Notes for CPSC 474 – Computational Intelligence for Games

James Glenn

Draft; Updated on Dec 7, 2022

Contents

List of Figures

List of Theorems

LIST OF THEOREMS

List of Definitions

LIST OF DEFINITIONS

List of Abbreviations

List of Abbreviations

List of Symbols

Chapter 1

Game Trees

1.1 Puzzles

1.1.1 Backtracking

1.1.2 A^*

1.2 Two-Player Games

What it means to solve a game.

A game is said to be *weakly solved* if the game-theoretic value of the initial position is known and there is an algorithm that achieves that value for the winning player (or either player if the initial position is a draw) starting from the initial position, no matter what move the other player makes during the course of the game. A game is said to be *strongly solved* if the value of *all* positions are known and there is an algorithm that, starting at any position, will achieve that value for the winning player at that position (or either if a draw).

1.2.1 Exhaustive Search

For a zero-sum game, we have $v_I(G) = -v_{II}(G)$, and so maximizing $v_{II}(G)$ is the same as maximizing $-v_I(G)$, which is the same as minimizing $v_I(G)$.

function MINIMAX(G)
  if G is terminal **then**
    return $v_I(G)$ as determined by the rules of the game
  else
    Enumerate the options of G, $G_0, \ldots, G_{k-1}$
    $(v_0, m_0) \ldots, (v_{k-1}, m_{k-1}) \leftarrow Minimax(G_0), \ldots, Minimax(G_{k-1})$
    if $next(G) = I$ **then** ▷ Player I is next to move
      $m \leftarrow \arg\max_{0 \le j < k} v_j$
    else
      $m \leftarrow \arg\min_{0 \le j < k} v_j$
    end if
    return (v_m, m)
  end if
end function

(we assume that the function that enumerates the possible moves is deterministic, so returning the index m of the best move is sufficient to report to the caller what the best move is).

We can also write the base case to use the value from the perspective of the player to make the next move (or, rather, the player who would have made the next move if the game hadn't just ended), and $v_I(G) = -v_{II}(G)$ again to collapse the cases for which player makes the next move into one.

function NEGAMAX(G)
  if G is terminal **then**
    $A \leftarrow next(G)$ ▷ player to make next move
    return $v_A(G)$ as determined by the rules of the game
  end if
  Enumerate the options of G, $G_0, \ldots, G_{k-1}$
  $(v_0, m_0) \ldots, (v_{k-1}, m_{k-1}) \leftarrow Negamax(G_0), \ldots, Negamax(G_{k-1})$
  $m \leftarrow \arg\max_{0 \leq j < k} -v_j$
  return $(-v_m, m)$
end function

We see later how to handle games that may not be finite because of cycles in the positions, and games for which the state space is too large to feasibly explore exhaustively.

1.2.2 Impartial Games

An *impartial* game is one in which the set of moves available to the next player to move does not depend on whether that player was the first to move in the game or the second. 1-2-3 Nim are impartial – the moves available are to take 1, 2, or 3 stones (subject to the number of stones remaining), regardless of whether the next player to move is Player I or Player II. In an impartial game, there is no notion of ownership of playing pieces or locations. Games like Chess, Checkers, and Go are not impartial. In chess, for example, you cannot determine the legal moves just by looking at the current locations of all the pieces – since players can only move pieces of their own color, you need to know which player makes the next move as well, so that you can determine if the legal moves move the white pieces or the black pieces.

For an impartial game with no draws, because we do not need to keep track of whose turn it is in order to determine the legal moves for an impartial game, we can simplify the MINIMAX algorithm for impartial games. The value of a position can no longer be -1 or 1, since we would need to know whose turn it at the end of the game in order to assign such a value. Instead, we can classify positions by whether or not the next player to move has a winning strategy. We define this recursively: at the terminal positions, the player who would make a move if the game hadn't ended has either won or lost. In the normal form of 1-2-3 Nim, for example, the terminal position is the position with 0 stones, and the player whose turn it is has lost because the other player just made the last move and won. Otherwise, the player to make the next move has a winning move if there is move to a position where the player to make the next move from that new position (the *other* player) doesn't have a winning move. We call those two classes of positions N-positions (winning positions for the next player to move) and P-positions (winning positions for the previous player to move) and can define them inductively. In a finite, impartial game with no draws, each position falls into one of those two *outcome classes*.

Definition 1.1. An N-position is either 1) a terminal position in which the last player to move has lost; or 2) a non-terminal position for which at least one option is a P-position.

Definition 1.2. A P-position is either 1) a terminal position in which the last player to move has won; or 2) a non-terminal position for which all options are N-positions.

This implies a recursive procedure for determining whether a position in a finite, impartial game with no draws is an N- or P-position.

function OUTCOME(G)

```
    if G is terminal then
        return P or N as determined by the rules of the game        ▷ P for normal; N for misere
    end if
    Enumerate the options of G, G_0, ..., G_{k-1}
    j ← 0                                                           ▷ j iterates through list of options
    while j < k and Outcome(G_j) ≠ P do
        j ← j + 1                                                   ▷ search for reachable P-position
    end while
    if j < i then
        return N
    else
        return P
    end if
end function
```

This is just the maximizing case of MINIMAX. Treating N as 1 and P as -1, determining the maximum of the negatives of the values of the options is equivalent to determining if there is a 1 among those negatives: the values of the options are all either -1 or 1, and if any single one of them is 1, then the maximum is 1; otherwise it is -1.

1.2.3 Dynamic Programming

The recursive OUTCOME algorithm suffers from the problem of overlapping subproblems – different parent recursive calls may make child recursive calls with the same parameters; those child calls will return the same result since they are made with the same parameters. But the second parent call to run does not know that the first parent has already made a recursive call with the same parameters as one it needs to make, because there is nothing in the program to remember which calls have already been made and what the results were. And so the second parent call redoes the work of the child call. Those child calls may also have overlapping problems, so the amount of unnecessary repeated work can be exponential. For example, for 1-2-3 Nim starting with 6 stones, the initial call is to $Outcome(6)$. The call to $Outcome(6)$ calls $Outcome(5)$, $Outcome(4)$, and $Outcome(3)$. All three of those calls will themselves call $Outcome(2)$. There are also multiple calls to $Outcome(4)$, $Outcome(3)$, $Outcome(1)$, and $Outcome(0)$, and the total number of function calls, and hence the total running time, is exponential in n, where n is the number of stones at the beginning of the game.

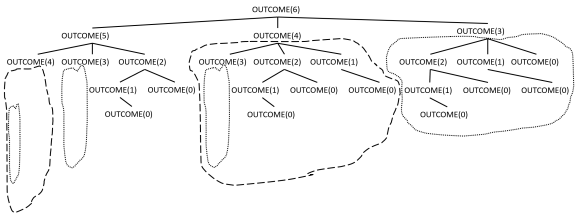

Figure 1.1: Recursion tree for $OUTCOME(6)$. The outlined subtree for $OUTCOME(3)$ appears four times total; only the leftmost call is completely shown. The outlined subtree for $OUTCOME(3)$ appears twice.

Overlapping subproblems result in a difference between the *game tree complexity* and the *state space complexity* of a game. The game tree complexity is a measure of the number of different histories (sequences of moves) that are possible in a game. The game tree complexity can be approximated by estimating the *branching factor* – the average number of moves available at a position, so the average number of children in

the game tree – and the average depth of the terminal nodes game tree – the average number of moves from the initial position to the terminal positions, assuming that each history is equally likely. If the branching factor is b and the depth is d, then the game tree complexity is approximately b^d. On the other hand, the state space complexity is the number of distinct positions (states) in the game. Two histories can lead to the same state, for example in 1-2-3 Nim starting with 12 stones, Player I taking 1 stone followed by Player II taking 3 leads to the same state (8 stones left, Player I's turn) as both players taking 2 stones, and also the same state as Player I taking 3 and Player II taking 1. In the game tree, those three sequences lead to different nodes, but those three nodes represent the same state. 1-2-3 Nim starting with n stones has a linear number of states ($2n + 1$, to be exact), but an exponential number of nodes in the game tree.

The solution to the problem of overlapping subproblems is to remember the results of previous calls so that the calculations required to obtain those results need not be repeated; the result can simply be reused. This technique is called *dynamic programming*. Dynamic programming employs a table (in the language of data structures, a map, dictionary, or associative array) that keeps track of the results of the recursive calls. The keys are the arguments to the recursive calls and the associated values are the results of the calls. We then fill out the table in order starting with the terminal positions and working in order of increasing length. In that way, when we are working on a position with at most k moves until the end of the game, all the positions reachable in one move are at most $k - 1$ moves until the end of the game, and have already been computed and stored in the table.

Since the argument to OUTCOME is the number of stones, and the number of stones is the the range $\{0, 1, \ldots, n\}$, where n is the initial number of stones, the table can be a plain array (and in general, if the arguments are all integers in the range $\{0, 1, \ldots, n_i\}$ then we can use a multi-dimensional array with one dimension per argument). Working from the end of the game (0 stones) to the beginning (n stones) is working through in order of increasing numbers of stones. We give the pseudocode for our dynamic programming 1-2-3 Nim solver as 1-2-3-NIM-DP.

```
function 1-2-3-NIM-DP(n)
    outcome[0] ← N                          ▷ terminal positions are base cases
    move[0] ← NIL
    i ← 1                                   ▷ i is current number of stones
    while do i ≤ n
        table[i] ← P                        ▷ default to P-position
        move[i] ← NIL
        j ← max(0, i − 3)                   ▷ j iterates over legal moves
        while do j < i and outcome[j] ≠ P   ▷ search for reachable P-position
            j ← j + 1
        end while
        if then j < i
            outcome[i] ← N                  ▷ move from i to j is to a P-position, so i is an N-position
            move[i] ← j
        end if
    end while
end function
```

We now have a linear-time solver for 1-2-3 Nim. In general, the dynamic programming solutions will make $s \cdot b$, table lookups where s is the number of positions in the game, and b is the number of possible moves. The total running time will depend on the amount of time it takes to generate all the possible moves from a given position, and the resulting positions (for 1-2-3 Nim, that is just a single subtraction for each of up to three possible moves, so $O(1)$ time).

The approach presented in 1-2-3-NIM-DP is referred to as the *bottom-up* dynamic programming approach,

because it considers and solves subproblems in order from lower in the game tree (closer to terminal positions) to higher (closer to the initial position). This is easiest to implement when it is easy to determine the terminal positions, the positions at most one move away from the terminal positions, then two moves away, etc. That is the case for 1-2-3 Nim, because the number of stones remaining determines the number of moves until the terminal position.

For a game like Tic-Tac-Toe, one could consider positions in order of number of open spaces left on the board. This is not the same as ordering positions by moves until terminal positions – for example, there are terminal positions with 5 total X's and O's (that result from poor game play by one of the players) and nonterminal positions with 6, 7, and 8 total marks. However, considering the positions in order of increasing free spaces still works, since the bottom-up method only requires that when a position is considered, all positions that are reachable from it have already been considered. Any ordering of positions that satisfies that property will work. For Tic-Tac-Toe, each move decreases the number of open spaces, so if we consider positions with 0 open spaces, 1 open space, etc, it will always be the case that when the loop reaches a position G, all the options of G have fewer open spaces and so have already been considered (in graph theory terms, we need a *topological sort* of the positions).

The bottom-up dynamic programming approach may end up considering unreachable positions, but these may be easy to filter out, and it does not affect the correctness of the other results if we "solve" them. But the bottom-up approach may be inefficient (although probably still more efficient than the straightforward recursive implementation) if there are too many unreachable positions that are difficult to filter out. It may also be difficult to implement an ordering of the positions that results in a topological sort. In these cases, we can implement a top-down *memoized* version of dynamic programming. The memoized solution also keeps track of a table of already-solved positions and their values. But instead of solving them in a bottom-up way, they are solved in the same order as the top-down recursive solutions. Every time a position is solved, the result is added to the table. The recursive function adds a base case that checks whether the current problem has already been solved, and if it has, then it simply returns the value stored in the table.

function OUTCOME-MEMOIZED($G, memo$) ▷ the initial call is made with empty memo
  if G is terminal **then**
    return P or N as determined by the rules of the game ▷ P for normal; N for misere
  else if $memo$ contains G **then**
    return $memo[G]$ ▷ G was solved previously
  end if
  Enumerate the options of G, $G_0, \ldots, G_{k-1}$
  $j \leftarrow 0$ ▷ j iterates through list of options
  while $j < k$ and $OUTCOME - MEMOIZED(G_j, memo) \neq P$ **do**
    $j \leftarrow j + 1$ ▷ search for reachable P-position
  end while
  if $j < i$ **then**
    $result \leftarrow N$
  else
    $result \leftarrow P$
  end if
  $memo[G] \leftarrow result$ ▷ save result in memo
  return $result$
end function

In general, the bottom-up approach is faster than the top-down approach because it avoids the additional overhead of the recursive function calls. It is also easier to reduce the size of the table/memo with the bottom-up approach by forgetting entries that are not needed any more (for example, with Tic-Tac-Toe,

when considering a position with 5 empty spaces, the table lookups will be on positions with 4 empty spaces, so the table can forget the positions with 3, 2, 1, and 0 empty spaces). However, the top-down approach may be preferable to avoid unreachable positions and when it is difficult to order the moves.

Chapter 2

Combinatorial Games

2.1 Nim

The the previous chapter, we used a version of Nim as an example of why dynamic programming is important for solving small, finite games. There are many different versions of Nim with initial setups (the number of rows, and the number of stones in each row), different legal moves (for example, allowing taking 1, 2, or 3 stones, or allowing to take any non-zero number of stones from a row), and different winning conditions (the last move loses rule in the previous example is the *misere rule; last move wins is the* normal rule).

We introduce here a more common version of Nim than the single-row, misere version from the previous chapter. In the common versions, play typically starts with multiple rows of stones, with the number of stones increasing by some constant from the first row to the last row, for example 5, 7, and 9 stones. On a player's turn, that player can take any non-zero number of stones from a single row. We consider the normal form of the game: the last player to make a move wins (or, equivalently, the first player who has no possible moves loses). The misere form is actually more common, but the winning strategy for the normal form has the astonishing property that it generalizes to any finite, impartial, normal game.

2.1.1 Finite, Normal, Impartial Combinatorial Games

The rest of this chapter is devoted to developing the mathematical concepts that say no, the professor didn't cheat, in the sense that the move that replacing a group of 7 Kayles pins with a row of 2 Nim stones will never change the outcome class of the game – there was a winning move for the next player before the switch if and only if there was a winning move for the next player after the switch.

2.1.2 Game Sums

The multi-row version of Nim can be viewed as a collection of separate games, one on each row. On a player's turn, that player chooses one and only one of the games to make a move in. Eventually, some games in the collection (the individual rows) have no moves left, and the players must choose to play from among the remaining games (rows). For a normal game, the player who makes the last move in any of the games is the winner; for a misere game, the player who makes the last move loses.

We can formalize this notion of a collection of games as a game sum.

Definition 2.1. For impartial games G and H, the game sum $G + H$ is the game where the options are $\{G' + H \mid G' \text{is an option of } G\} \cup \{G + H' \mid H' \text{is an option of } H\}$.

Moves from the first term in the union are moves in G; moves from the second term are moves in H. There are no moves in the sum that change both G and H at the same time. When there are no moves

available in G or H, that term in the union is empty, so the only moves are moves in the other game. If there are no moves available in either game, the union is empty, and so the sum $G + H$ has no options, so is terminal (the game is over).

Because games sums are defined in terms of set union, and set union is associative and commutative, game sums are too: for any impartial games G, H, and K,

$$\begin{aligned} G + H &= H + G \\ G + (H + K) &= (G + H) + K. \end{aligned}$$

Because the order of operations doesn't matter, we can drop the parentheses in sums of more than two games: $G + (H + K) = (G + H) + K = G + H + K$.

For example, consider a single-row version of Nim in which the players can take any non-zero number of stones on their turn. This by itself is not an interesting game, since the winning move in the normal version is to take all the stones, and the winning move in the misere version is to take all the stones except one. But when combined into sums, the game becomes interesting: the multi-row version of Nim is the sum of the games played on the individual rows.

Write the normal form of the single-row, take-any-stones version of Nim starting with n stones as $*n$ (this is called a *nimber and is the starting point for defining a new type of number that is inspired by the correspondence between the natural numbers and nimbers and the game sum operation; these so-called* surreal numbers are derived from extending these notions to more general forms of games). Then the three-row version of Nim starting with 5, 7, and 9 stones is $*5 + *7 + *9$. A simpler game with 3 and 1 stone is $*3 + *1$. In that game, the options are to take from the first row to leave $*2 + *1$, $*1 + *1$, or $*0 + *1$, or to take from the second to leave $*3 + *0$. Note that any sum containing the term $*0$ is the same as the sum without that term, since there are no options of $*0$ and so the term contributes nothing to the union that defines the game sum.

Theorem 2.1. *for any finite, impartial game G, $G + *0 = G$.*

Proof. Games are defined in terms of the options (moves) available: we can write G as $G = \{G' \mid G' \text{ is an option of } G\}$. By the definition of game sums,

$$\begin{aligned} G + *0 &= \{G' \mid G' \text{ is an option of } G\} \cup \{H' \mid H' \text{ is an option of } *0\} \\ &= \{G' \mid G' \text{ is an option of } G\} \\ &= G. \end{aligned}$$

□

2.1.3 Game Equivalence

We can now define formally when it is not cheating to replace one game with another.

Definition 2.2. For finite, impartial games G and H, say that G is equivalent to H ($G \approx H$) if, for any other finite, impartial game K, $G + K$ and $H + K$ have the same outcome class.

Because game equivalence is defined in terms of equality of outcome classes, game equivalence is an equivalence relation.

