CSC384
Game Tree Search
Part 1

Bahar Aameri & Sonya Allin

Winter 2020

Credits

These slides are drawn from or inspired by a multitude of sources including :

Faheim Bacchus
Sheila McIlraith
Andrew Moore
Hojjat Ghaderi
Craig Boutillier
Jurgen Strum
Shaul Markovitch

- Chapter 5
  - Chapter 5.1, 5.2, 5.3 cover some of the material we cover here.
  - Section 5.5 extends the ideas to games with uncertainty (We won't cover that material but it makes for interesting reading).
  - Section 5.6 has an interesting overview of State-of-the-Art game playing programs.

Generalizing Search Problem

- So far our search problems have assumed **agent has complete control** of environment:
  - State does not change unless the **agent changes it**.
  - All we need to compute is a single **path to a goal** state.
- This assumption is **not** always reasonable:
  - **Stochastic** environment (e.g., the weather, traffic accidents).
  - **Other agents** whose interests conflict with yours.
    Search can find a path to a goal state, but the actions might not lead you to the goal as the state can be changed by other agents.
- We need to **generalize** our view of search to handle state changes that are not in the control of our agent.
  - 2 or more agents;
  - All agents acting to maximize their own profits.

General Games

What makes something **a game**?

- There are two (or more) agents making changes to the **world** (the state).
- Each agent has their own interests and goals.
  Each agent assigns **different costs** to different paths/states.
- Each agent independently tries to alter the world so as to best benefit itself.
- **Co-operation** can occur but only if it benefits both parties.

What makes games **hard**?

- How you should play depends on how you **think** the other person will play;
- How the other person plays depends on how they **think** you will play.

Hence, a joint-dependency.

Properties of Games

- **Two player:**
  Note: Algorithms presented can be extended to multiplayer games, but multi-player games can involve alliances where some players cooperate to defeat another player (see Chapter 5.2.2)

- **Finite:** Finite number of states and moves from each state.
  - Techniques can be extended to deal with infinite games by applying heuristic cutoffs.
  - When the game is too large finite becomes as bad as infinite and heuristic cutoffs need to be used.

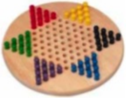
- **Zero-sum:** Fully competitive, **total payoff** to all players is constant.
  If one player gets a higher payoff, the other player gets a lower payoff.
  Example: Poker – you win what the other player lose

- **Deterministic:** No chances involved, no dice, or random deals of cards, or coin flips.

- **Perfect Information:** All aspects of the state are fully observable.
  Example: Chess.

Which of these are: 2-player zero-sum discrete finite deterministic games of perfect information

- **Two player:** Duh!
- **Zero-sum:** In any outcome of any game, Player A's gains equal player B's losses.
- **Discrete:** All game states and decisions are discrete values.
- **Finite:** Only a finite number of states and decisions.
- **Deterministic:** No chance (no die rolls).
- **Perfect information:** Both players can see the state, and each decision is made sequentially (no simultaneous moves).



Which of these are: 2-player zero-sum discrete finite deterministic games of perfect information

- Two player: Duh!
- Zero-sum: In any outcome of any game, Player A's gains equal player B's losses.
- Discrete: All game states and decisions are discrete values.
- Finite: Only a finite number of states and decisions.
- Deterministic: No chance (no die rolls).
- Perfect information: Both players can see the state, and each decision is made sequentially (no simultaneous moves).

- Scissors cut paper, paper covers rock, rock smashes scissors
- Represented as a **matrix**:
  Player I chooses a row, Player II chooses a column.
- 1: win
  0: tie
  -1: loss

	R	P	S
R	0/0	-1/1	1/-1
P	1/-1	0/0	-1/1
S	-1/1	1/-1	0/0

Game 2: Prisoner's Dilemma

- Two prisoners in separate cells.
  The sheriff doesn't have enough evidence to convict them.
  They agree ahead of time to both **deny** the crime (they will **cooperate**).

- If one confesses and the other doesn't:
  - Confessor goes free;
  - Other sentenced to 4 years.

- If both confess:
  - both sentenced to 3 years.

- If both cooperate (neither confesses):
  - both sentenced to 1 year.

