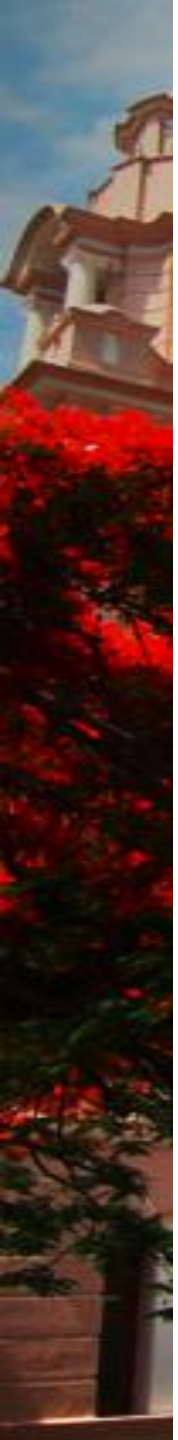
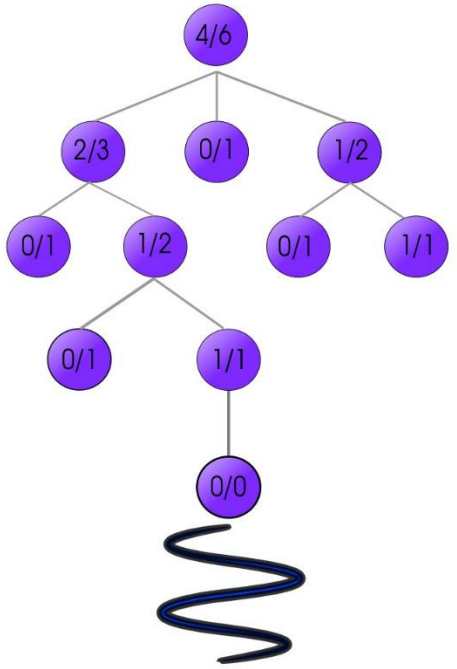
MSBA7003 Quantitative Analysis Methods

04 Monte Carlo Tree Search



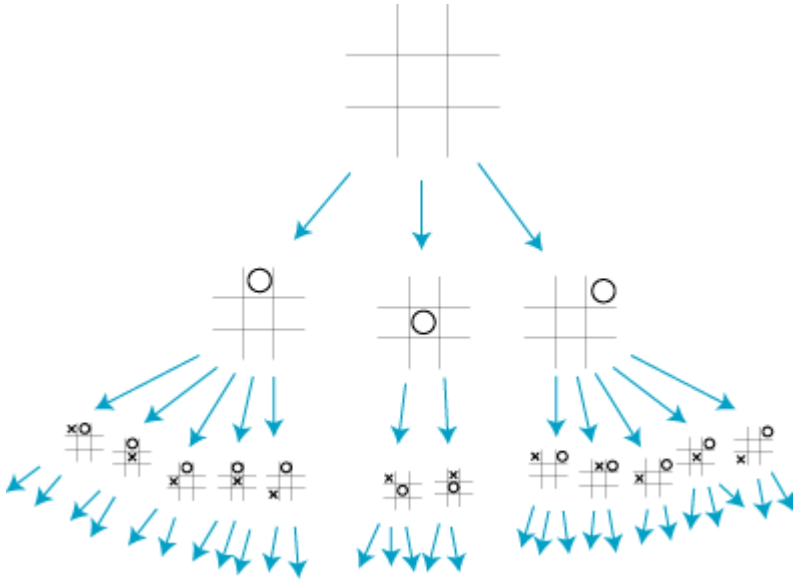
Agenda

- AlphaGo and Game AI
- Monte Carlo Tree Search
  - Concepts
  - Search Strategy
  - Simulation Strategy
  - Implementation
- Applications



AlphaGo and Game AI

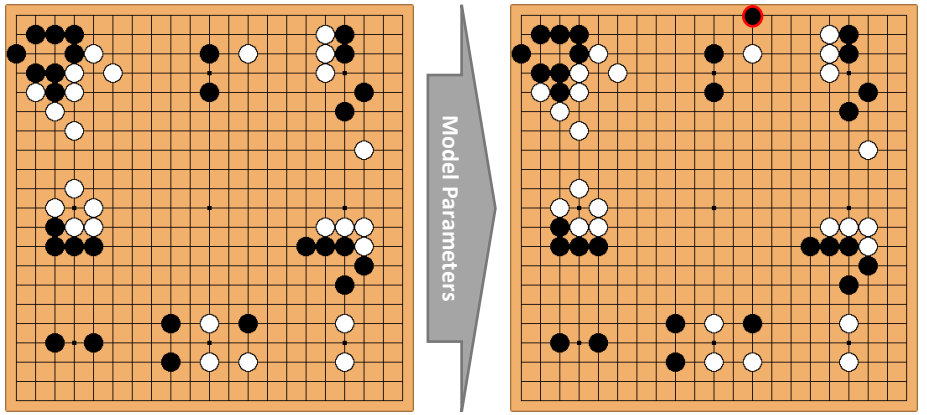
- Normally, a game can be solved by backward induction.
- Backward induction fails if the game tree is too large or the random events follow complicated or unknown probability distributions.



