(Provisional) Lecture 30: Game Trees
10:00 AM, Nov 13, 2019

Contents

1 Notes 1
2 Introduction 1
3 Definition 2
4 Example (Tic-Tac-Toe) 3
5 Game Trees 4
  5.1 A Game Tree for Yucky Chocolate 4
  5.2 Using a Game Tree to Look Ahead 5
  5.3 Putting it to Code 6
6 Summary 8

1 Notes

Spike started the lecture (at the lovely TAs’ request) noting that there are two distinct categories of recursion that we have seen so far in CS17:

- structural recursion: this is recursion based on the structure of the data that we are recurring on. For instance, when we have encountered lists, we have almost almost done something with the head, and somehow added it on to the recursive call on the tail (head and tail being the structure of a list). With trees, we have done something with the current node, and somehow added that on to the recursive call on one or both children. In both of these examples, we are using the structure of the data (lists and trees respectively) to inform which element(s) of the data we perform some computation on and which element(s) we recur on.

- non-structural recursion: this is recursion that is not based on the structure of the data. For instance, take mergesort. At each step, instead of using the head and tail of the list (the basic structure of a list), we say split the list in half and call mergesort on each of those. This case, in which we are not using the basic structure of the list to inform our recursion, is an example of non-structural recursion.

Please note that, in class, we only got as far as section 4.1, so do not worry about internalizing all of this information for class on Friday. It will, however, be helpful as you work on Game to have all of this information in the same place so I am just leaving it all together. Spike will go more into the actual coding part during Friday’s lecture.
2 Introduction

This lecture is concerned with search spaces that model parlor games such as go, chess, checkers, othello, and backgammon. At an abstract level, we are concerned with finite, two-player, zero-sum games of perfect information in which the players’ moves are sequential:

- **Finite** - the game eventually comes to an end due to constraints, i.e. the board gets filled
- **Zero-Sum** - a move that is good for Player 1 is equally bad for Player 2, and vice versa
- **Perfect Information** - all players know everything about the state of the game

Assuming the two players are adversaries — meaning they are rational and self-interested, i.e. they play to win — we can use the concept of search trees to get an idea of potential winning strategies. Adversarial search algorithms are designed to return optimal paths that represent these winning strategies, through game trees.

From now on, we’ll refer to finite, two-player, zero-sum games of perfect information with sequential moves as “games”, which will save a lot of typing.

3 Definition

Let’s write down the abstraction of a CS17 game and its components. We’ve also provided some terms and symbols to represent these components in the form of a mathematical model.

A game consists (for now) of 7 main elements:

1. **state**: describes the state of a game; a given game has a finite set of states
   - $X$ is a game’s set of states
   - $x$ is an individual state
2. **start state**: the initial state of a game
   - $x_0 \in X$ is the start state (an element of $X$)
3. **terminal state**: the end state of a game; a given game has a set of possible terminal states. In practice, this set might consist of a *win for Player 1* (a.k.a. a loss for Player 2), a *win for Player 2* (a.k.a. a loss for Player 1), or a *tie*
   - $T \subseteq X$ is a nonempty set of terminal states
4. **status**: conveys current information about a game, such as whether it’s *ongoing* or has terminated in a *terminal state*
5. **state transition**: a function representing the transition of a game into its *next state*; a given state has a set of possible successor states
   - $\delta : X \rightarrow 2^X$ is a state transition function
   - $\delta(x)$ is the set of successor states of $x$
6. **current player**: a label associated with a given state, describing which player makes a move at that state
   - \( \ell : X \to \{1, 2\} \) labels state \( x \) with the player who moves at \( x \)

7. **value**: a real-valued vector describing a terminal state (i.e. \([-1, 1]\), which represents a negative payoff for Player 1, and a positive payoff for Player 2)
   - \( u : T \to [-1, 1] \times [-1, 1] \) maps terminal states into real-valued vectors
   - \( u_i(x) \in [-1, 1] \) is the payoff to player \( i \) at state \( x \in T \)

4 **Example (Tic-Tac-Toe)**

Now that we have an idea of what a game looks like, let’s observe a game of tic-tac-toe, where the "X" represents Player 1 (and therefore plays first), and the "O" represents Player 2 (who goes second). A board is 3x3 and therefore has 9 possible spots. Through this example, we’ll see how we can apply the concepts we previously defined.

