Chapter 2: Intelligent Agents

(2.1): Agents and Environments

(2.2): The Concept of Rationality

(2.3): The Nature of Environments Be comfortable with the PEAS description and environment properties.

(2.4): The Structure of Agents You may be asked to pick the best agent type for some problem and justify your answer.

(2.5): Summary Go through the chapter summary.

Chapter 3: Solving Problems by Search

(3.1): Problem-Solving Agents Be comfortable defining a search problem.

(3.2): Example Problems

(3.3): Search Algorithms & Uniform Search Strategies Ignore sections 3.4.4 and 3.4.5 for the exam.

(3.5): Informed (Heuristic) Search Strategies Informed search relies on domain-specific knowledge / hints that help locate the goal state. $h(n) = h(\text{State n})$ = relevant information about State n

You may be asked to solve a search problem by hand.

(3.6): Heuristic Functions Straight-line heuristics is admissible: it never overestimates the cost. An admissible heuristics is guaranteed to give you the optimal solution. Every consistent heuristics is admissible, but not the other way around. A dominating heuristic is a heuristic that estimates closer to the actual cost than another heuristic.

(3.7): A* The evaluation function for A* is $f(n) = g(\text{State}_n) + h(\text{State}_n)$ where $g(n)$ is the path cost from the initial node to node n, and $h(n)$ is the estimated cost of the best path that continues from node n to a goal node. A node n with minimum (or maximum if necessary) $f(n)$ should be chosen for expansion.

Algorithm 0.1 Best-First Search: A* Pseudocode

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
    node ← NODE(State = problem.Initial)
    frontier ← a priority queue ordered by f, with node as an element
    reached ← a lookup table, with one entry with key problem.INITIAL and value node
    while not IS-EMPTY(frontier) do
        node ← POP(frontier)
        if problem.IS-GOAL(node.State) then return node
        end if
        for all child in EXPAND(problem, node) do
            s ← child.STATE
            if s not in reached or child.PATH-COST < reached[s].PATH-COST then
                reached[s] ← child
                child to frontier
            end if
        end for
    end while
    return failure
end function
function EXPAND(problem, node) yields nodes
    s ← node.STATE
    for all action in problem.ACTIONS(s) do
        s′ ← problem.RESULT(s, action)
        cost ← node.PATH-COST + problem.ACTION-COST(s, action, s′)
        yield NODE(State = s′, Parent = node, Action = action, Path − Cost = cost)
    end for
end function
```

(3.8): Summary Go through the chapter summary. FOCUS ON A* algorithm

Chapter 4: Search in Complex Environments

(4.1): Local Search and Optimization Problems Local search doesn't care about the path to the goal, just getting to the goal. They're useful for pure optimization problems (finding the best state according to an objective function.) Generally use a single current state and generally move to neighbors of that state. Two key advantages are: little memory usage (usually a constant amount) and can find reasonable solutions in large of infinite (continuous) states spaces. The performance can be measured using completeness (guaranteed to find a solution when there is one and report when there isn't), cost-optimality (does it find a solution with the lowest path cost of all solutions), or time or space complexity.

Hill-climbing search

The most primitive informed search approach; it is a naive greedy algorithm and the objective function is the value of the next state. The agent can get stuck on peaks (local maxima), ridges (sequences of peaks), and plateaus (areas where the evaluation function has the same value).

Algorithm 0.2 Hill-climbing search

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
    current ← problem.INITIAL
    while true do
        neighbor ← a highest-valued successor state of current
        if VALUE(neighbor) ≤ VALUE(current) then return current
        end if
        current ← neighbor
    end while
end function
```

Simulated Annealing

Accepts a move if it improves the objective value, and accepts some "bad" moves given some probability depending on the current objective value.

Converges to a global optimum; in practice, it can give excellent results.

Algorithm 0.3 Simulated Annealing

```
function SIMULATED-ANNEALING(problem, Schedule) returns a solution state
    current ← problem.INITIAL
    for t = 1 to ∞ do
        T ← SCHEDULE(t)
        if T == 0 then return current
        end if
        next ← a randomly selected successor of current
        ΔE ← VALUE(current) − VALUE(next)
        if ΔE > 0 then current ← next
        else current ← next only with probability e^(ΔE/T)
        end if
    end for
end function
```

Evolutionary algorithms

Algorithm 0.4 Genetic Algorithm Pseudocode

```
function GENETIC-ALGORITHM(population, fitness) returns an individual
    repeat
        weights ← WEIGHTED-BY(population, fitness)
        population2 ← empty list
        for i = 1 to SIZE(population) do
            parent1, parent2 ← WEIGHTED-RANDOM-CHOICES(population, weights, 2)
            child ← REPRODUCE(parent1, parent2)
            if small random probability then child ← MUTATE(child)
            end if
            add child to population2
        end for
        population ← population2
    until some individual is fit enough, or enough time has elapsed
    return the best individual in population, according to fitness
end function
function REPRODUCE(parent1, parent2) returns an individual
    n ← LENGTH(parent1)
    c ← random number from 1 to n
    return Append(SUBSTRING(parent1, 1, c), SUBSTRING(parent2, c + 1, n))
end function
```

. . . and everything related to Evolutionary algorithms that I covered in class (especially: EVERYTHING about GENETIC ALGORITHM)
IGNORE TABU SEARCH

Chapter 5: Adversarial Search and Games

(5.1): Game Theory

(5.2): Optimal Decision in Games You may be asked to solve an adversarial problem by hand using Min-Max and alpha-beta pruning. Ignore section 5.2.2.

(5.3): Summary Go through the chapter summary.

Chapter 6: Constraint Satisfaction Problems

(6.1): Defining CSPs You may be asked to formally define a constraint satisfaction problem.

(6.2): Constraint Propagation: Inference in CSPs Ignore sections 6.2.4 and 6.2.5.

(6.3): Backtracking Search for CSPs Ignore sections 6.3.3 and 6.3.4.

(6.4): Summary Go through the chapter summary.

Chapter 7: Ant Colony Optimization