Artificial Intelligence **(Harvard CS182, Fall 2015)**

Lecture Notes on Game Playing

Alexander Rush

Contents

1 Introduction

Game playing is one of the earliest pursuits of AI researchers. John McCarthy's students were already working on chess in 1959. Research into Chess culminated in 1997 when IBM's Deep Blue beat Gary Kasparov, becoming the first computer program to defeat the reining grand champion. Today you can download programs that are world-class at Chess even on standard hardware.

However there are still many open challenges in adversarial search and game playing. XKCD is keeping score:

In this lecture we will use examples from Tic-Tac-Toe, later in the term we will talk about Jeopardy and StarCraft. Backgammon is also a particularly interesting example that we'll discuss briefly. If you're interested MIT has run a great IAP tournaments in the past focusing on Poker and StarCraft. There used to be a substantial reward for solving Go. However, Seven Minutes in Heaven is beyond the scope of this course.

2 Game Playing

We will be treating game playing as a search problem using a similar abstract specification. However there are several additional elements that we need to include.

2.1 Game Model

Game playing (GP) differs from search in that we will assume there are two game players Min and Max. Also unlike search where there was a cost associated with each action, in game playing there is a single **utility** that is only given at **terminal** (end) states. The Min player is aiming to minimize utility and the Max player to maximize utility. We assume that the game is **zero-sum**, i.e. at any terminal state the utility for Min and Max sums to zero, and **perfect information**, i.e. both players see the full state of the game.

Name (AIMA)	Type	Description
State space	$\mathcal{S}$	All possible states of the game, e.g. unique board positions
Action space	$\mathcal{A}$	All possible actions of the game i.e. moves
Actions	$\textsc{Actions} : \mathcal{S} \mapsto 2^{\mathcal{A}}$	Actions are applicable at a given state, i.e. available moves.
Transition model	$\textsc{Result} : \mathcal{S} \times \mathcal{A} \mapsto \mathcal{S}$	Update the state after a move.
Initial state	$S_0 \in \mathcal{S}$	The starting state of the game.
Player	$\textsc{Player} : \mathcal{S} \to \{\text{Min}, \text{Max}\}$	Function indicating the current turn.
Terminal test	$\textsc{Terminal} : \mathcal{S} \mapsto \{0, 1\}$	Predicate indicating if this state is terminal, i.e. is the game over.
Utility	$\textsc{Utility} : \mathcal{S} \to \mathbb{R}$	The utility at a (terminal) state.

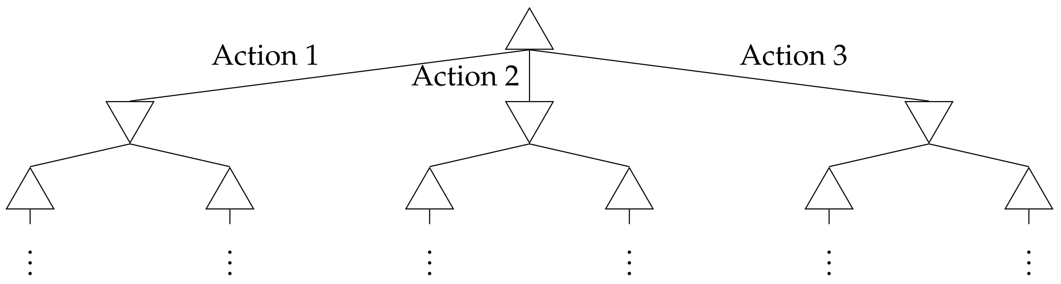
2.2 Minimax

The key challenge of adversarial search is that the agent does not control the decisions of its opponent. Therefore to behave rationally i.e. to minimize or maximize utility, the agent must model its opponents decisions.

Generally we do this by assuming that the opponent will behave rationally as well, so if we are minimizing utility, we assume the opponent will act to maximize their utility. This leads to the Minimax formula for computing the utility of a nonterminal game state.

$$\textsc{Minimax}(s) = \begin{cases} ❂ & \text{if } \textsc{Terminal}(s) \\ ❂ & \text{if } \textsc{Player}(\textsc{s}) = \text{Max} \\ ❂ & \text{if } \textsc{Player}(\textsc{s}) = \text{Min} \end{cases}$$

We can expand this function into a data-structure known as a **game tree**. Each node in this tree represents a move by Max or Min. The triangles pointing up indicate that the node is maximizing over its children, and triangles pointing down indicate that the node is minimizing over its children.



2.3 Example: Tic-Tac-Toe

Let's now consider one of the simplest zero-sum, perfect-information games, Tic-Tac-Toe. Consider the model for the following game state:

O		
O	X	X
X	X	O

Question 1 *What is the model state at this stage of tic-tac-toe?*

Name	Type	Description
Actions	❂	❂
Player	❂	❂
Terminal test	❂	❂
Utility	❂	❂

Next we can ask what is the utility that the Min player should assign to this state. We can do this by expanding out the recursive MINIMAX definition. This gives:

$$\begin{aligned}\textsc{Minimax}(s) = \min\{ \quad & \max\{\textsc{Utility}(\textsc{Result}(\textsc{Result}(s, \text{OTop})), \text{XTopRight})\}, \\ & \max\{\textsc{Utility}(\textsc{Result}(\textsc{Result}(s, \text{OTopRight})), \text{XTop})\}\}\end{aligned}$$

Since either way O plays, it leads to an X win, this utility of this state is 1.

