# Introduction to Artificial Intelligence

Daniel Deng

## 1 Agents

**Definition 1.1** (Reflex Agent)**.** A reflex agent chooses actions based on its current perception of the world.

**Definition 1.2** (Planning Agent)**.** A planning agent chooses actions based on hypothesized consequences of actions.

# 2 General Search Problems

## 2.1 Heuristics

**Definition 2.1** (Heuristic)**.** A heuristic $h(n)$ is a function that estimates the distance from state $n$ to the goal state for a particular search problem. It is often solutions of relaxed problems.

1. A heuristic is <u>admissible</u> if $0 \leq h(n) \leq h^*(n)$ where $h^*$ is the true cost to goal state;
2. A heuristic is <u>consistent</u> if $h(n) - h(n+1) \leq c(n, n+1)$ where $c$ is the cost between states $n$ and $n+1$.

*Remark.* Consistency necessarily implies admissibility.

## 2.2 Search Algorithms

Table 1: Search algorithms.

| | Fringe | Complete | Optimal | Time | Space |
|---|---|---|---|---|---|
| Depth-First Search | Stack | *iff* no cycle | No | $O(b^m)$ | $O(bm)$ |
| Breadth-First Search | Queue | Yes | *iff* uniform cost | $O(b^s)^1$ | $O(b^s)^1$ |
| Uniform Cost Search | PQ $(g(n))^2$ | *iff* positive cost | Yes | $O(b^{c^*/\epsilon})^3$ | $O(b^{c^*/\epsilon})^3$ |
| Greedy Search | PQ $(h(n))$ | - | No | - | - |
| $A^*$ Tree Search | PQ $(h(n) + g(n))$ | - | *iff* $h(n)$ admissible | - | - |
| $A^*$ Graph Search[4] | PQ $(h(n) + g(n))$ | - | *iff* $h(n)$ consistent | - | - |

[1] $s$ = depth of solution.
[2] $g(n)$ = cumulative path cost.
[3] $c^*/\epsilon$ = effective solution depth ($c^*$ = cost of the cheapest solution; $\epsilon$ = minimum cost of cost-contour arcs).
[4] Compared to tree search, graph search keeps a closed set of expanded states to check against to prevent duplicate expansions.

*Remark.* Implementation of search algorithms differ only in fringe strategies.

# 3 Constrained Satisfaction Problems

**Definition 3.1** (Constrained Satisfaction Problems)**.** Constrained Satisfaction Problems (CSPs) are a type of **identification problem** defined by variable $X_0, \ldots, X_n$ with values from a domain $D$ that satisfies a set of constrains.

**Algorithm 1:** Backtracking search.

**Input:** A constraint satisfaction problem $P$.
**Output:** A complete assignment $A$.

```
Function BS(P):
    return EXPLORE (P, {})
Function EXPLORE(P, A):
    if A is complete then
        return A
    unassigned ← an unassigned VARIABLES(P)
    foreach value ∈ DOMAIN(unassigned) do
        if value is consistent with all CONSTRAINTS(P) then
            add {unassigned ← value} to A
            attempt ← EXPLORE(P, A)
            if attemp failed then
                remove {unassigned ← value} from A
            else
                return attempt
    return failed
```

## 3.1 Filtering

**Definition 3.2** (Arc Consistency)**.**

Arc $X \to Y$ is consistent $\Leftrightarrow$

$$(\forall x \in D_x)(\exists y \in D_y)(y \text{ can be assigned to } Y \text{ without violating a constraint.})$$

**Algorithm 2:** Arc consistency filtering.

**Input:** A constraint satisfaction problem $P$.

```
/* O(|S||A|^3) runtime                                   */
Function AC-3(P):
    Q ← empty Queue
    enqueue all arcs ∈ P
    while Q is not empty do
        (X_i, X_j) ← dequeue from Q
        if FILTER(X_i, X_j) is successful then
            foreach X_k ∈ NEIGHBORS(X_i) do
                enqueue (X_k, X_i)

Function FILTER (tail, head):
    result ← false
    foreach value ∈ DOMAIN(tail) do
        if value violates some constraint with all values in DOMAIN(head) then
            delete value from DOMAIN(tail)
            result ← true
    return reuslt
```

## 3.2 Ordering

**Definition 3.3** (Minimum Remaining Values)**.** The MRV policy chooses an unassigned variable that has the fewest valid remaining values in order to induce backtracking earlier and reduce potential node expansions.

