Lecture 7: Two-Player Games

Viliam Lisý & **Branislav Bošanský**

Artificial Intelligence Center
Department of Computer Science, Faculty of Electrical Eng.
Czech Technical University in Prague

bosansky@fel.cvut.cz

April, 2024

Moving to Two-Player Setting

Up to this point → finding optimal plan / best actions to be played in an environment

The agent was the only one changing the environment (**deterministic environment**) or there were **stochastic events**.

The stochastic events happen according to a known probability (probability of a box slipping out of the crane, etc.)

What if the environment (or another agent) is deliberately choosing the actions? → What is the "optimal plan" and how do we find it?

Games and Game Theory

Game Theory

An agent explicitly reasons about the possible actions of the other agents, their goals, and seeks own actions to be played w.r.t. to what other agents are going to play → such optimal behavior is defined by **game theory**.

Game theory is a broad scientific field covering parts of computer science, mathematics, economy.

We will only scratch the surface (see B4M36MAS Computational Game Theory or XEP36AGT for more in-depth topics regarding game theory).

Games, Game Theory, and Artificial Intelligence

What does this have to do with AI?

Playing games well has been a challenge from the beginning of AI and computer science.

John von Neumann, one of the founders of computer science, also established game theory (von Neumann minmax theorem) and was interested in poker and bluffing.

Games are very good benchmarks for algorithms (popular, known, well-defined rules), they are challenging (the state space is huge).

What are we going to cover?

We need to restrict to one of the most simple class of games:

- two players
- strictly competitive (or *zero-sum*) – win of one player is the loss of the opponent
- perfect information (chess, go, tic-tac-toe, ...)

What are we going to learn (this week)?

- how to find the optimal solution (optimal strategy)
- classical algorithms (pre 2006)
    - variants of branch-and-bound algorithm
    - useful heuristics

Next week → scaling-up and Monte Carlo sampling.

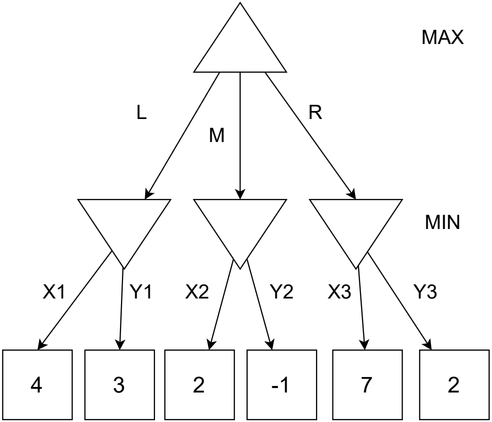
Extensive-Form Games with Perfect Information

Sequential games with finite horizon use **extensive-form representation** that generalize search trees:

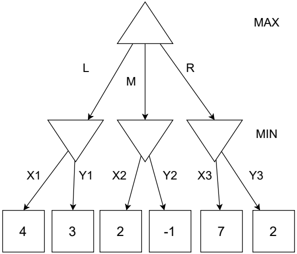
- P is a set of players $P = \{1, 2\}$ (or $\textsc{Max}$ and $\textsc{Min}$)
- H is a finite set of histories of actions from the initial positions where some player makes a decision. H_i are decision points of player $i \in P$.
- A is a finite set of actions, $A(h)$ denotes a set of actions applicable in a decision node $h \in H$
- $Z \subseteq H$ is a set of terminal histories where the game ends
- u is the utility function that assigns an outcome of the game to each terminal history, $u : Z \to \mathbb{R}$

Technically, in game theory, every player maximizes their own utility function. In zero-sum games, utility of player 1 equals negative utility of player 2. Hence minimizing the utility of player 1 is the same as maximizing its negative value.

Solving the Two-Player Games



Solving the Two-Player Games

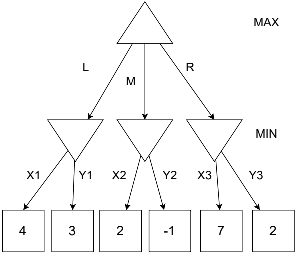


How to solve two-player games?

In MDPs, we have sought for an optimal plan or the best action for each state.

In games, we can also select the best action to be played in each decision point. Best action can either maximize (player 1) or minimize (player 2) the utility.

Solving the Two-Player Games



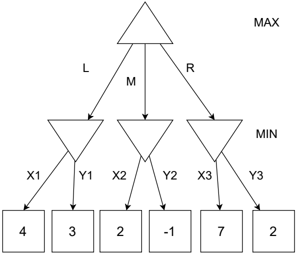
Similarly to deterministic uninformed search, we can use a depth-first search algorithm. For a history h:

1. if h is a terminal history ($h \in Z$), then return $u(z)$,
2. if h is a decision node, evaluate all children $v_a = \text{search}(ha),\ a \in A(h)$ and
   1. if $h \in H_1$, return $\max_{a \in A(h)} v_a$
   2. if $h \in H_2$, return $\min_{a \in A(h)} v_a$

This baseline algorithm is known as **minimax** algorithm or simply a **backward induction** in two-player perfect information games.

