CS2109S

AY24/25 SEM 1

01. Introduction to Artificial Intelligence

Properties of the Task Environment

Fully VS Partially Observable	Complete or partial access to the state of the environment?
Deterministic VS Stochastic	Is the next state determined by the current state?
Episodic VS Sequential	Is each action by the agent independent of the past?
Static VS Dynamic	Does the environment when the agent is deliberating?
Discrete VS Continuous	Is the number of percept and actions limited?
Single VS Multi Agent	Number of agents operating in the environment

- **Rational Agent** $\rightarrow$ Chooses action that maximises performance measure

02. Solving Problems with Searching

Problem Formulation

- States: State representation of the problem / current condition of the problem
- Initial State: The initial state the agent starts in.
- Goal State(s) / Test: If we don't know the exact goal state, use a test to check for the conditions if the agent has reached the goal state.
- Actions: Things the agent can do
- Transition Model: A description of what each action does to a given state.
- Action Cost Function: The cost of performing an action

State Representation Invariant $\rightarrow$ Constraints that must remain true across all possible states of the problem.

Search Algorithms

- Takes in a search problem as input and returns a solution or failure.
- **Order of node expansion** $\rightarrow$ The sequence in which nodes (or states) are explored or expanded during the search process, affected by the underlying data structure of the **frontier**.

Evaluation Criteria

- **Time Complexity:** Number of nodes generated or expanded.
- **Space Complexity:** Maximum number of nodes in memory.
- **Completeness:** Does it always return a solution if it exists?
- **Optimality:** Does it always find the least cost solution?

Measure

- Branching factor (b), Depth (d), Maximum Depth (m)

Tree Search vs Graph Search

- Graph search maintains a list of visited states. Each state is added into the list of visited states when first visited.
- If a state has been visited, no action is performed.

```
create frontier
create visited # Only for Graph Search

insert initial state to frontier
while frontier is not empty:
  state = frontier.pop()
  if state is goal: return solution

  if state in visited: continue # Only for Graph Search
  visited.add(state) # Only for Graph Search

  for action in actions(state):
    next state = transition(state, action)
    frontier.add(next state)
return failure
```

Uninformed Search

Type	Breath-First	Uniform-Cost	Depth-First
Frontier	Queue	Priority Queue (Path Cost)	Stack
Time	$O(b^d)$	$O(b^{C^*/\epsilon})$	$O(b^m)$
Space	$O(b^d)$	$O(b^{C^*/\epsilon})$	$O(bm)$
Complete	Yes if b is finite	Yes if $\epsilon > 0$ and C^* is finite	No
Optimal	Yes (step cost same)	Yes if $\epsilon > 0$	No

- C^* cost of optimal solution
- ϵ minimum edge cost
- Depth-First Search is not complete when depth is infinite or when we can go back and forth between states.
- In uniform-cost search, $\epsilon = 0$ may cause zero cost cycle.

Depth-Limited Search

- Limit the search depth to l. (Not complete if solution depth $> l$)
- For DFS: Backtrack once depth limit is reached.

Iterative Deepening Search

- Do depth-limited search with max-depth $0 \cdots \infty$
- Return solution if found, increase max-depth otherwise.

Type	Depth-Limited	Iterative Deepening
Frontier	Stack / Queue	Stack
Time	$O(b^l)$	$O(b^d)$
Space	$O(bl)$	$O(bd)$
Complete	No	Yes
Optimal	No	Yes (step cost same)

Informed Search

- Frontier: priority queue with an evaluation function $f(n)$ that estimates how good a state is.
- Heuristic Function $h(n)$: Estimates the cost or distance from a given state n to the goal state.
- Path Cost $g(n)$: Cost to reach a given state n.

Type	Greedy Best First	A*
Frontier	Priority Queue	Priority Queue
$f(n)$	$h(n)$	$g(n) + h(n)$
Time	$O(b^m)$	$O(b^m)$
Space	$O(b^m)$	$O(b^m)$
Complete	No	Yes
Optimal	No	**Depends on the heuristic**

- Best First Search is not complete or not optimal when the heuristic is wrong.

Heuristics

- **Admissible** $\rightarrow h(n) \leq h^*(n)$ for every node n.
  - where $h^*(n)$ is the true cost function.
  - An admissible heuristic never over-estimates the true cost to reach the goal.
  - **Theorem:** If $h(n)$ is admissible, then A* using **tree search** is optimal.
- **Consistent** $\rightarrow h(n) \leq c(n, a, n') + h(n')$ and $h(G) = 0$
  - $c(n, a, n')$ is the actual cost of performing an action a that transitions from state n to state n'.
  - $h(n')$ is the estimated cost or distance of state n' from the goal state.
  - **Theorem:** If $h(n)$ is consistent, then A* using **graph search** is optimal.
- **Dominance** $\rightarrow$ If $h_2(n) \geq h_1(n)$ for all n, then h_2 **dominates** h_1.
  - If h_2 is admissible, then it is closer to $h^*(n)$ and better for search.

03. Local and Adversarial Search

Local Search Characteristics

- To solve problems with **very large state space**
- A **good enough** solution is preferrable
- The **state is the solution**, the search path is not important.

