Midterm Practice Problem Solutions

General notes: This is by no means representative of an actual midterm you might expect to see, in terms of length, difficulty, or content. So no worries if some of the questions turned out to be more tedious than they were worth.

Search space formulation

You have a U.S. tablespoon, which holds 3 teaspoons, and an Australian tablespoon, which hold 4 teaspoons. In this idealized problem, we’ll assume that you can pour without spilling, even when you’re holding one spoon in each hand. You have a bag of sugar, and both spoons start off empty. You need 2 teaspoons of sugar for a recipe.

Formulate this problem as a search problem, defining the state space, operators, initial state, and goal state. (Hint: all of your operators will involve transferring sugar from one place to another; there will be 6 of them).

Let A = the Australian tablespoon
Let U = the US tablespoon
Let (Z, x) mean that spoon Z has x teaspoons of sugar in it

Initial state: (A, 0), (U, 0)

Goal state: (A, 2) or (U, 2) -- note that we don’t care what’s in the other spoon, just so long as we have 2 teaspoons of sugar in one of the spoons

Operators:
1. Fill A from the sugar bag, result: (A, 4)
2. Fill U from the sugar bag, result: (U, 3)
3. Dump contents of A into sugar bag, result: (A, 0)
4. Dump contents of U into sugar bag, result: (U, 0)
5. Transfer sugar from A to U
6. Transfer sugar from U to A

Operations 5 & 6 may not fill the destination spoon completely if there’s not enough sugar in the source spoon, and it may not empty the source spoon completely, if there is too much sugar to fill the destination spoon.
DFS/BFS

a. We are searching a maze, where in each step you can move N, E, S, W one square, but only where there are white squares. We start at S, and want to get to E. Assume that, when expanding a node, we explore possible alternatives in the order N (first), E, S, W (last). Number the squares in the diagram below in the order in which they would be visited by a Depth First Search algorithm.

Note that both the start square and the end square count as visited.

b. Now, number the squares in the diagram below in the order in which they would be visited by a Breadth First Search algorithm.
A*
Consider the search problem below.

Suppose we ask A* search to compute the least-cost path from S to G in this graph. The arrows indicate possible successor states, and the numbers on them indicate the path costs. The ‘h=’ numbers show the (admissible) heuristic values used for each state.

Fill in the following table with the contents of the priority queue after each iteration of the A* search algorithm loop (the best – lowest scoring – entry in the priority queue is written to the left). After the iteration that A* decides to finish and return, write “Finished”. (That is, you are not expected to fill in all lines of the table. At some point the A* algorithm will terminate.) We’ve started the table for you.

<table>
<thead>
<tr>
<th>After iteration</th>
<th>The priority queue contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>(S, 3)</td>
</tr>
<tr>
<td>1</td>
<td>(B, 3), (A, 5), (C, 7)</td>
</tr>
<tr>
<td>2</td>
<td>(E, 3), (A, 5), (C, 6)</td>
</tr>
<tr>
<td>3</td>
<td>(A, 5), (C, 6), (G, 7)</td>
</tr>
<tr>
<td>4</td>
<td>(C, 4), (D, 5), (G, 7)</td>
</tr>
<tr>
<td>5</td>
<td>(G, 4), (D, 5), (E, 5)</td>
</tr>
<tr>
<td>6</td>
<td>FINISHED</td>
</tr>
</tbody>
</table>

Note that as better values for nodes in the pqueue are generated, these values are updated.
Delayed reinforcement learning

Consider the following 3x3 grid world, with initial node values as shown:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>.11</td>
<td>.19</td>
<td>.10</td>
</tr>
<tr>
<td>1</td>
<td>.47</td>
<td>.71</td>
<td>.76</td>
</tr>
<tr>
<td>2</td>
<td>.92</td>
<td>.61</td>
<td>.12</td>
</tr>
</tbody>
</table>

There is a reward of 5 for moving up or right into (2, 2)
There is a cost of 0.5 for moving up or right into (2,1)
There is a cost of 0.5 for moving up or right into (1,1)

\[ \gamma = 0.9 \text{ and } \beta = 0.3 \]

If the robot starts at (2, 0), show where the robot will be after six moves, using the policy decision function you implemented in your programming assignment. Also, draw an updated value grid showing the result of value iteration at the end of the six moves. Assume that once the robot reaches (2, 2), it is transported back to (2, 0) without expending a move.

