|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Reflex | | | |  |  | |
| Reduce training cost by 1) less data, 2) more features, 3) make hypothesis set larger  If generalises well test error equals train error  Use validation set to 1) feature engineering, 2) algo comparison, 3)tune hyperparams | | | |  |  | |
| Find gradient of hinge loss | | | |  |  | |
|  | | | |  |  | |
| State Based | | | |  |  | |
| **Reflex** > **States** (search, MDPs, games) > **Variables** (CSP, Markov Networks, Bayesian Network)> **Logic**   * Search is powerful iff **well understood world states and actions**. So key skill is defining and decomposing problems into states * State is a summary of past actions sufficient to choose future actions optimally. State collapses tree into having only info that we use to choose future actions optimally to avoid the exponential blow ups * Search problem is an abstraction that provides a clean interface to the world to find optimality * Paradigm is the trifecta: modelling, learning (structured perceptron), inference | | | | Definition of search problem   1. Sstart Start state 2. Action(s): possible actions from states 3. Cost(s,a): cost of the action from state s 4. Succ(s,a): successor state from state s given action a 5. IsEnd(s): reached end state?   Search problem | | |
| **Tree search** Enumerating all states and actions but not done in practice so we build algorithms to help instantiate a search tree. Tree search is memory efficiency but exponential time complexity | | | | **2. DP**  DFS with reuse | | **3. UCS** The analog of DP for BFS |
| **1a. Exhaustive**  If b actions per state and max depth of D | **1b. DFS**  Key idea is backtracking search plus terminate when it finds first end state | **1c. BFS**  Key idea: explore all nodes in order of increasing depth  Space > DFS because @ lowest level have to remember the queue of nodes to explore | **1d. DFS-ID** | Definition of DP   1. Recurse 2. If already computed for s, return cached answer   Effect: recasts tree search problem as a DAG. If not DAG then DP breaks! | | 1. Expand sates close to the start 2. Use past-cost to re-use computation   Key idea: UCS enumerates sates in order of increasing past cost.  Implementation diff: UCS start to end, Djikstras all nodes |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | Exhaustive | DFS | BFS | DFS-ID | | Edge Cost | Any | 0 | ≥ 0 | ≥ 0 | | Time | O(bD) | O(bD) | O(bd) | O(bd) | | Space | O(D) which is small | O(D) | O(bD) > DFS | O(d) | | | | | |  |  |  | | --- | --- | --- | |  | DP | UCS | | Cycles | No | Yes | | Edge cost | Any | ≥ 0 | | Time | O(n) | O(nlogn) | | Space | ? | ? | | | |
| **4. A\*** Key idea is to distort edge costs to favour certain end states  A\* explores in order of past cost(s,a) and future cost (h(s))  Cost’(s,a) = cost(s,a) + h(succ(s,a)) –h(s)  Definition of h(s) is any estimate of future cost (s). If h(s) =0 means I’m on the optimal path. | | | |  | | |
| MDP | | | |  |  | |
| Relationship to search. Many similarities but one main one and one minue one. Major is the transition probabilities minor is changed from minimising cost to maximising rewards.   1. succ(s,a) becomes T(s,a,s’). Transition distribution for each state, s is . Introduced via the chance nodes. 2. cost(s,a) becomes Reward(s,a,s’). Reward may be positive or negative   Definitions   * Policy: A mapping from each state s \in States to an action a \in Actions * Utility: Following a policy yields a random path. The utility of a policy is the discounted sum of the rewards on the path. This is a random variable. * Path: s0 → a1,r1,s1 → a2,r2,s2 … (action, reward, new state) * Utility: r1 + r2 + r3 + … * Value: The value of a policy at a state is the expected utility. Also known as expected utility | | | | Definition   1. Sstart Start state 2. Action(s): possible actions from states 3. T(s,a,s’): transition prob of s’ by taking action a from state s 4. R(s,a,s’): reward for taking T(s,a,s’) 5. Succ(s,a): successor state from state s given action a 6. IsEnd(s): reached end state? 7. 0 ≤ ≤ 1: discount factor (default 1) | | |
| What is a solution for MDP?   * Policy * Utility   Policy evaluation: (MDP, Pi) -> V\_pi  Value iteration: MDP -> (Q\_opt, Pi\_opt) | | | |  | | |
|  | | | |  |  | |
| Reinforcement Learning | | | |  |  | |
| Bad news, if don’t know transition probability which algorithms can compute optimal policy? Unknown rewards?   * Model-based value iteration: estimate MDP Transitions and Rewards. * Model-based monte carlo: estimate Q\_opt directly. * Model-free monte carlo: estimates Q\_pi (on-policy) * SARSA: estimates Q\_pi * Q-Learning: estimates Q\_opt (off-policy)     On-policy: the estimate value is dependent on the policy pi used to generate the data.  Off-policy: | | | | Model-based MC    Model-free MC | | |
