Intelligent Agents

PEAS Framework
- **P**erformance Measure
- **E**nvironment
- **A**ctuators
- **S**ensors

A **rational agent** will choose actions that maximize performance measure.

Properties of Task Environment

1. **Fully Observable** (vs. **Partially Observable**)
   - Agent has full environmental awareness
2. **Deterministic** (vs. **Stochastic**)
   - Next environment state is only determined by agent's action
   - **Strategic:** If next environment state also depends on other agents
3. **Episodic** (vs. **Sequential**)
   - Agent's actions depend solely on individual episodes.
4. **Static** (vs. **Dynamic**)
   - Environment unchanged during agent deliberation; semi-dynamic if only performance score changes.
5. **Discrete** (vs. **Continuous**)
   - Finite discrete actions
6. **Single Agent** (vs. **Multi-agent**)

Structure of Agents

Agent Function:
- completely specifies an agent.
- is <u>sufficient</u> to define an agent.

Common Agent Structures:
- Simple reflex agents
- Model-based reflex agents
- Goal-based agents
- Utility-based agents
- Learning agents

Increasing Complexity

Exploration vs Exploitation

Exploration: Discover new strategies or information that may lead to better long-term outcomes.

Exploitation: Maximize immediate rewards using known information.

Problem Formulation

- State representation
- Initial state
- Goal state / Goal test
- Actions
- Transition model
- Action cost function

State Representation Invariant

Abstract states (in the problem model) must have corresponding concrete states (in the real world).

Uninformed Search Algorithms

Breadth-first Search (BFS)

b: branching factor
d: depth of solution

- Frontier: Queue
- Time Complexity: $O(b^d)$
- Space Complexity: $O(b^d)$
- Complete?: Yes, if b is finite
- Optimal?: Yes, if step cost is same everywhere

Uniform-cost Search (UCS)

ϵ: minimum edge cost
C^*: cost of optimal solution

- Frontier: Priority queue
- Time Complexity: $O\left(b^{\frac{C^*}{\epsilon}}\right)$
- Space Complexity: $O\left(b^{\frac{C^*}{\epsilon}}\right)$
- Complete?: Yes, if $\epsilon > 0$ and C^* finite
- Optimal?: Yes, if $\epsilon > 0$
  - $\epsilon = 0$ may cause zero cost cycle

Depth-first Search (DFS)

m: maximum depth

- Frontier: Stack
- Time Complexity: $O(b^m)$
- Space Complexity: $O(bm)$
- Complete?: No, when depth infinite or loops present
- Optimal?: No

Depth-limited Search (DLS)

l: depth limit of search

- Backtrack when limit is hit
- Time Complexity: $O(b^l)$
- Space Complexity: $O(bl)$
- Complete?: No
- Optimal?: No, if used with DFS

Iterative Deepening Search (IDS)
- Time Complexity: $O(b^d)$
- Space Complexity: $O(bd)$, if used with DFS
- Complete?: Yes
- Optimal?: Yes, if step cost is same everywhere

Informed Search Algorithms (1)

Best-first Search
- Like UCS, but uses an evaluation function instead of step cost

Greedy Best-first Search
- Evaluation function: Heuristic (cost of n to goal)
- Time/space complexity, completeness and optimality same as DFS

Informed Search Algorithms (2)

A* Search
- Evaluation function: Cost to reach n + Heuristic
- Time/space complexity: Same as DFS
- Complete?: Yes
- Optimal?: Depends on heuristic
- Heuristic = 0 → UCS

Admissible Heuristic

$$\forall n\ \left(h(n) \le h^*(n)\right)$$

$h^*(n)$: true cost

Admissible heuristic never over-estimates the cost to reach the goal.

A* using **tree search** is optimal.

Consistent Heuristic

$$\forall n\ \left(h(n) \le c(n,a,n') + h(n')\right) \land \mathrm{h(G)} = 0$$

$c(n,a,n')$: cost of n to n'
n': a successor of n

Consistent heuristic is non-decreasing along any path.

A* using **graph search** is optimal.

Dominance

$$\forall n\ (h_2(n) \ge h_1(n) \to h_2\ dominates\ h_1)$$

If h_2 is admissible, it is better for search.

"Inventing" Admissible Heuristics

- A problem with fewer restrictions on the actions is called a relaxed problem.
- **The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem.**

Local Search

Type of problems:
- Very large state space
- Good enough solution better than no solution

Local Search Features:
- Given time constraint, outputs a good enough solution rather than no solution
- Search path irrelevant, **state is the solution**

Hill Climbing

- Pick best among neighbours until no better neighbour

Simulated Annealing

- Method to escape local minima
- Decrease temperature (randomness) slowly

```
current = initial state
T = a large positive value
while T > 0:
    next = a randomly selected successor of current
    if value(next) > value(current): current = next
    else with probability P(current, next , T): current = next
    decrease T
return current
```

Allow "bad moves" from time to time

$$\mathrm{P(current,\ next,\ T)} = e^{\frac{value(next) - value(current)}{T}}$$

- If T decreases slowly enough, high probability to find global optimum.

