# EXERCISES 6

# ADVERSARIAL SEARCH AND GAMES

## 6.1 Game Theory

**Exercise 6.**ORAC

Suppose you have an oracle, $OM(s)$, that correctly predicts the opponent's move in any state. Using this, formulate the definition of a game as a (single-agent) search problem. Describe an algorithm for finding the optimal move.

The translation uses the model of the opponent $OM(s)$ to fill in the opponent's actions, leaving our actions to be determined by the search algorithm. Let $P(s)$ be the state predicted to occur after the opponent has made all their moves according to $OM$. Note that the opponent may take multiple moves in a row before we get a move, so we need to define this recursively. We have $P(s) = s$ if PLAYER$s$ is us or TERMINAL-TEST$s$ is true, otherwise $P(s) = P(\text{RESULT}(s, OM(s))$.

The search problem is then given by:

**a**. Initial state: $P(S_0)$ where $S_0$ is the initial game state. We apply $P$ as the opponent may play first

**b**. Actions: defined as in the game by ACTIONS$s$.

**c**. Successor function: $\text{RESULT}'(s, a) = P(\text{RESULT}(s, a))$

**d**. Goal test: goals are terminal states

**e**. Step cost: the cost of an action is zero unless the resulting state $s'$ is terminal, in which case its cost is $M - \text{UTILITY}(s')$ where $M = \max_s \text{UTILITY}(s)$. Notice that all costs are non-negative.

Notice that the state space of the search problem consists of game state where we are to play and terminal states. States where the opponent is to play have been compiled out. One might alternatively leave those states in but just have a single possible action.

Any of the search algorithms of Chapter 3 can be applied. For example, depth-first search can be used to solve this problem, if all games eventually end. This is equivalent to using the minimax algorithm on the original game if $OM(s)$ always returns the minimax move in $s$.

**Exercise 6.**TEPZ

Consider the problem of solving two 8-puzzles: there are two 8-puzzle boards, you can make a move on either one, and the goal is to solve both.

**a**. Give a complete problem formulation.
**b**. How large is the reachable state space? Give an exact numerical expression.
**c**. Suppose we make the problem adversarial as follows: two players take turns moving; a coin is flipped to determine the puzzle on which to make a move in that turn: heads puzzle one; tails puzzle two. The winner is the first to solve either puzzle. Which algorithm can be used to choose a move in this setting?
**d**. In the adversarial problem, will one player eventually win if both play perfectly?

**a**. Initial state: two arbitrary 8-puzzle states. Successor function: one move on an unsolved puzzle. (You could also have actions that change both puzzles at the same time; this is OK but technically you have to say what happens when one is solved but not the other.) Goal test: both puzzles in goal state. Path cost: 1 per move.
**b**. Each puzzle has $9!/2$ reachable states (remember that half the states are unreachable). The joint state space has $(9!)^2/4$ states.
**c**. This is like backgammon; expectiminimax works.
**d**. No. Consider a state in which the coin flip mandates that you work on a puzzle that is 2 steps from the goal. Should you move one step closer? If you do, your opponent wins if they get to work on that puzzle on the next turn, or on any subsequent turn before you do. So the opponent's probability of winning is *at least* $1/2 + 1/8 + 1/32 + \cdots = 2/3$. Therefore you're better off moving *away* from the goal. (There's no way to stay the same distance from the goal.)

**Exercise 6.**PUEV

Imagine that the problem in Exercise 3.ROMF, in which two friends try to meet up on the map of Romania, is modified so that one of the friends wants to avoid the other. The problem then becomes a two-player **pursuit–evasion** game. We assume now that the players take turns moving. The game ends only when the players are on the same node; the terminal payoff to the pursuer is minus the total time taken. (The evader "wins" by never losing.) An example is shown in Figure **??**.

**a**. Copy the game tree and mark the values of the terminal nodes.
**b**. Next to each internal node, write the strongest fact you can infer about its value (a number, one or more inequalities such as "$\geq 14$", or a "?").
**c**. Beneath each question mark, write the name of the node reached by that branch.
**d**. Explain how a bound on the value of the nodes in (c) can be derived from consideration of shortest-path lengths on the map, and derive such bounds for these nodes. Remember the cost to get to each leaf as well as the cost to solve it.
**e**. Now suppose that the tree as given, with the leaf bounds from (d), is evaluated from left to right. Circle those "?" nodes that would *not* need to be expanded further, given the bounds from part (d), and cross out those that need not be considered at all.
**f**. Can you prove anything in general about who wins the game on a map that is a tree?

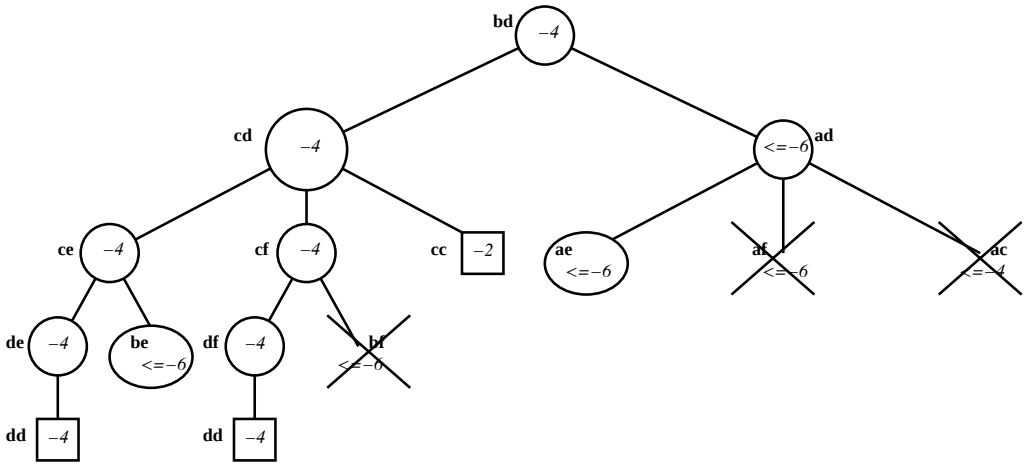


**Figure S6.1** Pursuit-evasion solution tree.

a. See Figure S6.1; the values are just (minus) the number of steps along the path from the root.