The utility of player 1 when both players play optimally is called **the value of the game**.

We Do Not Need To Consider Everything



Search through the complete game tree can be impractical and unnecessary. Consider how the algorithm advances through the search tree:

- after fully evaluating action L in the root node, $v_L = 3$,
- when evaluating M, the algorithm first visits $X2$ and determines that $v_{X2} = 2$

Obsevation

Regardless of the utility action after playing $Y2$, action M is never going to be selected in the root node as the best action!

Alpha-Beta Pruning

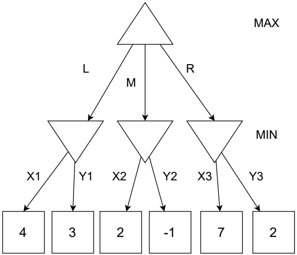
Minimax search with **alpha-beta pruning**:

- variant of a branch-and-bound algorithm
- extend the search with lower (α) and upper-bound (β) estimates on the value of the game.

For a decision point h (initial values for bounds α_h and β_h are passed as parameters):

```
1 if $h$ is a leaf ($h \in Z$), then return $u(z)$,
2 if $h$ is a decision node and $h \in H_1$ then for each $a_h \in A(h)$:
    1 $v_h = \max(v_h, \text{search}(a_h, \alpha_h, \beta_h))$    $\alpha_h = \max(\alpha_h, v_h)$
    2 if $\beta_h \leq \alpha_h$ then break
3 if $h$ is a decision node and $h \in H_2$ then for each $a_h \in A(h)$:
    1 $v_h = \min(v_h, \text{search}(a_h, \alpha_h, \beta_h))$    $\beta_h = \min(\beta_h, v_h)$
    2 if $\beta_h \leq \alpha_h$ then break
4 return $v_h$
```

Example with Alpha-Beta Pruning



Alpha-beta pruning starts with $\alpha_\emptyset = -\infty$ and $\beta_\emptyset = \infty$. After evaluating $X1$, β_L is set to 4. The value is updated to 3 after evaluating $Y1$.

The solution of v_L, value 3, is then propagated to $\alpha_\emptyset$ and thus the next recursive call uses updated lower bound (hence, α_M is initialized to 3).

After evaluating $X2$, $\beta_M = 2$. Therefore, we have $\beta_M < \alpha_M$ and we can stop exploring node M.

Negamax

In many standard board games, players are alternating (Max player moves, then Min, etc.).

We can simplify the pseudocode of the algorithm by reverting the rewards and bounds:

```
1. if $h$ is a leaf ($h \in Z$), then return $u(z)$,
2. for each $a_h \in A(h)$:
    1. $v_h = \max(v_h, -\text{search}(a_h, -\beta_h, -\alpha_h))$    $\alpha_h = \max(\alpha_h, v_h)$
    2. if $\beta_h \leq \alpha_h$ then break
3. return $v_h$
```

This algorithm is known as **Negamax**. It is just a more compact way of describing the same behavior.

For educational purposes (learning and understanding the algorithm), I recommend using the full version (2 players, Max, Min; this holds also for follow-up algorithms, such as Negascout).

Changing the Bounds

Recall, what we stated before: Alpha-beta pruning starts with $\alpha_\emptyset = -\infty$ and $\beta_\emptyset = \infty$.

Setting the bounds this way is definitely correct. But what happens if we run Alpha-Beta pruning with some other initial values?

Imagine that we have a position in a game and we estimate (using heuristic evaluation) that the value of the game should be around 10.

What if we run Alpha-Beta pruning with interval $[\alpha_\emptyset, \beta_\emptyset] = [9, 11]$?

Positive Impact:
We can evaluate significantly fewer nodes!

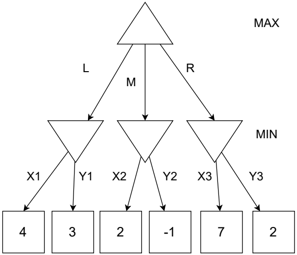
Negative Impact:
We can miss the correct solution!

Changing the Bounds

What if we run Alpha-Beta pruning with interval $[\alpha_\emptyset, \beta_\emptyset] = [9, 11]$?

Positive Impact:
We can evaluate significantly fewer nodes!

Negative Impact:
We can miss the correct solution!



