CSE 473 Notes

Contents

1 Agents and Environment

1.1 Agents

An agent is an entity that perceives its environment through sensors and acts upon it through actuators

- A rational agent is an entity that selects actions that maximizes the expected value of the performance measure

1.2 Agent Functions

An agent function implemented by an agent program running on a machine describes what the agent does in all circumstances

- Agent functions consist of policies, which determine how the agent acts in an environment

1.3 Performance Measure

A performance measure is a function that evaluates the performance of an agent

1.4 Environment Types and Agent Design

The environment type largely determines the agent design

- Partially observable: agent requires memory to remember its environment
- Stochastic: agent may have to prepare for contingencies
- Multi-agent: agent may need to behave randomly
- Static: agent has time to compute a rational decision
- Continuous time: continuously operating controller
- Unknown physics: agent needs to explore environment
- Unknown performance measure: agent needs to observe and interact with human principal

1.5 Agent Types

- Simple reflex agents
  A simple reflex agent performs actions based solely on the percept
- Reflex agents with internal state
  A reflex agent with internal state performs actions based on the current situation, as defined by the percept and the stored internal state
- Planning agents
  A planning agent performs actions based on evaluating future action sequences
- Goal-based agents
  A goal-based agent performs actions based on what the future goal is
- Utility-based agents
  A utility-based agent performs actions based not only on what the future goal is, but what the best way to reach that goal is

2 Search Problems

2.1 Search Problems

A search problem consists of

- A state space $\mathcal{S}$
  The state space is the set of all possible states of a search problem
- An initial state s_0
  The initial state is the starting state of a search problem
- A set of actions in each state $\mathcal{A}(s_t)$
  The set of actions describes all possible agent actions at a given state
- A transition model $\mathrm{Result}(s_t, \mathcal{A}(s_t))$
  The transition model / successor function describes how an action at a given state alters the state
- A goal test $G(s_t)$
  The goal test checks whether the current state is the goal state
- An action cost $c(s_t, \mathcal{A}(s_t), {s_t}')$
  The action cost describes how an action at a given state affects the cost function

A solution is an action sequence that reaches a goal state

- An optimal solution has the least cost among all solutions

2.2 Search Algorithm Properties

- A search algorithm is complete if it is guaranteed to find a solution if one exists
- A search algorithm is optimal if it is guaranteed to find the least cost path

2.3 State Space Graphs

A state space graph is an abstract representation of a search problem

- Nodes represent world states
- Edges represent action results
- The goal test is a set of one or more goal nodes

State space graphs are useful for visualizing problems, but are often too large to construct in practice

2.4 Search Trees

A search tree is an abstract representation of possible outcomes in a search problem

- Root node represents initial state
- Children nodes represent successor states
- Edges represent action results

Search trees can be constructed on demand, and are often procedurally constructed to minimize memory requirements

2.5 General Tree Search

```
function treeSearch(problem, strategy) {
    <initialize the search tree using the initial state of problem>
    while <exists candidates for tree expansion> {
        <select a leaf node for expansion according to strategy>

        if <leaf node corresponds to a goal state> {
            return <corresponding solution>
        } else {
            <expand leaf node and add the expanded nodes to the search tree>
        }
    }
    return <failure>
}
```

- The branching factor b of a search tree is the number of children at each node
- The depth m of a search tree is the maximum number of edges in a path from the root to a leaf
- The depth s of a solution is the number of edges in a path from the root to a goal node
- A search tree has $O(b^m)$ nodes

2.6 Uninformed Search

Uninformed search is a searching technique that has no additional information about the cost from the current state to the goal

2.7 Breadth First Search

Given a search tree with unweighted edges, breadth first search is complete and optimal

- Strategy: expand the shallowest node first
- Implementation: frontier is a FIFO queue
- *An implementation of breadth first search can be found in **CSE 421 Notes**, page 5*

Characteristics of breadth first search

- Breadth first search has $O(b^s)$ running time and $O(b^s)$ space complexity
- Breadth first search is preferred when s is much smaller than m

2.8 Depth First Search

Given a finite search tree, depth first search is complete but not optimal

- Strategy: expand the deepest node first
- Implementation: frontier is a LIFO stack
- *An implementation of depth first search can be found in **CSE 421 Notes**, page 6*