b. See Figure S6.1; note that there is both an upper bound and a lower bound for the left child of the root.

c. See figure.

d. The shortest-path length between the two players is a lower bound on the total capture time (here the players take turns, so no need to divide by two), so the "?" leaves have a capture time greater than or equal to the sum of the cost from the root and the shortest-path length. Notice that this bound is derived when the Evader plays very badly. The true value of a node comes from best play by both players, so we can get better bounds by assuming better play. For example, we can get a better bound from the cost when the Evader simply moves backwards and forwards rather than moving towards the Pursuer.

e. See figure (we have used the simple bounds). Notice that once the right child is known to have a value below –6, the remaining successors need not be considered.

f. The pursuer always wins if the tree is finite. To prove this, let the tree be rooted as the pursuer's current node. (I.e., pick up the tree by that node and dangle all the other branches down.) The evader must either be at the root, in which case the pursuer has won, or in some subtree. The pursuer takes the branch leading to that subtree. This process repeats at most $d$ times, where $d$ is the maximum depth of the original subtree, until the pursuer either catches the evader or reaches a leaf node. Since the leaf has no subtrees, the evader must be at that node.

**Exercise 6.**GAME

Describe and implement state descriptions, move generators, terminal tests, utility functions, and evaluation functions for one or more of the following stochastic games: Monopoly,

Scrabble, bridge play with a given contract, or Texas hold 'em poker.

The basic physical states of these games are fairly easy to describe. One important thing to remember for Scrabble and bridge is that the physical state is not accessible to all players and so cannot be provided directly to each player by the environment simulator; rather each player should be provided with the part of the state they can observe: the board and their hand. Particularly in bridge, to play well each player needs to maintain some best guess (or multiple hypotheses) as to the actual state of the world.

**Exercise 6.**RTSG

Describe and implement a *real-time*, *multiplayer* game-playing environment, where time is part of the environment state and players are given fixed time allocations.

Code not shown.

**Exercise 6.**PHGM

Discuss how well the standard approach to game playing would apply to games such as tennis, pool, and croquet, which take place in a continuous physical state space.

The most obvious change is that the space of states, and of actions, are now continuous. For example, in pool, the cueing direction, angle of elevation, speed, and point of contact with the cue ball are all continuous quantities.

The simplest solution is just to discretize the action space and then apply standard methods. This might work for tennis (modelled crudely as alternating shots with speed and direction), but for games such as pool and croquet it is likely to fail miserably because small changes in direction have large effects on action outcome. Instead, one must analyze the game to identify a discrete set of meaningful local goals, such as "potting the 4-ball" in pool or "laying up for the next hoop" in croquet. Then, in the current context, a local optimization routine can work out the best way to achieve each local goal, resulting in a discrete set of possible choices. Typically, these games are stochastic, so the backgammon model is appropriate provided that we use sampled outcomes instead of summing over the infinite number of possible outcomes.

Whereas pool and croquet are modelled correctly as turn-taking games, tennis is not. While one player is moving to the ball, the other player is moving to anticipate the opponent's return. This makes tennis more like the simultaneous-action games studied in Chapter 16. In particular, it may be reasonable to derive *randomized* strategies so that the opponent cannot anticipate where the ball will go.

## 6.2 Optimal Decisions in Games

**Exercise 6.**MMXO

Prove the following assertion: For every game tree, the utility obtained by MAX using minimax decisions against a suboptimal MIN will be 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?

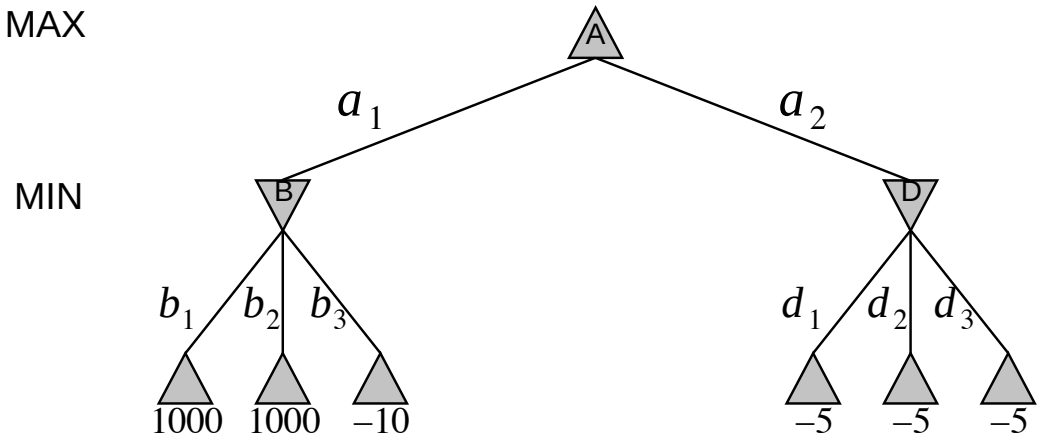


**Figure S6.2** A simple game tree showing that setting a trap for MIN by playing $a_i$ is a win if MIN falls for it, but may also be disastrous. The minimax move is of course $a_2$, with value $-5$.

