Supervision Assignments in AI
Easter Term 2019
Set 2: Adversarial Seach, CSPs, Knowledge Representation and Reasoning

Supervisor : Dionysis Manousakas
dm754@cam.ac.uk

May 16, 2019

1. (*AIMA5.7*) Prove the following assertion: For every game tree, the utility obtained by MAX using minimax decisions against a suboptimal MIN will never be lower than the utility obtained playing against an optimal MIN. Can you come up with a game tree in which MAX can do still better using a suboptimal strategy against a suboptimal MIN?

   Consider a MIN node whose children are terminal nodes. If MIN plays suboptimally, then the value of the node is greater than or equal to the value it would have if MIN played optimally. Hence, the value of the MAX node that is the MIN node's parent can only be increased. This argument can be extended by a simple induction all the way to the root. If the suboptimal play by MIN is predictable, then one can do better than a minimax strategy. For example, if MIN always falls for a certain kind of trap and loses, then setting the trap guarantees a win even if there is actually a devastating response for MIN.

2. (*AIMA5.12*) Describe how the minimax and alphabeta algorithms change for two-player, nonzero-sum games in which each player has a distinct utility function and both utility functions are known to both players. If there are no constraints on the two terminal utilities, is it possible for any node to be pruned by alphabeta? What if the players utility functions on any state differ by at most a constant k, making the game almost cooperative?

The minimax algorithm for non-zero-sum games works exactly as for multiplayer games, described on p.165-6, *AIMA*; that is, the evaluation function is a vector of values, one for each player, and the backup step selects whichever vector has the highest value for the player whose turn it is to move. Well be backing up a vector of evaluations and at each level the player will choose what is best for him, even if it is also good for the other player. Thus if we assign increasingly positive values for states increasingly better for Max and increasingly negative values for states increasingly better for Min, then minimax will work unmodified. If both players have increasingly positive values, each player just picks the maximum value, so its a maximax algorithm.
The example at the end of Section 5.2.2 (p.165) *AIMA*, shows that alphabeta pruning is not possible in general non-zero-sum games, because an unexamined leaf node might be optimal for both players. Alphabeta pruning will not work because built into it is the idea that whats good for max is bad for min for example min wont let max go down a path since min can force something worse, so max knowing this doesn't have to explore that path. But without zero-sum assumption, the same state could be good for both min and max; you can't assume that just because max likes it that min wont, and vice versa. An unexamined leaf node may have the best outcome for both players. So pruning can only hurt Max (or Min) and the players effectively wind up collaborating by being selfish. This same example could be true of course in the constrained utilities, so the constraint doesn't help pruning.

3. (*AIMA5.14*) Prove that alphabeta pruning takes time $O(2^{m/2})$ with optimal move ordering, where m is the maximum depth of the game tree.

Even though we have a perfectly ordered list of children at each node, the algorithm will have to explore all the child nodes of the 1st player to find the best value. Thus will have a branching factor of b. For the second player, it is enough to expand the first child. Because that value, being the best value, will make all values but the first value of the previous node pruned. Thus will have an effective branching factor of 1. This means that we will be having two types of threads down the tree when optimal ordering is there. Type A: will have number 1 in even levels and some other number $x(1 < x < b)$. Type B: will have number 1 in odd levels and some other number $y(1 < y < b)$.
Given that we are at the ith level, i. Number of type A threads: $b^{\lceil i/2 \rceil}$ Note: we have to take the ceiling value here because we are counting threads that have 1 in even levels. ii. Number of type B threads: $b^{\lfloor i/2 \rfloor}$ (Note: we have to take the floor value here because we are counting threads that have 1 in odd levels.) Now note that the special thread 111111....1 has been counted twice! (Both as type A and type B) Thus if we have m levels the number of nodes visited will be; $O(b^{\lceil m/2 \rceil} + b^{\lfloor m/2 \rfloor} - 1)$ [-1 to prevent 1111...1 thread being counted twice] Simplifying we get $O(2^{m/2})$.

4. (*AIMA5.18*) Prove that with a positive linear transformation of leaf values (i.e., transforming a value x to $ax + b$ where $a > 0$), the choice of move remains unchanged in a game tree, even when there are chance nodes.