Characteristics of depth first search

- Depth first search has $O(b^m)$ running time and $O(bm)$ space complexity
- Depth first search is preferred when s is close to m

2.9 Iterative Deepening Search

Given a search tree with unweighted edges, iterative deepening search is complete and optimal

- Strategy: expand the deepest node first until depth d, then repeat until depth $d+1$
- Implementation: run depth first search with depth limit d, then repeat with depth limit $d+1$

Characteristics of iterative deepening search

- Iterative deepening search has $O(b^s)$ running time and $O(bs)$ space complexity
- Iterative deepening search combines the space advantage of depth first search with the running time advantage of breadth first search

2.10 Uniform Cost Search

Given a search tree with weighted edges, uniform cost search is complete and optimal

- Strategy: expand the node with the smallest cost from root to node
- Implementation: frontier is a minimum priority queue sorted by cost from root to node

Characteristics of uniform cost search

- Uniform cost search has $O\left(b^{C/\varepsilon}\right)$ running time and $O\left(b^{C/\varepsilon}\right)$ space complexity
- Uniform cost search processes nodes in order of cost

where C is the least cost of a solution and ε is the least cost of an edge

2.11 Informed Search

Informed search is a searching technique that has additional information about the estimate cost from the current state to the goal and uses it to guide search towards the goal

2.12 Admissible Heuristics

A heuristic is admissible if it never overestimates the cost to nearest goal

- A heuristic $h(n)$ is admissible if $0 \leq h(n) \leq h^*(n)$ where $h^*(n)$ is the true cost to nearest goal
- Admissible heuristics are often solutions to relaxed problems

2.13 Consistent Heuristics

A heuristic is consistent if its estimate is always $\leq$ the estimated distance from any neighboring vertex to the goal plus the cost of reaching that neighbor

- A heuristic $h(n)$ is consistent if $h(A) \leq h(B) + c(A, B)$
  - Alternatively, a heuristic $h(n)$ is consistent if $h(A) - h(B) \leq c(A, B)$
- If a heuristic is consistent, then it is admissible

2.14 Greedy Search

Given a search tree with weighted edges, greedy search is neither complete nor optimal

- Strategy: expand the node with smallest estimated cost to nearest goal
- Implementation: frontier is a minimum priority queue sorted by estimated cost to nearest goal
- Heuristic: estimate of cost to nearest goal for each node

2.15 A* Search

Given a search tree with weighted edges and an admissible heuristic, A* search is complete and optimal

- Strategy: expand the node with smallest $g(n) + h(n)$
  - $g(n)$ is cost from root to node
  - $h(n)$ is estimated cost from node to nearest goal
- Implementation: frontier is a minimum priority queue sorted by $g(n) + h(n)$
- Heuristic: estimate of cost to nearest goal for each node

3 Game Problems

3.1 Game Problems

A game is a task environment with more than one agent. A game problem consists of

- An initial state s_0
  The initial state is the starting state of a game problem
- A player in each state $\mathrm{Player}(s_t)$
  The player describes whose turn it is at a given state
- A set of actions in each state $\mathcal{A}(s_t)$
  The set of actions describes all possible player actions at a given state
- A transition model $\mathrm{Result}(s_t, \mathrm{Action}(s_t))$
  The transition model / successor function describes how an action at a given state alters the state
- A terminal test $\mathrm{TerminalTest}(s_t)$
  The terminal test checks whether the game is over
- A terminal value $\mathrm{Utility}(s_t, p_i)$ for player p_i
  The terminal value assigns a score to a completed game
- A utility value $\mathrm{Utility}(s_t, p_i)$ for player p_i
  The utility value assigns a score to a state

3.2 Contingent Plan

A contingent plan is a strategy or policy which recommends a move for every possible eventuality

3.3 Zero-Sum Games

Zero-sum games are those where agents have opposite utilities

- One agent maximizes utility while the other agent minimizes utility
- Agents in zero-sum games are in direct competition with one another

3.4 General Games

General games are those where agents have independent utilities

- Agents in general games may cooperate, ignore, or compete with one another

3.5 Standard Games

Standard games are deterministic, observable, two-player, turn-taking, and zero-sum

