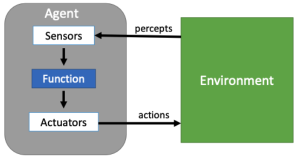
CS2109S Midterms
AY24/25 SEM 1
github/SelwynAng

1 Introduction to AI

1.1 Intelligent Agents



- **PEAS**
  1. Performance Measure: Best for whom, what are we optimizing, what information is available, any unintended effects, what are the costs
  2. Environment: Refer to Environment section
  3. Actuators: Allow intelligent agent to take actions or affect its environment
  4. Sensors: Allow intelligent agent to perceive information about its environment
- **Agent Function:** Maps from percept histories to actions, refer to Agent Function section

1.2 Task Environment

1. **Fully Observable VS Partially Observable**
   - Fully Observable: Agent has complete & accurate info about env state at all times (Eg. Chess)
   - Partially Observable: Agent has access to incomplete, uncertain or noisy info about env state (Eg. Self-driving cars)
2. **Deterministic VS Stochastic VS Strategic**
   - Deterministic: Next env state is completely determined by current state & agent action | Outcome is fully predictable (Eg. Sudoku)
   - Stochastic: Next env state is not completely determined by current state & agent action | Outcome is uncertain (Eg. Self-driving car)
   - Strategic: Env is deterministic, but outcomes depend on other agents' actions, requiring agent to consider strategies & behaviors of others (Eg. Chess)
3. **Episodic VS Sequential**
   - Episodic: Agent actions are divided into discrete periods, each episode is independent of one another, agent makes decisions based on current episode (Eg. Classification task)
   - Sequential: Agent actions are inter-dependent, each action affects future states & decisions, agent considers sequence of actions over time (Eg. Chess)
4. **Static VS Dynamic**
   - Static: Env state does not change while agent is deliberating
   - Dynamic: Env state changes over time even when agent is deliberating
   - Semi-dynamic: Env state does not change, but agent's performance score does
5. **Discrete VS Continuous**
   - Discrete: Finite # of distinct, clearly defined percepts & actions
   - Continuous: Infinite # of percepts & actions
6. **Single Agent VS Multi Agent**
   - Single Agent: Agent operating by itself in an env
   - Multi Agent: Multiple agents in an env

1.3 Agent Structures

Note: Agent is completely specified by Agent Function mapping percept sequences to actions

1. **Simple Reflex Agent:** Operates based on a set of predefined rules or conditions $\longrightarrow$ Reacts to current state of env with a corresponding action $\longrightarrow$ Does not have memory of past states or actions & does not consider future consequences
2. **Model-based Reflex Agent:** Extends simple reflex agent by maintaining internal model of world $\longrightarrow$ Allows agent to keep track of current env state & handle situations where env state is partially observable or changes over time
3. **Goal-based Agent:** Operates with specific goals in mind $\longrightarrow$ Selects actions based on ability to achieve these goals $\longrightarrow$ Considers future & plans its actions to achieve desired end state | Uses goal representation & perform search and planning
4. **Utility-based Agent:** Extends goal based agent by considering not just whether goals are achieved, but how well they are achieved $\longrightarrow$ Assigns utility value to different states & chooses actions that maximize overall utility
5. **Learning Agent:** Improves performance over time by learning from its experiences | Can be reflex, model, goal & utility based

- **Exploitation:** Maximize expected utility according to current knowledge about world
- **Exploration:** Trying to learn more about the world

2 Solving Problems by Searching

2.1 Designing an Agent

- **Assumptions:** Goal-based agent | Env is fully observable, deterministic, static, discrete
- **Problem-solving Agent:** Agent that plans ahead (considers a seq. of actions that form a path to a goal state), undertakes SEARCH process
- **Steps:**
  1. Goal Formulation: *(What do we want?)*
  2. Problem Formulation: *(How the world works?)* $\longrightarrow$ States (state space), Initial State(initial state of agent), Goal State/Test (goal state of agent), Actions (things that agent can do in a given state), Transition Model (specifies outcome of an action to a given state & how it leads to new states), Action Cost Function (cost of performing an action)
  3. Search: *(How to achieve it?)* $\longrightarrow$ Path (seq. of actions), Solution (path to a goal)
  4. Execute
- **Representation Invariant:** A condition that must be true over all valid concrete representations of a class

2.2 Search Algorithms (Introduction)

- **Search Algorithm:** Takes in search problem (input), returns solution/failure (output) | Defined by Order of Expansion (FRONTIER)
- **Evaluation Criteria:**
  1. Time Complexity: # of nodes generated/expanded
  2. Space Complexity: Max # of nodes in memory
  3. Completeness: Does it return solution if it exists?
  4. Optimality: Does it always find least cost solution?