This is an example of a problem with too much busy work to make it onto an exam. I wanted to illustrate how costs and rewards interact with each other. All in all, it probably wasn’t worth it.

**Iteration one:**
Robot moves from (2,0) to (2,1)
\[
V(2,0) = (1 - 0.3)*(.10) + (0.3)*(-0.5 + (0.9)*(.76)) \\
= .1252
\]

**Iteration two:**
Robot moves from (2,1) to (2,2)
\[
V(2,1) = (1 - 0.3)*(.76) + (0.3)*(5 + (0.9)*(.12)) \\
= 2.0644
\]

**Iteration three:**
Robot moves from (2,0) to (2,1)
\[
V(2,0) = (1 - 0.3)*(.1252) + (0.3)*(-0.5 + (0.9)*(2.0644)) \\
= .495
\]

**Iteration four:**
Robot moves from (2,1) to (2,2)
\[
V(2,1) = (1 - 0.3)*(2.0644) + (0.3)*(5 + (0.9)*(.12)) \\
= 2.977
\]

**Iteration five:**
Robot moves from (2,0) to (2,1)
\[
V(2,0) = (1 - 0.3)*(.495) + (0.3)*(-0.5 + (0.9)*(2.977)) \\
= 1.000
\]

**Iteration six:**
Robot moves from (2,1) to (2,2)
\[
V(2,1) = (1 - 0.3)*(2.977) + (0.3)*(5 + (0.9)*(.12)) \\
= 3.616
\]

Final grid just changes (2,0) to 1.000 and (2,1) to 3.616
Consider this game tree for a two-player game. The first player wants to maximize his score, and the second player wants to minimize his score. The players will take turns, with the maximizing player to move first. The values of nodes A and B are positive integers.

a. For A = B = 5, write the minimax value backed up at each node in the tree. Then circle the node(s) that would be pruned using the alpha-beta procedure.

b. If B = 5, for what value(s) of A (if any) will A be the node that minimax finds, assuming both players play optimally, and in the case of a tie the left most node is chosen?

Assuming A is an integer, only A = 5 or A = 6

c. If A = 5, for what value(s) of B (if any) will B be the node that minimax finds, assuming both players play optimally, and in the case of a tie the left most node is chosen?

There are no values of B that will cause B to be chosen.
Propositional Logic

Recall that in propositional logic resolution, we can combine two clauses \( \{A, \neg B\} \) and \( \{B, C\} \) to yield a new clause \( \{A, C\} \) – this is called the resolution inference rule. Use truth tables to prove that this rule is sound.

The truth table:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>(\neg B)</th>
<th>(A \lor B)</th>
<th>(\neg B \lor C)</th>
<th>((A \lor B) \land (\neg B \lor C))</th>
<th>(A \lor C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

Note that whenever \((A \lor B) \land (\neg B \lor C)\) is true, \(A \lor C\) is true (though not vice versa). Soundness means that any sentence that we infer from our knowledge base must be true, given that the sentences in the knowledge base are true (i.e., the inferred sentence must be entailed by the knowledge base). Our knowledge base includes \((A \lor B) \land (\neg B \lor C)\), and we infer \(A \lor C\).
Predicate Logic

Assume that we have defined the following relations:

Student(x) – x is a student
Course(x) – x is a Course
Taking(x, y) – x is taking course y
Lazy(x) – x is lazy
Prepared(x, y) – x prepared for subject y
Ace(x, y) – x will ace the midterm in subject y

And that we define the following constants: Me, CS121

And that our knowledge base contains the following sentences:

∀x (Student(x) ∧ Taking(x, CS121)) ⊃ ¬Lazy(x)
∀x, y (Student(x) ∧ ¬Lazy(x) ∧ Course(y) ∧ Taking(x, y)) ⊃ Prepared(x, y)
∀x, y (Student(x) ∧ Course(y) ∧ Prepared(x, y)) ⊃ Ace(x, y)

Student(Me)
Course(CS121)
Taking(Me, CS121)

a. Convert the sentences in the knowledge base to CNF clauses.

