our current node with probability $e^{\Delta E/T}$, where $\Delta E = value(s) - value(curr)$ measures the badness of the move (the more negative ΔE is the less likely the move is to be chosen) and T is a function the decreases (towards 0) over time (at time progresses bad moves are less likely to be chosen).

3.3 Local Beam Search

Initially uses k states (gene pool), and checks if any is a goal state. If not, we pick their best successors, then repeat.

A **genetic algorithm** is a variation on LBS. We initially have k random states in our population. Typically, k is represented as a bit string. Each state is rated by a **fitness function**. The probability of being chosen for production directly correlates with the nodes fitness. For example,

$$P(s) = \frac{f(s)}{\sum_{v \in \text{current states}} f(v)}.$$

Then we pair off states, according to their probabilities. A **crossover** is performed which crosses over the parent strings at a randomly selected crossover point to form two offspring. Finally, random mutations are performed on the string.

3.4 Non-deterministic Actions

The RESULT function returns a set of possible outcomes, instead of a single state. Solutions for these problems are trees, rather than sequences of actions.

To find solutions, we build an **and-or tree**. The or nodes correspond to branching introduced by the agent's choices. The and nodes correspond to the environments choices. A solution for an and-or search problem is a subtree that has a goal node at every leaf, specifies one action at each or node and includes every outcome at every and node.

3.5 Partial Observation Search

3.5.1 No Observation

A belief state search problem given an underlying problem P is defined as follows

- The set of (belief) states if $\mathcal{P}(N)$, where N is the sets of states in P. The agent may not know exactly what state an action will take him to, or what state he was initially, we must keep track of all possible current states.
- The initial state is typically N, unless the agent has more knowledge at the beginning.
- Actions. We can take the union of all possible available actions in each state within our current belied state (this is dangerous as an action available in one state may "end the world" in another). Otherwise we can take the intersection over all such states.
- $\text{RESULT}(b, a) = \{s' : s' = \text{RESULT}_P(s, a) \text{ and } s \in b\}$, if RESULT_P is deterministic.
- Goal test. All states in belief state must be goal states.

3.5.2 With Some Observations

If a belief state b on action a,

$$\text{PREDICT}(b, a) = \bigcup_{s \in b} \text{RESULTS}_P(s, a).$$

POSSIBLE-PERCEPTS(b) is the set of all percepts that may be received at some state in b. Once were given a percept, we can update our belief about the current state of the world,

$$\text{UPDATE}(b, o) = \{s : o = \text{PERCEPT}(s) \text{ and } s \in b\}.$$

Then

$$\text{RESULTS}(b, a) = \{b_o : b_o = \text{UPDATE}(\text{PREDICT}(b, a), o) \text{ and } o \in \text{POSSIBLE-PERCEPTS}(\text{PREDICT}(b, a))\}$$

4 Constraint Satisfaction

Definition. A **constraint satisfaction problem (CSP)** is a 3-tuple (X, D, C) where

- X is a set of variables;
- D is a set of domains, one for each variable; and
- C is a set of constraints.

A domain is a set of allowable values for a variable. A constraint is a pair (s, R) where s is a tuple of variables in the relation R and R is a relation restricting the values of the variables in s.

Definition. A state in a CSP is an **assignment** of values to the variables. An assignment that doesn't violate any constraint is **consistent**. A **complete** assignment assigns values to every variable. A **solution** is a complete, consistent assignment.

Constraint propagation uses constraints to reduce the number of legal states.

A CSP is **k-consistent** if for any set of $k-1$ variables and a consistent assignment on those variables, a consistent value can be assigned for the k-th variable. A **strongly k-consistent** CSP is i-consistent for $1 \leq i \leq k$.

- A variable X is **1-consistent** or **node-consistent** if all values in D_X satisfy the unary constraints on X.
- A variable X is **2-consistent** or **arc-consistent** with respect to variable Y if for all $x \in D_x$ there is a $y \in D_Y$ such that (x, y) satisfies the binary constraints on (X, Y).
- 3-consistency is sometimes called **path-consistent**.