Question 2 *Can we also write the same calculation as a game tree?*

Now let's consider a slightly more interesting position from X's perspective

O		
X	O	
X	O	X

Question 3 *Let's write the same calculation as a game tree.*

The max player (X) can at best play for a draw here. Clearly the right play is XTop which gives the utility of 0.

Note though that for these problems we are only considering a branching factor of at most 3, and a depth of 2. In practice the tree grows very large, very quickly. AIMA has a nice graphic showing a partial expansion of a game tree for tic-tac-toe from scratch.

3 Search for Game Playing

As with standard search, we have to decide how to enumerate possible game states in order to find a path to take. We do this by implementing search over the Minimax game tree.

In particular, we utilize uninformed depth-first tree search where are terminal states are goal states.

Question 4 *Why do we not need to use graph search for games? Why not use uniform-cost search?*

❂

In this situation, DFS has clear advantages over BFS in terms of memory-efficiency. Because of the nature of game playing, we do not have to worry about maintaining the costly expand list. Note also that instead of explicitly using a stack, we can implement DFS using recursion. Each recursive call to MINIMAX simulates a pushing and popping from the stack. This leads to a very simple implementation of Minimax Search.

```
procedure MINIMAX($s$)
    if TERMINAL($s$) then return UTILITY($s$)
    else if PLAYER($s$) = Max then
        $v \leftarrow$ ❂
        for $a \in$ ACTIONS($s$) do
            ❂
    else if PLAYER($s$) = Min then
        $v \leftarrow$ ❂
        for $a \in$ ACTIONS($s$) do
            ❂
    return $v$
```

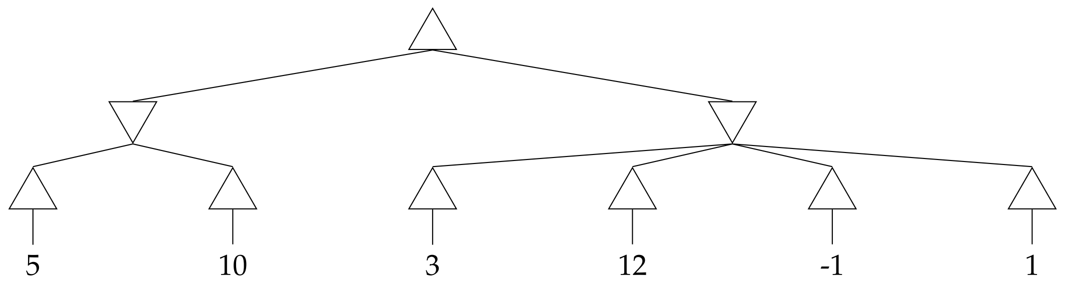
Question 5 *What is the space and time complexity of this algorithm?*

❂

3.1 Alpha-Beta Pruning

We can further speed up game-playing search by exploiting some properties about the game tree. One nice property of the max (min) operator, is that if we know after seeing i elements of the set that known of the future elements will be greater (less) then it is not necessary to compute these values at all. We can therefore **short circuit** the computation and return.

Consider the following example:



Question 6 *Go through this example and calculate the value of each state of the game tree top-down, left-to-right. What is the minimax value of the top state?*

Question 7 *If instead of* -1 *the utility was* -1000*, would anything change? What if instead* 5 *was equal to* 2*?*

What happened in this case was that the already computed value 5 already dominated the value of 3 in the min. Since 3 is already lower than 5 there is no need to keep on calculating the rest of the values.

Alpha-beta pruning exploits this idea to speed up Minimax computation without changing the final result. Formally this comes up when where there is a min embedded inside a max:

$$\max\{\alpha, \ldots, \min\{\ldots, v, \ldots, v'\}\}$$

Let's say we have already computed the value of α and v, but are deciding whether to compute the rest of the min including v'. Consider the two cases:

1. $v \geq \alpha$; the inner min may end with a value higher than α
2. $v \leq \alpha$; the inner min will never affect the outer max, so calculating the value of v' tells us no new information.

In our examples the **alpha** value was $\alpha = 5$ and $v = 3$. Since we are in the second case there is no need to keep calculating values.

We can generalize this idea to any of the outer max's:

$$\max\{\ldots, \alpha, \ldots, \min\{\max\{\ldots \min\{\ldots, v, \ldots, v'\}\}\}, \ldots\}$$

If there is a future value v' with $v' \leq v$ it may be selected by the inner min. However we know $v' < v \leq \alpha$ so it will not be selected by the outer max. The bigger α we have, the more likely we can apply this pruning, so we maintain the largest α seen at ancestor max node.