| Refinforcement learning algorithm template. For t = 1, 2, 3   * Choose action (how?) * Receive reward and observe new state * Update parameters (how?) | | | |  |  | |
| Week 6 Adversarial Games | | | |  |  | |
|  | | | |  |  | |
|  | | | |  |  | |
| Alpha beta pruning is a technique to minimise search of minimax   * When beta <= alpha then we break * The recursion is DFS. It evaluates the first leaf as a baseline. | | | |  |  | |
| Week 7 CSPs | | | | CSP 2: probabilistic CSPs, hidden markov model | | |
| Factor graphs   * Variables: * Values: the set of all values for each variable * Domain: Set of remaining values for each variable   + Value is an element in the domain   + Domain mutates whereas values don’t. It can reduce with each look ahead. * Factor vs. constraint: Factor is a weight. Constraint is a Boolean. * Constraints/factor: scope * Constraints/factor: expression * Number of consistent values = num of values that satisfies factors   + Most consistent var = var with most num of vals in domain   + Lease consistent variable = var with fewest num of vals in domain | | | |  | | |
| What does that union notation mean? x is the partial assignment of all the variables. | | | | 1. Beam search (most important algos for language. Used by google translate to hold K num of best translation solusions). Extends partial assignemnts    1. Exhaustive, K = inf is BFS so tie O(bn)    2. **Greedy** **K=1**: not global optimum but fast. DFS only one path for one partial assignment. K=1. Time = O(nb)    3. Beam search. Considers K candidates of partial assignments rather than one partial assignment. Beam size K controls tradeoff between efficiency and accuracy.    4. n variables (depth), branching factor: b = domain len, beam size = K, Time = O(nKblogK) 2. Local search. Modify complete assignments. 3. Gibbs sampling (probabilistic) 4. Conditional independence | | |
| Which variable to assign next? **Most constrained variable.**   1. Given a list of the remaining unassigned variable, which one to assign next? 2. Choose the variable that has the *fewest* consistent values, in other words, which variable has the fewest remaining options left in their domain? 3. Must assign **every** variable = fail early = more pruning. When to use? **When SOME factors are constraints.** | | | | Hints from section   * Valid assignment = multiplication of all factors is non-zero * find probability of CSP= multiplication of all factors / Z, where z is the sum of all factors for all values for all variables. * State the arity of each factor: the number of variables as parameters into the factor function. * Factors are always functions of the variables! F(X1, X2). * Indicator function needs a big 1 notation 1[function] * Domains can add a **null** to domain reflect no choice. Remember that scheduling appended [0] to list? * Which CSP technique will guarantee the maximum weight solution? Least constrained value, most constrained variable, iterated conditional modes, backtracking search. Answer: Backtracking only and MCV. | | |
| Dynamic Ordering. **Least constrained variable**   1. Order values of selected variable by decreasing number of consistent values of neighbouring variables. Need to choose **some** value most likely to lead to a solution. 2. **Use when ALL factors are constraints (CSPs). If some are preferences.** | | | |  |  | |
| Look ahead   * Forward checking (enforces arc consistency on neighbours). Need to actually prune domains to make heuristics useful. If constraints are long distance from each other or rely on many many variables no good! Look ahead: assign to current variable, then look @ its neighbours. Figure out what values would violate the factors and remove them from the domain of the set for that particular neighbour. Reason why it’s useful is because it allows you to exit early out of incompatible scenarios. If domain empty for a variable then terminate early * AC3 takes forward look ahead to the extreme by enforcing arc consistency on neighbours and their neighbours etc. Can solve large problems. Limitation: only looks at pairwise constraints (local structure) not global structures | | | |  |  | |
| Modelling: binary constraints easiest to reason about, lots of tools. Every n-ary CSP can be reduced to binary CSPs.   * Modelling NxN chess board HW6. Each queen assigned column. | | | |  |  | |
| **Week 7 Bayesian Networks** | | | |  |  | |
| Why?   * Handle heterogeneously missing information * Incorporate prior knowledge * Can interpret all the intermediate variables * Precursor to causal models | | | |  | | |
|  | | | | Parameter sharing  Laplace smoothing |  | |
|  | | | |  |  | |
|  | | | | Expectation Maximisation (EM). Generalisation of the Kmeans algorithm. H is hidden variables, e is observed   * E step is inference, un-normalised likelihoods or aka hallucinated counts * M step. Freeze hidden, estimate parameters | | |
|  | | | | Forward backwards algorithm   * User DP to calculate probabilities | | |