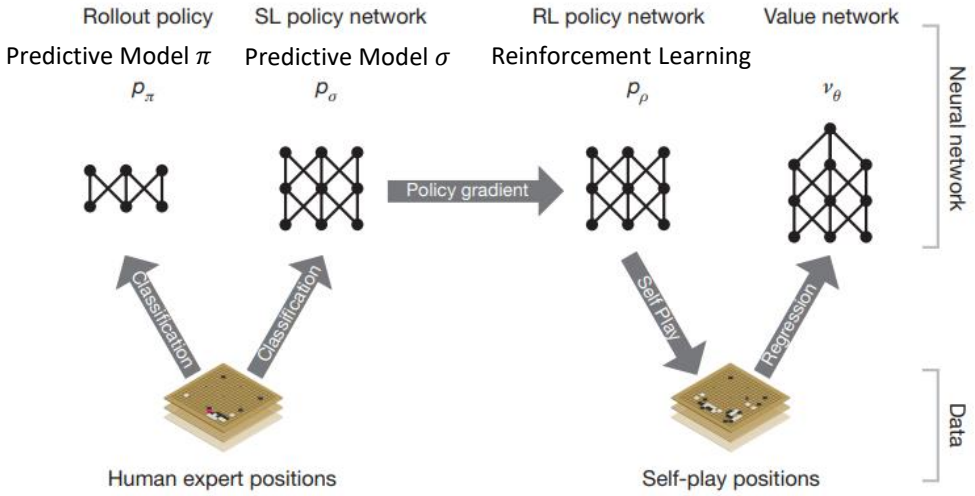
AlphaGo and Game AI

- AlphaGo won Lee Sedol and Ke Jie, the top two go players.
- Monte Carlo Tree Search (MCTS) is at the heart of AlphaGo's algorithm.

