Lecture 32: Game
10:00 AM, Nov 19, 2018

Contents

1 Game tree 1
2 Graph search applied to game strategy? 1
3 Super moves and winning states 2
4 Writing \texttt{winningP} 2
5 Writing \texttt{losingP} 3
6 Writing \texttt{winningP} and \texttt{losingP} 4
7 Choosing a winning move 4
8 Fifty shades of winning 5

1 Game tree

Consider a game like Connect 4. It consists of a $7 \times 6$ grid. Players alternate adding tokens to the board. Consider the board at a given moment during the game play. Which tokens occupy which grid squares can be represented by a data object in, say, OCaml. This is called the state of the game.

Now let’s consider a graph whose nodes are the different states of the game. There is one special node in which no tokens occupy squares; this is the initial state. Some states are winning states for Player 1 and some states are winning states for Player 2. The neighbors of the node representing a state $s$ are the the states that can be reached from $s$ by different single moves. We will call them successors in this lecture.

Last week we studied graph search. Can any of these be applied to game strategy?

The first thing to note is that the number of possible states is enormous (we estimate somewhere around $2^{49}$.) If you had to write down all the states, you would require an extraordinary computer. This tells us that it is unrealistic to represent a visited set and use it in graph search. The algorithm for graph search thus cannot tell if it has already visited a node; it must treat the graph instead as a tree. Note that the graph does not have cycles, so it’s okay if the graph search “pretends” that the graph is a rooted tree. Graph search won’t notice if the search visits the same node more than once but that isn’t in itself a deal-breaker. So this graph is called a game tree even though it is not actually a tree.
In some games (e.g., chess), the states can cycle, which makes for more problems, but we will disregard this problem for now because there are more significant problems.

2 Graph search applied to game strategy?

So how do we use graph search? How about we try to find a path from the initial state to a winning state?

Graph search is actually a useful strategy for some games—namely, single-player games, which are typically called puzzles. However, graph search as we have seen it so far is not useful in two-player games because of that pesky second player.

Suppose you found a path from the initial state to a winning state. You could choose as your first move the transition from the initial state to the next node on this winning path. However, you cannot force the game to continue down this path because the other player gets to make the decision as to which node of the game tree to go to next.

3 Super moves and winning states

What is the best one could hope for in a first move in Connect 4? A super first move would be one with this property: if you make that move, no matter how your opponent moves in response, you can guarantee a win. What if there were such a super move—how would you find it?

Let’s think about a recursive approach. The goal is a procedure that, given any game state \( s \), finds a super move from \( s \) if one exists. We can design a procedure like this along the lines of our graph search procedure. For simplicity, let’s assume that players strictly alternate turns. (Some but not all games obey this assumption.)

Let’s more carefully define a super move. A super move from state \( s \) is a move that takes the game to a state \( t \) such that, no matter what move your opponent makes from state \( t \), you can continue the game so as to guarantee a win.

Notice that if there is a super move from state \( s \) then it is possible to guarantee a win if the game is started in state \( s \). Let us say such a game state is a winning state.

Using the idea of winning state and super move, we can define a recursive procedure, winningP:

- **input:** a game state \( s \)
- **output:** true if \( s \) is a winning state

Before we try to code this, let’s consider a closely related concept. We say \( s \) is a losing state if no matter what move you select from that state, your opponent can respond with a move that leads to a certain win for the opponent.

Consider the procedure losingP:

- **input:** a game state \( s \)
- **output:** true if \( s \) is a losing state
A note from Prof. Klein: Please note that “losing”, despite the sound of the word, has only one o. Please, please, never write “loosing” for “losing”. It would break my heart. I see this mistake all the time on the Interwebs.

Note that winning and losing are not exact opposites. For example, a state might be neither a winning state nor a losing state. However, they are closely related. Let’s see if we can write these two procedures together.

4 Writing winningP

Suppose you had a procedure successors: state -> state list that, given a game state, returned the list of possible states that could result from doing a single move.

Quiz: Suppose you have as helper procedures successors and losingP. Write winningP in terms of losingP and successors.

Hint: Recall from a couple lectures back the procedure we used in graph search: successP

- input: a procedure f and a list
- output: true if there exists some element of the list for which f returns true

Examples:

1. successP (evenP, [1;2;3;4]) evaluates to true after two applications of evenP
2. successP (evenP, [1; 3; 5;7]) evaluates to false after three applications of evenP

winningP can be written concisely as follows:

let winningP : state -> bool = function s -> successP (losingP, successors s) ;;

The lesson is this: a state is winning if it has any successor state that can guarantee a loss for your opponent (there is a successor state that is a losing state for your opponent).

5 Writing losingP

Can we formulate a similar lesson for losing? First attempt:

A state is losing if it has any successor state that is winning.

What do you think? This cannot be right. Let’s say you’re playing chess. You could put your queen in danger, or even resign. Just because there is a bad move available to you doesn’t mean that you’ve effectively lost.