Assume CSP C has n variables, the size of any domain is $\leq d$ and c constraints.

```
AC−3(CSP $C$): #updates domains so $C$ is arc-consistent. $O(cd^3)$.
    $S \leftarrow$ set of arcs in $C$
    while $S$ is not empty:
        $(X_i, X_j) \leftarrow S$.pop
        if Revise($C$, $X_i$, $X_j$):
            if size($D_i$) = 0: return false #$C$ is inconsistent
            for each $X_k$ in $X_i$.neighbors - $\{X_j\}$:
                $S$.add(($X_k, X_i$))
return true
```

```
Revise(CSP $C$, $X_i$, $X_j$):
    $R \leftarrow$ false
    for $x \in D_i$:
        if $\nexists y \in D_j$ such that $(x, y)$ satisfies constraint on $(X_i, X_j)$
            delete $x$ from $D_i$
            $R \leftarrow$ true
return $R$
```

4.1 Backtracking

```
Backtrack(Assignment $A$, CSP $C$):
    if $A$ is complete: return $A$
    $V \leftarrow$ getUnassigned($C$)
    for $x \in$ OrderDomain($C, A, V$)
        if $x$ is consistent with $A$:
            add $\{V = x\}$ to $A$
            $I \leftarrow$ Inference($C, V, x$)
            if $I \neq$ fail:
                add $I$ to $A$
                result $\leftarrow$ Backtrack($A$, $C$)
                if result $\neq$ fail:
                    return result
        remove $\{V = x\}$ and $I$ from $A$ #backtrack @ inconsistent assignment
```

- `getUnassigned` is usually chosen with the **minimum remaining values** heuristic (choosing the variable with the fewest no. of remaining legal moves) or the **degree** heuristic (choosing the variable involved in the largest no. of constraints on unassigned variables).
- `OrderDomain` typically uses the **least-constrained-value** heuristic (choose value ruling out fewest choices for neighboring variables).
- `Inference`.
  - **Forward Checking** or **MAC**.
- We may also consider the question of how to prevent searches from arriving at the same inconsistent assignment.

5 Propositional Logic

If a sentence α is true in some model $\mathcal{M}$, we say $\mathcal{M}$ **satisfies** α. We let $M(\alpha)$ denote the class of all models of α. Given two sentences α, β we say α **logically entails** β if and only if $M(\alpha) \subseteq M(\beta)$. We write $\alpha \vDash \beta$. If an inference algorithm i can derive β from α we write $\alpha \vdash_i \beta$. An inference algorithm is **sound** if it only derives entailed sentences; it is **complete** if it can derive any entailed sentence.

Section 7.4 describes the syntax and semantics of predicate logic and a brute force inference algorithm (check if every model satisfying α also satisfies β).

5.1 Theorem Proving

We say two sentences α and β are logically equivalent if $M(\alpha) = M(\beta)$, that is if

$$\alpha \vDash \beta \text{ and } \beta \vDash \alpha.$$

A sentence is **valid** (or tautological) if it is satisfied in all models. A sentence is **satisfiable** if it is satisfied in some model. Clearly, α is valid if and only if $\neg\alpha$ is unsatisfiable. Proof by contradiction corresponds to the statement: $\alpha \vDash \beta$ if and only if $(\alpha \wedge \neg\beta)$ is unsatisfiable. A logical system is **monotonic** if the set of entailed sentences increases as more axioms are added.

One important inference rule for the completeness of propositional logic is **resolution**.

$$\frac{\ell_1 \vee \ldots \vee \ell_l \quad \neg\ell_i}{\ell_1 \vee \ldots \vee \ell_{i-1} \vee \ell_{i+1} \vee \ldots \vee \ell_k}.$$

5.1.1 Conjunctive Normal Form

A sentence expressed as a conjunction of clauses—disjunction of literals—is said to be in **conjunctive normal form (CNF)**.

Proposition 5.1. Every sentences in PL is logically equivalent to a sentence in CNF.

- Eliminate $\Leftrightarrow$: replace $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta)(\beta \Rightarrow \alpha)$.
- Eliminate $\Rightarrow$: replace $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$.
- Move $\neg$ inwards: DeMorgan's and double negation.
- Distribute $\vee$'s over $\wedge$'s.

