CS2109S Midterm Cheatsheet AY23/24 Sem 2 (Gavin)

Intelligent Agents

Agents

- PEAS = Performance Measure, Environment, Actuators, Sensors

Task Environment

- Fully observable vs Partially observable
- Deterministic vs Stochastic vs Strategic
  (Deterministic) Next state of the environment is completely determined by current state and agent action.
  (Strategic) Environment is deterministic except for actions of other agents.
- Episodic vs Sequential
- Static vs Dynamic vs Semi-dynamic
  (Semi-dynamic) Environment does not change with time, but agent's performance score does.
- Discrete vs Continuous
- Single Agent vs Multi-agent

Agent Structure

- Simple Reflex Agent:
  Selects action based on current percept, ignoring percept history. Uses condition-action(if-then) rules
- Model-based Agent:
  Keep track of the part of the world it can't see now. This internal state is updated through the transition model of the world (i.e. knowledge on how the world changes) and the sensor model, i.e. information from percepts.
- Goal-based Agent:
  In addition to tracking the state of the world, also track a set of goals, then pick the action that brings it closer to the goal.
- Ultility-based Agent:
  Uses an utility function to assign a score to any given percept sequence, i.e. an internalisation of the performance measure. If the utility function and the performance measure are aligned, then the agent will be rational.
- Learning Agent:
  Uses a performance element to select external actions, a critic to give feedback on how the agent is doing and how the performance element can be improved, a learning element responsible for making improvements, and a problem generator that suggests actions that will lead to new and informative experiences.

Uninformed Searching

Representation invariant
Conditions that ensure abstract states have corresponding concrete states.

Tree Search

Define order of node expansion

```
create frontier

insert initial state
while frontier is not empty:
    state = frontier.pop()
    for action in actions(state):
        next state = transition(state, action)

        if next state is goal: return solution
        frontier.add(next state)
return failure
```

Breadth-first search(BFS):

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

Uniform-cost search (UCS):

- Frontier: priority queue (path cost)
- Time Complexity: $O(b^{C*/\epsilon})$
- Space Complexity: $O(b^{C*/\epsilon})$
- Complete? Yes if $\epsilon > 0$ and $C*$ finite
- Optimal? Yes, if $\epsilon > 0$

$C*$: Cost of optimal solution; ϵ: min edge cost.
Note: Goal-test is done after popping from frontier instead of goal-testing next state.

Depth-first search(DFS):

- Frontier: stack
- Time Complexity: $O(b^m)$
- Space Complexity: $O(bm)$
- Complete? No. Possibility of infinite loops with infinite trees
- Optimal? No

Depth limited search(DLS): DFS with limited search depth

- Time Complexity: $b^0 + b^1 + \ldots + b^l = O(b^l)$
- Space Complexity: $O(bl)$
- Complete? No
- Optimal? No

Iterative deepening search(IDS): Do DLS with increasing l

- Time Complexity:
  $O(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + 2b^{d-1} + b^d = O(b^d)$
- Space Complexity: $O(bd)$
- Complete? Yes
- Optimal? Yes, if step cost is same everywhere

Bidirectional search:: Combine search from forward and backwards

- Time Complexity: $2 * O(b^{d/2}) < O(b^d)$

Graph search:

- Use a set to keep track and avoid expanding visited states.
- Version 1: If next state visited: continue. Add to frontier and visited in the end of action loop.
  Expand less states, may skip states with less cost
- Version 2: After popping from frontier, immediately goal-test → visited-check: continue → visited.add(state)
  Expand more states, will not skip states with less cost

Informed Searching

Greedy best first search:
Pick a node n from the frontier with minimum value of an evaluation function $f(n) = h(n)$. If goal state, return it, else expand to add its child nodes to the frontier. Child nodes are only added if they are unvisited or previously visited with a higher path cost.

