CS4710: Artificial Intelligence Search

Many AI problems boil down to graph search. What are the primary techniques for searching graphs? What do we do when the graph is very large?

Game Playing Systems

Games!

- Games are well-defined problems that are generally interpreted as requiring intelligence to play well.
- There is definitely some uncertainty in games because you don't know what your opponent will do.
- Usually only interesting when search spaces are VERY large (no surprise there).

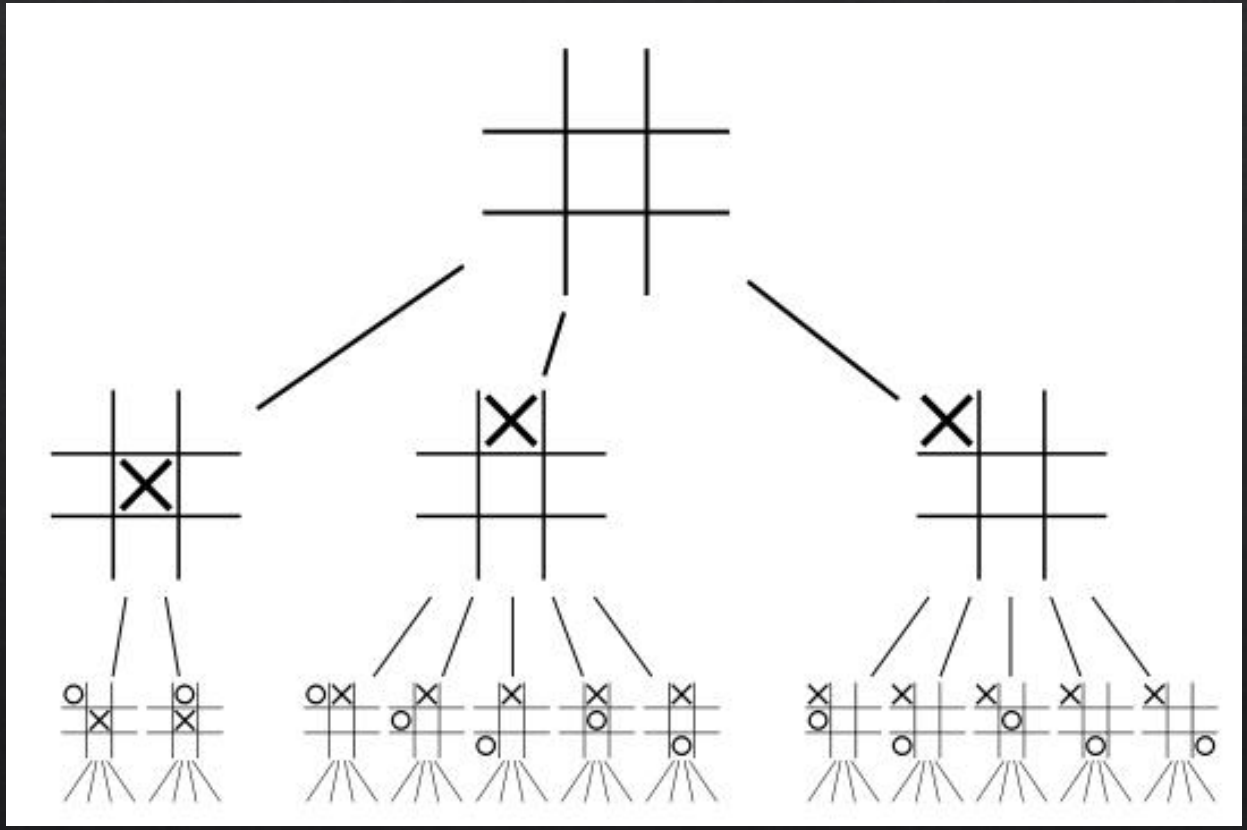
Humans are pretty good at games

- Human players do quite well at games, but it probably is not because humans are good at exhaustively searching the possible outcomes in a game.
- We are good at ignoring irrelevant paths in the “game tree” and remembering patterns we’ve seen before.
- Thus, this is a good test domain for search and for AI

Game Playing AIs

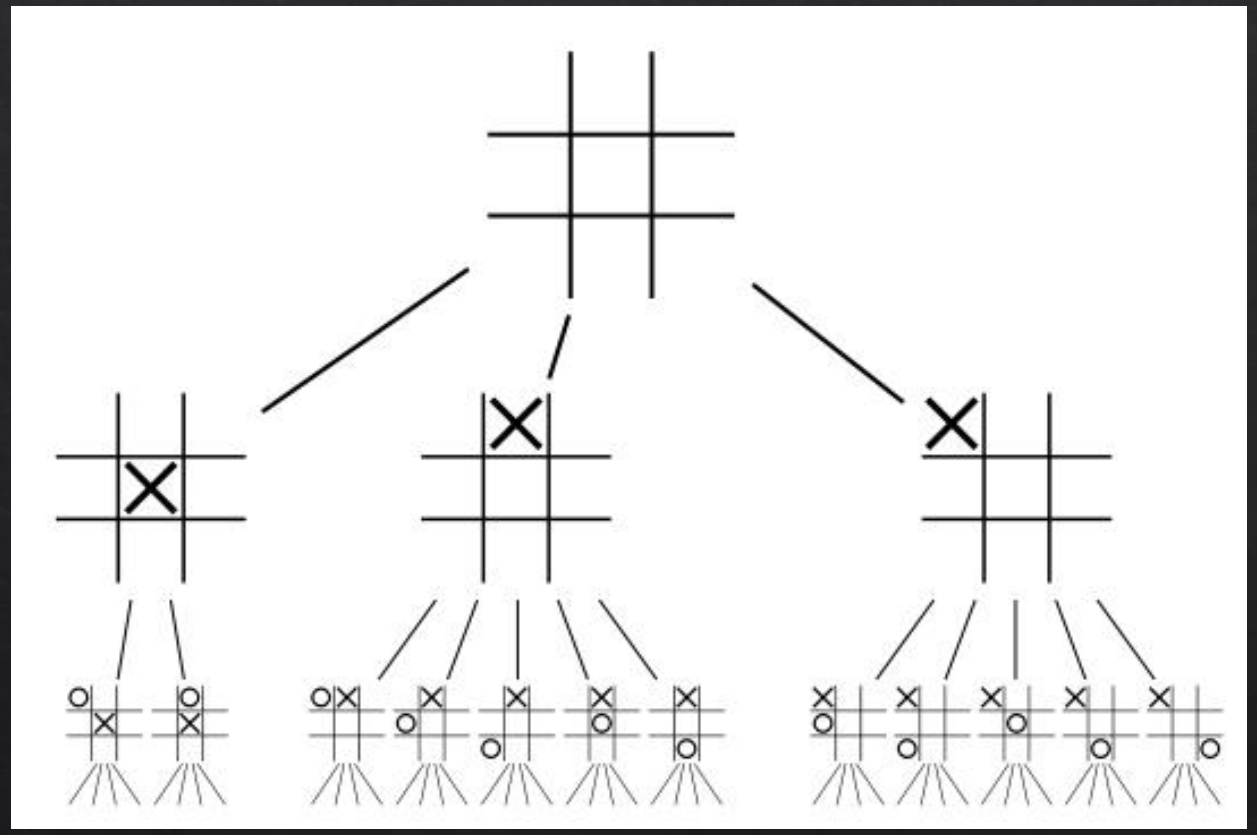
- Main idea: Represent the game as a search tree
  - If the search tree is small, we can search every possible way the game can play out and the AI can always play optimally!
  - If not, we need some heuristics that help the AI focus its search on more promising areas.
  - Let's take a look!

Tic-Tac-Toe



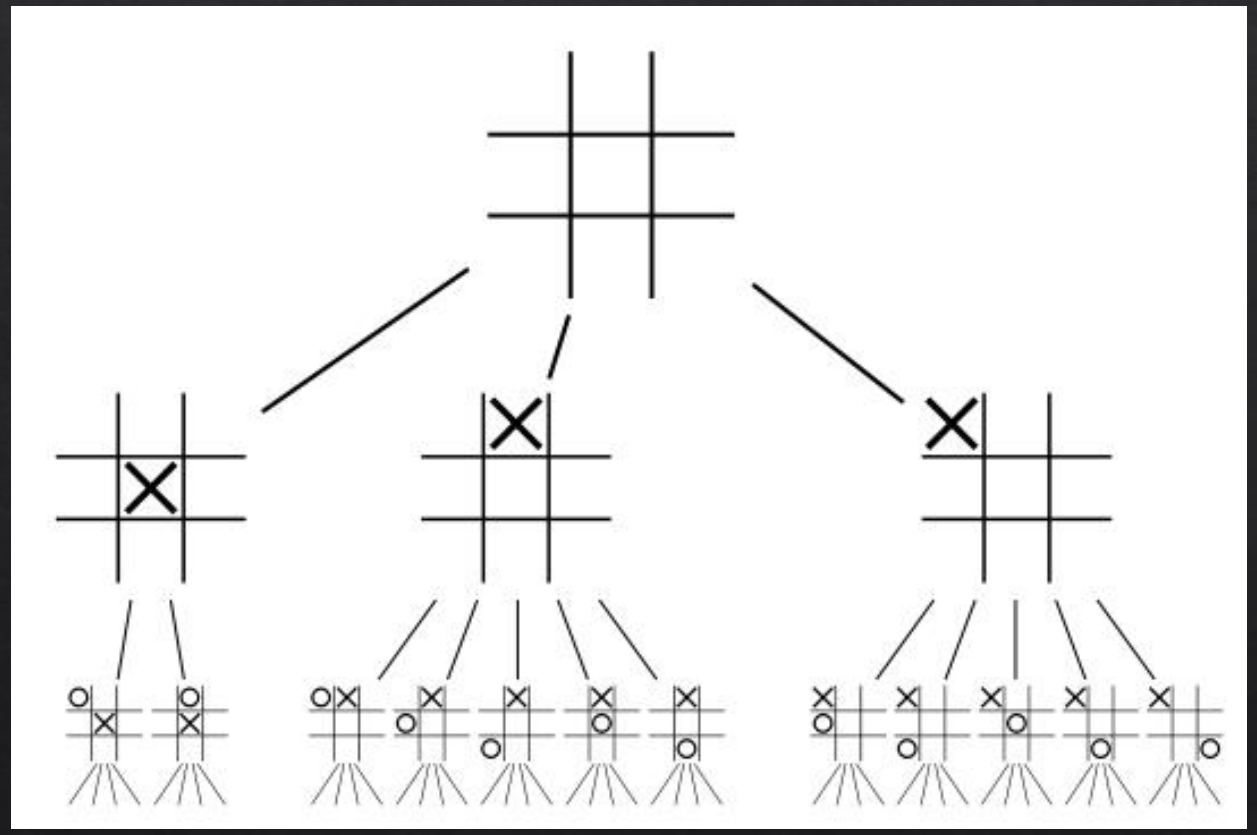
- How to make an “AI” that plays Tic-Tac-Toe
- Many games can be modeled as graphs or trees and reduce to a search problem
- Consider the partial tree to the left
- How can we search this tree to figure out how to play the game?