Theorem 2.2. *Game equivalence ($\approx$) satisfies the properties of an equivalence relation: 1) for any finite, impartial game G, $G \approx G$ (the reflexive property); 2) for any finite, impartial games G and H, if $G \approx H$ then $H \approx G$ (the symmetric property); and 3) for any finite, impartial games G, H, and L, if $G \approx H$ and $H \approx L$ then $G \approx L$ (the transitive property).*

Proof. Game equivalence is defined in terms of equality of equivalence classes, so the three properties follow immediately from the reflexive, symmetric, and transitive properties of equality.

(Reflexive): Let G be any finite, impartial game. Then, for any other finite, impartial game K, $G+K$ and $G+K$ have the same outcome class because they are the same game, so $G \approx G$.

(Symmetric): Let G and H be finite, impartial games. Suppose that $G \approx H$. Then, by definition of $\approx$, $G+K$ and $H+K$ have the same outcome class for any finite, impartial game K. That is the same thing as saying that $H+K$ and $G+K$ have the same outcome class for any K, so $H \approx G$ as well.

(Transitive): Let G, H, and L be finite, impartial games, and suppose that $G \approx H$ and $H \approx L$. Then, by definition of $\approx$, $G+K$ and $H+K$ have the same outcome class for any finite, impartial game K, and $H+K$ and $L+K$ have the same outcome class for any finite, impartial game K. Let K be any finite, impartial game. Now $G+K$ and $H+K$ have the same outcome class, and $H+K$ and $L+K$ have the same outcome class. Since $G+K$ and $L+K$ both have the same outcome class as $H+K$, they both have the same outcome class as each other. So $G+K$ and $L+K$ have the same outcome class for any finite, impartial game K, so satisfy the definition of $G \approx L$. □

We will eventually see that Kayles started with a single group of 7 pins is equivalent to Nim starting with 2 stones, so when our professor changed one group of 7 pins when playing Kayles and confronted with the losing position consisting of groups of 9, 7, 7, and 9, the outcome didn't change: let G be Kayles with 7 pins and let K be Kayles with groups of 9, 7, and 9 pins. The the position the professor tampered with was $G+K$ (remember that game sums are commutative). With $H = *2$ and $G \approx H$, we have, by definition of game equivalence, that $H+K$ has the same outcome class as $G+K$. So if the students had a winning move on $G+K$ ($G+K$ is an N-position), then the students still had a winning move on $H+K$, even without resorting to the same trick the professor used.

Note that game equivalence is a refinement of outcome classes. Outcome classes divide games into two classes: those where the next player to move has a winning strategy (the N-positions), and those where the player after the next player has a winning strategy no matter what the next player does (the P-positions). We will see that all P-positions are equivalent to each other, but that is not the case for all N-positions. However, it follows immediately from the fact that $\approx$ is an equivalence relation that whenever $G \equiv H$, G and H have the same outcome class.

Corollary 2.3.

Proof. Let G and H be any finite, impartial games such that $G \approx H$. Then, by definition of $\approx$, $G+K$ and $H+K$ have the same outcome class for any finite, impartial game K; in particular, consider $K = *0$. Then $G+K = G$ and $H+K = H$, and so $G+K$ and $H+K$ having the same outcome class is the same as G and H having the same outcome class. □

It is easy to show when two games G and H are not equivalent: it suffices to find a single K for which $G+K$ and $H+K$ are in different outcome classes. For example, let $G = *2$ and $H = *3$ (both N-positions, but not equivalent to each other). Then $*2 \not\approx *3$: consider $K = *2$. Then $G+K = *2+*2$, which is a P-position by the mimicry strategy (or by enumerating all of the possible moves: taking 1 from a row of 2 leaves $*2+*1$, which is an N-position because of the winning move to $*1+*1$, which itself is a P-position because the only move is to take one stone, after which your opponent takes the other and wins; taking 2 from a row of 2 in $*2+*2$ leaves $*2+*0$, which is an N-position because of the winning move to $*0+*0$; since both moves from $*2+*2$ are to N-positions, $*2+*2$ is a P-position). But $H+K = *3+*2$ is an N-position by the winning move to $*2+*2$, which we just showed is a P-position. Since there is a K such that $G+K$ and $H+K$ are in different outcome classes, $G \not\approx H$.

Now consider $G = *2+*1$ and $H = *3$. Let $K = *n$ for $n = 0, 1, 2, \ldots$.

$G+*0$ and $H+*0$ are both N-positions: from $*2+*1+*0$, take 1 from the row of 2 to leave $*1+*1+*0$, which is a P-position (we leave that reasoning to the reader); from $*3+*0$, take the entire row of three to win.

$G+*1$ and $H+*1$ are both N-positions: from $*2+*1+*1$, take 2 from the row of 2 to leave $*0+*1+*1$, which is a P-position; from $*3+*1$, take two from the row of 3 to leave $*1+*1$, which is also a P-position.

$G+*2$ and $H+*2$ are both N-positions: from $*2+*1+*2$, take 1 from the row of 1 to leave $*2+*0+*2$, which is a P-position; from $*3+*2$, take one from the row of 3 to leave $*2+*2$, which is also a P-position.

$G+*3$ and $H+*3$ are both P-positions: from $*2+*1+*3$, the legal moves are to the N-positions $*1+*1+*3$ (winning move to $*1+*1+*0$), $*0+*2+*3$ (winning move to $*0+*2+*2$), $*2+*0+*3$ ($*2+*0+*2$), $*2+*1+*2$ ($*2+*0+*2$), $*2+*1+*1$ ($*0+*1+*1$), and $*2+*1+*0$ ($*1+*1+*0$). $H+K=*3+*3$ can be shown to be a P-position using the mimicry strategy.

$G+*4$ and $H+*4$ are both N-positions: take 1 from the row of 4 to leave $G+*3$ or $H+*3$, which we just showed are P-positions. In fact, this works for any $n \geq 4$: reduce the row of n to a row of 3 to leave the opponent in the P-position $G+*3$ or $H+*3$.

So we have shown that $G+K$ and $H+K$ have the same outcome class for $K=*n$. However, to show that $*2+*1 \approx *3$, we would have to show that $*2+*1+K$ and $*3+K$ have the same outcome class not just for $K=*n$, but for *any* finite, impartial game K, no matter how crazy the rules are. It is not immediately clear that there are *any* examples of games G and H that are equivalent to each other except for the trivial case of $G=H$. However, it is true that $*2+*1 \approx *3$, and in fact, any finite, normal, impartial game is equivalent to some nimber, and the method for computing that nimber is straightforward (if time-consuming in some cases). The following Lemmas set the stage for that main result.

Lemma 2.1. *For any finite, normal, impartial games G and H, $G+H$ is an N-position if G and H are in different outcome classes (one is an N-position and the other is a P-position), and $G+H$ is a P-position if G and H are both P-positions*

Proof. The proof proceeds by induction on the length of $G+H$.

Base case ($G+H$ has length 0). Then $G=H=\{\}=G+H$, so all of G, H, and $G+H$ are P-positions.

Induction step: suppose $k>0$, that all games G' and H' so that the length of $G'+H'$ is less than k obey the $N+P=P$ and $P+P=P$ rules, and that $G+H$ is a game of length k. To show that G, H, and $G+H$ obey the rules, we consider each case of the rule in turn. In each case, we consider the options of G and/or H in order to relate the game $G+H$ to shorter games in order to apply the inductive hypothesis.

Case 1: G is an N-position and H is a P-position. The Lemma says that $G+H$ should be an N-position. By definition of an N-position, there is an option G' of G that is a P-position. $G'+H$ is an option of $G+H$ and is of length less than that of $G+H$, which is k. So the inductive hypothesis applies and, since $G'+H$ is the sum of two P-positions, tells us that $G'+H$ is also a P-position. So $G+H$ has an option that is a P-position ($G'+H$), so is itself an N-position.

Case 2: G is a P-position and H is an N-position. This case is similar to case 1.

Case 3: G and H are both P positions. The Lemma says that $G+H$ should be a P-position. By the definition of P-positions, all the options G' of G and H' of H are N-positions. By the definition of game sum, all the options of $G+H$ are of the form $G'+H$ for some option G' of G or $G+H'$ for some option H' of H. All those options of $G+H$ are of length shorter than $G+H$, and are the sum of an N-position (G') and a P-position (H) or a P-position (G) and an N-position (H'). The inductive hypothesis then says that all the options of $G+H$ are N-positions. And a position where all the options are N-positions is, by definition, a P-position, as required. or G' □

Note that Lemma 2.1 doesn't say anything about the outcome class of $G+H$ when G and H are both N-positions – in that case it is possible for $G+H$ to be an N-position (as is the case with $*1+*2$, and it is possible for $G+H$ to be a P-position (for example, $*1+*1$). A subsequent Lemma will distinguish between those two cases.

Lemma 2.2. *For any finite, normal, impartial games G and A such that A is a P-position, $G + A \approx G$*

Proof. We use the definition of $\approx$ to show that $G + A \approx G$: we show that for any H, $(G + A) + H$ and $G + H$ are in the same outcome class. There are two cases based on the outcome class of $G + H$.

Case 1: $G + H$ is a P-position. Then, by commutativity and associativity of game sums, $(G + A) + H = (G + H) + A$, where $G + H$ and A are both P-positions. By Lemma 2.1, the sum of two P-positions is also a P-position. So $G + Hand(G + A) + H$ are in the same outcome class – they are both P-positions.

Case 2: $G + H$ is an N-position. Now $(G + A) + H = (G + H) + A$ is the sum of an N-position $(G + H)$ and a P-position (A) and so by Lemma 2.1 is an N-position, just like $G + H$.

In both cases, $G+H$ and $(G+A)+H$ are in the same outcome class no matter what H is, so by definition of $\approx$, $G \approx G + A$. □

Lemma 2.3. *For any finite, normal, impartial games G and H, $G \approx H$ if and only if $G + H$ is a P-position.*

Proof. $\leftarrow$: Suppose $G \approx H$. Then, by definition of $\approx$, $G+H$ and $H+H$ have the same outcome class. $H+H$ is a P-position, so $G + H$ is too.

$\rightarrow$: Suppose $G + H$ is a P-position. The by Lemma 2.2, $G + (G + H) \approx G$ and $H + (G + G) \approx H$. By the commutative and associative properties of game sums, $H + (G+G) = (H+G)+G = G+(H+G) = G + (G + H)$. And so by transitivity of $\approx$, $G \approx H$. □

2.1.4 The Sprague-Grundy Theorem

Finally we can prove the main result of this chapter: the Sprague-Grundy Theorem. The Sprague Grundy Theorem states that every finite, normal, impartial game is equivalent to some version of one-row Nim.

Theorem 2.4. *For any finite, normal, impartial game G, there is some $n \in \mathbb{N}$ such that $G \approx *n$*

Proof. As is often the case with finite games, the proof proceeds by induction on the length of the game. Base case: Suppose G is of length 0. Then G has no options and $G = \{\}$ Note that this is not an equivalence, but an equality – a game is defined mathematically as the set of options (moves) available, and for G to have length 0, G must have no available moves, otherwise G would have length at least 1. But $*0 =$ and so $G = *0$ and hence $G \approx *0$ by the reflexive property of $\approx$. So there is an n, namely 0, so that $G \approx *n$.

Induction step: Suppose $k > 0$ and for all games G' of length $k' < k$, there is an n' so that $G' \approx *n'$. We must find an n so that $G \approx *n$. As usual with induction on the length of the game, we write G in terms of its options in order to use what the inductive hypothesis says about those options – we will find n based on the nimbers that the options of G are equivalent to.

So write $G = \{G_1, ..., G_l\}$. Each of the G_i is a game of length at most $k - 1$, so the inductive hypothesis applies to them. For each G_i, find the n_i so that $G_i \approx *n_i$ and let $G' = \{*n_1, \ldots, *n_l\}$.

Now we have $G \approx G'$. (We are actually using another lemma here: that if a game is equal to a set of options and each option is equivalent to some nimber, that the original game is equivalent to the game where the options are each of those nimbers. Formally, this lemma states that if for some game H, we have $H = H_1, \ldots, H_l$ and $H_1 \approx *n_1$, $H_2 \approx *n_2$, and so on, then $H \approx \{*n_1, \ldots * n_l\}$. We leave the proof to the reader (hint: use Lemma 2.3 and the definitions of game sums, N-positions, and P-positions).

If we can show that there is an n such that $G' \approx *n$, then we would also have $G \approx *n$ by transitivity of $\approx$, and the proof would be complete. We show that such an n exists by showing how to compute it from the n_i: let n be the smallest non-negative integer that is not equal to any of the n_i. This quantity is called the *minimum excludant*, or mex. Formally, for a set S of non-negative integers, $mex(S) = \min_{i \in \mathbb{N} - S} i$. Call the mex of all the n_i n: $n = mex(\{n_1, \ldots, n_l\})$. To show that $G' \equiv *n$, we use the lemma that states that two games are equivalent if and only if their sum is a P-position (losing). We show that $G' + *n$ is a P-position using the definition of P-position: that every move is to an N-position (winning). We do that by dividing

the possible moves from $G' + *n$ into 3 classes based on which game in that sum they move on and, for the former term, what the resulting game is equivalent to, and then showing that all the moves in each class are to N-positions.

Case 1: moves from $G' + *n$ on $*n$. The game $*n$ is one-row Nim starting with n stones, so the options available are to take some stones so that the remaining number of stones is in the range 0 to $n - 1$. So any such move on $G' + *n$ results in a position $G' + *j$ where $j < n$. Those positions are all N-positions (winning); to show that, it suffices to find a move on them to P-positions. The winning move from $G' + *j$ is to $G_i + *j$ where $G_i \equiv *j$; using that equivalence with Lemma 2.3 yields that $G_i + *j$ is a P-position. We know that there is such a G_i by the choice of n to be the mex of all the n_i: n is the smallest non-negative integer so that n is not among $n_1, \ldots, n_l$, so anything smaller is on that list somewhere; $j < n$, so some $n_i = j$. And that n_i is what G_i is equivalent to.

Case 2: moves from $G' + *n$ on G' to an n_i such that $n_i < n$. G' is defined as the game with options $\{*n_1, \ldots, *n_l\}$, so any move on G' is to one of those options. When the option chosen is $*n_i$ with $n_i < n$, the resulting position $*n_i + *n$ is an N-position because it is legal to move from there to the P-position $*n_i + *n_i$ by removing stones from the second game in the sum.

Case 3: moves from $G' + *n$ on G' to an n_i such that $n_i > n$. This is possible! In one-row Nim, the only moves will decrease the equivalent number of stones, but in general that is not the case. Even so, the same strategy as in the previous case applies, but by moving on the other game in the resulting sum $*n_i + *n$: $n_i > n$, so it is legal to move on the first term $*n_i$ to $*n$ to leave $*n + *n$, which is a P-position.

We've looked two steps ahead from $G' + *n$: any move from there falls into one of the three cases, and any move that falls into one of those cases has a response that is a winning move for the responding player. So, no matter what a player does from $G' + *n$, their opponent has a winning response. So $G' + *n$ is a P-position (losing). And to recap, Lemma equivP then says that $G' \equiv *n$, and since $G \equiv G'$, we also have $G \equiv *n$. So there is an n (namely $mex(\{n_1, \ldots, n_l\})$) so that $G \equiv *n$. That was the goal of the inductive step, and the proof is complete. □

The following corollary follows from the proof of the Sprague-Grundy theorem, since the proof shows not just that there exists an n so that $G \approx *n$, but also constructively shows how to compute that n.

Corollary 2.5. *For any finite, normal, impartial game $G = \{G_1, \ldots, G_l\}$ where, for each i, $G_i \approx *n_i$, $G \approx *mex(\{n_1, \ldots, n_l\})$.*

The n such that G is equivalent to the corresponding nimber ($G \approx *n$) is called the *Grundy number* for or the *nim-value* of G.

For the multi-row version of Nim, the Grundy number of the game is the Nim-sum. We start by proving that for two-row Nim and then showing how to extend that result to an arbitrary number of rows.

Theorem 2.6. *For any non-negative n and m, $*n + *m \approx *(n \oplus m)$.*

Proof. The proof is by induction on the length of the game (n+m).

Base case (n + m = 0). In this case, $n = m = 0$, so we need to show that $*0 + *0 \approx *(0 \oplus 0) = *0$. The game $*0 + *0$ has no options and so is equal to (not just equivalent to) the game $\{\}$. But we also have $*0 = \{\}$, so, by transitivity of $=$, $*n + *m = *0$. And, by the reflexive property of $\approx$, $*n + *m \approx *(0)$.

Inductive Step. Assume that $k > 0$ and that for all non-negative integers n' and m' with $n' + m' < k$, $*n' + *m' \approx *(n' \oplus m')$. Suppose that n and m are non-negative integers such that $n + m = k$. We need to show that $*n + *m \approx *(n \oplus m)$.

We know from Corollary 2.5 that to compute the Grundy number for $*n + *m$, we can enumerate the options of $*n + *m$, find the Grundy numbers for those options, and find the minimum excludant. The options of $*n + *m$ come from moving on $*n$ to leave $*n' + *m$ where $n' < n$, or from moving on $*m$ to leave $*n + *m'$ where $m' < m$. In both cases, the resulting game is shorter and so the inductive hypothesis applies:

$*n' + *m \approx *(n' \oplus m)$ when $n' < n$ and $*n + *m' \approx *(n \oplus m')$ when $m' < m$. From Corollary 2.5 we conclude that $*n + *m \approx *mex(\{n' \oplus m' \mid (n' < n \, and \, m' = m) \, or (n' = n \, and \, m < m')\})$. To complete the proof, we show that $mex(\{n' \oplus m' \mid (n' < n \, and \, m' = m) \, or (n' = n \, and \, m < m')\}) = n \oplus m$.

