Faculty of Mathematics and Physics
Charles University
10.12.2024

NAIL134 AI for Games

Modeling RTS Combat Scenarios

Jakub Gemrot (Tomáš Dvořák)

Modelling RTS combat
States, units and moves

Problem

How to create combat AI for real-time strategy game?

- Whom to target
- where to move
- in order to minimize own losses
- while maximizing damage to the opponent?

☝ *It can be surprisingly hard even if AI*

- *controls one unit*
- *and faces the opposition of N enemy units !*

Mathematical Games

Games

Mathematically speaking

Informal definition

Game is any set of circumstances that has a result dependent on the actions of two or more decision-makers (players).

Games

Mathematically speaking

Concepts of theoretical games

Players	a finite set of players
Actions	(Moves) available to players at certain moments of the game
Nodes	(States) moments of the game where players can perform actions == make moves
Game Tree	rooted tree, root = initial state, edges = moves
Payoffs	utilities (gains / losses) of players at the game tree leaves
Information set	"fog of war", a set of nodes, which are indistinguishable for a given player, i.e., a given player lacks information to recognize what concrete game node they are in right now
Nature	(Environment / Chance) a player used to model randomness

Extensive form games

Extensive form games

Describing the (Tic-tac-toe) game in its entirety

Tic-tac-toe game tree

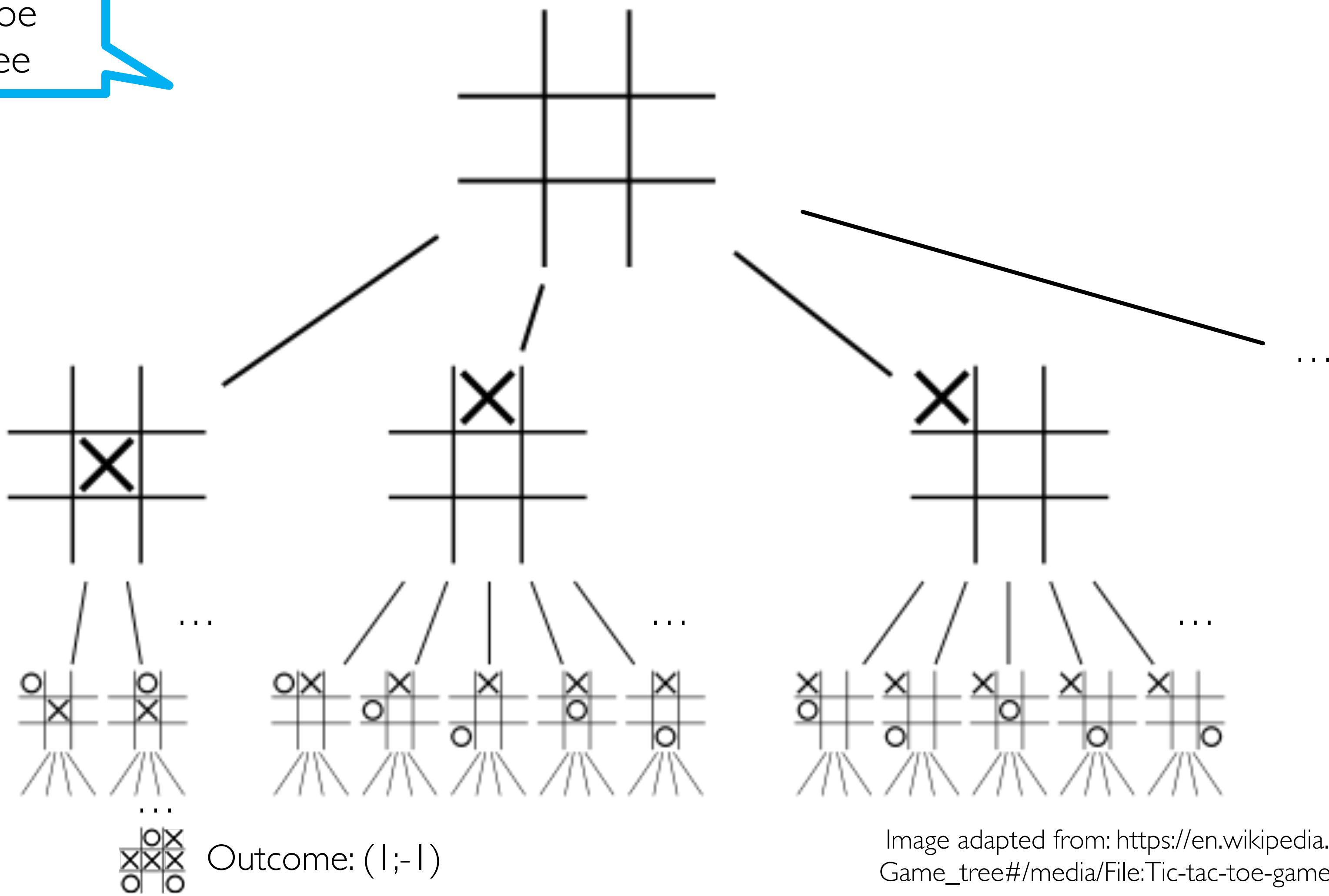
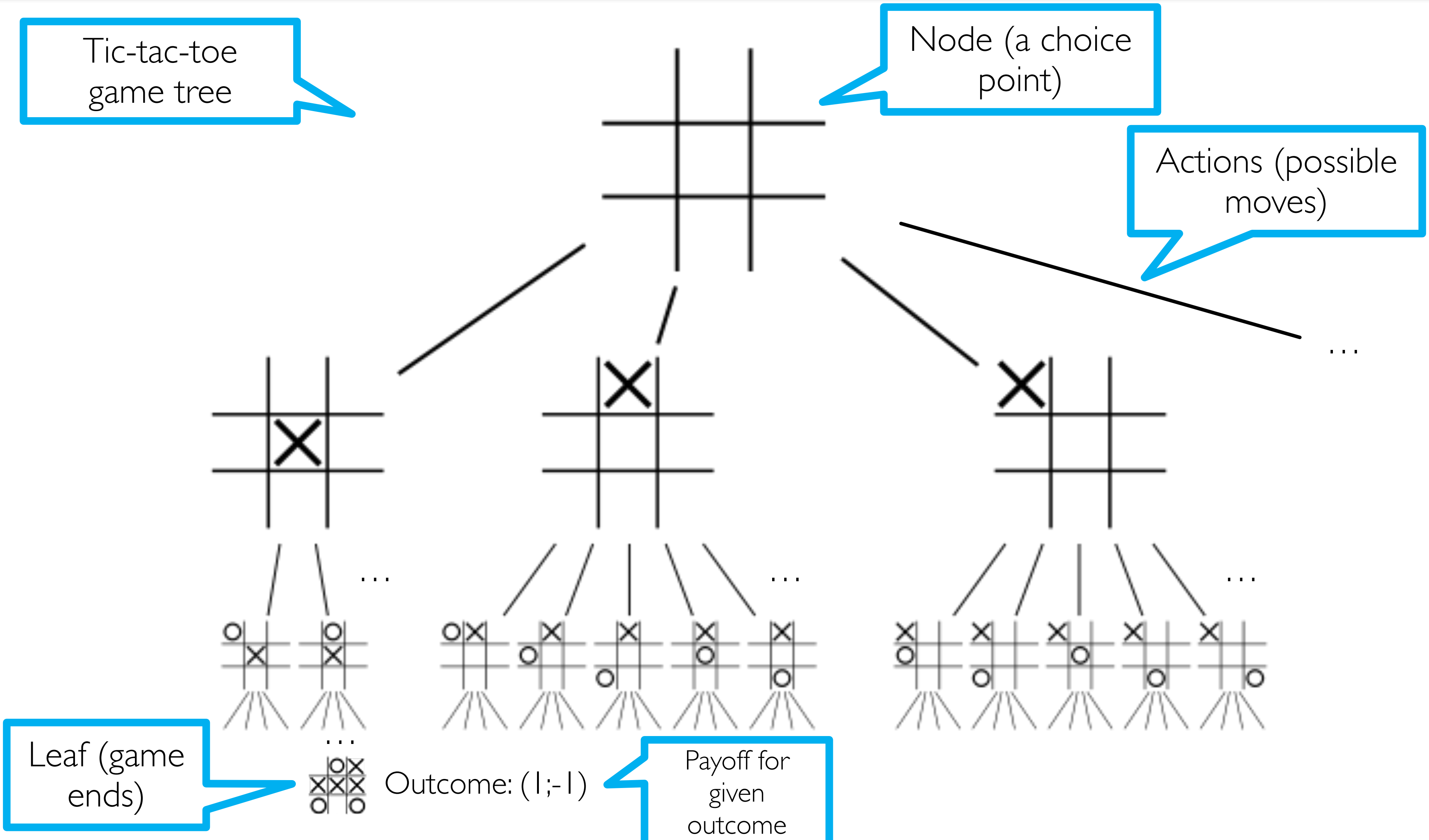


Image adapted from: https://en.wikipedia.org/wiki/Game_tree#/media/File:Tic-tac-toe-game-tree.svg