The general strategy is to reduce a general game tree to a one-ply tree by induction on the depth of the tree. The inductive step must be done for min, max, and chance nodes, and simply involves showing that the transformation is carried though the node. Suppose that the values of the descendants of a node are $x_1 ... x_n$, and that the transformation is $ax + b$, where a is positive. We have

$$min(ax_1 + b, ax_2 + b, ..., ax_n + b) = amin(x_1, x_2, ..., x_n) + b$$

$$max(ax_1 + b, ax_2 + b, ..., ax_n + b) = amax(x_1, x_2, ..., x_n) + b$$

$$p_1(ax + b) + p_2(ax_2 + b) + ... + p_n(ax_n + b) = a(p_1x_1 + p_2x_2 + ... + p_nx_n) + b$$

Hence the problem reduces to a one-ply tree where the leaves have the values from the original tree multiplied by the linear transformation. Since $x > y \implies ax + b > ay + b$ if $a > 0$, the best choice at the root will be the same as the best choice in the original tree.

5. (*AIMA6.6*) Show how a single ternary constraint such as "$A + B = C$" can be turned into three binary constraints by using an auxiliary variable. You may assume finite domains. (Hint: Consider a new variable that takes on values that are pairs of other values, and consider constraints such as "X is the first element of the pair Y .") Next, show how constraints with more than three variables can be treated similarly. Finally, show how unary constraints can be eliminated by altering the domains of variables. This completes the demonstration that any CSP can be transformed into a CSP with only binary constraints.

The problem statement sets out the solution fairly completely. To express the ternary constraint on A, B and C that $A + B = C$, we first introduce a new variable, AB. If the domain of A and B is the set of numbers N, then the domain of AB is the set of pairs of numbers from N, i.e. $N \times N$. Now there are three binary constraints, one between A and AB saying that the value of A must be equal to the first element of the pair-value of AB; one between B and AB saying that the value of B must equal the second element of the value of AB; and finally one that says that the sum of the pair of numbers that is the value of AB must equal the value of C. All other ternary constraints can be handled similarly.

Now that we can reduce a ternary constraint into binary constraints, we can reduce a 4-ary constraint on variables A, B, C, D by first reducing A, B, C to binary constraints as shown above, then adding back D in a ternary constraint with AB and C, and then reducing this ternary constraint to binary by introducing CD. By induction, we can reduce any n-ary constraint to an $(n1)$-ary constraint.

We can stop at binary, because any unary constraint can be dropped, simply by moving the effects of the constraint into the domain of the variable

6. (*AIMA6.12*) What is the worst-case complexity of running AC-3 on a tree-structured CSP?

On a tree-structured graph, no arc will be considered more than once, so the AC- 3 algorithm is $O(ED)$, where E is the number of edges and D is the size of the largest domain.

7. (*AIMA6.13*) AC-3 puts back on the queue every arc (X_k, X_i) whenever any value is deleted from the domain of X_i, even if each value of X_k is consistent with several remaining values of X_i. Suppose that, for every arc (X_k, X_i), we keep track of the number of remaining values of X_i that are consistent with each value of X_k. Explain how to update these numbers efficiently and hence show that arc consistency can be enforced in total time $O(n^2d^2)$.

The basic idea is to preprocess the constraints so that, for each value of X_i, we keep track of those variables X_k for which an arc from X_k to X_i is satisfied by that particular value of X_i. This data structure can be computed in time proportional to the size of the problem representation. Then, when a value of X_i is deleted, we reduce by 1 the count of allowable values for each (X_k, X_i) arc recorded under that value.

8. (*AIMA7.21*) Is a randomly generated 4-CNF sentence with n symbols and m clauses more or less likely to be solvable than a randomly generated 3-CNF sentence with n symbols and m clauses?

It is more likely to be solvable: adding literals to disjunctive clauses makes them easier to satisfy.

9. (*AIMA8.2*) Consider a knowledge base containing just two sentences: $P(a)$ and $P(b)$. Does this knowledge base entail $\forall x\ P(x)$?

The knowledge base does not entail $\forall x P(x)$. To show this, we must give a model where $P(a)$ and $P(b)$ but $\forall x\ P(x)$ is false. Consider any model with three domain elements, where a and b refer to the first two elements and the relation referred to by P holds only for those two elements.

