# Lecture 5: Markov Decision Processes I

Fall Semester ‘2022

# Project #1 Hexapawn Game

- Due Date: 10/14, PM 11:59:59 (-10 pts/day after deadline, 3 days of delayed submission at the max.)
  - Source code (in any C/C++ or Python)
  - Project report (~ 5 pages incl. project description, overall architecture, main data structures,  input/output, major functions, etc.)

# Project #1 Hexapawn Game

- "Hexapawn" is a 3X3 board game, with two players moving alternately, starting with three pawns each of opposite color (White and Black). Each pawn can move one step horizontally or vertically (up, down, left or right) to an unoccupied square. A pawn can also "kill" an opponent's pawn which is diagonally adjacent. For example, from the position

| | | |
|---|---|---|
| W | | W |
| B | W | |
| | B | B |

W can move next to any of the following positions:

| | | |
|---|---|---|
| | W | W |
| B | W | |
| | B | B |

| | | |
|---|---|---|
| W | W | W |
| B | | |
| | B | B |

| | | |
|---|---|---|
| W | | W |
| B | | W |
| | B | B |

| | | |
|---|---|---|
| W | | W |
| B | | |
| | B | W |

| | | |
|---|---|---|
| W | W | |
| B | W | |
| | B | B |

| | | |
|---|---|---|
| W | | |
| B | W | W |
| | B | B |

# Rules of hexapawn

- The game is played on a 3x3 board.
- Each player begins with three pawns lined up on opposite sides of the board. There are three white pawns and three black pawns, which gives us a grand total of six pawns, hence the name hexapawn.
- White always moves first, just like in chess. The players take turns moving their pawns.
- A pawn can move one square forward to an empty square, or it can move one square diagonally ahead (either to the left or right) to a square occupied by an opponent's pawn, in which case the opponent's pawn is removed from the board.
- One player wins when one of these three conditions is true:
  - One of that player's pawns has reached the opposite end of the board
  - The opponent's pawns have all been removed from the board
  - It's the opponent's turn to move but the opponent can't move any pawns.

# Implementation

- Implement a program that will play the hexapawn game using a 3-ply game tree search with alpha-beta pruning. The program starts displaying the initial board state read from an input file which will contain a number of different board states. For each initial board state, the user decides his color and who is to start first. The game continues by accepting the user's next move or computing the computer's next move, alternately, followed by displaying the next board state. This process is repeated until the game arrives at one of the terminal conditions (Win/Loss/Draw)
- Use the following evaluation function for this games:

$$\text{Eval}(s) = \begin{cases} 100 & \text{if the board is won by W} \\ -100 & \text{if the board is won by B} \\ \sum_{i \in O} r_i^2 - \sum_{j \in X} (4 - r_j)^2 & \text{otherwise} \end{cases}$$

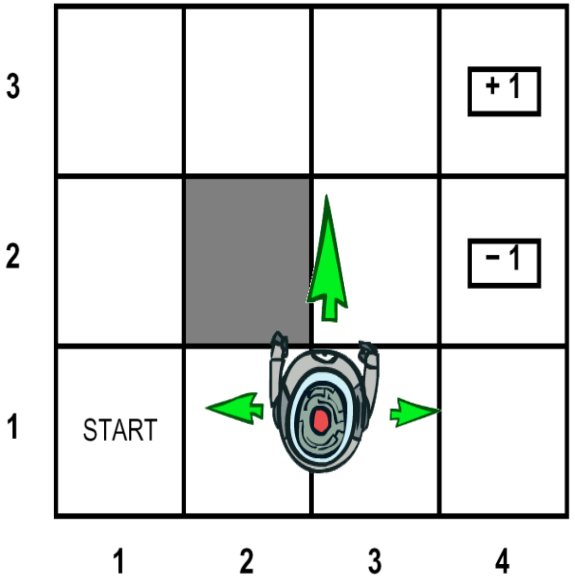
where $r_i$ denotes the row number of the $i$th piece.