Tree Search:

```
create frontier

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

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

Graph Search:

```
create frontier
create visited
insert initial state to frontier
while frontier is not empty:
  state = frontier.pop()
  if state is goal: return solution
  if state in visited: continue
  visited.add(state)
  for action in actions(state):
    next state = transition(state, action)
    frontier.add(next state)
return failure
```

- **Checking of Goal State:**
  - New state is checked for goal state before new states are PUSHED to frontier $\longrightarrow$ Expand less states, may skip states with less cost
  - State is checked for goal state after state is POPPED from frontier $\longrightarrow$ Expand more states, will not skip states with less cost

2.3 Search Algorithms (Uninformed Search)

- **Key Idea:** Search Algo is given no clue about how close a state is to the goal | Can be Tree or Graph Search
- **BFS:** Queue Frontier | Time Complexity: $O(b^d) = 1 + b + b^2 + \ldots + b^d$, where b is branching factor, d is depth of optimal solution | Space Complexity: $O(b^d)$ when expanded until last child in worst case | Completeness: Complete if b is finite | Optimality: Optimal if step cost is same everywhere
- **UCS:** Priority Queue (path cost) Frontier, where path cost == cost from root to a state | Time Complexity: $O(b^{C^*/\epsilon})$, where C^* is cost of optimal solution, ϵ is minimum edge cost $\longrightarrow$ C^*/ϵ is est. depth of optimal solution in worst case | Completeness: Complete if $\epsilon > 0$ and C^* is finite (if $\epsilon = 0$, zero cost cycle may occur) | Optimality: Optimal if $\epsilon > 0$
  Note: BFS is special case of UCS where step cost == 1 for every edge
- **DFS:** Stack Frontier | Time Complexity: $O(b^m)$ where b is branching factor, m is max depth | Space Complexity: $O(bm)$ as only 1 path is expanded at one time | Completeness: Not complete (when depth is infinite or can go back or forth) | Optimality: Not optimal (there can be paths with less cost not explored yet)
- **DLS (Depth Limited Search):** Limit the search depth to l where $l <= m$, backtrack once depth limit is reached | Time Complexity: $O(b^l)$ | Space Complexity: $O(bl)$ | Completeness: Not complete when soln lies deeper l | Optimality: Not optimal when soln lies deeper than l
  Note: We dk the depth of solution, which is a downside
- **IDS (Iterative Deepening Search):** Do DLS with max depth of $0, \ldots, \infty$ $\longrightarrow$ return soln if found, otherwise increase depth | Time Complexity: $O(b^d), Overhead = (n_{IDS} - n_{DLS})/n_{DLS}$ | Space Complexity: $O(bd)$ | Completeness: Complete | Optimality: Optimal if step cost is same everywhere
  Note: IDS is not always faster than DFS $\longrightarrow$ Consider state space s.t. each state have only single successor & goal node is at depth n $\longrightarrow$ IDS will run in $O(n^2)$, DFS will run in $O(n)$
- **Backward Search:** Search from goal
- **Bidirectional Search:** Combine forward search & backward search, stop when 2 searches meet | Time Complexity: $2 * O(b^{d/2}) < O(b^d)$

2.4 Search Algorithms (Informed Search)

- **Key Idea:** Search Algo has a clue on how close a state is to the goal
- **Best First Search:** Priority Queue ($f(n)$) Frontier, where $f(n)$ estimates the goodness of a state (Node with lowest $f(n)$ is selected first to be expanded) | $f(n)$ can be purely heuristic (estimated cost from n to goal) or a combi of path cost & heuristic
