COMP111: Artificial Intelligence

Section 6. Adversarial search (Game playing)

Frank Wolter

Outline

We will look at how search can be applied to playing games

- Types of Games
- Perfect play:
  - minimax algorithm
  - α–β pruning
- Playing with limited resources (heuristics)

Games vs. search problems

- In search we make all moves. In games we play against an unpredictable opponent:
  - solution is a strategy specifying a move for every possible move of the oponent.

Games vs. search problems

- In search we make all moves. In games we play against an unpredictable opponent:
  - solution is a strategy specifying a move for every possible move of the oponent.
- A method is needed for selecting good moves that stand a good chance of achieving a winning state whatever the opponent does!

Games vs. search problems

- In search we make all moves. In games we play against an unpredictable opponent:
  - solution is a strategy specifying a move for every possible move of the oponent.
- A method is needed for selecting good moves that stand a good chance of achieving a winning state whatever the opponent does!
- Because of combinatorial explosion, in practice we must approximate using heuristics.

Types of Games

- In some games we have a fully observable environment. The position is known completely. These are called games with perfect information.
- Examples: chess, go, backgammon, monopoly.

Types of Games

- In some games we have a fully observable environment. The position is known completely. These are called games with perfect information.
- Examples: chess, go, backgammon, monopoly.
- In others we have a partially observable environment. For example, we cannot see the opponents cards. These are called games with imperfect information.
- Examples: battleships, bridge, poker.

Types of Games

- In some games we have a fully observable environment. The position is known completely. These are called games with perfect information.
- Examples: chess, go, backgammon, monopoly.
- In others we have a partially observable environment. For example, we cannot see the opponents cards. These are called games with imperfect information.
- Examples: battleships, bridge, poker.
- Some games are deterministic: chess, go.

Types of Games

- In some games we have a fully observable environment. The position is known completely. These are called games with perfect information.
- Examples: chess, go, backgammon, monopoly.
- In others we have a partially observable environment. For example, we cannot see the opponents cards. These are called games with imperfect information.
- Examples: battleships, bridge, poker.
- Some games are deterministic: chess, go.
- Others have an element of chance: backgammon, monopoly, bridge, poker

The games we consider

We consider special kinds of games

- Deterministic
- Two-player
- Zero-sum:
  - the utility values at the end are equal and opposite
  - example: one wins $(+1)$ the other loses (-1).
- Perfect information

Problem Formulation

The search graph gives for every state the successor states obtained by making a move. The set of goal states is replaced by a utility function.

Problem Formulation

The search graph gives for every state the successor states obtained by making a move. The set of goal states is replaced by a utility function.

- Initial state s_{start}:
  - Initial board position. Which player moves first.

Problem Formulation

The search graph gives for every state the successor states obtained by making a move. The set of goal states is replaced by a utility function.

- Initial state s_{start}:
  - Initial board position. Which player moves first.
- Successor function:
  - provides for every state s and move the new state after the move.

Problem Formulation

The search graph gives for every state the successor states obtained by making a move. The set of goal states is replaced by a utility function.

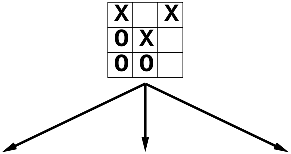
- Initial state s_{start}:
  - Initial board position. Which player moves first.
- Successor function:
  - provides for every state s and move the new state after the move.
- Terminal test
  - Determines when the game is over

Problem Formulation

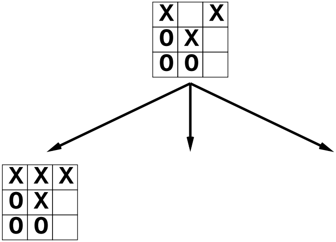
The search graph gives for every state the successor states obtained by making a move. The set of goal states is replaced by a utility function.