5.1.2 Complete PL Theorem Prover

Now suppose we have a CNF representation of our knowledge base K and a sentence α and we want to prove $K \vDash \alpha$. This can be done by demonstrating $(K \wedge \neg\alpha)$ is unsatisfiable. Let $S_0 = K \wedge \neg\alpha$ and for $i \geq 1$, let $S_i = S_{i-1} \cup R(S_{i-1})$, where $R(S_{i-1})$ is the set of clauses derived by resolving any pair of clauses in S_{i-1} (removing redundancies). We define the **resolutional closure** of S_0 to be

$$RC(S_0) = \bigcup_{i=0}^{\infty} S_i.$$

Since we remove redundancies, $RC(S_0)$ is finite. Hence the algorithm is complete.

Theorem 5.2

If a set of clauses is unsatisfiable, then its resolutional closure contains the empty clause.

Proof. Suppose $RC(S_0)$ does not contain the empty clause. We'll construct a model for S_0. Let $P_1, \dots, P_k$ be the variables in S_0.

- For $i = 1$ to k, if a clause in $RC(S)$ contains the literal $\neg P_i$ and all of its other literals are false under the assignment of $P_1, \dots, P_{i-1}$, then assign $P_i = F$. Otherwise, $P_i = T$.

Suppose this assignment weren't a model of S_0. Say at stage i, assigning P_i makes clause C false. Thus all other literals in C must have already been falsified by the assignment to $P_1, \dots, P_{i-1}$. C must be of the form

$$(F \vee \dots \vee F \vee P_i) \text{ or } (F \vee \dots \vee F \vee \neg P_i).$$

Moreover, both of these clauses must lie in $RC(S_0)$, otherwise there'd be a possible assignment of P_i. By closure, $RC(S_0)$ contains the resolution of these clauses, which would've already been falsified in an earlier step, a contradiction. In fact, we have a model of $RC(S_0)$ which includes S_0. ∎

5.1.3 Horn & Definite Clauses

A **Horn clause** contains at most one positive literal. A **definite clause** is a Horn clause with exactly one positive literal. A **goal clause** has no positive literals. Note the resolution of two Horn clauses is a Horn clause. A definite clause $(\neg P_1 \vee \ldots \vee \neg P_{k-1} \vee P_k)$ may be rewritten

$$(P_1 \wedge \ldots \wedge P_{k-1}) \Rightarrow P_k.$$

Inference on Horn clauses is done with forward or backward chaining and deciding entailment can be achieved in time linear in the size of the knowledge base.

5.1.4 Forward & Backward Chaining

Forward chaining decides whether a query Q is entailed by a knowledge base of definite clauses. It is initialized with positive literals in the knowledge base and if all the premises of an implication are known, the conclusion is added to the set of known facts (apply modus ponens). The algorithm runs until Q is entailed or no new inferences can be made.

Proposition 5.3. Forward chaining is complete.

Proof. Let S be the set of inferred symbols when the algorithm terminates (no new literals can be entailed). We say a symbol is true iff it is in S. Under this model, all definite clauses in the knowledge base are true. Suppose not, say $(P_1 \wedge \ldots \wedge P_{k-1}) \Rightarrow P_k$ is false. Then $P_1 \wedge \ldots \wedge P_{k-1}$ is true and P_k is false. Since P_k is false, it has not yet been inferred. But $P_1, \ldots, P_{k-1}$ are true, so FC should infer P_k, contradicting the fact that the algorithm has reached a fixed point. Therefore S induces a model $\mathcal{M}$ of the knowledge base K and any sentence q entailed by the K, is true in all models of K, in particular $\mathcal{M}$. Hence q is inferred by FC. ■

Backwards chaining works backwards from the query q. It finds implications whose conclusion is q and if all premises of some implication are known to be true (by BC), then q will be true.

6 Uncertain Inference

6.1 Bayesian Networks

A **Bayesian network** is a directed acyclic graph (V, E) where V is a set of nodes, consisting of a random variable and a conditional probability distribution. Two random variables X, Y are connected by an edge, $X \to Y$, if X directly causes Y. The conditional probability distribution in a node is $\mathbf{P}(X|Parents(X))$, where $Parents(X)$ are the in-neighbors of X.

$$\mathbf{P}(\mathbf{X}|\mathbf{e}) = \alpha\mathbf{P}(\mathbf{X}, \mathbf{e}) = \alpha \sum_{\mathbf{y}} \prod_{i=1}^{N} \mathbf{P}(\mathbf{X}_i|parents(\mathbf{X}_i))$$

6.2 Temporal Bayesian Inference

- Let $\mathbf{X}_t$ denote the set of state variables at time $t \geq 0$.
- Let $\mathbf{E}_t$ denote the set of observed variables at time $t \geq 1$. We say $\mathbf{E}_t = \mathbf{e}_t$.