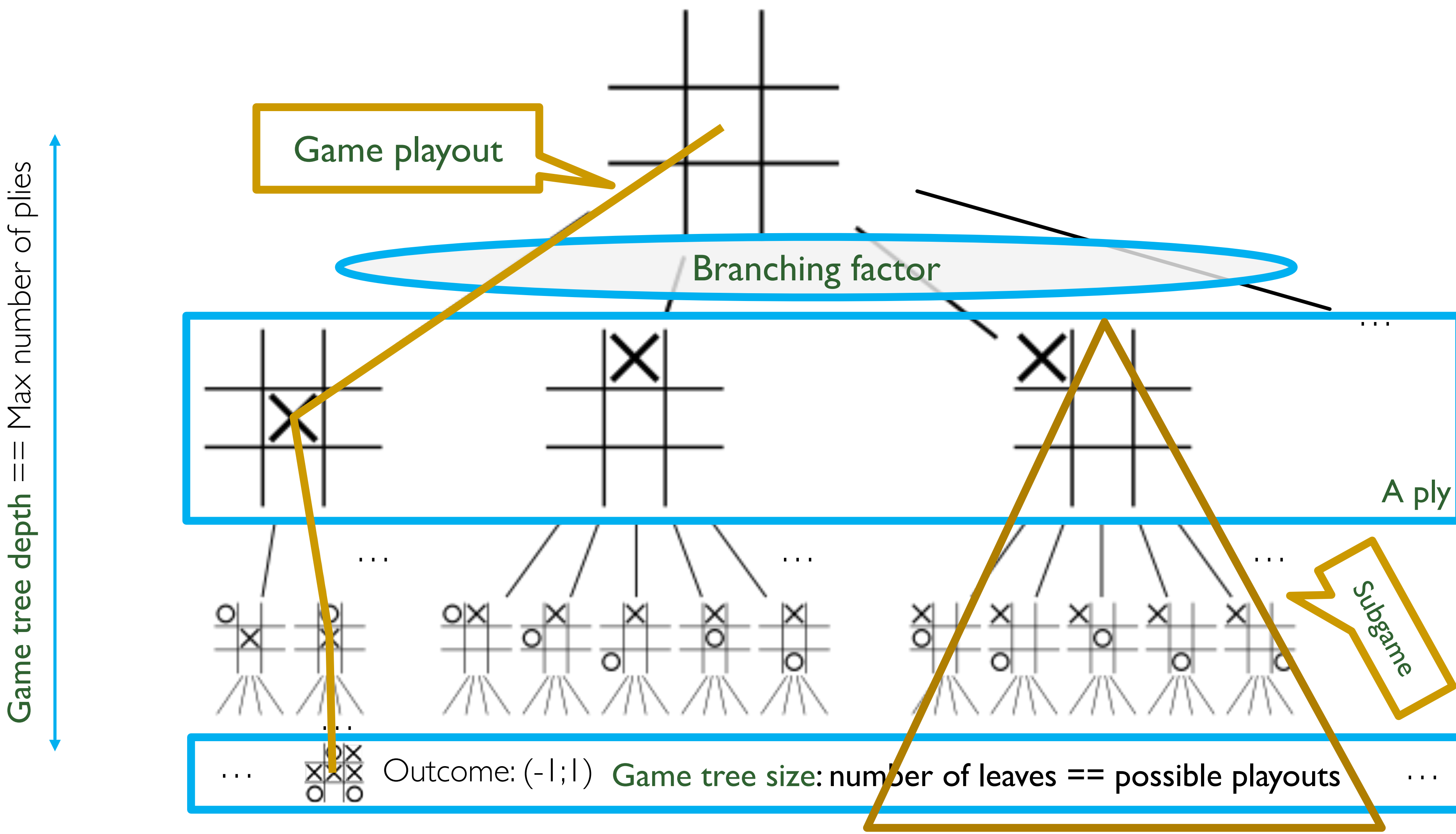
Extensive form games

Describing the (Tic-tac-toe) game in its entirety



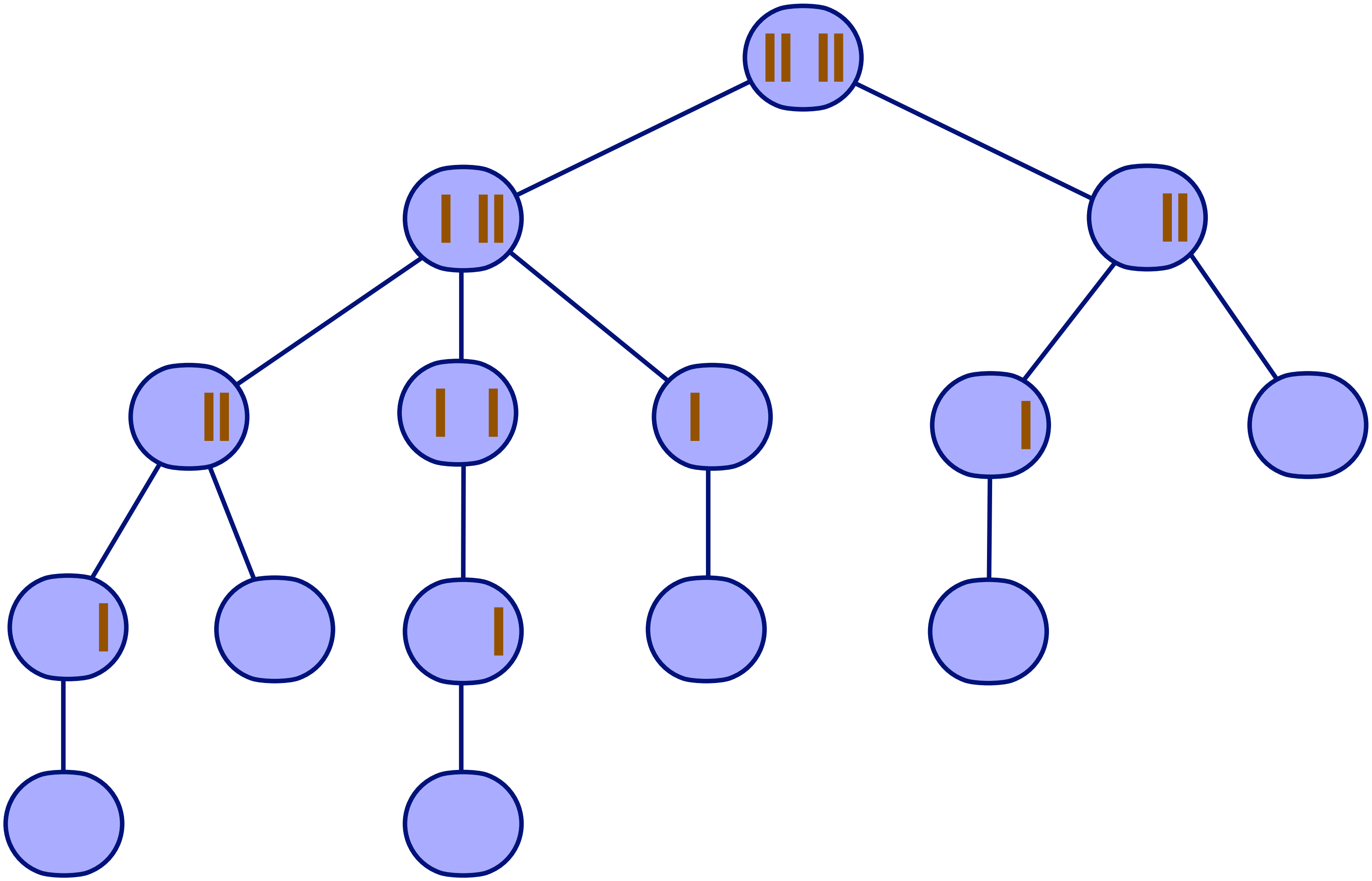
Extensive form games

Describing the (Tic-tac-toe) game in its entirety



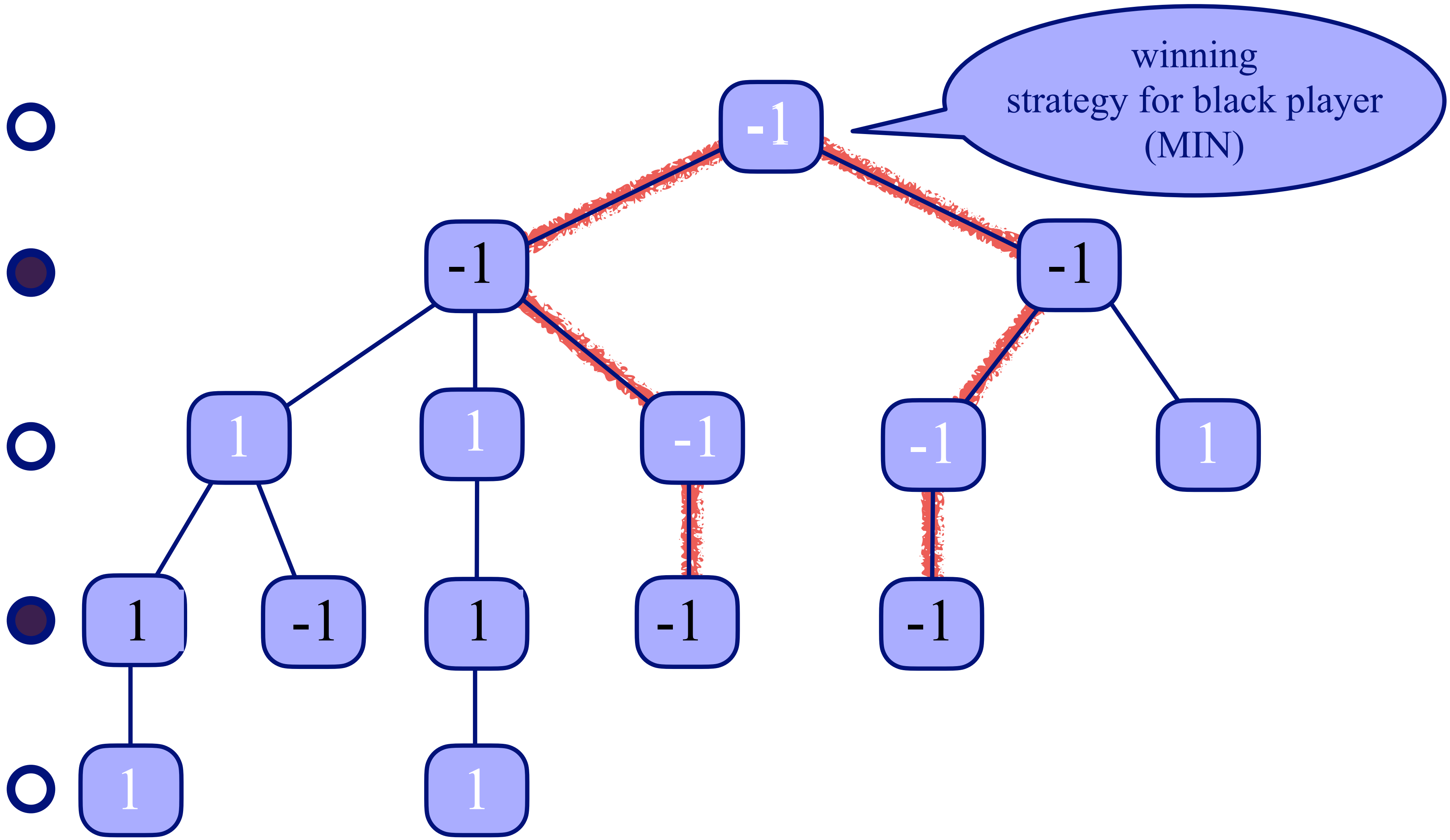
Quick recap

NIM game tree



Quick recap

Minimax algorithm



Quick recap

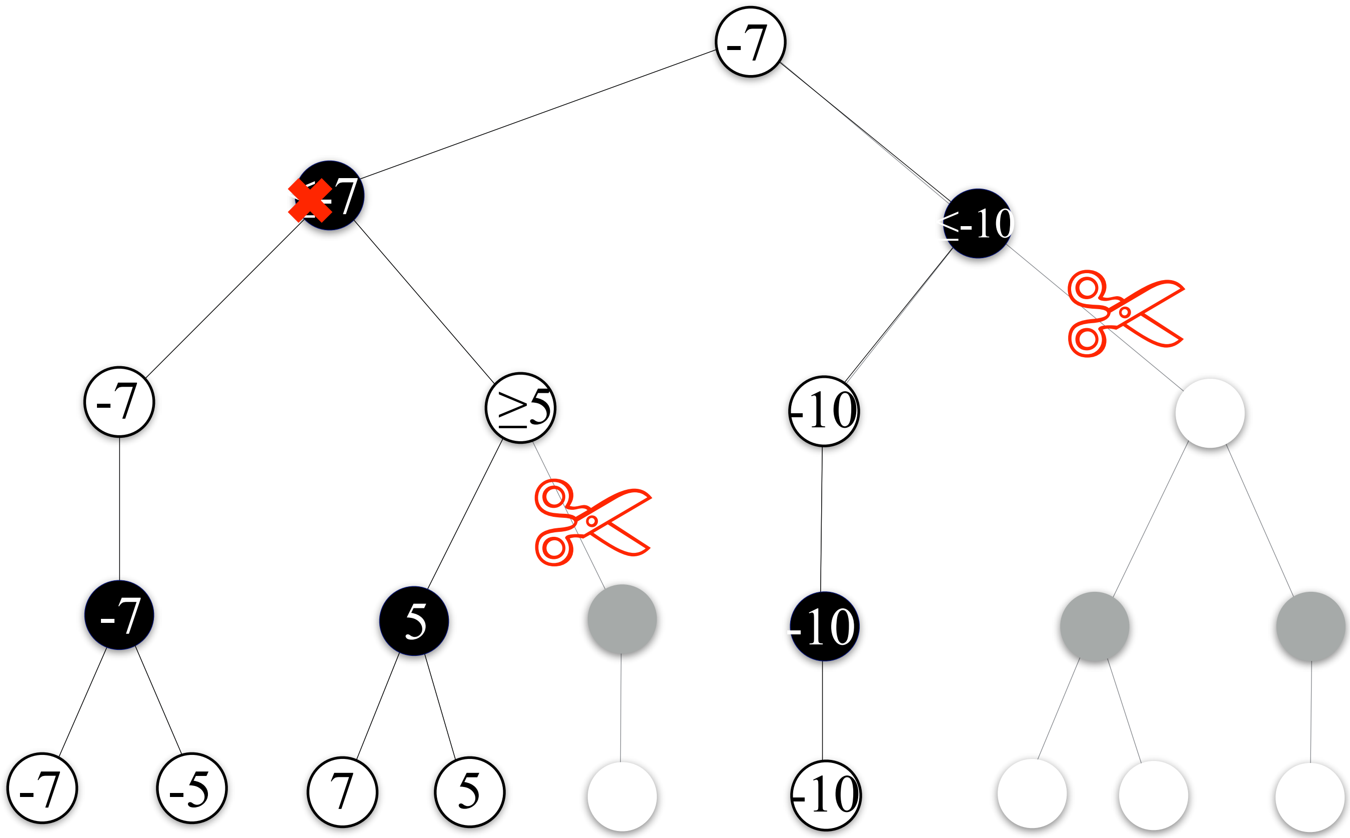
Zermelo's theorem

☝ **Zermelo's theorem** (1913 [pdf]). For every finite game of two players with perfect information and zero sum, one of the players has

- a winning strategy (draw not allowed)
- or non-losing strategy (draw allowed).

Jonathan Schaeffer, Neil Burch, Yngvi Bjornsson, Akihiro Kishimoto, Martin Muller, Rob Lake, Paul Lu, Steve Sutphen, *Checkers is Solved*, Science 317 (2007), 1518-1522 [doi]

Quick recap
Alpha-beta pruning



Games

Mathematically speaking

Main features of games important to us

Sequential / Simultaneous
Perfect / Imperfect information

Additional types of games

Zero-sum / Non-zero sum
Cooperative / Non-cooperative
Symmetric / Asymmetric

Normal form games

Normal form games
Theoretical background

Also called strategic form games

Concepts of theoretical games

Strategy	a complete „algorithm“ for a player how to play the game
Strategy profile	an assignment of strategies to each player
Pure strategy	a strategy, where the player has exactly one action planned for every node where the player can make a move
Strategy set	all pure strategies available to a player to choose from
Mixed strategy	a probability distribution over a strategy set, associated with expected payoff

Normal form games
Theoretical background

Normal form game definition

Players indexed 1.. *n*

S_i strategy set for a player *i*

u payoff function: $S_1 \times \ldots \times S_n \rightarrow R^n$
i.e., for strategies $s_i \in S_i$ chosen by players (strategy profile)
the function returns players' payoffs $r_i \in R$
(+ … gains, - … losses)

Many games and even social situations falls under the notion of normal form games!

Two player games : payoff function → matrix

Normal form games
Theoretical background

Normal form game matrix representation

Prisoners' dilemma

		Player 2: confess	Player 2: don't confess
Player 1	confess	(-6, -6)	(0, -10)