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. This is shown in Figure S6.2.

**Exercise 6.**AKQP

Consider the simplified game of poker where there are exactly two players and three cards, ranked in the following order: Ace beats King and King beats Queen. Each player is dealt one of the three cards, which they view independently. Then the players simultaneously declare whether they are going to wager $1. If both players wager $1, then the player with the higher card takes all the money. If only one player wagers, then that player takes all the

money regardless of the cards. If neither player bets, no mey changes hands.

**a**. Which formalisms would work for this game? (a) expectiminimax search, (b) Greedy search, (c) CSP, (d) Minimax search?

**b**. Under what conditions would $\alpha$–$\beta$ pruning work?

**a**. expectiminimax is not appropriate because there are

**Exercise 6.**GMTR

Consider the two-player game described in Figure **??**.

**a**. Draw the complete game tree, using the following conventions:

- Write each state as $(s_A, s_B)$, where $s_A$ and $s_B$ denote the token locations.
- Put each terminal state in a square box and write its game value in a circle.
- Put *loop states* (states that already appear on the path to the root) in double square boxes. Since their value is unclear, annotate each with a "?" in a circle.

**b**. Now mark each node with its backed-up minimax value (also in a circle). Explain how you handled the "?" values and why.

**c**. Explain why the standard minimax algorithm would fail on this game tree and briefly sketch how you might fix it, drawing on your answer to (b). Does your modified algorithm give optimal decisions for all games with loops?

**d**. This 4-square game can be generalized to $n$ squares for any $n > 2$. Prove that $A$ wins if $n$ is even and loses if $n$ is odd.

a. The game tree, complete with annotations of all minimax values, is shown in Figure S6.3.

b. 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 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 149. If the game tree is acyclic, then the minimax algorithm solves these equations by

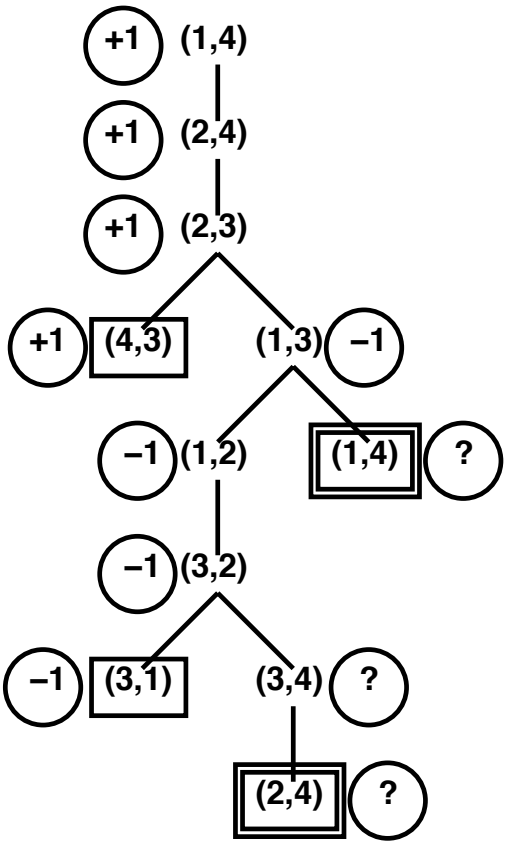


**Figure S6.3** The game tree for the four-square game in Exercise 6.8. Terminal states are in single boxes, loop states in double boxes. Each state is annotated with its minimax value in a circle.

propagating from the leaves. If the game tree has cycles, then a dynamic programming method must be used, as explained in Chapter 16. (Exercise 16.7 studies this problem in particular.) 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.

## 6.3 Heuristic Alpha–Beta Tree Search

**Exercise 6.TTTG**

This problem exercises the basic concepts of game playing, using tic-tac-toe (noughts and crosses) as an example. We define $X_n$ as the number of rows, columns, or diagonals with exactly $n$ $X$'s and no $O$'s. Similarly, $O_n$ is the number of rows, columns, or diagonals with just $n$ $O$'s. The utility function assigns $+1$ to any position with $X_3 = 1$ and $-1$ to any position with $O_3 = 1$. All other terminal positions have utility 0. For nonterminal positions, we use a linear evaluation function defined as $Eval(s) = 3X_2(s) + X_1(s) - (3O_2(s) + O_1(s))$.

**a**. Approximately how many possible games of tic-tac-toe are there?

**b**. Show the whole game tree starting from an empty board down to depth 2 (i.e., one $X$ and one $O$ on the board), taking symmetry into account.

**c**. Mark on your tree the evaluations of all the positions at depth 2.

**d**. Using the minimax algorithm, mark on your tree the backed-up values for the positions at depths 1 and 0, and use those values to choose the best starting move.

**e**. Circle the nodes at depth 2 that would *not* be evaluated if alpha–beta pruning were applied, assuming the nodes are generated in the optimal order for alpha–beta pruning.