	Coop	Confess
Coop	1/1	4/0
Confess	0/4	3/3

Payoff: 4 minus sentence

Extensive Form Two-Player Zero-Sum Games

- The previous games are simple "**one shot**" games.

- Many games extend over multiple moves turn-taking: players act alternatively.
  Examples: chess, checkers, etc.

Two-Player Zero-Sum Game – Definition

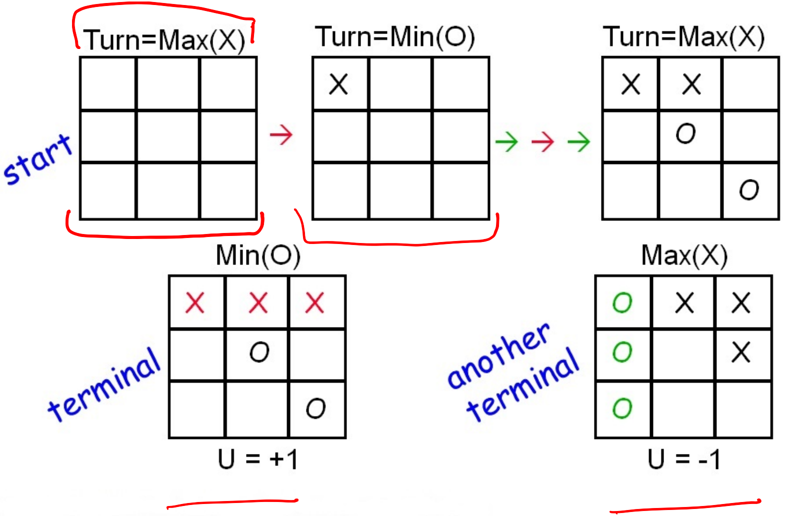
A **Two-Player Zero-Sum** game consists of the following components:

- Two **players** Max and Min.
- A set of **positions** P (states of the game).
- A **starting position** $p \in P$ (where game begins).
- A set of **Terminal positions** $T \subseteq P$ (where game can end).
- A set of directed **edges** E_{Max} between some positions, representing Max's moves.
- A set of directed **edges** E_{Min} between some positions, representing Min's moves.
- A **utility (or payoff) function** $U : T \rightarrow \mathbb{R}$, representing how good each **terminal state** is for player **Max**.

Why don't we need a utility function for Min?

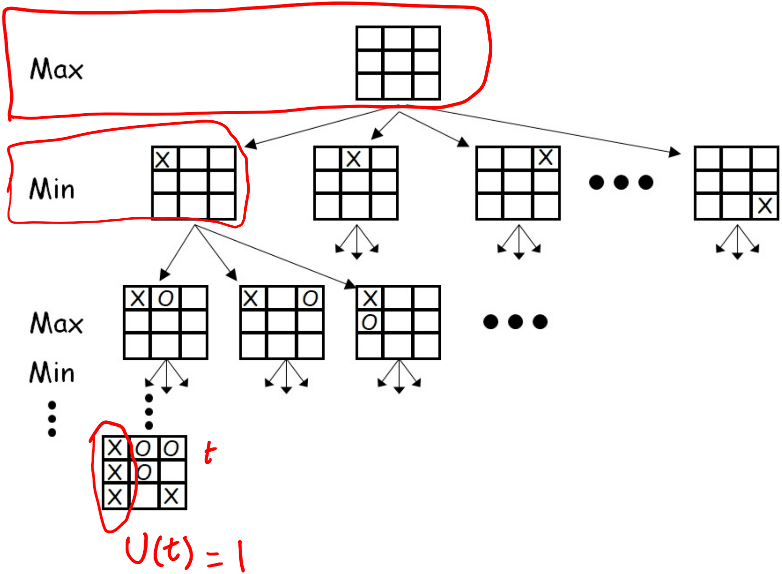
$U(t)$

$-U(t)$


Turn=Max(X)
Turn=Min(O)
Turn=Max(X)
start
X
X X
O
O
Min(O)
X X X
O
O
terminal
U = +1
Max(X)
O X X
O X
O
another terminal
U = -1

Game Tree

- A **Game Tree** consists of **layers** reflect **alternating moves** between Max and Min.
- **Root** is **start** state.
- **Starting** with Max, players alternate moves.
- **Game State:** a state-player **pair**, specifies the current state and whose turn it is.
- **Game ends** when some terminal $p \in T$ is reached.
- **Utility function** and terminals replace goals:
  - Terminal nodes t are labeled with utilities $U(t)$.
  - Max gets $U(t)$, Min gets $-U(t)$ for terminal node t.