Tic-Tac-Toe



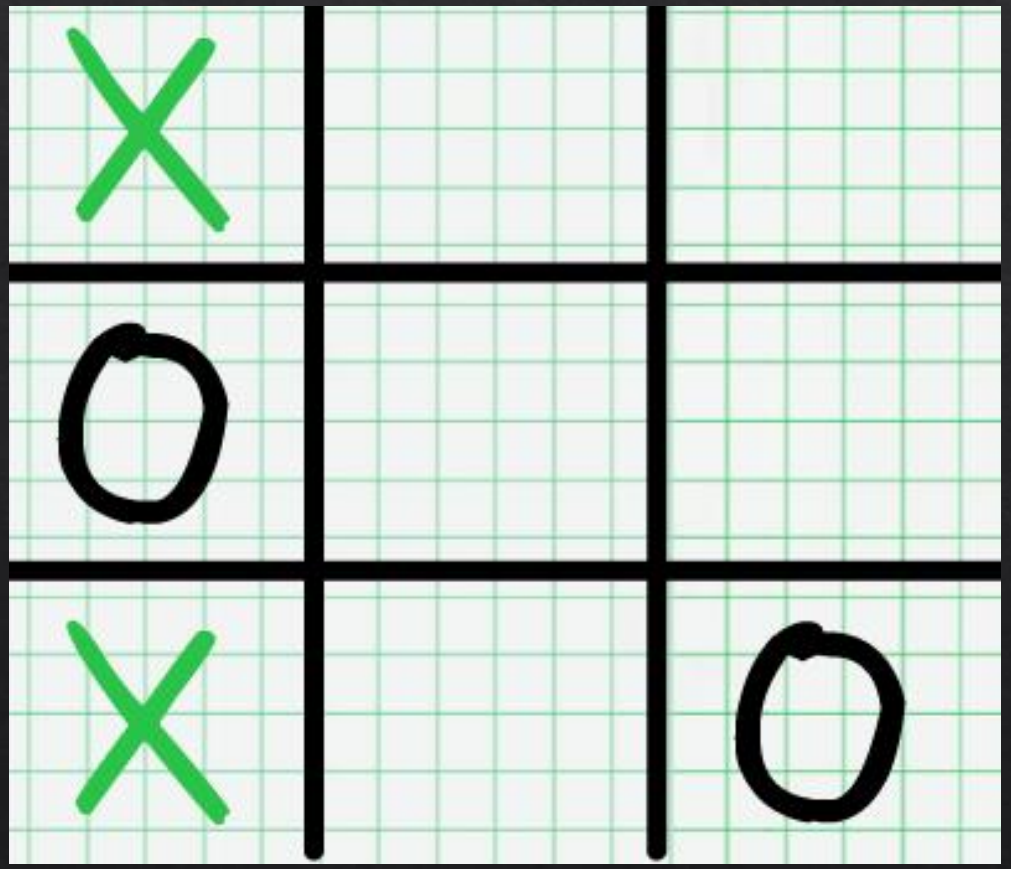
- States where the game is over is called a **Terminal State**
- A **utility function** determines the payout or value of a terminal state (more on this in a moment)
- Each layer in the search tree is called a **ply**

Tic-Tac-Toe



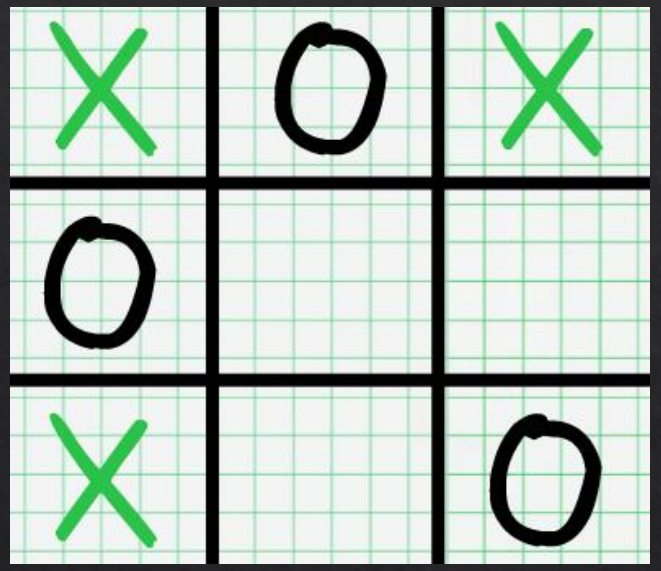
- Attempt 1!
  - Simply search the tree and find a path to the “good” terminal states
  - What are the good terminal states? When I have three X’s in a row!
- Problem!
  - I’m not making every decision of which branch to follow! My opponent makes half of the choices. Need to account for this.

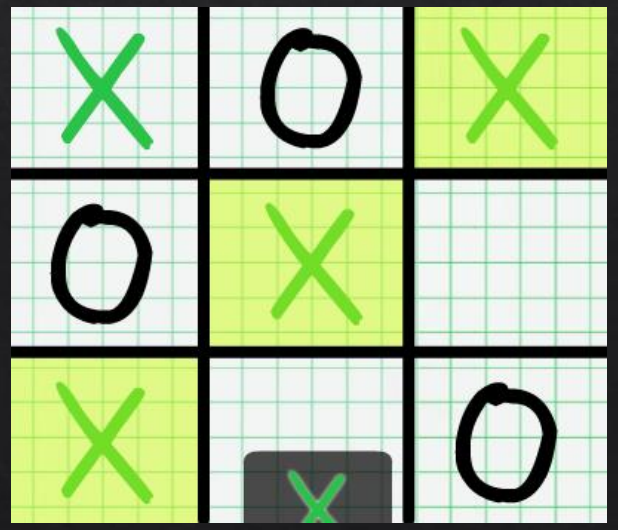
Tic-Tac-Toe



- Idea! Some states are special in that I can always win if I reach that state.
- The state to the left for example (I'm playing X).
- I have already won because I will go in the upper right corner.
- No matter what opponent does, I eventually win