**Definition 3.4** (Least Constraining Value)**.** The LCV policy chooses a value assignment that violates the least amount of constraints, which requires additional computation such as running arc consistency test on each value.

### 3.3 Structure

Given a tree-structured CSP, represent it as a directed acyclic graph. Enforcing arc consistency in reverse topological order then assigning in topological order ensures a runtime of $O(nd^2)$ (as opposed to $O(d^n)$ in the general case).
TODO: nearly tree-like CSPs and tree decomposition.

# 4 Local Search

**Definition 4.1** (Local Search)**.** A search strategy that improve a single option until no further improvements can be made. Typically, local search is faster and more memory efficient than other search algorithms but is generally neither complete nor optimal.

**Definition 4.2** (Hill Climbing)**.** A CSP strategy that randomly selects a conflicting variable and reassign values using min-conflicts heuristics.

*Remark.* Efficiency of the algorithm depends on $R = \frac{\text{number of constraints}}{\text{number of variables}}$; computation time is approximately constant time except when $R$ approaches the *critical ratio.*

## 4.1 Simulated Annealing

**Algorithm 3:** Simulated annealing.

**Input:** A problem $P$ and a schedule/mapping from time to ”temperature” $T$.
**Output:** A solution state.

```
/* Escape local maxima by allowing downhill movement based on a
   "temperature"-dependent probabilistic function.              */
current ← initial state of P
for t ← 1 to ∞ do
    temp ← T[t]
    if temp = 0 then
        return current
    else
        next ← a randomly selected successor of current
        Δ ← VALUE[next] - VALUE[current]
        if Δ > 0 then
            current ← next
        else
            currnet ← next with probability e^(Δ/temp)
```

## 4.2 Genetic Algorithm

**Definition 4.3** (Genetic Algorithm)**.** A strategy that keeps the best $N$ hypothesis at each step based on a fitness function and generate next generation using pairwise cross-over operations (and, optionally, mutation operations).

# 5 Games

**Definition 5.1** (Games)**.** Games are multi-agent search problems that could be zero-sum (adversarial) or general sum.

## 5.1 Minimax

**Definition 5.2** (Minimax)**.** A zero-sum-game algorithm that assumes the opponent is an optimal adversary.

**Algorithm 4:** Minimax with alpha-beta pruning.

```
Input: A game state S
Output: The root minimax value.
/* initialize α ← −∞ and β ← ∞                                  */
Function VALUE(S, α, β):
    if S is a terminal state then
        return knwon temrinal value
    if the agent is maximizing then
        return MAX-VALUE(S, α, β)
    if the agent is minimizing then
        return MIN-VALUE (S, α, β)
Function MAX-VALUE(S, α, β):
    v ← −∞
    foreach successor S′ of S do
        v ← MAX(v, VALUE(S′))
        if v ≥ β then
            return v
        α ← MAX(α, v)
    return v
Function MIN-VALUE(S, α, β):
    v ← ∞
    foreach successor S′ of S do
        v ← MIN(v, VALUE(S′))
        if v ≤ α then
            return v
        β ← MIN(β, v)
    return v
```

### 5.2 Expectimax

**Definition 5.3** (Expectimax)**.** A zero-sum-game algorithm that assumes the opponent acts based on some probability distribution.

**Algorithm 5:** Expectimax.

```
Input: A game state S
Output: The root minimax value.
Function VALUE(S):
    if S is a terminal state then
        return knwon temrinal value
    if the agent is maximizing then
        return MAX-VALUE(S)
    if the agent is randomizing then
        return EXP-VALUE(S)
Function MAX-VALUE(S):
    v ← −∞
    foreach successor S′ of S do
        v ← MAX(v, VALUE(S′))
    return v
Function EXP-VALUE(S):
    v ← 0
    foreach successor S′ of S do
        v ← 𝔼[VALUE(S′)] // expected value
    return v
```

# 6 Markov Decision Processes

*Remark.* Finite horizons (finite timestep before an agent terminates) and/or discount factors ($\gamma$) ensure an agent terminates in MDP.