We can also reverse this trick for min nodes by maintaining the smallest β value:

❂

The modified Minimax search algorithm is shown below. The upside of this method is that it yields a significant speed-up in practice with very minimal bookkeeping and no effect on the result. A rare combination.

```
Require: $\alpha$ is highest value of ancestor max-node, $\beta$ is lowest value of ancestor min-node
1: procedure AlphaBetaMinimax($s$, $\alpha$, $\beta$)
2:     if Terminal($s$) $= 1$ then return Utility($s$)
3:     else if Player($s$) $=$ Max then
4:         $v \leftarrow$ ❂
5:         for $a \in$ Actions($s$) do
6:             $v \leftarrow$ ❂
7:             if $v \geq \beta$ then return $v$
8:             $\alpha \leftarrow \max\{\alpha, v\}$
9:     else if Player($s$) $=$Min then
10:        $v \leftarrow$ ❂
11:        for $a \in$ Actions($s$) do
12:            $v \leftarrow$ ❂
13:            if ❂ then return $v$
14:            ❂
15:    return $v$
```

3.2 Real-Time Decisions

While alpha-beta pruning does give a speed-up, most games are still much too large to compute Minimax in practice. Chess for instance has a average branching factor of 35 and an average depth of 40 (apparently there was a 277 move game once). Of course $O(35^{40})$ is cartoonishly large:

5790576106764117854019273308244010877388063818216323852539062 5

In practice game playing systems use either a depth-limited search approach or an iterative deepening approach. For both, a depth limit is selected and search is cutoff at this depth.

$$\text{H-Minimax}(s,d) = \begin{cases} \text{Utility}(s) & \text{if Terminal}(s) \\ h(s) & \text{if } d > \text{limit} \\ \max_{a \in \text{Actions}(s)} \text{H-Minimax}(\text{Result}(s,a), d+1) & \text{if Player(s)} = \text{Max} \\ \min_{a \in \text{Actions}(s)} \text{H-Minimax}(\text{Result}(s,a), d+1) & \text{if Player(s)} = \text{Min} \end{cases}$$

However, unlike in the standard search setup, GP only receives a score when it reaches a terminal state. To handle depth limited cases, we need a function $h : \mathcal{S} \mapsto \mathbb{R}$ to award a heuristic utility to the state. As with general search, coming up with good game-specific heuristics is a very challenging problem but crucial to the construction of effective and practical GP agents.

3.3 Heuristics

Generally game playing heuristics estimate how good a position is for each player. Note that unlike search heuristics we will not make any attempt to maintain admissibility or consistency here, although similar conditions do exist.

Perhaps the simplest heuristic is the value system taught to beginners learning to play chess. This system simply assigns value to the **material remaining** on the board.

However, note that because this is just an approximation and ignores so much of the other complications of chess, there have been many, many other heuristics for this problem. (Wikipedia is amazing.) I am not a chess player, but it is pretty remarkable to me that not even the relative values of bishop and knight are consistent.

Another method that is used is brute force calculation of certain positions. In chess this often means utilizing an end-game solver to play out the results of a large set of standard positions (such as a kings and pawns ending) . Given that each extra depth is a multiplicative cost, having an endgame database can turn a failed search into a successful one.

3.4 Expecti-Max

So far we have consider only deterministic games with perfect information. We will end by discussing games like Backgammon, which have perfect information but incorporates randomness. Each turn in backgammon consists of two steps:

- Roll two dies
- Select checkers to move based on dice values.

Our focus will be on the question of how to model our opponent Min's action. Since we don't know what value they will roll, we could be pessimistic and assume that they get the best possible roll (min over rolls) or optimistic assume they get the worst possible roll (max over rolls).

Instead, we will directly model the randomness of the roll. We treat the dice roll as a third play Exp and the value of the dice as an action a. Given the state of the world s, we say the probability of Exp taking action a is given as $P(a|s)$.

In the case of Backgammon, there are 36 possible actions corresponding to each pair of dice rolls, each with probability $\frac{1}{36}$. The turn order then alternates: Exp, Min, Exp, Max, Exp, Min, ... Once we explicitly incorporate the last roll into the state s, the moves of Max and Min become deterministic.

Why do we call this third player Exp? Well, similar to Max and Min, nodes in the game tree associated with Exp will take the **expectation** over children nodes. Define the expectation of a function f with respect to a random variable x as:

$$\mathbb{E}_{x|y}[f(x)] = \sum_{x} p(x|y) f(x)$$

In the case of the utility of a state, this gives us a method for computing the expected utility of an Exp state.

$$\mathbb{E}_{a|s}[\textsc{Utility}(\textsc{Result}(s,a))] = \sum_{a \in \textsc{Actions}(s)} p(a|s)\textsc{Utility}(\textsc{Result}(s,a))$$

This formula can then be directly plugged into minimax, to yield an algorithm known as expectimax

$$\text{E-}\textsc{Minimax}(s) = \begin{cases} \text{❂} & \text{if } \textsc{Terminal}(s) \\ \text{❂} & \text{if } \textsc{Player(s)} = \text{Max} \\ \text{❂} & \text{if } \textsc{Player(s)} = \text{Min} \\ \text{❂} & \text{if } \textsc{Player(s)} = \text{Exp} \end{cases}$$