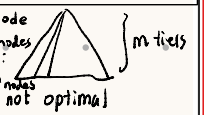
Tree Search

```
func TREE-SEARCH (problem, frontier) return a solution or failure
    frontier ← INSERT (MAKE-NODE (INITIAL-STATE[problem]), frontier)
    while not IS-EMPTY(frontier) do
        node ← POP(frontier)
        if problem.IS-GOAL(node.STATE) then return node
        for each child-node in EXPAND(problem, node) do
            add child-node to frontier
    return frontier
```

Depth-First-Search (DFS)



Completeness: maybe not complete.
Optimality: find the leftmost, might not optimal
Time complexity: $O(b^m)$ Space Complexity: $O(bm)$

Breadth-First-Search (BFS)

Completeness: Complete
Optimality: not optimal
Time complexity: $O(b^s)$ Space Complexity: $O(b^s)$

Uniform Cost Search

Completeness: complete.
Optimality: optimal
Time complexity: $O(b^{\frac{C^*}{\varepsilon}})$ C^* is the optimal path cost
minimal cost between 2 nodes: ε
space complexity: $O(b^{\frac{C^*}{\varepsilon}})$

Admissibility, $\forall n,\ 0 \le h(n) \le h^*(n)$

Theorem: for a given search problem, if the admissibility constraint is satisfied by a heuristic function h, using A* search with h will yield an optimal solution.

```
function A*-Graph-SEARCH (problem, frontier)
    reached ← an empty dic mapping nodes to the cost to each one
    frontier ← INSERT (MAKE-NODE (INITIAL-STATE[problem]), frontier)
    while not IS-EMPTY(frontier) do
        node, node.costToNode ← POP(frontier)
        if problem.IS-GOAL(node.STATE) then return node
        if node.STATE is not in reached or reached[node.STATE] > node.costToNode then
            reached[node.STATE] = node.costToNode
            for each child-node in EXPAND(problem, node) do
                frontier ← INSERT((child-node, child-node.costToNode), frontier)
    return failure
```

Dominance: If heuristic is dominant over heuristic b, then the estimated goal distance for a is greater than the estimated goal distance for b for every node in state space.
$\forall n.\ h_a(n) \ge h_b(n)$

BACKTRACKING

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING({}, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment given CONSTRAINTS[csp] then
            add {var = value} to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result
            remove {var = value} from assignment
    return failure
```

OPTIMIZATION 1. Filtering (Forward checking)

```
function AC-3(csp) returns the CSP, possibly with reduced domains
    inputs: csp, a binary CSP with variables {X1, X2, ..., Xn}
    local variables: queue, a queue of arcs, initially all the arcs in csp
    while queue is not empty do
        (Xi, Xj) ← REMOVE-FIRST(queue)        (tail, head)
        if REMOVE-INCONSISTENT-VALUES(Xi, Xj) then
            for each Xk in NEIGHBORS[Xi] do
                add (Xk, Xi) to queue

function REMOVE-INCONSISTENT-VALUES(Xi, Xj) returns true iff succeeds
    removed ← false
    for each x in DOMAIN[Xi] do
        if no value y in DOMAIN[Xj] allows (x,y) to satisfy the constraint Xi ↔ Xj
            then delete x from DOMAIN[Xi]; removed ← true
    return removed
```

if tree structured with topo: $O(nd^2)$.

AC-3 has a worst-case time complexity of $O(ed^3)$

OPTIMIZATION 2. Ordering

MRV (Minimum Remaining Values) (choose variable)
LCV (Least-Constraining Values) (choose value)
Topological Sort

Minimax

- $\forall$ agent-controlled states, $V(s) = \max_{s' \in \text{successors}(s)} V(s')$
- $\forall$ opponent-controlled states $V(s) = \min_{s' \in \text{successors}(s)} V(s')$
- $\forall$ terminal states $V(s) = \text{known}$

```
def value(state):
    if the state is a terminal state: return the state's utility
    if the agent is MAX: return max-value(state)
    if the agent is MIN: return min-value(state)

def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v

def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor))
    return v
```

Alpha-Beta Pruning

α: MAX's best option on path to root
β: MIN's best option on path to root