AlphaGo and Game AI

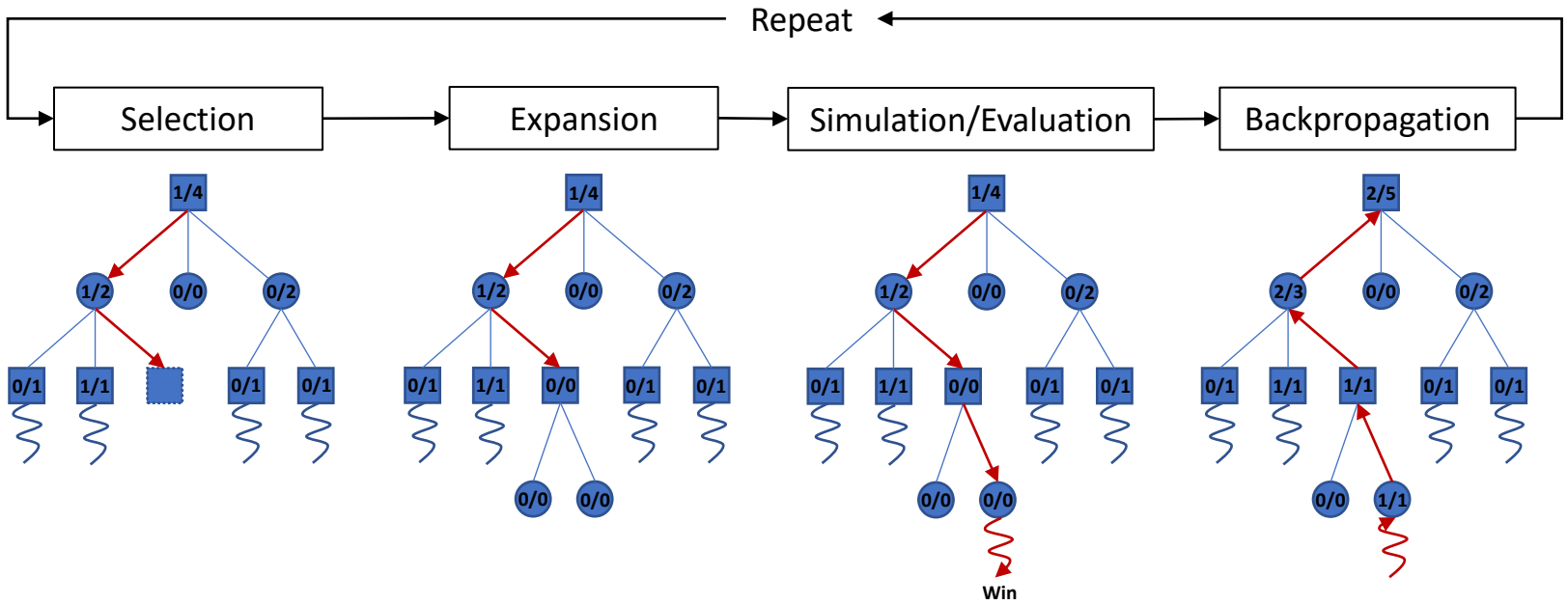


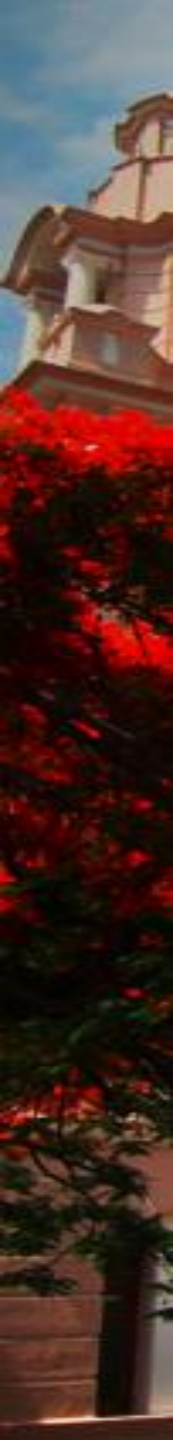
- AlphaGo's algorithm
  - Step 1: historical data => predictive models π and σ
  - Step 2: predictive model σ => preliminary reinforcement learning (RL) model ρ
  - Step 3: preliminary RL model ρ => self play data => value network
  - Step 4: (model π + model σ + value network) => Monte Carlo Tree Search



Monte Carlo Tree Search

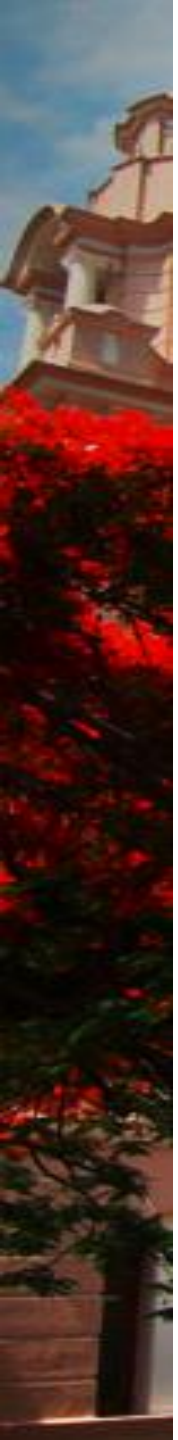
- The basic MCTS algorithm is to build a search tree, node by node, according to the outcomes of simulated playouts.
- The process can be broken down into four steps.





Monte Carlo Tree Search

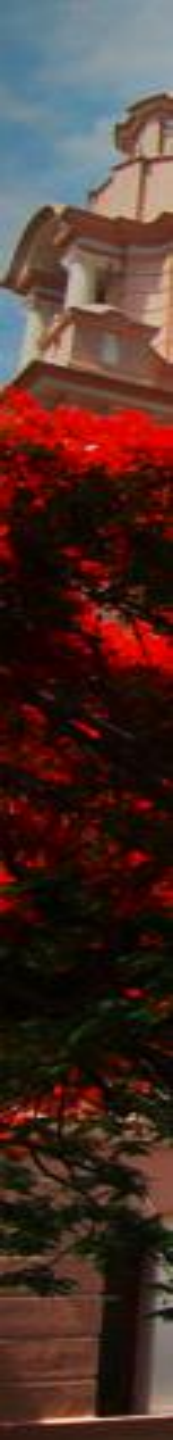
- Initialization: Create the root node (i.e., the level 1 decision node) and the child nodes (i.e., the state-of-nature nodes) following each option.
- Selection: Search forward by selecting a state-of-nature node (or an option) according to the recorded data and a given policy A. After that, select or create a child node (or a state of nature) through simulation. Repeat until no more child nodes are available as options.
- Expansion: If the newly created node is not a terminal node, create all (or part) of the state-of-nature nodes (i.e., all the options) as the child nodes for the current node.
- Simulation/Evaluation: Repeat the previous two steps (with a different selection policy B & expansion) until a terminal node is reached and the final payoff is received. Or, directly evaluate the payoff of the current node.
- Backpropagation: Update the summary statistics of each node along the path according to the result.



Selection Strategy

- Random Search
  - Focuses on exploration (i.e., look in areas that have not been well sampled yet) but ignores exploitation (i.e., look in areas which appear to be promising)
  - The “optimal” option given by the tree in the end may not be truly optimal

- Upper Confidence Bound (UCB1) Strategy
  - Balances exploitation of the currently most promising option with exploration of alternatives which may later turn out to be better
  - Converges to optimal decisions given sufficient time



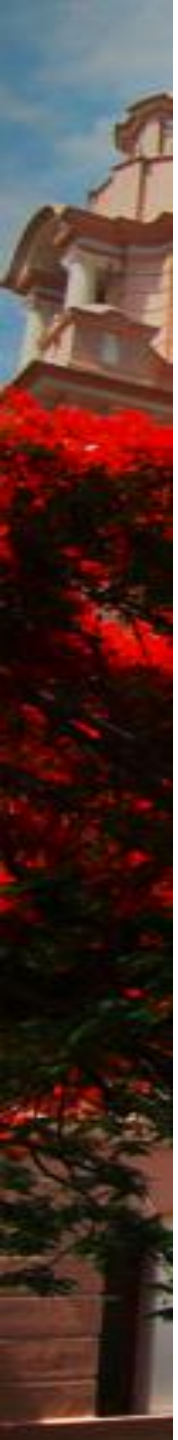
UCB1 Strategy

- The upper confidence bound for node i (an option) is given by

$$B_i = V_i + 2\sqrt{\frac{\ln N_i}{n_i}}$$

- V_i is the average total reward/value achieved by paths after node i (going forward)
- N_i is the number of times the parent node of i has been visited
- n_i is the number of times node i has been visited

- At a decision node, select the child node that maximizes the UCB.