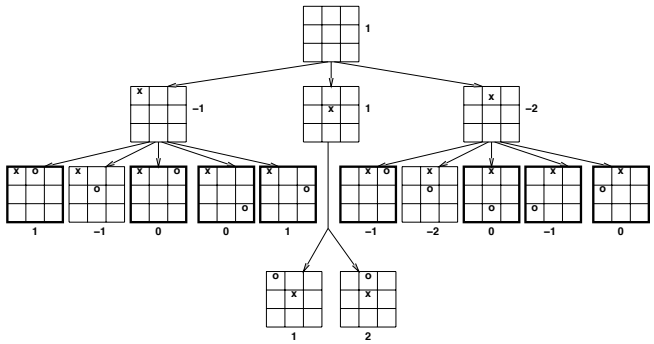

**Figure S6.4** Part of the game tree for tic-tac-toe, for Exercise 6.9.

For **a**, there are less than 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 S6.4 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.

**Exercise 6.**GTTT

Consider the family of generalized tic-tac-toe games, defined as follows. Each particular game is specified by a set $\mathcal{S}$ of *squares* and a collection $\mathcal{W}$ of *winning positions*. Each winning position is a subset of $\mathcal{S}$. For example, in standard tic-tac-toe, $\mathcal{S}$ is a set of 9 squares and $\mathcal{W}$ is a collection of 8 subsets of $\mathcal{W}$: the three rows, the three columns, and the two diagonals. In other respects, the game is identical to standard tic-tac-toe. Starting from an empty board, players alternate placing their marks on an empty square. A player who marks every square in a winning position wins the game. It is a tie if all squares are marked and neither player has won.

**a**. Let $N = |\mathcal{S}|$, the number of squares. Give an upper bound on the number of nodes in the complete game tree for generalized tic-tac-toe as a function of $N$.

**b**. Give a lower bound on the size of the game tree for the worst case, where $\mathcal{W} = \{\,\}$.

**c**. Propose a plausible evaluation function that can be used for any instance of generalized tic-tac-toe. The function may depend on $\mathcal{S}$ and $\mathcal{W}$.

**d**. Assume that it is possible to generate a new board and check whether it is a winning position in $100N$ machine instructions and assume a 2 gigahertz processor. Ignore memory limitations. Using your estimate in (a), roughly how large a game tree can be completely solved by alpha–beta in a second of CPU time? a minute? an hour?

**a**. An upper bound on the number of terminal nodes is $N!$, one for each ordering of squares, so an upper bound on the total number of nodes is $\sum_{i=1}^{N} i!$. This is not much bigger than $N!$ itself as the factorial function grows superexponentially. This is an overestimate because some games will end early when a winning position is filled.

This count doesn't take into account transpositions. An upper bound on the number of distinct game states is $3^N$, as each square is either empty or filled by one of the two players. Note that we can determine who is to play just from looking at the board.

**b**. In this case no games terminate early, and there are $N!$ different games ending in a draw. So ignoring repeated states, we have exactly $\sum_{i=1}^{N} i!$ nodes.

At the end of the game the squares are divided between the two players: $\lceil N/2 \rceil$ to the first player and $\lfloor N/2 \rfloor$ to the second. Thus, a good lower bound on the number of distinct states is $\binom{N}{\lceil N/2 \rceil}$, the number of distinct terminal states.

**c**. For a state $s$, let $X(s)$ be the number of winning positions containing no $O$'s and $O(s)$ the number of winning positions containing no $X$'s. One evaluation function is then $Eval(s) = X(s) - O(S)$. Notice that empty winning positions cancel out in the evaluation function.

Alternatively, we might weight potential winning positions by how close they are to completion.

**d**. Using the upper bound of $N!$ from (a), and observing that it takes $100NN!$ instructions.

At 2GHz we have 2 billion instructions per second (roughly speaking), so solve for the largest $N$ using at most this many instructions. For one second we get $N = 9$, for one minute $N = 11$, and for one hour $N = 12$.

**Exercise 6.**MMBD

In a full-depth minimax search of a tree with depth $D$ and branching factor $B$, with $\alpha - \beta$ pruning, what is the minimum number of leaves that must be explored to compute the best move?

There are $B^D$ leaf nodes. In $\alpha - \beta$ pruning we have to look at all the first player's moves but we only have to look at one of the second players moves. Thus we skip every other level and only have to look at $B^{D/2}$.

**Exercise 6.**MMTD

In a minimax tree with a branching factor of 3 and depth 2 (one max layer, one min layer with 3 nodes, and a leaf layer with 9 total nodes) what is the maximum number of nodes that can be pruned by alpha-beta pruning?

4 (2 each in the rightmost 2 leaf layers).

**Exercise 6.**ABTF

Which of the following statements about alpha-beta pruning are true or false?

**a**. Alpha-beta pruning may find an approximately optimal strategy, rather than the minimax optimal strategy.

**b**. Alpha-beta prunes the same number of subtrees regardless of the order of child nodes.

**c**. Alpha-beta generally requires more run-time than minimax on the same game tree.

None of these are true. Alpha-beta will always find the optimal strategy against an opponent that plays optimally. If an ordering heuristic is available, we can expand nodes in an order that maximizes pruning. Alpha-beta will require less run-time than minimax except in contrived cases.

**Exercise 6.**GGAM

Develop a general game-playing program, capable of playing a variety of games.

**a**. Implement move generators and evaluation functions for one or more of the following games: Kalah, Connect Four, Othello (Reversi), checkers, and chess.

**b**. Construct a general alpha–beta game-playing agent.

**c**. Compare the effect of increasing search depth, improving move ordering, and improving the evaluation function. How close does your effective branching factor come to the ideal case of perfect move ordering?

**d**. Implement a selective search algorithm, such as B* (Berliner, 1979), conspiracy number search (McAllester, 1988), or MGSS* (Russell and Wrefald, 1989) and compare its performance to A*.