- **Greedy Best First Search:** Priority Queue ($f(n) = h(n)$) Frontier, where $h(n)$ is heuristic function that est. cost from n to goal (Expands node that seems closest to goal according to $h(n)$ without considering path cost so far) | Time Complexity: $O(b^m)$ | Space Complexity: $O(b^m)$ | Completeness: Not complete since GBFS might keep expanding nodes based on $h(n)$ without ever finding goal | Optimality: Not optimal since GBFS selects nodes based on $h(n)$ without considering path cost
- A^* **Search:** Priority Queue ($f(n) = g(n) + h(n)$) where $g(n)$ is cost so far to reach n | Time Complexity: $O(b^m)$ | Space Complexity: $O(b^m)$ | Completeness: Complete | Optimality: Optimal
  - If $h(n)$ is admissible $\longrightarrow$ A^* using Tree search is optimal
  - If $h(n)$ is consistent $\longrightarrow$ A^* using Graph search is optimal
  - *Note: UCS is special case of A^* search where $h(n) = 0$*

2.5 Heuristics

- Estimate cost from node n to goal
- **Admissible Heuristics:** For every node n, $h(n) \leq h^*(n)$, where $h^*(n)$ is true cost to reach goal state from n (Never over-estimate)
- **Consistent Heuristics:** For every node n, every successor n' generated by action a, $h(n) \leq c(n, a, n') + h(n')$ and $h(G) = 0$ (Proof $h(n) - h(n') \leq c(n, a, n')$)
  Note: If $h(n)$ is consistent, $f(n') \geq f(n) \rightarrow f(n)$ is non-decreasing along any path $\longrightarrow$ Nodes are expanded in order of increasing f cost
- **Dominance:** If $h_2(n) \geq h_1(n)$ for all n $\rightarrow$ h_2 dominates h_1 | If h_2 is admissible $\rightarrow$ h_2 is better for search
- **Creating Admissible Heuristics:**
  - Problem with fewer restrictions on actions is called a relaxed problem
  - Cost of an optimal soln to a relaxed problem is an admissible h for original problem

3 Local Search & Adversarial Search

3.1 Local Search

- **Assumptions:** Agent is a Goal/Utility-based agent, Env has a very large state space
- **Informed & Uninformed Search VS Local Search**
  1. IUS: Low to moderate state space | Optimal or no soln | Search path is usually the soln
  2. LS: Very large state space | Good enuf soln is preferable rather than no soln | State is the soln (don't care about search path)
- **Local Search Overview:**
  - Basic Idea: Start somewhere in state space, move towards a better spot
  - Problem Formulation: States(state space), Initial State(initial state of agent), Goal test (optional, coz we actually dk the goal state, rely on eval function instead), Successor Function (possible states from a state), Evaluation Function (Output value/goodness of a state)
- **Hill Climbing Algorithm**

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

- Known as Greedy Local Search (pick best amongst neighbors, repeat)
- Best Soln: State space where eval. function has a max value (global max)
- Disadvantages: Cannot reach global max if it enters local max, plateau | Sensitive to choice of initial state, poor initial state may result in poor final state (Can overcome with random restarts, walks)
- **Simulated Annealing**

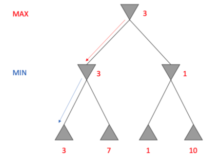
```
current = initial state
T = a large positive value
while T > 0:
  next = a randomly selected successor of current
  if value(next) > value(current): current = next
  else with probability P(current, next , T): current = next
  decrease T
return current
```

- P(current, next, T) = $e^{(value(next) - value(current))/T}$
- More exploration of bad states is allowed when T is high, more exploitation is done when T is low $\longrightarrow$ basically choosing worse successor may lead to a better max
- Theorem: If T decreases slowly enough, SA will find global optimum with high probability

3.2 Adversarial Search

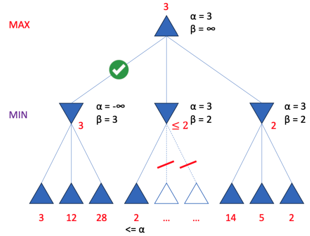
- **Assumptions:** Agent is Utility-based | Env is a game (game cannot be single player, partially observable, stochastic, but must be fully observable, deterministic, discrete, terminal states exist, 2 players, zero-sum, turn taking)
- **Minimax Algorithm:**