- Initial state s_{start}:
  - Initial board position. Which player moves first.
- Successor function:
  - provides for every state s and move the new state after the move.
- Terminal test
  - Determines when the game is over
- Utility function
  - Numeric value for terminal states
  - E.g. Chess +1, -1, 0
  - E.g. Backgammon +192 to -192

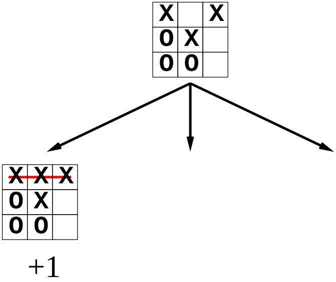
Possible Development



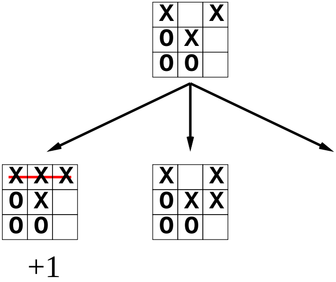
Possible Development



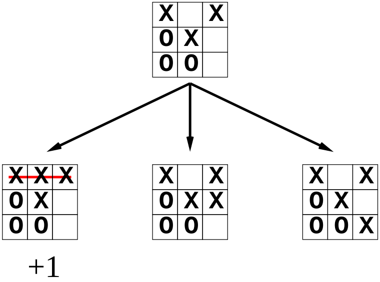
Possible Development



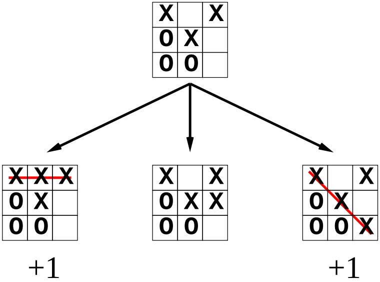
Possible Development



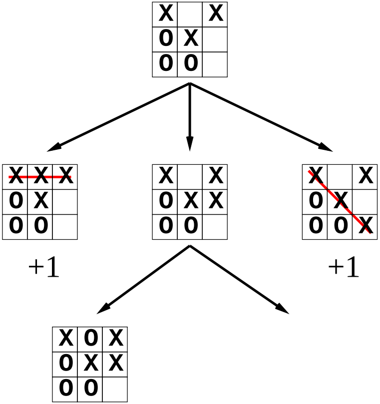
Possible Development



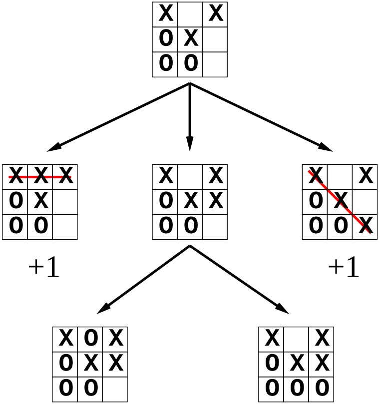
Possible Development



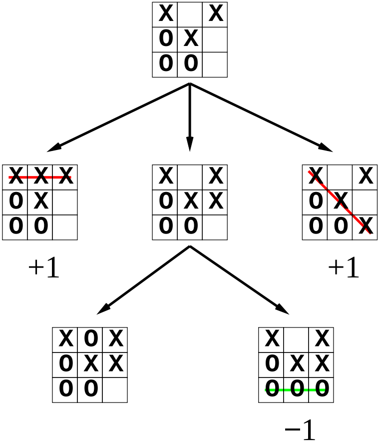
Possible Development



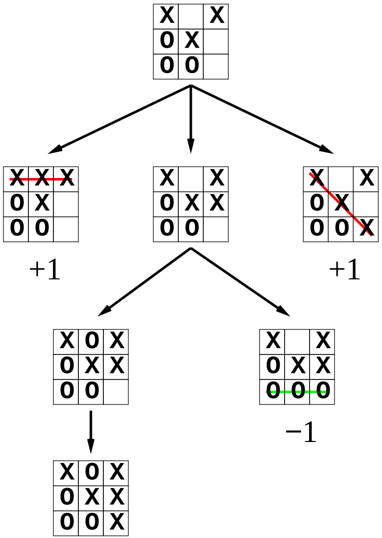
Possible Development



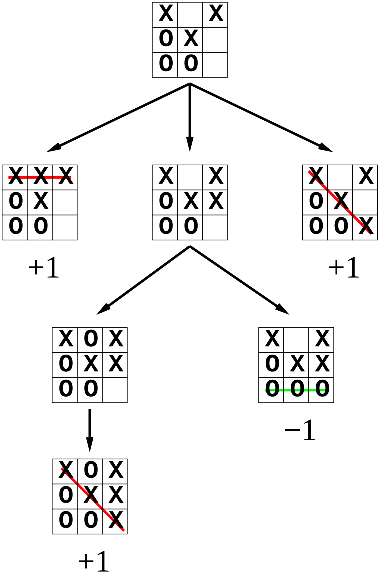
Possible Development



Possible Development



Possible Development



Game Tree

- Each level labelled with player to move
- Each level represents a ply
  - Half a turn
- Represents what happens with competing agents

Introducing MIN and MAX

MIN and MAX are two players:

- MAX wants to win (maximise utility)
- MIN wants MAX to lose (minimise utility for MAX)
- MIN is the opponent.

Introducing MIN and MAX

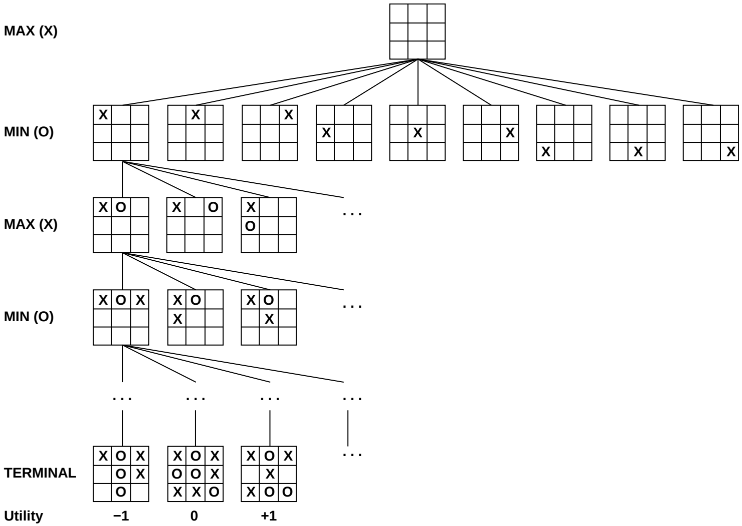
MIN and MAX are two players:

- MAX wants to win (maximise utility)
- MIN wants MAX to lose (minimise utility for MAX)
- MIN is the opponent.

Both players will play to the best of their ability

- MAX wants a strategy for maximising utility assuming MIN will do best to minimise MAX's utility
- Consider minimax value of each state: the utility of a state given that both players play optimally.