See the online code repository for definitions of several games and game-playing agents. Notice that the game-playing environment is essentially a generic environment with the update function defined by the rules of the game. Turn-taking is achieved by having agents do nothing until it is their turn to move. At the time of writing, `blog.gamesolver.org` has a nice writeup of solving Connect Four.

Few students will be familiar with kalah, so it is a fair assignment, but the game is boring—depth 6 lookahead and a purely material-based evaluation function are enough to beat most humans. Othello is interesting and about the right level of difficulty for most students. Chess and checkers are sometimes unfair because usually a small subset of the class will be experts while the rest are beginners.

**Exercise 6.**TPNZ

Describe how the minimax and alpha–beta algorithms change for two-player, non-zero-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 alpha–beta? What if the player's utility functions on any state sum to a number between constants $-k$ and $k$, making the game almost zero-sum?

The minimax algorithm for non-zero-sum games works exactly as for multiplayer games; 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. Alpha-beta pruning is not possible in general non-zero-sum games, because an unexamined leaf node might be optimal for both players.

**Exercise 6.**SUNF

The Sunfish project, `github.com/thomasahle/sunfish` describes a simple chess engine, a little over 100 lines of Python (plus some data files), that plays at an Expert level. Load the project, experiment with it, see what you can learn, and what you can improve. Suggestions:

**a**. Improve the speed with a bitboard representation, parallel search, or a port to C or Rust.

**b**. Explain how the MTD search improves on alpha-beta without set bounds.

**c**. Add some endgame information.

**d**. Add extensions to search when the board is in a nonquiescent state.

**e**. Run experiments where you compare two versions of Sunfish, with and without a change, and see which one wins more in a series of games.

Each student will have their own way of approaching this exercise.

**Exercise 6.**ABPR

Develop a formal proof of correctness for alpha–beta pruning. To do this, consider the situation shown in Figure **??**. The question is whether to prune node $n_j$, which is a max-node and a descendant of node $n_1$. The basic idea is to prune it if and only if the minimax value of $n_1$ can be shown to be independent of the value of $n_j$.

**a**. Mode $n_1$ takes on the minimum value among its children: $n_1 = \min(n_2, n_{21}, \ldots, n_{2b_2})$. Find a similar expression for $n_2$ and hence an expression for $n_1$ in terms of $n_j$.

**b**. Let $l_i$ be the minimum (or maximum) value of the nodes to the *left* of node $n_i$ at depth $i$, whose minimax value is already known. Similarly, let $r_i$ be the minimum (or maximum) value of the unexplored nodes to the right of $n_i$ at depth $i$. Rewrite your expression for $n_1$ in terms of the $l_i$ and $r_i$ values.

**c**. Now reformulate the expression to show that in order to affect $n_1$, $n_j$ must not exceed a certain bound derived from the $l_i$ values.

**d**. Repeat the process for the case where $n_j$ is a min-node.

This question is not as hard as it looks. The derivation below leads directly to a definition of $\alpha$ and $\beta$ 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 $n_{i1} \ldots n_{i_{b_i}}$ 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 $\max(l_3, l_5, \ldots, l_k)$.

**Exercise 6.**ABPT

Prove that alpha–beta pruning takes time $O(2^{m/2})$ with optimal move ordering, where $m$ is the maximum depth of the game tree.

The result is given in Section 6 of Knuth and Moore (1975). The exact statement (Corollary 1 of Theorem 1) is that the algorithms examines $b^{\lfloor m/2 \rfloor} + b^{\lceil m/2 \rceil} - 1$ nodes at level $m$. These are exactly the nodes reached when Min plays only optimal moves and/or Max plays only optimal moves. The proof is by induction on $m$.

**Exercise 6.**CHET

Suppose you have a chess program that can evaluate 5 million nodes per second. Decide on a compact representation of a game state for storage in a transposition table. About how many entries can you fit in a 32-gigabyte in-memory table? Will that be enough for the three minutes of search allocated for one move? Can you estimate how many table lookups can you do in the time it would take to do one evaluation?

With 32 pieces, each needing 6 bits to specify its position on one of 64 squares, we need 24 bytes (6 32-bit words) to store a position, and let's say 4 more bytes to store a pointer to it. So we can store a little over a billion positions in the table. This is enough for the 900 million positions generated during a three-minute search.

Generating the hash key directly from an array-based representation of the position might be quite expensive. Modern programs (e.g., Heinz, 2000) carry along the hash key and modify it as each new position is generated. Suppose this takes on the order of 20 operations and an evaluation takes 2000 operations. Then we can do roughly 100 lookups per evaluation.

**Exercise 6.**PRUP

This question considers pruning in games with chance nodes. Figure **??** shows the complete game tree for a trivial game. Assume that the leaf nodes are to be evaluated in left-to-right order, and that before a leaf node is evaluated, we know nothing about its value—the range of possible values is $-\infty$ to $\infty$.

**a**. Copy the figure, mark the value of all the internal nodes, and indicate the best move at the root with an arrow.

**b**. Given the values of the first six leaves, do we need to evaluate the seventh and eighth leaves? Given the values of the first seven leaves, do we need to evaluate the eighth leaf? Explain your answers.

**c**. Suppose the leaf node values are known to lie between –2 and 2 inclusive. After the first two leaves are evaluated, what is the value range for the left-hand chance node?

