Adversarial Search (Games)

KUO-SHIH TSENG (曾國師)
DEPARTMENT OF MATHEMATICS
NATIONAL CENTRAL UNIVERSITY, TAIWAN

2020/03/17

Course Announcement

- HW1 was released on 3/02. Download it from eeclass.
- 1 programming problem and 2 theory problems.
- Deadline: 3/24(Wed.) 00:00am
- Delivery: Please update your HW to eecalss using electrical format. Compress your HW into a zip file including PDF and code.
- **Late policy**: If your HW is late for 1 day, the discount rate is 0.8. For 2 days, the discount rate is 0.8^2. and so on.
- Start to work on it this week. You have 3 weeks to work.

Outline

- Games
- Optimal decisions
  - The minimax algorithm
  - Alpha-beta pruning
- Imperfect real-time decisions
- State of the art games
  - Function approximation
  - Monte-Carlo tree search
- EX: AlphaGo

Outline

[Problem solving]

Search problems

Adversarial Search

[Perception and Uncertainty]

Bayes Theorem

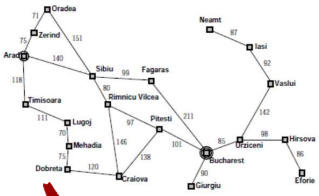
Bayes Filter and Smoothing

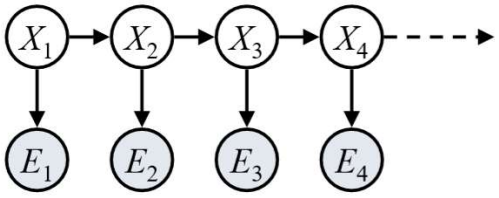
[Learning and Decision-making]

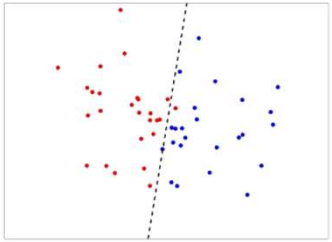
Supervised learning

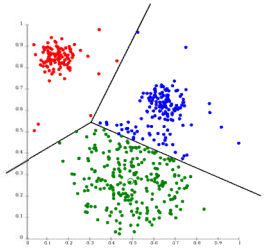
Unsupervised learning

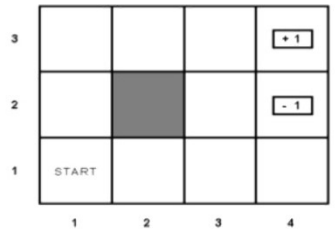
Reinforcement learning











Games

- Humans simplify real world wars into turned based games. We set some game rules and winning situations.

- Typical types of games:
  - Turn based games (e.g., chess)
  - Real-time strategy (e.g., starcraft)
  - Role playing games (RPG)
  - etc.

- This lecture will teach you how to utilize “search” to play a game!

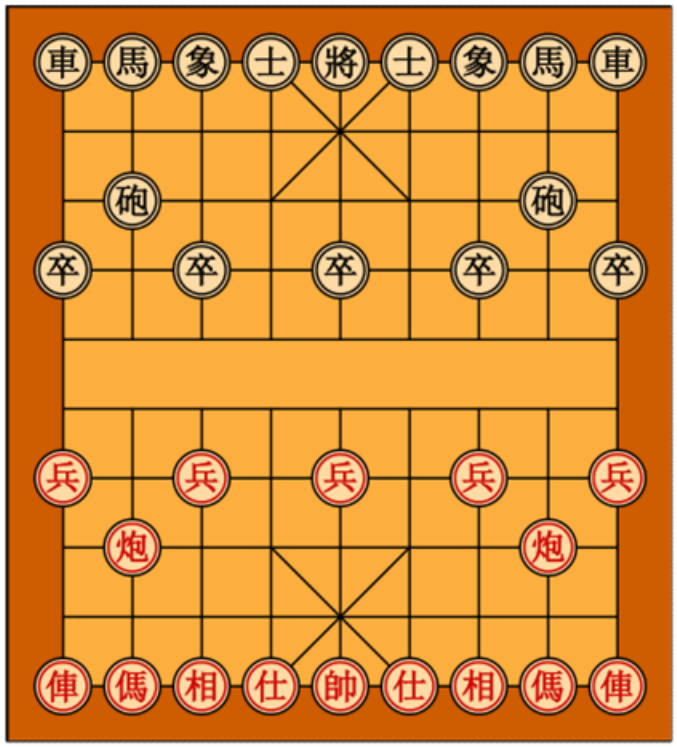
Games

- Tic-tac-toe
- Wining situation:
  - If the “O” or “X” in a straight line, the game is over.