3.6 Minimax Algorithm

The minimax algorithm is a contingent plan used in zero-sum games

- Selects actions leading to states with best minimax value
- Assumes all future moves will be optimal
- Algorithm is rational against a rational player

The minimax value depends on whether the agent seeks to maximize or minimize utility

- For agents that maximize utility, the minimax value is the maximum score out of all terminal states reachable from the current state
  - $\text{Utility}(s_t, p_i) = \max_{s_{t+1} \in \text{Result}(s_t, \mathcal{A}(s_t))} (\text{Utility}(s_{t+1}, p_i))$
- For agents that minimize utility, the minimax value is the minimum score out of all terminal states reachable from the current state
  - $\text{Utility}(s_t, p_i) = \min_{s_{t+1} \in \text{Result}(s_t, \mathcal{A}(s_t))} (\text{Utility}(s_{t+1}, p_i))$

3.7 Minimax Implementation

```
function decision(s) {
    if (Player(s) == minAgent) {
        return min(value(Result(s, a))) for all a in Action(s);
    } else if (Player(s) == maxAgent) {
        return max(value(Result(s, a))) for all a in Action(s);
    }
}

function value(s) {
    if TerminalTest(s) {
        return Utility(s);
    } else if (Player(s) == minAgent) {
        return min(value(Result(s, a))) for all a in Action(s);
    } else if (Player(s) == maxAgent) {
        return max(value(Result(s, a))) for all a in Action(s);
    }
}
```

- Minimax has $O(b^m)$ running time and $O(bm)$ space complexity

3.8 Minimax Algorithm Generalization

The minimax algorithm may be modified to work in games which are not zero-sum or have multiple players

- Terminals and nodes have utility tuples with each component corresponding to each player's utility
- Each player maximizes/minimizes its own component

3.9 Alpha-Beta Pruning

Alpha–beta pruning is a search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree

- Pruning has no effect on the minimax value computed for the root

Pruning children of min agent's nodes

1. Let α be the best value that the max agent can get so far at any choice point along the current path from the root
2. Let n be the min value at the current node
3. Loop over the current node's children, updating n as we go
4. If n becomes worse than α, then the max agent will avoid the current node
   - This means that we can prune the current node's remaining children

Pruning children of max agent's nodes

1. Let β be the best value that the min agent can get so far at any choice point along the current path from the root
2. Let n be the max value at the current node
3. Loop over the current node's children, updating n as we go
4. If n becomes better than β, then the min agent will avoid the current node
   - This means that we can prune the current node's remaining children

3.10 Alpha-Beta Pruning Implementation

```
1   function maxValue(state, alpha, beta) {
2       v = -infinity;
3       for each successor of state {
4           v = max(v, minValue(successor, alpha, beta));
5           if (v >= beta) {
6               return v;
7           }
8           alpha = max(alpha, v);
9       }
10      return v;
11  }
12
13  function minValue(s) {
14      v = infinity;
15      for each successor of state {
16          v = min(v, maxValue(successor, alpha, beta));
17          if (v <= alpha) {
18              return v;
19          }
20          beta = min(beta, v);
21      }
22      return v;
23  }
```

- Alpha-Beta pruning has $O(b^{m/2})$ running time, assuming perfect ordering of nodes

3.11 Expectiminimax

The expectiminimax algorithm is a variation of the minimax algorithm used in games which incorporate an element of chance

- Random effects of the game are treated as a third agent

3.12 Expectiminimax Implementation

```
function decision(s) {
    if (Player(s) == minAgent) {
        return min(value(Result(s, a))) for all a in Action(s);
    } else if (Player(s) == maxAgent) {
        return max(value(Result(s, a))) for all a in Action(s);
    }
}

function value(s) {
    if TerminalTest(s) {
        return Utility(s);
    } else if (Player(s) == minAgent) {
        return min(value(Result(s, a))) for all a in Action(s);
    } else if (Player(s) == maxAgent) {
        return max(value(Result(s, a))) for all a in Action(s);
    } else if (Player(s) == chanceAgent) {
        return sum(value(Result(s, a)) * Pr(a)) for all a in Actions(s);
    }
}
```

- Expectiminimax has $O(b^m)$ running time and $O(bm)$ space complexity

3.13 Evaluation Functions

Evaluation functions score non-terminals in depth-limited search

- Evaluation functions are often a weighted linear sum of features
- Evaluation functions may be a complex non-linear function generated by machine learning