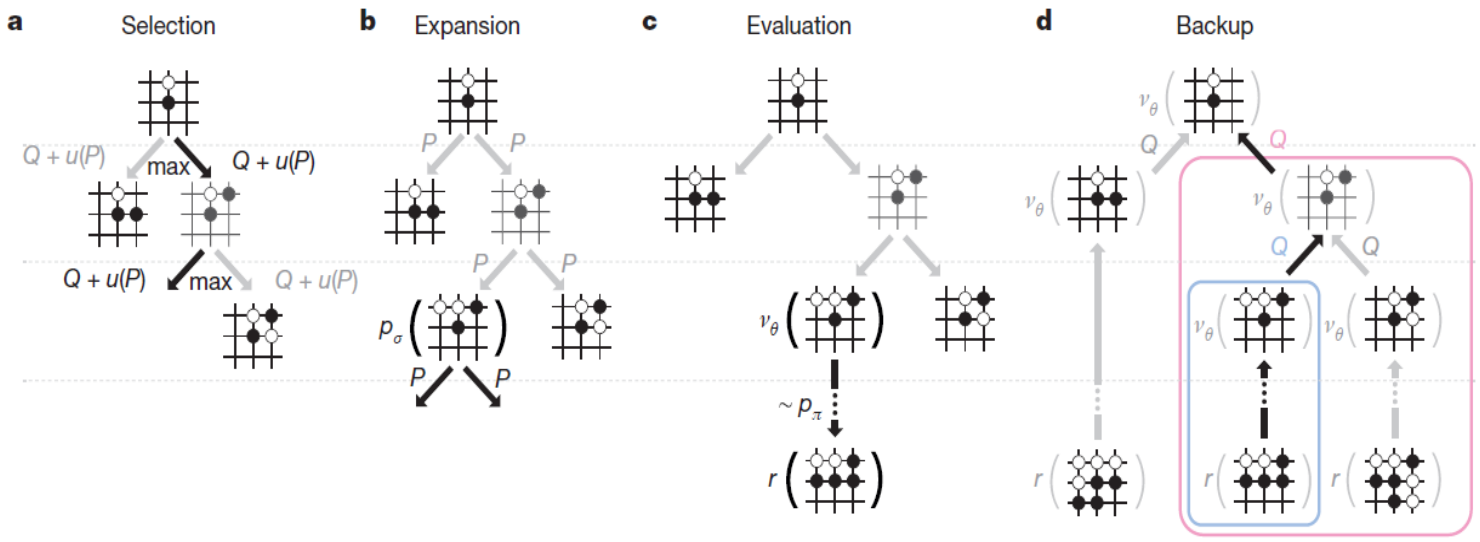


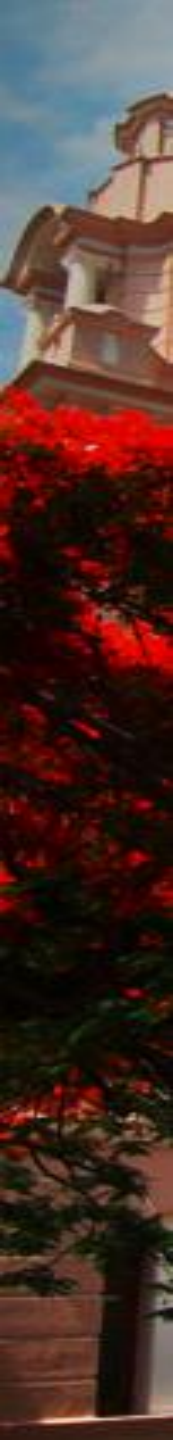
UCB1 Strategy

- If more than one child node has the same maximum UCB value, the tie is usually broken randomly.
- The first term drives exploitation and the second term drives exploration. If a child node is selected, the value of its second term will decrease but the value will increase for other child nodes.
- $n_i = 0$ yields a UCB value of infinity, so previously unvisited child nodes are assigned the largest value in order to ensure that all child nodes are considered at least once before any child node is further expanded.
- The formula can be modified to $B_i = V_i + 2C\sqrt{\frac{\ln N_i}{n_i}}$, where C can be adjusted to lower or increase the amount of exploration performed. If the reward is beyond the range of [0,1], the value of C can be carefully chosen to achieve the balance.