In general, to show that $x = mex(S)$, we show that x is the minimum excludant of S: $x \notin S$, and every smaller non-negative integer y satisfies $y \in S$. So, to show that $n \oplus m = mex(\{n' \oplus m' \mid (n' < n \, and \, m' = m) \, or (n' = n \, and \, m < m')\})$, we show first that for any non-negative n' and m' with either $n' < n$ and $m' = m$ or $n' = n$ and $m' < m$, $n \oplus m \neq n' \oplus m'$, and second, that if $y < n \oplus m$, then $y = n' \oplus m'$ for some non-negative n and m with either $n' < n$ and $m' = m$ or $n' = n$ and $m' < m$.

We divide the first part into two cases: a) $n' < n$ and $m' = m$; and b) $n' = n$ and $m' < m$. If $n' < n$ and $m' = m$, then, examining the binary representation of n' and n, it must be that the most significant bit that is different in n and n' is a 1 in n and a 0 in n'. That means that the corresponding bits in $n \oplus m$ and $n' \oplus m$ are also different: if the corresponding bit in m is a 0, then the two bits are $1 \oplus 0 = 1$ and $0 \oplus 0 = 0$ in $n \oplus m$ and $n' \oplus m$ respectively, and if the bit in m is a 1, then the two bits are $1 \oplus 1 = 0$ and $1 \oplus 0 = 1$. Either way, $n' \oplus m$ and $n \oplus m$ differ in that bit, so must be unequal: $n' \oplus m \neq n \oplus m$ The $n' = n, m' < m$ case is similar.

(Equivalently, to show that any two numbers are unequal, it suffices to show that their exclusive or is non-zero. $(n' \oplus m) \oplus (n \oplus m) = (n' \oplus n) \oplus (m \oplus m)$ by associativity and commutativity of $\oplus$. Since $m \oplus m = 0$, that means $(n' \oplus m) \oplus (n \oplus m) = (n' \oplus n)$, which is non-zero since $n' \neq n$. Again, the $n + m'$ case is similar.)

So $n \oplus m$ is an excludant. The second part is to show that it is the *minimum* excludant: that for every non-negative $y < n \oplus m$, $y = n' \oplus m$ for some non-negative $n' < n$ or $y = n \oplus m'$ for some non-negative $m' < m$. So let $y < n \oplus m$ and examine the bits again – find the most significant bit in which y and $n \oplus m$ differ. Since $y < n \oplus m$, y must have a 0 there and $n \oplus m$ must have a 1. To get a 1 in $n \oplus m$, either n has a 1 and m a 0, or the other way around. Assume, without loss of generality, that n has a 1 and m has a 0. Now, solve $y = n' \oplus m$ for n' by exclusive-or-ing both sides with m to get $n' = y \oplus m$. This n' is non-negative because $\mathbb{N}$ is closed under $\oplus$. If we can show that $n' < n$ then we have found exactly what we are looking for. In the more significant bits than the most significant difference between y and $n \oplus m$, $n' = y \oplus m = n \oplus m \oplus m = n$, so n' agrees with n. At the most significant difference between y and $n \oplus m$, $n' = y \oplus m = 0 \oplus 0 = 0$ and n has a 1 and so $n' < n$. This shows that there is a non-negative n' such that $n' < n$ and $y = n' \oplus m$ (namely, $n' = y \oplus m$).

□

For multi-row Nim, take advantage of the associative property of game sums and $\oplus$. For example,

$$\begin{aligned} *n_1 + *n_2 + *n_3 + *n_4 &= (*n_1 + *n_2) + (*n_3 + *n_4) \\ &\approx *(n_1 \oplus n_2) + *(n_3 \oplus n_4) \\ &\approx *(n_1 \oplus n_2) \oplus (n_3 \oplus n_4) \\ &= *(n_1 \oplus n_2 \oplus n_3 \oplus n_4). \end{aligned}$$

(Formally, one could prove this by induction on the number of rows.)

2.1.5 Using Sprague-Grundy

The Sprague-Grundy Theorem (or, more precisely, Corollary 2.5) tells us how to compute Grundy numbers. Start with the terminal positions, which are equivalent to the terminal position $*0$ in one-row Nim because any terminal position H has no options, so $H = \{\}$, and by Corollary 2.5, $H \approx mex(\{\})$, and $mex(\{\}) = 0$. Then compute the Grundy numbers for the rest of the positions in order of increasing length by enumerating all of the options, looking up the Grundy number for each option, which you will have computed previously since those options are shorter, and computing the mex of the corresponding set of integers.

For example, for all of the positions possible in Welter's game starting with $1/1/2$, we have

G	Options	Equivalents	Grundy(G)
$4/0/0$	$\{\}$	$\{\}$	$mex(\{\}) = 0$
$3/1/0$	$\{4/0/0\}$	$\{*0\}$	$mex(\{0\}) = 1$
$2/2/0$	$\{4/0/0, 3/1/0\}$	$\{*0, *1\}$	$mex(\{0, 1\}) = 2$
$3/0/1$	$\{4/0/0, 3/1/0\}$	$\{*0, *2\}$	$mex(\{0, 2\}) = 1$
$2/1/1$	$\{3/0/1, 3/1/0, 2/2/0\}$	$\{*1, *1, *2\}$	$mex(\{1, 2\}) = 0$
$1/3/0$	$\{4/0/0, 3/1/0, 2/2/0\}$	$\{*0, *1, *2\}$	$mex(\{0, 1, 2\}) = 3$
$1/2/1$	$\{3/0/1, 2/1/1, 2/2/0, 1/3/0\}$	$\{*1, *0, *2, *3\}$	$mex(\{0, 1, 2, 3\}) = 4$
$2/0/2$	$\{4/0/0, 3/0/1, 2/2/0, 2/1/1\}$	$\{*0, *1, *2, *0\}$	$mex(\{0, 1, 2\}) = 3$
$1/1/2$	$\{2/0/2, 3/1/0, 2/1/1, 3/1/0, 1/2/1, 1/3/0\}$	$\{*3, *1, *0, *1, *4, *3\}$	$mex(\{0, 1, 3, 4\}) = 2$

For a sum of games, we make use of the fact that game equivalence and game sums behave like equality and addition between integers – just as $a = b$ and $c = d$ means $a + b = c + d$, if $A \approx B$ and $C \approx D$, then $A + C \approx B + D$. So, to compute the Grundy number for $A + C$, we can find the b and d so that $A \approx *b$ and $C \approx *d$, so $A + C \approx *b + *d \approx *(b \oplus d)$.

This is particularly useful for games like Kayles in which the moves result in positions that can be viewed as the sum of smaller games. For example, starting with 10 pins, one might take the one pin from as close to the middle as possible, leaving one group of 4 pins, a space, and a group of five pins. Subsequent moves then can't alter both the group of 4 and the group of 5 at the same time – the resulting position is equivalent to $K4+K5$. So when computing the Grundy numbers for a game of Kayles starting with 10 pins, we don't need to compute the Grundy numbers for all 512 possible positions. Instead, we need only tabulate the Grundy numbers for $K0, K1, \ldots, K10$ and then use Corollary 2.5 to compute the Grundy number for other positions as we need them. For example,

G	Options	Equivalents	
$K0$	$\{\}$	$\{\}$	n
$K1$	$\{ K0\}$	$\{ *0 \}$	m
$K2$	$\{ K0, K1\}$	$\{ *0, *1 \}$	me
$K3$	$\{K1, K1 + K1, K2\}$	$\{*1, *1 + *1, *2\}$	$mex(\{1, 1 \oplus 1,$
$K4$	$\{K2, K1 + K1, K3, K1 + K2\}$	$\{*2, *1 + *1, *3, *1 + *2\}$	$mex(\{2, 1 \oplus 1, 3, 1$
$K5$	$\{K3, K1 + K2, K4, K1 + K3, K2 + K2\}$	$\{*3, *1 + *2, *1, *1 + *3, *2 + *2\}$	$mex(\{3, 1 \oplus 2, 1, 1 \oplus 3,$
$K6$			
$K7$			
$K8$			
$K9$			
$K10$			

We have taken advantage of the commutative and associative properties of game sums to take some shortcuts, for example taking the second and next-to-last pins from $K4$ results in $K1 + K2$ and $K2 + K1$ respectively, but those are the same position. We leave it as an exercise to work through the last several rows of that table.

Finally, this gives us a way of determining a winning move for a sum of finite, normal, impartial games (if such a winning move exists).

- Find the Grundy number for each term in the sum
- Find a winning move (for the corresponding game of multi-row Nim by finding a move on a row that results in making the Nim-sum over all the rows 0.
- Find the move in the corresponding term of the original game that produces an equivalent result.

For example, for the position $K9 + K6 + K1 + K7$, we find the Grundy number for each term (each group of consecutive pins) to get the equivalent multi-row Nim position $*4 + *3 + *1 + *2$. The Nim-sum

is $4 \oplus 3 \oplus 1 \oplus 2 = 100_2 \oplus 011_2 \oplus 001_2 \oplus 010_2 = 100_2 = 4$. The most significant 1 is in the 4's place, and only $*4$ also has a 1 in that position, so we choose to move on that group. The move on that row that makes the Nim-sum 0 is to reduce that row to size $4 \oplus 4 = 0$. The term in the original sum of Kayles games that corresponds to the row $*4$ is $K9$. We need to find a move on $K9$ that results in a position that is equivalent to $*0$ (what we reduced $*4$ to for our winning move). Enumerating all of the possible moves on $K9$ and computing the Grundy numbers of the resulting positions will enable us to search for a move that results in a position equivalent to $*0$. We know that such a move must exist because $K9 \approx *4$, and the mex rule says that can only happen if there are moves to positions equivalent to each of $*0, *1, *2$, and $*3$. Performing the search reveals that taking the middle pin to leave $K4 + K4$ is the winning move. In general, there may be more that one winning move on the selected group, and there may be winning moves on groups other than the one corresponding to the rows with winning moves in the corresponding Nim game. For example, the winning move on $K8 + K6$ found by this procedure is to take two from the end of the group of 6 to leave $K8 + K4$, but another winning move is to take two from the end of the group of 8 to leave $K6 + K6$ – this *increases* the Grundy number of the group we moved on, which is not possible in one-row Nim. However, this method will always find *a* winning move if one exists.

Chapter 3

Stochastic Games

3.1 Finite Games

3.2 Cyclic Games

3.2.1 Policy Iteration

3.2.2 Value Iteration

Chapter 4

Classical Game Theory

s

4.1 Introduction

4.2 Zero-Sum Games

Note that any constant-sum game can be converted to an equivalent zero-sum game (in the sense that the equilibrium strategies will be the same). Suppose A, B are the payoff matrices for a constant sum game with total payoff for any combination of strategies for Player I and Player II equal to c (so $a_{ij} + b_{ij} = c$ for all i and j). Define a game with payoff matrices $A' = A - c$ and $B' = B$. The new game (A', B') is zero-sum, since for all i and j we have $a'_{ij} + b'_{ij} = a_{ij} - c + b_{ij} = a_{ij} + b_{ij} - c = c - c = 0$. Now consider the following process.

(1) Players I and II play the zero-sum game A'

(2) Player I is awarded an additional payoff of c

This can also be modelled as a matrix game, and the payoffs will be $A' + c = A$ and B – this is equivalent to the original game. But there is no strategy in step 2 since there are no choices to make – Player I is simply always awarded an additional payoff of c. So all of the strategy is in step 1, and so optimizing the strategy there is the same as optimizing the strategy for the whole game.

Definition 4.1. $v^- = max_i min_j a_{ij}$ is the minimum value Player I is guaranteed to win by employing the pure strategy with the best worst case from Player I's point of view. $v^+ = min_j max_i a_{ij}$ is the maximum value Player II is guaranteed to give up by employing the pure strategy with the best worst case from Player II's point of view.

Theorem 4.1. *For any zero-sum game A, $v^- \leq v^+$.*

Proof. Let A be a zero-sum game. Then

$$\begin{aligned}
\forall i, j', \min_{i,j'} a_{ij'} &\leq a_{ij} && \text{(min is less than any term in the min)} \\
\forall j, \max_{i} \min_{i,j'} a_{ij'} &\leq \max_{i} a_{ij} && \text{(max of smaller things is smaller)} \\
\min_{j} \max_{i} \min_{i,j'} a_{ij'} &\leq \min_{j} \max_{i} a_{ij} && \text{(min of smaller things is smaller)} \\
\max_{i} \min_{i,j'} a_{ij'} &\leq \min_{j} \max_{i} a_{ij} && \text{(j does not appear in the terms of the outer max on the LHS)} \\
\max_{i} \min_{i,j} a_{ij} &\leq \min_{j} \max_{i} a_{ij} && \text{(change of variable on LHS)} \\
v^- &\leq v^+ && \text{(definition of } v- \text{ and } v+\text{)}
\end{aligned}$$

□

If the maximum of the row minimums and the minimum of the column maximums coincide at the same location a_{ij} then the strategy profile where Player I plays column i and Player II plays column j has the special property that if either player were to unilaterally deviate from their strategy, the deviating player would be worse off (or, at least, no better off). This is a special case of a *Nash equilibrium*. In general, a Nash equilibrium is a strategy profile where any player who unilaterally deviates from their strategy in that profile is worse off. We consider here Nash equilibria for zero-sum games where the component strategies of the equilibrium are pure strategies; we later extend the notion to other strategies and then to non-zero-sum games. Computing v^- and v^+ will tell us whether a game has a Nash equilibrium in pure strategies.

Definition 4.2. A Nash equilibrium in pure strategies for a two-player, zero-sum game A is a strategy profile (i, j) such that 1) $a_{i'j} \leq a_{ij}$ for all rows i, and 2) $a_{ij'} \geq a_{ij}$ for all rows j.

Theorem 4.2. *Let A be a zero-sum game. A has a Nash equilibrium in pure strategies if and only if $v^- = v^+$.*

Proof. ($\Leftarrow$) : Suppose A has a Nash equilibrium in pure strategies. Let (i, j) be the equilibrium strategy profile. We will show that $v^- = a_{ij} = v^+$. To show that $v^- = a_{ij}$, we first show that a_{ij} is the minimum of its row, and then that no other row can have a higher minimum, so that a_{ij} is the maximum row minimum, which is, by definition, v^-.

First, a_{ij} is the minimum of row i. $a_{ij} \leq a_{ij'}$ for all columns j' by definition of a pure strategy Nash equilibrium. So $a_{ij} \leq \min_j a_{ij}$. But because a_{ij} is a term in the min, $\min_j \leq a_{ij}$. Therefore $a_{ij} \min_j \leq a_{ij}$ because we have the two terms less than or equal to each other.

Second, a_{ij}, the minimum of row i, is the maximum row minimum. For suppose that some other row i'' has a higher minimum: $\min_{j'} a_{i''j'} > a_{ij}$. Every column in row i'' has a value that is at least that minimum: for all j'', $ai''j'' \geq \min_{j'} a_{i''j'} > a_{ij}$. In particular, $a_{i''j} \geq \min_{j'} a_{i''j'} > a_{ij}$. But by the definition of a Nash equilibrium, $a_{ij} > a_{i'j}$ for all rows i'. In particular, when $i' = i''$ we have $a_{ij} > a_{i''j}$, which contradicts the previous conclusion that $a_{ij} < a_{i''j}$. We must rejection the assumption that there was a row i'' with a higher minimum entry than a_{ij}, so for all rows i', we must have $\min_{j'} a_{i'j} \leq a_{ij} = \min_{j'} a_{ij'}$. Taking the maximum over all rows i' gives $\max_{i'} \min_{j'} a_{i'j} \leq \max_{i'} \min_{j'} a_{ij'}$ and, since i' does not appear in the terms in the outer max on the right-hand side, $\max_{i'} \min_{j'} a_{i'j} \leq \min_{j'} a_{ij'}$, and finally $v^- = a_{ij}$ by substitution.

The argument for $a_{ij} = v^+$ is similar and we leave it as an exercise.

($\Rightarrow$): Suppose $v^- = v^+$; call this value v. We need to show that there is a Nash equilibrium in pure strategies. We find that equilibrium from v^- and v^+: let i be the (or a) row that maximizes $\min_{j'} a_{ij'}$, so $\min_{j'} a_{ij'} = \max_{i'} \min_{j'} a_{i'j'} = v^- = v^+ = v$, and let j be the (or a) column that minimizes $\max_{i'} a_{i'j}$, so $\max_{i'} a_{i'j} = \min_{j'} \max_{i'} a_{i'j'} = v^+ = v^- = v$. Since a_{ij} is in row i, it is at least as large as the minimum in that row: $a_{ij} \geq \min_{j'} a_{ij'} = v$. Since a_{ij} is in column j, it is no larger than the maximum in that column:

$a_{ij} \leq \max_{i'} a_{i'j} = v$. Now we have both $a_{ij} \geq v$ and $a_{ij} \leq v$, so $a_{ij} = v$. Hence $a_{ij} = v = \min_{j'} a_{ij'}$, in other words, a_{ij} is the minimum of row i: for all columns j', $a_{ij} \leq a_{ij'}$. Also, $a_{ij} = v = \max_{i'} a_{i'j}$, or a_{ij} is the maximum of column j: for all rows i', $a_{ij} \geq a_{i'j}$. So the strategy profile (i, j) satisfies the definition of a Nash equilibrium in pure strategies: if Player I deviates from row i, their payoff goes down, and if Player II deviates from row j, the payoff to Player I goes up, and hence the payoff to Player II goes down – each player is worse off after a unilateral change. □

Nash equilibria are one *solution concept* for matrix games. A solution concept attempts to predict the behavior of players in a game. A Nash equilibrium describes how rational actors who know they are all rational would choose strategies if their plans were leaked and made public. For a two-player game, if Player I knew what column Player II was going to choose, they could choose the row with the maximum value in that column. But Player I also knows that Player II could foresee that and play instead the column with the minimum value in that new row. But then Player I can choose yet another row that maximizes the value in that new column, and so on. A Nash equilibrium in pure strategies stops that sequence of reasoning.