Example Game Tree



Minimax Value

- The utility (=minimax value) of a terminal state is given by its utility already (as an input).

Minimax Value

- The utility (=minimax value) of a terminal state is given by its utility already (as an input).
- The utility (=minimax value) of a MAX-state (when MAX moves) is the maximum of the utilities of its successor states.

Minimax Value

- The utility (=minimax value) of a terminal state is given by its utility already (as an input).
- The utility (=minimax value) of a MAX-state (when MAX moves) is the maximum of the utilities of its successor states.
- The utility (=minimax value) of a MIN-state (when MIN moves) is the minimum of the utilities of its successor states.

Minimax Value

- The utility (=minimax value) of a terminal state is given by its utility already (as an input).
- The utility (=minimax value) of a MAX-state (when MAX moves) is the maximum of the utilities of its successor states.
- The utility (=minimax value) of a MIN-state (when MIN moves) is the minimum of the utilities of its successor states.
- Thus, we can compute the minimax value recursively starting from the terminal states.

Minimax Value

- The utility (=minimax value) of a terminal state is given by its utility already (as an input).
- The utility (=minimax value) of a MAX-state (when MAX moves) is the maximum of the utilities of its successor states.
- The utility (=minimax value) of a MIN-state (when MIN moves) is the minimum of the utilities of its successor states.
- Thus, we can compute the minimax value recursively starting from the terminal states.

Formally, let $\mathrm{Succ}(s)$ denote the set of successors states of state s.