There is a mistake in the scope of the universal quantifiers for all three sentences – the quantification should be over the entire sentence in each case.

{¬Student(x1), ¬Taking(x1, CS121), ¬Lazy(x1)}
{¬Student(x2), Lazy(x2), ¬Course(y1), ¬Taking(x2, y1), Prepared(x2, y1)}
{¬Student(x3), ¬Course(y2), ¬Prepared(x3, y2), Ace(x3, y2)}

{Student(Me)}
{Course(CS121)}
{Taking(Me, CS121)}
b. Use resolution chaining to prove Ace(ME, CS121)

[1] \{\neg \text{Ace(ME, CS121)}\} \quad \text{(negated conclusion)}
[2] \{\neg \text{Student(x3)}, \neg \text{Course(y2)}, \neg \text{Prepared(x3,y2)}, \text{Ace(x3,y2)}\} \quad \text{(given)}
[3] \{\neg \text{Student(ME)}, \neg \text{Course(CS121)}, \neg \text{Prepared(ME, CS121)}\}
   \quad \text{Resolve [1],[2] using } s\{\text{ME/x3, CS121/y2}\}
[4] \{\text{Course(CS121)}\} \quad \text{(given)}
[5] \{\neg \text{Student(ME)}, \neg \text{Prepared(ME, CS121)}\} \quad \text{Resolve [3],[4]}
[6] \{\text{Student(ME)}\} \quad \text{(given)}
[7] \{\neg \text{Prepared(ME, CS121)}\} \quad \text{Resolve [5],[6]}
[8] \{\neg \text{Student(x2)}, \text{Lazy(x2)}, \neg \text{Course(y1)}, \neg \text{Taking(x2,y1)}, \text{Prepared(x2,y1)}\}
   \quad \text{(given)}
[9] \{\text{Lazy(ME)}, \neg \text{Course(y1)}, \neg \text{Taking(ME,y1)}, \text{Prepared(ME,y1)}\}
   \quad \text{Resolve [6],[8] using } s\{\text{ME/x2}\}
[10] \{\text{Lazy(ME)}, \neg \text{Taking(ME,CS121)}, \text{Prepared(ME,CS121)}\}
   \quad \text{Resolve [4],[9] using } s\{\text{CS121/y1}\}
[11] \{\text{Lazy(ME)}, \neg \text{Taking(ME,CS121)}\} \quad \text{Resolve [7],[10]}
[12] \{\text{Taking(ME, CS121)}\} \quad \text{(given)}
[13] \{\text{Lazy(ME)}\} \quad \text{Resolve [11],[12]}
[14] \{\neg \text{Student(x1)}, \neg \text{Taking(x1, CS121)}, \neg \text{Lazy(x1)}\} \quad \text{(given)}
[15] \{\neg \text{Taking(ME, CS121)}, \neg \text{Lazy(ME)}\}
   \quad \text{Resolve [6],[14] using } s\{\text{ME/x}\}
[16] \{\neg \text{Lazy(ME)}\} \quad \text{Resolve [12],[15]}
[17] \text{NIL} \quad \text{Resolve [13],[16]}

c. Translate the following English sentences into predicate logic, then convert them to CNF:

Any student who takes CS 121 is not lazy.

\[ \forall x ((\text{Student}(x) \land \text{Taking}(x, \text{CS121})) \supset \neg \text{Lazy}(x)) \]

*Yup, it’s the same one as in the KB.*

There is a course that all students are taking.

\[ \exists x \forall y (\text{Course}(x) \land \text{Student}(y) \supset \text{Taking}(y, x)) \]

There is at least one student in a course I am taking who will ace all of his midterms.

\[ \exists x \exists y (\text{Student}(x) \land \text{Course}(y) \land \text{Taking(ME, y}) \land \text{Taking}(x, y) \land \forall z ((\text{Taking}(x, z) \land \text{Course}(z)) \supset \text{Ace}(x, z))) \]