1. Our set of **states**, \( X \), consists of all possible tic-tac-toe boards, like the empty board, or ones with 5 Xs and 4 Os but no sequences of 3-in-a-row (meaning the board is filled, but there is no winner – these are called “draws” or “ties”), or any other board that can be reached in the course of a tic-tac-toe game.
   - Note: A board with 3 Xs and one O is not a valid state, because X and O alternate plays!

2. The **start state** \( x_0 \) is the empty board, which is how we start the game of tic-tac-toe.

3. The **terminal states** for tic-tac-toe are the draws, and the states in which one player or the other has won the game.

4. A game of tic-tac-toe’s **status** is either in progress, or has ended in a **terminal state**.

5. The notation \( 2^X \) denotes the set of all subsets of \( X \). The **state transition function** \( \delta \) tells us, for each state \( x \in X \), what are the possible next states after someone makes a move.
   - For instance, in tic-tac-toe, the start state \( x_0 \) is the empty board. \( \delta(x_0) \) is then the set containing the 9 possibilities of what the next state might look like, in which there is exactly one X and zero Os. (There are 9 possibilities because there are 9 spots on a tic-tac-toe board, and if the previous state was empty, choosing any one of 9 spots is a possible move.)

6. The function \( \ell \) marking the **current player** is pretty simple for tic-tac-toe: you take the state \( x \), and if there are the same number of Xs and Os, then \( \ell(x) = 1 \) (O has just made a move, so it’s about to be X’s turn), but if the number of Xs is one more than the number of Os, then \( \ell(x) = 2 \) (X has just made a move, so it’s about to be O’s turn).

7. The function \( u \) tells us, for each state \( x \) of tic-tac-toe, two **values**: the first is “how good is this state for player 1” and the second tells us “how good is this state for player 2”:
   - The mathematical convention is that a value of +1.0 is good, and −1.0 is bad, and 0.0 is neutral.
• Because our game is zero-sum, we know that if \( u(x) = (a, b) \), then \( a + b = 0 \), i.e., \( b = -a \).
• So if we know the utility or value of the game to Player 1, we automatically know the value for Player 2: it’s just the negative of Player 1’s value.
• So from now on, we’ll only talk about the value for Player 1, which we’ll call \( v(x) \).
  
  (a) \( v(x) = 0.0 \) when a state is in a tie  
  (b) \( v(x) = 1.0 \) when a state has three Xs in a row (a win for Player 1)  
  (c) \( v(x) = -1.0 \) when a state has three Os in a row (a loss for Player 1)
• Because the final states for tic-tac-toe are always in exactly one of these three categories (we never reach a final state containing 3 Xs in one row, and 3 Os in another row, for instance!), this completely describes the function \( v : T \rightarrow [-1, 1] \).

**Note:** One odd thing about this representation of a game is that it totally ignores the idea of “moves”, i.e., the consideration of how you get from some state of the game to a successor state. For our Game project, we have to actually include a notion of moves, as we’ll see soon.

## 5 Game Trees

Let’s look at the game “yucky chocolate”, in which we start with a Hershey bar, consisting of \( n \) rows and \( k \) columns of chocolate, but the lower left corner piece is not chocolate — it’s soap! Players take turns breaking off any number of columns from the right, or any number of rows from the bottom, and eating them. The person who eats the soap loses.

This overarching game is really a whole family of games — one for each pair of numbers \( n \) and \( k \) (one for each state). In class, Spike played a game with a student, who lost. Let’s say they reached this state, and it was the student’s move:

\[
\begin{array}{ccc}
  & X & \\
  & o & o \\
  & o & o \\
\end{array}
\]

where “X” denotes the soap. The student probably started thinking to himself “let me see...if I do this, then Spike will do that, and then I’ll have to do this, and he’ll do that and I eat the soap. But if I do this OTHER thing, then ...”

Let’s say he ended up eating one column; Spike took one row; the student took one row; Spike took two columns, and the student had to eat the soap. (Of course, this was all done on the whiteboard, and no actual soap was involved.)

### 5.1 A Game Tree for Yucky Chocolate

Let’s draw out a game tree to represent a new \( 2 \times 2 \) game of Yucky Chocolate. At the top is the \( 2 \times 2 \) board.