	don't confess	(-10, 0)	(-1, -1)

Image source: https://www.thoughtco.com/the-prisoners-dilemma-definition-1147466

Nash equilibrium

Nash equilibrium

Theoretical background

Nash equilibrium is a strategy profile such that no player can increase his own payoff by changing his strategy while the other players keep theirs unchanged.

It represents a stable point in a game: stable in the sense that there is no rational incentive for any player to deviate.

John Forbes Nash (1925-2015)

Nash equilibrium
Theoretical background

Nash's Theorem

Every finite, non-cooperative game of two or more players has a Nash equilibrium either in pure or mixed strategies.

In other words…

If the proceeding of the game is in Nash equilibrium, then even if I know, what strategies others are playing, I cannot do better then to stick with strategy I'm currently playing.

Normal form games
Theoretical background

Pure strategy Nash equilibrium example

Prisoners' dilemma

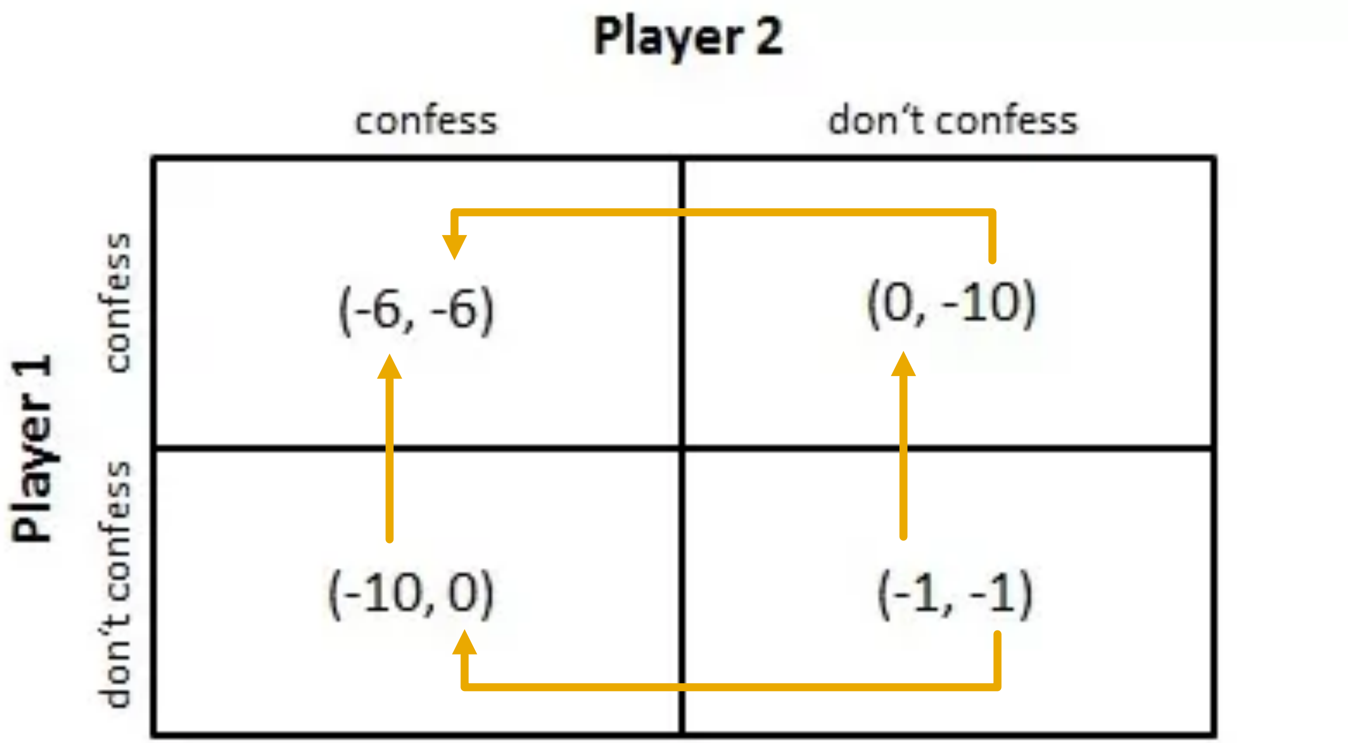


Image source: https://www.thoughtco.com/the-prisoners-dilemma-definition-1147466

Mixed strategy Nash equilibrium example

Rock paper scissors

		Player 2		
		Rock	Paper	Scissors
Player 1	Rock	0, 0	-1, 1	1, -1
	Paper	1, -1	0, 0	-1, 1
	Scissors	-1, 1	1, -1	0, 0

Normal form games
Theoretical background

Mixed strategy Nash equilibrium

Rock paper scissors – Linear programming

Player 1 \ Player 2	Rock	Paper	Scissors
Rock	0, 0	-1, 1	1, -1
Paper	1, -1	0, 0	-1, 1
Scissors	-1, 1	1, -1	0, 0

Consider a probability distribution: $r + p + s = 1$

Expected utilities of P2 for strategies of P1:

P1 plays rock: $u = r \cdot 0 + p \cdot 1 + s \cdot (-1) = p - s$

P1 plays paper: $u = r \cdot (-1) + p \cdot 0 + s \cdot 1 = s - r$

P1 plays scissor: $u = r \cdot 1 + p \cdot (-1) + s \cdot 0 = r - p$

Now P1 is trying to minimize P2 utilities, so we have

$$u \leq p - s \; ; \; u \leq s - r \; ; \; u \leq r - p$$

P2 is trying to maximize the utility, which yields the result

$$r = p = s = \frac{1}{3}$$

Battle of Sexes

Backward induction

Battle of Sexes

The Game

Normal form game

A boy and a girl are spending evening together, though they forgot, where they agreed to meet. The boy would like to go and see a football match, whereas the girl would like to go to the opera.