3.14 Evaluation Function Values

- Minimax decisions are invariant with respect to monotonic transformations on values
  - A monotonic transformation f is one where if $x > y$, then $f(x) > f(y)$
- Expectiminimax decisions are invariant with respect to affine transformations on values
  - An affine transformation f is one of the form $f(x) = ax + b$

3.15 Rollouts

A rollout is the execution of a simple fast policy from the current state to the terminal state

- Often multiple rollouts are executed and the fraction of wins is recorded
- The fraction of wins correlates with the true value of the state

3.16 Monte Carlo Tree Search

Monte Carlo Tree Search is a stochastic and heuristic driven search algorithm used in games where it is not feasible to deterministically explore the search tree

Monte Carlo Tree Search performs the following at each turn

- Evaluation by rollouts
  - Rollouts are allocated to more promising nodes with higher fraction of wins
  - Rollouts are allocated to more uncertain nodes with fewer executed rollouts
- Selective search
  - The node with the highest fraction of wins is selected

As the number of rollouts executed approaches infinity, the behavior defined by Monte Carlo Tree Search approaches that of minimax

4 Constraint Satisfaction Problems

4.1 Constraint Satisfaction Problems

A constraint satisfaction problem is a subset of search problems. A constraint satisfaction problem consists of

- Variables X_i
  Variables define the current state
- A domain D
  The domain is the set of all possible values of a variable
- A set of constraints
  The constraints checks whether the current state is a goal state

4.2 Backtracking

Backtracking incrementally assigns values to variables, and abandons an assignment as soon as it determines that the assignment cannot possibly satisfy the constraints

- Backtracking is a modified version of depth first search

4.3 Backtracking Implementation

```
1   function backtracking(constraints) {
2       return recursiveBacktracking(constraints, {});
3   }
4
5   function recursiveBacktracking(constraints, assignment) {
6       if assignment is complete {
7           return assignment;
8       }
9       variable = selectUnassignedVariable();
10      for each value in domain {
11          if assignment + {variable, value} is consistent with constraints {
12              assignment.add(variable, value);
13              result = recursiveBacktracking(constraints, assignment);
14
15              if result != failure {
16                  return result;
17              }
18              assignment.remove(variable, value);
19          }
20      }
21      return failure;
22  }
```

- Backtracking has $O(b^m)$ running time and $O(bm)$ space complexity

4.4 Backtracking Optimizations

There are three main ways in which backtracking can be optimized

- Reducing the size of the domain of the variables
- Changing the order in which variables are assigned
- Changing the order in which values are assigned to a variable

4.5 Arc-Consistency

An arc $X \to Y$ is consistent if and only if for every $x \in X$, there exists some $y \in Y$ which could be assigned without violating a constraint

- We can enforce arc-consistency while backtracking to reduce the size of the domain of the variables
- Arc-consistency is also known as 2-consistency

4.6 Enforcing Arc-Consistency

```
function arcConsistency(constraints) {
    queue = <a queue of all the arcs in the constraints>

    while queue is not empty {
        (x, y) = queue.remove();

        if removeInconsistentValues(x, y) {
            for each k in x.getNeighbors() {
                queue.add(k, x);
            }
        }
    }
}

function removeInconsistentValues(x, y) {
    removed = false;
    for each u in domain(x) {
        satisfied = false;
        for each v in domain(y) {
            if (u, v) satisfies the constraint x <-> y {
                satisfied = true;
            }
        }
        if not satisfied {
            domain(x).remove(u);
            removed = true;
        }
    }
    return removed;
}
```

- Enforcing arc-consistency has $O(n^2d^2)$ running time

4.7 K-Consistency

A path $X_1, ..., X_{k-1} \to Y$ is consistent if and only if for every consistent assignment $x_1, ..., x_{k-1} \in X_1, ..., X_{k-1}$, there exists some $y \in Y$ which could be assigned without violating a constraint

- We can enforce K-consistency while backtracking to reduce the size of the domain of the variables
- K-consistency offers a greater reduction in the size of the domain of the variables than arc-consistency
- Enforcing K-consistency has greater running time as K increases

4.8 Minimum Remaining Values Ordering

Minimum remaining values (MRV) selects the variable with the fewest values left in its domain