```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v

def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v
```

EXPECTIMAX

```
def value(state):
    if the state is a terminal state: return the state's utility
    if the agent is MAX: return max-value(state)
    if the agent is EXP: return exp-value(state)

def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v

def exp-value(state):
    initialize v = 0
    for each successor of state:
        p = probability(successor)
        v += p * value(successor)
    return v
```

Markovianess

$$P(S_{t+1} = s_{t+1} \mid S_t = s_t, A_t = a_t, S_{t-1} = s_{t-1}, A_{t-1} = a_{t-1} \cdots S_0 = s_0) = P(S_{t+1} = s_{t+1} \mid S_t = s_t, A_t = a_t)$$

$$T(s, a, s') = P(s' \mid s, a)$$

Bellman Equation

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

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

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

Value Iteration

$$\forall s \in S,\ V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_k(s')]$$

when convergence is reached,

$$V_k(s) = V_{k+1}(s) = V^*(s)$$

Policy Extraction

$$\forall s \in S,\ \pi^*(s) = \arg\max_a Q^*(s,a) = \arg\max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s)]$$

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

$$V^{\pi}(s) = \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V^{\pi}(s')]$$

$$V_{k+1}^{\pi_i}(s) \leftarrow \sum_{s'} T(s, \pi_i(s), s')[R(s, \pi_i(s), s') + \gamma V_k^{\pi_i}(s')]$$

$$\pi_{i+1}(s) = \arg\max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^{\pi_i}(s')]$$

Reinforcement Learning

- Model-based Learning $\Rightarrow T(s,a,s')$
- Model-Free Learning. $= \frac{\#(s,a,s')}{\#(s,a)}$

Temporal Difference Learning

$$V^{\pi}(s) = \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V^{\pi}(s')]$$

$$\text{Sample} = R(s, \pi(s), s') + \gamma V^{\pi}(s')$$

$$V^{\pi}(s) \leftarrow (1-\alpha) V^{\pi}(s) + \alpha \cdot \text{sample}$$

$$V_k^{\pi}(s) \leftarrow (1-\alpha) V_{k-1}^{\pi}(s) + \alpha \cdot \text{Sample}_k$$

The recurrence definition of $V_k^{\pi}(s)$:

$$V_k^{\pi}(s) \leftarrow \alpha \cdot [(1-\alpha)^{k-1} \text{sample}_1 + \cdots + (1-\alpha) \text{Sample}_{k-1} + \text{Sample}_k]$$

Q-Learning

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

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

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

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

Approximate Q-Learning

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

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

where $\vec{f}(s) = [f_1(s)\ f_2(s) \cdots f_n(s)]^T$

and $\vec{f}(s,a) = [f_1(s,a)\ f_2(s,a) \cdots f_n(s,a)]^T$

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

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

$$Q(s,a) \leftarrow Q(s,a) + \alpha \cdot \text{difference}$$

ε-greedy

$0 \le \varepsilon \le 1$. if ε is large, prefer randomly
if ε is small prefer learn by slowly

Exploration Functions

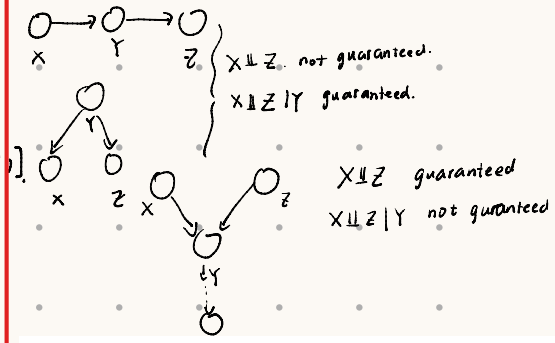
$$Q(s,a) \leftarrow (1-\alpha) Q(s,a) + \alpha [R(s,a,s') + \gamma \max_{a'} f(s',a')]$$

$$f(s,a) = Q(s,a) + \frac{\kappa}{N(s,a)}$$

Bayesian Network

$$P(X_1, X_2 \cdots X_n) = \prod_{i=1}^{n} P(X_i \mid \text{parents}(X_i))$$

Casual Chains



1. Shade all observed nodes $\{Z_1, \ldots Z_k\}$ in the graph.
2. Enumerate all undirected paths from X to Y.
3. For each path:
   a. Decompose the path into triples (segments of 3 nodes).
   b. If all triples are active, this path is active and *d-connects* X to
4. If no path d-connects X and Y, then $X \perp\!\!\!\perp Y \mid \{Z_1, \ldots Z_k\}$.

At convergence · obtain an optimal policy.

$$\log l(w_1 \cdots w_k) = \sum_{i=1}^{n} \sum_{k=1}^{k} t_{i,k} \log \left(\frac{e^{w_k^T f(x_i)}}{\sum_{l=1}^{k} e^{w_l^T f(x_i)}} \right)^{t_{i,k}}$$