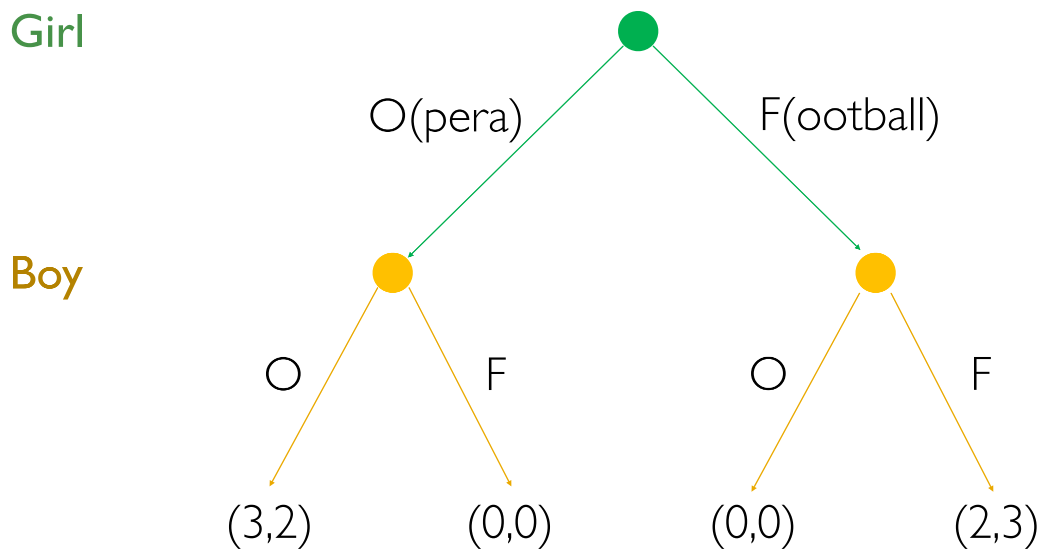
	Opera	Football
Opera	3,2	0,0
Football	0,0	2,3

Battle of Sexes

The Game

Altered game

Suppose the girl has a chance to text the boy where is she's going.
The simultaneous game turns into sequential.

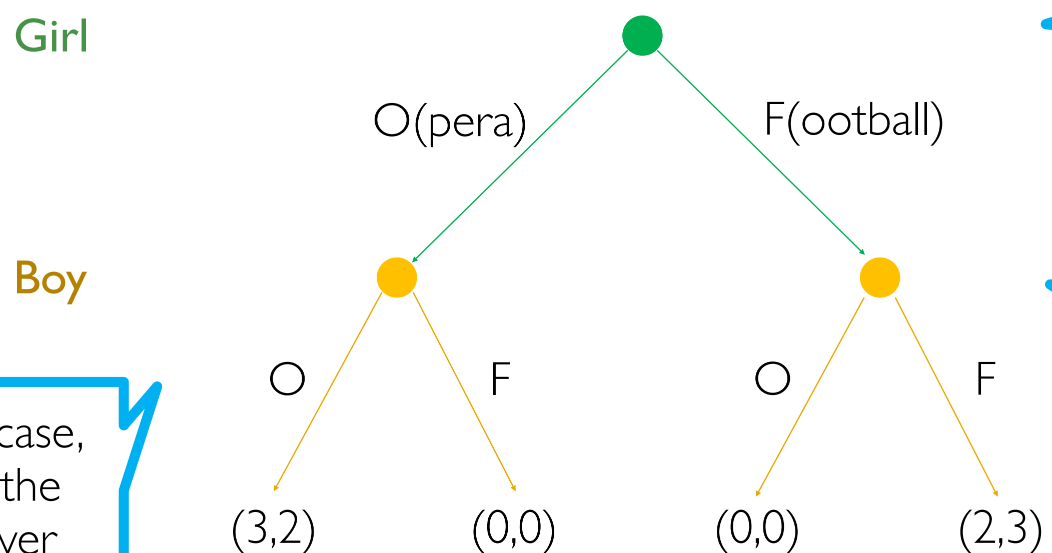


Battle of Sexes

The Game

Altered game

Suppose the girl has a chance to text the boy where is she going. The simultaneous game turns into sequential.



1. In each subgame, the boy (if rational) has to adopt "copy" strategy.

2. Knowing that and applying backward induction, the girl knows what to do

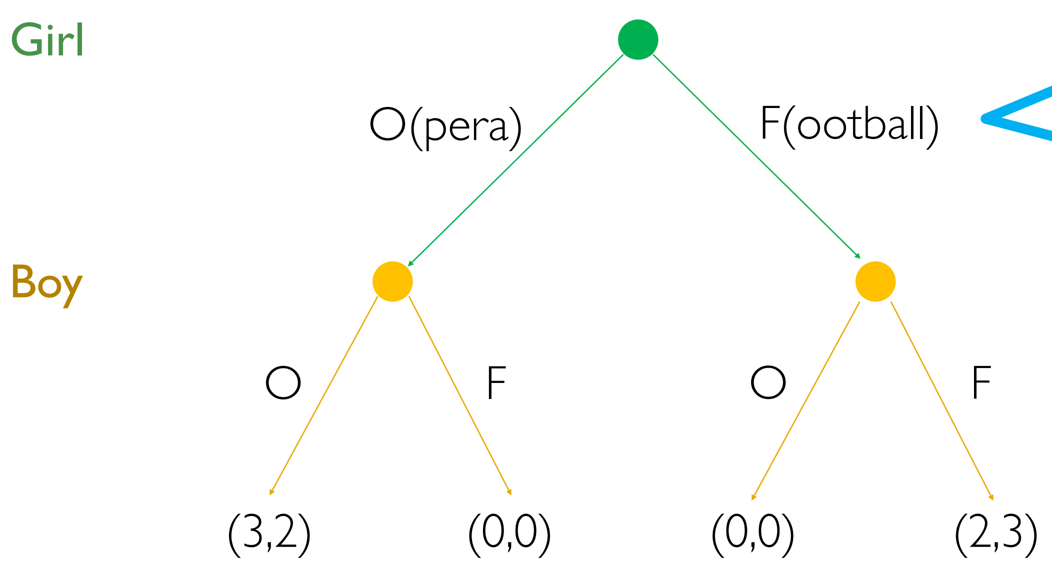
3. In this case, there is the first mover advantage.

Battle of Sexes

The Game

Altered game

Suppose the girl has a chance to text the boy where is she going. The simultaneous game turns into sequential.



4. Finally, perfect information simultaneous games are (sometimes) referred to as **stacked matrix games**.

5. That's because each subgame can be described using a normal form, i.e., the payoff matrix.

Nash equilibrium
for extensive form games

Sequential game with perfect information

How to compute Nash equilibrium?

☝ Generalize minimax search ☞ backward induction:

- for each nonterminal node
- if all the children have been labeled with a payoff profile
- then label parent with a payoff profile from the child node that
- maximizes the payoff of the player making the decision at parent
- if there is a tie, then choose arbitrarily
- if we have chance nodes, then compute expected utility

☝ Payoff profile labeling the initial state = payoff profile
that would be obtained by playing Nash equilibrium strategies

Nash equilibrium

for extensive form games

Sequential game with perfect information

☞ Nash equilibrium strategies for extensive-form games can be computed in polynomial time using backward induction

☞ Every extensive-form game has at least one Nash equilibrium in pure strategies

A profile of strategies forms a subgame perfect Nash equilibrium in a game G if it is a Nash equilibrium in every subgame of G.

☞ Backward induction computes subgame perfect Nash equilibrium

Back to RTS combat! … Sort of.

Before taking on RTS, we're going to study turn-based combat

Furtak, T., & Buro, M. On the complexity of two-player attrition games played on graphs [doi]

Attrition games on graphs
And its intricacies

Attrition Game on Graph (AGG)

Two-player perfect information simultaneous game on a graph :

- every node belongs to a single player (either white and black here)
- each node described by $\langle h; a \rangle$

h	…	health
a	…	attack value

- graph is directed, an $x \to y$ edge means x can attack y

Discrete version: series of rounds, all nodes choose their target, then simultaneously attack, all nodes with zero or lower health are removed. Players may have various objectives (e.g. minimize damage taken).

Continuous version: units attack constantly and are immediately removed when their health reaches 0.

Attrition games on graphs

And its intricacies

1 vs N units; minimize dmg taken

White: 1 unit $\langle h_0; a_0 \rangle$; Black: N units $\langle h_1; a_1 \rangle, \ \ldots, \langle h_n; a_n \rangle$

White objective: minimize damage taken by its unit

Theorem

In discrete version, to minimize white's total sustained damage it is sufficient to order its targets by nonincreasing value of $a_i / \lceil h_i / a_0 \rceil$ and never change targets until they have been destroyed.

Intuitively, white wants to neutralize a threat with high attack value and low health while not over overkilling it.

Proof by contradiction, order units, try to swap two black units j, j+1

Attrition games on graphs

And its intricacies

1 vs N units; white is maximizing reward (AGG-1:N-Rew)

White: 1 unit $\langle h_0; a_0 \rangle$; Black: N units $\langle h_1; a_1 \rangle,\ \ldots, \langle h_n; a_n \rangle$

White objective: maximize reward from neutralizing black units; each black unit associated with reward $r_i \geq 0$.

Theorem

Given a discrete AGG scenario with n black units with health h_i, attack a_i, and kill reward $r_i \geq 0$ for white, and a single white unit with health h_o and attack a_o, it is NP-hard for white to decide what the reward-maximal target ordering is, in case white does not survive.

Attrition games on grap

And its intricacies

1. In SC or AoE, consider facing multiple enemies of different kinds including special power units.

1 vs N units; white is maximizing reward (AGG-1:N-Rew)

White: 1 unit $\langle h_0; a_0 \rangle$; Black: N units $\langle h_1; a_1 \rangle, \ \ldots, \langle h_n; a_n \rangle$

White objective: maximize reward from neutralizing black units; each black unit associated with reward $r_i \geq 0$.

2. We will show how to encode the 0-1 knapsack problem as AG-1:N-Rew.

Theorem

Given a discrete AGG scenario with n black units with health h_i, attack a_i, and kill reward $r_i \geq 0$ for white, and a single white unit with health h_o and attack a_o, it is NP-hard for white to decide what the reward-maximal target ordering is, in case white does not survive.