https://en.wikipedia.org/wiki/Tic-tac-toe

Games

- Chinese Chess
- Wining situation:
  - If you took enemy's general, you win the game.



https://en.wikipedia.org/wiki/Xiangqi

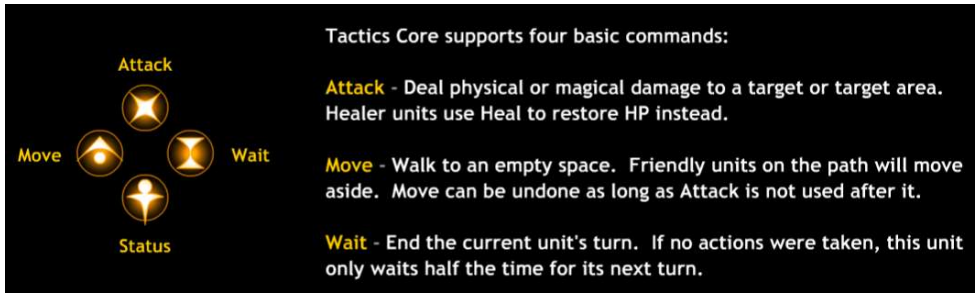
Games

- GO
- Wining situation:
  - The player having more territory wins

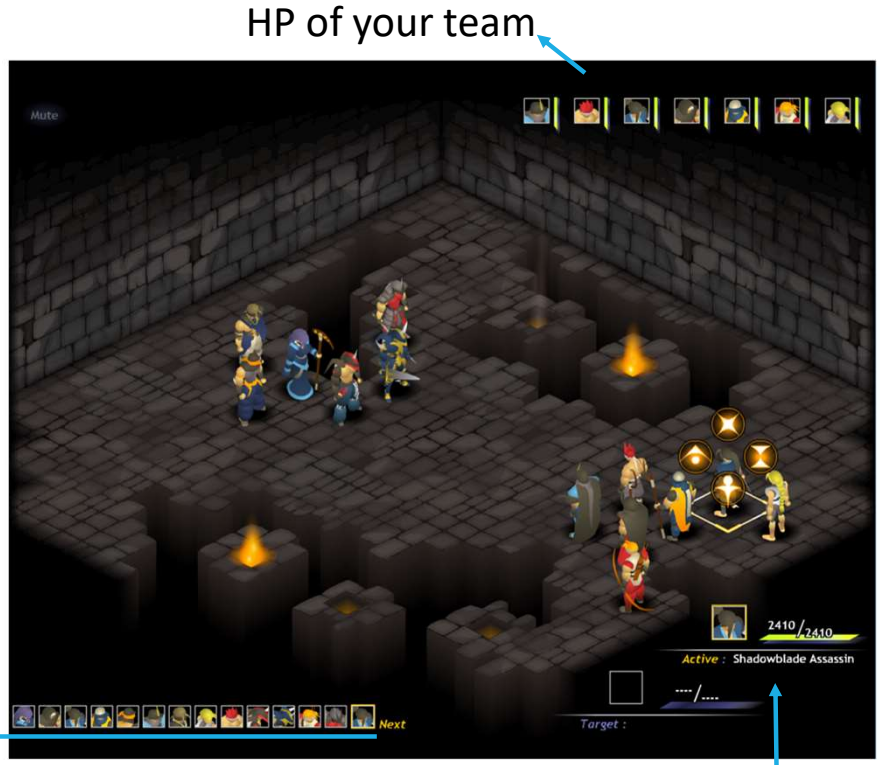
https://en.wikipedia.org/wiki/Go_(game)

Games

- Tactics Core
  - How to winning the battle for your team?
    - Shadowblade Assassin
    - Shadowfist Fighter
    - Ironclad Sentinel
    - Hunter
    - Mystical Healer
    - Templar Mage
    - Red Wizard



https://zh.y8.com/games/tactics_core
https://www.youtube.com/watch?v=3rOTj8l_tKA



Games

- Each soldier has different attack range, move range and skills.