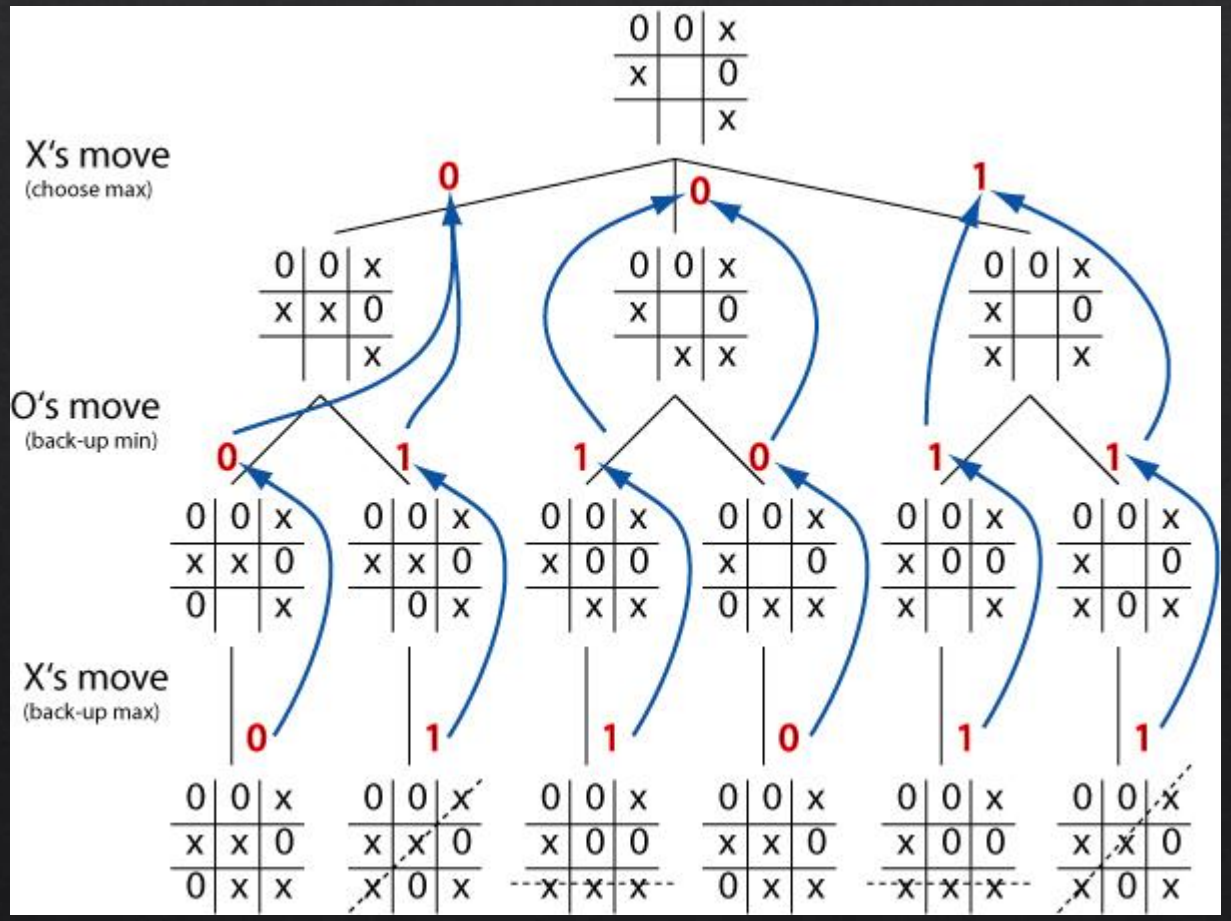
Tic-Tac-Toe





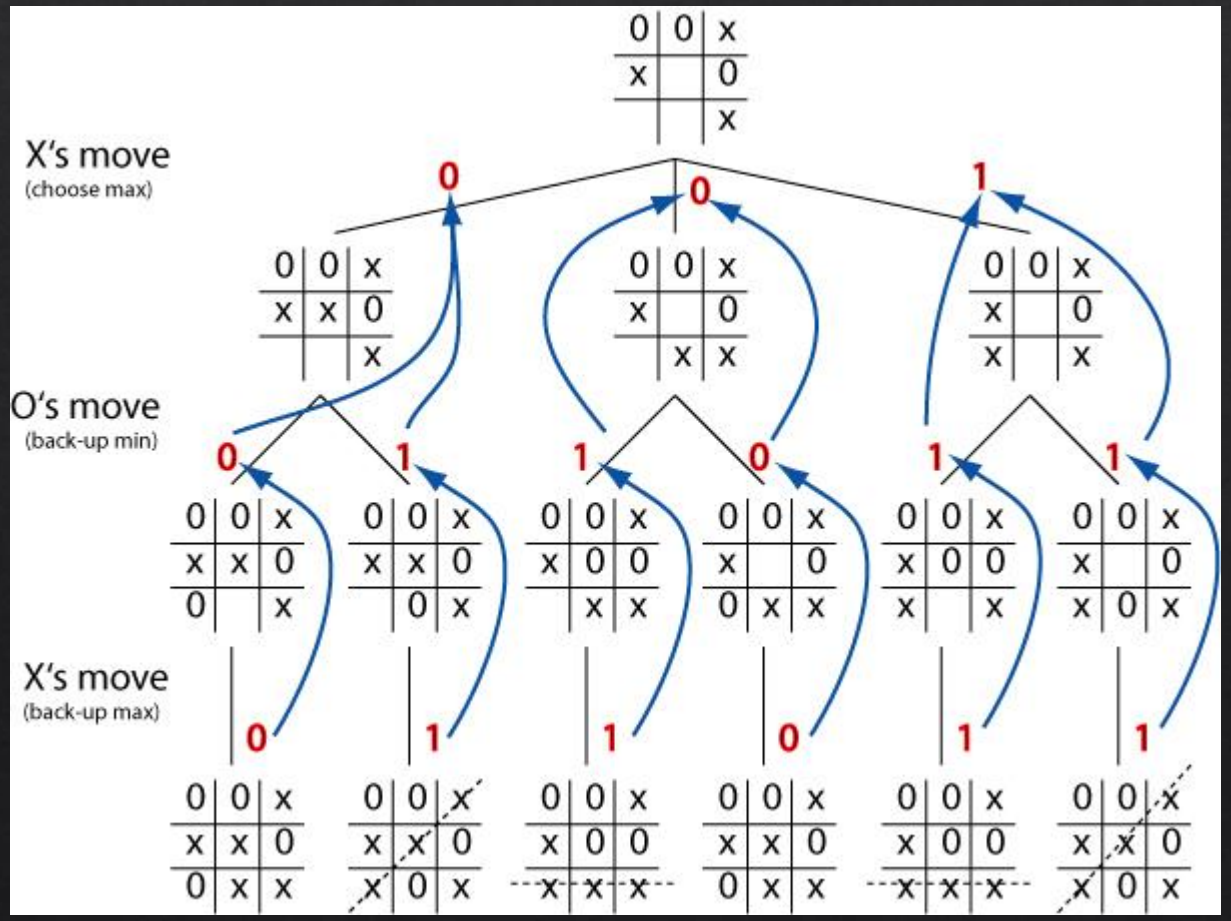
- More obvious example:
  - Bottom state is a win state.
  - If I'm at the upper state (and it is my turn) then I will definitely win.
- How do we extrapolate this to an algorithm?

Tic-Tac-Toe: Minimax Procedure



- Give leaf nodes a score
  - 1 if AI wins
  - -1 if AI loses
  - 0 if draw
- Base Case:
  - If node on decision tree is a final state (win or lose or draw).
  - Return 1, -1, or 0 respectively.
- So this algorithm will need to find the bottom of the tree in order to work.

Tic-Tac-Toe: Minimax Procedure



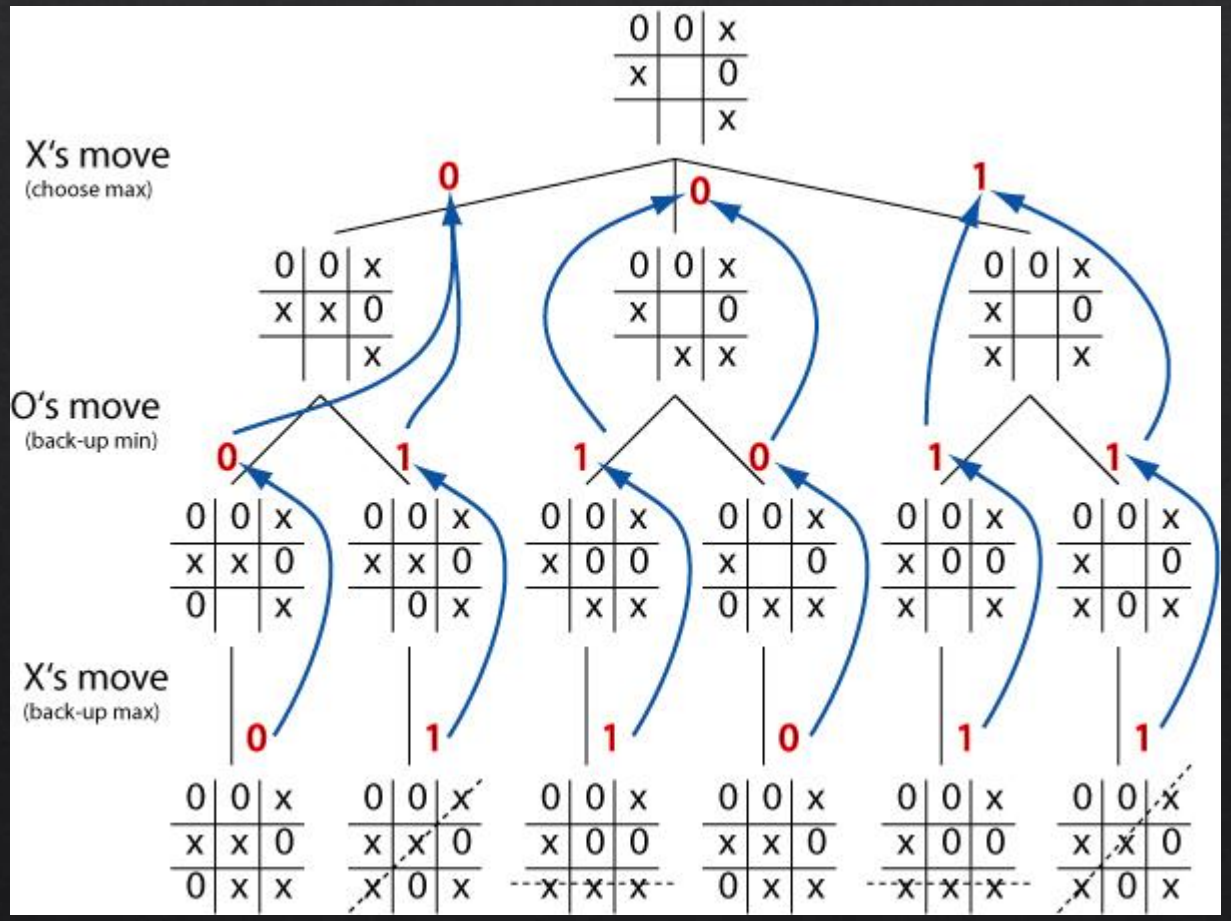
- Give leaf nodes a score
  - 1 if AI wins
  - -1 if AI loses
  - 0 if draw
- Recursive case: If internal node:
  - Take the minimum score if it is the human's turn (assume they make best play possible)
  - Take max score if it is AI's turn (AI takes best move possible)
  - AI simply follows the maximum score path through the tree

Minimax Procedure

```
function MINIMAX-DECISION(game) returns an operator

    for each op in OPERATORS[game] do
        VALUE[op] ← MINIMAX-VALUE(APPLY(op, game), game)
    end
    return the op with the highest VALUE[op]
```

