Honors Examination
Swarthmore College
CS63 Artificial Intelligence

Examiner:
Jerod Weinman
Grinnell College

Instructions:

1. This is a closed book, closed notes, closed web examination.
2. Calculators and scratch paper are allowed, if not overtly necessary.
3. Answers may be submitted directly on the exam itself, in an accompanying separate written document (with problem numbers—\textit{i.e.}, 2.1.d.ii—clearly specified), or in combination, as you deem appropriate.
4. If you find any question unclear, clearly identify any assumptions you believe are necessary to answer the question and then proceed to answer under your stated assumptions.
1 Agents and Environments

Consider the game Tetris, described as follows:

Tetriminos are game pieces shaped like tetrominoes, geometric shapes composed of four square blocks each. A random sequence of Tetriminos fall down the playing field (a rectangular vertical shaft, called the “well” or “matrix”). The objective of the game is to manipulate these Tetriminos, by moving each one sideways (if the player feels the need) and rotating it by 90 degree units, with the aim of creating a horizontal line of ten units without gaps. When such a line is created, it gets destroyed, and any block above the deleted line will fall. When a certain number of lines are cleared, the game enters a new level. As the game progresses, each level causes the Tetriminos to fall faster, and the game ends when the stack of Tetriminos reaches the top of the playing field and no new Tetriminos are able to enter. ¹

In Artificial Intelligence: A Modern Approach, Russell and Norvig offer six dimensions for classifying a task environment. One property of the Tetris agent environment is given below. Classify the five additional environment properties for Tetris and provide a brief justification for each.

Single agent, no other players are involved

2 Solving Problems by Searching

2.1 Search

Suppose your lab partner proffered the following algorithm for conducting a search.

```python
function SEARCH(state, problem) returns a solution, or failure
    if problem.GOAL(state) then
        return [] /* The empty list of actions */
    for each action in problem.ACTIONS(state)
        next ← problem.RESULT(state, action) /* State resulting from the action */
        solution ← SEARCH(next, problem) /* List of actions leading to goal */
        if solution is not failure
            return [action | solution] /* Prepend action onto solution */
    return failure
```

a. Identify this search routine as specifically as possible by providing its name.

b. Briefly explain whether the algorithm is complete.

c. Briefly explain whether the algorithm is optimal.

d. Give the complexity of the algorithm, defining/naming/describing (in English) any variables used—e.g., “O(x^2), where x is the horizontal position in meters”:

   i. Time:

   ii. Space:
2.2 Informed Search

Consider the following hexagonal “maze” problem.

Starting in the white center of a hex grid, you must find a path to the green goal. “Your path can only turn gently (or go straight) in a yellow hexagon, and can only turn sharply (or go straight) in an orange hexagon.”

The number of rings determines the puzzle topology; above are puzzles of size 1–4. Each tile is addressed by a row and column offset from the center, as shown in the largest example.

Beginning at the center tile, moves may go to adjacent, unvisited tiles, subject to the color constraints. More formally:

**States:** The position $p_t$ after $t$ moves may be any of the valid tiles $(i, j)$ where $i$ is the row and $j$ is the column. The state at time $t$ is $s_t = \langle p_t, p_{t-1}, ..., p_0 \rangle$, which is the sequence of all positions traversed.

**Initial state:** We begin in the center tile, so $p_0 = (0, 0)$ with $s_0 = \langle p_0 \rangle$.

**Actions:** Moves (i.e., “sharp left”, “gentle left”, “straight”, “gentle right” “sharp right”) are subject to the current tile’s color constraints and may only lead to previously unvisited tiles, so $p_{t+1} \neq p_k$ for $k = 0...t$. Bear in mind the move direction is relative to the hexagon side from which the previous state would bring the player. (All six directions are available from the initial white center tile.)

**Transition model:** The move at time $t$ takes the position to an adjacent tile, so with $p_t = (i, j)$, we have $p_{t+1} = (i + \delta_i, j + \delta_j)$ where $(\delta_i, \delta_j) \in \{(0, 2), (0, -2), (-1, 1), (-1, -1), (1, -1), (1, 1)\}$. Note that after $p_0$, not all of these deltas will be possible, as some are excluded by unavailable actions.

**Goal test:** The goal position $p_{\text{goal}} = (g, h)$ has the color green. A goal state $s_t$ has $p_t = p_{\text{goal}}$.

**Path cost:** Each step has unit cost.

The rightmost illustration shows an example valid solution sequence in blue with moves (after the first) that are sharp left, straight, gentle left, sharp right, sharp left, gentle left, straight, with final state $s_7 = \langle (2, -2), (1, -3), (0, -4), (1, -3), (0, -2), (1, -1), (1, 1), (0, 0) \rangle$.

(Continued on next page.)

---

Design the best heuristic for this hex turn puzzle that you are able to. You should strive to come up with an admissible heuristic. If you cannot, you should create one that is not admissible but may outperform other uninformed searches for efficiency (i.e., node expansions).

a. Write a clear, unambiguous description of your best heuristic in a mixture of high-level English, pseudo-code and/or mathematical formalism, as appropriate. Your description ought to allow a programmer to both create and understand a working implementation from your description.

