Sample Solution to Homework 2

Com S 472/572

September 30, 2020

4.1 (a) Local beam search with $k = 1$ is hill-climbing search.

(b) Local beam search with one initial state and no limit on the number of states retained, resembles breadth-first search in that it adds one complete layer of nodes before adding the next layer. Starting from one state, the algorithm would be essentially identical to breadth-first search except that each layer is generated all at once.

(c) Simulated annealing with $T = 0$ at all times: ignoring the fact that the termination step would be triggered immediately, the search would be identical to first-choice hill climbing because every downward successor would be rejected with probability 1. (Exercise may be modified in future printings.)

(d) Simulated annealing with $T = \infty$ at all times is a random-walk search: it always accepts a new state.

(e) Genetic algorithm with population size $N = 1$: if the population size is 1, then the two selected parents will be the same individual; crossover yields an exact copy of the individual; then there is a small chance of mutation. Thus, the algorithm executes a random walk in the space of individuals.

4.7 A sequence of actions is a solution to a belief state problem if it takes every initial physical state to a goal state. We can relax this problem by requiring it take only *some* initial physical state to a goal state. To make this well defined, we'll require that it finds a solution for the physical state with the most costly solution. If $h^*(s)$ is the optimal cost of solution starting from the physical state s, then

$$h(S) = \max_{s \in S} h^*(s)$$

is the heuristic estimate given by this relaxed problem. This heuristic assumes any solution to the most difficult state the agent things possible will solve all states.

On the sensorless vacuum cleaner problem in Figure 4.14, h correctly determines the optimal cost for all states except the central three states (those reached by [*suck*], [*suck*, *left*] and [*suck*, *right*]) and the root, for which h estimates to be 1 unit cheaper than they really are. This means A* will expand these three central nodes, before marching towards the solution.

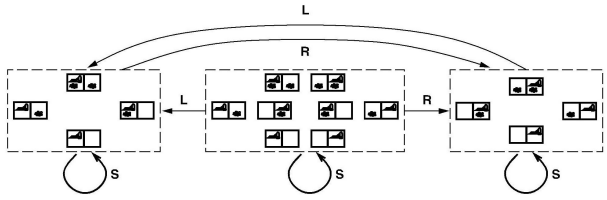
4.8 **(a)** An action sequence is a solution for belief state b if performing it starting in any state $s \in b$ reaches a goal state. Since any state in a subset of b is in b, the result is immediate.

Any action sequence which is not a solution for belief state b is also not a solution for any superset; this is the contrapositive of what we've just proved. One cannot, in general, say anything about arbitrary supersets, as the action sequence need not lead to a goal on the states outside of b. One can say, for example, that if an action sequence solves a belief state b and a belief state b' then it solves the union belief state $b \cup b'$.

(b) On expansion of a node, do not add to the frontier any child belief state which is a superset of a previously explored belief state.

(c) If you keep a record of previously solved belief states, add a check to the start of OR-search to check whether the belief state passed in is a subset of a previously solved belief state, returning the previous solution in case it is.

4.10 The belief state space is shown in the figure below. No solution is possible because no path leads to a belief state all of whose elements satisfy the goal. If the problem is fully observable, the agent reaches a goal state by executing a sequence such that *Suck* is performed only in a dirty square. This ensures deterministic behavior and every state is obviously solvable.



The belief state space for the sensorless vacuum world under Murphy's law.

5.8 **(a)** (5) The game tree, complete with annotations of all minimax values, is shown in Figure 1.

(b) (5) The "?" values are handled by assuming that an agent with a choice between winning the game and entering a "?" state will always choose the win. That is, min(–1,?) is –1 and max(+1,?) is +1. If all successors are "?", the backed-up value is "?".