Attrition games on graphs

And its intricacies

0-1 knapsack → AGG-1:N-Rew

For 0-1 knapsack problem instance of n items $\langle w_i; v_i \rangle$ and a bag capacity w_{max}, we define AGG-1:N-Rew input as follows:

White:
single unit $\langle w_{max}; 1 \rangle$

Black:
n units $\langle w_i; 0 \rangle$ and reward v_i
1 unit $\langle \infty; 1 \rangle$

Equivalence of instances:

- white unit is destroyed in w_{max} steps
- white may eliminate black units with total health $\leq w_{max}$
- reward for destroyed units = value of items put into the bag

Finally, the RTS combat!

Churchill, D., Saffidine, A., & Buro, M. Fast heuristic search for RTS game combat scenarios [doi]

Modelling RTS combat

RTS Combat Game - States, units and moves

Units

Unit $u = \langle p, \mathrm{hp}, \mathrm{hp}_{\max}, \mathrm{t}_a, \mathrm{t}_m, v, w \rangle$

- Position $p = (x, y)$ in $\mathbb{R}^2$
- Current hit points hp and maximum hit points $\mathrm{hp}_{\max}$
- Time step when unit can next attack t_a, or move t_m
- Maximum unit velocity v
- Weapon properties $w = \langle \mathrm{damage}, \mathrm{cooldown} \rangle$

We also track damage-per-frame:

- $u.dpf = \dfrac{u.w.damage}{u.w.cooldown}$

Modelling RTS combat

RTS Combat Game - States, units and moves

Moves

Move $m = \{a_0, \ldots, a_k\}$ which is a combination of unit actions $a_i = \langle u, \text{type}, \text{target}, t \rangle$, with

- Unit u to perform this action
- The type type of action to be performed:

  Attack unit target

  Move u to position target

  Wait until time t

States

State $s = \langle \text{t}, p, m, U_1, U_2 \rangle$

- Current game time t
- Player p who performed move m to generate s
- Sets of units U_i under control of player i

Modelling RTS combat

RTS Combat Game - States, units and moves

Legal moves for units

Given a state s and unit u, its legal actions are:

$u.t_a \leq s.t$ … u may attack anything in range

$u.t_m \leq s.t$ … u may move in any direction

$u.t_m \leq s.t \leq u.t_a$ … u may wait

$s.t < u.t_m,\ s.t < u.t_a$ … u has no legal actions

Modelling RTS combat

RTS Combat Game - States, units and moves

Game terminal condition

All units of a player reach zero HP

Further assumptions

Zero-sum game (i.e., no asymmetric rewards)

Limitations wrt real RTS games

- no special powers or spells
- no hit point or shield regeneration
- no travel time for projectiles
- no unit collisions
- no unit acceleration, deceleration or turning
- no fog of war

=> Yet the game is harder than AGG == at least NP-hard => backward induction is unfeasible => we need to resort to (heuristic) searches

Scripted Behaviors

The obvious heuristic

Scripted behaviors
Defining strategy through reactive behavior

Different types of scripts

Random	[RND]	Pick a legal move with uniform probability distribution
Attack-Closest	[AC]	1. Attack closest in range if able 2. If within range & reloading, wait 3. If not in range, move to range
Attack-Weakest	[AW]	Dtto AC, but: 1. attacks weakest in range
Kiting	[Kit]	Dtto AC, but: 2. move away from closest enemy
Attack-Value	[AV]	Dtto AC, but: 1. attacks u w/ highest $u.dpf / u.hp$
No-OverKill-Attack-Value	[NOK-AV]	Dtto AV, but: will no try to attack unit that has been assigned lethal damage, choose next priority target
Kiting-AV	[Kit-AV]	Dtto Kit, but: 1. attacks u w/ highest $u.dpf / u.hp$

Scripted strategy w/ script X: assign script X to all units a player controls

Search approximations

Adapting alpha-beta search to simultaneous games

Using searches

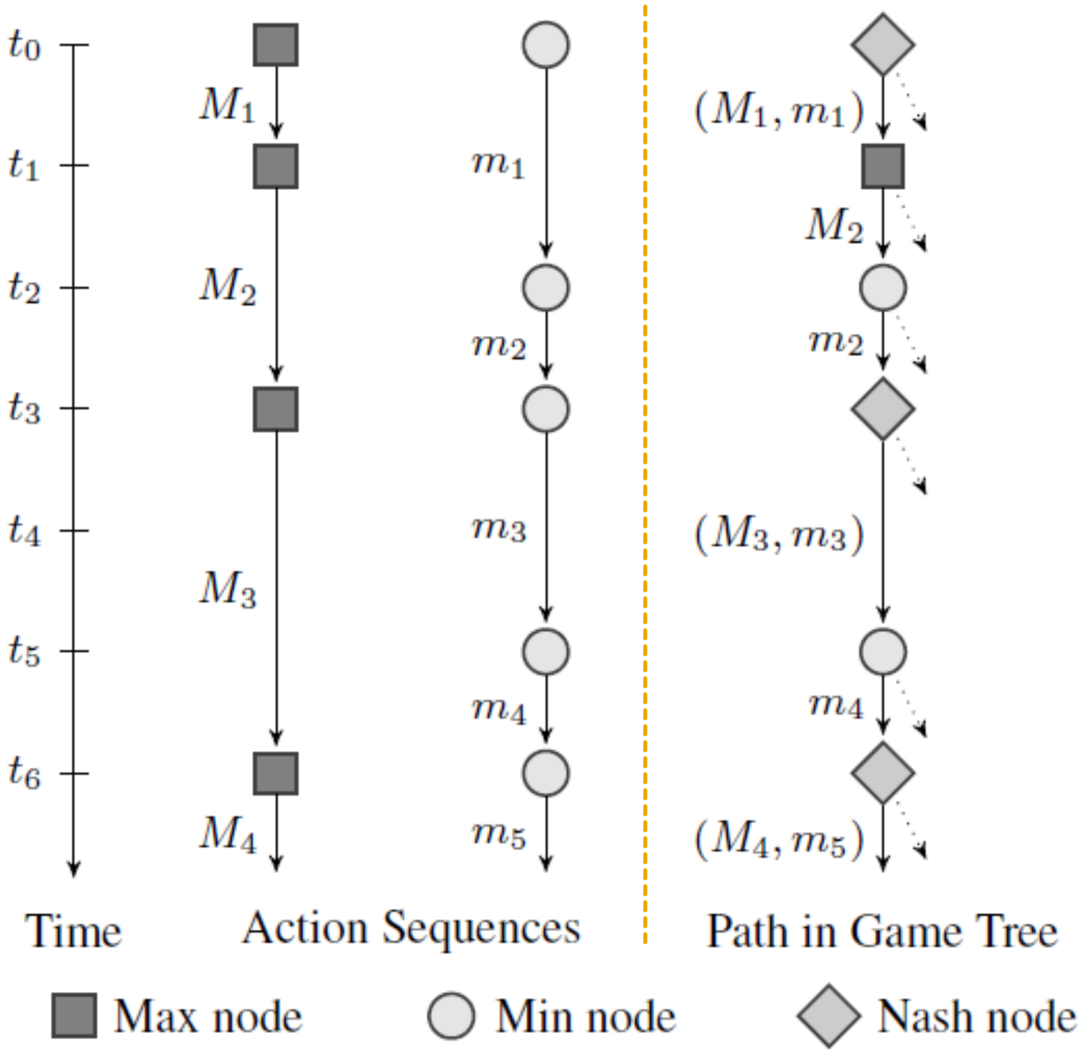
As an approximation of Nash equilibrium

Quick recap

Minimax (or maximin)	recursive algorithm to find the best move in sequential non-cooperative games
Alpha-beta pruning	technique improving minimax by pruning game tree of branches, which cannot bring better results than already found
Move ordering	heuristically ordering possible moves from (seemingly) better to worse to improve AB pruning
Evaluation function	a function evaluating a state used at depths where we stop searching
Iterative deepening	search technique where we incrementally increasing search depths until time runs out; allows for any-time results
Transposition tables	cache to maintain previously seen states that allows to reuse results especially during iterative deepening

Durative actions

The problem of simultaneous games



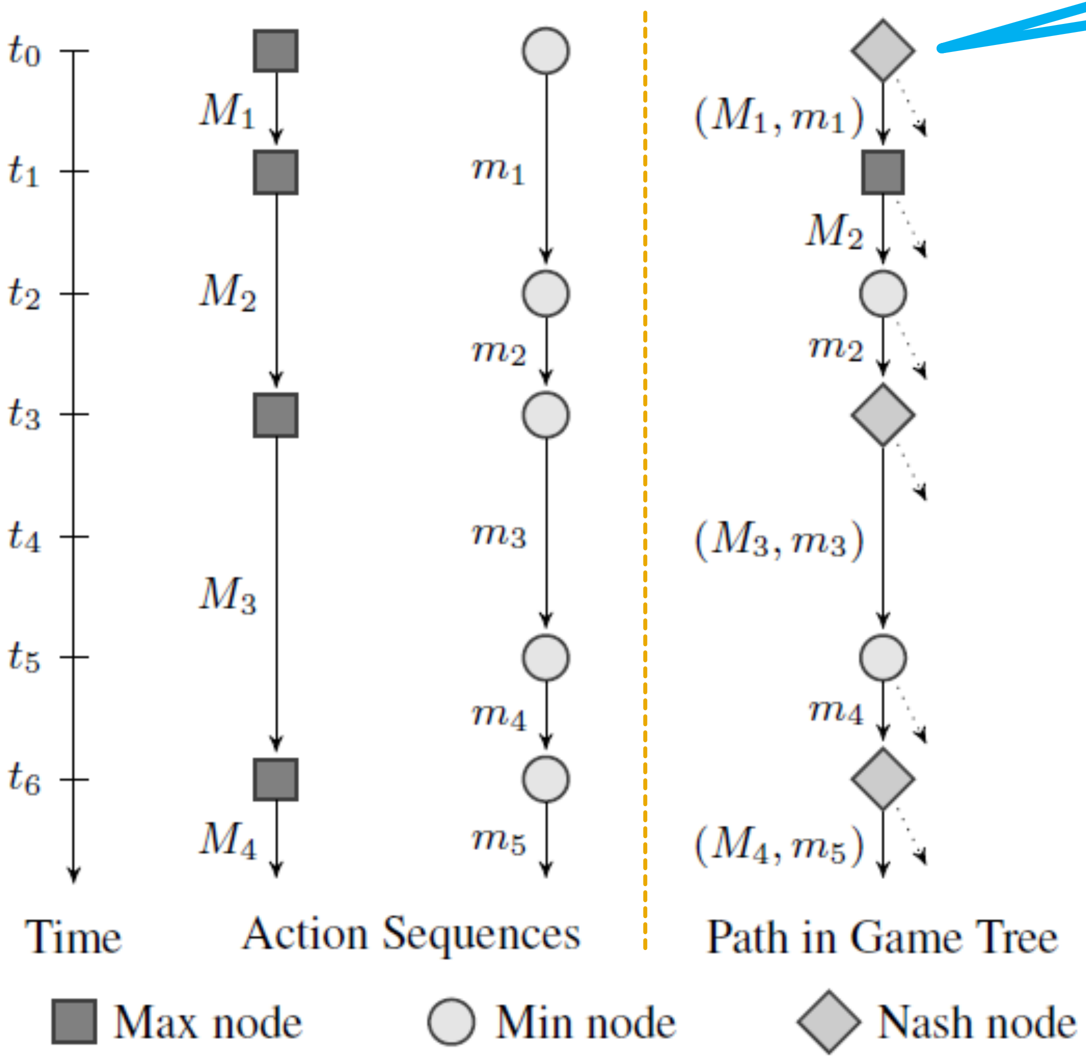
In sequential games, players alternate; in RTS combat, they might not. The same player might decide multiple times (easy to handle) and there are points in time where players decide on moves simultaneously (troublesome).