We can deduce that Player 1 has several options for their first move (each of which gets a tree branch): eat 1 column, eat 1 row, eat 2 columns, or eat 2 rows. (Eating two rows or two columns
seems like a bad idea, but it’s a legal move, so we’ll include it in our tree! However, since both options result in a loss for Player 1, we’ll represent them in a single branch).

We draw a tree, with an edge for each possible move, and all the new states in the next row. Then we do that again and again until we end up with an empty chocolate bar (denoted by “...”).

(A, -1)  
X  
---  
(B, -1)  (C, -1)  (D, -1)  
X  X  ---  

This game tree, with some annotation, is a visual representation of the stuff we called a \textit{game}:

- The nodes of the tree correspond to different \textbf{states} of the game (denoted by distinct capital letters).
- The \textbf{start state} is the node at the top of the tree.
- The children of a node \(y\) are exactly what we called \(\delta(y)\) (the set of successor/next states).
- The only way the game ends is when the whole Hershey bar is gone, so the nodes that look like “empty” chocolate bars — drawn as dots — are the \textbf{terminal states} \(T\).
- For each state, we write a number in red, saying how good this state is for Player 1. These red numbers are the \textbf{values} of the function \(v\). For instance, in the second row, the “empty” bar that comes from Player 1 eating two rows has value \(-1.0\), because Player 1 ate the chocolate (which was not good for them (haha)).

What’s missing so far is the function \(\ell\) denoting the \textbf{current player} — but this is easy to depict: we can just label the first row “1” (because it’s Player 1’s turn), the second row “2”, the third row “1”, and so on. We really should have a specified \(\ell\)-value for each node, but we’ll just write these at the edge to save clutter.

\section*{5.2 Using a Game Tree to Look Ahead}

Now when Spike and the student played the game, the student, when the board was \(3 \times 3\), had already figured out that they were in a bad position. Somehow, they’d managed to assign a “value” to a state that was \textit{not} a terminal state.

They did this by \textit{looking ahead}, saying “if I do this, and then Spike makes the best possible move, then my next best-possible move will still end up with me eating the soap, so ‘this’ must be a bad choice. But if I do THAT, ...” and so on.
Roughly speaking, this line of reasoning can be summarized as follows: for each node whose children all have values assigned (either by this process, or because they are terminal states, and hence have a value decided by the game rules), we can compute the value to Player 1 in one of two ways:

1. If it’s Player 1’s move, the value of this state is the max of the values of its children, because Player 1 can choose the move that gets the game to that best-possible child state.

2. If it’s Player 2’s move, the value of the state (to player 1) is the min of the values of its children, because Player 2, in an attempt to maximize Player 2’s value, wants to minimize the value to Player 1 (because it’s a zero-sum game).

In short, we can propagate the red values at the terminal node up into the tree, labelling internal nodes, using this alternating-maxes-and-mins approach. In this way, we can compute a value, $V(y)$, for any node of the tree, so that $V$ is defined on all of $X$, while $v$ was defined only on $T$, the terminal states. Doing so is called the minimax algorithm. You’ll be implementing this for the Game project.

### 5.3 Putting it to Code

Now let’s try to convert all that into code. A Yucky Chocolate game state consists of a piece of chocolate of $n$ rows and $k$ columns, as well as an indication of the player whose turn it is, so we define:

```ocaml
type which_player = P1 | P2 ;

type state = (int, int, which_player) ;
```

The initial state (for $2 \times 2$ yucky chocolate) is just $(2, 2, P1)$, so we say:

```ocaml
let initial_state = (2, 2, P1) ;;
```

Now we need to talk about moves. A typical move consists of eating some number of rows or columns. Let’s include that:

```ocaml
type move = Row(int) | Col(int);  
```

When the chocolate has $n$ rows and $k$ columns left, what are the available moves? That’s not part of the formal mathematical notion of a game that we’ve constructed, but we need it for our practical implementation. So we write:

```ocaml
let available_moves (s:state): move list = ...
```

Well... the available moves depend on how many rows and cols there are, so let’s rewrite slightly:

```ocaml
let available_moves = ((n, k, _): state): list(move) =>
  List.append(row_moves(n), col_moves(k));

let rec row_moves = (n: int): list(move) =>
  switch (n) {
```
That turned out pretty nicely. For many games, this kind of thing happens, but for some (like chess) computing the available moves is a big pain!