4.2.1 Finding a Nash Equilibrium

The definition of a Nash equilibrium in mixed strategies says that if (X, Y) is a Nash equilibrium then if either player changes unliaterally to any other mixed strategy, that player is no better off than they were before the change. But it is also a necessary and sufficient condition for (x, Y) to be a Nash equilibrium that either player making a unilateral change to any *pure* strategy leaves that player no better off. This eases verification of a Nash equilibrium, because to show that (X, Y) satisfies the definition would require evaluating the inequalities for an infinite number of mixed strategies. Using this new theorem only requires comparison to the finite number of pure stratgies. And finding a Nash equilibrium is then equivalent to finding a mixed strategy profile that satisfies the finite number of conditions, which can be done with linear programming.

Theorem 4.3. *For any zero-sum game A, a mixed strategy profile (X, Y) is a Nash equilibrium and $value(A) = E(X, Y)$ if and only if $E(i, Y) \leq E(X, Y) \leq E(X, j)$ for all rows i and columns j.*

Proof. ($\Rightarrow$): Suppose that strategy profile (X, Y) is a Nash equilibrium. Let i be any row and j be any column. The definition of a Nash equilibrium requires that any player making a unilateral change from their strategy in the equilibrium profile to another mixed strategy leaves that player no better off. But a pure strategy is just a special case of a mixed strategy, so the "no better off" condition applies to the pure strategy where Player I chooses row i and Player II chooses column j: $E(i, Y) \leq E(X, Y) \leq E(X, j)$ as stated by the theorem.

($\Leftarrow$): Suppose $E(i, Y) \leq E(X, Y) \leq E(X, J)$ for all rows i and columns j. Let X' and Y' be any mixed strategies for Player I and Player II respectively. Write $X' = (x'_1, \ldots, x'_n)$ and $Y' = (y'_1, \ldots, y'_m)$. We show first that if Player I unilaterally changes to X' that they are no better off first ($E(X', Y) \leq E(X, Y)$) and then that if Player II unilaterally changes to Y' that they are no better off ($E(X, Y) \leq E(X, Y')$.

$$
\begin{aligned}
E(X', Y) = & \quad \sum_{i=1}^{n} x'_i \cdot E(i, Y) && \text{(definition of value of mixed strategy profile)} \\
\leq & \quad \sum_{i=1}^{n} x'_i \cdot E(X, Y) && \text{(assumption that any switch to pure strategy leaves PI no better off)} \\
= & \quad E(X, Y) \cdot \sum_{i=1}^{n} x'_i && \text{(factor out constant term)} \\
= & \quad E(X, Y). && \text{(components of mixed strategy must sum to 1)}
\end{aligned}
$$

The reasoning is the same to show that $E(X,Y) \leq E(X,Y')$, or, equivalently, that $E(X,Y') \geq E(X,Y)$:

$$\begin{aligned} E(X',Y) &= \sum_{j=1}^{m} y'_j \cdot E(i,Y) \\ &\geq \sum_{j=1}^{m} y'_j \cdot E(X,Y) \\ &= E(X,Y) \cdot \sum_{j=1}^{m} y'_j \\ &= E(X,Y). \end{aligned}$$

□

Example: we use Theorem 4.3 to verify that (X,Y) where $X = (\frac{1}{3}, \frac{5}{12}, \frac{1}{4})$ and $Y = (\frac{1}{3}, 0, \frac{1}{6}, \frac{1}{2})$ is a Nash equilibrium for

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 2 & 4 & 1 & 2 \\ 4 & 2 & 3 & 0 \end{pmatrix}$$

First, compute the values of X against Player II's pure strategies:

$$\begin{aligned} E(X,1) &= \frac{1}{3} \cdot 0 + \frac{5}{12} \cdot 2 + \frac{1}{4} \cdot 4 = \frac{11}{6} \\ E(X,2) &= \frac{1}{3} \cdot 1 + \frac{5}{12} \cdot 4 + \frac{1}{4} \cdot 2 = \frac{5}{2} \\ E(X,3) &= \frac{1}{3} \cdot 2 + \frac{5}{12} \cdot 1 + \frac{1}{4} \cdot 3 = \frac{11}{6} \\ E(X,4) &= \frac{1}{3} \cdot 3 + \frac{5}{12} \cdot 2 + \frac{1}{4} \cdot 0 = \frac{11}{6} \end{aligned}$$

Having these allows us to compute $E(X,Y)$: $E(X,Y) = y_1 \cdot E(X,1) + y_2 \cdot E(X,2) + y_3 \cdot E(X,3) + y_4 \cdot E(X,4) = \frac{1}{3} \cdot \frac{11}{6} + 0 \cdot \frac{5}{2} + \frac{1}{6} \cdot \frac{11}{6} + \frac{1}{2} \cdot \frac{11}{6} = (\frac{1}{3} + \frac{1}{6} + \frac{1}{2}) \cdot \frac{11}{6} = \frac{11}{6}$.

We then confirm that $E(X,j) \geq E(X,Y)$ for all columns j. The only column for which they are unequal is $j = 2$ and indeed $E(X,2) = \frac{5}{2} \geq \frac{11}{6} = E(X,Y)$.

Now compute the values of Y against Player I's pure strategies and confirm that all are no greater than $E(X,Y) = \frac{11}{6}$.

$$\begin{aligned} E(1,Y) &= \frac{1}{3} \cdot 0 + 0 \cdot 1 + \frac{1}{6} \cdot 2 + \frac{1}{2} \cdot 3 = \frac{11}{6} \leq \frac{11}{6} \\ E(2,Y) &= \frac{1}{3} \cdot 2 + 0 \cdot 4 + \frac{1}{6} \cdot 1 + \frac{1}{2} \cdot 2 = \frac{11}{6} \leq \frac{11}{6} \\ E(3,Y) &= \frac{1}{3} \cdot 4 + 0 \cdot 2 + \frac{1}{6} \cdot 3 + \frac{1}{2} \cdot 0 = \frac{11}{6} \leq \frac{11}{6} \end{aligned}$$

So (X,Y) is a Nash equilibrium for A and $value(A) = \frac{11}{6}$.

We can also confirm that (X, Y') where $X' = (\frac{1}{3}, \frac{1}{3}, \frac{1}{3}$ and $Y' = (\frac{1}{2}, 0, 0, \frac{1}{2})$ is not a Nash equilibrium. Now

$$\begin{aligned} E(X', 1) &= \frac{1}{3} \cdot 0 + \frac{1}{3} \cdot 2 + \frac{1}{3} \cdot 4 = & 2 \\ E(X', 2) &= \frac{1}{3} \cdot 1 + \frac{1}{3} \cdot 4 + \frac{1}{3} \cdot 2 = & \frac{7}{3} \\ E(X', 3) &= \frac{1}{3} \cdot 2 + \frac{1}{3} \cdot 1 + \frac{1}{3} \cdot 3 = & 2 \\ E(X', 4) &= \frac{1}{3} \cdot 3 + \frac{1}{3} \cdot 2 + \frac{1}{3} \cdot 0 = & \frac{5}{3} \end{aligned}$$

and $E(X', Y') = \frac{1}{2} \cdot 2 + \frac{1}{2} \cdot \frac{5}{3} = \frac{11}{6}$, but Player II is better off switching to playing column 4 all the time: $E(X', 4) < E(X', Y')$ (and Player II's best response to X' is playing column 4 all the time, since $j = 4$ minimizes $E(X', j)$).

We already know that (X', Y') is not a Nash equilibrium. Completing the calculations of $E(i, Y')$ will tell us Player I's best response to Y'.

$$\begin{aligned} E(1, Y') &= \frac{1}{2} \cdot 0 + \frac{1}{2} \cdot 3 = & \frac{3}{2} \\ E(2, Y') &= \frac{1}{2} \cdot 2 + \frac{1}{2} \cdot 2 = & 2 \\ E(3, Y') &= \frac{1}{2} \cdot 4 + \frac{1}{2} \cdot 0 = & 2 \end{aligned}$$

So either row 2 or row 3, or any mixed combination of those is a best response to Y' (and we have $E(2, Y'), E(3, Y') \geq E(X', Y')$, showing again that (X', Y') is not a Nash equilibrium.

Linear Programming

Linear programming is the problem of finding values for n real-valued variables that maximize or minimize some linear combination of those variables, subject to bounds on other linear combinations of those variables. In the *standard form* of a linear programming problem, the objective is to maximize a linear combination of the variables, the values assigned to the variables must be non-negative, and the bounds are expressed as upper bounds on linear combinations of the variables. So, in general, a linear program is

$$\begin{aligned} \text{Find } & x_1, x_2, \ldots, x_n \\ \text{To maximize } & c_1 x_1 + \cdots + c_n x_n \\ & \text{Subject to} \\ a_{11} x_1 + a_{12} x_2 + \cdots + a_{1n} x_n &\leq & b_1 \\ a_{21} x_1 + a_{22} x_2 + \cdots + a_{2n} x_n &\leq & b_2 \\ & \vdots \\ a_{k1} x_1 + a_{k2} x_2 + \cdots + a_{kn} x_n &\leq & b_k \\ x_1, \ldots, x_n &\geq & 0. \end{aligned}$$

As a matrix problem, this is the problem of finding an n-element column vector $\boldsymbol{x}$ to maximize $\boldsymbol{c}^T\boldsymbol{x}$ subject to the constraints $\boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}$ and $\boldsymbol{x} \geq 0$.

The conditions on the $E(X, j)$ from Theorem 4.3 give us the bulk of the constraints on the x_i: for an n-by$m$ matrix game A, we get

$$\begin{aligned} a_{11}x_1 + a_{21}x_2 + \cdots + a_{n1}x_n &\geq & v \\ a_{12}x_1 + a_{22}x_2 + \cdots + a_{n2}x_n &\geq & v \\ &\vdots & \\ a_{1m}x_1 + a_{2m}x_2 + \cdots + a_{nm}x_n &\geq & v \end{aligned}$$

Note that the matrix of coefficients in the constraints here is the transpose of the payoff matrix ($\boldsymbol{A}^T$). v here is the value of the game and is a variable in the linear program; the objective is to maximize v (Player I wants to find a strategy that maximizes the payoff to them). To get this into standard form, multiply each inequality by -1 and move the v to the other side to yield

$$\begin{aligned} -a_{11}x_1 - a_{21}x_2 - \cdots - a_{n1}x_n + v &\leq & 0 \\ -a_{12}x_1 - a_{22}x_2 - \cdots - a_{n2}x_n + v &\leq & 0 \\ &\vdots & \\ -a_{1m}x_1 - a_{2m}x_2 - \cdots - a_{nm}x_n + v &\leq & 0 \end{aligned}$$

For the standard form, linear programming will find values of the variables that are non-negative. Since the linear program will find the value of v, v must also be non-negative. One way to ensure that the value of the game is non-negative is to add a constant to the payoff matrix A to ensure that every entry is positive, since if the payoff for every combination of pure strategies is positive, the payoff for every combination of mixed strategies will be positive as well. $1 - \min_{i,j} a_{ij}$ will suffice for the constant to add to the entries.

The final constraint is that $x_1 + \cdots + x_n = 1$, which is expressed in standard form as $x_1 + \cdots + x_n \leq 1$. The rest of the constraints coupled with the objective of maximizing the value v ensure that the sum of the x_i will be 1 – a sum less than 1 would correspond to choosing to not play some of the time, but with all the payoffs positive, there is always incentive for Player I to play.

We can go a step further and transform the linear program into an equivalent one that does not include the extra variable for v. Let $p_i = \frac{x_i}{v}$ and transform the main set of inequalities by dividing by v (which is legal and does not change the direction of the inequalities since we have ensured that v is positive) to get

$$\begin{aligned} -a_{11}p_1 - a_{21}p_2 - \cdots - p_{n1}x_n &\leq & -1 \\ -a_{12}p_1 - a_{22}p_2 - \cdots - p_{n2}x_n &\leq & -1 \\ &\vdots & \\ -a_{1m}p_1 - a_{2m}p_2 - \cdots - p_{nm}x_n &\leq & -1 \end{aligned}$$

The objective was to maximize v; with v eliminated we maximize $-p_1 - \cdots - p_n$. This is equivalent to the original objective since $-p_1 - \cdots - p_n = -\frac{x_1}{v} - \cdots - \frac{x_n}{v} = -\frac{x_1+\cdots+x_n}{v} = -\frac{1}{v}$, and maximizing $-\frac{1}{v}$ is the same as maximizing v.

The linear program will find the p_i, from which we can calculate v, and then the x_i since $p_i = \frac{x_i}{v}$ and hence $x_i = p_i v$.

A separate linear program will find the y_i; it is based on the conditions on $E(i, Y)$ from Theorem 4.3: find $y_1, \ldots, y_m$ and v to maximize $-v$ (equivalent to Player II's goal of minimizing v, the payoff to Player I) subject to the constraints

$$\begin{aligned} a_{11}y_1 + a_{12}y_2 + \cdots + a_{1m}y_m - v &\leq 0 \\ a_{21}y_1 + a_{22}y_2 + \cdots + a_{2m}y_m - v &\leq 0 \\ &\vdots \\ a_{n1}y_1 + a_{n2}y_2 + \cdots + a_{nm}y_m - v &\leq 0 \end{aligned}$$

and $y_1 + \cdots + y_m \leq 1$ and $-y_1 - \cdots - y_m \leq -1$ (which together guarantee that the sum of the y_i is 1). After eliminating v and setting $q_i = \frac{y_i}{v}$, this becomes the problem of maximizing $q_1 + \cdots + q_m$ subject to

$$\begin{aligned} a_{11}q_1 + a_{12}q_2 + \cdots + a_{1m}q_m &\leq 1 \\ a_{21}q_1 + a_{22}q_2 + \cdots + a_{2m}q_n &\leq 1 \\ &\vdots \\ a_{n1}q_1 + a_{n2}q_2 + \cdots + a_{nm}q_n &\leq 1 \end{aligned}$$

And again, once solving the linear program for the q_i, compute v and use $y_i = q_i v$ to determine the equilibrium strategy for Player II.

Note that there may be multiple solutions to a linear program; each one is a Nash equilibrium.

4.2.2 Special Cases

The constraints in a linear program define a convex polytope (the generalization of a polygon to an arbitrary number of dimensions). Linear programming algorithms find the vertex of that polytope that maximizes (or minimizes) the objective function. The *simplex method* works by starting at one vertex and moving to an adjacent vertex with a higher (or equal) value of the objective function. The simplex method is exponential in the worst case, but generally performs well on practical problems such as those that arise from finding a Nash equilibrium. There are other algorithms that solve linear programming problems that work in polynomial time.

For a 2-by-2 matrix, the resulting linear program is easy to solve by hand. When there are only two choices of row and column, we can describe any strategy for Player I or Player II by just one variable, since $x_2 = 1 - x_1$ and $y_2 = 1 - y_1$. The constraints from Theorem 4.3 then define regions on one side of lines in the plane. For example, for the matrix

$$\begin{pmatrix} 5 & 3 \\ 2 & 8 \end{pmatrix}$$

we have

$$\begin{aligned} E(X, 1) = \quad & 5x_1 + 2(1 - x_1) \geq \quad v \\ E(X, 2) = \quad & 3x_1 + 8(1 - x_1) \geq \quad v \end{aligned}$$

which simplifies to $v \leq 2 + 3x_1$ and $v \leq 8 - 5x_1$. Graphing the lines that border those regions yields Figure ??.

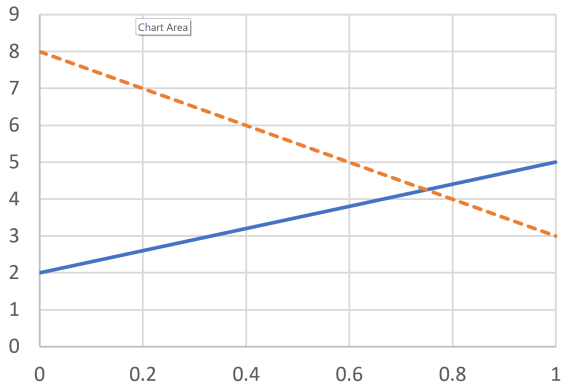

Figure 4.1: The x-axis is x_1 and the y-axis is v. The solid blue line is the constraint from $E(X, 1) \geq v$ and the dashed yellow line is from $E(X, 2) \geq v$.

Since the constraints require that v be less than the value of those lines, for each constraint, the region that satisfies the constraint is the area below the line. The feasible region – the region where all the constraints are satisfied – is the intersection of those below-the-line regions, or the region below all the lines. For any x_1, the lines correspond to $E(X, 1)$ and $E(X, 2)$; since there is always a best response in pure strategies, one of those is Player II's best response, and since Player II wants to minimize the payoff to Player I, the best response is the column that corresponds to the lower line – where $E(X, 1)$ is below $E(X, 2)$, $Y = (10)$ is the best response, and where $E(X, 2)$ is below $E(X, 1)$, $Y = (01)$ is the best response.

The objective of the linear program to find $\boldsymbol{x}$ is to maximize v, and the maximum possible value of v is the highest point in the feasible region. That point is at the intersection of the two lines, so solving for the intersection of the lines gives x_1.

$$\begin{aligned} 2 + 3x_1 &= & 8 - 5x_1 \\ 8x_1 &= & 6 \\ x_1 &= & \frac{3}{4} \end{aligned}$$

And since $x_2 = 1 - x_1$, we have $x_2 = \frac{1}{4}$ and so the strategy for Player I in the Nash equilibrium is $(\frac{3}{4}, \frac{1}{4})$.

The same process using the constraints on $E(1, Y)$ will give the equilibrium strategy for Player II:

$$\begin{aligned} E(1, Y) &= & 5y_1 + 3(1 - y_1) &\leq & v \\ E(2, Y) &= & 2y_1 + 8(1 - y_1) &\leq & v \end{aligned}$$

or $v \geq 3 + 2y_1$ and $v \geq 8 - 6y_1$.