AlphaGo's Selection Strategy

- $Q_i = V_i$
- $u_i \propto \frac{P_i}{1+n_i}$, where P_i is given by model σ
- The opponent's next move is also given by model σ



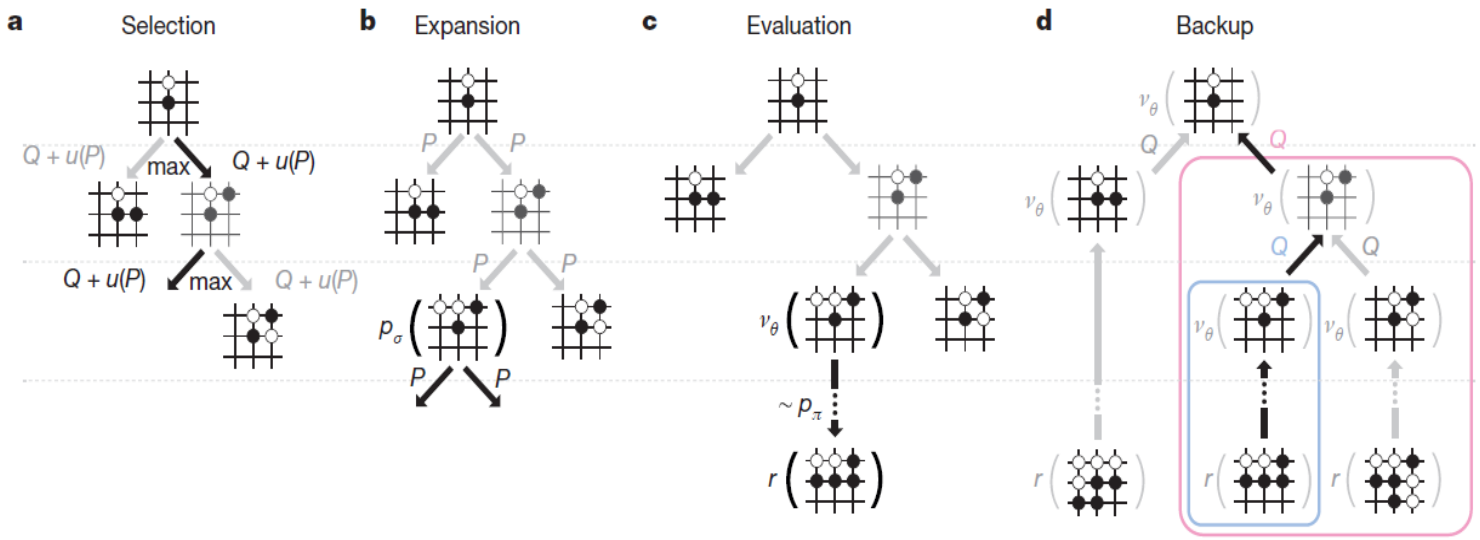


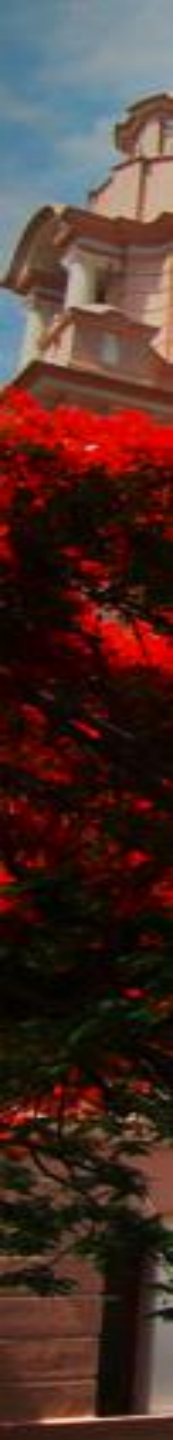
Simulation/Evaluation Strategy

- Simulation/evaluation is initiated when the search goes to a new decision node. The key is to simulate the decision maker's move in the unexplored part of the tree and quickly evaluate the performance.
- The quality of S/E strategy determines the quality of decisions and the speed of convergence.
- Possible strategies:
  - Random choice
  - Predictive/prescriptive model
- Note: simulation of the opponent's move or the realization of state is needed in both selection and S/E stages.

AlphaGo's Simulation/Evaluation Strategy

- Step 1: starting from the current decision node, finish the game using policy π
- Step 2: evaluate the chance of success for the simulated move using the value network
- Step 3: calculate the weighted average of the two results and give the value to Q of the simulated move

