Game AIs

https://xkcd.com/1002/
Game tree

States of game represented by nodes of a graph
Neighbors of a node are the possible next states (called *successors*)

Let *successors* be procedure that, for given state \( s \), returns list of successors of \( s \)

Too many nodes for algorithm to maintain a *visited* set.
Algorithm for graph search cannot tell if it has already visited a node.
Algorithm must treat graph as if a tree.
Called a *game tree*
Graph search for game trees?

Find a path from initial state to winning state?
Works for single-player games, a.k.a. puzzles.
Not good for two-player games because other player can influence the path.

Instead, use ideas from graph search to address game-tree search.

Best one could hope for in a first move?
A super move would be one such that:

no matter what move your opponent makes in response, you can force a win

Want a procedure that, given a state \( s \), finds a super move from \( s \) if one exists.

If there is a super move from state \( s \) then can guarantee a win if you start in state \( s \).

Say such a state is a winning state.
Let’s try to write a procedure that can tell if a state is a winning state.
Identifying winning states

Say a state is a *winning* state if from that state you can guarantee a win.

Goal: a procedure \texttt{winningP} with the following spec:

- \textit{input}: a game state \( s \)
- \textit{output}: true if \( s \) is a winning state

Say a state is a *losing* state if no matter what move you make, your opponent can respond in a way that guarantees you will lose.

Secondary goal: a procedure \texttt{losingP} with the following spec:

- \textit{input}: a game state \( s \)
- \textit{output}: true if \( s \) is a losing state

\textbf{Note}: winning state and losing state are \textit{not} opposites.
A state can be neither winning nor losing.
Quiz: Suppose you have helper procedures `successors` and `losingP`. Write `winningP`.

**Hint:** Recall from lecture the procedure we used in graph search:

\[\text{successP} \rightarrow \text{existsP}\]

**input:** a procedure \(f\) and a list

**output:** does there exist a element of list for which \(f\) returns true?

**Examples:**

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

I call it `existsP` because it’s true if there exists some element of list for which \(f\) returns true.
Quiz: Suppose you have helper procedures `next` and `losingP`. Write `winningP`.

`winningP`:
- **input**: a game state `s`
- **output**: true if `s` is a winning state

```
let winningP = function s ->
  if successP (losingP, successors s) then true else false
```

Similar lesson for `losing`?
**winningP**

*input*: a game state $s$

*output*: true if $s$ is a winning state

**losingP**

*input*: a game state $s$

*output*: true if $s$ is a losing state

**Lesson**: a state is *winning* if it has any successor state that is a *losing state* for your opponent

Similar lesson for *losing*?

*First attempt*: “A state is losing if it has any successor state that is winning.”

That cannot be right.

Say you are playing chess.

You could put your queen in danger or even knock over your own king.

Just because a bad move is available to you doesn’t mean you’re destined to lose.

How do you know you’re in bad shape? When *every* move available to you leads to losing.

**Lesson**: a state is *losing* if *every* successor state is a *winning state* for your opponent
Lesson: a state is **winning** if it has any successor state that is a **losing state** for your opponent.

Lesson: a state is **losing** if **every** successor state is a **winning state** for your opponent.

Quiz: Suppose you have helper procedures `successors` and `winningP`. Write `losingP`.

```
let losingP = function s -> ...
```

Can you use `existsP`/`successP`? Not directly.

`existsP` searches for first element in a list for which `f` returns true. We want to search list of successors of `s` to see if **every** state in that list is a winner.

`allP`: takes a procedure `f` and a list, and returns true if `f` applied to every elt returns true.

Revised Quiz: Write `losingP` in terms of `successors` and `winningP` and `allP`.
**Lesson**: a state is *winning* if it has any successor state that is a *losing* state for your opponent.

**Lesson**: a state is *losing* if every successor state is a *winning* state for your opponent.

allP: takes a procedure f and a list, and returns true if f applied to every elt returns true.

**Revised Quiz**: Write losingP in terms of successors and winningP and allP.

... and write allP

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

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

Could also write allP recursively from scratch.
**input:** a game state $s$
**output:** true if $s$ is a winning state

**input:** a game state $s$
**output:** true if $s$ is a losing state

**Lesson:** a state is *winning* if it has any successor state that is a *losing* state for your opponent

**Lesson:** a state is *losing* if every successor state is a *winning* state for your opponent

**Mutual recursion:**

```
let winningP = function s ->
  existsP (losingP, successors s)
and losingP = function s ->
  allP (winningsP, (successors s))
```

What about base case?
When does recursion stop?
When game ends, it could be a win or loss or tie.

This consideration needs to be incorporated into these procedures.
Finding winning move

Original goal: to find a winning move.

In graph search, we first wrote code to determine existence of a path, then wrote procedure to actually return the path itself.

You could similarly use ideas from winningP to write a procedure that given a state $s$, returns a super move if one exists.

```
find_super_move
input: game state s
output: a super move from s if one exists

Should use a move option as the return type.
```
“It’s not just about winning and losing… It’s about the point spread.”

**Payoff:** One player wants it as big as possible, the other wants it as small as possible.

Zero-sum assumption: my gain is your loss, and vice versa.

Can formulate basic win-loss scenario in this setting.

What if one move guarantees me a payoff of 1.0 … but another move guarantees me a payoff of 10.0? I would go for the bigger payoff!

**Principle:** Suppose for each move available to me, I know the largest guaranteed payoff. Which move should I choose? The one that guarantees the largest payoff!
Two players: payoff maximizer and payoff minimizer.

We'll define two procedures, \texttt{max\_payoff} and \texttt{min\_payoff}:

\texttt{max\_payoff}, given a state \( s \), finds the maximum payoff that can be guaranteed by a move from that state.

\texttt{min\_payoff}, given a state \( s \), finds the minimum payoff that can be guaranteed by a move from that state.