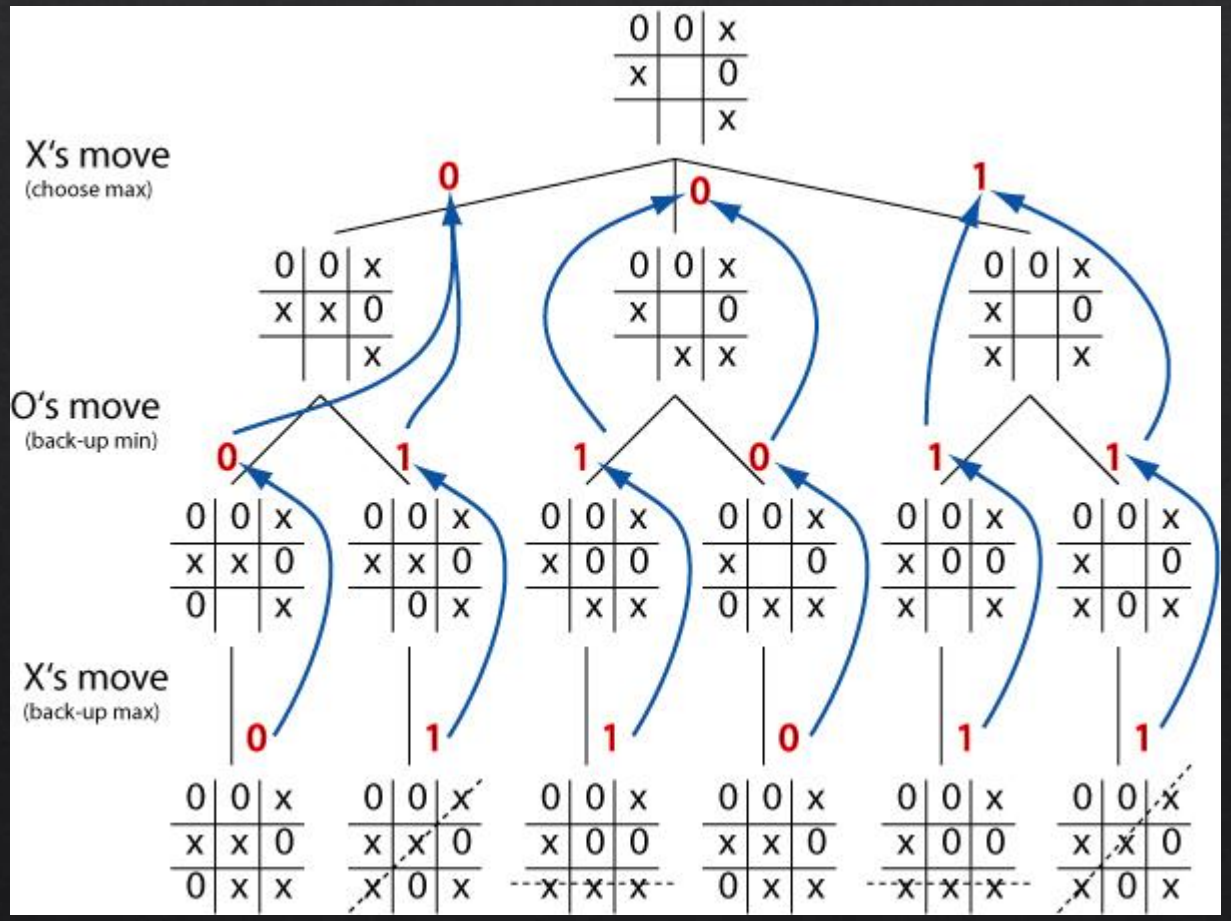
```
function MINIMAX-VALUE(state, game) returns a utility value

    if TERMINAL-TEST[game](state) then
        return UTILITY[game](state)
    else if MAX is to move in state then
        return the highest MINIMAX-VALUE of SUCCESSORS(state)
    else
        return the lowest MINIMAX-VALUE of SUCCESSORS(state)
```

Tic-Tac-Toe: Minimax Procedure



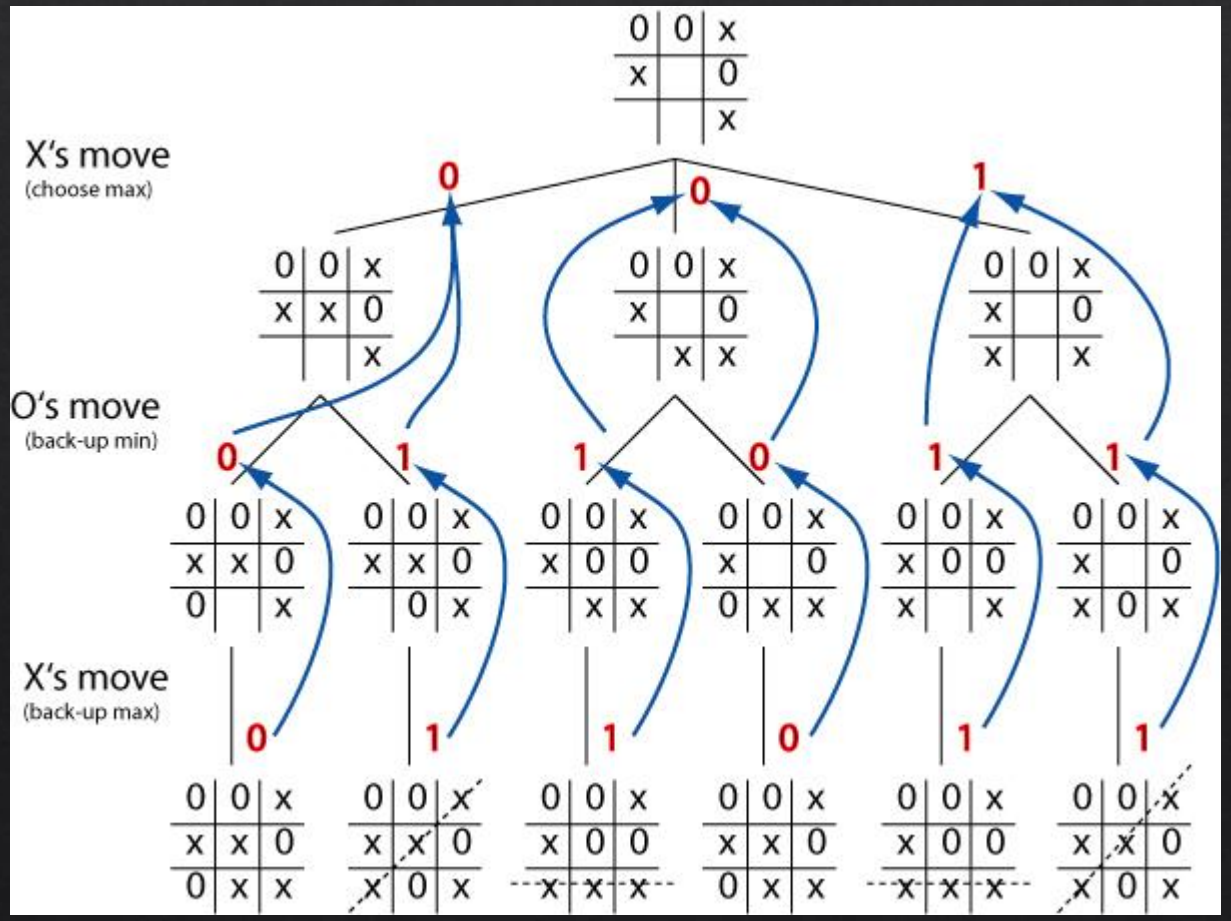
- Another thing to think about:
- Minimax has a major assumption:
  - The opponent is as knowledgeable as the computer and will ALWAYS make the best choice.
  - This could lead the algorithm to be pessimistic in the sense that it never tries to sneak a counter-intuitive move past it's opponent.
- Example?

Tic-Tac-Toe: Minimax Procedure



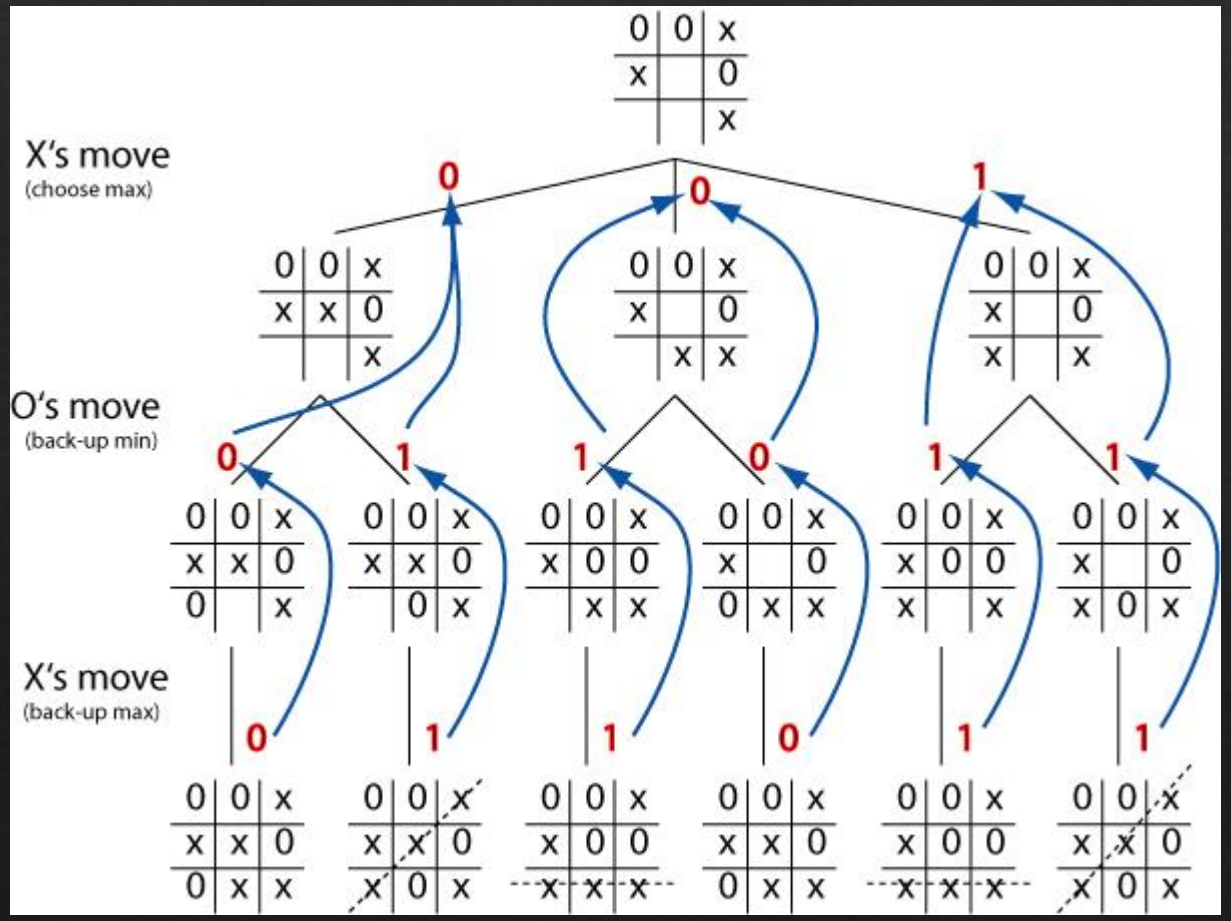
- Corollary:
- For many games (like tic-tac-toe) pure minimax will conclude that there is NO opening move that guarantees victory.
  - Because there is not, and if there was no one would play the AI!
- Thus, it will choose optimal moves until (hopefully) the opponent messes up and leads to a node with a positive value.