We use $\mathbf{X}_{a:b}$ to denote the set of variables from $\mathbf{X}_a$ to $\mathbf{X}_b$, inclusive.

- **Markov Assumption**: The current state depends only on a fixed, finite number of previous states. Typically, we work with **first-order Markov processes**, in which the current state depends only on the previous state and nothing else. Hence

  $$\mathbf{P}(\mathbf{X}_t|\mathbf{X}_{0:t-1}) = \mathbf{P}(\mathbf{X}_t|\mathbf{X}_{t-1}).$$

  Furthermore, we also assume that changes in the world are caused by a **stationary process**, that is the rules by which the world changes, *do not themselves change*, i.e.

  $$\mathbf{P}(\mathbf{X}_t|\mathbf{X}_{t-1}) \text{ is fixed for all } t \geq 1.$$

- **Sensor Markov assumption**: Says the current sensor value depends only on the current state of the world,

  $$\mathbf{P}(\mathbf{E}_t|\mathbf{X}_{0:t}, \mathbf{E}_{1:t-1}) = \mathbf{P}(\mathbf{E}_t|\mathbf{X}_t).$$

  Again, we assume that this process is stationary,

  $$\mathbf{P}(\mathbf{E}_t|\mathbf{X}_t) \text{ is fixed for all } t \geq 1.$$

- We must also specify $\mathbf{P}(\mathbf{X}_0)$, the prior distribution of the state variables.

Under these assumptions, we need only specify 3 distributions to full describe the corresponding temporal Bayesian network.

We can do Bayesian inference as follows:

$$\mathbf{P}(\mathbf{X}_{0:t}, \mathbf{E}_{1:t}) = \mathbf{P}(\mathbf{X}_0) \prod_{i=1}^{t} \mathbf{P}(\mathbf{X}_t|\mathbf{X_{t-1}})\mathbf{P}(\mathbf{E}_t|\mathbf{X}_t).$$

6.2.1 Filtering

Filtering is the process of computing the current belief state, that is, the posterior distribution of the most recent state given the evidence to date:

$$\mathbf{f}_{1:t} := \mathbf{P}(\mathbf{X}_t|\mathbf{e}_{1:t}).$$

We compute this as follows:

$$\mathbf{f}_{1:t} = \alpha\textsc{Forward}(\mathbf{f}_{1:t-1}, \mathbf{e}_t), \tag{6.1}$$

where

$$\textsc{Forward}(\mathbf{f}_{1:t-1}, \mathbf{e}_t) = \mathbf{P}(\mathbf{e}_t|\mathbf{X}_t) \sum_{\mathbf{x}_{t-1}} \mathbf{P}(\mathbf{X}_t|\mathbf{x}_{t-1}) P(\mathbf{x}_{t-1}|\mathbf{e}_{1:t-1}).$$

Note: $P(\mathbf{x}_{t-1}|\mathbf{e}_{1:t-1})$ is contained in $\mathbf{f}_{1:t-1}$. Hence in order to compute the filter of a new time step, we need only perform a constant amount of work (the # of assignments to the state variables). The update is not dependent on t.

6.2.2 Prediction

We can derive a similar recursive procedure ($k \geq 0$):

$$\mathbf{P}(\mathbf{X}_{t+k+1}|\mathbf{e}_{1:t}) = \sum_{\mathbf{x}_{t+k}} \mathbf{P}(\mathbf{X}_{t+k+1}|\mathbf{x}_{t+k}) P(\mathbf{x}_{t+k}|\mathbf{e}_{1:t}).$$

If $k = 0$, we have $P(\mathbf{x}_{t+k}|\mathbf{e}_{1:t}) \in \mathbf{f}_{1:t}$. Thus, after we compute the current belief state, we can predict the next belief state in constant time (this can then be used to compute the belief state 2 time steps from now, and so on).