10. (*AIMA8.18*) Write out the axioms required for reasoning about the wumpus's location, using a constant symbol Wumpus and a binary predicate At(Wumpus, Location). Remember that there is only one wumpus.

We need the following sentences:

$$\forall\ s_1\ Smelly(s_1) \iff \exists\ s_2\ Adjacent(s_1, s_2)\ \wedge\ At(Wumpus, s_2)$$

$$\exists\ s_1\ At(Wumpus, s_1) \wedge\ \forall\ s_2\ (s_1 \neq\ s_2) \implies \neg At(Wumpus, s_2)$$

11 (*AIMA 8.22*) Write in first-order logic the assertion that every key and at least one of every pair of socks will eventually be lost forever, using only the following vocabulary: Key(x), x is a key; Sock(x), x is a sock; Pair (x, y), x and y are a pair; Now, the current time; Before(t1, t2), time t1 comes before time t2; Lost (x, t), object x is lost at time t.

$$\forall\, k\ Key(k) \implies [\exists t_0\ Before(Now, t_0) \wedge \forall t\ Before(t_0, t) \implies Lost(k, t)]$$

$$\forall\, s_1,\ s_2\ Sock(s_1) \wedge Sock(s_2) \wedge Pair(s_1, s_2) \implies$$
$$[\exists t_1\ Before(Now, t_1) \wedge \forall t\ Before(t_1, t) \implies Lost(s_1, t)]$$
$$\vee[\exists t_2\ Before(Now, t_2) \wedge \forall t\ Before(t_2, t) \implies Lost(s_2, t)]$$

12. (*AIMA 9.2*) From $Likes(Jerry, IceCream)$ it seems reasonable to infer $\exists x\ Likes(x, IceCream)$. Write down a general inference rule, **Existential Introduction**, that sanctions this inference. State carefully the conditions that must be satisfied by the variables and terms involved.

For any sentence a containing a ground term g and for any variable v not occuring in a, we have $\frac{a}{\exists v SUBST^*(\{g/v\},a)}$, where $SUBST^*$ is a function that substitutes any or all of the occurrences of g with v. Notice that substituting just one occurrence and applying the rule multiple times is not the same, because it results in a weaker conclusion. For example, $P(a, a)$ should entail $\exists x\ P(x, x)$ rather than the weaker $\exists x, y\ P(x, y)$.

13. (*AIMA 9.18*) Suppose that successor(X,Y) is true when state Y is a successor of state X; and that goal(X) is true when X is a goal state. Write a definition for solve(X,P), which means that P is a path (list of states) beginning with X, ending in a goal state, and consisting of a sequence of legal steps as defined by successor. You will find that depth-first search is the easiest way to do this. How easy would it be to add heuristic search control?

Once you understand how Prolog works, the answer is easy:

```
solve(X,[X]) :- goal(X).
solve(X,[X|P]) :- successor(X,Y), solve(Y,P).
```

We could render this in English as "Given a start state, if it is a goal state, then the path consisting of just the start state is a solution. Otherwise, find some successor state such that there is a path from the successor to the goal; then a solution is the start state followed by that path."

Notice that `solve` can not only be used to find a path P that is a solution, it can also be used to verify that a given path is a solution. If you want to add heuristics (or even breadth-first search), you need an explicit queue.

14. (*AIMA 10.5*) A finite Turing machine has a finite one-dimensional tape of cells, each cell containing one of a finite number of symbols. One cell has a read and write head above it. There is a finite set of states the machine can be in, one of which is the accept state. At each time step, depending on the symbol on the cell under the head and the machines current state, there are a set of actions we can choose from. Each action involves writing a symbol to the cell under the head, transitioning the machine to a state, and optionally moving the head left or right. The mapping that determines

which actions are allowed is the Turing machines program. Your goal is to control the machine into the accept state. Represent the Turing machine acceptance problem as a planning problem. If you can do this, it demonstrates that determining whether a planning problem has a solution is at least as hard as the Turing acceptance problem, which is PSPACE-hard.