Minimax Value

- The utility (=minimax value) of a terminal state is given by its utility already (as an input).
- The utility (=minimax value) of a MAX-state (when MAX moves) is the maximum of the utilities of its successor states.
- The utility (=minimax value) of a MIN-state (when MIN moves) is the minimum of the utilities of its successor states.
- Thus, we can compute the minimax value recursively starting from the terminal states.

Formally, let $\mathsf{Succ}(s)$ denote the set of successors states of state s. Define the function $\mathsf{MinimaxV}(s)$ recursively as follows:

$$\mathrm{MinimaxV}(s) = \begin{cases} \mathrm{Utility}(s) & s \text{ is Terminal} \\ \max_{n \in \mathrm{Succ}(s)} \mathrm{MinimaxV}(n) & \text{MAX moves in } s \\ \min_{n \in \mathrm{Succ}(s)} \mathrm{MinimaxV}(n) & \text{MIN moves in } s \end{cases}$$

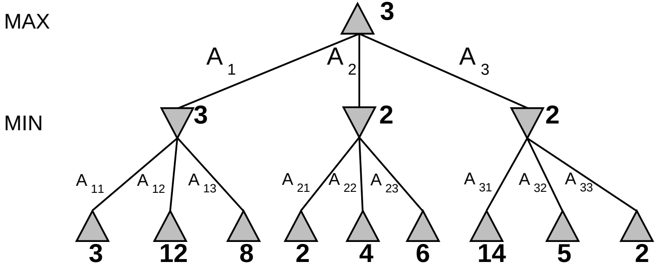
Minimax algorithm

- Calculate minimax value of each state using the equation above starting from the terminal states.
- Game tree as minimax tree:

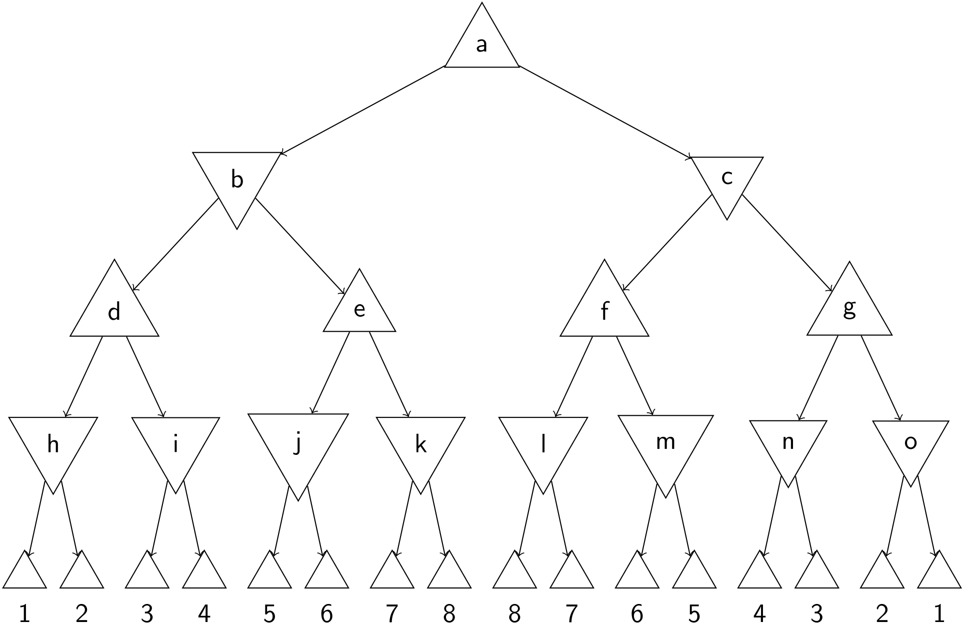
Minimax Tree

- MIN takes the minimal value from its successors.
- MAX takes the maximal value from its successors.

Consider



What is the minimax value of *a*?



Properties of minimax

Assume MAX always move to the state with the maximal minimax value.

- Optimal: against an optimal opponent. Otherwise MAX will do even better. There may, however, be better strategies against suboptimal opponents.

Properties of minimax

Assume MAX always move to the state with the maximal minimax value.