```
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

- Intuition: MAX wins when utility is high, MIN wins when utility is low | Assign utility values to all terminal states & start tracing from terminal states $\longrightarrow$ Eventually, all states will have utility values, starting player can choose a state that will max/min his utility
- Analysis: Completeness: Complete if tree is finite | Time Complexity: $O(b^m)$ | Space Complexity: $O(bm)$ depth first exploration | Optimality: Optimal against optimal opponent
- **Alpha-beta Pruning**

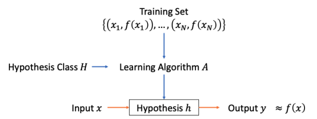


- Definitions: α is best explored option to the root for MAX player (Highest value for MAX) | β is best explored option along path to the root for MIN player (Lowest value for MIN)
- Procedure: 1. Assign $\alpha = -\infty$, $\beta = \infty$ for root 2. Propagate values down to the terminal node 3. Update α value at MAX node, β value at MIN node 4. Propagate values up 5. Prune branches of nodes where $\alpha \geq \beta$
- **Minimax with Cutoff**
  - Instead of calling $is\_terminal$, call is_cutoff which returns TRUE if (1): State is terminal or (2): Cut-off is reached
  - Instead of using $utility$, call $eval$ which is an eval. function that returns (1): Utility for terminal states or (2): Heuristic value for non-terminal states

4 Introduction to ML & Decision Trees

4.1 Introduction to ML

- **Definitions:** Computer program is said to learn from experience E w.r.t. some class of tasks T & performance measure P, if its performance at tasks in T, as measured by P, improves with experience E
- **Types of Feedback:**
  1. Supervised Learning: Involves training a model on a labeled dataset, where input data is paired with correct output $\longrightarrow$ Model learns to map inputs to outputs based on this labeled data, allowing it to make predictions on new data
     - Regression: Predict continuous input
     - Classification: Predict discrete input
  2. Unsupervised Learning: Deals with dataset that do not have labeled outputs $\longrightarrow$ Goal is to identify patterns & structures within data
  3. Reinforcement Learning: Agent learns to make decisions by interacting with an environment $\longrightarrow$ Agent receives feedback in the form of rewards or penalties based on its actions $\longrightarrow$ Learns optimal behaviors over time
- **Formal Definitions:**



4.2 Performance Measure

- **Regression:**

For a set of N examples $\{(x_1, y_1), \ldots, (x_N, y_N)\}$ we can compute the average (mean) squared error as follows.

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

Where $\hat{y}_i = h(x_i)$ and $y_i = f(x_i)$.

For a set of N examples $\{(x_1, y_1), \ldots, (x_N, y_N)\}$ we can compute the average (mean) absolute error as follows.

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

Where $\hat{y}_i = h(x_i)$ and $y_i = f(x_i)$.

- **Classification:**

Classification is correct when the prediction $\hat{y} = y$ (true label).

For a set of N examples $\{(x_1, y_1), \ldots, (x_N, y_N)\}$ we can compute the average correctness (accuracy) as follows.

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

Where $\hat{y}_i = h(x_i)$ and $y_i = f(x_i)$.

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

- **Accuracy:** $\frac{TP+TN}{TP+FN+FP+TN}$
- **Precision:** $\frac{TP}{TP+FP}$ (How many selected items are relevant, maximise if FP is costly)
- **Recall:** $\frac{TP}{TP+FN}$ (How many relevant items are selected, maximise if FN is dangerous)
- **F1 Score:** $\frac{2}{1/precision + 1/recall}$

4.3 Decision Trees

- **Traits of Decision Trees:**
  - Decision Trees can express any function of input attributes
  - Consistent Decision Tree for any training set, but probably will not generalize to new examples
  - # of distinct decision trees with n boolean attributes = 2^{2^n}
- **Decision Tree Learning Algorithm**

```
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
```

  - mode: Category with the highest number
  - choose attribute: Chooses attribute with the highest information gain
- **Choosing an attribute:**
  - Ideally select an attribute that splits examples into "all positive" or "all negative"
  - Entropy (Measure of randomness):
    $I(P(v_1), \ldots, P(v_n)) = -\sum_{i=1}^{n} P(v_i) log_2 P(v_i)$,
    where for data set containing p positive & n negative examples,
    $I(\frac{p}{p+n}, \frac{n}{p+n}) = -\frac{p}{p+n} log_2 \frac{p}{p+n} - \frac{n}{p+n} log_2 \frac{n}{p+n}$
    Note: I(1,0) = 0, I(0,1) = 0, I(0.5, 0.5) = 1
  - Information Gain (Entropy of curr. node - Total Entropy of children nodes):
    $IG(A) = I(\frac{p}{p+n}, \frac{n}{p+n}) - remainder(A)$
    remainder(A) = $\sum_{i=1}^{v} \frac{p_i + n_i}{p+n} I(\frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i})$, where examples are split into v subsets by attribute A
- **Dealing with continuous valued attributes:** Define a discrete valued input attribute to partition values into discrete set of intervals
- **Dealing with missing values:** Assign most common value of attribute, assign probability to each value and sample, drop attribute, drop rows
- **Overfitting:** Decision Tree is perfect on training data, but worse on test data
- **Occam Razor:** Prefer short/simple hypothesis (long/complex hypothesis that fits data may be coincidence)
- **Pruning:** Prevents nodes from being split even when it fails to cleanly separate examples (Min samples leaf: Merge until leaf node is above min. samples number | Max depth: Merge until leaf nodes are at depth less than max depth)