6.2.3 Smoothing

Smoothing is the process of updating the distribution of past states given evidence up to the present:

$$\mathbf{P}(\mathbf{X}_k|\mathbf{e}_{1:t}), \quad 0 \leq k < t.$$

We have

$$\mathbf{P}(\mathbf{X}_k|\mathbf{e}_{1:t}) = \alpha \mathbf{P}(\mathbf{X}_k|\mathbf{e}_{1:k}) \mathbf{P}(\mathbf{e}_{k+1:t}|\mathbf{X_k}) = \alpha \mathbf{f}_{1:k} \cdot \mathbf{b}_{k+1:t}.$$

(The $\cdot$ indicates a pointwise product).

$$\mathbf{b}_{k+1:t} := \mathbf{P}(\mathbf{e}_{k+1:t}|\mathbf{X}_k) = \sum_{\mathbf{x}_{k+1}} P(\mathbf{e}_{k+1}|\mathbf{x}_{k+1}) P(\mathbf{e}_{k+2}|\mathbf{x}_{k+1}) \mathbf{P}(\mathbf{x}_{k+1}|\mathbf{X}_k),$$

where $P(\mathbf{e}_{k+2}|\mathbf{x}_{k+1}) \in \mathbf{b}_{k+2:t}$. So we can compute the $\mathbf{b}$'s in constant time from t down to $k+1$, using $\mathbf{b}_{t+1:t} = \mathbf{1}$. This is called the **Forward-Backward** algorithm.

6.2.4 Most Likely Sequence (Viterbi's Algorithm)

$$X^* = \arg\max_{\mathbf{x}_{1:t}} \mathbf{P}(\mathbf{x}_{1:t}|\mathbf{e}_{1:t}).$$

The main idea is that the most likely paths to each state in $\mathbf{x}_t$ can be extended to form the most likely paths to $\mathbf{x}_{t+1}$.

$$\max_{\mathbf{x}_{1:t}} \mathbf{P}(\mathbf{x}_{1:t}, \mathbf{X}_{t+1}|\mathbf{e}_{1:t+1}) = \alpha \mathbf{P}(\mathbf{e}_{t+1}|\mathbf{X}_{t+1}) \max_{\mathbf{x}_t} \left(\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{x}_t) \max_{\mathbf{x}_{1:t-1}} P(\mathbf{x}_{1:t-1}, \mathbf{x}_t|\mathbf{e}_{1:t}) \right). \tag{6.2}$$

We can compute this recursively by defining: $\mathbf{m}_{1:t} := \max_{\mathbf{x}_{1:t-1}} P(\mathbf{x}_{1:t-1}, \mathbf{X}_t|\mathbf{e}_{1:t})$

A **Hidden Markov Model** (HMM) is a temporal probability model with one state variable. (We can transform any temporal model (with state vars: X_i) to an HMM by creating a new state variable with domain $\prod D_i$).

7 Learning

Suppose we are given a **training set** of N input-output samples

$$(x_1, y_1), \ldots, (x_N, y_N)$$

where $y_i = f(x_i)$ for some unknown function f. We want to find a **hypothesis** function h which approximates f. We take h from a **hypothesis space** $\mathcal{H}$. We say h is **consistent** if it correctly identifies all data in the training set. We say h **generalizes** if it handles the **test set** (input-output examples different from the training data) well.

7.1 Decision Trees - Classification

A decision tree has the following structure:

- The internal nodes represent a test of some attribute A_i of the input vector.
- The branches are labelled with the possible attributes values v_i of A_i.
- The leaves of the tree specify a "decision".

The following algorithm builds a (hopefully small) decision tree that is consistent with the example data. `PluralityVal`($\mathbf{S}$) returns the most common classification of the examples in $\mathbf{S}$.

```
DecisionTreeLearn(examples: $\mathbf{X}$, attributes: $A$, parentExs: $\mathbf{P}$):
    if $X$ is empty, return PluralityVal($\mathbf{P}$):
    elif all $x \in \mathbf{X}$ have same classification $C$ return $C$
    elif $A$ is empty then return PluralityVal($\mathbf{X}$)
    else # add new test
        $attr \leftarrow \arg\max_{a \in A}$ Importance($a$, $\mathbf{X}$) # see 7.1.1
        $T \leftarrow$ new decision tree with root $attr$
        for $v_k \in attr$:
            $exs \leftarrow \{e \in X : e.attr = v_k\}$
            $t \leftarrow$ DecisionTreeLearn($exs$, $A - attr$, $\mathbf{X}$)
            add branch in $T$ with label $attr = v_k$ to $t$
        return $T$
```