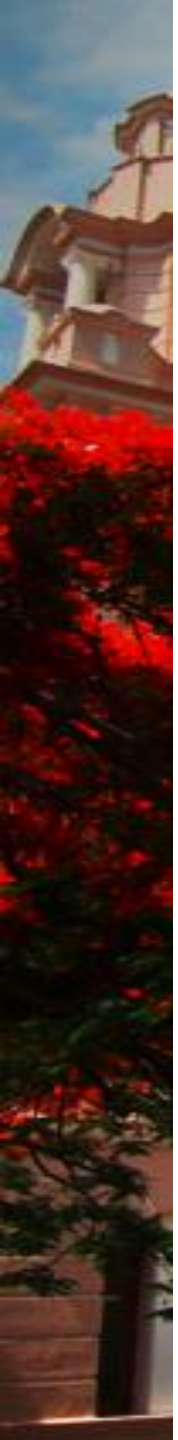




Implementation: Building Search Tree

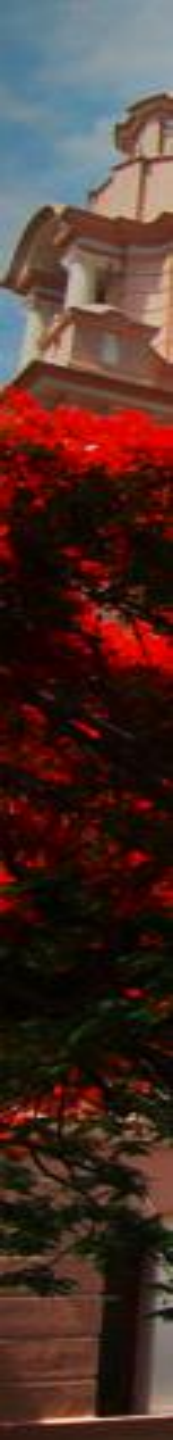
- The MCTS algorithm with UCB1 selection strategy can be implemented with a table.

Node Index	Parent Index	Child Index	Node Type	Meaning	n	V	UCB
1	0	{2,3,4}	Decision	The root node	2	1/2	-
2	1	{…}	State	Option 1-1	1	1/1	2.6651
3	1	{}	State	Option 1-2	0	0	∞
4	1	{…}	State	Option 1-3	1	0	1.6651
…							



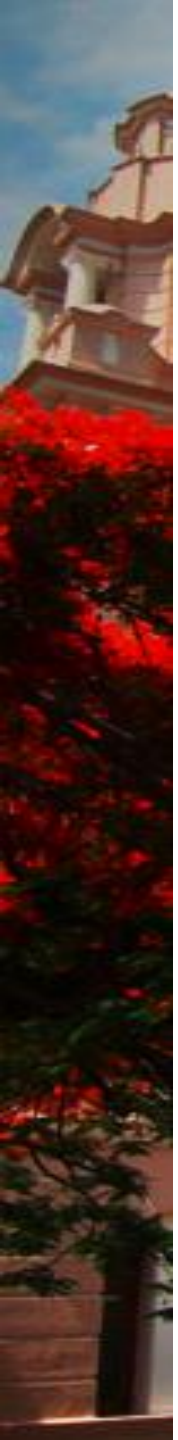
Implementation: Building Search Tree

Node Index	Parent Index	Child Index	Node Type	Meaning	n	V	UCB
1	0	{2,3,4}	Decision	The root node	2	1/2	-
2	1	{…}	State	Option 1(a)	1	1/1	2.6651
3	1	{K+1}	State	Option 1(b)	0	0	∞
4	1	{…}	State	Option 1(c)	1	0	1.6651
…							
K + 1	3	{}	Decision	State 3(a)	0	0	∞



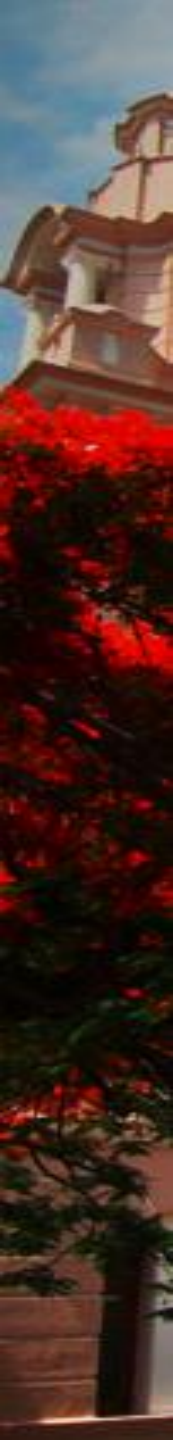
Implementation: Building Search Tree

Node Index	Parent Index	Child Index	Node Type	Meaning	n	V	UCB
1	0	{2,3,4}	Decision	The root node	2	1/2	-
2	1	{…}	State	Option 1(a)	1	1/1	2.6651
3	1	{K+1}	State	Option 1(b)	0	0	∞
4	1	{…}	State	Option 1(c)	1	0	1.6651
…							
K + 1	3	{K+2,K+3}	Decision	State 3(a)	0	0	∞
K + 2	K + 1	{}	State	Option (K+1)(a)	0	0	∞
K + 3	K + 1	{}	State	Option (K+1)(b)	0	0	∞