Games

- We can apply “search” to games.
- Why is it difficult?
  - You don’t know the enemy’s actions
  - The search space could be huge (combinatorial explosion)
  - The outcome of the game is in the **bottom** of the tree
  - Greedy approaches could not be good decisions
- This lecture will cover
  - How to find optimal solutions? → minimax & alpha-beta pruning
  - How to find suboptimal solutions? → function approximation & MCTS

Games

- [Elements of a game]
- Initial state: The game is set up at the start.
- Player(s): The player has the move in a state.
- Actions(a): Returns the set of possible moves in a state.
- Result(s,a): The result of a move.
- Terminal-Test(s): Return 1 or 0 for the termination.
- Utility(s,p): The final numeric value for the termination state.
  - Win: +1, Loss: -1, draw: 0

Games

- Zero-sum game (constant sum): the total payoff to all players is the same for every instance of the game.
- EX:
  - 1st player wins: 1+0
  - 2nd player wins: 0+1
  - Draw : 0.5+0.5

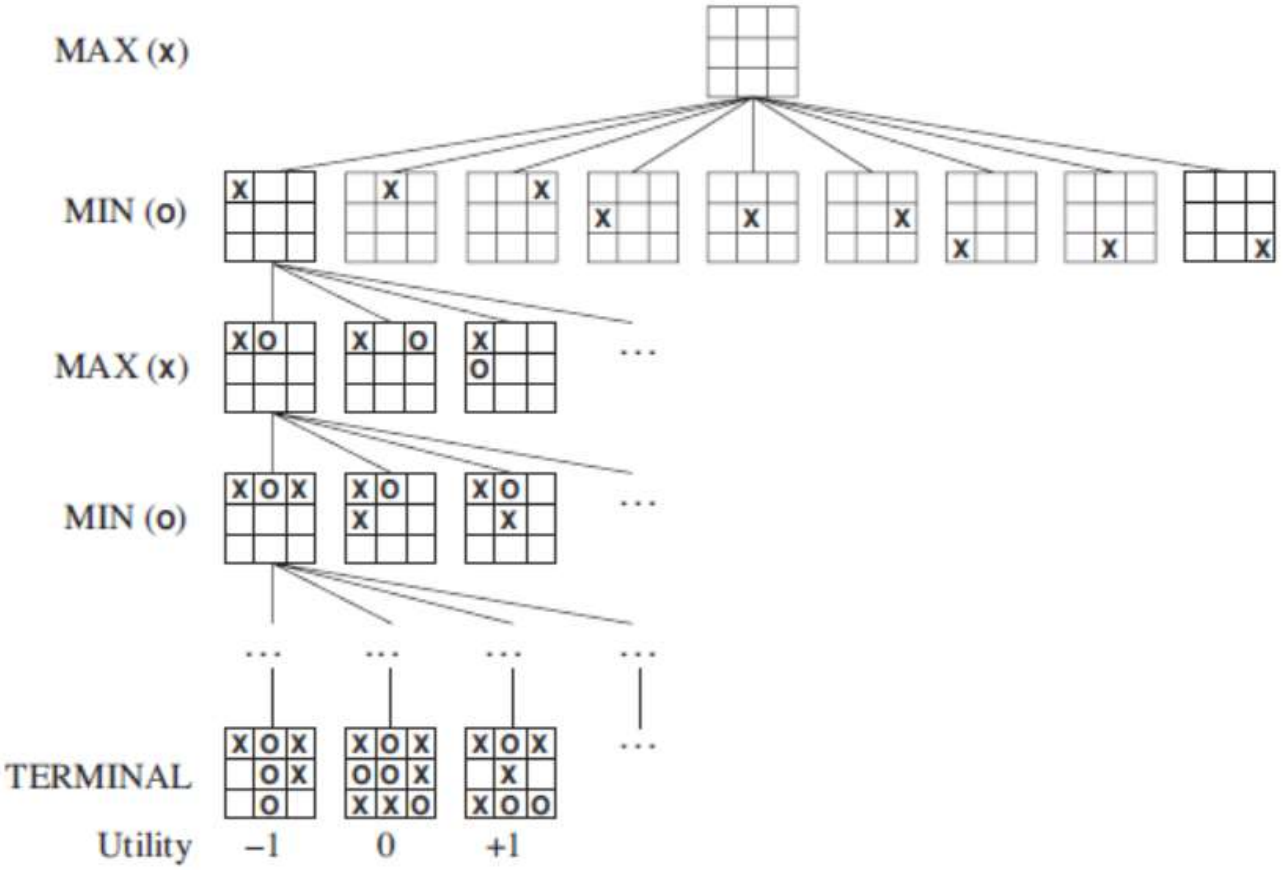
Optimal decisions