How do you know you’re in bad shape? When every move available to you is a bad move. This suggests the right lesson: a state is losing if every successor state is a winning state for your opponent.
Given this principle...

*Quiz:* Suppose you have as helper procedures `successors` and `winningP`. Write `winningP` in terms of `winningP` and `successors`.

```plaintext
let losingP : state -> bool = function s ->
...
```

Can you use `successP`? Not directly. That procedure finds the first element in a list for which a procedure $f$ returns true. We want to search the list of successors of $s$ to see if *every* state in that list is a winner. Suppose we had a procedure called `allP` that took a procedure $f$ and a list, and returned true if $f$ applied to *every* element of the list returned true.

*Revised Quiz:*

- Write `losingP` in terms of `winningP` and `successors` and `allP`.
- Write `allP` in terms of `successP`

```plaintext
let losingP = function s ->
  allP (winningP, (successors s)) ;;

let allP = function (f, mylist) ->
  not (existsP ((function x -> not (f x)), mylist)) ;;
```

Example: `allP even? [a;b;c]` is true if `existsP odd? [a;b;c]` is false. All three integers are even if none of them is odd.

### 6 Writing `winningP` and `losingP`

Now we can put our procedures together using *mutual recursion*.

```plaintext
let winningP : state -> bool = function s ->
  existsP (losingP, successors s)
and losingP : state -> bool = function s ->
  allP (winningP, (successors s)) ;;
```

That’s it!

What about the base cases? When will the recursion stop?

If the game is finite—if there are no cycles, which is true, for example, for Connect 4—then there must be states that have zero successors. So it seems that the base cases involve `successors s` being an empty set. However, these cases are handled in `existsP` and `allP`.

Note that `existsP` on an empty list returns false, and `allP` on an empty list returns true. (Study this on your own if it is not clear to you that these are the right answers.)
7 Choosing a winning move

Remember that our original goal was to find a winning move. When we studied graph search, we first wrote code to determine the existence of a path from a source to a destination. We then wrote a procedure to actually return the path if it existed.

You could similarly use the ideas from winningP to write a procedure that, given a state, returned a super move if one existed:

\[
\text{find\_super\_move} \quad \text{\textbf{input:} a state } s \quad \text{\textbf{output:} a super move from } s \text{ if one exists}
\]

It’s natural to use a move option as the return type to account for the possibility of there not existing a super move.

We won’t write this in the notes, but you’re encouraged to attempt to write a procedure like this on your own!

8 Fifty shades of winning

In reality, it is not practical to use this approach. Game trees will get too broad as the number of states increases; in addition, many games involve cycles, which will interfere with this methodology. Further, in some games, it’s not just about winning or losing. There is a score, and the goal is to achieve the biggest score, or the biggest point spread. How can we adapt what we have learned to this scenario?

To make things simple, let’s focus on the point spread: my score minus your score. I want to make this quantity (called the payoff) as big as possible (I want to maximize the payoff), and you want to make it as small as possible, to minimize it. (This is the zero-sum assumption; a zero-sum game is one in which every dollar you lose is a dollar I gain.) Instead of talking about you and me, let’s talk about the payoff-maximizer, and the payoff-minimizer.

The basic winning/losing scenario can be accommodated in this new scoring framework: just say the payoff is 1.0 if the payoff-maximizer wins, and -1.0 if the payoff-maximizer loses, and is zero if there is a tie.

Then the procedure winningP tells me if there is a move that can guarantee me a payoff of 1.0, a super move. That is, a super move is one that can guarantee a payoff of 1.0.

Back to a more general scenario: what if one move guarantees me a payoff of 1.0 but another move guarantees me a payoff of 10.0? I want to choose the latter. Why? Because bigger payoff is better! So suppose that for each move I know the largest payoff guaranteed if I make that move—which move should I choose? The one that guarantees the largest payoff!

Let’s try to use this insight to write a mutually recursive solution to finding the best achievable payoff.

Let’s say I’m the payoff maximizer and you’re the payoff minimizer. I am given a state \( s \). There are moves available to me, giving rise to successor states \( s_1, \ldots, s_7 \). Which move should I make? Which
state do I want the game to be in when it is your turn?

Given any state, you will do your best to achieve a small payoff (maybe even negative). I anticipate that. I want to put the game in the state where the smallest payoff you can achieve is as large as possible. So if I know the smallest payoff you can achieve for each of the states $s_1$ through $s_7$, I know which one to choose—I choose the one for which that smallest payoff is the largest.

We’ll define two procedures, `max_payoff` and `min_payoff`:

- `max_payoff`, given a state $s$, finds the maximum payoff that can be guaranteed by a move from that state.
- `min_payoff`, given a state $s$, finds the minimum payoff that can be guaranteed by a move from that state.

**Quiz:** Write the pseudocode for these two procedures. Due to time restrictions, this quiz was pushed to the next lecture. Feel free to give it some thought and come prepared on Wednesday to write these procedures!

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](http://cs.brown.edu/courses/csci0170/feedback)