# Project Report

1. Introduction
   - Overall description
   - Requirements
   - Sample input/output
2. Overall system design
   - Functional modules and their relationships to each other in block-diagrams
3. Data structures
   - Diagrams of the main and auxiliary data structures used
4. Algorithm specification
   - Pseudocodes for the following functions:
     A. Main function
     B. Each command (INSERT, DELETE, UPDATE, PRINT) processing
     C. Input file scan
   - Performance Evaluation
5. Discussion and Conclusion

# Non-deterministic Search
## Example: Grid World

- A maze-like problem
  - The agent lives in a grid
  - Walls block the agent's path
- Noisy movement: actions do not always go as planned
  - 80% of the time, the action North takes the agent North (if there is no wall there)
  - 10% of the time, North takes the agent West; 10% East
  - If there is a wall in the direction the agent would have been taken, the agent stays put
- The agent receives rewards
  - Small "living" reward each step (can be negative)
  - Big rewards come at the end (good or bad)
- Goal: maximize sum of rewards



# Markov Decision Processes

- An MDP is defined by:
  - A set of states $s \in S$
  - A set of actions $a \in A$
  - A transition function T(s, a, s')
    - Probability that a from s leads to s', i.e., P(s'| s, a)
    - Also called the model or the dynamics
  - A reward function R(s, a, s')
    - Sometimes just R(s) or R(s')
  - A start state
  - Maybe a terminal state

# What is Markov about MDPs?

- "Markov" generally means that given the present state, the future and the past are independent
- For Markov decision processes, "Markov" means action outcomes depend only on the current state

$$P(S_{t+1} = s' | S_t = s_t, A_t = a_t, S_{t-1} = s_{t-1}, A_{t-1}, \ldots S_0 = s_0)$$
$$= \quad P(S_{t+1} = s' | S_t = s_t, A_t = a_t)$$

- This is just like search, where the successor function could only depend on the current state (not the history)

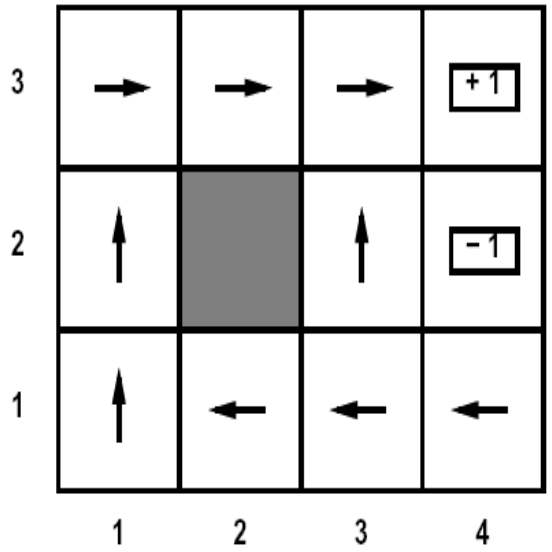
Andrey Markov (1856-1922)

# Policies

- In deterministic single-agent search problems, we wanted an optimal plan, or sequence of actions, from start to a goal
- For MDPs, we want an optimal

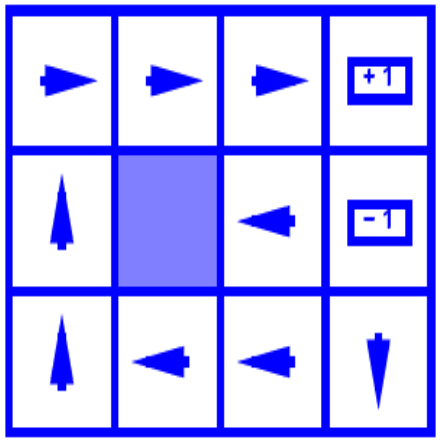
  policy $\pi^*: S \rightarrow A$

  - A policy $\pi$ gives an action for each state
  - An optimal policy is one that maximizes expected utility if followed
  - An explicit policy defines a reflex agent