**d**. Circle all the leaves that need not be evaluated under the assumption in (c).

**a**. See Figure S6.5.

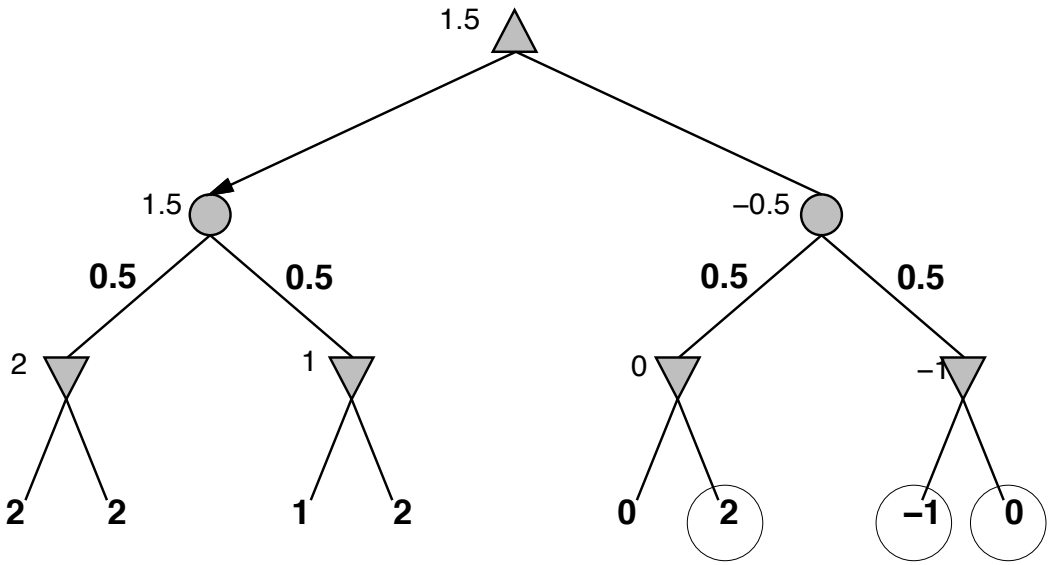


**Figure S6.5** Pruning with chance nodes solution.

**b**. Given nodes 1–6, we would need to look at 7 and 8: if they were both $+\infty$ then the values of the min node and chance node above would also be $+\infty$ and the best move would change. Given nodes 1–7, we do not need to look at 8. Even if it is $+\infty$, the min node cannot be worth more than $-1$, so the chance node above cannot be worth more than $-0.5$, so the best move won't change.

**c**. The worst case is if either of the third and fourth leaves is $-2$, in which case the chance node above is 0. The best case is where they are both 2, then the chance node has value 2. So it must lie between 0 and 2.

**d**. See figure.

## 6.4 Monte Carlo Tree Search

**Exercise 6.**MCTG

Take one of the games (tic-tac-toe, Connect Four, Kalah, Othello, checkers, etc.) for which you implemented a move generator, and apply Monte Carlo tree search to the move generator. Compare the performance of an MCTS player to an alpha-beta player.

Typically it is easier to apply MCTS; you don't need to be clever about an evaluation function. Performance with MTCS will be good on games that end in about a dozen moves, and will not be as good on games that take a hundred moves (unless students have access to a large computing cluster).

**Exercise 6.**KRIG

Read the paper "Monte Carlo Tree Search Techniques in the Game of Kriegspiel" by Paolo Ciancarini and Gian Piero Favini, and report on how well Monte Carlo tree search works in a game of imperfect information. Can you replicate the experiment, either for Kriegspiel or for another partially observable game? Can you improve on the techniques used in the paper?

Each student will have their own approach to this exercise.

**Exercise 6.**SGSM

Read *Monte Carlo Tree Search: A Review of Recent Modifications a nd Applications*, by Maciej Świechowski, Konrad Godlewski, Bartosz Sawicki, and Jacek Mańdziuk, arXiv:2103.04931. Report on one or more of the following ideas for MCTS. Is the idea more help for minimax search or for MCTS? For what kinds of applications is the idea good for?

**a**. Transposition tables
**b**. History heuristic
**c**. Opening books
**d**. Action reduction
**e**. Early termination

Each student will have their own approach to this exercise.

## 6.5 Stochastic Games

**Exercise 6.**EMMX

Implement the expectiminimax algorithm and the *-alpha–beta algorithm, which is described by Ballard (1983), for pruning game trees with chance nodes. Try them on a game such as backgammon and measure the pruning effectiveness of *-alpha–beta.

**Exercise 6.**EMXC

True or False: In expectiminimax search with two players, one max and the other chance, one can use pruning to reduce the search cost.

Pruning is about exploiting your knowledge of how both players will choose actions to look at less nodes. Since in expectiminimax one player will be picking randomly we have to look at every one of their actions, which will affect the expected value and the value of the node above it.

**Exercise 6.**EMXP

In each of the cases below, state whether a node can be pruned always, sometimes, or never. Assume that in the expectiminimax game that outcome values are bounded between $+1$ and $-1$.

**a**. In a **minimax** game, a leaf node that is the **first** child of its parent.

**b**. In a **expectiminimax** game, a leaf node that is the **first** child of its parent.

**c**. In a **minimax** game, a leaf node that is the **last** child of its parent.

**d**. In a **expectiminimax** game, a leaf node that is the **last** child of its parent.

The first child can never be pruned. Subsequent children can be pruned either in minimax or expectiminimax games. Note that if the game outcomes were not bounded–say in a game of backgammon where the value of the game could be doubled any number of times–then pruning can never be done in expectiminimax games.