- Optimal: against an optimal opponent. Otherwise MAX will do even better. There may, however, be better strategies against suboptimal opponents.
- Time complexity: can be implemented (depth-first) so that time complexity is b^m (branching factor b, depth m).

Properties of minimax

Assume MAX always move to the state with the maximal minimax value.

- Optimal: against an optimal opponent. Otherwise MAX will do even better. There may, however, be better strategies against suboptimal opponents.
- Time complexity: can be implemented (depth-first) so that time complexity is b^m (branching factor b, depth m).
- Space complexity: can be implemented (depth-first) so that space complexity is bm.

Properties of minimax

Assume MAX always move to the state with the maximal minimax value.

- Optimal: against an optimal opponent. Otherwise MAX will do even better. There may, however, be better strategies against suboptimal opponents.
- Time complexity: can be implemented (depth-first) so that time complexity is b^m (branching factor b, depth m).
- Space complexity: can be implemented (depth-first) so that space complexity is bm.

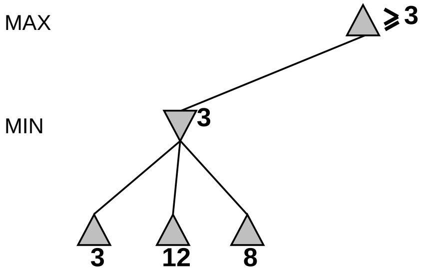
For chess, $b \approx 35$, $m \approx 100$ for “reasonable” games

- 10^{154} paths to explore
- infeasible

But do we need to explore every path?

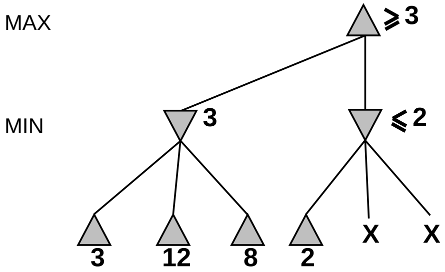
Removing redundant information: α-β-Pruning

If you know half-way through a calculation that it will succeed or fail, then there is no point in doing the rest of it!



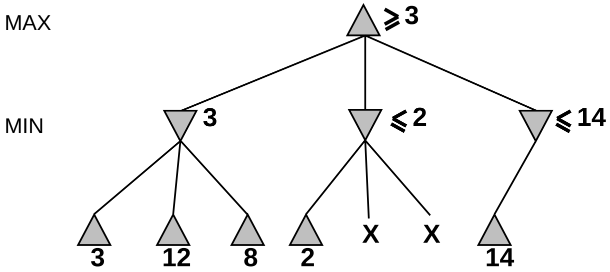
Removing redundant information: α-β-Pruning

If you know half-way through a calculation that it will succeed or fail, then there is no point in doing the rest of it!



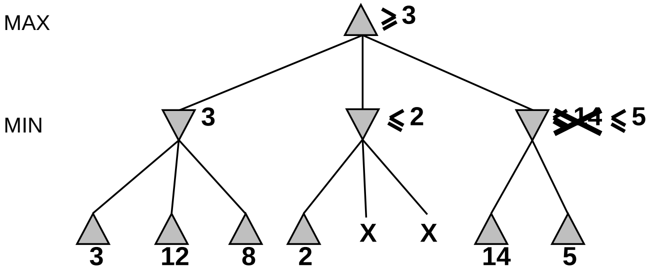
Removing redundant information: α-β-Pruning

If you know half-way through a calculation that it will succeed or fail, then there is no point in doing the rest of it!



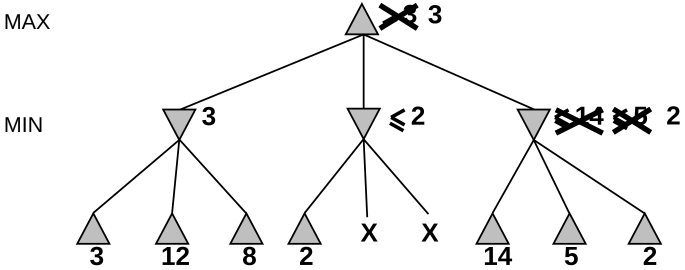
Removing redundant information: α-β-Pruning

If you know half-way through a calculation that it will succeed or fail, then there is no point in doing the rest of it!