Consider our example: After evaluating $X1$, β_L is set to 4 and if $\alpha_\emptyset$ (and thus initial value of α_L) is set to 9, then $\beta_L < \alpha_L$. Therefore, $Y1$ is never evaluated and the correct solution for this subtree (and also the whole game) is not found.

Question

What is the value returned if we continue with the algorithm?

Changing the Bounds

Question

What is the value returned if we continue with the algorithm?

The algorithm returns value 7. This is not the value of the game (we know it is 3).

What can we conclude from return value 7?

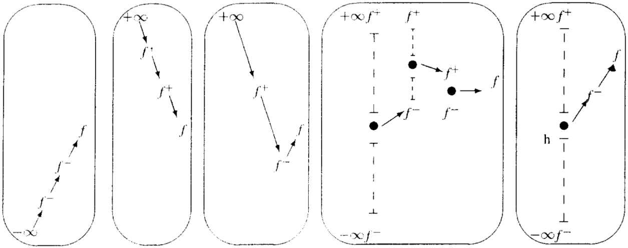
- the value of the game is not in the interval $[9, 11]$ → our initial estimation was incorrect
- since the return value is **lower** than the estimated lower bound, we know that some actions of $\textsc{Min}$ player were not properly explored
- hence, the actual value of the game can only be lower → we can rerun the search with interval $[-\infty, 7]$

If the return would be higher than the estimated upper bound, we know that the true value is higher.

Changing the Bounds

We can systematically search for a correct value of the game (thus the optimal strategy) by:

- (e.g., binary) search over the interval of values
- repeated calls to the alpha-beta algorithm with modified bound interval (called **window**)



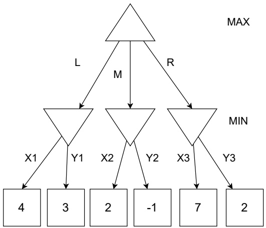
Dual* SSS* MTD(step) MTD(bi) MTD(f) [1]

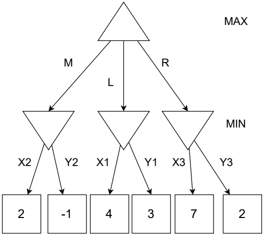
[1] Best-first fixed-depth minimax algorithms. Plaat et. al. , In Artificial Intelligence, Volume 87, Issues 1-2, November 1996, Pages 255-293

Action Ordering

Can we use the idea of quick check **during** the search?

How does the ordering in which the actions are evaluated affect the number of searched nodes?





Assuming the actions are evaluated from left to right, Alpha-Beta pruning will not prune out anything in the second game.

The estimated searched space of Alpha-Beta pruning is $O(b^{d/2})$ in case actions are evaluated in the optimal order (as opposed to $O(b^d)$ for minimax).

NegaScout (or Principal variation search)

What if we assume have the optimal ordering of actions?

We can, for example, have a good heuristic that sorts the actions prior to an evaluation in each node.

Idea:

- fully evaluate the first action (i.e., with the full-sized interval),
- evaluate subsequent actions with a minimal-sized interval (**null window**),
- based on the null-window evaluation:
  - if the returned value indicates that the value of the subtree is the same or worse, the algorithm proceeds with next action
  - if the returned value indicates that the value of the subtree can be better, the algorithm must evaluate this action again properly

NegaScout (or Principal variation search)

For a decision point h (initial values for bounds α_h and β_h are passed as parameters):

```
1 if $h$ is a leaf ($h \in Z$), then return $u(z)$,
2 $\beta'_h = \beta_h$, $\alpha'_h = \alpha_h$
3 if $h$ is a decision node and $h \in H_1$ then for each $a_h \in A(h)$:
    1 $v_h = \text{search}(a_h, \alpha_h, \beta'_h)$
    2 if (($\alpha_h < v_h < \beta_h$) and ($h$ is not the first child))
        • $v_h = \text{search}(a_h, v_h, \beta_h)$
    3 $\alpha_h = \max(\alpha_h, v_h)$
    4 if $\beta_h \leq \alpha_h$ then break
    5 $\beta'_h = \alpha_h + 1$
    6 return $\alpha_h$
4 ... (similarly for player 2) ...
```

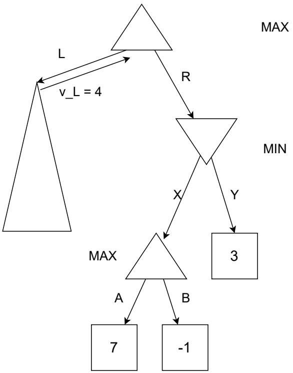
NegaScout (or Principal variation search) – Player 2

For a decision point h (initial values for bounds α_h and β_h are passed as parameters):

```
1 ...
2 if $h$ is a decision node and $h \in H_2$ then for each $a_h \in A(h)$:
    1 $v_h = \text{search}(a_h, \alpha'_h, \beta_h)$
    2 if $((\alpha_h < v_h < \beta_h)$ and ($h$ is not the first child))
        • $v_h = \text{search}(a_h, \alpha_h, v_h)$
    3 $\beta_h = \min(\beta_h, v_h)$
    4 if $\beta_h \leq \alpha_h$ then break
    5 $\alpha'_h = \beta_h - 1$
    6 return $\beta_h$
```

NegaScout vs. Alpha-Beta

NegaScout can evaluate some nodes multiple times. At the same time, it can never evaluate more different histories and can be $10 - 20\%$ faster than Alpha-Beta pruning.