**Definition 6.1** (Transition Function)**.** $T(s, a, s') = P(s' \mid s, a)$

## 6.1 Value Iteration

**Algorithm 6:** Value Iteration

**Input:** A MDP $(S, A, R, T, \gamma)$.
**Output:** The optimal policy $\pi^*(s)$ for all $s \in S$.

**Function** `POLICY-EXTRACTION():`
  $V^* \leftarrow$ `VALUE-ITERATION() // optimal value`
  **foreach** $s \in S$ **do**
    $\pi^*(s) \leftarrow \underset{a}{\operatorname{argmax}} \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V_k(s')]$
  **return** $\pi^*(s) \mid \forall s \in S$

**Function** `VALUE-ITERATION():`
  `//` $O(|S|^2|A|)$ `runtime`
  Initialize $V_0(s) \leftarrow 0$ for all $s \in S$
  **while** $V_{k+1}(s) \neq V_k(s) \mid \forall s \in S$ **do**
    `// repeat until values converge`
    $V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V_k(s')]$
  **return** $V^*(s) \leftarrow V_{k+1}(s) \mid \forall s \in S$

## 6.2 Policy Iteration

**Definition 6.2** (Policy Iteration)**.** Define an initial policy $\pi_0$ (can be arbitrary, but ideality close to the optimal policy). Then, iteratively solve $\forall s \in S$

$$V^{\pi_i}(s) = \sum_{s'} T(s, \pi_i(s), s')[R(s, \pi_i(s), s') + \gamma V^{\pi_i}(s')] \qquad \text{(policy evaluation)}$$

$$\pi_{i+1}(s) \leftarrow \underset{a}{\operatorname{argmax}} \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V^{\pi_i}(s')] \qquad \text{(policy improvement)}$$

until $\pi(s)$ converges for all $s \in S$ (yields $pi^*$.

*Remark.* Policy evaluation solves a system of $|S|$ liear equations.

*Remark.* Policy iteration converges faster than value iteration.

# 7 Reinforcement Learning

*Remark.* Reinforcement learning operates on MDP problems where the $T$ and $R$ functions are unknown.

## 7.1 Model-Based Learning

**Definition 7.1** (Model-Based RL)**.** An algorithm that counts and normalizes sample outcomes $s'$ for each $s, a$ to construct $\hat{T}(s, a, s')$ and discovers each $\hat{R}(s, a, s')$ through exploration. The approximated MDP is then solved by value or policy iteration.

## 7.2 Model-Free Learning

**Definition 7.2** (Direct Evaluation)**.** An algorithm that fixes some policy $\pi$ and empirically computes $V^\pi(s)$ for all $s \in S$ by averaging total sample utility for a given state.

*Remark.* Direct evaluation wastes information about state connections and will take a long time to learn.

**Definition 7.3** (Temporal Difference Learning)**.**

$$V^\pi(s) \leftarrow (1 - \alpha)V^\pi(s) + \alpha[R(s, \pi(s), s') + \gamma V^\pi(s')]$$

*Remark.* Can't extract policy.

**Definition 7.4** (Q-Learning)**.**

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

*Remark.* Q-learning is an example of off-policy learning, in which the algorithm can learn the optimal policy even by taking sub-optimal or random actions.

**Definition 7.5** (Approximate Q-Learning)**.** Represent q-values as weighted sums of features $Q(s, a) := \vec{w} \cdot \vec{f}(s, a)$. The update rule for Q-Learning then becomes

$$\begin{aligned} \Delta \leftarrow [R'(s, a, s') + \gamma \max_{a'} Q(s', a')] - Q(s, a) \\ w_i \leftarrow w_i + \alpha \cdot \Delta \cdot f_i(s, a) \end{aligned}$$

**Definition 7.6** ($\epsilon$-Greedy Policies)**.** Define some probability $0 \leq \epsilon \leq 1$ to act randomly and explore. $\epsilon$ should be lowered over time to favor more exploitation as the learning becomes complete.