One representation is as follows. We have the predicates:
a. $HeadAt(c)$: tape head at cell location c, true for exactly one cell.
b. $State(s)$: machine state is s, true for exactly one cell.
c. $ValueOf(c, v)$: cell c's value is v.
d. $LeftOf(c_1, c_2)$: cell c_1 is one step left from cell c_2.
e. $TransitionLeft(s_1, v_1, s_2, v_2)$: the machine in state s_1 upon reading a cell with value v_1 may write value v_2 to the cell, change state to s_2, and transition to the left.
f. $TransitionRight(s_1, v_1, s_2, v_2)$: the machine in state s_1 upon reading a cell with value v_1 may write value v_2 to the cell, change state to s_2, and transition to the right.
The predicates $HeadAt$, $State$, and $ValueOf$ are fluents, the rest are constant descriptions of the machine and its tape. Two actions are required:

$$Action(RunLeft(s_1, c_1, v_1, , s_2, c_2, v_2),$$
$$PRECOND : State(s_1) \wedge HeadAt(c_1) \wedge ValueOf(c_1, v_1)$$
$$; \wedge TransitionLeft(s_1, v_1, s_2, v_2) \wedge LeftOf(c_2, c_1)$$
$$EFFECT : \neg State(s_1) \wedge State(s2) \wedge \neg HeadAt(c_1) HeadAt(c_2)$$
$$\wedge \neg ValueOf(c_1, v_1) \wedge ValueOf(c_1, v_2))$$

$$Action(RunRight(s_1, c_1, v_1, , s_2, c_2, v_2),$$
$$PRECOND : State(s_1) HeadAt(c_1) \wedge ValueOf(c_1, v_1)$$
$$; \neg TransitionRight(s_1, v_1, s_2, v_2) \wedge LeftOf(c_1, c_2)$$
$$EFFECT : \neg State(s_1) \wedge State(s_2) \wedge \neg HeadAt(c_1) \wedge HeadAt(c_2)$$
$$\wedge \neg ValueOf(c_1, v_1) \wedge ValueOf(c_1, v_2))$$

The goal will typically be to reach a fixed accept state. A simple example problem is:

$$Init(HeadAt(C_0) \wedge State(S_1) \wedge ValueOf(C_0, 1) \wedge ValueOf(C_1, 1)$$
$$\wedge ValueOf(C_2, 1) \wedge ValueOf(C_3, 0) \wedge LeftOf(C_0, C_1) \wedge LeftOf(C_1, C_2)$$
$$\wedge LeftOf(C_2, C_3) \wedge TransitionLeft(S_1, 1, S_1, 0) \wedge TransitionLeft(S_1, 0, Saccept, 0)$$
$$Goal(State(S_{accept}))$$

Note that the number of literals in a state is linear in the number of cells, which means a polynomial space machine require polynomial state to represent.

15. *AIMA 10.12* a. Would bidirectional state-space search be a good idea for planning?

    b. What about bidirectional search in the space of partial-order plans?

c. Devise a version of partial-order planning in which an action can be added to a plan if its preconditions can be achieved by the effects of actions already in the plan. Explain how to deal with conflicts and ordering constraints. Is the algorithm essentially identical to forward state-space search?

a. It is feasible to use bidirectional search, because it is possible to invert the actions. However, most of those who have tried have concluded that biderectional search is generally not efficient, because the forward and backward searches tend to miss each other. This is due to the large state space. A few planners, such as PRODIGY (Fink and Blythe, 1998) have used bidirectional search.
b. Again, this is feasible but not popular. PRODIGY is in fact (in part) a partial-order planner: in the forward direction it keeps a total-order plan (equivalent to a state-based planner), and in the backward direction it maintains a tree-structured partial-order plan.
c. An action A can be added if all the preconditions of A have been achieved by other steps in the plan. When A is added, ordering constraints and causal links are also added to make sure that A appears after all the actions that enabled it and that a precondition is not disestablished before A can be executed. The algorithm does search forward, but it is not the same as forward state-space search because it can explore actions in parallel when they dont conflict. For example, if A has three preconditions that can be satisfied by the non-conflicting actions B, C, and D, then the solution plan can be represented as a single partial-order plan, while a state-space planner would have to consider all 3! permutations of B, C, and D.