- MRV is an ordering on *variables*
  - i.e. it selects a variable
- MRV ordering enables backtracking to fail fast

4.9 Least Constraining Values Ordering

Least constraining values (LCV) selects the value which rules out the fewest values in the remaining variables

- LCV is an ordering on *values*
  - i.e. it selects a value to be assigned to a variable
- LCV ordering selects the value that best guarantees success

5 Markov Decision Processes

5.1 Markov Decision Processes

A Markov decision problem is a fully observable stochastic search problem. A Markov decision problem consists of

- A state space $\mathcal{S}$
  The state space is the set of all possible states of a Markov decision process
- An initial state s_0
  The initial state is the starting state of a Markov decision process
- A set of actions in each state $\mathcal{A}(s)$
  The set of actions describes all possible agent actions at a given state
- A transition model $T(s, a, s') = P(s' \mid s, a)$
  The transition model is the probability that action a from state s leads to state s'
- A reward function $R(s, a, s')$
  The reward that action a from state s to state s' will yield
- A goal test $G(s)$
  The goal test checks whether the current state is the goal state
- A utility function $U_h([s_0, a_0, s_1, a_1, s_2, ...])$
  The utility function assigns a score to a sequence of actions

5.2 Utility Functions

The utility function of a sequence is defined as the sum of the rewards along the sequence

$$U([s_0, a_0, s_1, a_1, s_2, ...]) = R(s_0, a_0, s_1) + \gamma R(s_1, a_1, s_2) + \gamma^2 R(s_2, a_2, s_3) + ...$$

- γ represents the discount factor
  - If $\gamma < 1$, then the agent prefers rewards in the immediate future
  - If $\gamma > 1$, then the agent prefers rewards in the distant future
  - If $\gamma = 1$, then the agent has no preference as to when it receives rewards

5.3 Policies

A policy is a function $\pi : S \to a$ that gives an action for each state

- An optimal policy π^* gives the action that maximizes the expected reward at each state

5.4 Policy Utilities

The utility of a policy π is defined as the sum of rewards along the generated sequence

$$U^\pi(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, \pi(s), s_{t+1})\right] = \mathbb{E}[R(s_1, \pi(s_1), s_2) + \gamma R(s_2, \pi(s_2), s_3) + ...]$$

5.5 Action Value

The action value $Q^*(s, a)$ is the expected utility of taking action a in state s and thereafter acting optimally

$$Q(s, a) = \sum_{s'} P(s' \mid s, a) \left[R(s, a, s') + \gamma U(s') \right]$$

- In a stochastic environment, taking action a may result in an unexpected outcome
- The action value represents the utility of the action across all possible outcomes

5.6 Bellman Equation

The utility of a state is the sum of the expected reward for the next transition and the discounted utility of the next state, assuming that the agent chooses the optimal action

$$U(s) = \max_{a \in \mathcal{A}(s)} Q^*(s, a)$$

$$U(s) = \max_{a \in \mathcal{A}(s)} \Big(\sum_{s'} P(s' \mid s, a) \left[R(s, a, s') + \gamma U(s') \right] \Big)$$

5.7 Value Iteration

Value iteration computes the optimal state utility function by iteratively improving the estimate of $U(s)$

- As the number of iterations goes to ∞, $U(s)$ converges to its unique optimal value
- Value iteration provides an implicit policy that can be obtained via policy extraction

5.8 Value Iteration Implementation

```
1   function valueIteration(ε) {
2       repeat {
3           Δ = 0;
4           for (State s : S) {
5               u = U(s);
6               U(s) = max_{a∈A(s)} (Σ_{s'} P(s' | s, a)[R(s, a, s') + γU(s')]);
7               Δ = max(Δ, |u − U(s)|);
8           }
9       } until (Δ < ε);
10  }
```

- Value iteration has $O(|S|^2|A|)$ running time per iteration

5.9 Policy Extraction

Given the optimal state utility function values $U(s)$, policy extraction finds the optimal policy function π_U implied by the values $U(s)$

$$\pi_U(s) = \arg\max_{a \in \mathcal{A}(s)} \Big(\sum_{s'} P(s' \mid s, a) \left[R(s, a, s') + \gamma U(s') \right] \Big)$$