Max
Min
Max
Min
t
U(t) = 1

Game Playing Strategies

- Max wants to maximize the terminal payoff.
- Min wants to minimize the terminal payoff.
- Max **doesn't** decide which terminal state is reached alone.
  After Max moves to a state, Min decides **which subsequent** state to move to.
- Thus Max must have a **strategy**:
  - Must know what to do for **each possible move** of Min.
  - One sequence of moves will not suffice: "What to do" will depend on how Min will play.

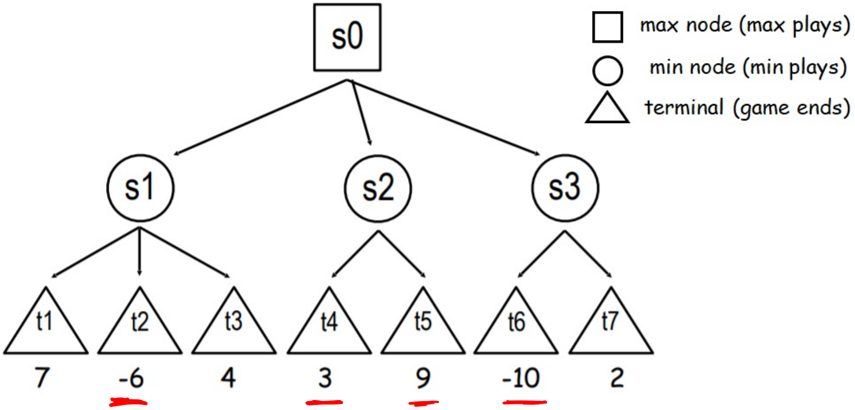
Minimax Strategy

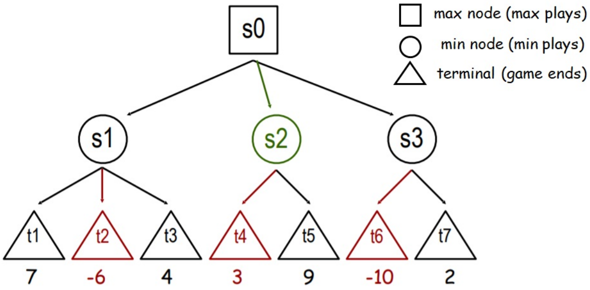
- **Minimax Strategy:** Assuming that the other player will always play its best move, play a move that will **minimize** the payoff that could be gained by the **other player**.

- Minimizing the other player's payoff to maximize yours.

Minimax plays it safe!

Minimax Strategy



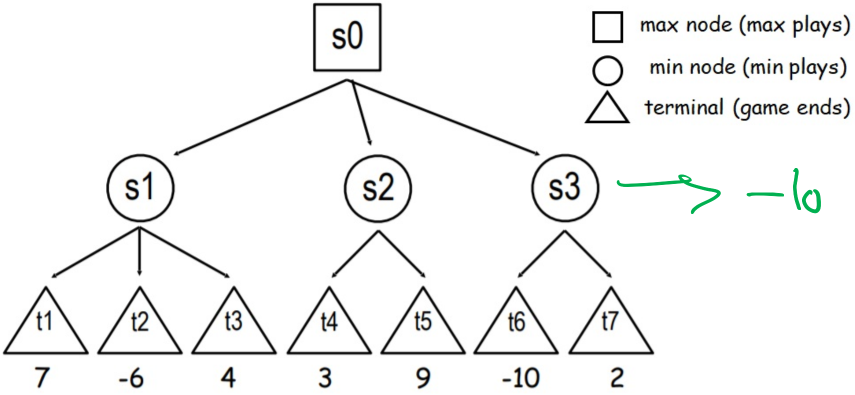


Minimax Strategy:

- Max always plays a move to change the state to the highest valued child.
- Min always plays a move to change the state to the lowest valued child.