$$\log l(w_1 \cdots w_k) = \sum_{i=1}^{n} \sum_{k=1}^{k} t_{i,k} \log \left(\frac{e^{w_k^T f(x_i)}}{\sum_{l=1}^{k} e^{w_l^T f(x_i)}} \right)$$

$$\nabla_{w_j} \log l(w) = \sum_{i=1}^{n} \nabla_w \sum_{k=1}^{k} t_{i,k} \log(\dots)$$

$$= \sum_{i=1}^{n} \left(t_{i,j} - \frac{e^{w_j^T f(x_i)}}{\sum_{l=1}^{k} e^{w_l^T f(x_i)}} \right) f(x_i)$$

- [x] Model-based learning of $T(s, a, s')$ and $R(s, a, s')$.
- [ ] Direct Evaluation to estimate $V(s)$.
- [ ] Temporal Difference learning to estimate $V(s)$.
- [x] Q-Learning to estimate $Q(s, a)$.

1

(ii) In the limit of infinite timesteps, under which of the following exploration policies is Q-learning guaranteed to converge to the optimal Q-values for all state? (You may assume the learning rate α is chosen appropriately, and that the MDP is ergodic: i.e., every state is reachable from every other state with non-zero probability.)

- [x] A fixed policy taking actions uniformly at random.
- [ ] A greedy policy.
- [x] An ϵ-greedy policy
- [ ] A fixed optimal policy.

heuristic

graph search: must consistent

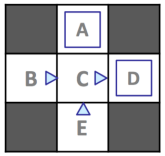
tree search: can be admissible

CNF & DNF

CNF: And of ORs : $(A \vee B) \wedge (C \vee D)$

DNF: OR of ANDS: $(A \wedge B) \vee (C \wedge D)$

Direct Evaluation



Episode 1	Episode 2
B, east, C, -1 C, east, D, -1 D, exit, x, +10	B, east, C, -1 C, east, D, -1 D, exit, x, +10
Episode 3	**Episode 4**
E, north, C, -1 C, east, D, -1 D, exit, x, +10	E, north, C, -1 C, east, A, -1 A, exit, x, -10

Walking through the first episode, we can see that from state D to termination we acquired a total reward of 10, from state C we acquired a total reward of $(-1) + 10 = 9$, and from state B we acquired a total reward of $(-1) + (-1) + 10 = 8$. Completing this process yields the total reward across episodes for each state and the resulting estimated values as follows:

s	Total Reward	Times Visited	$V^{\pi}(s)$
A	-10	1	-10
B	16	2	8
C	16	4	4
D	30	3	10
E	-4	2	-2

Multi-Class Logistic Regression

$$P(y = i \mid f(x); w) = \frac{e^{w_i^T f(x)}}{\sum_{k=1}^{k} e^{w_k^T f(x)}}$$

$$l(w_1 \cdots w_k) = \prod_{i=1}^{n} P(y_i \mid f(x_i); w)$$

$$= \prod_{i=1}^{n} \prod_{k=1}^{k} \left(\frac{e^{w_k^T f(x_i)}}{\sum_{l=1}^{k} e^{w_l^T f(x_i)}} \right)^{t_{i,k}}$$

Neural Networks

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

Sometimes, we want an output more expressive than a binary label. It then becomes useful to produce a probability for each of the (N) classes, reflecting our degree of certainty that the data point belongs to each class. As in multi-class logistic regression, we store a weight vector for each class (j), and estimate probabilities with the softmax function:

$$\sigma(\mathbf{x}_i)_j = \frac{e^{\mathbf{f}(\mathbf{x}_i)^T \mathbf{w}_j}}{\sum_{\ell=1}^{N} e^{\mathbf{f}(\mathbf{x}_i)^T \mathbf{w}_\ell}} = P(y_i = j | \mathbf{f}(\mathbf{x}_i); \mathbf{w}).$$

Given a vector output by $\mathbf{f}$, softmax normalizes it to output a probability distribution. To derive a general loss function for our models, we can use this probability distribution to generate an expression for the likelihood of a set of weights:

$$\ell(\mathbf{w}) = \prod_{i=1}^{n} P(y_i | \mathbf{f}(\mathbf{x}_i); \mathbf{w}).$$

This expression denotes the likelihood of a particular set of weights explaining the observed labels and data points. We seek the set of weights that maximizes this quantity. This is equivalent to finding the maximum of the log-likelihood expression:

$$\log \ell(\mathbf{w}) = \log \prod_{i=1}^{n} P(y_i | x_i; \mathbf{w}) = \sum_{i=1}^{n} \log P(y_i | \mathbf{f}(\mathbf{x}_i); \mathbf{w}).$$

Step function

$$f(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad \text{sigmoid}$$

$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases} \quad \text{ReLU}$$

Loss Functions and Multiplicative Optimization

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