This time the feasible region is the area above all of the lines, but the objective is to minimize v, and the minimum value of v is at the intersection of the lines again. For any given y_1, Player I's best response is the row corresponding to the higher of the two lines. Solving for the intersection yields

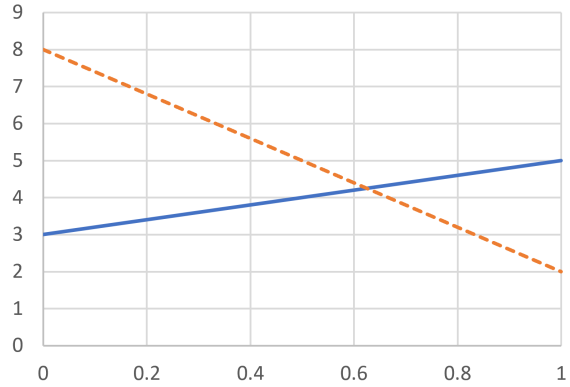

Figure 4.2: The x-axis is y_1 and the y-axis is v. The solid blue line is the constraint from $E(1, Y) \geq v$ and the dashed yellow line is from $E(2, Y) \geq v$.

$$\begin{aligned} 3 + 2y_1 &= & 8 - 6y_1 \\ 8y_1 &= & 5 \\ y_1 &= & \frac{5}{8} \end{aligned}$$

and so $y_2 = 1 - y_1 = \frac{3}{8}$. The Nash equilibrium is then (X^*, Y^*) where $X^* = ((\frac{3}{4}, \frac{1}{4})$ and $Y^* = (\frac{5}{8}, \frac{3}{8})$.

4.3 Non-constant-sum Games

We can define Nash equilibrium for non-constant-sum games too. The idea is the same: a Nash equilibrium is a strategy profile such that if any player unilaterally deviates from the equilibrium strategy, then that player is no better off than they were before. Because the payoffs to the players can be independent of each other, the definition must use the payoffs to each player.

Definition 4.3. For a two-player non-constant-sum game, a mixed strategy profile (X, Y) is a Nash equilibrium if both 1) for any Player I mixed strategy X', $E_I(X', Y) \leq E_I(X, Y)$ and 2) for any Player II mixed strategy Y', $E_{II}(X, Y') \leq E_I I(X, Y)$.

And, just as for constant-sum games, every [two-player?] non-constant-sum game has a Nash equilibrium, and there is a theorem analagous to Theorem 4.3.

Theorem 4.4. ?? *For any two-player game, a mixed strategy profile (X, Y) is a Nash equilibrium if and only if $E_I(i, Y) \leq E_I(X, Y)$ for all rows i, and $E_{II}(X, j) \leq E_{II}(X, Y)$ for all columns j.*

For a zero-sum game, every Nash equilibrium has the same value as every other Nash equilibrium. That is not necessarily the case for a non-constant-sum game.

For example, for the Prisoners Dilemma with payoffs

$$\begin{array}{c} \\ Cooperate \\ Defect \end{array} \begin{array}{c} \begin{array}{cc} Cooperate & Defect \end{array} \\ \begin{pmatrix} 3,3 & 0,5 \\ 5,0 & 1,1 \end{pmatrix} \end{array}$$

there is a Nash equilibrium in pure strategies: both players defect. This generalizes to games of the form

$$\begin{array}{c} \\ Cooperate \\ Defect \end{array} \begin{array}{c} \begin{array}{cc} Cooperate & Defect \end{array} \\ \begin{pmatrix} R,R & S,T \\ T,S & P,P \end{pmatrix} \end{array}$$

where $T > R > P > S$.

When $R > T = P > S$ the game is called *Stag Hunt*, after a situation in which two hunters must cooperate to catch big game, or defect from the other to catch a smaller prize they can capture solo. For example,

$$\begin{array}{c} \\ Cooperate \\ Defect \end{array} \begin{array}{c} \begin{array}{cc} Cooperate & Defect \end{array} \\ \begin{pmatrix} 2,2 & 0,1 \\ 1,0 & 1,1 \end{pmatrix} \end{array}$$

Stag Hunt has two Nash equilibria in pure strategies: both cooperate, and both defect. Note that the values for each player are higher in the former than in the latter.

Opera vs. Baseball is another example of a non-constant-sum game. In this game, two people need to, without communicating, decide whether to attend a baseball game or the opera [1]. The payoffs reflect the preferences of each player – one likes baseball and gets a higher payoff from attending the baseball game than the opera, and one likes the opera and so gets a higher payoff from attending the opera. However, the value is wiped out if the players choose different events because of the ensuing argument, and if both choose the opera then the baseball fan still gets some value from the pleasure of the opera fan's company (and vice versa).

$$\begin{array}{c} \\ Baseball \\ Opera \end{array} \begin{array}{c} \begin{array}{cc} Baseball & Opera \end{array} \\ \begin{pmatrix} 4,1 & 0,0 \\ 0,0 & 1,4 \end{pmatrix} \end{array}$$

Again, there are two Nash equilibria where both players choose the same event. In this case there is also a mixed strategy equilibrium (X, Y) where $X = (\frac{4}{5}, \frac{1}{5})$ and $Y = X = (\frac{4}{5}, \frac{1}{5})$. This can be verified using Theorem **??**:

$$\begin{aligned} E_I(X,Y) &= \frac{4}{5}\cdot\frac{1}{5}\cdot 4 + \frac{4}{5}\cdot\frac{1}{5}\cdot 1 = \frac{4}{5} \\ E_I(1,Y) &= \frac{1}{5}\cdot 4 = \frac{4}{5} \leq \frac{4}{5} \\ E_I(2,Y) &= \frac{4}{5}\cdot 1 = \frac{4}{5} \leq \frac{4}{5} \\ E_{II}(X,Y) &= \frac{4}{5}\cdot\frac{1}{5}\cdot 1 + \frac{4}{5}\cdot\frac{1}{5}\cdot 4 = \frac{4}{5} \\ E_{II}(X,1) &= \frac{4}{5}\cdot 1 = \frac{4}{5} \leq \frac{4}{5} \\ E_{II}(X,2) &= \frac{1}{5}\cdot 4 = \frac{4}{5} \leq \frac{4}{5} \end{aligned}$$

For a non-constant-sum game with no pure strategy Nash equilibrium, we use Theorem **??** to set up an optimization problem. However, since the game is not zero-sum and the values at different equilibria may be different to the two players, we must solve the optimization problem for X and Y together. $E_I(X,Y) = \sum_i, jx_iy_ja_{ij}$ and $E_{II}(X,Y) = \sum_{i,j} x_iy_jb_{ij}$ appear in the system of constraints, and the x_iy_j terms are the product of two variables in the system, so are nonlinear. Since the system is nonlinear, we need different

[1] This is called Battle of the Sexes in the literature, but, at odds with the usual formulation, my father loved the opera and my mother loves baseball. Also, their battle ended in divorce.

methods to find solutions than Linear Programming. Those are out of the scope of these notes, but see for example [**Barron**] for more information.

Consider the game with payoffs

$$\begin{pmatrix} -1,1 & 0,2 & 0,2 \\ 2,1 & 1,-1 & 0,0 \\ 0,0 & 1,1, & 1,2 \end{pmatrix}$$

Verify that there is no pure strategy Nash equilibrium. To find a mixed strategy Nash equilibrium, we solve the following optimization problem.

$$\begin{aligned}
&\text{maximize} && -x_1y_1 + 2x_1y_1 + x_2y_2 + x_3y_2 + x_3y_3 \\
& && +x_1y_1 + 2x_1y_2 + 2x_1y_3 + x_2y_1 - x_2y_3 + x_3y_3 + 2x_3y_3 \\
& && -p - q \text{subject to} \\
E_I(1,Y) = & && -y_1 \leq && p \\
E_I(2,Y) = & && 2y_1 + y_2 \leq && p \\
E_I(3,Y) = & && -y_1 \leq && p \\
E_I(X,1) = & && x_1 + x_2 \leq && q \\
E_I(X,2) = & && 2x_1 - x_2 + x_3 \leq && q \\
E_I(X,3) = & && 2x_1 + 2x_3 \leq && q \\
x_1, x_2, x_3, y_1, y_2, y_3 \geq & && 0 && \\
x_1 + x_2 + x_3 = 1 & && && \\
y_1 + y_2 + y_3 = 1 & && &&
\end{aligned}$$

The objective is $E_I(X,Y) + E_{II}(X,Y) - p - q$, where p and q are new variables. At equilibrium, $p \geq E_I(X,Y)$ and $q \geq E_{II}(X,Y)$ in order for the other constraints to be satisfied. But since $-p$ and $-q$ appear in the function to be maximized, they will be chosen to be as small as possible, so $p = E_I(X,Y)$ and $q = E_{II}(X,Y)$. For any strategy profile (X,Y) that is not an equilibrium, there will be an i such that $E_I(i,Y) > E_I(X,Y)$ and/or a j such that $E_{II}(X,j) \geq E_{II}(X,Y)$. So in order to satisfy all the constraints, $p > E_I(X,Y)$ and/or $q > E_{II}(X,Y)$ and so the function to be maximized will be negative. So, the maximum value of the function to be maximized is 0, which occurs when (X,Y) is a Nash equilibrium and $p = E_I(X,Y)$ and $q = E_{II}(X,Y)$.

4.4 Extensive vs Normal Form

Imagine the following form of chess: two players write deterministic computer programs to play chess. These programs will take any position in the game in which it is the corresponding player's turn, and output the move selected by the player's program. The two players then run the programs against each other, and the author of the winning program is the winner.

The programming version of chess is equivalent to the standard version. A human player could simply study every possible position, write down what they would do in each position, and write a program that outputs the move on that list. The turn-based version of chess simply lessens the burden on the human players by only making them decide what moves to make in the dozens or so of situations that actually arise in the game rather than the astronomical number of possible chess positions.

The programming version of the chess is an example of turning a combinatorial game into a simultaneous game – each player writes their program independently, and they reveal them simultaneously and let them

run to determine the reward. The standard representation of a combinatorial game like chess as a game tree is called the *extensive form* and the version that has been converted to a simultaneous game is the *normal form*.

Poker is a popular imperfect information game and was the subject of many breakthroughs in the 2010s: [PLURUBUS, DEEP STACK, LIBRATUS, REBEL].

Imperfect information games can be handled the same way. In the extensive form, the game is represented as a tree. The initial state of the game is the root of the tree, and at each node there are child nodes corresponding to the actions the current player can take. In addition, to represent stochastic events like dealing cards or rolling dice, there can be chance nodes with a child node for each possible outcome of the random event.

Consider, for example, *Kuhn Poker*, a simplified version of poker. In the two-player version, each player antes 1 chip and is dealt one card from a deck containing only three cards: a jack, a queen, and a king. Each player sees only their card. The first player then bets 1 chip or checks (passes). The second player's possible actions then depend on the first player's action: if the first player bet then the second player can fold, or call (match Player I's bet of 1), ending the betting round either way; if the first player checked, then the second may also check, ending the betting, or the second player may bet 1, in which case Player I then has the option of folding or calling to end the betting round. If betting ended with one player folding, then the non-folding player wins the 2 chips anted plus any additional chips that the players bet; otherwise both players' cards are revealed and the player holding the highest card wins the ante plus subsequent bets.

[INSERT GAME TREE WITH INFORMATION SETS]

In the extensive form of Kuhn poker, the root represents the initial, pre-deal state of the game. The six children of the root represent the six possible outcomes of the deal, each occurring with probability $\frac{1}{6}$. The children of those nodes represent the initial check/bet decision for Player I, then the next layer of nodes corresponds to the fold/call or check/bet decision for Player II, and the last level of non-leaf nodes represents Player I's response when Player II bet after Player I called. The leaf nodes are labelled with the payoff for Player I (because Kuhn poker is a zero-sum game, that determines Player II's payoff, but for a non-constant-sum game, the leaves would be labelled with both players' payoffs).

The nodes in the extensive form game tree represent perfect information about the game. However, in an imperfect information game, players do not necessarily know what state the game is in – having been dealt a K in Kuhn poker, Player I does not know Player II's card, and so does not know whether the state of the game is KQ or KJ. Each player does know their own private *action-observation history* – a sequence of their observations of the actions of the game, where the observation of the results of an action may be different for different players. Under the assumption of *perfect recall*, each player's action-observation history contains all of their own moves, and upon transition from one state to the next, nothing is deleted from a player's observation history, but information may be added to it. Actions in the games we use as examples are fully observable, it may be the case that other players' actions are only *partially observable* – for example, there may be two different actions another player can take that result in the same observation for player i.

As an example of action-observation histories, consider the following events in Five-card Draw Poker.

Action	Player I's Observation	Player II's Observation
Deal P1:3C 3S 4D 7H QC P2: 5D 6C 7C QC KH	P1:3C 3S 4D 7H QC	P2: 5D 6C 7C 9C KH
P1 bets 1	P1 bets 1	P1 bets 1
P2 calls	P2 calls	P2 calls
P1 discards 4D 7H QC, draws 8D 9S KH	P1 discards 4D 7H QC draws 8D 9S KH	P1 discards/draws 3 cards
P2 discards KH, draws 8D	P2 discards/draws 1	P2 discards KH, draws 8D
⋮		

The resulting action-observation history for each player is the sequence of their observations after each action. Note that for betting actions, all information is public and the players have the same observations. For other actions, the players get different observations. And for discards, not only does the other player not know which cards replaced the discards, the other player doesn't know the exact action – which cards were discarded.

An *information set* is the set of possible perfect information states that the game might be in for a particular private action-observation history. Since action-observation histories vary by player, each player can have a different information set at a particular point in the game. For example, if the initial deal for Kuhn poker was KJ, then Player I's information set is $\{KQ, KJ\}$ and Player II's information set is $\{KJ, QJ\}$.

A *behavioral strategy* for a player in poker is a function from their information sets to a probability distribution over the possible actions in the states in that information set (it is necessary that each state in an information set has the same set of possible actions, since a player must know what actions are available to them and so if there are two states with a different set of possible actions, there must be something in the current player's private history that tells them what the possible actions are, and there is a one-to-one correspondence between private histories and information sets). For example, in Kuhn poker, Player II may have a behavioral strategy of calling Player I's initial bet with probability $1, 0.9, 0.1$ when holding K, Q, J respectively and folding otherwise, and betting with probability $1, 0, \frac{1}{2}$ respectively in response to an initial check, and otherwise checking. For a general extensive form poker game, and assuming that bets are allowed in discrete amounts (for example, integer units of chips) and there is a limit on the size of bets (and the total of all players' initial stack sizes can serve as that bound), then a strategy is a function from a very large, but finite, set of information sets to a continuous action space (probability distributions over the possible game actions). Continuous action spaces can be difficult to deal with. Fortunately, behavioral strategies for extensive form games games can be viewed as mixed strategies for the normal form game.

A pure strategy for player i in an extensive form game is a function from player i's information sets to a single action available at the information set (instead of a probability distribution over the actions for a behavioral strategy). In the normal form, there is a row or column for every possible pure strategy (so the pure strategies for the normal form are the pure strategies for the extensive form). When limited to these pure strategies, Kuhn poker is similar to chess – the difference is that since chess is a perfect information game, each information set has only one state in it, and the player's always have the same information set.

For Kuhn poker, there are 6 possible histories corresponding to points where Player I selects an action and so six information sets: 3 possible histories for the first post-deal action, and 3 possible histories for the choice Player I makes if their first action was a check and Player II's subsequent action was to bet 1 chip. So a pure strategy for Player I is a way of filling out the following table with a selected action

PI's Private History	abbrev	action
Dealt K	K	_
Dealt Q	Q	_
Dealt J	J	_
Dealt K, PI check, PII bet	K01	_
Dealt Q	Q01	_
Dealt J	K01	_

Each entry in the table can have one of two values (0 for check/fold or 1 for bet/call), so there are $2^6 = 64$ possible pure strategies for each player. We can reduce that to 27 for Player I because if Player I's selected action for the first action is to bet 1, then they will never reach the point in the game when they need to select a second action, so, for example, the strategies $K \to 1, Q \to 0, J \to 0, K01 \to 0, Q01 \to 0, J01 \to 0$ and $K \to 1, Q \to 0, J \to 0, K01 \to 1, Q01 \to 0, J01 \to 0$ are equivalent since they differ only in the selected action for $K01$, which is unreachable with a bet selected for the history K).

And just as we can view a chess match as a selection of a chess strategy followed by a playout of the game according to the strategies the players selected, we can do the same with Kuhn poker (and other forms of poker): players select one of the pure strategies, and then the hand is played out according to the selected strategies. Chess, being a combinatorial game, has no element of chance, so the result of each possible combination of Player I's strategy and Player II's strategy is a fixed value. Poker has a stochastic element, so rather than determining who wins for any pair of pure strategies, we determine the expected winnings. Since poker is zero-sum, it suffices to compute the expected net winnings for Player I for each combination of pure strategies; the matrix of those expected values is the payoff matrix for the normal form of the game.

For example, if Player I's strategy is $K \to 1, Q \to 0, J \to 0, K01 \to X, Q01 \to 0, J01 \to 0$ and Player II's strategy is $K0 \to 1, Q0 \to 1, J0 \to 1, K1 \to 1, Q1 \to 0, J1 \to 0$ (so PI only makes an initial bet with a K and folds if PII bets, while PII always bets if PI checked, and only calls with a K), then the possible outcomes of each deal are

Deal(PI/PII)	Player Actions	PI Net
K/Q	PI bets 1, PII folds	+1
K/J	PI bets 1, PII folds	+1
Q/K	PI checks, PII bets 1, PI folds	-1
Q/J	PI checks, PII bets 1, PI folds	-1
J/K	PI checks, PII bets 1, PI folds	-1
J/Q	PI checks, PII bets 1, PI folds	-1

Each of the outcomes happens with probability $\frac{1}{6}$, so the expected net winnings for Player I are $\frac{1}{6}(1 + 1 - 1 - 1 - 1 - 1) = -\frac{2}{3}$.