If **Min plays poorly** (does not always move to lowest value child), Max could do better, but **never worse**.

Minimax Strategy



If Max goes to s_1, MIN goes to t_2

$\min\{U(t_1), U(t_2), U(t_3)\} = -6 \rightarrow t_2$

If max goes to s_2, min goes to t_4

$\min\{U(t_4), U(t_5)\} = \underline{3} \rightarrow t_4$

If max goes to s_3, min goes to t_6

$\min\{U(t_6), U(t_7)\} = \underline{-6} \rightarrow t_6$

max: $\max\{U(s_1), U(s_2), U(s_3)\}$

Minimax Strategy

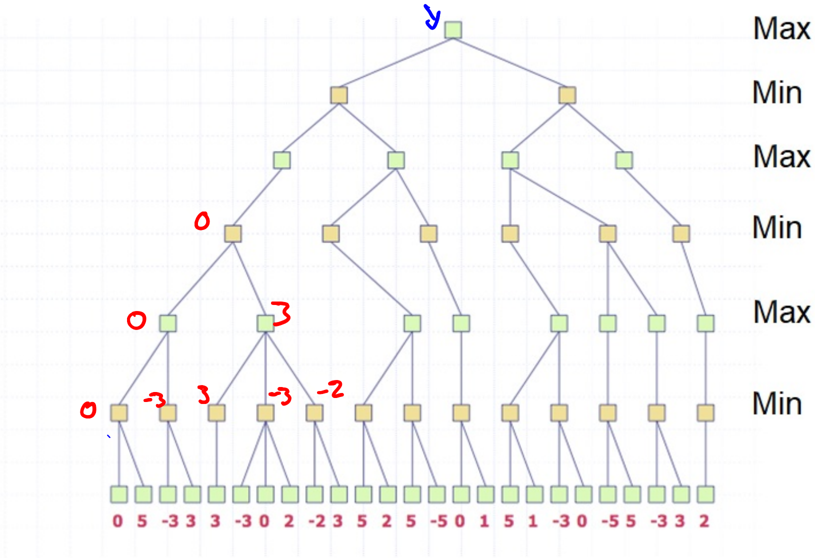
- We can compute a **utility** (aka **MinMAx value**) for the **non-terminal** states by assuming both players always play their best move.

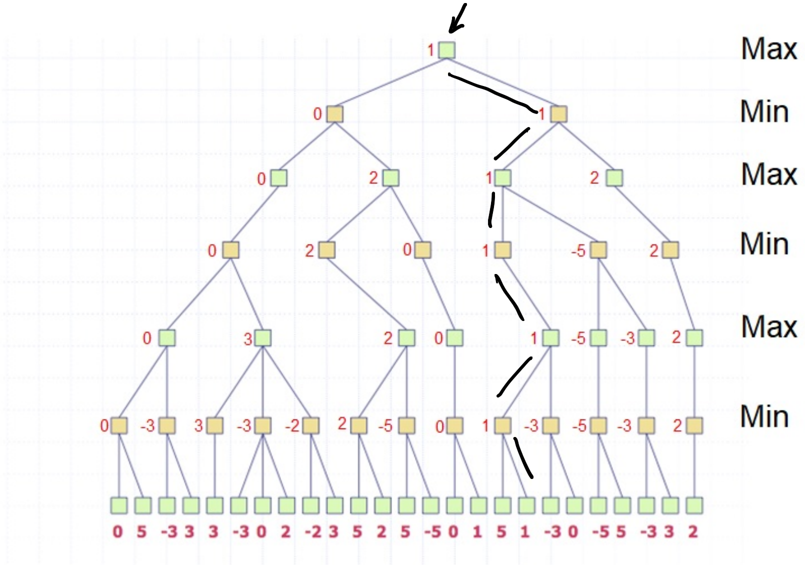
- Back the **utility values** up the **tree**:

$$U(s) = \begin{cases} U(s) & \text{if } s \text{ is a terminal (} U \text{ is defined} \\ & \text{(} U \text{ is defined for all terminals} \\ & \text{as part of input)} \\ min\{U(c) : c \text{ is a child of } s\} & \text{if } s \text{ is a Min node.} \\ max\{U(c) : c \text{ is a child of } s\} & \text{if } s \text{ is a Max node.} \end{cases}$$

- The values $U(s)$ labeling each state s are the values that Max will achieve in that state if both **Max and Min** play their best move.