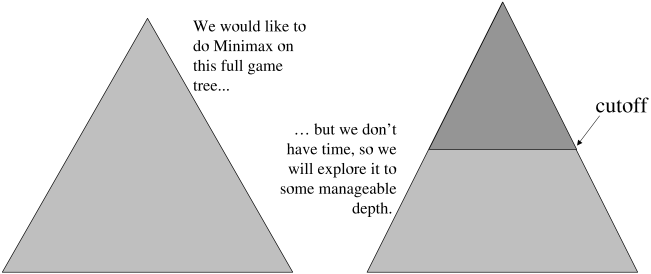
Removing redundant information: α-β-Pruning

If you know half-way through a calculation that it will succeed or fail, then there is no point in doing the rest of it!



Cutoffs and Heuristics

- Cutoff search according to some cutoff test.
- Problem: utitilies are defined only at terminal states.
- Solution: Evaluate the pre-terminal leaf states using heuristic evaluation function rather than using the actual utility function.



Cutoff Value

Instead of MiniMaxV(s) we compute CutOffV(s).

Cutoff Value

Instead of MiniMaxV(s) we compute CutOffV(s).
Assume that we can compute a function Evaluation(s) which gives us a utility value for any state s which we do not want explore (every cutoff state).

Cutoff Value

Instead of MiniMaxV(s) we compute CutOffV(s).
Assume that we can compute a function Evaluation(s) which gives us a utility value for any state s which we do not want explore (every cutoff state).
Then define CutOffV(s) recursively:

$$\mathrm{CutoffV}(s) = \begin{cases} \mathrm{Utility}(s) & s \text{ is Terminal} \\ \mathrm{Evaluation}(s) & s \text{ is Cutoff} \\ \max_{n \in \mathrm{Succ}(s)} \mathrm{CutoffV}(n) & s \text{ is MAX} \\ \min_{n \in \mathrm{Succ}(s)} \mathrm{CutoffV}(n) & s \text{ is MIN} \end{cases}$$

Example: Chess (I)

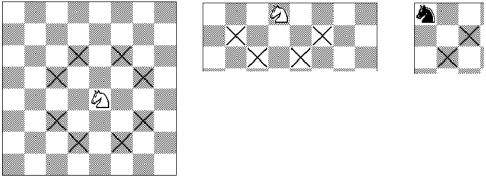
- Assume MAX is white
- Assume each piece has the following material value:
  - pawn $= 1$;
  - knight $= 3$;
  - bishop $= 3$;
  - rook $= 5$;
  - queen $= 9$;
- let w = sum of the value of white pieces
- let b = sum of the value of black pieces

$$\mathrm{Evaluation}(s) = \frac{w - b}{w + b}$$

Example: Chess (II)

- The previous evaluation function naively gave the same weight to a piece regardless of its position on the board...
- Let X_i be the number of squares the ith piece attacks

$$\mathrm{Evaluation}(s) = piece_1 value * X_1 + piece_2 value * X_2 + \ldots$$



(a) (b) (c)

Game playing: summary

- Minimax algorithm (with α-β pruning) fundamental for game playing.

Game playing: summary

- Minimax algorithm (with α-β pruning) fundamental for game playing.
- Not efficient enough for games such as chess, go, etc.

Game playing: summary

- Minimax algorithm (with α-β pruning) fundamental for game playing.
- Not efficient enough for games such as chess, go, etc.
- Evaluation functions are needed to replace terminal states by cutoff states.

Game playing: summary

- Minimax algorithm (with α-β pruning) fundamental for game playing.
- Not efficient enough for games such as chess, go, etc.
- Evaluation functions are needed to replace terminal states by cutoff states.
- Various approaches to define evaluation function.

Game playing: summary

- Minimax algorithm (with α-β pruning) fundamental for game playing.
- Not efficient enough for games such as chess, go, etc.
- Evaluation functions are needed to replace terminal states by cutoff states.
- Various approaches to define evaluation function.
- Most successful approach: machine learning. Evaluate positions using experience from previous games.