**Exercise 6.**GAML

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 $x_1 \ldots x_n$, and that the transformation is $ax+b$, where $a$ is positive. We have

$$\begin{aligned} \min(ax_1 + b, ax_2 + b, \ldots, ax_n + b) &= a \min(x_1, x_2, \ldots, x_n) + b \\ \max(ax_1 + b, ax_2 + b, \ldots, ax_n + b) &= a \min(x_1, x_2, \ldots, x_n) + b \\ p_1(ax_1 + b) + p_2(ax_2 + b) + \cdots + p_n(ax_n + b) &= a(p_1x_1 + p_2x_2 + \cdots p_nx_n) + b \end{aligned}$$

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 $x > y \Rightarrow ax + b > ay + b$ if $a > 0$, the best choice at the root will be the same as the best choice in the original tree.

**Exercise 6.**MCGM

Consider the following procedure for choosing moves in games with chance nodes:

- Generate some dice-roll sequences (say, 50) down to a suitable depth (say, 8).
- With known dice rolls, the game tree becomes deterministic. For each dice-roll sequence, solve the resulting deterministic game tree using alpha–beta.
- Use the results to estimate the value of each move and to choose the best.

Will this procedure work well? Why (or why not)?

This procedure will give incorrect results. Mathematically, the procedure amounts to assuming that averaging commutes with min and max, which it does not. Intuitively, the choices made by each player in the deterministic trees are based on full knowledge of future dice rolls, and bear no necessary relationship to the moves made without such knowledge. (Notice the connection to the discussion of card games in Section 6.6.2 and to the general problem of fully and partially observable Markov decision problems in Chapter 16.) In practice, the method works reasonably well, and it might be a good exercise to have students compare it to the alternative of using expectiminimax with sampling (rather than summing over) dice rolls.

**Exercise 6.**SGEP

Consider a game in which three players, A, B, and C, are trying to solve an 8-puzzle. A player receives +1 for making the final move that solves the puzzle, –1 if another player does so. If the same state is repeated 3 times, the game ends with everyone receiving 0. But the players do not just alternate turns; instead before each turn a fair die roll determines whether A, B, or C will move. The die has been cast, it is A's turn to move, and A sees that the puzzle is two steps from being solved. To keep things simple, remember that in an 8-puzzle every move takes you either one step closer or one step further away from the goal (there is no possibility to stay the same distance from the goal), so each player has essentially two choices. Should A move towards the goal or away from it? Justify your answer *quantitatively*.

If A moves towards the goal, whoever goes next will win; A has a 1/3 chance of going next, so A's expected payoff is $(1/3)(+1) + (2/3)(-1) = -1/3$. If A moves away, the game is now three steps from being solved. If it ever gets to being two steps away, whoever has the next move will do the same analysis as A and also move away, so no one will ever win. Because the state space is finite, some position will repeat three times. This confirms that the expected payoff of moving away is 0, which is better than $-1/3$, so moving away is the best move.

**Exercise 6.**MAXT

In the following, a "max" tree consists only of max nodes, whereas an "expectiminimax" tree consists of a max node at the root with alternating layers of chance and max nodes. At chance nodes, all outcome probabilities are nonzero. The goal is to *find the value of the root* with a bounded-depth search. For each of (a)–(f), either give an example or explain why this is impossible.

**a**. Assuming that leaf values are finite but unbounded, is pruning (as in alpha–beta) ever possible in a max tree?

**b**. Is pruning ever possible in an expectiminimax tree under the same conditions?

**c**. If leaf values are all nonnegative, is pruning ever possible in a max tree? Give an example, or explain why not.

**d**. If leaf values are all nonnegative, is pruning ever possible in an expectiminimax tree? Give an example, or explain why not.

**e**. If leaf values are all in the range $[0, 1]$, is pruning ever possible in a max tree? Give an example, or explain why not.

**f**. If leaf values are all in the range $[0, 1]$, is pruning ever possible in an expectiminimax tree?

**g**. Consider the outcomes of a chance node in an expectiminimax tree. Which of the following evaluation orders is most likely to yield pruning opportunities?

(i) Lowest probability first
(ii) Highest probability first
(iii) Doesn't make any difference

**a**. No pruning. In a max tree, the value of the root is the value of the best leaf. Any unseen leaf might be the best, so we have to see them all.

**b**. No pruning. An unseen leaf might have a value arbitrarily higher or lower than any other leaf, which (assuming non-zero outcome probabilities) means that there is no bound on the value of any incompletely expanded chance or max node.

**c**. No pruning. Same argument as in (a).

**d**. No pruning. Nonnegative values allow *lower* bounds on the values of chance nodes, but a lower bound does not allow any pruning.

**e**. Yes. If the first successor has value 1, the root has value 1 and all remaining successors can be pruned.

**f**. Yes. Suppose the first action at the root has value 0.6, and the first outcome of the second action has probability 0.5 and value 0; then all other outcomes of the second action can be pruned.

**g**. (ii) Highest probability first. This gives the strongest bound on the value of the node, all other things being equal.

**Exercise 6.**MMXF

Players MAX and MIN are playing a game with a finite depth of possible moves. MAX calculates the minimax value of the root to be $M$. Assume that each player has at least 2 possible actions at every turn and that every distinct sequence of moves leads to a distinct score. Which of the following are true?

**a**. Assume MIN is playing suboptimally, and MAX **does not** know this. The outcome of the game can be better than $M$ (i.e. higher for MAX).

**b**. Assume MAX **knows** player MIN is playing randomly. There exists a policy for MAX such that MAX can guarantee a better outcome than $M$.