(c) Standard minimax is depth-first and would go into an infinite loop. It can be fixed by comparing the current state against the stack; and if the state is repeated, then return a "?" value. Propagation of "?" values is handled as above. Although it works in this case, it does not always work because it is not clear how to compare "?" with a drawn position; nor is it clear how to handle the comparison when there are wins of different degrees (as in backgammon). Finally, in games with chance nodes, it is unclear how to compute the average of a number and a "?". Note that it is not correct to treat repeated states automatically as drawn positions; in this

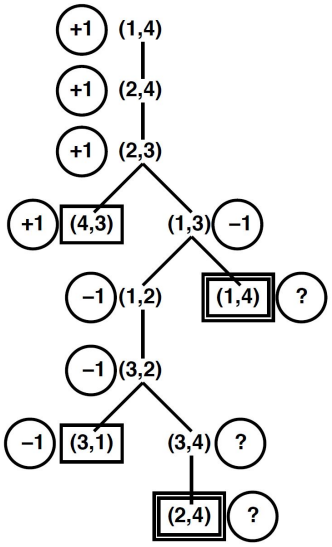


Figure 1: The game tree for the four-square game in Exercise 5.8. Terminal states are in single boxes, loop states in double boxes. Each state is annotated with its minimax value in a circle.

example, both (1,4) and (2,4) repeat in the tree but they are won positions. What is really happening is that each state has a well-defined but initially unknown value. These unknown values are related by the minimax equation at the bottom of 164. If the game tree is acyclic, then the minimax algorithm solves these equations by propagating from the leaves. If the game tree has cycles, then a dynamic programming method must be used, as explained in Chapter 17. These algorithms can determine whether each node has a well-determined value (as in this example) or is really an infinite loop in that both players prefer to stay in the loop (or have no choice). In such a case, the rules of the game will need to define the value (otherwise the game will never end). In chess, for example, a state that occurs 3 times (and hence is assumed to be desirable for both players) is a draw.

(d) This question is a little tricky. One approach is a proof by induction on the size of the game. Clearly, the base case $n = 3$ is a loss for A and the base case $n = 4$ is a win for A. For any $n > 4$, the initial moves are the same: A and B both move one step towards each other. Now, we can see that they are engaged in a subgame of size $n - 2$ on the squares $[2, \ldots, n - 1]$, except that there is an extra choice of moves on squares 2 and $n - 1$. Ignoring this for a moment, it is clear that if the "$n - 2$" is won for A, then A gets to the square $n - 1$ before B gets to square 2 (by the definition of winning) and therefore gets to n before B gets to 1, hence the "n" game is won for A. By the same line of reasoning, if "$n - 2$" is won for B then "n" is won for B. Now, the presence of the extra moves complicates the issue, but not too much. First, the player who is slated to win the subgame $[2, \ldots, n - 1]$ never moves back to his home square. If the player slated to lose the subgame does so, then it is easy to show that he is bound to lose the game itself - the other player simply moves forward and a subgame of size $n - 2k$ is played one step closer to

the loser's home square.

5.9 For **a**, there are at most 9! games. (This is the number of move sequences that fill up the board, but many wins and losses end before the board is full.) For **b-e**, Figure 2 shows the game tree, with the evaluation function values below the terminal nodes and the backed-up values to the right of the non-terminal nodes. The values imply that the best starting move for X is to take the center. The terminal nodes with a bold outline are the ones that do not need to be evaluated, assuming the optimal ordering.