- Time Complexity $O(b^m)$
- Space Complexity: $O(b^m)$
- Complete? No
- Optimal? No

A* search:
This is a Best-First Search that uses the evaluation function $f(n) = g(n) + h(n)$, where $g(n)$ is the path cost from the initial node to node n. Assuming all action costs are $> \epsilon > 0$ and that state space is finite, $A*$ search is complete. If there are infinitely many nodes with $f(n) \leq f(goal)$, then it is not complete. Time and space complexities are both exponential $O(bd)$ for a poor heuristic. Whether it is cost-optimal depends on certain properties of the heuristic:

- **Admissibility**. Never overestimates the cost to reach a goal, i.e. optimistic $h(n) \leq h^*(n)$. If $h(n)$ is admissible, then A^* using tree-like search is cost-optimal.
- **Consistency**. Stronger than admissibility. $h(n)$ is consistent if, for every node n and every successor n' of n generated by an action a, we have $h(n) \leq c(n, a, n') + h(n')$, i.e. triangle inequality. If $h(n)$ is consistent, then $A*$ using graph search is cost-optimal.
  A consistent heuristic is admissible, but not the other way around.
- **Dominance**. If $h_2(n) \geq h_1(n)$ for all n, then h_2 dominates h_1. More dominant heuristics are better for search.

A problem with fewer restrictions on actions is called a **relaxed problem**. The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem.
Iterative Deepening A* (IDA*): Use iterative deepening search. Cutoff using f-cost $f(n) = g(n) + h(n)$ instead of depth.
Simplified memory-bounded A* (SMA*): Drop the nodes with worst f-cost in frontier when memory is full.

Local Search

Path is not important, state is the solution.
Consists of initial state, goal test, successor function(heuristic or objective function)
Hill climbing algorithm
Find local maxima by travelling to neighbouring states with the highest value, and terminating when no neighbour has a higher value. Easy objective function is to negate the heuristic function. This algorithm suffers from local maxima (i.e. non-global maximum), ridges (sequence of local maxima), plateaus (sequence of same values, but is local maxima) and shoulders (same values, but progress is possible).
Solutions are:

- Sideways Move. We do a limited number of consecutive sideways move, in hopes that the plateau is really a shoulder.
- Stochastic Hill Climbing. Chooses a random uphill move, with probabilities based on the steepness of the move.
- First-choice Hill Climbing. Randomly generate successors until one is better than the current state. Useful when a state has e.g. thousands of successors.
- Random-Restart Hill Climbing. Perform a fixed number of steps from some randomly generated initial steps, then restart if no maximum found.

Simulate annealing
Basically, randomly pick a next move. If it's a better state, go for it, else accept it with a probability less than 1, and this probability decreases exponentially with the "badness" of the move. Idea is to escape local maxima by allowing some random moves once in a while. If T decreases slowly enough, simulated annealing will find a global optimum with high probability.
Local beam search
Pick k random initial states **deterministically**, then generate their successors. If goal is found, terminate, else pick the k best successors and repeat. Similar to k random restarts but information is shared. It may suffer from a lack of diversity if the k states start to cluster.
Stochastic beam search alleviates this problem by choosing k successors with **probability** proportional to their value.
Genetic algorithm
Population $\underbrace{\rightarrow}_{\text{Fitness Function}}$ Selection → Cross-Over → Mutation → New Population

Adversarial Search and Games

The most commonly studied games are deterministic, two-player, turn-taking, perfect information, zero-sum games. Perfect information just means fully observable, and zero sum means that there is no "win-win" situation.
Minimax algorithm
Opponent has to behave optimally for algorithm to work.