5 Tutorials Pointers

- **DFS:** DFS utilizes $O(bm)$ memory as DFS must store all nodes along the current path from root to the deepest node explored, along with branching factor b at each level
- **UCS vs Dijkstra:** 2 algos are the same (both traverse search space in the manner, using a PQ to keep track of which nodes/states to visit next), but can argue that Dijkstra finds the shortest path to every node from a single source, while UCS only finds the shortest path to goal states

6 Midterms PYP Pointers

- **Tree Search VS Graph Search:** Assuming the problem consists of discrete states (eg. pitcher problem), search space using tree search may be infinite (Eg. we keep filling up and emptying pitcher) $\longrightarrow$ but using graph search will make search space finite (graph search will not revisit visited states)
- **Tree Search:**
  - If optimal solution is needed: use BFS (for problems where each action has same cost), use UCS (for problems where each action has different cost), use IDS (if space is more important concern than time, since BFS is space inefficient)
  - If optimal solution is not needed, but we need to preserve space: use DLS
  - Termination & Completeness on Tree Search Algorithms:
    - BFS is complete, but may not terminate
    - DFS is incomplete and may not terminate
    - UCS is complete, but may not terminate
    - IDS is complete, but may not terminate (unless depth limiting condition is applied)
    - DLS is complete and terminates regardless of whether a solution exists due to its depth limiting property
- **Heuristics:**
  - Max of 2 admissible heuristics ($max(h_0, h_1)$) for a problem is also an admissible heuristic $\rightarrow$ $max(h_0, h_1)$ is also dominant over h_0 & h_1 since it takes max of both heuristics
  - Given h_0 is admissible for problem p_0 & h_1 is admissible for problem p_1 $\rightarrow$ h_0 & h_1 are admissible for the combined problem $p_0 + p_1$ $\rightarrow$ $max(h_0, h_1)$ is also admissible and dominant for $p_0 + p_1$
  - If heuristic is **consistent**, it must be **admissible** too!
  - If heuristic is **non-admissible**, it must be **non-consistent** too!
  - Common Proving Techniques:
    - *Consistency:* Decrease in heuristic value must not exceed action cost ($h(n) - h(n') \leq c(n, a, n')$) $\rightarrow$ Focus on maximum decrease in heuristic value and showing it is less than actual action cost

- *Relaxed Problem:* True cost/admissible heuristic of a relaxed problem is an admissible heuristic to original problem, True cost of relaxed problem is consistent heuristic to original problem
- **Dominance:** Not useful to describe dominance between non-admissible heuristics since non-admissible heuristics lead to sub-optimal search $\longrightarrow$ for consistency sake, just treat it such that dominance concept can apply to both admissible & non-admissible heuristics

6.1 Log Values (Base 2)

1. $log_2 1 = 0$
2. $log_2 2 = 1$
3. $log_2 3 = 1.58496$
4. $log_2 4 = 2$
5. $log_2 5 = 2.32192$
6. $log_2 6 = 2.58496$
7. $log_2 7 = 2.80735$
8. $log_2 8 = 3$
9. $log_2 9 = 3.16993$
10. $log_2 10 = 3.32193$

6.2 Entropy Table

	0	1	2	3	4	5	6	7	8	9	10
0	0.0										
1	0.0	1.0									
2	0.0	0.918	1.0								
3	0.0	0.811	0.971	1.0							
4	0.0	0.722	0.918	0.985	1.0						
5	0.0	0.65	0.863	0.954	0.991	1.0					
6	0.0	0.592	0.811	0.918	0.971	0.994	1.0				
7	0.0	0.544	0.764	0.881	0.946	0.98	0.996	1.0			
8	0.0	0.503	0.722	0.845	0.918	0.961	0.985	0.997	1.0		
9	0.0	0.469	0.684	0.811	0.89	0.94	0.971	0.989	0.998	1.0	
10	0.0	0.439	0.65	0.779	0.863	0.918	0.954	0.977	0.991	0.998	1.0