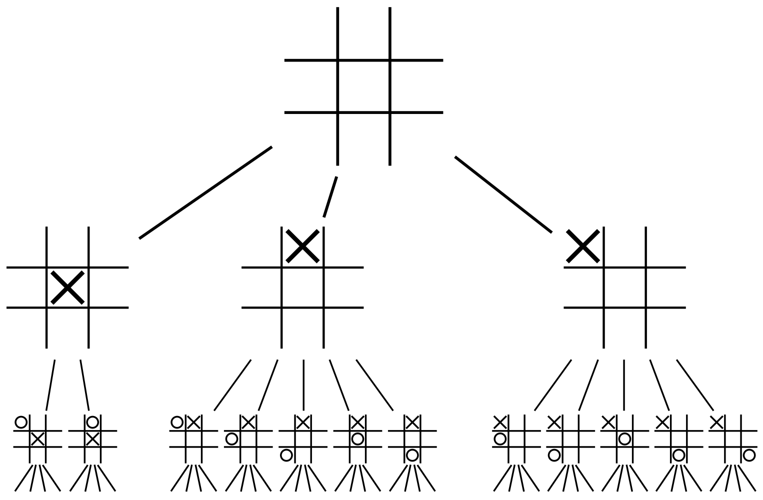
Task 4 Spike: Graphs, Search & Rules

Context

The purpose of this lab is to give you experience representing games as discrete states. A graph of states allows us to explore the outcome of a game by applying actions, and to search for game results using graph-based search.

A sample game graph or game tree for tic-tac-toe is presented to the right. We can treat each possible state the game can be in as a node or vertex on the tree, and each move is an edge, or a way of moving from one node to another.

Our AIs can then evaluate moves by searching the tree to see where their moves and potential opponent moves may take them.



Knowledge/Skill Gap:

Developers of game AI need to know how to represent games and graphs and identify the best move in the current game state by searching the graph. The developer needs to know the pitfalls of random search and hot to optimise for both efficiency and effectiveness.

Goals/Deliverables:

1. Tic-tac-toe code modified to represent the game state as a graph.
2. An AI that randomly searches the graph
3. An AI that improves the efficiency of a basic random search
4. An AI that improves the effectiveness of a basic random search

Planning Notes:

- Look at the picture above! Each node in the graph is the state of the tic tac toe game after a given move. In order to use a graph based approach, your AI will need copies of these boards. How can your code pass these to your AI? Or will there be a function your AI can call to get the board state after a given move?
- An AI that randomly searches the graph will look something like the following pseudocode:

```
1. Get the current state of the board (our root node)
2. Randomly attempt a move, push the new board state onto an internal list
3. Does this move result in a victory? Yes, return the list
4. Otherwise take the last board state in the list and return to #2
```

  This will result in a particularly stupid (it doesn't even recognise that another player will be taking moves between it's random moves!) and inefficient (it can attempt hundreds or even thousands of invalid moves before it stumbles across a correct move, let alone a winning game – if you set your AI up to play itself, it could take several minutes to win [statistically it's possible it could never finish a game], even with on a powerful computer). But the next steps are about finding ways to improve.
- So how can we improve efficiency? Efficiency is about not taking moves we know are wasteful – a good basic metric is to check if a move is legal before you add it to our list.

Note: We can also address 'efficiency' by only searching one game state deep – after all, the other player will be moving between us, so any plan may be immediately ruined. But we address this in the 'effectiveness' section, so you should look for other ways to improve efficiency at this stage!

- How can we improve effectiveness? Effectiveness is usually what we consider first when describing an AI as 'good' or 'bad' – is it capable of beating opponents? How can searching the graph help us beat an opponent or win more often? Can you change the search to work backwards from a known win state or a known good state? Or maybe you can try and anticipate an opponent's moves? The gold standard here is the minimax algorithm – perhaps you could attempt an implementation for your AI?

Extensions

- Can you assign probabilities to an opponent's moves? Then use these probabilities to allow your AI to block more aggressively.
- Can you create an AI that always plays a perfect game? Can your AI play perfectly on a 4x4 board? Or on a 3x3x3 board? Or on any board size variation?
- Can you visualize your AI's decision making? Printing out a chart like the image at the top of this spike? Or like xkcd: Tic-Tac-Toe? As you'll find out in later spikes, being able to visualise an AIs decision making is critical for debugging.