7.1.1 Attribute Importance

The **entropy** of a random variable V with values v_k having probability p_k, respectively, is

$$H(V) = -\sum_{k=1}^{n} p_k \lg(p_k).$$

For example, if $X \sim \text{Bernoulli}(p)$ then

$$H(X) = -p \lg(p) - (1-p)\lg(1-p) := B(p).$$

The entropy of a random variable is a measure of the amount of information it conveys. If a training set has p positive examples and n negative examples, then

$$H(Goal) = B\left(\frac{p}{p+n}\right).$$

Now suppose attribute A divides the examples into d categories: $E_1, \ldots, E_d$. Let p_k be the number of positive samples and n_k the number of negative samples in E_k. Then the branch $A = v_k$, requires $B(p_k/(p_k + n_k))$ extra bits of information to classify the goal. Let $Rem(A)$ denote the expected value of the remaining entropy after testing A:

$$Rem(A) = \sum_{k=1}^{d} \left(\frac{p_k + n_k}{p + n} \right) B \left(\frac{p_k}{p_k + n_k} \right).$$

We denote the **information gain** of A to be

$$Gain(A) := B \left(\frac{p}{p + n} \right) - Rem(A).$$

Then the importance of an attribute is directly correlated with its gain.

7.1.2 Overfitting

Overfitting is when a hypothesis fits the training data too closely. This tends to reduce the ability of the hypothesis to generalize. Overfitting occurs when the hypothesis is overly complex, having to many parameters, allowing random noise in the sample data to become built into the model. Decision tree pruning combats overfitting. We prune as follows:

- Look at a node with only leaves as descendants. If the test at that node is irrelevant we eliminate it and replace it with a leaf.
- Repeat this process until all nodes are either accepted or have been pruned.

An important question is how to determine the relevance of a node. Intuitively, a node is irrelevant if, on a sample with p positive and n negative examples, the test splits the samples into categories with the same proportion of positive to negative samples as the original, resulting in a gain near 0. We use **significance testing** to rule out irrelevant nodes.

- First, assume there is no underlying pattern in the data (H_0 - **null hypothesis**). That is, as the sample size goes to ∞, the gain goes to 0.
- Now compute the probability that given H_0, a sample of size $v = p + n$ would exhibit the observed deviation from the expected distribution $\langle \hat{p}_k, \hat{n}_k \rangle$ of positive and negative samples for each E_k.

  $$\hat{p}_k = p \left(\frac{p_k + n_k}{p + n} \right) \qquad \hat{n}_k = n \left(\frac{p_k + n_k}{p + n} \right)$$

  We measure the total deviation as

  $$\Delta := \sum_{k=1}^{d} \frac{(p_k - \hat{p}_k)^2}{\hat{p}_k} + \frac{(n_k - \hat{n}_k)^2}{\hat{n}_k}.$$

  Under H_0, $\Delta \sim \chi^2$-distribution with $v - 1$ degrees of freedom.
- If $\Delta \geq \chi^2(v-1; 0.05)$ (that is, if the probability of experiencing this measured deviation under H_0 is less than 5% [or perhaps 1%]), we reject H_0 and conclude the attribute is significant. Otherwise, we prune.

7.2 Regression

Given a weight vector $\mathbf{w} = (w_0, w_1)$ and an input-output pair (x_1, y_1), we set $x_0 = 1$ and define $\mathbf{x} = (x_0, x_1)$. We define or hypothesis function to be $h_{\mathbf{w}}(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x}$. We want to find the weight vector $\mathbf{w}^*$ minimizing the L_2-norm over the data set $\{(x_j, y_j) : 1 \leq j \leq N\}$:

$$Loss(\mathbf{w}) := \sum_{j=1}^{n} (y_j - h_{\mathbf{w}}(\mathbf{x}_j))^2.$$

Using basic calculus, we obtain

$$w_1 = \frac{N(\sum x_j y_j) - (\sum x_j)(\sum y_j)}{N(\sum x_j^2) - (\sum x_j)^2} \qquad w_0 = \frac{\sum y_j - w_1 \sum x_j}{N}.$$

For cases where an analytic solution is not viable, we can use gradient descent to find $\mathbf{w}^*$.

```
$w \leftarrow$ some initial point in parameter space
loop until convergence:
    for $w_i \in \mathbf{w}$:
        $w_i \leftarrow w_i - \alpha \frac{\partial L_2(\mathbf{w})}{\partial w_i}$.
```

Note we call α the **step size**. It's often chosen to either be constant or to decay at a rate $O(1/t)$, e.g. $\alpha(t) = 1/(1+t)$.