Tic-Tac-Toe: Minimax Procedure



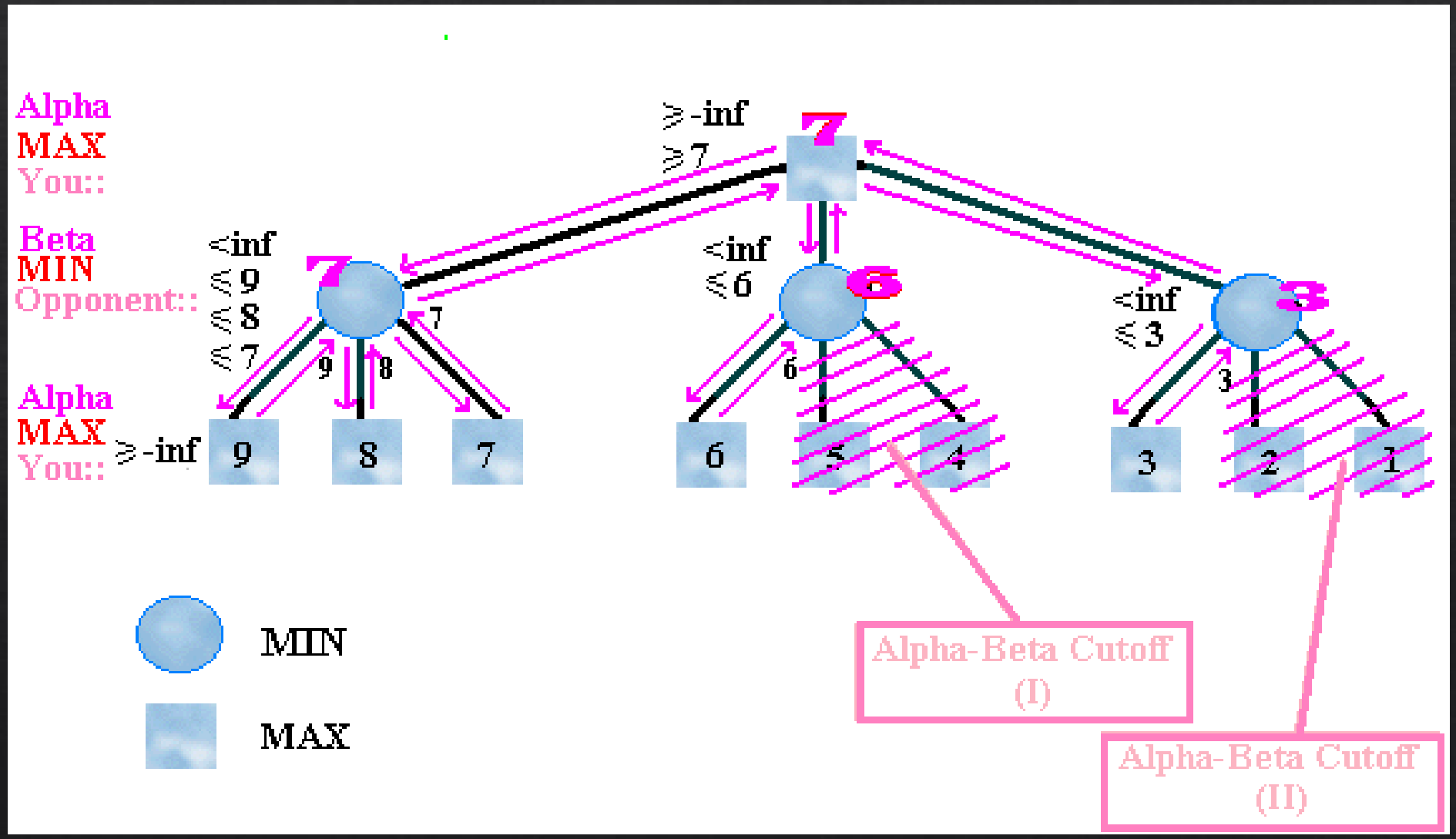
- What about games like backgammon that have an actual score (not just a win or lose state like tic-tac-toe).
- How would we alter minimax algorithm to account for that?

Tic-Tac-Toe: Minimax Procedure

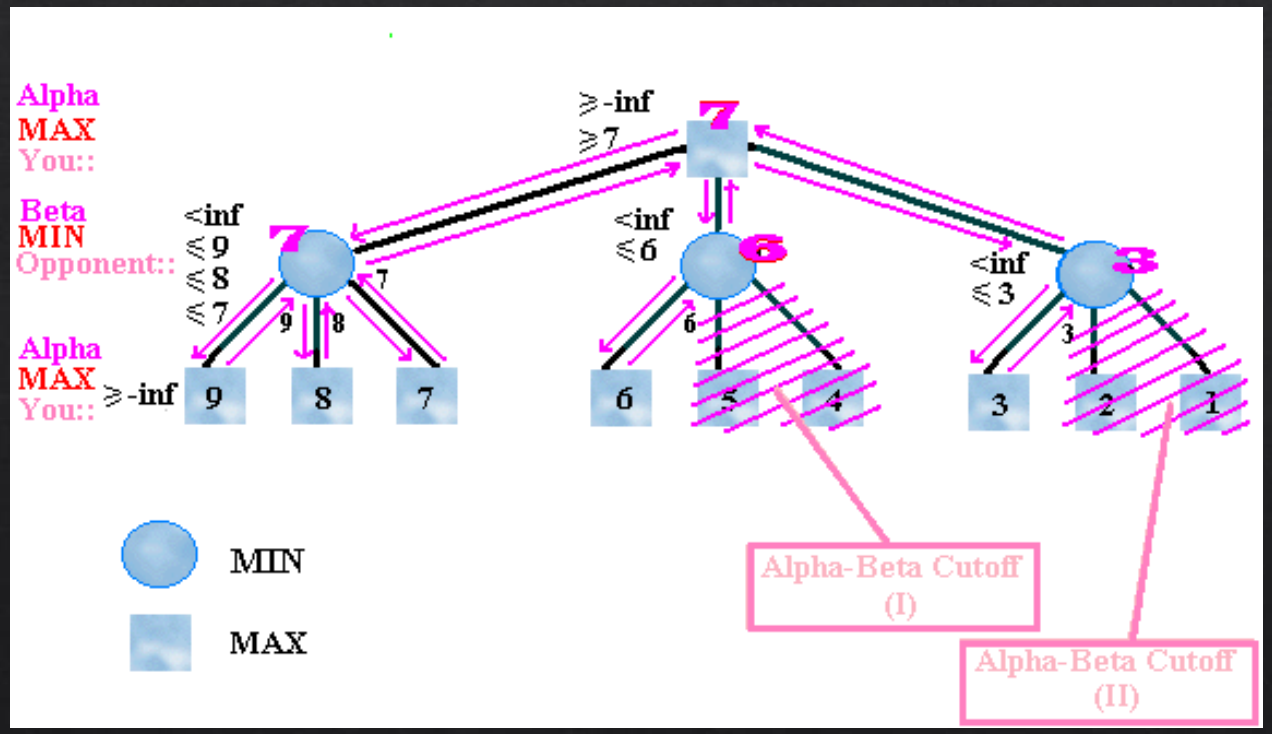


- Randomness:
- Life is complicated even further if a game has random components to it.
  - Like backgammon
- Most systems, again, employ some kind of expected value function to choose moves.
  - This is called the *expectimax* approach

Tic-Tac-Toe: Alpha-Beta Pruning

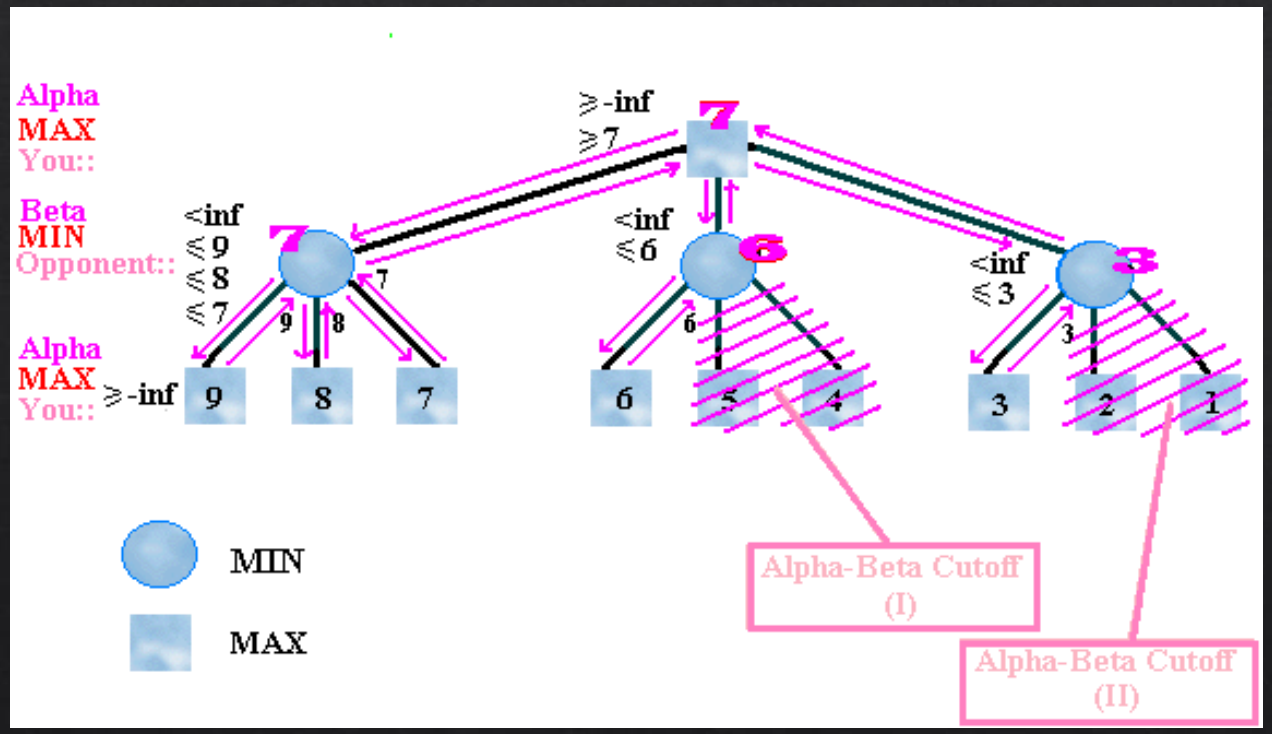


Tic-Tac-Toe: Alpha-Beta Pruning



- Slight optimization to the minimax procedure.
- Don't evaluate branches if they won't affect the AI's choice.
  - E.g., if I have a winning path I can take already, then don't waste time calculating other paths that can't exceed it
- PL's do this with Boolean expressions
  - A || B. If A is already evaluated to true, no need to evaluate B