Adversarial Search

Properties:
- Fully observable
- Deterministic
- Discrete
- Terminal states exist (no infinite runs)
- Two-player zero-sum
- Turn-taking

Minimax

```
def minimax(state):
    v = max_value(state)
    return action in successors(state) with value v

def max_value(state):
    if is_terminal(state): return utility(state)
    v = -∞
    for action, next_state in successors(state):
        v = max(v, min_value(next_state))
    return v

def min_value(state):
    if is_terminal(state): return utility(state)
    v = ∞
    for action, next_state in successors(state):
        v = min(v, max_value(next_state))
    return v
```

Assumes opponent play **optimally**: trying to **minimize** player's value

- Time Complexity: $O(b^m)$
- Space Complexity: $O(bm)$ with DFS
- Complete?: Yes, if tree finite
- Optimal?: Yes, against optimal opponent

Alpha-beta Pruning

```
def alpha_beta_search(state):
  v = max_value(state, -∞, ∞)
  return action in successors(state) with value v

def max_value(state, α, β):
  if is_terminal(state): return utility(state)
  v = -∞
  for action, next_state in successors(state):
    v = max(v, min_value(next_state, α, β))
    α = max(α, v)
    if v >= β: return v
  return v

def min_value(state , α, β):
  if is_terminal(state): return utility(state)
  v = ∞
  for action, next_state in successors(state):
    v = min(v, max_value(next_state, α, β))
    β = min(β, v)
    if v <= α: return v
  return v
```

Supervised Learning

Regression: Predict continuous output
Classification: Predict discrete output

Performance Measure

For Regression:

- Mean Squared Error

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2$$

- Mean Absolute Error

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|\hat{y}_i - y_i|$$

For Classification:

- Correctness
- Accuracy (Average correctness)

$$Accuracy = \frac{1}{N}\sum_{i=1}^{N}\mathbb{1}_{\hat{y}_i = y_i}$$

Classification: Confusion Matrix

Accuracy = $\frac{TP+TN}{TP+FN+FP+TN}$

FP: Type I error

FN: Type II error

Can we combine the best of both metrics?

F1 Score
$$F1 = \frac{2}{\frac{1}{P} + \frac{1}{R}}$$

Actual Label

Predicted Label	Cancer	Benign
Cancer	2 True Positive	1 False Positive
Benign	3 False Negative	4 True Negative

Precision
P = TP / (TP+FP)

How many **selected** items are relevant?
How **precise** were the positive predicted instances?

Maximize this if false positive (FP) is very costly.
E.g., email spam, satellite launch date prediction

Recall
R = TP / (TP+FN)

How many relevant items are **selected**?
How many positive instances can be **recalled** (predicted)?

Maximize this if false negative (FN) is very dangerous.
E.g., cancer prediction but not music recommendation

Decision Trees

- 2^{2^n} trees for n boolean attributes

Entropy

$$I\left(\frac{p}{p+n}, \frac{n}{p+n}\right) = -\frac{p}{p+n}\log_2\frac{p}{p+n} - \frac{n}{p+n}\log_2\frac{n}{p+n}$$

Remainder

$$remainder(A) = \sum_{i=1}^{v}\frac{p_i + n_i}{p+n} I\left(\frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i}\right)$$

Information Gain

$$IG(A) = \underbrace{\boxed{I\left(\frac{p}{p+n}, \frac{n}{p+n}\right)}}_{\text{Entropy of this node}} - \underbrace{\boxed{remainder(A)}}_{\text{Entropy of children nodes}}$$

```
def DTL(examples, attributes, default):
  if examples is empty: return default
  if examples have the same classification:
    return classification
  if attributes is empty:
    return mode(examples)
  best = choose_attribute(attributes, examples)
  tree = a new decision tree with root best
  for each value v_i of best:
    examples_i = {rows in examples with best = v_i}
    subtree = DTL(examples_i, attributes − best, mode(examples))
    add a branch to tree with label v_i and subtree subtree
```

Q: How to deal with continuous-valued attributes?
A: Partition them into discrete sets of intervals

Q: What if some values are missing?
A:
- Assign the most common value of the attribute
- Assign the most common value of the attribute with the same output
- Assign probability to each possible value and sample
- Drop the attribute
- Drop the rows
- ...

Overfitting

- Occurs when algorithm fits training data too closely, resulting in worse performance on test data

Occam's Razor

- **Short/simple** hypotheses that fit data **unlikely to be coincidence**
- **Long/complex** hypotheses that fit data may be **coincidence**

Pruning

- Methods: Min samples leaf, max depth, ...
- Choose the majority at the node to be pruned

Core idea: Prevent overfitting by ignoring noise.