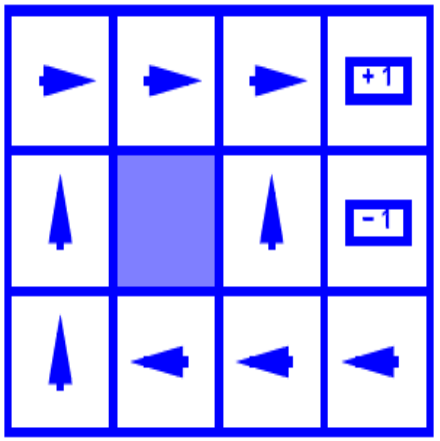



Optimal policy when R(s, a, s′) = -0.03 for all non-terminals s

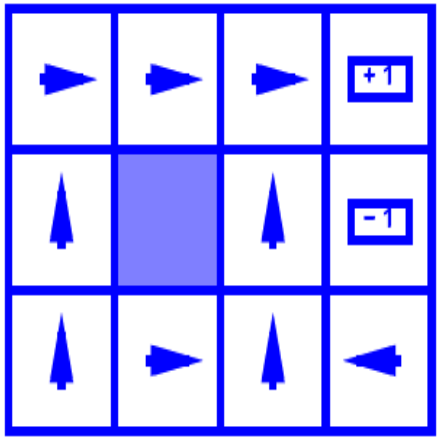
# Optimal Policies



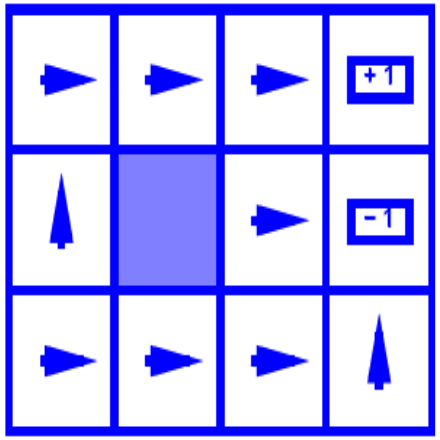

R(s) = -0.01




R(s) = -0.03




R(s) = -0.4




R(s) = -2.0

# Utilities of Sequences

- What preferences should an agent have over reward sequences?
- More or less? [1, 2, 2] or [2, 3, 4]
- Now or later? [0, 0, 1] or [1, 0, 0]

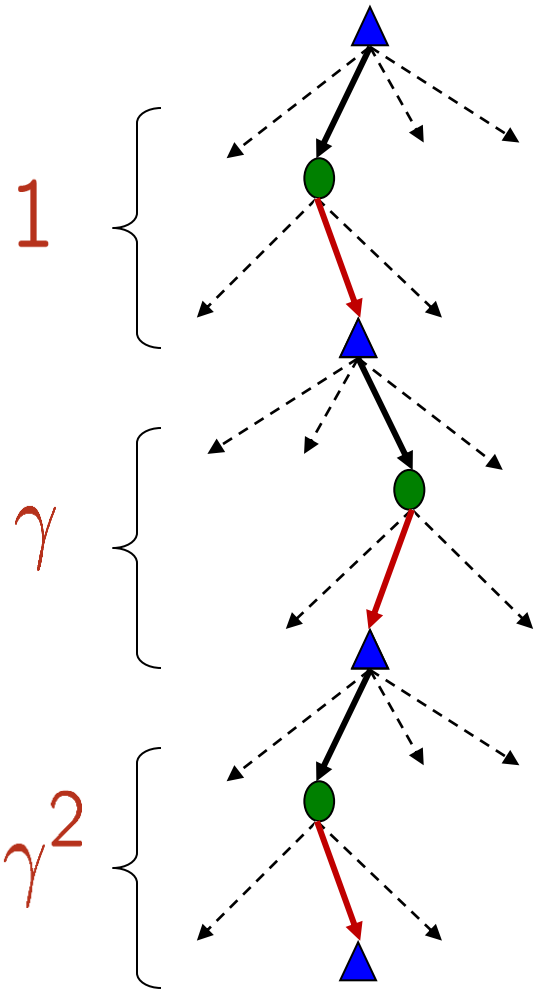
# Discounting

- It's reasonable to maximize the sum of rewards
- It's also reasonable to prefer rewards now to rewards later
- One solution: values of rewards decay exponentially

| $1$ | $\gamma$ | $\gamma^2$ |
| --- | --- | --- |
| Worth Now | Worth Next Step | Worth In Two Steps |

# Discounting

- How to discount?
  - Each time we descend a level, we multiply in the discount once
- Why discount?
  - Reward now is better than later
  - Can also think of it as a 1-gamma chance of ending the process at every step
  - Also helps our algorithms converge
- Example: discount of 0.5
  - U([1,2,3]) = 1*1 + 0.5*2 + 0.25*3
  - U([1,2,3]) < U([3,2,1])



# Infinite Utilities

- Problem: What if the game lasts forever? Do we get infinite rewards?

- Solutions:
  - Finite horizon: (similar to depth-limited search)
    - Terminate episodes after a fixed T steps (e.g. life)
    - Gives nonstationary policies ($\pi$ depends on time left)
  - Discounting: use $0 < \gamma < 1$

$$U([r_0, \ldots r_\infty]) = \sum_{t=0}^{\infty} \gamma^t r_t \leq R_{\max}/(1-\gamma)$$

    - Smaller $\gamma$ means smaller "horizon" – shorter term focus
  - Absorbing state: guarantee that for every policy, a terminal state will eventually be reached (like "overheated" for racing)