5.10 Policy Iteration

Policy iteration determines the optimal policy by iteratively improving its current policy π

- As the number of iterations goes to ∞, π converges to the optimal policy function
- Policy iteration provides an explicit policy

5.11 Policy Iteration Implementation

```
1   function policyIteration(ε) {
2       π(s) ∈ A(s) arbitrarily for all s ∈ S;
3
4       policyStable = true;
5       while (policyStable) {
6           oldPolicy = π;
7
8           U = policyEvaluation(π, ε);
9           π = policyImprovement(U);
10
11          if (oldPolicy != π) {
12              policyStable = false;
13          }
14      }
15  }
16
17  function policyEvaluation(π, ε) {
18      repeat {
19          Δ = 0;
20          for (State s : S) {
21              u = U(s);
22              U(s) = Σ_s' P(s' | s, π(s)) [R(s, π(s), s') + γU(s')];
23              Δ = max(Δ, |u − U(s)|);
24          }
25      } until (Δ < ε);
26      return U
27  }
28
29  function policyImprovement(U(s)) {
30      for (State s : S) {
31          oldAction = π(s);
32          π(s) = arg max_a Σ_s' P(s' | s, a) [R(s, a, s') + γU(s')];
33      }
34      return π;
35  }
```

- Policy evaluation takes $O(|S|^2)$ per iteration
- Policy improvement takes $O(|S|^2|A|)$ per iteration

6 Reinforcement Learning

6.1 Reinforcement Learning Environments

A reinforcement learning environment is a partially observable stochastic search problem with unknown physics. A reinforcement learning environment consists of

- A state space $\mathcal{S}$
  The state space is the set of all possible states of a reinforcement learning environment
- An initial state s_0
  The initial state is the starting state of a reinforcement learning environment
- A set of actions in each state $\mathcal{A}(s)$
  The set of actions describes all possible agent actions at a given state
- A transition model $T(s, a, s') = P(s' \mid s, a)$
  The transition model is the probability that an action a from state s leads to state s'
- A reward function $R(s, a, s')$
  The reward that action a from state s to state s' will yield
- A goal test $G(s)$
  The goal test checks whether the current state is the goal state

In a reinforcement learning environment, the transition model and reward function is unknown and must be learned through exploration of the environment

6.2 Passive Reinforcement Learning

A passive learning agent has a fixed policy that determines its behavior

- The agent is told what to do in the environment

6.3 Active Reinforcement Learning

An active learning agent decides what actions to take and uses this experience to improve its policy

- The agent decides what to do in the environment

6.4 Model-Based Learning

In model-based learning, the agent learns an approximate transition model and reward function based on experiences and solves for values as if the learned models were correct

- In model-based learning, the agent attempts to predict the next state based on experiences

6.5 Model-Free Learning

In model-free learning, the agent approximates the utility of each state by averaging together observed sample values obtained through trial and error

- In model-free learning, the agent learns a policy π directly from rewards

6.6 Sample-Based Policy Evaluation

Sample-based policy evaluation is a model-based reinforcement learning algorithm

$$U_{k+1}^{\pi}(s) = \sum_{s'} T(s, \pi(s), s') \underbrace{\left[R(s, \pi(s), s') + \gamma U_k^{\pi}(s')\right]}_{\text{sample}}$$

where γ is the discount factor

- As the discount factor decreases, more weight is given to more recent samples

6.7 Temporal Difference Learning

Temporal difference learning is a model-free passive reinforcement learning algorithm that learns from every experience

$$U_{k+1}^{\pi}(s) = U_k^{\pi}(s) + \alpha \underbrace{\left[R(s, \pi(s), s') + \gamma U_k^{\pi}(s')\right]}_{\text{sample}} - U_k^{\pi}(s)\big]$$

where α is the learning rate and γ is the discount factor

- As the learning rate increases, more weight is given to more recent samples
- If the learning rate α decreases with the number of iterations, then the exponential moving average will converge

6.8 Q-Learning

Q-learning is a model-free active reinforcement learning learning algorithm that learns from every experience

$$Q_{k+1}(s, a) = Q_k(s, a) + \alpha \underbrace{\left[R(s, \pi(s), s') + \gamma \max_{a' \in \mathcal{A}(s')} Q_k(s', a')\right]}_{\text{sample}} - Q_k(s, a)\big]$$

where α is the learning rate and γ is the discount factor

- The action value $Q(s, a)$ is the expected utility of taking action a in state s
- Q-learning converges to the optimal policy regardless of the initial policy