Local Search Formulation

- States (State Space): Often close enough to the solution
- Initial state: Usually start at a random state
- Goal test (optional)
- Successor function: possible states from a state
- Evaluation / Objective functions: output the value / goodness of a state

Hill Climbing Algorithm

- Start somewhere in the state space, move towards a better spot
- Find the maximum value for a given state.
- The state landscape can be a shoulder, global maximum, local maximum or even flat local maximum

```
current = initial state
while True:
  neighbour = highest valued successor of current
  if value(neighbour) <= value(current):
    return current
  current = neighbour
```

Escape Techniques

- **Simulated Annealing** $\rightarrow$ Allow bad moves from time to time when hill climbing.

```
current = initial state
T = a large positive value # Temperature
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
```

- Probability Function: $P(current, next, T) = e^{\frac{value(next) - value(current)}{T}}$
- Other Techniques: Random Restarts, Random Walk, Tabu Search

Adversarial Search

Adversarial search is often used in fully obseravble, deterministic, two player turn-based games where the game will definitely terminate.

Problem Formulation

- States (positions), Initital State, Actions (possible moves), Transition
- **Terminal States** $\rightarrow$ states where the game ends
- **Utility Function** $\rightarrow$ output the value of a state

Minimax

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

def max_value(state):
  if is_terminal(state) or is_cutoff(state):
    return util(state)
  v = −∞
  for action, next_state in succ(state):
      v = max(v, min_value(next_state))
  return v

def min_value(state):
  if is_terminal(state) or is_cutoff(state):
    return util(state)
  v = ∞
  for action, next_state in succ(state):
      v = min(v, max_value(next_state))
  return v
```

Analysis

- **Complete:** Yes if tree is finite.
- **Time Complexity:** $O(b^m)$
- **Space Complexity:** $O(bm)$, with depth first exploration
- **Optimal:** Yes, against optimal opponent.

Alpha-beta Pruning

- $\alpha =$ highest value for MAX player
- $\beta =$ lowest value for MAX player

```
def alpha_beta_search(state):
  v = max_value(state, −∞, ∞)

def max_value(state, α, β)
  if is_terminal(state): return util(state)
  v = −∞
  for action, next_state in succ(state):
      v = max(v, min_value(next_state))
      α = max(α, v)
      if v >= β: return v
  return v

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

Alpha-beta Pruning: Analysis

- Pruning does not affect the final result
- Good move ordering improves the effectiveness of pruning
  - Perfect Ordering: $O(b^{\frac{m}{2}})$
  - Pruning is done to the other nodes in a subtree from the MAX player's perspective when a leaf node value is greater than β the lowest value for the MAX player.
  - This is because there is no point finding a greater value for the MAX player in the subtree knowing that the MIN player will choose the lower value β.

04. Intoduction to Machine Learning

Types of Feedback in Machine Learning

- **Supervised Learning** → Learns from being given the right answers.
  - **Classification** → Predict **discrete** output (eg: Cat vs Dog)
  - **Regression** → Predict **continuous** output (eg: Housing Price)
- **Unsupervised Learning** → No feedback
- **Reinforcement Learning** → Trial and Error

Formalizing Supervised Learning

- Assume that y the value we are trying to predict is generated by a **true mapping function** $f : x \to y$.
- We want to find a **hypothesis** $h : x \to \hat{y}$ from a **hypothesis class** H such that $h \approx f$ given a training set $\{(x_1, f(x_1), \ldots, (x_N, f(x_N)))\}$.
- We use a learning algorithm A to find the hypothesis

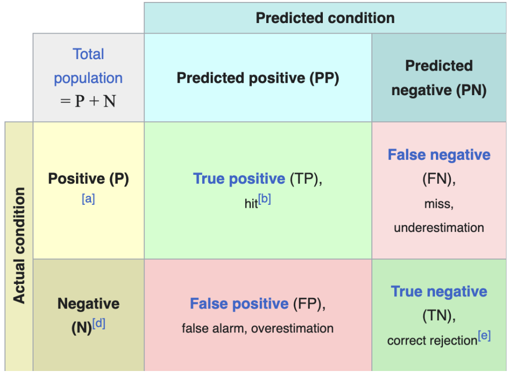
Performance Measures

Regression

- Absolute Error = $|\ \hat{y} - y\ |$
- Squared Error = $(\hat{y} - y)^2$
- Mean Absolute Error (MAE) = $\frac{1}{N}\sum_{i=1}^{N} |\ \hat{y} - y\ |$ for N examples
- Mean Squared Error (MSE) = $\frac{1}{N}\sum_{i=1}^{N} (\hat{y} - y)^2$ for N examples

Classification

- Accuracy = $\frac{1}{N}\sum_{N}^{i=1} 1_{\hat{y_i}=y_i}$
- Confusion Matrix



		Predicted condition	
	Total population $= \mathrm{P} + \mathrm{N}$	**Predicted positive (PP)**	**Predicted negative (PN)**
Actual condition	**Positive (P)** [a]	**True positive** (TP), hit[b]	**False negative** (FN), miss, underestimation
	Negative (N)[d]	**False positive** (FP), false alarm, overestimation	**True negative** (TN), correct rejection[e]

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

- Recall = $\frac{TP}{TP+FN}$
- F1 Score = $\frac{2}{\frac{1}{Precision}+\frac{1}{Recall}}$

Decision Trees

- Decision Trees can express **any function** of the input attributes
- There is a consistent decision tree for any training dataset, but probably **won't generalize to new examples.**
- Decision Trees capture the data perfectly inclduing noise, hence its performance is **perfect on training data but worse on test data.**
- **Overfitting** → Algorithm fits too closely or even exactly to its training data.
- **Occam's Razor** → Prefer short/simple is *unlikely to be coincidence*.