Once you have a state and a move, what’s the next state?

```
let next_state = ((n, k, w): state, m: move): state =>
  switch (m, w) {
    | (Row(p), P1) => (n - p, k, P2)
    | (Row(p), P2) => (n - p, k, P1)
    | (Col(p), P1) => (n, k - p, P2)
    | (Col(p), P2) => (n, k - p, P1)
  };
```

Now we can write something that tells us a little bit about the state, i.e., combines the function $\ell$ and the set $T$ to give us the status of the game.

```
type status = Win of which_player | Draw | Ongoing of which_player ;;
```

We need to know what the status actually is, so let’s write a procedure (NOTE: there are no “ties” in Yucky Chocolate):

```
let game_status = ((n, k, w): state): status =>
  switch (n, k, w) {
    | (0, 0, P1) => Win(P1)
    | (0, 0, P2) => Win(P2)
    | (_, _, w) => Ongoing(w)
  };
```

Finally, we need to write $v$, which we’ll call value:

```
let value = ((n, k, w): state): float =>
  switch (n, k, w) {
    | (0, 0, P1) => 1.0
    | (0, 0, P2) => (-1.0)
    | _ => failwith("value undefined for nonterminal states")
  };
```

Now, let’s put it all together:
type which_player =
    | P1
    | P2; /* moved up so it could be used in `state' */
type state = (int, int, which_player);
let initial_state = (2, 2, 1);
type move =
    | Row(int)
    | Col(int);

let available_moves = ((n, k, _): state): list(move) =>
    List.append(row_moves(n), col_moves(k));

let rec row_moves = (n: int): list(move) =>
    switch (n) {
    | 0 => []
    | p => [Row(p), ...row_moves(p) - 1]
   );

let rec col_moves = (k: int): list(move) =>
    switch (k) {
    | 0 => []
    | p => [Col(p), ...col_moves(p) - 1]
    );

let next_state = ((n, k, w): state, m: move): state =>
    switch (m, w) {
    | (Row(p), P1) => (n - p, k, P2)
    | (Row(p), P2) => (n - p, k, P1)
    | (Col(p), P1) => (n, k - p, P2)
    | (Col(p), P2) => (n, k - p, P1)
    );

Hypothetically, we could now define another function \( V \) that associates all states with a value, that gets computed by the minimax algorithm (which will be discussed in more depth next lecture!) for non-terminal states, and which matches “value” for the terminal states.

In practice, that works great for \( 2 \times 2 \) Yucky Chocolate, but not so well for things like Chess, where the game tree is so large that you cannot possibly store it on a contemporary computer.

Here’s what we do instead: to find the “minimax” value at some node, we draw a subtree of the gametree, that’s a few levels deep. Some of the leaves of this subtree may be terminal states; for those, we know the value of the leaf. But for others … they’re just internal nodes of the game tree, and we don’t actually know the value. But we want to run minimax, which requires values at all leaves. What can we do?

Answer: Fake it. We can assign a value to the state based on some sort of best-guess.

For more information about this and minimax, see the next set of lecture notes.
6 Summary

- All finite, two-player, zero-sum games of perfect information where the players move sequentially, can be represented mathematically. More importantly, they can be represented by our game module!

- As we saw in class, we can use a tree to keep track of all of the possible states which can result from one player starting at some initial state, and making a move. In “Yucky Chocolate,” each node in the tree is a chocolate bar that resulted from a player choosing to eat a certain number of available rows or columns (i.e., making one of several possible legal moves) from some initial chocolate bar. The leaves would be terminal states of the game, i.e., states in which one player clearly won the game because somebody was forced to eat the soap. Moves are represented by the edges which connect the nodes, and each edge indicates the move that transformed the board from the initial state (i.e., the root of the tree), to the next state (the next node in the tree).

- The actual values in the tree represent the value of any given state of the game for each of the two players. Since we assume the two players play to win, we know that a good outcome for player 1 (usually represented with a positive value) must represent a bad outcome for player 2 (usually represented by a negative value).

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS 17 document by filling out the anonymous feedback form: http://cs.brown.edu/courses/csci0170/feedback.