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

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

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

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))
    if v >= β: return v
    α = max(α, 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))
    if v <= α: return v
    β = min(β, v)
  return v
```

is_cutoff() returns true if state is terminal or cutoff (depth/time) is reached.
eval() returns utility for terminal states, and heuristic values for non-terminal states.
Alpha-beta pruning

- Max node: only v and α is modified
- Min node: only v and β is modified
- When modifying α, find max of previous α and β values.
- When modifying β, find min of previous α and β values.
- Prune when $\alpha \geq \beta$
- Remember to update α and β values of parent when recursing back upwards.
- Good move ordering improves effectiveness of pruning. Perfect ordering: $O(b^{m/2})$

Optimizing search: Transposition table, pre-computation of best moves.

Machine Learning

In supervised learning, we assume that y is generated by a true mapping function $f : x \rightarrow y$. We want to find hypothesis $h : x \rightarrow \hat{y}$ (from hypothesis class H), s.t $h \approx f$ given a training set $(x_1, f(x_1)), \ldots, (x_N, f(x_N))$
Classification: Output is one of a finite set of values.
Regression: Output is a number.
Types of Feedback:

- Supervised Learning. Agent observes input-output pairs and learns a function that maps from input to output.
- Unsupervised Learning. Agent learns patterns in the input without any explicit feedback. Most common task is clustering.
- Reinforcement Learning. Agent learns from a series of reinforcements: rewards and punishments.

Regression: Error
Let $\hat{y_i} = h(x_i)$ (prediction) and $y_i = f(x_i)$ (actual)

- Mean Absolute Error $= \frac{1}{N}\sum_{i=1}^{N} |\hat{y_i} - y_i|$
- Mean Squared Error $= \frac{1}{N}\sum_{i=1}^{N} (\hat{y_i} - y_i)^2$

Classification Correctness and Accuracy

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

Classification: Confusion Matrix

- Accuracy $= \frac{TP+TN}{TP+FN+FP+TN}$
- Precision $= \frac{TP}{TP+FP}$ (Maximise this if FP is costly)
- Recall $= \frac{TP}{TP+FN}$ (Maximise this if FN is costly)
- F1 Score $= \frac{2}{\frac{1}{Precision}+\frac{1}{Recall}}$

Decision Trees

```
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
```

Need to define this!

Entropy

- Entropy $= -\sum_{i=1}^{n} p_i \log_2(pi)$
- Remainder $= -\sum_{i=1}^{v} \frac{p_i+n_i}{p+n} I(\frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i})$
- Information Gain(A) $= I(P_1, P_2, \ldots, P_n) - remainder(A)$
- Formulas can and should be extended when more than 2 categories are involved.
- Split Information $= -\sum_{i=1}^{d} \frac{|E_i|}{E} \log_2 \frac{|E_i|}{E}$
- Gain Ratio $= \frac{IG(A)}{SplitInformation(A)}$ (Use on attributes with many values)

```
I(1/2, 1/2) = 1.000000
I(1/3, 2/3) = 0.918296
I(1/4, 3/4) = 0.811278
I(1/5, 4/5) = 0.721928
I(2/5, 3/5) = 0.970951
I(1/6, 5/6) = 0.650022
I(1/7, 6/7) = 0.591673
I(2/7, 5/7) = 0.863121
I(3/7, 4/7) = 0.985228
I(1/8, 7/8) = 0.543564
I(3/8, 5/8) = 0.954434
```

```
log_2(1)  = 0.000000
log_2(2)  = 1.000000
log_2(3)  = 1.584963
log_2(4)  = 2.000000
log_2(5)  = 2.321928
log_2(6)  = 2.584963
log_2(7)  = 2.807355
log_2(8)  = 3.000000
log_2(9)  = 3.169925
log_2(10) = 3.321928
```

Notes:
- When dealing with Attributes of differing cost: Use Cost-Normalised Gain to ultilise low-cost attributes where possible.
- Continuous-valued attributes should be partitioned into discrete set of intervals to use decision trees.
- When dealing with missing values, you can assign most common value (with same output), drop the attribute, use probability, drop the row and so on.

Pruning:

- **Occam's Razor**: Prefer short/simple hypothesis, hypothesis that fits are unlikely to be coincidence.
- min-sample: each node should have minimum around of values.
- max-depth: limit the depth of the decision tree.
- Pruning can be propagated upwards with majority combination.