Tic-Tac-Toe: Alpha-Beta Pruning



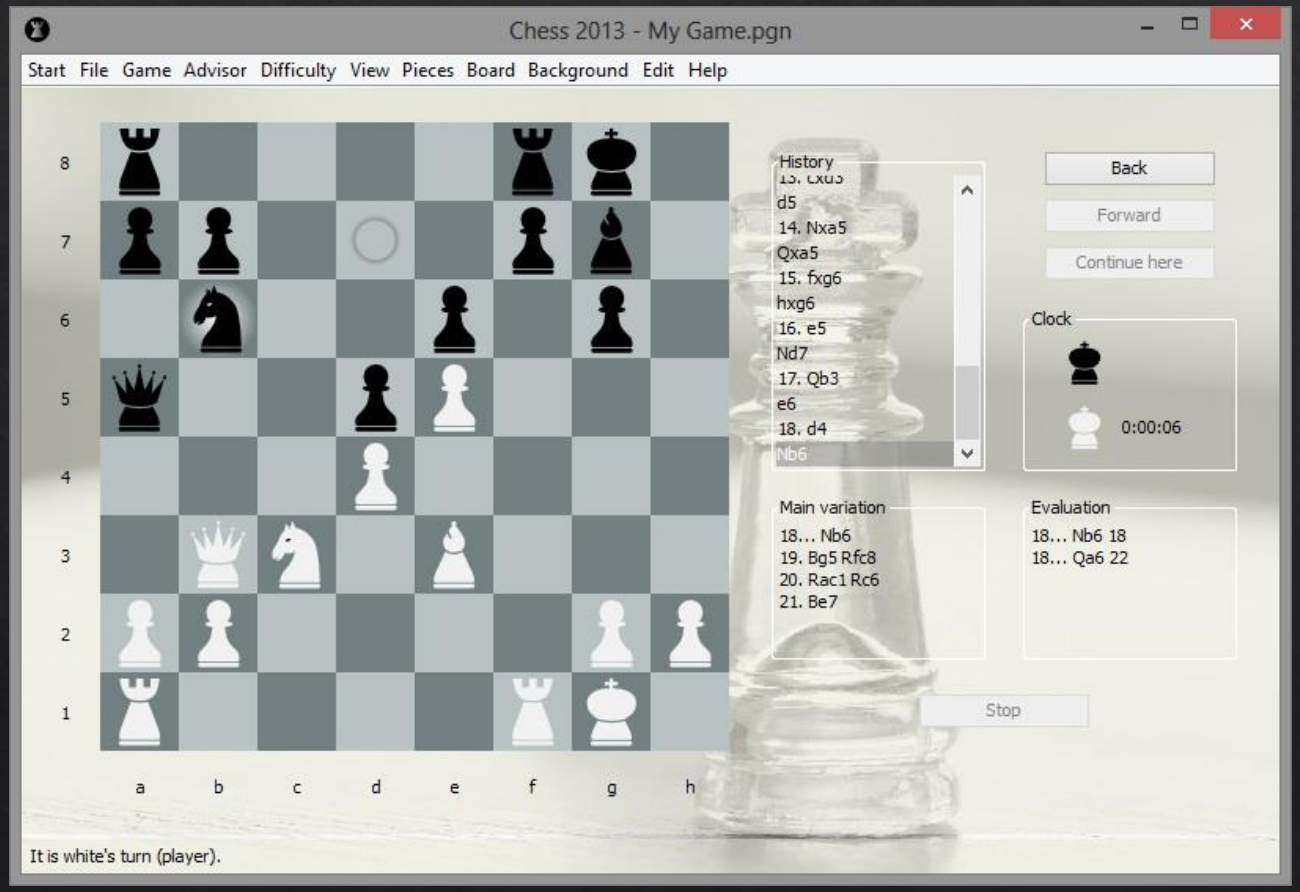
- This is an example of a Branch-and-Bound algorithm.
- This means that we bound the solution with the best already discovered path, and stop searching once this bound is surpassed.

Tic-Tac-Toe: Killer Heuristic

- Alpha-Beta works best when best moves are considered first.
  - Why?

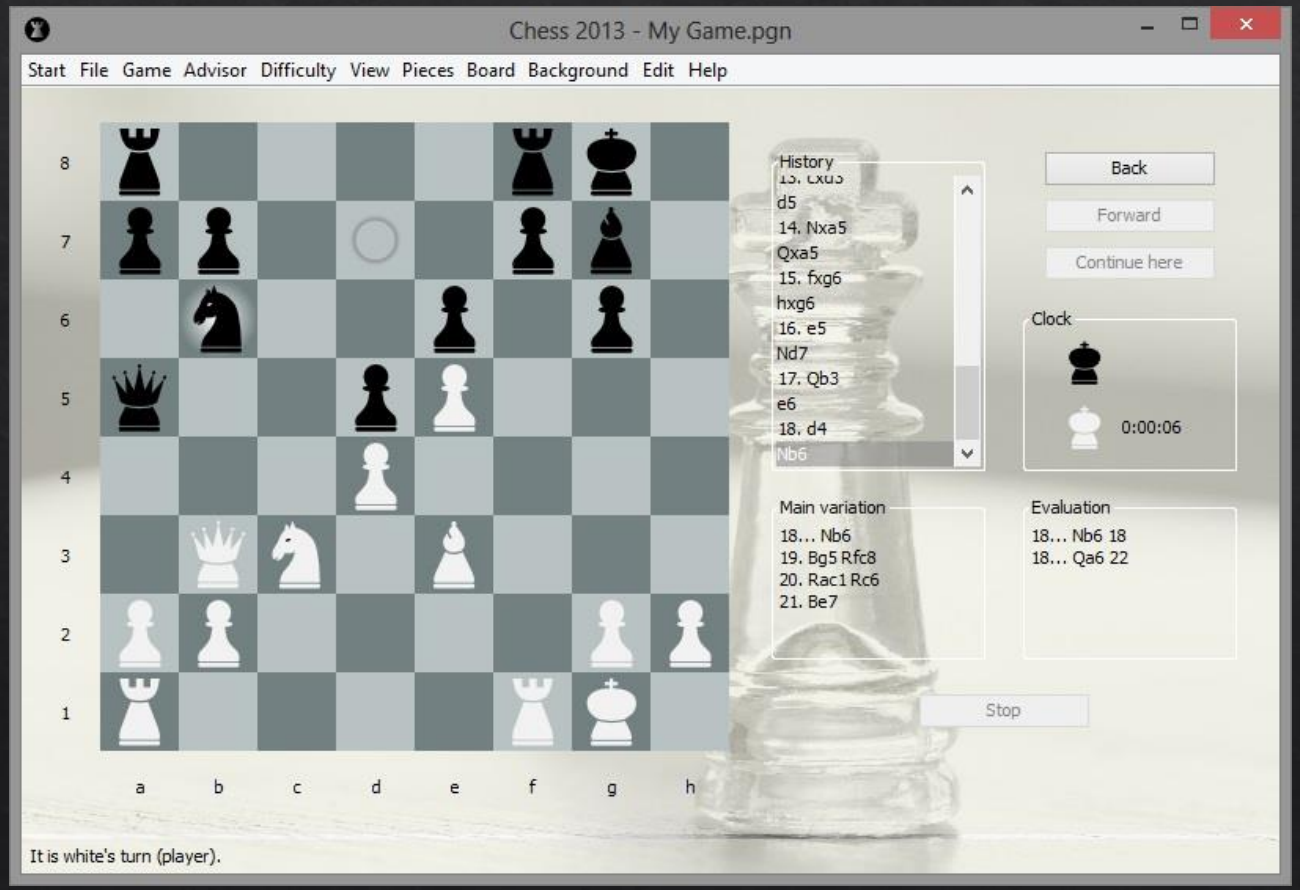
- Consider we are in a state where:
  - On a tic tac toe board, placing an X in the corner results in a win (eventually)
  - On another branch of the tree, we have a VERY similar board…is it more likely another X in the corner will be a good move?