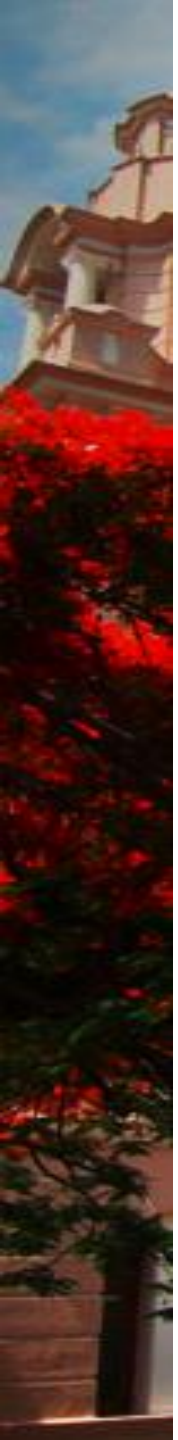
Implementation: Building Search Tree

Node Index	Parent Index	Child Index	Node Type	Meaning	n	V	UCB
1	0	{2,3,4}	Decision	The root node	3	2/3	-
2	1	{…}	State	Option 1(a)	1	1/1	3.0963
3	1	{K+1}	State	Option 1(b)	1	1/1	3.0963
4	1	{…}	State	Option 1(c)	1	0	?
…							
K + 1	3	{K+2,K+3}	Decision	State 3(a)	1	1	1
K + 2	K + 1	{…}	State	Option (K+1)(a)	1	1	1
K + 3	K + 1	{}	State	Option (K+1)(b)	0	0	∞
…							
K’	…	{}	Terminal	Outcome	1	1	1



Implementation: Optimization

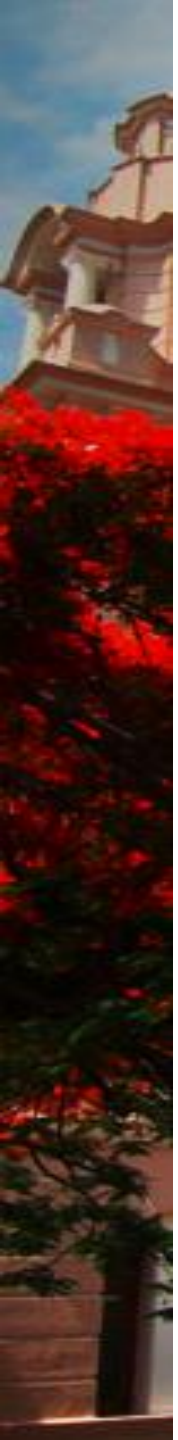
- When doing optimization:
  - Select the option that maximizes the total reward/value V.
  - The state of nature realizes according to the real situation.
  - If the state does not exist in the current tree, perform the MCTS algorithm in real time and then do the optimization.
- Note: If the initial state is uncertain, you need to train multiple trees.
  - E.g., you are playing white stones in the game of go.
- The quality of the search tree can be measured by the V values of the child nodes of the root.



The Connect-Four Game

- Instructions
  - Play on a 7-by-7 board. Select a column and drop a disc. The discs will pile up from the bottom. You and your opponent take turns to make the move.
  - To win: connect 4 of your discs so that they form a line in horizontal, vertical, or diagonal direction.

A	B	C	D	E	F	G



The Tank-War Game

```
import copy
import random
import math
board_depth=6; tanks=['LC','LB','LA','RA','RB','RC']; UCB_Constant=1.2

def initial_board_survival(): # Initialize the board and the survival state
    B={'Player_U':{},'Player_D':{}}
    S={'Player_U':{},'Player_D':{}}
    for tank in tanks:
        B['Player_U'][tank]=1
        B['Player_D'][tank]=6
        S['Player_U'][tank]=1
        S['Player_D'][tank]=1
    return {0:B,1:S}

def occupied_boxes(B,S): # Calculate the number of occupied boxes for player D
    N = 0
    for k in range(len(tanks)):
        N = N + min(6 - B['Player_U'][tanks[len(tanks)-1-k]]*S['Player_U'][tanks[len(tanks)-1-k]],
                    max(3, 7 - B['Player_D'][tanks[k]]))
    return N
```