- If the search tree is not huge (limited depth), the AI can compute optimal decisions. → **minimax** algorithm
- To reduce the unnecessary computation, **Alpha-beta pruning** was proposed.
- If the search tree is huge (grows exponentially), the AI only can compute partial branches of the search tree. Then, it is an imperfect decision.
- To estimate the decision of a branch, **Monte-Carlo tree search** and **function approximation** were adopted.

Optimal decisions— Minimax

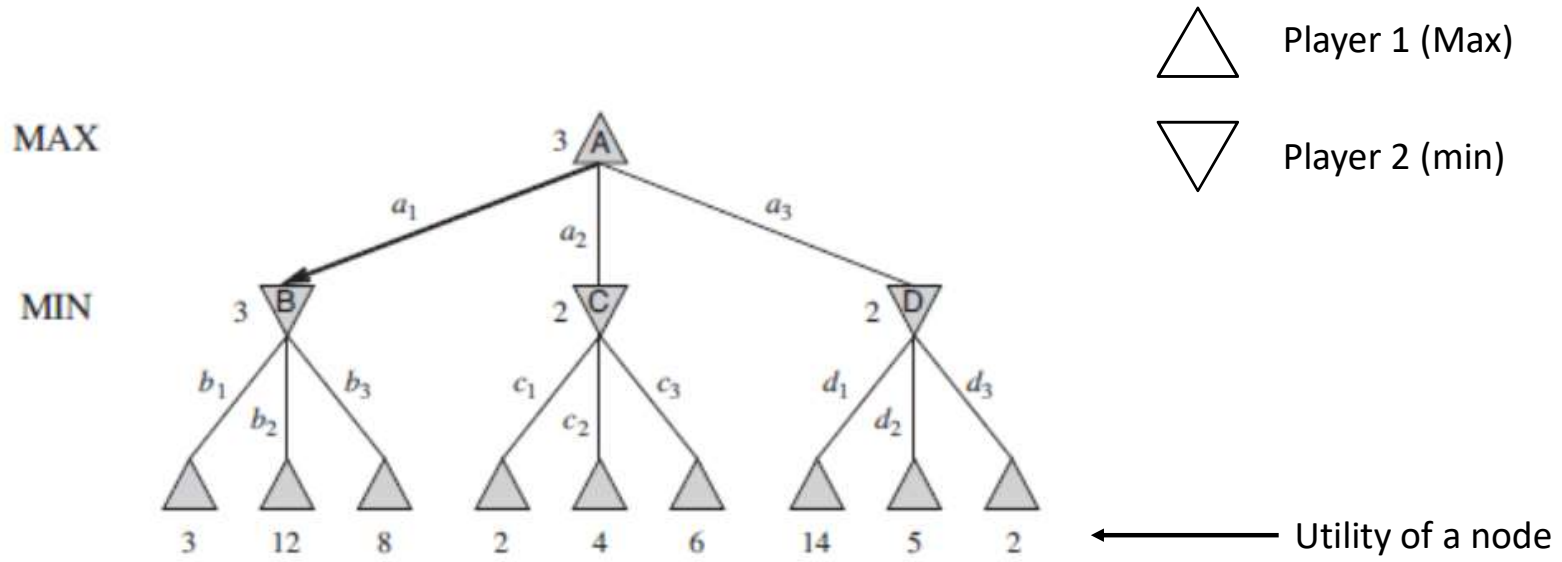
- Minimax algorithm assumes that the 1st player tries to maximize his/her winning probability while the 2nd player tries to minimize the 1st player's winning probability. The algorithm expands the search tree from the root to the leaf.
- Since the search tree grows exponentially, there are two ways to apply Minimax
  - Apply it for small scale games
  - Set a fixed depth and use an evaluation function.

Optimal decisions— Minimax



Optimal decisions— Minimax

- Assume you have this search tree. Minimax can find an optimal action at this turn assuming the enemy actions optimally.