Chess: Minimax Procedure



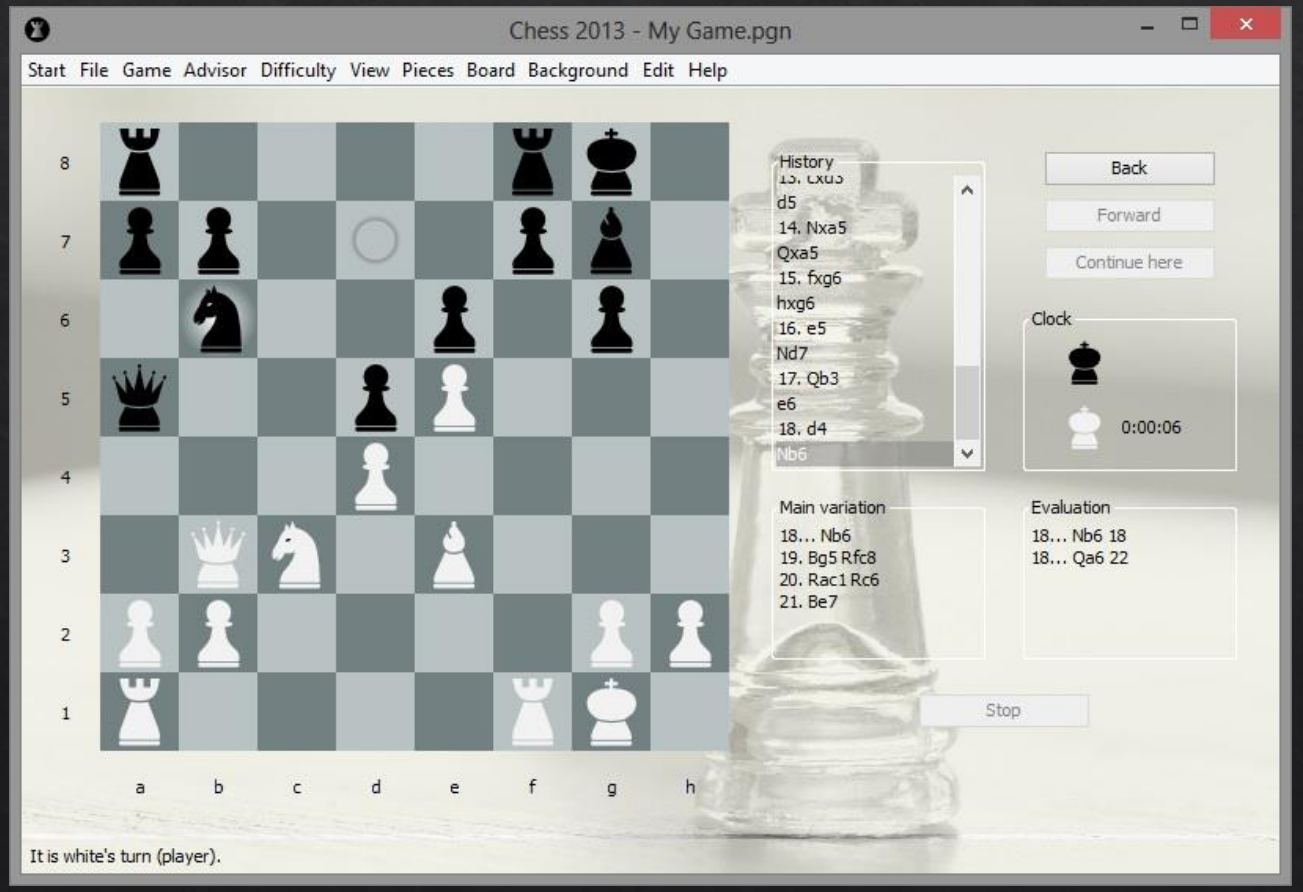
- Discussion!
- What to do if the search tree is HUGE!
- Chess has a very large search space
  - b > ~20 or so
  - d >= ~50 'ish

Chess



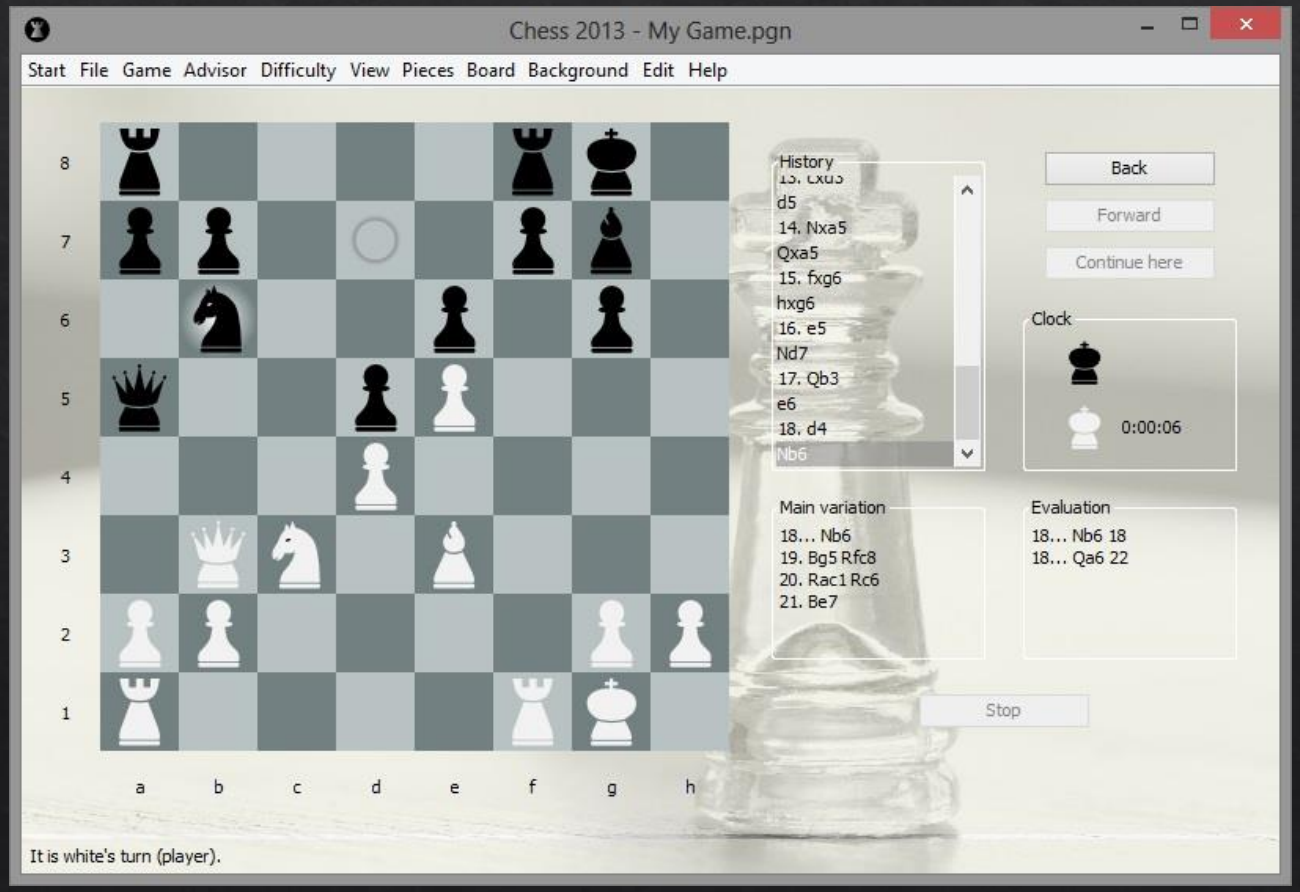
- Minimax (for all but simplest search trees) is intractable.
- So…we cut off search at some nodes and estimate utility using a heuristic evaluation function.
- Ideally, this function is very close to the actual utility you would calculate at that node.

Chess



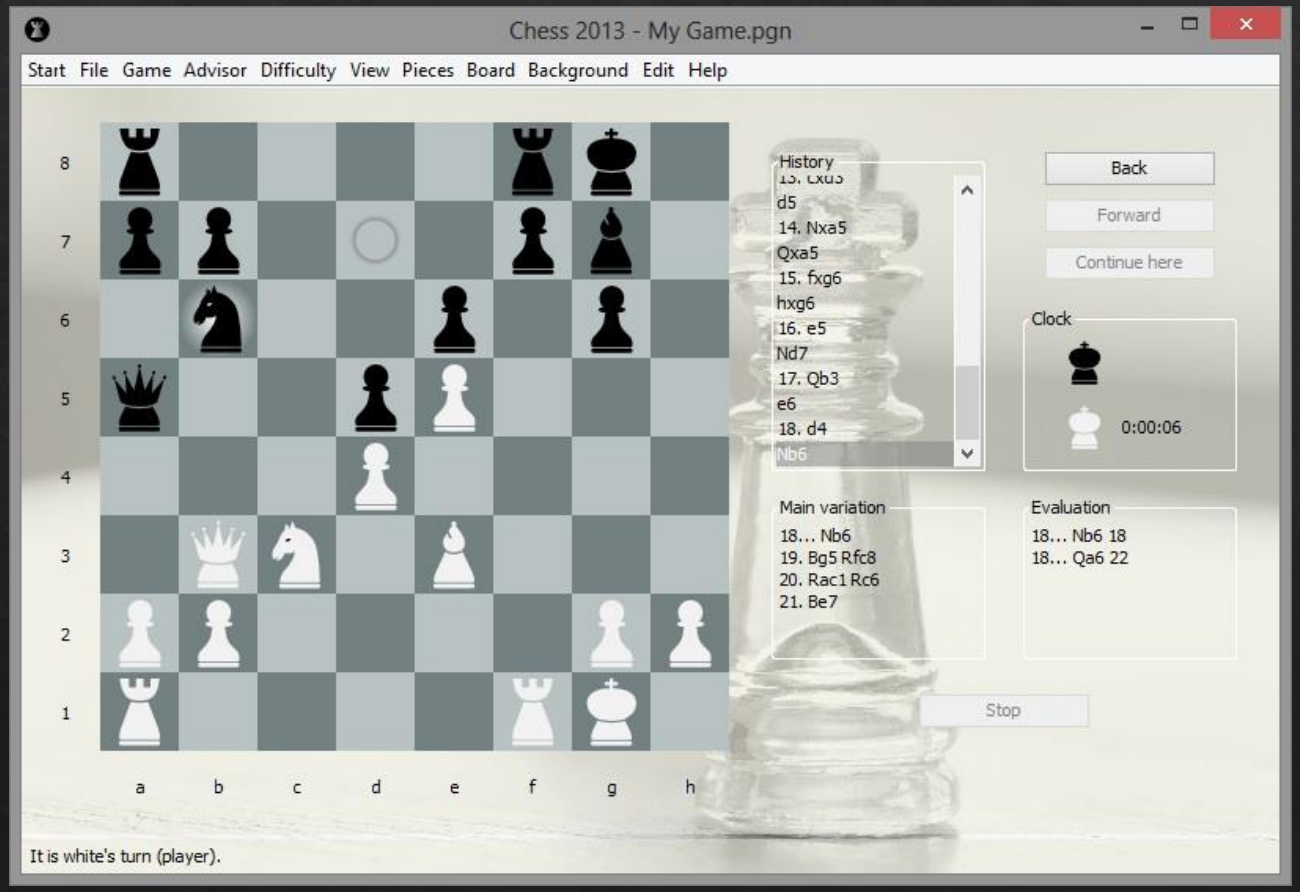
- Example Chess Heuristic:
  - Give values to pieces:
    - Pawn=1, knight/bishop=3, etc.
  - Score board with weighted function:
    - w1*f1 + w2*f2 + … + wn*fn
- Not the best heuristic but is a start.

