Search

Tree Search

procedure Tree Search($problem$, $strategy$)
  $fringe \leftarrow$ start state of $problem$
  while $fringe$ is not empty **do**
    $node \leftarrow$ Remove Front($fringe$, $strategy$)
    if $node$ has goal state **then**
      return solution
    else
      Insert children of $node$ into $fringe$
  return failure

Graph Search

procedure Graph Search($problem$, $strategy$)
  $closed \leftarrow$ empty set
  $fringe \leftarrow$ start state of $problem$
  while $fringe$ is not empty **do**
    $node \leftarrow$ Remove Front($fringe$, $strategy$)
    if $node$ has goal state **then**
      return solution
    if state($node$) not in $closed$ **then**
      Add state($node$) to $closed$
      Insert children of $node$ into $fringe$
  return failure

A* Heuristics

Admissible: For every state s, $h(s) \leq$ actual cost from s.
Consistent: For every arc (s_1, s_2), $h(s_1) - h(s_2) \leq$ cost from s_1 to s_2.

CSPs

Search problem with n variables X_i that must be assigned to values from domains D_i, subject to constraints $X_i \rightarrow X_j$. A state is a full assignment of values: has d^n states. Goal is to find an assignment to every variable that satisfies all constriants.

Backtracking Search

procedure Recursive Backtracking($assignment$, csp)
  if $assignment$ is complete **then**
    return $assignment$
  $var \leftarrow$ Select Unassigned Var(csp, $assignment$)
  for $value$ in Order Values(var, $assignment$, csp) **do**
    if $value$ is consistent with $assigment$ **then**
      add $\{var = value\}$ to $assignment$
      $result \leftarrow$ Recursive Backtracking($assignment$, csp)
      if $result$ is not failure **then return** $result$
      else remove $\{var = value\}$ from $assignment$
  return failure

MRV and LCV

MRV: Select Unassigned Var should choose the variable with fewest values remaining.
LCV: Order Values should choose the value that leaves the most values free in the future.

Forward Checking

procedure Forward Checking(csp, $assignment$)
  for every unassigned var in csp **do**
    remove values in var's domain that conflict with $assignment$

Arc Consistency

procedure Arc Consistency(csp)
  $arcs \leftarrow$ queue of all binary constraints
  while $arcs$ is not empty **do**
    Constraint $(X \rightarrow Y) \leftarrow$ pop($arcs$)
    for every value x in domain of X **do**
      if there is no y in domain of Y consistent with x **then**
        remove x from domain of X
    if values were removed from X **then**
      Insert all constraints $Z \rightarrow X$ into $arcs$

Tree-Structured CSPs

A tree-structured CSP can be solved in $O(nd^2)$ time with no backtracking.

procedure Solve Tree CSP(csp)
  Linearize the constraint graph
  for i from 2 to n **do**
    Enforce consistency of Parent(X_i) $\rightarrow X_i$
  for i from 1 to n **do**
    Assign X_i consistently with Parent(X_i)

Note: For undirected constraints, any level-order traversal is a linearization.
Enforcing arc consistency on a tree-structured CSP will always result in an empty domain if no solution exists with the current partial assignment.

Cutset Conditioning

Remove c variables from CSP such that remaining constraint graph is tree-structured. For *every possible assignment of cutset*, solve residual tree-structured CSP until solution is found. Runs in $O(d^c(n-c)d^2)$ time. The naive solution would be $O(d^n)$ worst-case.

Iterative Improvement

Begin with an assignment for every variable. Randomly select some variable X that violates a constraint, and reassign it to the value x that violates the fewest constraints.

Games

Multi-agent search problems with utilities for each agent at the leaves of the search tree.

Pruning

Don't consider nodes in game tree that are guaranteed not to change outcome. Alpha-beta max node code below:

procedure Max Value($state$, α, β)
  $v \leftarrow -\infty$
  for each successor of $state$ **do**
    $v \leftarrow \max(v,$ Value($successor$, α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow \max(\alpha, v)$

In general games, pruning may be possible if there are dependencies between utilities. Pruning with expectimax is possible if we can guarantee something about the ranges of chance node values. For prune tree pattern questions: node two levels above x must always have at least one fully returned child to be able to prune x.