For chess, there will be a Nash equilibrium in pure strategies. For poker, the Nash equilibrium will be a mixed strategy. So, to find a Nash equilibrium for Kuhn poker, we need to solve the corresponding linear programming problem for the resulting 27-by-64 matrix, which standard algorithms can do in a fraction of a second on modern hardware.

One equilibrium for Player I is

000100	0.1058966081881498
000110	0.13199985251106872
00110X	0.050359255122485354
00111X	0.04547848713105562
100X00	0.22610809248049912
100X10	0.3139068437995626
101X0X	0.062214106812445685
101X1X	0.06403674603712124

There is a one-to-one correspondence between mixed and behavioral strategies, assuming perfect recall. The above mixed strategy for Player II corresponds to the behavioral strategy of making an initial bet 22.2% of the time when holding a J, never when holding a Q, and 66.6% of the time with a K; and for responding to a bet from Player II, always calling with a K, calling with probability 55.5% when holding a Q, and always folding a J. There are in fact an infinite number of Nash equilibrium strategies for Player I: pick any $0 \leq alpha \leq \frac{1}{3}$ and then make an initial bet with probabilities $\alpha, 0, 3\alpha$ when holding a J, Q, or K respectively, and call Player II's bet after an initial check with probabilities $0, \alpha + \frac{1}{3}, 1$ [**kuhn˙poker**]. Player II's only behavioral Nash equilibrium strategy is responding to an initial call by betting with probabilities $\frac{1}{3}, 0, 1$ for J, Q, K respectively, and responding to an initial bet by calling with probability $0, \frac{1}{3}, 1$ respectively. The value for Player I of a Nash equilibrium strategy profile is $-\frac{1}{18}$ (which makes sense since Player I should be at a disadvantage picking

their initial action with no information beyond their own card, while Player II always has information from Player I's initial action before selecting their first action).

However, for Kuhn poker generalized to decks of size n, the number of columns in the matrix is 2^{2n}, so this approach is impractical beyond very small deck sizes. Koller, Megiddo, and von Stengel introduce a third form of imperfect information games, the *sequence form*, using which a Nash equilibrium can be computed more efficiently, also using linear programming [KM92] [KMv96] [von96]. For the sequence form, the linear program is of size polynomial in the size of the game tree, rather than exponential. Even so, the linear program for the sequence form is infeasible to solve for commonly played forms of poker, so other techniques centered on the extensive form are used to get very good approximations of the Nash equilibrium. See, for example Neller and X's introduction to Counterfactual Regret Minimization [**cfr˙tutorial**], and then Facebook's work combining that with neural networks [BS17] [BS19] [Bro+20] or UofA's work [Mor+17].

Chapter 5

Tree Search

5.1 Minimax

The MINIMAX algorithm of Section 1.2.1 is infeasible if the state space complexity of the game is too high. Go, for example, has an average branching factor of about 250 and an average depth of about 150, for a game tree complexity in the neighborhood of $10^3 60$. If we has a system that could examine a quadrillion ($10^1 5$) paths through the tree per second (which is still very fast compared to systems available as of this writing), it would still take $10^3 37$ years to examine the entire tree. The state space complexity is close to $10^1 72$, so using dynamic programming does not make the problem feasible, even if we had enough memory to store the value of each position.

To employ MINIMAX on a game with a large state space requires us to limit the depth to which we search the tree by keeping track of how deep the current recursive call is, and adding a base case for when we are at the limiting depth. We retain the base case for terminal positions, but this new base case will be reached for nonterminal positions, for which we cannot simply use the rules of the game to determine who won and assign an appropriate value. Instead, we must make some guess at which player has a winning strategy and our confidence in that guess, by using +1 to mean that we are confident that Player I will win, -1 to indicate extreme confidence that Player II will win, and values in between to indicate confidence. The function that outputs these confidence values given a position in the game is called a *heuristic*. It is common to scale the result of the heuristic to a larger range than $[-1, 1]$, for example $[-100, 100]$. Regardless of scaling, Player I should want to move to positions with higher heuristic values, and Player II should want to move to positions with lower values, so the code for MINIMAX (and NEGAMAX) is the same, except for the addition of the new base case and logic to keep track of the current depth of the search, or, as we present it here, of the depth remaining (so the algorithm counts down from d depth at the root to 0 when it is as deep in the tree as it is allowed to go).

function MINIMAX-WITH-HEURISTIC(G, h, d) ▷ Precondition: G nonterminal, $d \geq 1$
  if G is terminal **then**
    $value \leftarrow v_I(G)$ as determined by the rules of the game
    return $(value, NIL)$
  else if $d = 0$ **then**
    return $(h(G), NIL)$
  else
    Enumerate the options of G, $G_0, \ldots, G_{k-1}$
    $(v_0, m_0) \ldots, (v_{k-1}, m_{k-1}) \leftarrow Minimax(G_0, h, d-1), \ldots, Minimax(G_{k-1}, h, d-1)$ ▷ the m_j will be ignored here
    if $next(G) = I$ **then** ▷ Player I is next to move
      $m \leftarrow \arg\max_{0 \leq j < k} v_j$

```
        else
            m ← arg min_{0≤j<k} v_j
        end if
        return (v_m, m)
    end if
end function
```

If the heuristic function h is more accurate the deeper in the tree we search, then the new Minimax will give better results the higher the depth bound. Of course, the higher the depth bound, the more of the game tree the algorithm needs to search, and the longer it will take. There is then a balance between the amount of time allowed to Minimax and the quality of the moves it chooses.

5.1.1 Iterative Deepening

It is common for the branching factor of a game to change as the game proceeds. For example, Othello has an average branching factor of 10, but there are only 4 moves possible at the beginning of the game (which are all equivalent due to symmetry), and, close to the end of the game when only 4 squares are unfilled, 4 moves possible again (at most, since playing in some of those 4 squares may not lead to a capture and so may not be legal).

So, for a given constant depth bound d, the size of the game tree searched by Minimax may vary during course of the game, and so the time to determine a good move may vary considerably. Choosing a d that strikes the right balance between time and quality during the beginning phase of a game may then result in the search taking too long during the middle phase. It may be possible to heuristically choose d depending on the current phase of the game, or we can automatically determine the depth to search using *iterative deepening*. Iterative deepening limits the depth of search using a time budget instead of a struct depth budget. It starts with a search to depth 1, and, if that finishes within the time limit, restarts the search to a depth of 2. If there is still time left after *that* search, the search is restarted with a depth of 3, and so on. A timer can interrupt the current search when time expires. The result of the last completed search is the one used.

```
function Minimax-Iterative-Deepening(G, h, t)
    d ← 1
    while less than time t has elapsed do
        (v, m) ← Minimax(G, h, d)        ▷ each completed call overwrites v and m
        d ← d + 1
    end while
    return (v, m)
end function
```

The iterative-deepening version of Minimax can be made *anytime* – we can remove the time bound and instead interrupt the while loop at any point and use the result of the last completed search. This can be useful when we want to return a move on demand, as when the user gets impatient, or, with some additional modifications, to use the human player's thinking time to search for the computer's next move, and then return the best move found by the search as soon as the human player has made their move.

5.1.2 Transposition Tables

We may still have the overlapping subproblem that we had with the exhaustive-search version of Minimax. The table-based dynamic programming solution from Section 1.2.3 is not likely to be a practical solution to that problem when using a depth-bound because that approach requires iterating through the subproblems from terminal positions to the initial position. It may be difficult to generate the positions depth d from the initial position without already finding the intermediate positions. The top-down memoized solution is then

a better choice. In this context, the memo used in memoization is called a *transposition table* (because the multiple paths to the same position often arise from switching the order, or transposing, moves in a sequence). The deeper the search is, the more positions will be in the search tree, and the larger the transposition table will be. To avoid running out of memory, the size of the transposition table can be capped. This requires a policy for determining what happens when the transposition table is full and a new entry needs to be stored Since transposition tables are often implemented as hash tables, the policy can simply determine what to happen when two positions collide at the same index in the hash table (so avoiding strategies like linear probing or chaining to resolve collisions). Collision resolution polices can be as simple as always replacing the old entry with the new entry, or can be based on how deep in the search the positions were encountered, or other properties of the positions and/or the state of the search.

The entries in the transposition table include the position, the value of that positions, and the depth to which the subtree rooted at the position was searched – when trying to search a position to depth 10, if there is a value for that position stored in the table with the result from a search to depth 8, one would not want to use the result of that shallower search. Using the result of a deeper search is possible and most likely desirable (although see the note about search instability below).

5.2 Alpha-Beta Pruning

Consider the operation of MINIMAX on the following tree, with the heuristic value of positions given at the leaves.

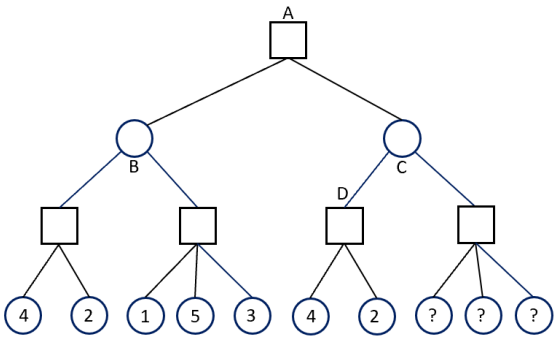

Figure 5.1: Example: Minimax tree in which the values of C's right grandchildren will not affect the value of A.

Assume that the computation of the maximum in MINIMAX is implemented in the straightforward way: we keep track of a running maximum-so-far, and update that as we iterate through all the child positions in order. Node A is a max node, so when the call to node B from node A returns 5, that running maximum-so-far is updated to 5 *and can only increase* as we consider the results from the subsequent children – the value of A is at least 5. Similarly, when computing the value of node C and getting 3 back from its first child D, the minimum-so-far will be set to 3 and can only decrease – the value of D is at most 3. Now consider the rest of the descendants of C. If they are all less than 3, then the running minimum will decrease from 3, and the value of C is less than 3. But then when C returns that value to A, the value is less than A's current maximum child, and so there is no update. If, on the other hand, the rest of the descendants of C are all very high values, the children of C after D will return values greater than 3, and C's minimum will not be updated as those values are returned. In that case, the value of C is 3, and again, when C's value is returned to A, the value is less than 5 and so A does not update its maximum. *The values of those other descendants of C do not have any effect on the value of A.* If we can detect this and other similar situations, then we can

skip searching subtrees that do not affect the values further up the tree (so here, we can skip searching the subtrees rooted at C's second and subsequent children). Skipping irrelevant subtrees can save enough time to allow deeper, and so more informed, searches.

We can detect these situations by passing bounds on relevant values through the recursive calls. The 5 in the above example was a lower bound: once the value of a child of A is known to be less than 5, that child is irrelevant. We could have a similar upper bound on the relevant values of children for min nodes. The ALPHA-BETA pruning algorithm keeps track of those lower bounds (α) and upper bounds (β), and stops searching the children of a node once the node's value is known to be outside that range. For a max node, that's when one of its children returns a value greater than β, because at that point, we know the maximum value of all the children is at least β and so it outside the range (α, β). For a min node, the point when we stop searching its children is when one of them returns a value less than α.

The (α, β) window passed to a node X defines the range of values where the precise value within that range will affect the value of ancestors of X. Values outside that range will not affect the value of the ancestors (or, at least, won't affect whether the values of the ancestors are within the (α, β) windows passed to them). Precisely, the postcondition of $ALPHA - BETA(()G, h, d, \alpha, \beta)$ is that the value returned is

1. the same as $MINIMAX(G, h, d)$ if either a) G is terminal, b) $d = 0$, or c) $\alpha < MINIMAX(G, h, d) < \beta$;
2. something in the range $(MINIMAX(G, h, d), \alpha)$ if $MINIMAX(G, h, d) \leq \alpha$
3. something in the range $(\beta, MINIMAX(G, h, d))$ if $MINIMAX(G, h, d) \geq \beta$.

In other words, ALPHA-BETA returns the correct MINIMAX value if that value is inside the range (α, β) (or the position examined is terminal or at the depth limit), and otherwise the value returned by ALPHA-BETA will be on the same side of the (α, β) range as the MINIMAX value and is an upper bound on the exact MINIMAX value if on the smaller side and is a lower bound if on the upper side. Note that $ALPHA - BETA(G, h, d, -\infty, \infty)$ will always return the same value as $MINIMAX(G, h, d)$ because clause c) of the first postcondition holds.

To compute the value of a node G (without simultaneously computing the best move; we leave it as an exercise to keep track of the best move), we have the following algorithm.

function ALPHA-BETA(G, h, d, α, β)
  if G is terminal **then**
    return $v_I(G)$ as determined by the rules of the game
  else if $d = 0$ **then**
    return $h(G)$
  else
    Enumerate the options of G, $G_0, \ldots, G_{k-1}$
    if $next(G) = I$ **then** ▷ Player I is next to move
      $j \leftarrow 0, a \leftarrow -\infty$
      while $j < k$ and $a < \beta$ **do**
        $a \leftarrow \max(a, ALPHA - BETA(G_j, h, d-1, \alpha, \beta))$
        $\alpha \leftarrow \max(\alpha, a)$
        $j \leftarrow j + 1$
      end while
      return a
    else
      $j \leftarrow 0, b \leftarrow \infty$
      while $j < k$ and $b > \alpha$ **do**
        $b \leftarrow \min(a, ALPHA - BETA(G_j, h, d-1, \alpha, \beta)$

```
            β ← min(β, b)
            j ← j + 1
        end while
        return b
    end if
  end if
end function
```

Note that it is correct to modify ALPHA-BETA to return α whenever the general case would otherwise return something strictly less than α, and to return β when the value returned would otherwise be strictly greater than β. This is referred to as *fail-hard* as opposed to the *fail-soft* version here (and fail-hard can also be applied to the base cases, although that changes the postconditions).

When using a transposition table, it is helpful to store values in the table even when they are not exact, although they must be marked as bounds in that case. If, when searching to depth d, values in the transposition table from depths greater than d are used, the postconditions may not be strictly met, since effectively a larger portion of the tree is being searched. This can give rise to search instabilities when used with the fail-soft version of ALPHA-BETA – situations where searches with different (α, β) windows give results that are inconsistent with each other, for example one search on a position G with one window resulting in an upper bound of 3 on G's value, and another resulting in a lower bound of 5 [].

5.2.1 Scout

When the ALPHA-BETA is processing a max node and gets a value of a back from its first child, the relevant information about the next child is 1) whether the value of that next child is greater than α, because then the next child becomes the running best child so far, and we know the value of the node is at least α), and 2) if so, what its exact value is, because that becomes the value of the running maximum so far (unless it is even greater than (or equal to) β, in which case all we need to know is that the value is at least β); if the value of the next child is less than or equal to a then nothing changes. The ALPHA-BETA algorithm gets the answer to those questions in a single call to ALPHA-BETA on the next child. If the result is less than α then the answer to the first question was no; if the result is greater than α, then it knows both that the answer is yes, and what the value actually is. But it is possible to separate those two questions; doing so results in the SCOUT algorithm. SCOUT determines whether the next child's value is greater than α by making a recursive call with window $(\alpha, \alpha + 1)$, the *null window*, so called because when the values of the nodes are integers, there are no values between the endpoints of the window. The postconditions are the same, so if the result comes back less than or equal to α, that result is an upper bound on the value of the next child, and so we know that the next child's value is no greater than α. If the result comes back greater than α, then we know that the value of the next child is greater than α, but all we know is that the result is a lower bound on the value of the next child. In order to get the exact value of the next child (or a lower bound greater than or equal to β), SCOUT must make an additional call with the full window.

For any child of G not better than the best so far, the second recursive call will not be made, and the first call with the null $(\alpha, \alpha + 1)$ window will likely result in more pruning than a call made with the full (α, β) window as made by ALPHA-BETA. However, for a child that is better than the best so far, there will be some additional work performed because of the additional search. If we end up in the former situation more often than the latter, then the tradeoff is worth it, and SCOUT will visit fewer nodes than ALPHA-BETA, and will return the same value. SCOUT can tilt the scales in its favor by ordering the children of the node to search in roughly decreasing order of quality. The order needn't be exact, but the better it is, the bigger an advantage SCOUT will have over ALPHA-BETA. The rough ordering can come from the heuristic that is normally applied to the leaves, another, possibly faster heuristic, or results from previous searches to lower depths, as would be available in later iterations using iterative deepening, or if the results of the search on

one turn were kept for later turns.

function SCOUT(G, h, d, α, β)
  if G is terminal **then**
    return $v_I(G)$ as determined by the rules of the game
  else if $d = 0$ **then**
    return $h(G)$
  else
    Enumerate the options of G, $G_0, \ldots, G_{k-1}$ in approximate order of goodness
    if $next(G) = I$ **then** ▷ Player I is next to move
      $j \leftarrow 1, a \leftarrow SCOUT(G_0, h, d-1, \alpha, \beta)$ ▷ full window for 1st child
      $\alpha \leftarrow \max(\alpha, a)$
      while $j < k$ and $a < \beta$ **do**
        $s \leftarrow ALPHA - BETA(G_j, h, d-1, \alpha, \alpha+1)$ ▷ null window
        if $s > \alpha$ and $s < \beta$ **then** ▷ check whether previous search cut off high
          $s \leftarrow SCOUT(G_j, h, d-1, s, \beta)$ ▷ full window
        end if
        $a \leftarrow \max(a, s), \alpha \leftarrow \max(\alpha, a)$ ▷ update best child and low end of window
        $j \leftarrow j + 1$
      end while
      return a
    else
      $j \leftarrow 1, b \leftarrow SCOUT(G_0, h, d-1, \alpha, \beta)$
      $\beta \leftarrow \max(\beta, b)$
      while $j < k$ and $b > \alpha$ **do**
        $s \leftarrow ALPHA - BETA(G_j, h, d-1, \beta, \beta-1)$
        if $s < \beta$ and $s > \alpha$ **then** ▷ check whether previous search cut off low
          $s \leftarrow SCOUT(G_j, h, d-1, \alpha, s)$
        end if
        $b \leftarrow \min(b, s), \beta \leftarrow \min(\beta, b)$
        $j \leftarrow j + 1$
      end while
      return b
    end if
  end if
end function

Note that the second recursive call on a child already has a null window and so won't benefit from any additional narrowing of the window, so can use regular ALPHA-BETA. And at the point of the second call, we know from the first call that the value of the child is at least s (when G is a max node), so we can use s instead of α as the lower endpoint of the search window. (And, with some slight differences, SCOUT is also known as PRINCIPAL-VARIATION-SEARCH. The version that combines the Player I case with the Player II case as NEGAMAX does is called NEGASCOUT).