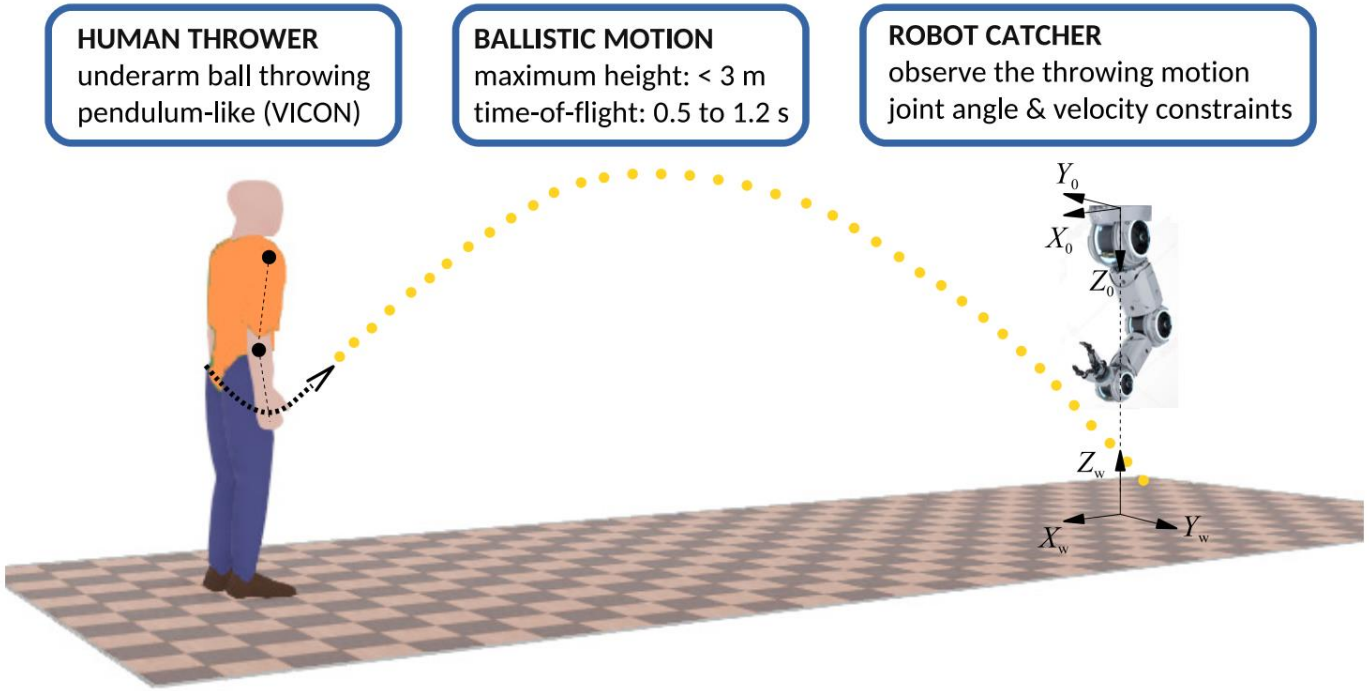
```
def fire_check(B,S): # Function to fire and to check if both players can still move forward
    temp_S = copy.deepcopy(S)
    # Check Player U's attack
    if S['Player_U']['LC'] == 1:
        if B['Player_D']['RC'] == B['Player_U']['LC'] + 2:
            temp_S['Player_D']['RC'] = 0
        if B['Player_D']['RA'] == B['Player_U']['LC']:
            temp_S['Player_D']['RA'] = 0
    if S['Player_U']['LB'] == 1:
        if B['Player_D']['RC'] == B['Player_U']['LB']:
            temp_S['Player_D']['RC'] = 0
        if B['Player_D']['RB'] == B['Player_U']['LB'] + 1:
            temp_S['Player_D']['RB'] = 0
        if B['Player_D']['RA'] == B['Player_U']['LB']:
            temp_S['Player_D']['RA'] = 0
```

```
def make_move(B,S,U,D): # Function to make a move (U and D are the tank indexes)
    B['Player_D'][D]=B['Player_D'][D]-1
    B['Player_U'][U]=B['Player_U'][U]+1
    if S['Player_U'][U] == 1 and S['Player_D'][D] == 1:
        success = 1
        # Check collision
        for k in range(len(tanks)):
            if (S['Player_D'][tanks[k]]*B['Player_D'][tanks[k]]+(1-S['Player_D'][tanks[k]])*7
                ) < S['Player_U'][tanks[len(tanks)-1-k]]*B['Player_U'][tanks[len(tanks)-1-k]]:
                #print('Wrong move! Road blocked! Error 1.')
                success = 0; break
            else:
                if (S['Player_D'][tanks[k]]*B['Player_D'][tanks[k]]+(1-S['Player_D'][tanks[k]])*7
                    ) == S['Player_U'][tanks[len(tanks)-1-k]]*B['Player_U'][tanks[len(tanks)-1-k]]:
                    if U == tanks[len(tanks)-1-k] and D == tanks[k]:
                        if B['Player_U'][tanks[len(tanks)-1-k]] > 3:
                            B['Player_U'][tanks[len(tanks)-1-k]] = B['Player_U'][tanks[len(tanks)-1-k]] - 1
                        else:
                            B['Player_D'][tanks[k]] = B['Player_D'][tanks[k]] + 1
                    else:
                        #print('Wrong move! Road blocked! Error 2.')
                        success = 0; break
    else:
        success=0
        #print('Wrong move! Tank no longer available!')
    if success == 0:
        B['Player_D'][D]=B['Player_D'][D]+1
        B['Player_U'][U]=B['Player_U'][U]-1
    return success
```

```
def oppo_move(B,S): # opponent is Player U; U's move given state {0:B,1:S}
    worst_payoff = {}
    for t in range(len(tanks)):
        worst_payoff[tanks[t]] = 6
        mobility = S['Player_U'][tanks[t]]*(B['Player_D'][tanks[len(tanks)-1-t]]*S['Player_D'][tanks[len(tanks)-1-t]]
                                            +(1-S['Player_D'][tanks[len(tanks)-1-t]])*7- B['Player_U'][tanks[t]] - 1)
        if mobility <= 0:
            worst_payoff[tanks[t]] = -6
        else:
            payoffs = [0]*len(tanks)
            for k in range(len(tanks)):
                o_mobility = S['Player_D'][tanks[k]]*(B['Player_D'][tanks[k]]
                                                      - B['Player_U'][tanks[len(tanks)-1-k]]*S['Player_U'][tanks[len(tanks)-1-k]] - 1)
```

Potential Applications

- Robotics / Human-robot collaboration



Potential Applications

- Carpark revenue management