We need to adapt standard MINMAX (alpha-beta) algorithm for durative actions.

Durative actions

The problem of simultaneous gam

How to deal with nodes, where players are deciding on actions simultaneously (labeled as Nash nodes)?

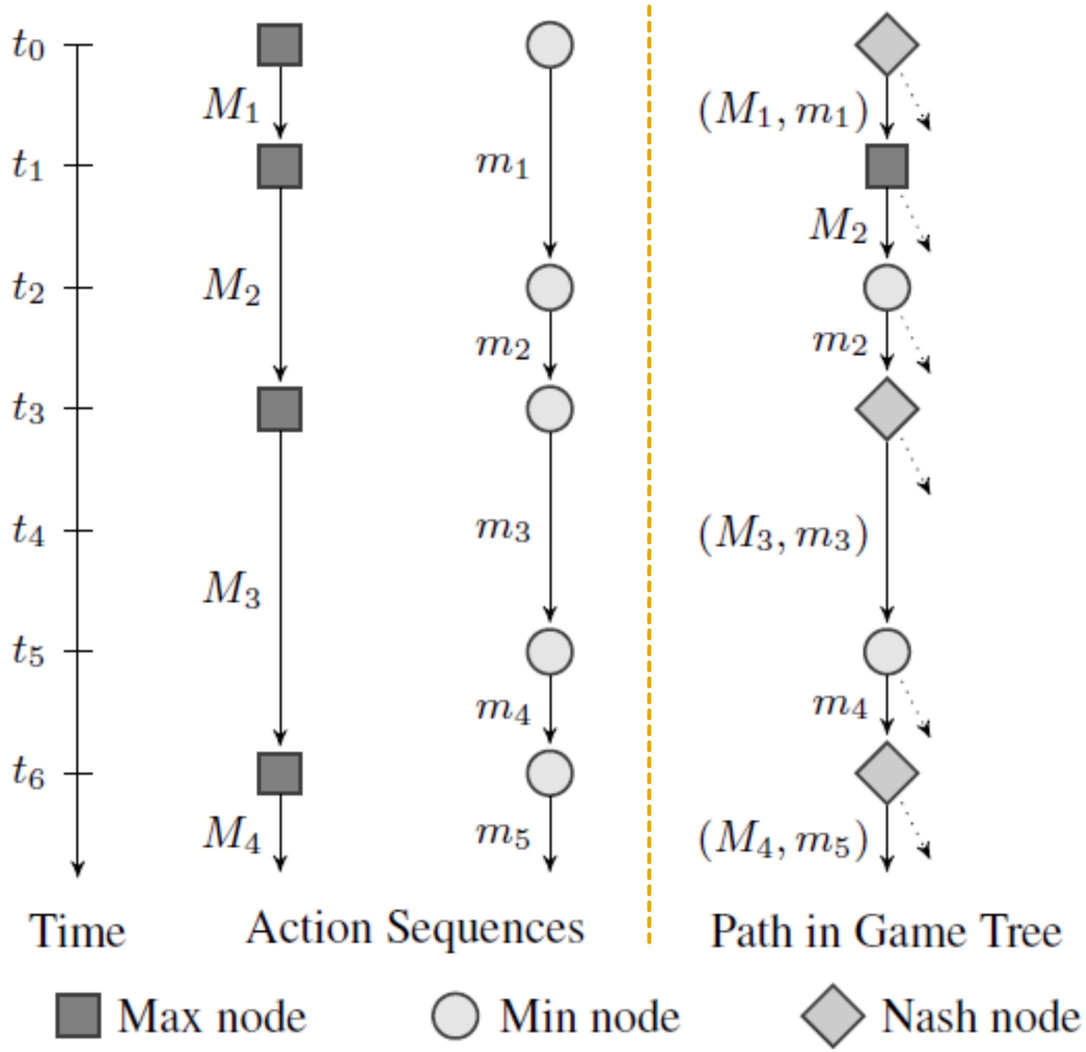


In sequential games, players alternate; in RTS combat, they might not. The same player might decide multiple times (easy to handle) and there are points in time where players decide on moves simultaneously (troublesome).

We need to adapt standard MINMAX (alpha-beta) algorithm for durative actions.

Stacked matrix games

Dealing with "simultaneous move" node



In simultaneous nodes, one may employ normal form of description for the node, i.e., both players decide at once, which leads to the full matrix of decisions and computing their payoffs.

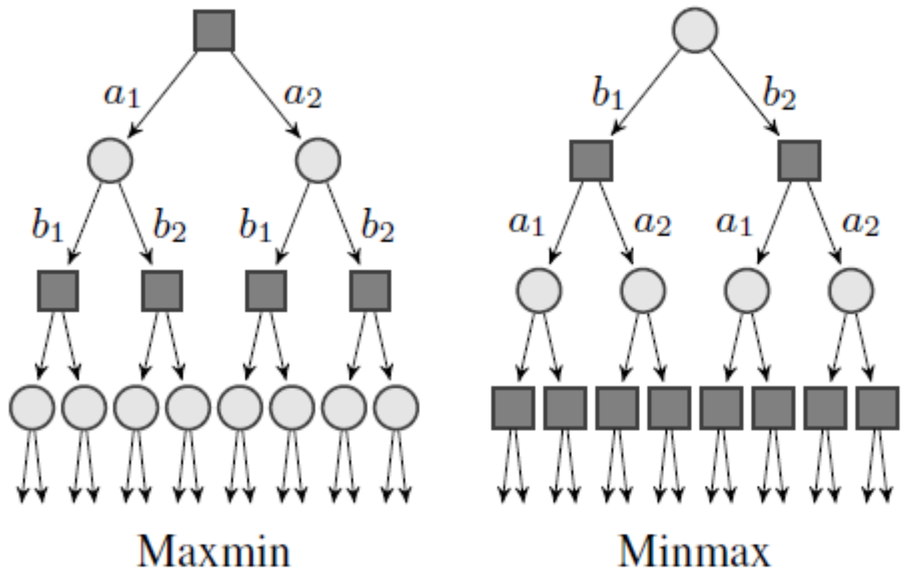
Such payoffs cannot be determined immediately, and we need to continue the search to determine them.

Alternate Alpha-Beta (ALT)

(policy for serialization of simultaneous nodes)

Alternate Alpha-Beta (RAB)

Serializing the simultaneous node



Assumption: not really, simultaneous node can be encountered anywhere along the search

Idea: Once simultaneous node is reached, alternate between maxmin and minimax e.g. on the first sim-node use maxmin, on the second minmax, etc.

Alpha-Beta Considering Durations
(ABCD)

Alpha-Beta Considering Durations

a.k.a. ABCD

Algorithm 1 Alpha-Beta (Considering Durations)

procedure ABCD(s, d, m_0, α, β)
    if computationTime.elapsed **then return** timeout
    else if terminal(s, d) **then return** eval(s)
    toMove $\leftarrow$ s.playerToMove(policy)
    while $m \leftarrow s$.nextMove(toMove) **do**
        if s.bothCanMove and $m_0 = \emptyset$ and $d \neq 1$ **then**
            $val \leftarrow$ABCD($s, d-1, m, \alpha, \beta$)
        else
            $s' \leftarrow$ copy(s)
            if $m_0 \neq \emptyset$ **then** s'.doMove(m_0)
            s'.doMove(m)
            $v \leftarrow$ABCD($s', d-1, \emptyset, \alpha, \beta$)
        if toMove $= MAX$ and ($v > \alpha$) **then** $\alpha \leftarrow v$
        if toMove $= MIN$ and ($v < \beta$) **then** $\beta \leftarrow v$
        if $\alpha \geq \beta$ **then break**
    return toMove $= MAX$? $\alpha : \beta$

s — state
d — depth to search
$\mathbf{m_0}$ — delayed action effect used for simultaneous nodes
$\boldsymbol{\alpha}, \boldsymbol{\beta}$ — bounds

Meant to be used with iterative deepening.

Alpha-Beta Considering Durations

a.k.a. ABCD

Algorithm 1 Alpha-Beta (Considering Durations)

procedure ABCD(s, d, m_0, α, β)
    if computationTime.elapsed **then return** timeout
    else if terminal(s, d) **then return** eval(s)
    toMove $\leftarrow$ s.playerToMove(policy)
    while $m \leftarrow s$.nextMove(toMove) **do**
        if s.bothCanMove and $m_0 = \emptyset$ and $d \neq 1$ **then**
            $val \leftarrow$ABCD($s, d-1, m, \alpha, \beta$)
        else
            $s' \leftarrow$ copy(s)
            if $m_0 \neq \emptyset$ **then** s'.doMove(m_0)
            s'.doMove(m)
            $v \leftarrow$ABCD($s', d-1, \emptyset, \alpha, \beta$)
        if toMove $= MAX$ and $(v > \alpha)$ **then** $\alpha \leftarrow v$
        if toMove $= MIN$ and $(v < \beta)$ **then** $\beta \leftarrow v$
        if $\alpha \geq \beta$ **then break**
    return toMove $= MAX$? $\alpha : \beta$

First, mind the real-time constraints. Note that this ABCD should be run in "iterative deepening" manner, thus timeout means "do not use this result at all".