Suppose our training data has more components say each input is of the form $(x_0, \ldots, x_n)$ (with $x_0 = 1$). We extend our hypothesis space $\mathcal{H} = \{h_{\mathbf{w}} : \mathbf{w} \in \mathbb{R}^{n+1}\}$. As before, we want to minimize $\sum (y_j - \mathbf{w} \cdot \mathbf{x})^2 = ||Xw - y||^2$. Setting the gradient to 0, we obtain

$$\mathbf{w}^* = (X^t X)^{-1} X^t y.$$

7.2.1 Linear Classifiers

A **decision boundary** is a line (or surface in higher dimensions) that separates data into classes. We call a linear decision boundary a **linear separator** and say data admitting such a separator is **linearly separable**. Our (hard threshold) classification hypothesis taken the following form:

$$h_w(x) = \begin{cases} 1 & \text{if } w \cdot x \geq 0 \\ 0 & \text{otherwise.} \end{cases}$$

Given a random sample x, we can use gradient descent to update the weights using the **perceptron learning rule**:

$$w_i \leftarrow w_i + \alpha(y - h_w(x))x_i.$$

For linearly separable data, this will eventually converge to a linear separator. Observe:

- If the output is correct, $y = h_w(x)$, so the weight remain unchanged.
- If $y = 1$ and $h_w(x) = 0$, then w_i increases if x_i is positive and decreases if x_i is negative. This makes sense since we want $w \cdot x$ to become bigger so $h_w(x)$ outputs 1.

- Similarly, if $y = 0$ and $h_w(x) = 1$, then w_i decreases if x_i is positive and increases if x_i is negative.

We can also use a soft threshold function called the **logistic function**, $L(z) = \frac{1}{1+e^{-z}}$. Our hypothesis $h_w(x) = L(w \cdot x)$. Using gradient descent, our update rule is

$$w_i \leftarrow w_i + \alpha(y - h_w(x))h_w(x)(1 - h_w(x))x_i.$$

7.3 Neural Networks

A neural network consists of **nodes** or **units** connected by weighted **links**. A link from node i to node j transmits an activation a_i from i to j with weight $w_{i,j}$. For each node j, we define its input to be

$$\mathsf{in}_j = \sum_{\text{in-neighbors } i} a_i w_{i,j}.$$

Then, according to the activation function of the network, g, the output of the node is $a_j = g(\mathsf{in}_j)$. A **feed forward network** (FFN) is a DAG arranged in layers (nodes in layer i have links only to nodes in layer $i + 1$). A **recurrent network** allows for cycles. *Each unit has a dummy input* $a_0 = 1$.

7.4 Backpropagation in FFNs

We can then compute the gradient for $Loss_k = (y_k - a_k)^2$ at the k-th output. Note the gradient with respect to weights connecting to a hidden layer to the output layer will be zero except for weights $w_{j,k}$ which connect to the k-th output unit. Thus

$$\begin{aligned}\frac{\partial Loss_k}{\partial w_{j,k}} &= -2(y_k - a_k)\frac{\partial a_k}{\partial w_{j,k}} = -2(y_k - a_k)g'(\mathsf{in}_k)\frac{\partial \mathsf{in}_k}{\partial w_{j,k}} \\ &= -2(y_k - a_k)g'(\mathsf{in}_k)a_j = -a_j\Delta_k,\end{aligned} \tag{7.1}$$

where $\Delta_k = (y_k - a_k)g'(\mathsf{in}_k)$ (the 2 is present in every component of the gradient and can be factored out and absorbed into α). For the gradient wrt $w_{i,j}$ weights connecting the inputs to the hidden layer,

$$\begin{aligned}\frac{\partial Loss_k}{\partial w_{i,j}} &= -2(y_k - a_k)g'(\mathsf{in}_k)\frac{\partial \mathsf{in}_k}{\partial w_{i,j}} = -2\Delta_k\frac{\partial\left(\sum_j a_j w_{j,k}\right)}{\partial w_{i,j}} \\ &= -2\Delta_k w_{j,k}\frac{\partial a_j}{\partial w_{i,j}} = -2\Delta_k w_{j,k} g'(\mathsf{in}_j)a_i.\end{aligned} \tag{7.2}$$

Taking $\Delta_j = g'(\mathsf{in}_j)\sum_k \Delta_k w_{j,k}$, we get the following update rule for weights in our network: $w_{i,j} \leftarrow w_{i,j} + \alpha \cdot a_i \cdot \Delta_j$ (the derivation above generalizes to networks with multiple hidden layers).