Decision Tree Learning

```
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
```

Choosing the Best Attribute

- Select an attribute that splits the examples into **all positive** or **all negative**
- **Entropy** → measure of randomness in data
  - Let $v_1, v_2, \ldots v_n$ be n possible values (labels) we are predicting.
  - $I(P(v_1), P(v_2), \ldots, P(v_n)) = -\sum_{i=1}^{n} P(v_i) \log_2 P(v_i)$ where $P(v_i)$ is the proportion of examples with the output label v_i
  - For a dataset containing p positive examples and n negative examples:
    - $I(\frac{p}{p+n}, \frac{n}{p+n}) = -\frac{p}{p+n}\log_2\frac{p}{p+n} - \frac{n}{p+n}\log_2\frac{n}{p+n}$
- **Information Gain** → Reduction in entropy
  - A chosen attribute A divides the training set E into subsets $E_1, \ldots, E_v$ according to their values for A, where A has v distinct values.
    - $remainder(A) = \sum_{i=1}^{v} \frac{p_i+n_i}{p+n} I(\frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i})$
    - $IG(A) = I(\frac{p}{p+n}, \frac{n}{p+n}) - remainder(A)$

Addressing Challenges in Training Dataset

- **Continuous Value attributes:** Define a discrete value input to partition the values into discrete set of intervals.
- **Missing Values:**
  - Assign most common value of the attribute.
  - Assign most common value of the attribute *with the same output*.
  - Assign probabilities to each possible value and sample.
  - Drop attribute/rows.

Pruning

- Prevent nodes from being split even when it *fails to cleanly seperate examples.*
- Results in a **smaller tree** which may have **higher accuracy**.
- Min sample leaf: Prune the tree if the number of samples under a leaf node is less than the specified amount.
- Max depth: Prune the leaves at depth levels $>$ specified max depth.

Formulas

Usage:

- $\text{Entropy}(p, n) = I\left(\frac{p}{p+n}, \frac{n}{p+n}\right) = -\frac{p}{p+n}\log_2\frac{p}{p+n} - \frac{n}{p+n}\log_2\frac{n}{p+n}$
  - where $\boldsymbol{p}$ is the number of positive labels and $\boldsymbol{n}$ is the number of negative labels
- This table is for two outcomes and for two branches.
- To read this table, $\boldsymbol{p}$ is the row headers (left vertical) and $\boldsymbol{n}$ is the column headers (top horizontal).

p\n	0	1	2	3	4	5	6	7	8	9	10
0	0										
1	0	1									
2	0	0.9183	1								
3	0	0.8113	0.971	1							
4	0	0.7219	0.9183	0.9852	1						
5	0	0.65	0.8631	0.9544	0.9911	1					
6	0	0.5917	0.8113	0.9183	0.971	0.994	1				
7	0	0.5436	0.7642	0.8813	0.9457	0.9799	0.9957	1			
8	0	0.5033	0.7219	0.8454	0.9183	0.9612	0.9852	0.9968	1		
9	0	0.469	0.684	0.8113	0.8905	0.9403	0.971	0.9887	0.9975	1	
10	0	0.4395	0.65	0.7793	0.8631	0.9183	0.9544	0.9774	0.9911	0.998	1

- $\log_2 \frac{x}{y} = \log_2 x - \log_2 y$
- $\log_2 1 = 0, \log_2 2 = 1, \log_2 3 = 1.5849, \log_2 4 = 2, \log_2 5 = 2.3219$
- $\log_2 6 = 2.5849, \log_2 7 = 2.8073, \log_2 8 = 3, \log_2 9 = 3.1699, \log_2 10 = 3.3219$

Tips and Tricks

Problem Formulation

1. Store only essential information required to describe the state of the problem.
2. Think in terms of what can change when you move from one state to another.
3. Identifying what must remain true across all states.
4. You can try working backwards from the goal state.

Choosing a Search strategy

- The number of goal states
- The distribution of goal states in the search tree
- Finite or infinite branching factor / depth
- Are there repeated states during search?
- Need for optimality? Eg: Least steps
- Need to know if there is no solution?

Ways to check if Heuristic is Admissible

- If $h(n)$ is consistent, then $h(n)$ is also admissible.
- If $h(n)$ is the cost of an optimal solution to a **relaxed problem** (with fewer restrictions on actions), then it is an admissible heuristic to the original problem.