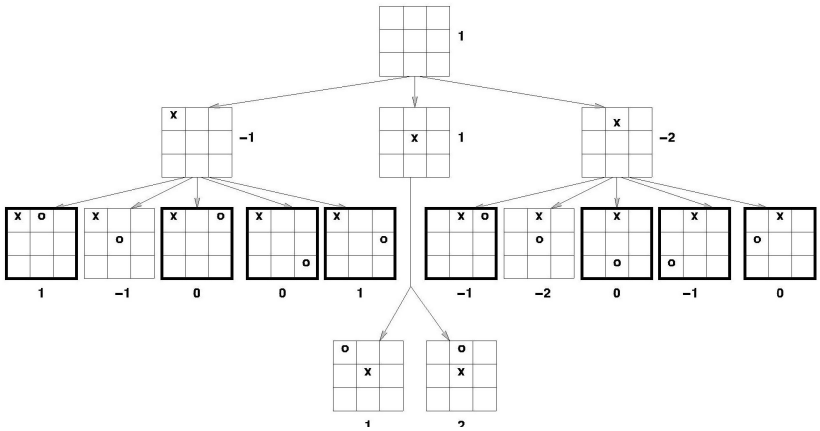


Figure 2: Part of the game tree for tic-tac-toe.

5.14 This question is not as hard as it looks. The derivation below leads directly to a definition of α and β values. The notation n_i refers to (the value of) the node at depth i on the path from the root to the leaf node n_j. Nodes $n_{i1} \ldots n_{ib_i}$ are the siblings of node i.

(a) We can write $n_2 = \max(n_3, n_{31}, \ldots, n_{3b_3})$, giving

$$n_1 = \min(\max(n_3, n_{31}, \ldots, n_{3b_3}), n_{21}, \ldots, n_{2b_2})$$

Then n_3 can be similarly replaced, until we have an expression containing n_j itself.

(b) In terms of the l and r values, we have

$$n_1 = \min(l_2, \max(l_3, n_3, r_3), r_2)$$

Again, n_3 can be expanded out down to n_j. The most deeply nested term will be $\min(l_j, n_j, r_j)$.

(c) If n_j is a max node, then the lower bound on its value only increases as its successors are evaluated. Clearly, if it exceeds l_j it will have no further effect on n_1. By extension, if it exceeds $\min(l_2, l_4, \ldots, l_j)$ it will have no effect. Thus, by keeping track of this value we can decide when to prune n_j. This is exactly what $\alpha - \beta$ does.

(d) The corresponding bound for min nodes n_k is $\max(l_3, l_5, \ldots, l_k)$.

Extra The execution of the alpha-beta pruning algorithm is shown in Figure 3.

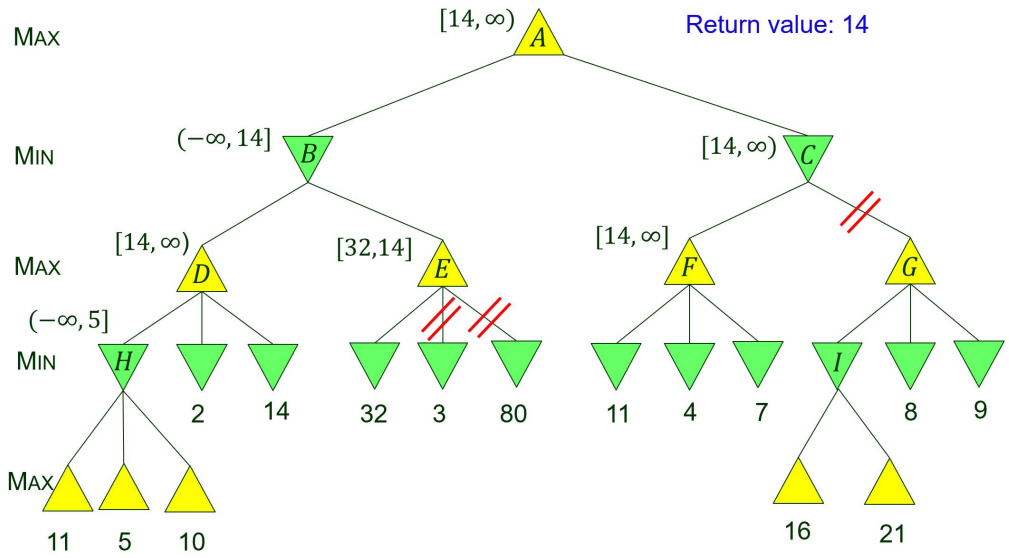


Figure 3: Alpha-beta pruning algorithm.