8 Bayesian Learning

Bayesian learning consists of computing probabilities of hypothesis given data and making predictions based on data.

$$P(h_i|\mathbf{d}) = \alpha P(\mathbf{d}|h_i)P(h_i) \tag{8.1}$$

$$P(X|\mathbf{d}) = \sum_i P(X|\mathbf{d}, h_i)P(h_i|\mathbf{d}) = \sum_i P(X|h_i)P(h_i|\mathbf{d}) \tag{8.2}$$

The keys to Bayesian learning are the **hypothesis prior**: $P(h_i)$ and the likelihood probabilities: $P(\mathbf{d}|h_i)$. We assume observations are i.i.d. so $P(\mathbf{d}|h_i) = \prod_j P(d_j|h_i)$. Equation (8.2) is likely intractable. So approximations, such as using the most likely hypothesis (given $\mathbf{d}$) to make predictions (**maximum a posteriori** or **MAP** hypothesis), are in general, necessary. By equation (8.1), if $P(h_i)$ is uniform, MAP reduces to choosing h_i that maximizes $P(\mathbf{d}|h_i)$. We call this the **maximum-likelihood** hypothesis.

8.1 Maximum Likelihood Estimation

We begin by assuming out data is **complete**, that is we have values for every variable being measured. **Parameter learning** is the process of estimating the numerical parameter is a probability distribution, given data.

Example. Suppose $X \sim \mathit{Bernoulli}(\theta)$ and we have N independent samples $X_1, \ldots, X_N$ of X. Our hypothesis h_θ is a guess of the parameter θ.

$$P(\mathbf{d}|h_\theta) = \prod_{j=1}^{N} P(d_j|h_\theta) = \theta^c(1-\theta)^{N-c},$$

where c is the number of successes amongst $X_1, \ldots, X_N$. Maximizing this is equivalent to maximizing $\lg(P(\mathbf{d}|h_\theta)) = c\lg\theta + (N-c)\lg(1-\theta)$. Taking the derivative and equating with zero, the probability is maximized when $\theta = \frac{c}{N}$, the sample mean.

Note the problem with a small data set is that some events may be unobserved, so the ML hypothesis will assign probability 0 to these events.

Example. Suppose we have a large bag of c cherry and ℓ lime candies and each candy has a red or green wrapper, say $c = r_c + g_c$ and $\ell = r_\ell + g_\ell$. Also suppose we have reason to believe the flavor of the candy impacts the wrapper. Let $\theta = P(\mathit{cherry})$, $\theta_1 = P(\mathit{red}|\mathit{cherry})$, and $\theta_2 = P(\mathit{green}|\mathit{cherry})$. Then

$$P(\mathbf{d}|h_{\theta,\theta_1,\theta_2}) = \theta^c(1-\theta)^\ell \cdot \theta_1^{r_c}(1-\theta_1)^{g_c} \cdot \theta_2^{r_\ell}(1-\theta_2)^{g_\ell}.$$

We use similar methods as above to derive the maximum log-likelihood estimators. Note: Because taking partials wrt each variable eliminates the remaining variables, *maximum-likelihood parameter learning in a Bayesian network decomposes into separate learning problems, one per parameter.* The Bayesian network in this problem is a naive Bayes model with class variable F and attribute W.

8.1.1 Naive Bayes

A naive Bayes model consists of a Bayesian network with a class variable C (to be predicted) at the root having attributes $X_1, \ldots, X_k$. By Bayes' rule,

$$P(C|x_1, \ldots, x_k) = \alpha P(C)\prod_{i=1}^{k} P(x_i|C).$$

8.2 Learning the Hypothesis Prior

More abstractly, we can model our Bayesian network for this problem by assuming the the flavor variable F is dependent on random variable Θ (similarly wrapper r.v. W depends on Θ_1 and Θ_2).

Contents