5.2.2 MTD-f

SCOUT can make use of the results of previous, shallower searches in order to determine an approximate move ordering. But there are other ways to use those previous results. The MTD-F algorithm, when searching position G to depth G, will first check whether a previous shallower search has resturned a value for G. If it has, then it will first determine whether that value is correct for the new, deeper search. It can determine

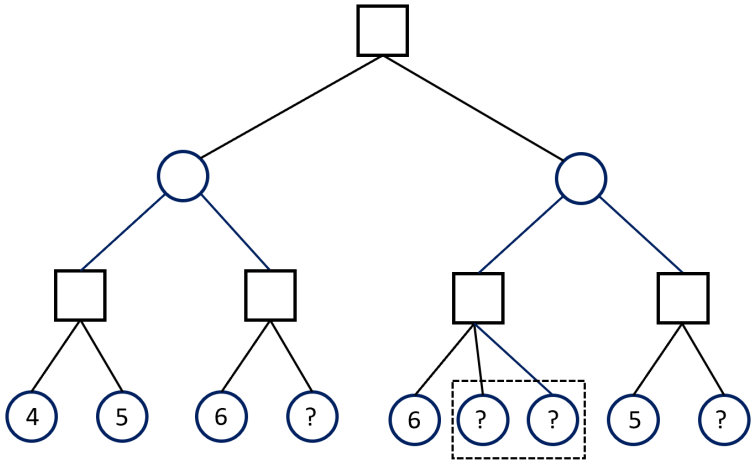

Figure 5.2: Example: The nodes marked with ? are not searched by SCOUT. The two nodes inside the dashed box are not pruned by ALPHA-BETA (the other three nodes are pruned by both algorithms).

whether the guess is correct with two ALPHA-BETA null-window searches. If it is incorrect, it will keep moving the window until it finds the correct value. The following pseudocode assumes, like SCOUT, that the heuristic is integer-valued.

```
function MTD-F(G, h, d, f)                          ▷ f is first guess at value of G
    if G is terminal then
        return v_I(G) as determined by the rules of the game
    else if d = 0 then
        return h(G)
    else
        lb ← −∞, ub ← ∞
        g ← f                                       ▷ current guess at value of G
        while lb < ub do
            β ← max(lb, g)
            g ← ALPHA − BETA(G, h, d, β − 1, β)
            if g < β then
                ub ← g                              ▷ ALPHA-BETA returned an upper bound
            else
                lb ← g                              ▷ ALPHA-BETA returned a lower bound
            end if
        end while
        return g
    end if
end function
```

5.3 Multi-player Games

5.3.1 MAX^N

All of the algorithms for two-player zero-sum games take advantage of the fact that maximizing the result for Player II is the same as minimizing the result for Player I. The same simplification can't be made for games with three or more players, since a gain for Player III could come at the expense of only Player I, only Player II, or both of them together. If we assume that all players' goals are to maximize their utility, and they are indifferent to how the remaining utility is distributed to the other players, then we need the evaluation of terminal positions and the heuristic to return the values for all players, and when it is a player's turn, we need to find the move that maximizes that player's utility. This leads to MAX^n, the generalization of MINIMAX to n-player games.

function MAX-N(G, h, d, n, i) ▷ n is number of players, i is index of current player
  if G is terminal **then**
    $value \leftarrow v(G)$ as determined by the rules of the game ▷ an n-tuple of utilties
    return $(value, NIL)$
  else if $d = 0$ **then**
    return $h(G)$ ▷ again, an n-tuple of utilities
  else
    Enumerate the options of G, $G_0, \ldots, G_{k-1}$
    $v_0 \ldots, v_{k-1} \leftarrow MAX{-}N(G_0, h, d{-}1, n, (i{+}1) \mod n), \ldots, MAX{-}N(G_{k-1}, h, d{-}1, n, (i{+}1) \mod n)$
    $m \leftarrow v_0$
    $j \leftarrow 1$
    while $j < k$ **do**
      if $v_j[i] > m[i]$ **then** ▷ compare utilities for player i
        $m \leftarrow v_j$
      end if
    end while
    return m
  end if
end function

Opportunities to prune are very limited in these situations. In general, the number of nodes pruned goes down dramatically as the number of players increases. The minimum number of nodes examined in a tree of depth d and branching factor b with n players is $b^{(1-\frac{1}{n})d}$ [**somepaper**].

5.3.2 Paranoid

"Kingmaking" can be an issue in multi-player games. Kingmaking refers to the ability of one player who has little chance of winning to influence the chances of other players to win. The influence can be intentional, such as favoring one player over another because of personal relationships, or unintentional, simply because the choice of actions with similar utility for the losing player may have very different effects on the utility for the other players. These can also be issues when all players still have a reasonable chance of winning. For that reason, the MAX^{n}algorithm can be too optimistic.

As an alternative, each player can play as if the other players are all colluding against them. The utility for the one player is then as usual, and the other players are assumed to play to minimize the utility for that player, regardless on the effects on their own individual utility. In effect, this transforms the game into a two-player game in which one player is the one colluded against, and the other player is the team colluding

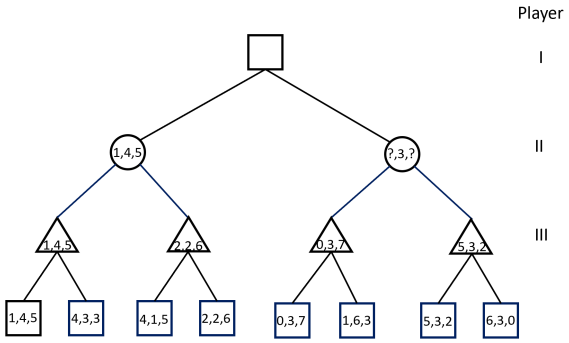

Figure 5.3: The value for Player I at the root depends on the tie-breaker for Player II at the second child of the root. If Player II chooses (0,3,7) at the second child, then the value at the root is 1. If Player II chooses $(5, 3, 2)$, then the value at the root is 5.

against them. The algorithm to decide moves for a player is then MINIMAX, modified to account for the second player now getting several turns in a row, making moves for each of the colluding players. This is called the PARANOID algorithm [SK00].

function PARANOID(G, h, d, i) ▷ i is the player colluded against
  if G is terminal **then**
    $value \leftarrow v(G)$ as determined by the rules of the game
    return $(value[i], NIL)$ ▷ return the value for player i
  else if $d = 0$ **then**
    return $(h(G)[i], NIL)$
  else
    Enumerate the options of G, $G_0, \ldots, G_{k-1}$
    $(v_0, m_0) \ldots, (v_{k-1}, m_{k-1}) \leftarrow PARANOID(G_0, h, d-1, p), \ldots, PARANOID(G_{k-1}, h, d-1, p)$
    if $next(G) = i$ **then**
      $m \leftarrow \arg\max_{0 \le j < k} v_j[i]$ ▷ solo player is next to move
    else
      $m \leftarrow \arg\min_{0 \le j < k} v_j[i]$ ▷ a colluding player is next to move
    end if
    return (v_m, m)
  end if
end function

In a multi-player game between PARANOID agents, each would assume the other players are colluding against it. Player I would pass $i = 1$ to determine its best move from a given position, Player II would pass $i = 2$, and so on.

As a side-effect of turning the game into a two-player game, $PARANOID$ can be implemented with ALPHA-BETA-style pruning to speed it up. Although PARANOID is very pessimistic, the additional depth of search allowed because of the pruning can result in a better player than the overly optimistic MAX^n[**sturtevant**].

5.3.3 Best Reply Search

BEST-REPLY-SEARCH is a compromise between the optimism of MAX^n's assumption that all players care only about maximizing their utility and the pessimism of PARANOID's assumption that all players are colluding against it. BEST-REPLY-SEARCH again treats the rest of the players as a team, but assumes only the

one player with the most damaging move to it (the player with the best response) will make that move, and the rest of the opponents will pass, even if that is not a legal move in the game. This compromise works well for some games, including some where a heuristic that doesn't consider the other players' positions is a useful one [SW11]. Because BEST-REPLY-SEARCH also reduces a multi-player game to a two-player game, it can be implemented with pruning. The algorithm without pruning is presented below. The initial call to determine a move for the player to move i at position G would be $BEST - REPLY - SEARCH(G, h, d, i, true)$.

function BEST-REPLY-SEARCH(G, h, d, i, f) ▷ i is the player at the root, f is a flag set to $true$ to maximize at G
  if G is terminal **then**
    $value \leftarrow v(G)$ as determined by the rules of the game
    return $(value[i], NIL)$ ▷ return the value for player i
  else if $d = 0$ **then**
    return $(h(G)[i], NIL)$
  else if $f = true$ **then** ▷ determine if player i or one of the other players is next
    Enumerate the options of G, $G_0, \ldots, G_{k-1}$
    $(v_0, m_0) \ldots, (v_{k-1}, m_{k-1}) \leftarrow BEST - REPLY - SEARCH(G_0, h, d-1, i, false), \ldots, BEST - REPLY$
    $m \leftarrow \arg\max_{0 \le j < k} v_j[i]$
    return $(v_{p,m}, m)$
  else
    for each player $p \neq i$ **do**
      Enumerate the options of G for player p, $G_{p,0}, \ldots, G_{p,k_p-1}$ ▷ pretending that p is the next player to move at G
      $(v_{p,0}, m_{p,0}) \ldots, (v_{p,k-1}, m_{p,k_p-1}) \leftarrow BEST - REPLY - SEARCH(G_{p,0}, h, d-1, i, false), \ldots, BEST$
    end for
    $m, p' \leftarrow \arg\min_{p \neq i, 0 \le j < k_p} v_{p,j}[i]$
    return $(v_{p',m}, NIL)$
  end if
end function

5.4 Aheuristic Search

The MINIMAX and ALPHA-BETA family of algorithms all require a heuristic to evaluate the nonterminal positions at their depth limit. Good heuristics can be difficult to discover – the best heuristics for chess incorporate hundreds of features. Stochastic methods can be used in place of heuristics, taking advantage of the property that moves that are good when followed by random play are generally good when followed by expert play too. The degree to which that property holds for a particular game will affect how good these stochastic, aheuristic methods perform for a game.

The FLAT-MONTE-CARLO is the simplest of this family of algorithms FLAT-MONTE-CARLO evaluates each possible move by following it with several samples of random play until a terminal state, keeping track of the mean reward (wins) obtained for the player at the original state. The move that is chosen is then the one with the highest average.

function FLAT-MONTE-CARLO(G) ▷ Precondition: G is nonterminal
  $i \leftarrow next(G)$
  Enumerate the options of G, $G_0, \ldots, G_{k-1}$
  $r_j \leftarrow 0$ for $j = 0, \ldots, k-1$ ▷ total reward for each option
  for $i \in \{0, \ldots, n-1\}$ **do** ▷ or set a time limit
    for $j \in \{0, \ldots, k-1\}$ **do** ▷ for each move

```
            simulate random play from G_j, ending in terminal position G'
            r_j ← r_j + v_i(G')                          ▷ accumulate total reward for move j
        end for
    end for
    m ← arg min_{0≤j<k} r_j/n                            ▷ find move with highest average reward
    return r_m/n, m
end function
```

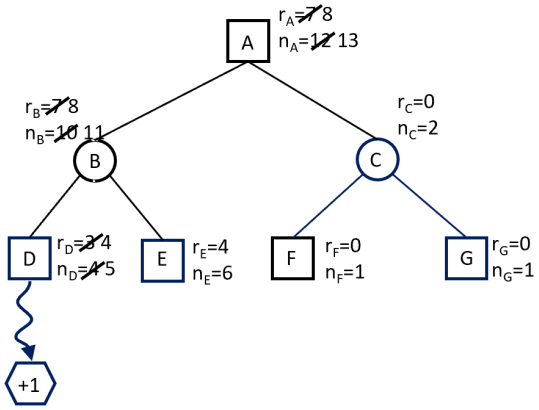

Figure 5.4: Example: After 10 iterations of the outer loop, the total reward for each child of the root are as shown. Supposing random play starting with each child results in the values shown, the totals are updated as given.

5.4.1 Multi-armed Bandit

The FLAT-MONTE-CARLO algorithm as presented in Section 5.4 always samples each child of G the same number of times. That may not be the most efficient use of time. For example, if, after 100 iterations, child G_0 has an average reward of 0.7 with little variation, and child G_1 has an average reward of -0.3 with little variation, it is probably safe to conclude that G_0 better than G_1. If there is another child G_2 with average reward 0.68, then it is probably more worthwhile to spend additional samples on G_0 and G_2 to determine which is truly better than it is to spend additional samples on G_1.

The problem of efficiently allocating samples to different options with unknown reward distributions is called the *multi-armed bandit* problem. The name refers to a slot machine (referred to colloquially as a "one-armed bandit" because their payoff probabilities are set to separate players from their money) with multiple arms, each with a different payoff distribution, or, more realistically, a bank of slot machines, each with different payoffs. This reflects actual practice in casinos, where the most visible machines tend to have better payoffs in order to entice people to play. If a player were to play for a long period of time, they would want to play the machine with the highest expected payoff per play. But (when casino design principles don't provide any clues), in order to determine the best machine, they player will have to try all the different machines (or arms).

The measure of efficiency for the multi-armed bandit problem is *regret* – the difference between what a player won vs. the expected value they would have gotten if they had played the arm with the highest average payoff every time.

Formally,suppose there are n arms with mean payoffs $\mu_0, \ldots \mu_{n-1}$. Let μ^* be the highest mean payoff:

$$\mu^* = \max_{i=0,\ldots,n-1} \mu_i$$

(and assume for simplicity that the highest mean payoff is unique). Suppose a player has played a total of T times, and let $i_0, \ldots, i_{T-1}$ be the indices of the arms chosen at each of those T times, with rewards obtained $r_0, \ldots, r_{T-1}$. Then the total regret over that time is the difference between the expected reward if the player had chosen the best arm each time and the sum of the rewards actually obtained.

$$\rho_T = T \cdot \mu^* - \sum_{j=0}^{T-1} r_j.$$

It is impossible to devise a policy that obtains zero total regret because without playing each arm at least once, it is impossible to determine which is the optimal arm. However, it is possible to devise a policy that, no matter what the unknown payoff distributions of the individual arms is, is guaranteed to obtain zero *average* regret as the number of plays increases towards infinity. Formally, what we want is a policy for choosing a arm at each individual step so that

$$P\left(\lim_{T\to\infty} \frac{\rho_T}{T} = 0\right) = 1 \tag{5.1}$$

But first, we verify that some simple policies do not guarantee zero average regret in the limit. Consider the *greedy* policy. The greedy policy plays each arm once and then forever plays the arm that gave the highest reward in the first round. While it is certainly possible and perhaps even likely that the best arm gives the highest payoff in the first round, there are plenty of distributions where that is not guaranteed. Consider, for example, the following setup with three arms, each with the given probability distributions over possible payoffs.

Arm 0		Arm 1		Arm 2	
Prob	Payoff	Prob	Payoff	Prob	Payoff
$\frac{1}{2}$	5	$\frac{1}{3}$	4	$\frac{1}{4}$	20
$\frac{1}{2}$	1	$\frac{1}{3}$	3	$\frac{3}{4}$	0
		$\frac{1}{3}$	2		
$\mu_0 = 2$		$\mu_1 = 3$		$\mu_2 = 5$	

For the greedy policy, the limit of the average regret will be the expected regret from the arm m that is eventually chosen, because in the limit, the first three plays do not matter, and the average regret is then

$$T \cdot \mu^* - \sum_{j=0}^{T-1} r_j,$$

and the limit of the average regret is then

$$\begin{aligned}
&\lim_{T\to\infty} \frac{T \cdot \mu^* - \sum_{j=0}^{T-1} r_j}{T} \\
= &\lim_{T\to\infty} \mu^* - \frac{\sum_{j=0}^{T-1} r_j}{T} \\
= &\lim_{T\to\infty} \mu^* - \lim_{T\to\infty} \frac{\sum_{j=0}^{T-1} r_j}{T} \\
= &\mu^* - \lim_{T\to\infty} \frac{\sum_{j=0}^{T-1} r_j}{T},
\end{aligned}$$

where the r_j are all random variables drawn from the same random process – playing arm m. The last term is then, by definition, the expected value of playing that arm, μ_m, so the limit of the average regret is $\mu^* - \mu_m$. When m is not the best arm, we have $\mu^* > \mu_m$ and so the limit of the average regret $\mu^* - \mu_m > 0$. In the given example, the best arm is chosen exactly when it hits its highest payoff during the first exploratory round, which happens with probability $\frac{1}{4}$. So the limit of the average regret isn't guaranteed to go to 0; it only goes to 0 one quarter of the time.

Even with multiple rounds before settling on playing the arm with the highest observed reward forever, the greedy policy will not guarantee that the average regret tends to zero, because after k rounds, there is always some non-zero probability that Arm 2 does not have the highest observed reward. In fact, it is necessary to play each arm infinitely often in order to know which has the highest expected payoff – it is always possible that one of the arms has a very rare but astronomical payoff. The *ϵ-greedy* policy balances exploiting the information obtained by the previous plays with always exploring all the arms. After playing each arm once, for the rest of the steps it will choose an arm uniformly randonly With probability ϵ; the rest of the time it plays the arm with the highest observed reward.