Chess



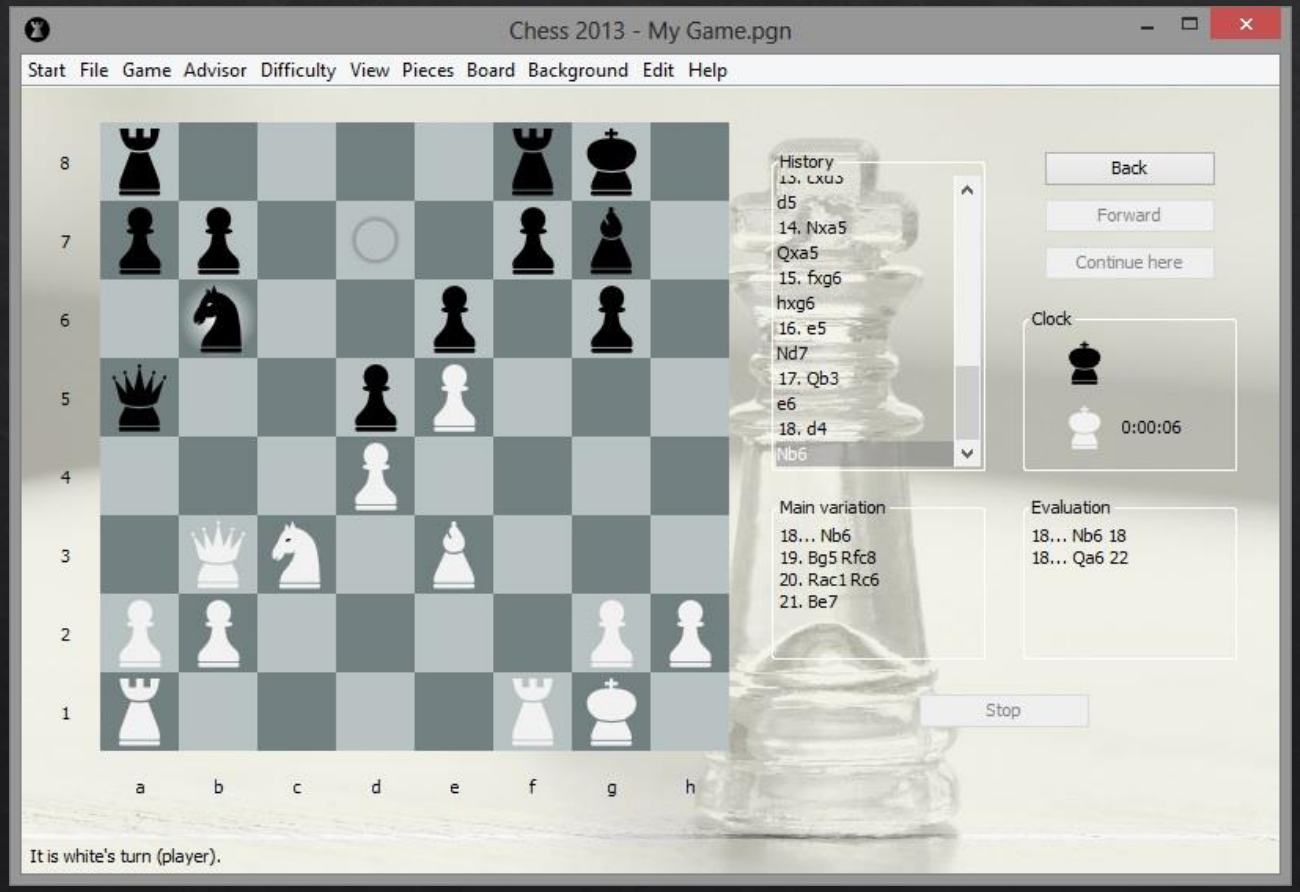
- Where to cut off the search?
- Normally we do depth limited search to some depth ‘d’.
- Those nodes at depth ‘d’ are estimated and we use minimax from there.
- Use iterative deepening to try to get further and further down search tree.

Chess



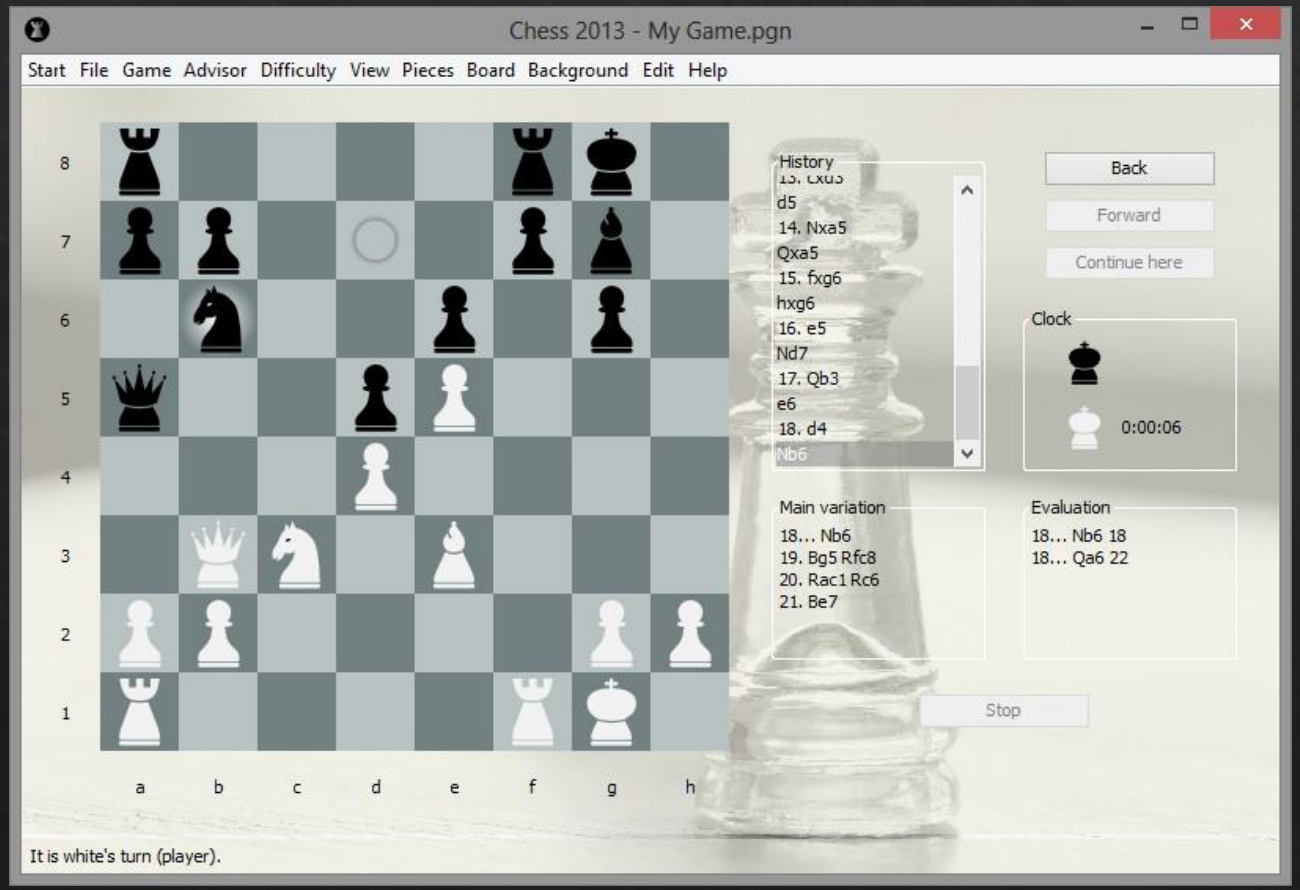
- In chess, we have certain states that are either **dynamic** or **quiescent**
- **Quiescent**: State is "calm" and no important pieces will be taken soon. I.e., the next move won't be a huge turning point in the game.
- **Dynamic**: In one or two more turns, a big turning point could occur

Chess



- In chess, we have certain states that are either **dynamic** or **quiescence**
- This leads to the **Horizon Problem**, where you search to a dynamic state and the moves at depth d+1, d+2 lead to your queen getting taken!

Chess



- Chess has some pretty standard opening and closing sequences…
- We could use expert game player moves as a heuristic for the first ‘x’ moves and last ‘y’ moves.
- Sounds a little like an expert system! Slightly hard-coded, but makes sense
- Will improve performance

Best Game Playing Systems

- Chess: Deep Blue
  - Beat world champion
- Checkers:
  - Samuel's learning program eventually beat developer
- Othello: Programs beat the best players
- Backgammon: Neural-net learning program (will see these later)
- Go: Branching factor of ~360 kills most search methods. Best programs mediocre

Conclusions

AI Problems Solved by Search

- Local Search
  - Characteristics: Path to solution doesn't matter. Concept of neighboring states is nebulous.
    - Simulated Annealing, Genetic Algorithms are good techniques for these.
- Classical Search
  - Path to solution does matter. Neighboring states is usually well-defined
    - BFS / DFS. Iterative deepening for large search spaces. A* with heuristics works well for many problems.
- Game Playing Systems
  - Similar to graph search but needs to account for game elements like scores, etc.
  - Minimax is an excellent starting algorithm
  - When search space is large, need heuristics, expected value functions, etc.

AI Problems Solved by Search

- We certainly have not studied all of the search algorithms related to AI.
- However, those presented give a good introduction to the ideas and issues related to most AI problems solved with search.
- At the end of the day, most interesting AI search problems boil down to finding good, relevant heuristics that enable an algorithm to shrink the size of the search space while still making good decisions / finding good solutions.