Assume that after evaluating subtree L, the value v_L equals to 4. The evaluation of R starts with $[\alpha_R, \beta_R] = [4, 5] = [\alpha_{RX}, \beta_{RX}]$. When the algorithm evaluates A, it updates α_{RX} to 7.Since β_{RX} has been manually set to 5, it holds that $\beta_{RX} < \alpha_{RX}$ and the algorithm prunes out evaluation of action B.

Note that this would not happen in Alpha-Beta pruning.

Game Solving vs. Game Playing

The algorithms that we have described so far are **offline (equilibrium computation) algorithms**.

Given a game, they compute an optimal strategy (for both players) and value of the game.

However, this is not tractable for most practical games due to the size of the game tree:

- chess has branching factor (number of applicable actions) $\approx$ 35, Go up to 360, etc.; games can take tens (hundreds) of moves of both players to terminate.

Game-playing algorithms are searching only to a limited depth. Instead of a utility function applied on terminal states, they return a value of a **heuristic evaluation function**.

Game Solving vs. Game Playing

Unfortunately, designing a well-informed heuristic evaluation function is much more challenging than in the single-player case.

Optimizing a strategy in a depth-limited game does not have to correspond to truly optimal strategy:

- consider chess and assume the algorithm searches to depth d actions for both players
- player 1 has to sacrifice a queen to avoid getting checkmate and losing
- however, the checkmate is beyond the horizon of d actions
- hence, from the perspective of evaluation function, it can be better to sacrifice a pawn or a knight instead of the queen – the algorithm does not know that sacrificing a different piece does not prevent the problem

There are heuristics tackling these issues (e.g., at certain dynamic positions, the search is not strictly terminated at the horizon).

Additional Heuristics

There are many useful heuristics developed for games, that can be useful in other problems.

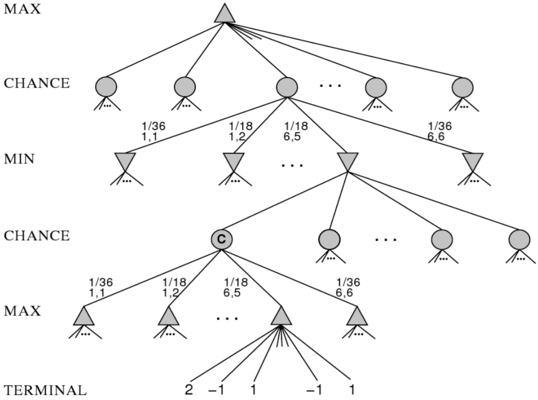
One of the best known are **transposition tables**:

- a cache table with previously evaluated positions,
- minimize negative effects of game tree (if the same position is reached with a different sequence of actions),
- for a depth-limited alpha-beta (negascout) algorithm, the depth limit and bounds can affect the computed (and stored) value

Games with Chance

Two-player games can easily be extended with stochastic events.

We can introduce another (Nature or chance) player that chooses actions according to a known distribution.

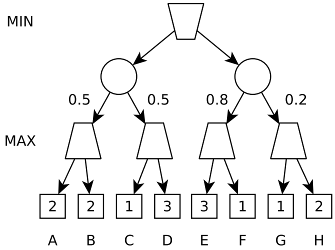


Games with Chance

Question

How does the Alpha-Beta pruning translates to a game with chance nodes?

Example:



Can Alpha-Beta pruning algorithm prune-out anything? Assume that the utility values can be $\in \mathbb{R}$.

What happens if the possible utility values are from interval $[1, \infty]$?

You should be able to figure this out! A similar task **can be** in the exam.

What is it all good for?

The first algorithm that beat a chess professional, Deep Blue, was built on these algorithms (link to a paper).

> *Deep Blue relies on many of the ideas developed in earlier chess programs, including quiescence search, iterative deepening, transposition tables (all described in [24]), and NegaScout [23].*

Besides that, Deep Blue was heavily using parallel search and special hardware "chess chips".

What is it all good for?

The techniques are applicable for other problems / search-based algorithms:

- use of bounds and pruning out not-perspective branches
- problems with the horizon (if a monotonic heuristic is not possible)
- use of cached values