6.9 Exploration Function

The exploration function prioritizes states whose utility are not yet well-established

$$f(u, n) = \frac{u + k}{n}$$

where u is the estimated utility, n is the number of visits, and k is a pre-determined constant. The exploration function can be used to guide Q-learning as follows

$$Q_{k+1}(s, a) = Q_k(s, a) + \alpha \underbrace{\left[R(s, \pi(s), s') + \gamma \max_{a' \in \mathcal{A}(s')} f(Q_k(s', a'), N(s', a'))\right]}_{\text{sample}} - Q_k(s, a)\big]$$

where α is the learning rate, γ is the discount factor, and $N(s', a')$ is the number of times q-state (s, a) has been visited

6.10 Feature-Based Representations

Feature-based representations describe a state using a vector of features

$$Q(s,a) = w_1 f_1(s,a) + w_2 f_2(s,a) + ... + w_n f_n(s,a)$$

where $f_i(s,a)$ is the i^{th} feature and w_i is the weight associated with that feature

6.11 Approximate Q-Learning

Approximate Q-learning is a version of Q-learning that makes use of feature-based representations

$$\begin{aligned}
&Q(s,a) = w_1 f_1(s,a) + w_2 f_2(s,a) + ... + w_n f_n(s,a) \\
&\text{difference} = \left[R(s,\pi(s),s') + \gamma \max_{a' \in \mathcal{A}(s')} Q_k(s',a') \right] - Q(s,a) \\
&w_i = w_i + \alpha \cdot \text{difference} \cdot f_i(s,a)
\end{aligned}$$

where α is the learning rate and γ is the discount factor

7 Graphical Models

7.1 Probability Theory

- $\mathbb{P}(X \mid Y) = \dfrac{\mathbb{P}(X, Y)}{\mathbb{P}(Y)}$
- $\mathbb{P}(X \mid Y) = \dfrac{\mathbb{P}(Y \mid X)\mathbb{P}(X)}{\mathbb{P}(Y)}$
- $\mathbb{P}(X_1, ..., X_n) = \mathbb{P}(X_1) \cdot \mathbb{P}(X_2 \mid X_1) \cdot \mathbb{P}(X_3 \mid X_1, X_2) \cdot ... \cdot \mathbb{P}(X_n \mid X_1, ..., X_{n-1})$
- $\mathbb{P}(X = x) = \sum_y \mathbb{P}(X = x, Y = y)$
- $X \perp\!\!\!\perp Y$ indicates that X and Y are independent
- X and Y are independent if and only if $\mathbb{P}(X, Y) = \mathbb{P}(X)\mathbb{P}(Y)$
- $X \perp\!\!\!\perp Y \mid Z$ indicates that X is conditionally independent of Y given Z
- X is conditionally independent of Y given Z if and only if $\mathbb{P}(X \mid Y, Z) = \mathbb{P}(X \mid Z)$
- X is conditionally independent of Y given Z if and only if $\mathbb{P}(X, Y \mid Z) = \mathbb{P}(X \mid Z)\mathbb{P}(Y \mid Z)$

7.2 Bayes Nets

Bayes nets are a technique for describing complex joint distributions using simple conditional distributions. A Bayes net consists of

- A set of nodes
  Each node represents a variable which can be assigned or unassigned
- A directed acyclic graph
  Each arc represents causal interactions between variables
- A conditional probability table
  Each node is associated with a conditional probability table, where each row represents a probability distribution for that node given values of its parents

Bayes nets have size $O(n \cdot d^k)$ where n is the number of variables, d is the maximum number of values for a variable, and k is the maximum number of parents

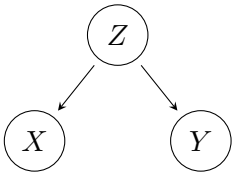
$$P(X_1 = x_1, X_2 = x_2, ..., X_n = x_n) = \prod_{i=1}^{n} P\left(X_i = x_i \mid \text{Parents}(X_i)\right)$$

- If X_i has no parents, then $P\left(X_i = x_i \mid \text{Parents}(X_i)\right) = P(X_i = x_i)$