MDPs

A problem characterized by states $s \in S$, actions $a \in A$, transition probabilities $T(s,a,s') = P(\text{moving to } s' \mid \text{took action } a \text{ from state } s)$, and rewards $R(s,a,s') =$ value of moving from s to s' through action a. Agents may have a time-discounting factor $0 < \gamma < 1$. Goal is to find a policy $\pi(s)\, \forall s \in S$ that will maximize expected sum of rewards from s.

Bellman Equations

EU of starting at state s and acting *optimally:*

$$V^*(s) \;=\; \max_{a \in A} \sum_{s' \in S} T(s,a,s') \left(R(s,a,s') +^{*}(s')\right) \;=\; \max_{a \in A} Q^*(s,a)$$

EU of starting at s, taking action a, and acting optimally:

$$Q^*(s,a) \;=\; \sum_{s' \in S} T(s,a,s') \left(R(s,a,s') + \gamma V^*(s')\right)$$

Value Iteration

Performs one step of expectimax. Initialize $V_0(s)$ arbitrarily for all $s \in S$. Then at each iteration, *for all* s,

$$V_{k+1}(s) \leftarrow \max_{a \in A} \sum_{s' \in S} T(s,a,s') \left(R(s,a,s') + \gamma V_k(s')\right)$$

Convergence is guaranteed if $\gamma < 1$ or MDP has finite horizon.

Policy Iteration

Start with an arbitrary policy π_0. For policy π_i, find values until convergence:

$$V^{\pi_i}_{k+1}(s) \leftarrow \sum_{s' \in S} T(s,\pi_i(s),s') \left(R(s,\pi_i(s),s') + \gamma V^{\pi_i}_k(s')\right)$$

Once V^{π_i} has converged, compute π_{i+1}:

$$\pi_{i+1}(s) \;=\; {}_{a \in A} \sum_{s' \in S} T(s,a,s') \left(R(s,a,s') + \gamma V^{\pi_i}(s')\right)$$

Learning

Agent is operating in an MDP where S and A are known, but T and R are not known and must be learned.

Direct Evaluation

Act according to policy π. Every time a state s is visited, write down what the eventual sum of rewards turned out to be when you stoped acting. For each s, average empirical sum of rewards over multiple trials.

TDL

Act according to π. Every time you start at s, move to s', and get a reward r, perform the update:

$$V^\pi(s) \;\leftarrow\; (1-\alpha)V^\pi(s) + \alpha\left(r + \gamma V^\pi(s')\right)$$

$0 < \alpha < 1$ is a learning rate parameter; small values privilege accumulated experience and large values privilege new sample: should decrease over time.

Q-Learning

Randomly choose actions at every state. Every time you start at s and move to s' with action a and reward r, perform the update:

$$Q(s,a) \; gets \; (1-\alpha)Q(s,a) + \alpha\left(r + \gamma \max_{a' \in A} Q(s',a')\right)$$

This will find the optimal policy even acting randomly. To limit regret, use an exploration plicy that efficiently explores unknown Q values until you have a good idea what they are.

Feature-Based

Describe states or Q-states as a vector of real-valued *features* f_i. Perform update not on state, but on the weights w_i we assign to feature f_i:

$$Q(s,a) \;=\; \sum_i w_i\, f_i(s,a)$$

Suppose you move from s to s' through action a and get reward r. Perform update:

$$d \;=\; \left(r + \gamma \max_{a' \in A} Q(s',a')\right) \;-\; Q(s,a)$$

$$\forall i,\;\; w_i \leftarrow w_i + \alpha d f_i(s,a)$$

Weights capture whether feature is good or bad (sign) and how important it is (magnitude). Since $f_i(s,a) = 0$ if feature is not present, weights only get updated for active features.