Under the ϵ-greedy policy, the expected regret will tend towards the weighted average of the regret from each arm, where the weights are the probabilities that each arm is played. As each arm is played infinitely often, the probability of the best arm being chosen when not exploring will tend to 1. However, each other arm still has an chance of being chosen on the exploration steps, so the probability that the other arms are chosen will tend towards $\frac{\epsilon}{n}$. The average regret will then approach

$$\frac{\epsilon}{n} \cdot \sum_{i=1}^{n} (\mu^* - \mu_i).$$

This is a guarantee, but since each term in the sum is positive when i is not the best arm, the guarantee is not that the average regret tends to 0.

In order to have guaranteed average zero regret in the limit, we need to balance exploration (so we don't miss out on an arm that is the best one because of rare jackpots) and exploitation (so that once the greedy choice is actually the best arm, we play that with increasing probability). A policy that achieves the right balance of exploration and exploitation is the Upper Confidence Bound (or *UCB*) policy. The UCB policy plays each arm once, and on subsequent steps chooses the arm that maximizes

$$\bar{r}_i + \sqrt{\frac{2 \ln T}{n_i}}$$

where $\bar{r}_i$ is the observed average reward from arm i, T is the number of steps completed so far (over all the arms), and n_i is the number of times arm i has been played. The first term exploits the information obtained so far. The second term encourages exploration of arms that haven't been played very often and so for which the player has less confidence in the current observed average reward as an estimation of the actual expected reward. For the mathematics showing that this does in fact guarantee expected zero average, regret we refer readers to the original paper describing UCB [ACF02].

FLAT-UCB

Incorporating UCB into FLAT-MONTE-CARLO allows for more efficient use of samples. The FLAT-UCB algorithm incorporates stochastic play in place of a heursitic, UCB to use samples more efficiently, and the lookahead of the MINIMAX family of algorithms. (We present FLAT-UCB as an enhancement of FLAT-MONTE-CARLO, although its inventors Coquelin and Munos actually developed it as an alternative to the MCTS, the third algorithm in the sequence presented here [CM07]). FLAT-UCB builds the game tree rooted at a position G to a given depth d. Then it repeatedly chooses a path through the tree from the root to a

leaf, using the UCB formula to determine which child to select at each level of the tree. Once it gets to a leaf, it simulated random play to a terminal position and propagates the reward back up the tree. After enough iterations (or when the allotted time has expired), FLAT-UCB chooses to move from G to the child of G with the highest average observed reward.

function FLAT-UCB(G, d, t) ▷ Precondition: G is nonterminal
  $i \leftarrow next(G)$
  Build the game tree T rooted at G to depth d
  for Node $j \in T$ **do**
    $r_j \leftarrow 0, n_j \leftarrow 0$ ▷ initialize total reward and visit count for each node
    $pos(j) \leftarrow$ position corresponding to node j
  end for
  for $k \in \{0, \ldots, t-1\}$ **do** ▷ or set a time limit
    $curr \leftarrow root(T)$
    $path \leftarrow curr$ ▷ list of nodes visited on iteration k
    while $curr$ is not a leaf **do**
      if $next(pos(curr)) = i$ **then** ▷ same player at $curr$ as at G
        $curr \leftarrow \arg\max_{c \text{ is a child of } curr} \frac{r_c}{n_c} + \sqrt{\frac{2 \ln n_{curr}}{n_c}}$
      else
        $curr \leftarrow \arg\max_{c \text{ is a child of } curr} \frac{-r_c}{n_c} + \sqrt{\frac{2 \ln n_{curr}}{n_c}}$
      end if
      append $curr$ to $path$
    end while
    if $pos(curr)$ is terminal **then**
      $r \leftarrow v_i(pos(curr))$ ▷ get actual reward for player moving at original position G
    else
      simulate random play from $pos(curr)$, ending in terminal position G'
      $r \leftarrow v_i(G')$
    end if
    for $k \in path$ **do** ▷ backpropagate reward along path from leaf to root
      $r_k \leftarrow r_k + r, n_k \leftarrow n_k + 1$
    end for
  end for
  $c^* \leftarrow \arg\max_{c \text{ is a child of } root(T)} \frac{r_c}{n_c}$ ▷ find child of root with highest average reward
  return the move that changes the position from G to $pos(c^*)$
end function

5.4.2 Monte Carlo Tree Search

The MINIMAX and ALPHA-BETA family of algorithms and the stochastic methods from Section 5.4 all have the nonterminal leaves of their search trees at the same level. However, if a move from the root can be easily seen to be particularly poor (for example, sacrificing a queen in chess for little gain), it is not worth exploring the corresponding subtree as extensively as moves that can be easily seen to confer some advantage (for example, capturing a queen with little sacrifice). Monte Carlo Tree Search (MCTS), invented by Chaslot *et al* [Cha+08], allows the combination of the stochastic estimate of values of positions used by FLAT-MONTE-CARLO and FLAT-UCB with asymmetrical search of the game tree, searching the subtrees corresponding to more promising moves more thoroughly than subtrees for less promising moves.

To achieve asymmetrical growth, MCTS starts with a tree containing only a root; the root corresponds to

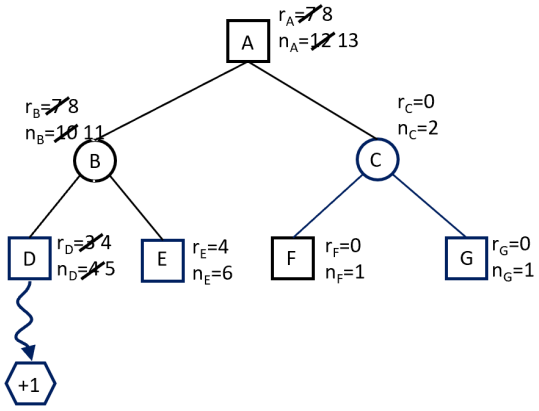

Figure 5.5: Example: Given the previous statistics, FLAT-UCB chooses the path ABD. If the random playout ends in a win for Player I, the statistics along that path are updated accordingly.

the position G for which we need to determine the best move. On each iteration, we move down the tree level by level, until we get to a node corresponding to a terminal position or an *expandable* node – a node at which there is a legal move that does not yet have the resulting position added as a child. The exact method by which we choose which child to visit as we descent the tree is not specified by MCTS; the method is referred to as the *tree policy*. If the leaf the tree policy ends at is expandable, then the missing child is then added (if there is more than one missing child, the child to add is determined arbitrarily), and its value is estimated. Again, the method by which the value is estimated – the *default policy* – is not specified by MCTS.

function MCTS(G, t) ▷ Precondition: G is nonterminal
  $i \leftarrow next(G)$
  Build a game tree T containing only a root, with $pos(root) = G$
  $r_{pos(root)} \leftarrow 0, n_{pos(root)} \leftarrow 0$ ▷ initialize total reward and visit count
  for $k \in \{0, \ldots, t-1\}$ **do** ▷ or until out of time
    $curr \leftarrow root(T)$
    $path \leftarrow curr$ ▷ list of nodes visited on iteration k
    while $curr$ is nonterminal and $|\{c \mid c \text{ is a child of } curr\}| = |\{G' \mid G' \text{ is an option of } G\}$ **do**
      $curr \leftarrow$ child of $curr$ chosen according to the tree policy
      append $curr$ to $path$
    end while
    if $pos(curr)$ is terminal **then**
      $r \leftarrow v_i(pos(curr))$ ▷ get actual reward for player moving at original position G
    else
      choose an option G' of G so that $pos(c) \neq G'$ for all children c of $curr$
      create a node c with $pos(c) = G'$ as a new child of G
      $r_c \leftarrow 0, n_c = 0$
      append c to $path$
      $r \leftarrow$ value for $pos(c)$ determined by default policy
    end if
    for $k \in path$ **do** ▷ backpropagate reward along path from leaf to root
      $r_k \leftarrow r_k + r, n_k \leftarrow n_k + 1$
    end for
  end for

$c^* \leftarrow \arg\max_{c \text{ is a child of } root(T)} \frac{r_c}{n_c}$ ▷ find child of root with highest average reward (alternatively, highest visit count)

return the move that changes the position from G to $pos(c^*)$

end function

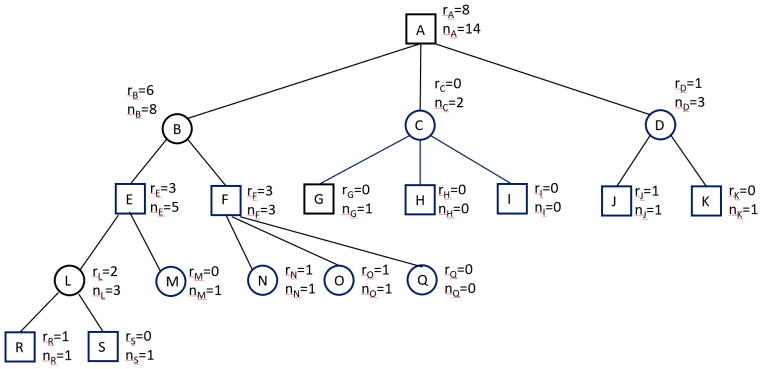

Figure 5.6: Example: Asymmetrical tree built by MCTS after random playouts and resulting rewards B1 C0 D0 E1 F1 L1 N1 M0 R1 O1 G0 J1 K0 S0.

MCTS allows for different implementations of the tree policy used to move from the root of the tree to a leaf and the default policy used to estimate the value of the leaf reached. When the tree policy is UCB and the default policy chooses moves randomly until a terminal state is reached, the corresponding algorithm is called UCB for Trees (*UCT*). Given enough iterations, UCT is guaranteed to return the same move that MINIMAX with no depth limit (so allowed to search all branches of the tree down to terminal positions) would. UCT has been one of the most popular algorithms for developing agents, and has been used successfully for a wide variety of games, including X, Y, and Z [**need˙some˙examples˙here**].

In games where transpositions are possible, UCT will create a separate node for each possible path to the same position. For example, if there is a position E that can be reached by a move from A to position B and then from position C, and also by a move from A to C and then to E, then UCT builds a tree with one node for E reachable along the path ABE, and another node, E' reachable along ACE. Each position reachable from E (and E') also has multiple copies.

The solution for the MINIMAX and ALPHA-BETA family of algorithms is to use a transposition table. The corresponding idea for UCT, due to Childs *et al*, is to build a directed acylic graph (DAG) instead of a tree, In the DAG representation, there is a single node for position E, and both B and C have edges to that node.

In addition to keeping a total reward and visit count for each node in the DAG, we also keep a visit count for each edge. When applying the UCB formula at a parent node to determine which child to visit, the resulting algorithm (*UCT2*) uses the mean reward for all visits of the children in their exploration terms, and the visit counts along the edges and the parent's own visit count in the exploration term (using the visit counts of the children can distort the values of the parent since most of the visits to the best child may come from a different path; seeing a child visited many times through another path would then make the parent explore the other children more, and those lesser children's backpropagated values would then have more weight in the parent's value). When considering node E from node C, the formula is then $-\frac{5}{8}+\sqrt{\frac{2\cdot\ln 3}{2}}$. After arriving at a leaf and applying the default policy, the statistics are updated along the path used to reach the leaf (Childs *et al* propose an alternative *UCT3* that updates statistics along all paths from the leaf to the root (so ACEG and ABEG in the pictured example), which performs better after the same number of iterations, but takes longer in terms of running time because of the possible exponential number of paths that would

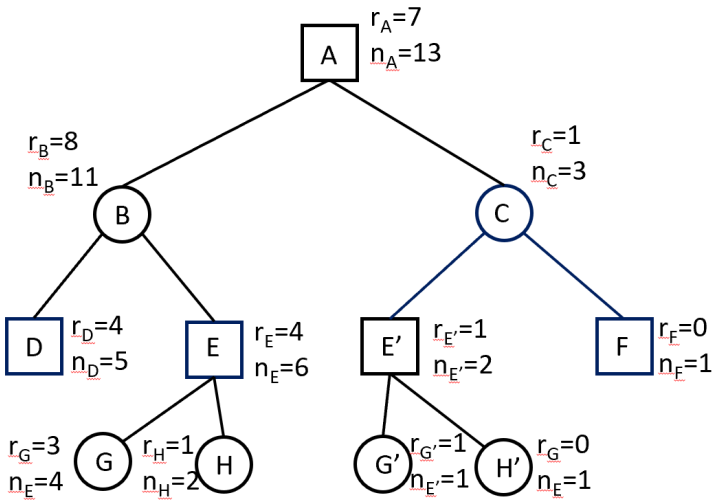

Figure 5.7: Example: E and E' represent the same position reached by different sequences of moves.

have to be updated. Their UCB1; they found that UCB2 is slightly better than UCB1 and comes at no addition cost in runtime [CBK08].)

Browne *et al* note that "the full benefit of MCTS is typically not realised until this basic algorithm is adapted to suit the domain at hand" [Bro+12], and you can refer to their excellent survey for some of the early research into enhancements to MCTS.

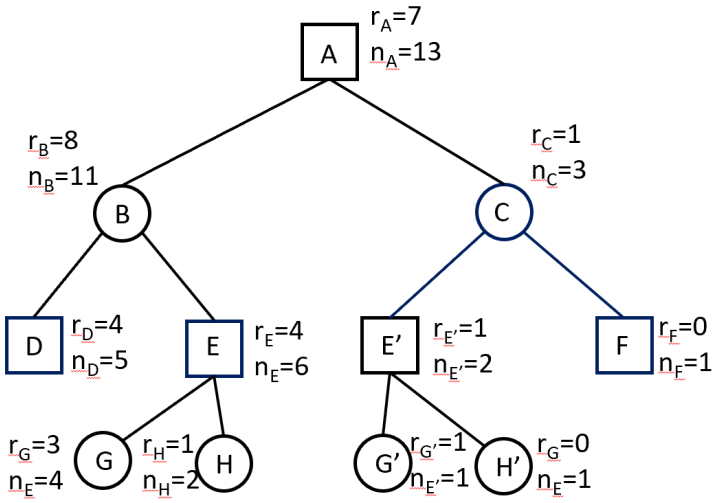

Figure 5.8: Example: Each position is represented by a single node. The path chosen by UCT1 is ACEG, and if the result of applying the default policy is a reward of 0, then the statistics are updated along that path (and only that path) as shown.

Chapter 6

Advanced Topics

6.1 Genetic Algorithms

6.2 Supervised Learning

6.2.1 Decision Trees

6.2.2 Neural Networks

6.3 Reinforcement Learning

Bibliography

[] *Chess Programming Wiki: Search Instability*. URL: https://www.chessprogramming.org/Search_Instability. (accessed: 05.14.2022) (cit. on p. 51).

[ACF02] P. Auer, N. Cesa-Bianchi, and P. Fischer. *Finite-time Analysis of the Multiarmed Bandit Problem*. Machine Learning 47 (2002), pp. 235–256 (cit. on p. 59).

[Bro+12] Cameron B. Browne et al. *A Survey of Monte Carlo Tree Search Methods*. IEEE Transactions on Computational Intelligence and AI in Games 4.1 (2012), pp. 1–43 (cit. on p. 63).

[Bro+20] Noam Brown et al. *Combining Deep Reinforcement Learning and Search for Imperfect-Information Games*. Advances in Neural Information Processing Systems. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 17057–17069 (cit. on p. 45).

[BS17] Noam Brown and Tuomas Sandholm. *Libratus: The Superhuman AI for No-Limit Poker*. Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17. 2017, pp. 5226–5228 (cit. on p. 45).

[BS19] Noam Brown and Tuomas Sandholm. *Superhuman AI for multiplayer poker*. Science 365.6456 (2019), pp. 885–890. eprint: https://www.science.org/doi/pdf/10.1126/science.aay2400 (cit. on p. 45).

[CBK08] Benjamin E. Childs, James H. Brodeur, and Levente Kocsis. *Transpositions and move groups in Monte Carlo tree search*. 2008 IEEE Symposium On Computational Intelligence and Games. 2008, pp. 389–395 (cit. on p. 63).

[Cha+08] Guillaume Chaslot et al. *Monte-Carlo Tree Search: A New Framework for Game AI*. Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference. AAAI Press, 2008, pp. 216–217 (cit. on p. 60).

[CM07] Pierre-Arnaud Coquelin and Rémi Munos. *Bandit Algorithms for Tree Search*. Proceedings of the Twenty-Third Conference on Uncertainty in Artificial Intelligence. UAI'07. Vancouver, BC, Canada: AUAI Press, 2007, pp. 67–74 (cit. on p. 59).

[KM92] Daphne Koller and Nimrod Megiddo. *The complexity of two-person zero-sum games in extensive form*. Games and Economic Behavior 4.4 (1992), pp. 528–552 (cit. on p. 45).

[KMv96] Daphne Koller, Nimrod Megiddo, and Bernhard von Stengel. *Efficient Computation of Equilibria for Extensive Two-Person Games*. Games and Economic Behavior 14.2 (1996), pp. 247–259 (cit. on p. 45).

[Mor+17] Matej Moravčík et al. *DeepStack: Expert-level artificial intelligence in heads-up no-limit poker*. Science 356.6337 (2017), pp. 508–513. eprint: https://www.science.org/doi/pdf/10.1126/science.aam6960 (cit. on p. 45).

[SK00] Nathan R. Sturtevant and Richard E. Korf. *On Pruning Techniques for Multi-Player Games*. Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence. AAAI Press, 2000, pp. 201–207 (cit. on p. 55).

[SW11] Maarten P. D. Schadd and Mark H. M. Winands. *Best Reply Search for Multiplayer Games*. IEEE Transactions on Computational Intelligence and AI in Games 3.1 (2011), pp. 57–66 (cit. on p. 56).

[von96] Bernhard von Stengel. *Efficient Computation of Behavior Strategies*. Games and Economic Behavior 14.2 (1996), pp. 220–246 (cit. on p. 45).