**c**. Assume MAX **knows** MIN is playing suboptimally on every move and **knows** the policy $\pi_{\text{MIN}}$ that MIN is using (MAX knows exactly how MIN will play). There exists a policy for MAX such that MAX can guarantee a better outcome than $M$.

**d**. Assume MAX **knows** MIN is playing suboptimally at all times but **does not know** the policy $\pi_{\text{MIN}}$ that MIN is using (MAX knows MIN will choose a suboptimal action at each turn, but does not know which suboptimal action). There exists a policy for MAX such that MAX can guarantee a better outcome than $M$.

**a**. True; MAX may take advantage of MIN's errors without having to know about it ahead of time.

**b**. False; MIN is playing randomly, but MIN might randomly make the right move each time.

**c**. True; Since MIN is playing a different move from the optimal one each time, the game will end up in a different sequence of moves from the optimal sequence, and we assumed that every sequence of moves arrives at a different outcome, so if MAX maximizes each move then the outcome will be $> M$.

**d**. True (same reason as previous answer).

## 6.6 Partially Observable Games

**Exercise 6.**TFGM

Which of the following are true and which are false? Give brief explanations.

**a**. In a fully observable, turn-taking, zero-sum game between two perfectly rational players, it does not help the first player to know what strategy the second player is using—that is, what move the second player will make, given the first player's move.

**b**. In a partially observable, turn-taking, zero-sum game between two perfectly rational players, it does not help the first player to know what move the second player will make, given the first player's move.

**c**. A perfectly rational backgammon agent never loses.

**a**. *In a fully observable, turn-taking, zero-sum game between two perfectly rational players, it does not help the first player to know what strategy the second player is using—that is, what move the second player will make, given the first player's move.*
True. The second player will play optimally, and so is perfectly predictable up to ties. Knowing which of two equally good moves the opponent will make does not change the value of the game to the first player.

**b**. *In a partially observable, turn-taking, zero-sum game between two perfectly rational players, it does not help the first player to know what move the second player will make, given the first player's move.*
False. In a partially observable game, knowing the second player's move tells the first player additional information about the game state that would otherwise be available only to the second player. For example, in Kriegspiel, knowing the opponent's future

move tells the first player where one of the opponent's pieces is; in a card game, it tells the first player one of the opponent's cards.

**c**. *A perfectly rational backgammon agent never loses.*
False. Backgammon is a game of chance, and the opponent may consistently roll much better dice. The correct statement is that the *expected* winnings are optimal. It is suspected, but not known, that when playing first the expected winnings are positive even against an optimal opponent.

**Exercise 6.**CHPI

Consider carefully the interplay of chance events and partial information in each of the games in Exercise 6.GAME.

a. For which is the standard expectiminimax model appropriate? Implement the algorithm and run it in your game-playing agent, with appropriate modifications to the game-playing environment.
b. For which would the scheme described in Exercise 6.MCGM be appropriate?
c. Discuss how you might deal with the fact that in some of the games, the players do not have the same knowledge of the current state.

One can think of chance events during a game, such as dice rolls, in the same way as hidden but preordained information (such as the order of the cards in a deck). The key distinctions are whether the players can influence what information is revealed and whether there is any asymmetry in the information available to each player.

**a**. Expectiminimax is appropriate only for backgammon and Monopoly. In bridge and Scrabble, each player knows the cards/tiles he or she possesses but not the opponents'. In Scrabble, the benefits of a fully rational, randomized strategy that includes reasoning about the opponents' state of knowledge is small (but can make the difference in championship play), but in bridge the questions of knowledge and information disclosure are central to good play.

**b**. None, for the reasons described earlier.

**c**. Key issues include reasoning about the opponent's beliefs, the effect of various actions on those beliefs, and methods for representing them. Since belief states for rational agents are probability distributions over all possible states (including the belief states of others), this is nontrivial.

## 6.7 Limitations of Game Search Algorithms

**Exercise 6.**LIMG

**a**. Implement a version of search (either alpha-beta or Monte Carlo) that cuts off the search early if it finds a move that is "obviously" best (using Sunfish or some other game

engine, or code of your own). Compare the performance of this version to the standard version.

**b**. Do the same for a version that, when two moves are completely symmetric, considers only one of the two.

[

Students should expect to see measurable but small improvements with either change.

**Exercise 6.**MENA

Re-implement one of the first machine learning game playing system, a version of MENACE, the Machine Educable Noughts and Crosses (Tic-Tac-Toe) Engine (Michie, 1963). The original version was implemented with 304 matchboxes filled with colored beads. You might find it easier to use a computer.

**a**. Before it has learned anything, MENACE will play randomly: each possible move in each position has $n$ chances of being selected. On the very first move there are 9 possible moves, so each has a $n/(9n)$ chance of being selected. MENACE makes its moves against a player (it could be an optimal minimax opponent) who makes their move until the game ends.

**b**. If MENACE has won, each of the moves it makes is rewarded by having 3 more chances added to its odds (i.e. an increase from $n$ to $n+3$). If the game is a draw, 1 more chance is added, and if a loss, 1 chance is subtracted.

**c**. Record MENACE's history of wins and losses over many games. How long does it take to reach equilibrium where it avoids losing?

**d**. Experiment with different choices of values of $n$, and of the 3/1/-1 rewards. Which choices lead to faster winning performance?

**e**. When your program has reached equilibrium, compare its policy to the optimal policy from a minimax algorithm. Report on the results.

It should take about 200 moves to reach equilibrium. The rate of training will vary somewhat depending on the opponent. Does the opponent ever make a mistake? When there are multiple moves that all lead to a draw with best play, does the opponent pick one at random, or always pick the same one?