Alpha-Beta Considering Durations

a.k.a. ABCD

Algorithm 1 Alpha-Beta (Considering Durations)

procedure ABCD(s, d, m_0, α, β)
    if computationTime.elapsed **then return** timeout
    else if terminal(s, d) **then return** eval(s)
    toMove $\leftarrow$ s.playerToMove(policy)
    while $m \leftarrow s$.nextMove(toMove) **do**
        if s.bothCanMove and $m_0 = \emptyset$ and $d \neq 1$ **then**
            $val \leftarrow$ABCD$(s, d-1, m, \alpha, \beta)$
        else
            $s' \leftarrow$ copy(s)
            if $m_0 \neq \emptyset$ **then** s'.doMove(m_0)
            s'.doMove(m)
            $v \leftarrow$ABCD$(s', d-1, \emptyset, \alpha, \beta)$
        if toMove $= MAX$ and $(v > \alpha)$ **then** $\alpha \leftarrow v$
        if toMove $= MIN$ and $(v < \beta)$ **then** $\beta \leftarrow v$
        if $\alpha \geq \beta$ **then break**
    return toMove $= MAX$? $\alpha : \beta$

If we are in terminal node (either depth reaches zero or maximal time for scenario is reached), we return evaluation of the state.

Alpha-Beta Considering Durations

a.k.a. ABCD

Algorithm 1 Alpha-Beta (Considering Durations)

procedure ABCD$(s, d, m_0, \alpha, \beta)$
    if computationTime.elapsed **then return** timeout
    else if terminal(s, d) **then return** eval(s)
    toMove $\leftarrow$ s.playerToMove(policy)
    while $m \leftarrow s$.nextMove(toMove) **do**
        if s.bothCanMove and $m_0 = \emptyset$ and $d \neq 1$ **then**
            $val \leftarrow$ABCD$(s, d-1, m, \alpha, \beta)$
        else
            $s' \leftarrow$ copy(s)
            if $m_0 \neq \emptyset$ **then** s'.doMove(m_0)
            s'.doMove(m)
            $v \leftarrow$ABCD$(s', d-1, \emptyset, \alpha, \beta)$
        if toMove $= MAX$ and $(v > \alpha)$ **then** $\alpha \leftarrow v$
        if toMove $= MIN$ and $(v < \beta)$ **then** $\beta \leftarrow v$
        if $\alpha \geq \beta$ **then break**
    return toMove $= MAX$? α : β

This line condenses a lot of stuff.

[A] If no unit can perform actions (different from "pass"), advance the time to the point some unit may perform an action first.

Alpha-Beta Considering Durations

a.k.a. ABCD

Algorithm 1 Alpha-Beta (Considering Durations)

procedure ABCD(s, d, m_0, α, β)
    if computationTime.elapsed **then return** timeout
    else if terminal(s, d) **then return** eval(s)
    toMove $\leftarrow$ s.playerToMove(policy)
    while $m \leftarrow s$.nextMove(toMove) **do**
        if s.bothCanMove and $m_0 = \emptyset$ and $d \neq 1$ **then**
            $val \leftarrow$ABCD($s, d-1, m, \alpha, \beta$)
        else
            $s' \leftarrow$ copy(s)
            if $m_0 \neq \emptyset$ **then** s'.doMove(m_0)
            s'.doMove(m)
            $v \leftarrow$ABCD($s', d-1, \emptyset, \alpha, \beta$)
        if toMove $= MAX$ and ($v > \alpha$) **then** $\alpha \leftarrow v$
        if toMove $= MIN$ and ($v < \beta$) **then** $\beta \leftarrow v$
        if $\alpha \geq \beta$ **then break**
    return toMove $= MAX$? $\alpha : \beta$

This line condenses a lot of stuff.

[B] Given the current state, i.e., state of units, determine which players can move.

If at this stage, we are in simultaneous node, use "policy" to determine, which player will make its decision first.

Alpha-Beta Considering Durations

a.k.a. ABCD

Algorithm 1 Alpha-Beta (Considering Durations)

procedure ABCD(s, d, m_0, α, β)
    if computationTime.elapsed **then return** timeout
    else if terminal(s, d) **then return** eval(s)
    toMove $\leftarrow$ s.playerToMove(policy)
    while $m \leftarrow s$.nextMove(toMove) **do**
        if s.bothCanMove and $m_0 = \emptyset$ and $d \neq 1$ **then**
            $val \leftarrow$ABCD($s, d-1, m, \alpha, \beta$)
        else
            $s' \leftarrow$ copy(s)
            if $m_0 \neq \emptyset$ **then** s'.doMove(m_0)
            s'.doMove(m)
            $v \leftarrow$ABCD($s', d-1, \emptyset, \alpha, \beta$)
        if toMove $= MAX$ and ($v > \alpha$) **then** $\alpha \leftarrow v$
        if toMove $= MIN$ and ($v < \beta$) **then** $\beta \leftarrow v$
        if $\alpha \geq \beta$ **then break**
    return toMove $= MAX$? $\alpha : \beta$

Next, we are iterate over moves player "toMove" can do, save current batch of moves we're going to investigate into ***m***.

Alpha-Beta Considering Durations

a.k.a. ABCD

Algorithm 1 Alpha-Beta (Considering Durations)

procedure ABCD(s, d, m_0, α, β)
    if computationTime.elapsed **then return** timeout
    else if terminal(s, d) **then return** eval(s)
    toMove $\leftarrow$ s.playerToMove(policy)
    while $m \leftarrow s$.nextMove(toMove) **do**
        if s.bothCanMove and $m_0 = \emptyset$ and $d \neq 1$ **then**
            $val \leftarrow$ABCD$(s, d-1, m, \alpha, \beta)$
        else
            $s' \leftarrow$ copy(s)
            if $m_0 \neq \emptyset$ **then** s'.doMove(m_0)
            s'.doMove(m)
            $v \leftarrow$ABCD$(s', d-1, \emptyset, \alpha, \beta)$
        if toMove $= MAX$ and $(v > \alpha)$ **then** $\alpha \leftarrow v$
        if toMove $= MIN$ and $(v < \beta)$ **then** $\beta \leftarrow v$
        if $\alpha \geq \beta$ **then break**
    return toMove $= MAX$? $\alpha : \beta$

If we are in simultaneous node, and there is nothing in $\mathbf{m_0}$ buffer, i.e., this ABCD has not been called from simultaneous node …

Alpha-Beta Considering Durations
a.k.a. ABCD

Algorithm 1 Alpha-Beta (Considering Durations)

procedure ABCD$(s, d, m_0, \alpha, \beta)$
  if computationTime.elapsed **then return** timeout
  else if terminal(s, d) **then return** eval(s)
  toMove $\leftarrow s$.playerToMove(policy)
  while $m \leftarrow s$.nextMove(toMove) **do**
    if s.bothCanMove and $m_0 = \emptyset$ and $d \neq 1$ **then**
      $val \leftarrow$ABCD$(s, d-1, m, \alpha, \beta)$
    else
      $s' \leftarrow$ copy(s)
      if $m_0 \neq \emptyset$ **then** s'.doMove(m_0)
      s'.doMove(m)
      $v \leftarrow$ABCD$(s', d-1, \emptyset, \alpha, \beta)$
    if toMove $= MAX$ and $(v > \alpha)$ **then** $\alpha \leftarrow v$
    if toMove $= MIN$ and $(v < \beta)$ **then** $\beta \leftarrow v$
    if $\alpha \geq \beta$ **then break**
  return toMove $= MAX$? $\alpha : \beta$

… then we continue with next ply of ABCD, but (!) passing actions "m" as an argument.

This "m" will act as $\mathbf{m_0}$ in next invocation, so we will not get into this branch next time.

Alpha-Beta Considering Durations

a.k.a. ABCD

Algorithm 1 Alpha-Beta (Considering Durations)

procedure ABCD(s, d, m_0, α, β)
    if computationTime.elapsed **then return** timeout
    else if terminal(s, d) **then return** eval(s)
    toMove $\leftarrow$ s.playerToMove(policy)
    while $m \leftarrow s$.nextMove(toMove) **do**
        if s.bothCanMove and $m_0 = \emptyset$ and $d \neq 1$ **then**
            $val \leftarrow$ABCD($s, d-1, m, \alpha, \beta$)
        else
            $s' \leftarrow$ copy(s)
            if $m_0 \neq \emptyset$ **then** s'.doMove(m_0)
            s'.doMove(m)
            $v \leftarrow$ABCD($s', d-1, \emptyset, \alpha, \beta$)
        if toMove $= MAX$ and ($v > \alpha$) **then** $\alpha \leftarrow v$
        if toMove $= MIN$ and ($v < \beta$) **then** $\beta \leftarrow v$
        if $\alpha \geq \beta$ **then break**
    return toMove $= MAX$? $\alpha : \beta$

Additionally, if we are near the end of the search depth, we do not bother resolving simultaneous node as we're terminating anyway.

Alpha-Beta Considering Durations

a.k.a. ABCD

Algorithm 1 Alpha-Beta (Considering Durations)

procedure ABCD(s, d, m_0, α, β)
    if computationTime.elapsed **then return** timeout
    else if terminal(s, d) **then return** eval(s)
    toMove $\leftarrow$ s.playerToMove(policy)
    while $m \leftarrow s$.nextMove(toMove) **do**
        if s.bothCanMove and $m_0 = \emptyset$ and $d \neq 1$ **then**
            $val \leftarrow$ABCD($s, d-1, m, \alpha, \beta$)
        else
            $s' \leftarrow$ copy(s)
            if $m_0 \neq \emptyset$ **then** s'.doMove(m_0)
            s'.doMove(m)
            $v \leftarrow$ABCD($s', d-1, \emptyset, \alpha, \beta$)
        if toMove $= MAX$ and ($v > \alpha$) **then** $\alpha \leftarrow v$
        if toMove $= MIN$ and ($v < \beta$) **then** $\beta \leftarrow v$
        if $\alpha \geq \beta$ **then break**
    return toMove $= MAX$? α : β

The else branch is then about solving the "delayed action" effect.

Alpha-Beta Considering Durations

a.k.a. ABCD

Algorithm 1 Alpha-Beta (Considering Durations)

procedure ABCD$(s, d, m_0, \alpha, \beta)$
  if computationTime.elapsed **then return** timeout
  else if terminal(s, d) **then return** eval(s)
  toMove $\leftarrow$ s.playerToMove(policy)
  while $m \leftarrow s$.nextMove(toMove) **do**
    if s.bothCanMove and $m_0 = \emptyset$ and $d \neq 1$ **then**
      $val \leftarrow$ABCD$(s, d-1, m, \alpha, \beta)$
    else
      $s' \leftarrow$ copy(s)
      if $m_0 \neq \emptyset$ **then** s'.doMove(m_0)
      s'.doMove(m)
      $v \leftarrow$ABCD$(s', d-1, \emptyset, \alpha, \beta)$
    if toMove $= MAX$ and $(v > \alpha)$ **then** $\alpha \leftarrow v$
    if toMove $= MIN$ and $(v < \beta)$ **then** $\beta \leftarrow v$
    if $\alpha \geq \beta$ **then break**
  return toMove $= MAX$? $\alpha : \beta$

First, this version of ABCD is not using reversible action. So even though we are performing DFS, we clone the state.