There should be no ambiguities that might render one implementation different from another, nor should the programmer have to intuit or interpret abstractly described calculations. However, you may assume standard mathematical operations and libraries (e.g., with sin, pow, etc.) are available.

b. Prove, or at least explain, why your heuristic is or is not admissible.
2.3 Adversarial Search

Consider the Minimax search tree below, where the utility value appears within terminal leaf states. Max nodes point upward (Δ) and Min nodes point downward (∇). The tree represents the play choice and utility values for the MAX player.

a. Assuming successors are examined from left to right, clearly indicate any branch that would be ignored when alpha-beta pruning is used by placing an “X” on the line leading to such child node(s).

Carefully note that sometimes the pruning condition can occur by stopping the loop upon examining the last node (before confirming no further nodes exist). In such cases, you are to indicate the pruning condition has been met by “pruning” (with an “X”) the rightmost unconnected branch to the right of each internal node.

b. Within each MAX and MIN node write the value assigned/returned by the alpha-beta algorithm. Any skipped/pruned nodes should be left blank.

Although not required, it is acceptable to write and cross out values beside nodes (i.e., \(-1, 2, 3\)) the slashed out values indicating they are no longer current, so long as you write the final value assigned by the alpha-beta pruning algorithm within the node.

c. Is it possible to have pruning at the root node? That is, would the root node bypass any of its children? Explain why or why not.
2.4 Relationships

Give the name of the functionally equivalent search algorithm that results from each of the following special cases; be as specific as possible. If you cannot name an equivalent search algorithm, clearly but succinctly describe the relevant, cogent behavior(s) of the special-case algorithm given.

a. $A^*$ search using a heuristic that always gives zero.

b. Local beam search with one initial state and no limit on the number of states retained (i.e., the beam width $k = \infty$).

c. Simulated annealing with temperature $T = 0$ at all times (and omitting the terminating condition that exits when $T = 0$; perhaps replacing with a goal test or iteration limit).

d. Uniform cost search on a problem with uniform step costs.

e. Genetic algorithm with a population size $N = 1$.

---

3 Artificial Neural Networks

3.1 Perceptron Learning

Recall the perceptron classifier

\[ h_w(x) = \begin{cases} 
1 & \text{if } w \cdot x \geq 0 \\
0 & \text{otherwise} 
\end{cases} \]

with the weight update rule

\[ w_i \leftarrow w_i + \alpha \times (y - h_w(x)) \times x_i, \text{ for all } i \]

for \( \alpha > 0 \) on an example vector \( x \) with label \( y \in \{0, 1\} \). (Updates to all elements \( w_i \) of the vector \( w \) happen at the same time.)

The following two questions are independent of one another. That is, they each concern distinct examples and weights.

a. Suppose an input feature vector \( x \) is the exact opposite of the weight vector \( w \); that is, \( x_i = -w_i \). Explain whether the perceptron will (i) always, (ii) sometimes (give the conditions), or (iii) never classify that example as the positive (\( y = 1 \)) class.

b. Consider a single application of the weight update rule to all elements of \( w \) for one incorrectly classified sample \( x \) in a perceptron learner. Explain whether, after this update to \( w \), the perceptron \( h_w(x) \) will (i) always, (ii) sometimes (give the conditions), or (iii) never classify that example correctly.
3.2 Activation Function

What is the importance of the activation function $g : \mathbb{R} \to \mathbb{R}$ in a multi-layer network? That is, for an internal/hidden layer computation, why wouldn’t we just omit it and use the identity function, i.e., $g(w \cdot x) = w \cdot x$ as the result passed to the next layer?

3.3 Backpropagation

Gradient descent with backpropagation can be used to learn multi-layer artificial neural network weights. Explain why the weights should be randomly initialized before training (as opposed to setting all weights to the same fixed value, whether zero or another constant).
4 Reinforcement Learning

Recall the update equation for temporal-difference, table-based Q-learning is

\[
Q(s, a) \leftarrow Q(s, a) + \alpha \times \left( R(s) + \gamma \max_{a' \in \mathcal{A}(s')} Q(s', a') - Q(s, a) \right),
\]

where \( s \) represents a state, \( a \) is the action taken by the agent in that state, \( \alpha > 0 \) is the step size/learning rate, \( R(s) \) the reward experienced by the agent in state \( s \), \( 0 < \gamma \leq 1 \) is the discount factor, and \( \mathcal{A}(s') \) is the set of actions available in the state \( s' \), which resulted from taking the action \( a \) in \( s \).

A related learning algorithm called SARSA has the similar update rule

\[
Q(s, a) \leftarrow Q(s, a) + \alpha \times \left( R(s) + \gamma Q(s', a') - Q(s, a) \right),
\]

where \( a' \) is the actual action taken by the agent from state \( s' \).

The two algorithms (Q-learning and SARSA) are equivalent for a greedy agent that simply chooses actions to exploit its current \( Q \) table values. However, they differ for an agent undergoing exploration. Updates in Q-learning learning ignore the exploration-based policy (hence, it is called an “off-policy” algorithm), while SARSA updates are based on the actual policy being followed (hence, an “on-policy” algorithm).

Describe the meaningful differences that may result between an exploring Q-learning and an exploring SARSA agent. In particular, what are possible relative strengths of a Q-learner over a SARSA agent? Conversely, when and why might a SARSA agent be preferred?

\textit{Hint:} An epsilon-greedy exploration policy is only one among many possibilities. Exploration could be frequency based (try every action at least \( N \) times), random (\( \epsilon = 1 \)), adversarial, or even coordinated with other agents. Because these two algorithms differ only in the context of exploration, be sure to consider a wide range of exploration-driven behaviors and the implications for what is learned.