The Minimax Algorithm

Arthur Ryman, arthur.ryman@gmail.com

August 3, 2021

Abstract

This article formalizes the minimax algorithm for playing 2-person games. Each player searches graph of moves for an optimal move. This article also formalizes the alpha-beta pruning heuristic for speeding up the search.

1 Games

For the purposes of this article, a game is an activity engaged in by one or more players who attempt to win by making moves that lead to a winning state.

1.1 Game States *State*

Let *State* denote the set of all game states.

$$[State]$$

1.2 The Game is Finite and Non-Empty *game_is_finite*

We assume that there are a finite, non-zero number of game states. Let *game_is_finite* denote this constraint.

$$State \in \mathbb{F}_1\, State$$

1.3 Start State *start*

The game has a unique, distinguished starting state. Let *start* denote the starting game state.

$$start : State$$

1.4 Moves *moves*

A player can select from zero or more moves that are available in any given game state. Let *moves* denote the binary relation on the set of game states that relates a given game state to another if and only if the player can move from the given game state to the other.

$$moves : State \leftrightarrow State$$

1.5 Children *children*

The set of all games states that can be reached in one move from a given state are called the children of that state. Let *children* denote the mapping from a game state to its children.

$$children == (\lambda\, p : State \bullet \{\, c : State \mid p \mapsto c \in moves \,\})$$

1.6 Terminal States *terminal*

A state that has no children is called a terminal state. Let *terminal* denote the set of all terminal states.

$$terminal == \{\, x : State \mid children(x) = \varnothing \,\}$$

The game terminates when a terminal state is reached.

1.7 Terminal Score *terminal_score*

Every terminal state is assigned a numeric score. Each player makes moves that they think will lead to a terminal state that optimizes their score.

$$terminal\_score : terminal \longrightarrow \mathbb{Z}$$

1.8 Parents *parents*

The set of all games states that a given game state can be reached from in one move are called the parents of the given state. Let *parents* denote that mapping from a game state to its parents,

$$parents == (\lambda\, c : State \bullet \{\, p : State \mid p \mapsto c \in moves \,\})$$

1.9 Initial States *initial*

A state that has no parents is called an initial state. Let *initial* denote the set of initial states.

$$initial == \{\, x : State \mid parents(x) = \varnothing \,\}$$

1.10 The Start State is an Initial State *start_is_initial*

We assume that the start state has no parents. Let *start_is_initial* denote this constraint.

$$start \in initial$$

1.11 Paths *paths*

A path is a sequence of two or more game states such each pair of successive states are related by a move.

$$\begin{array}{l} paths == \\ \quad \{\, p : \mathrm{seq}\, State \mid \#p \geq 2 \land \\ \quad\quad (\forall\, i : 1 \,..\, \#p - 1 \bullet \\ \quad\quad\quad p(i) \mapsto p(i+1) \in moves) \,\} \end{array}$$

1.12 The Game is Connected *game_is_connected*

We assume that every game state, other than the start state, is connected to the start state by a path. Let *game_is_connected* denote this constraint.

$$\begin{array}{l} \forall\, s : State \mid s \neq start \bullet \\ \quad \exists\, path : paths \bullet \\ \quad\quad path(1) = start \land \\ \quad\quad path(\#path) = s \end{array}$$

1.13 Cycles *cycles*

A path that begins and ends on the same state is called a cycle. Let *cycle* denote the set of all cycles.

$$cycles == \{\, p : paths \mid p(1) = p(\#p) \,\}$$

1.14 The Game Has No Cycles *game_is_acyclic*

We assume that the game has no cycles. Let *game_is_acyclic* denote this constraint.

$$cycles = \varnothing$$

2 2-Person Games

Games such as tic-tac-toe and chess are played by two players who take turns moving.

2.1 Players *Player*

Let *Player* denote the set of players.

$$Player ::= A \mid B$$

The players are denoted by A and B who we can think of as being Alice and Bob.

2.2 Who Moves *player*

The players take turns moving. The game state determines who moves next. Let *player* denote the mapping from the game state to the player who moves next.

$$player : State \longrightarrow Player$$

2.3 Player A Starts *player_A_starts*

Without loss of generality, we can assume that Alice moves first. Let *player_A_starts* denote this constraint.

$$player(start) = A$$

2.4 Players Alternate *players_alternate*

The players take turns moving. Let *players_alternate* denote this constraint.

$$\forall\, x, y : State \mid x \mapsto y \in moves \bullet player(x) \neq player(y)$$

2.5 Optimal Score *optimal_score*

We assume that the players are playing for money or some other fungible objects. The terminal score of a terminal game state is the amount that Alice wins and Bob loses. Equivalently, the negative of a terminal score is the amount that Bob wins and Alice loses. Each player tries to optimize their score. Thus, Alice tries to reach a terminal state that has the highest score while Bob tries to reach one with the lowest score.

We can therefore define the optimal score for each state by working backwards from the terminal states. Let *optimal_score* denote the function that assigns the optimal score to each state.

$$
\begin{array}{|l}
optimal\_score : State \longrightarrow \mathbb{Z} \\
\hline
\forall\, x : terminal \bullet \\
\qquad optimal\_score(x) = terminal\_score(x) \\
\forall\, x : State \setminus terminal \mid player(x) = A \bullet \\
\qquad optimal\_score(x) = max\{\, y : children(x) \bullet optimal\_score(y) \,\} \\
\forall\, x : State \setminus terminal \mid player(x) = B \bullet \\
\qquad optimal\_score(x) = min\{\, y : children(x) \bullet optimal\_score(y) \,\}
\end{array}
$$

This axiomatic description is recursive. The dependency graph of *optimal_score* is the same as the game moves graph. However, the game is acyclic so we can compute *optimal_score* bottom-up using a topological sort of the game graph. Therefore, *optimal_score* is uniquely defined.