**Definition 7.7** (Exploration Function)**.**

$$\begin{aligned} Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha[R(s, a, s') + \gamma \max_{a'} f(s', a')] \\ f(s, a) = Q(s, a) + \tfrac{\text{const.}}{\text{count}(Q(s,a))} \end{aligned}$$

*Remark.* $f(s, a)$ shown here is only a common example where the "bonus" for exploration diminishes as q-states are explored more and more.

# 8 Bayesian Network

**Definition 8.1** (Conditional Independence)**.** $\forall x, y, z : P(x|z, y) = P(x|z)$ (i.e., $x \perp\!\!\!\perp y \mid z$)

**Definition 8.2** (Bayes' Net)**.** Bayes' Net is a graphic model (DAG) that describes complex joint distributions using simple, local distributions (conditional probabilities)

- Nodes—variables (with domains) [assigned = observed, unassigned = unobserved]
- Arcs—interactions (i.e., encode conditional independence when lacking arrows) $\implies$ $P(x_i|x_1, \dots, x_{i-1}) = P(x_i|parents(X_i))$
- A conditional distribution for each node

*Remark.* Bayes' Net does not imply causation, only the lack thereof

## 8.1 Inference

**Definition 8.3** (Inference by Enumeration)**.** General procedure:

1. Join Factors: $\forall r, t : P(r, t) = P(r)P(t|r)$
2. Eliminate: $P(T) = \sum_r P(r, T)$
3. Normalize: $P(Q|e_1, \dots, e_k) = \frac{P(Q, e_1, \dots, e_k)}{\sum_q P(Q, e_1, \dots, e_k)}$

*Remark.* Variable elimination is faster than inference by enumeration via interleaving joining and marginalizing.

## 8.2 Sampling

## 8.3 D-Separation

**Definition 8.4** (Causal Chains)**.** $X \to Y \to Z$

$$P(x, y, z) = P(x)P(y \mid x)P(z \mid y) \implies X \perp\!\!\!\perp Z \mid Y$$

**Definition 8.5** (Common Causes)**.** $X \leftarrow Y \to Z$

$$P(x, y, z, ) = P(y)P(x \mid y)P(z \mid y) \implies X \perp\!\!\!\perp Z \mid Y$$

**Definition 8.6** (Common Effect)**.** $X \to Z \leftarrow Y \implies X \perp\!\!\!\perp Z$

**Definition 8.7** (D-Separation)**.** $X \perp\!\!\!\perp Y \mid Z$ *iff* $X$ and $Y$ are "d-separated" by $Z$ (i.e., all undirected path from $X$ to $Z$ are inactive). A path is active if all triple it contains is active:

- Causal chain $A \to B \to C$ where $B$ is unobserved
- Common cause $A \leftarrow B \to C$ where $B$ is unobserved
- Common effect $A \to B \leftarrow C$ where $B$ or one of its descendants is observed

**Algorithm 7:** Sampling.

```
Function PRIOR-SAMPLING:
    for i = 1, 2, ..., n in topological order do
        Sample x_i from P(X_i | Parents(X_i))
    return (x_1, x_2, ..., x_n)
Function REJECTION-SAMPLING:
    // Will reject lots of samples if evidence is unlikely.
    for i = 1, 2, ..., n in topological order do
        Sample x_i from P(X_i | Parents(X_i))
        if x_i inconsistent with evidence then
            return // no sample generated
    return (x_1, x_2, ..., x_n)
Function LIKELIHOOD-WEIGHTING-SAMPLING:
    w ← 1.0
    for i = 1, 2, ..., n in topological order do
        if X_i is an evidence variable then
            X_i ← observation x_i from X_i
            w ← w × P(x_i | Parents(X_i))
        else
            Sample x_i from P(X_i | Parents(X_i))
    return (x_1, x_2, ..., x_n), w
Function GIBBS-SAMPLING:
```