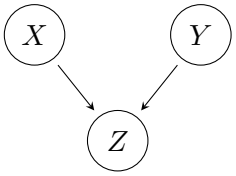
7.3 Bayes Net Independence Properties for Triples

- Causal chain
  - Y is not independent of X
  - Y is conditionally independent of X given Z

- Common cause
  - Y is not independent of X
  - Y is conditionally independent of X given Z



- Common effect
  - X and Y are independent of each other
  - X is not conditionally independent of Y given Z



7.4 Evidence Variables

A variable is an evidence variable if it is given or known

- i.e. If $X \perp\!\!\!\perp Y \mid Z$ and we know Z, then Z is an evidence variable

7.5 Active Triples

A triple (X, Y, Z) is active if X is dependent on Y or vice-versa

- If there are no evidence variables, then causal chain and common cause are active triples
- If Z is an evidence variable, then common effect is an active triple

7.6 Inactive Triples

A triple (X, Y, Z) is inactive if X is independent of Y and vice-versa

- If there are no evidence variables, then common effect is an inactive triple
- If Z is an evidence variable, then causal chain and common cause are inactive triples

7.7 Active Paths

A path is active if and only if every triple contained in the path is active

- Intuitively, a path is active if it carries dependence

7.8 d-Separation

Variables X, Y are d-separated given $\{Z_1, ..., Z_N\}$ if all paths between X and Y are inactive

- If X and Y are d-separated given $\{Z_1, ..., Z_N\}$, then $X \perp\!\!\!\perp Y \mid \{Z_1, ..., Z_N\}$

7.9 Factor Zoo

- Joint distribution, $P(X, Y)$
  - Entries $P(x, y)$ for all $x \in X$ and $y \in Y$
  - $|X| \times |Y|$ matrix
  - Entries sum to 1
- Projected joint, $P(x, Y)$
  - Entries $P(x, y)$ for some fixed $x \in X$ and all $y \in Y$
  - $|Y|$-vector
  - Entries sum to $P(x)$
- Single conditional, $P(Y \mid x)$
  - Entries $P(y \mid x)$ for some fixed $x \in X$ and all $y \in Y$
  - $|Y|$-vector
  - Entries sum to 1
- Family of conditionals, $P(Y \mid X)$
  - Entries $P(y \mid x)$ for all $x \in X$ and $y \in Y$
  - $|X| \times |Y|$ matrix
  - Entries sum to $|X|$

7.10 Exact Inference

Exact inference algorithms calculate the exact value of a probability $P(X \mid Y)$ given their conditional probability tables

- Exact inference explicitly calculates the probability
- Exact inference is NP-hard

7.11 Variable Elimination

```
function variableElimination(query, evidence) {
    currentFactors = load conditional probability tables instantiated by evidence
    hiddenVariables = variables - query - evidence

    for hiddenVariable in hiddenVariables {
        hiddenVariableFactors = all factors in currentFactors that contain hiddenVariable
        joinedFactor = join the factors in hiddenVariableFactors by taking their product
        eliminatedFactor = eliminate hiddenVariable from joinedFactor by summing it out

        remove hiddenVariableFactors from currentFactors
        add eliminatedFactor to currentFactors
    }
    fullJoinFactor = join all remaining factors in currentFactors
    normalizedFactor = normalize fullJoinFactor

    return normalizedFactor
}
```

- Variable elimination is an exact inference algorithm
- Variable elimination simplifies calculations by removing variables

7.12 Approximate Inference

Approximate inference algorithms calculate the approximate value of a probability $P(X \mid Y)$ given their conditional probability tables

- Approximate inference samples the distribution to approximate the probability
- Approximate inference trades accuracy for speed

7.13 Likelihood Weighted Sampling

```
1   function likelihoodWeighting(evidenceVariables) {
2       w = 1.0;
3
4       for (int i = 1; i <= n; i++) {
5           if (X_i in evidenceVariables) {
6               x_i = fixed observed value of X_i;
7               w = w * P(x_i | Parents(X_i));
8           } else {
9               sample x_i from P(x_i | Parents(X_i));
10          }
11      }
12      return (x_1, x_2, ..., x_n), w
13  }
```

7.14 Gibbs Sampling