Example


Max
Min
Max
Min
Max
Min
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2
y
0
0
3
0
-3
3
-3
-2



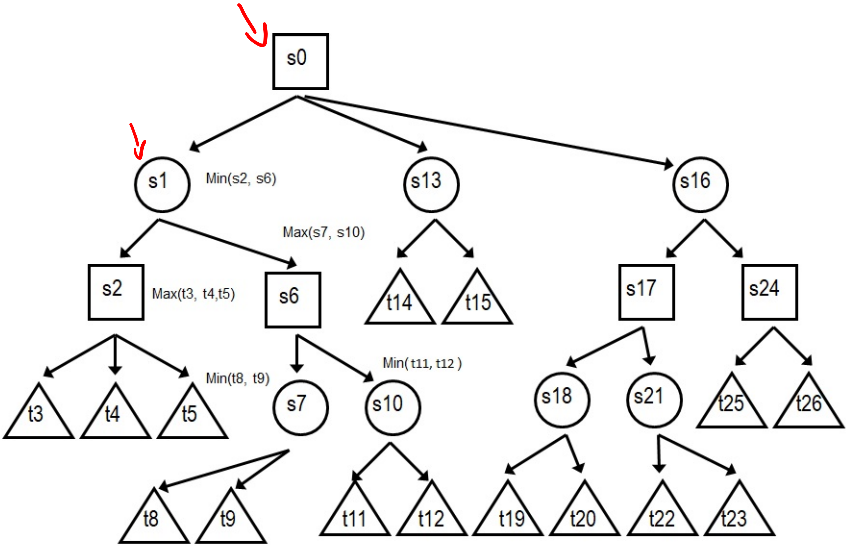
Max
Min
Max
Min
Max
Min
1
0
1
0
2
1
2
0
2
0
1
-5
2
0
3
2
0
1
-5
-3
2
0
-3
3
-3
-2
2
-5
0
1
-3
-5
-3
2
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

Depth-First Implementation of Minimax

- Building the **entire game tree** and **backing up values** gives each player their **strategy**.
- However, the game tree is **exponential** in size and might be too **large to store** in memory.
- We can save space by computing the minimax values with a **depth-first** implementation of minimax.
  Although **run-time** complexity is still **exponential**.
- We run the depth-first search **after each move** to compute what is the next move for the MAX player.
- This avoids explicitly representing the exponentially sized game tree: we just compute each move as it is needed.

Depth-First Implementation of Minimax

```
def DFMiniMax(s, Player):
//Return Utility of state s given that Player is MIN or MAX
1.  If s is TERMINAL
2.      Return U(s)    # Return terminal states utility,
                       # specified as part of game
//Apply Player's moves to get successor states.
3.  ChildList = s.Successors(Player)
4.  If Player == MIN
5.      return minimum of DFMiniMax(c, MAX) over c ∈ ChildList
6.  Else # Player is MAX
7.      return maximum of DFMiniMax(c, MIN) over c ∈ ChildList
```


s0
s1
Min(s2, s6)
s13
s16
Max(s7, s10)
s2
Max(t3, t4,t5)
s6
t14
t15
s17
s24
t3
t4
t5
Min(t8, t9)
s7
s10
Min(t11, t12)
s18
s21
t25
t26
t8
t9
t11
t12
t19
t20
t22
t23

Depth-First Implementation of Minimax

- The game tree has to have **finite depth** for DF Implementation to work.
- We must traverse the **entire search tree** to evaluate all options.
- **Time Complexity:** $\mathcal{O}(b^d)$ (both a BEST and WORSE case scenario), where b is the number of **legal moves at each state**, and d **maximum depth** of the tree.
- **Space Complexity:** $\mathcal{O}(bd)$.