Alpha-Beta Considering Durations

a.k.a. ABCD

Algorithm 1 Alpha-Beta (Considering Durations)

procedure ABCD(s, d, m_0, α, β)
    if computationTime.elapsed **then return** timeout
    else if terminal(s, d) **then return** eval(s)
    toMove $\leftarrow$ s.playerToMove(policy)
    while $m \leftarrow s$.nextMove(toMove) **do**
        if s.bothCanMove and $m_0 = \emptyset$ and $d \neq 1$ **then**
            $val \leftarrow$ABCD($s, d-1, m, \alpha, \beta$)
        else
            $s' \leftarrow$ copy(s)
            if $m_0 \neq \emptyset$ **then** s'.doMove(m_0)
            s'.doMove(m)
            $v \leftarrow$ABCD($s', d-1, \emptyset, \alpha, \beta$)
        if toMove $= MAX$ and ($v > \alpha$) **then** $\alpha \leftarrow v$
        if toMove $= MIN$ and ($v < \beta$) **then** $\beta \leftarrow v$
        if $\alpha \geq \beta$ **then break**
    return toMove $= MAX$? $\alpha : \beta$

Then, we are solving the delayed action effect, if $\mathbf{m_0}$ is containing some actions, we apply them here …

Alpha-Beta Considering Durations

a.k.a. ABCD

Algorithm 1 Alpha-Beta (Considering Durations)

procedure ABCD(s, d, m_0, α, β)
    if computationTime.elapsed **then return** timeout
    else if terminal(s, d) **then return** eval(s)
    toMove $\leftarrow$ s.playerToMove(policy)
    while $m \leftarrow s$.nextMove(toMove) **do**
        if s.bothCanMove and $m_0 = \emptyset$ and $d \neq 1$ **then**
            $val \leftarrow$ABCD$(s, d-1, m, \alpha, \beta)$
        else
            $s' \leftarrow$ copy(s)
            if $m_0 \neq \emptyset$ **then** s'.doMove(m_0)
            s'.doMove(m)
            $v \leftarrow$ABCD$(s', d-1, \emptyset, \alpha, \beta)$
        if toMove $= MAX$ and $(v > \alpha)$ **then** $\alpha \leftarrow v$
        if toMove $= MIN$ and $(v < \beta)$ **then** $\beta \leftarrow v$
        if $\alpha \geq \beta$ **then break**
    return toMove $= MAX$? $\alpha : \beta$

… before applying currently selected actions.

Alpha-Beta Considering Durations

a.k.a. ABCD

Algorithm 1 Alpha-Beta (Considering Durations)

procedure ABCD$(s, d, m_0, \alpha, \beta)$
  if computationTime.elapsed **then return** timeout
  else if terminal(s, d) **then return** eval(s)
  toMove $\leftarrow$ s.playerToMove(policy)
  while $m \leftarrow s$.nextMove(toMove) **do**
    if s.bothCanMove and $m_0 = \emptyset$ and $d \neq 1$ **then**
      $val \leftarrow$ABCD$(s, d-1, m, \alpha, \beta)$
    else
      $s' \leftarrow$ copy(s)
      if $m_0 \neq \emptyset$ **then** s'.doMove(m_0)
      s'.doMove(m)
      $v \leftarrow$ABCD$(s', d-1, \emptyset, \alpha, \beta)$
    if toMove $= MAX$ and $(v > \alpha)$ **then** $\alpha \leftarrow v$
    if toMove $= MIN$ and $(v < \beta)$ **then** $\beta \leftarrow v$
    if $\alpha \geq \beta$ **then break**
  return toMove $= MAX$? $\alpha : \beta$

The rest of the algorithm is standard alpha-beta pruning.

State Evaluation Function

(eval)

State evaluation function

eval

Algorithm 1 Alpha-Beta (Considering Durations)

procedure ABCD$(s, d, m_0, \alpha, \beta)$
    if computationTime.elapsed **then return** timeout
    else if terminal(s, d) **then return** eval(s)
    toMove $\leftarrow$ s.playerToMove(policy)
    while $m \leftarrow s$.nextMove(toMove) **do**
        if s.bothCanMove and $m_0 = \emptyset$ and $d \neq 1$ **then**
            $val \leftarrow$ABCD$(s, d-1, m, \alpha, \beta)$
        else
            $s' \leftarrow$ copy(s)
            if $m_0 \neq \emptyset$ **then** s'.doMove(m_0)
            s'.doMove(m)
            $v \leftarrow$ABCD$(s', d-1, \emptyset, \alpha, \beta)$
        if toMove $= MAX$ and $(v > \alpha)$ **then** $\alpha \leftarrow v$
        if toMove $= MIN$ and $(v < \beta)$ **then** $\beta \leftarrow v$
        if $\alpha \geq \beta$ **then break**
    return toMove $= MAX$? $\alpha : \beta$

The state evaluation function is used here to evaluate nodes at certain depths.

State evaluation function

eval

3: **else if** terminal(s, d) **then return** eval(s)

$$\mathrm{LTD}(s) = \sum_{u \in U_1} \mathrm{hp}(u) \cdot \mathrm{dpf}(u) - \sum_{u \in U_2} \mathrm{hp}(u) \cdot \mathrm{dpf}(u)$$

$$\mathrm{LTD2}(s) = \sum_{u \in U_1} \sqrt{\mathrm{hp}(u)} \cdot \mathrm{dpf}(u) - \sum_{u \in U_2} \sqrt{\mathrm{hp}(u)} \cdot \mathrm{dpf}(u)$$

LTD3(s): playout

- instead of evaluation finish the game by performing a playout
- i.e., on each unit decision point use preselected script to select an action
- play until either or both sides are annihilated

Move Ordering
in
ABCD Iterative Deepening

Move Ordering

Sequence of trying out the actions

Algorithm 1 Alpha-Beta (Considering Durations)

procedure ABCD(s, d, m_0, α, β)
    if computationTime.elapsed **then return** timeout
    else if terminal(s, d) **then return** eval(s)
    toMove $\leftarrow$ s.playerToMove(policy)
    while $m \leftarrow s$.nextMove(toMove) **do**
        if s.bothCanMove and $m_0 = \emptyset$ and $d \neq 1$ **then**
            $val \leftarrow$ABCD($s, d-1, m, \alpha, \beta$)
        else
            $s' \leftarrow$ copy(s)
            if $m_0 \neq \emptyset$ **then** s'.doMove(m_0)
            s'.doMove(m)
            $v \leftarrow$ABCD($s', d-1, \emptyset, \alpha, \beta$)
        if toMove $= MAX$ and ($v > \alpha$) **then** $\alpha \leftarrow v$
        if toMove $= MIN$ and ($v < \beta$) **then** $\beta \leftarrow v$
        if $\alpha \geq \beta$ **then break**
    return toMove $= MAX$? $\alpha : \beta$

If move ordering is done right, it improves the effect of alpha-beta pruning as better state values are found faster.

Move Ordering

Sequence of trying out the actions

Algorithm 1 Alpha-Beta (Considering Durations)

procedure ABCD(s, d, m_0, α, β)
  if computationTime.elapsed **then return** timeout
  else if terminal(s, d) **then return** eval(s)
  toMove $\leftarrow$ s.playerToMove(policy)
  while $m \leftarrow s$.nextMove(toMove) **do**
    if s.bothCanMove and $m_0 = \emptyset$ and $d \neq 1$
      $val \leftarrow$ABCD($s, d-1, m, \alpha, \beta$)
    else
      $s' \leftarrow$ copy(s)
      if $m_0 \neq \emptyset$ **then** s'.doMove(m_0)
      s'.doMove(m)
      $v \leftarrow$ABCD($s', d-1, \emptyset, \alpha, \beta$)
    if toMove $= MAX$ and ($v > \alpha$) **then** $\alpha \leftarrow v$
    if toMove $= MIN$ and ($v < \beta$) **then** $\beta \leftarrow v$
    if $\alpha \geq \beta$ **then break**
  return toMove $= MAX$? α : β

As we are running ABCD using iterative deepening, we can store information about promising moves from previous runs. This can be used to sort actions in consecutives ABCD invocations, allowing it to run faster. If no such information is available, we can still use a script to suggest "first move".

Paper Results

Setup
Paper Results

N vs. N combats in Star Craft; N ranged 2-8

Four different army types: Marine only, Marine+Zergling, Dragoon + Zealot, Dragoon + Marine (all combinations, up-to 4 of each unit type)

Symmetric starting locations

Max 500 actions, after that LTD was used to determine the winner.

Final player score: score = (#wins + #draws / 2) / #matches

Scripts vs. Searches

Paper Results

Alt': The difference from described Alt is, that in simultaneous nodes, we pick the player to move first as the one who moved last.

Opponent	ABCD Search Setting				
	Alt LTD	Alt LTD2	Alt NOK-AV	Alt' NOK-AV	RAB' Playout
Random	0.99	0.98	1.00	1.00	1.00
Kite	0.70	0.79	0.93	0.93	0.92
Kite-AV	0.69	0.81	0.92	0.96	0.92
Closest	0.59	0.85	0.92	0.92	0.93
Weakest	0.41	0.76	0.91	0.91	0.89
AV	0.42	0.76	0.90	0.90	0.91
NOK-AV	0.32	0.64	0.87	0.87	0.82
Average	0.59	0.80	0.92	0.92	0.91

Searches vs. Searches

Paper Results

Table 2: Playout-based ABCD performance

Opponent	Alt	Alt'	RAB'
	NOK-AV Playout		
Alt-NOK-AV		0.47	0.46
Alt'-NOK-AV	0.53		0.46
RAB'-NOK-AV	0.54	0.54	
Average	0.54	0.51	0.46

Scripts exploitability

Paper Results

Table 3: Real-time exploitability of scripted strategies.

Random	Weakest	Closest	AV	Kiter	Kite-AV	NOK-AV
1.00	0.98	0.98	0.98	0.97	0.97	0.95

Results of search with opponent modeling, i.e., the other player in the search was modeled by the exact script the search has been playing against.

We can see, that scripts are highly exploitable!