Optimal decisions— Minimax

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```

minimax performes a DFS of game tree

$Time : O(b^m)$

$Space : O(bm)$

$b : branch$

$m : depth$

Optimal decisions— Minimax

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))
```

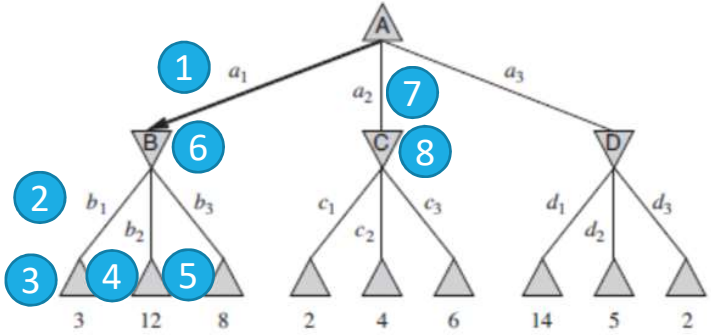
1 7 (stack)

```
function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞      (Initialization)
  for each a in ACTIONS(state) do (b1,b2,b3 or c1,c2,c3 or d1,d2,d3)
    v ← MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```

2 8 6

```
function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞   (Initialization)
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s, a)))
  return v
```

3 4 5

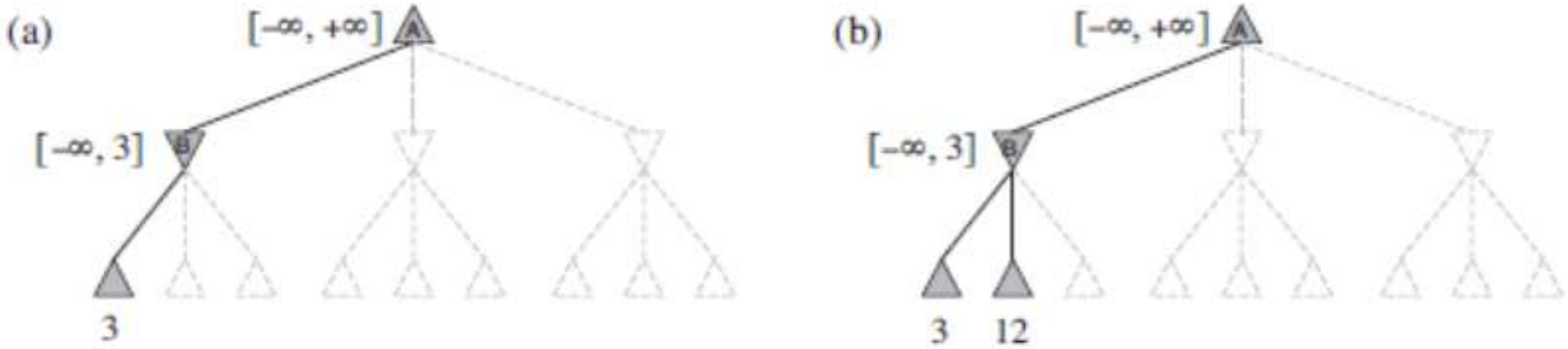


Optimal decisions— Alpha-beta pruning

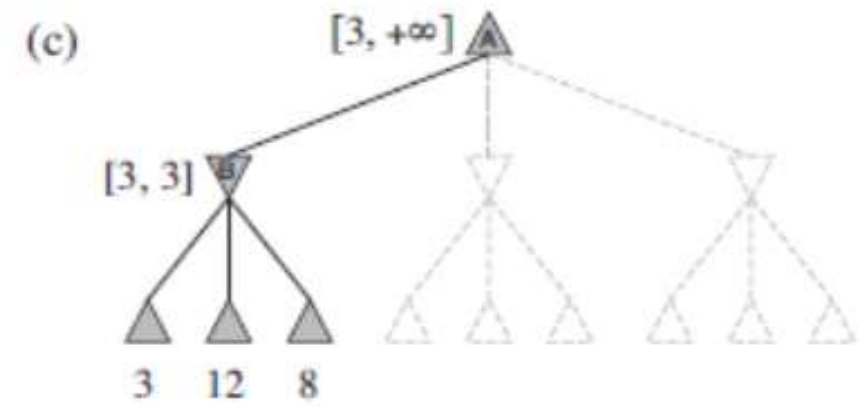
- It’s infeasible to expand the search tree even for a simple game. For example, Tic-tac-toe has 9!=362,880 Terminal nodes.
- Alpha-beta pruning was proposed to prune parts of minimax branches that will not affect the final decision.

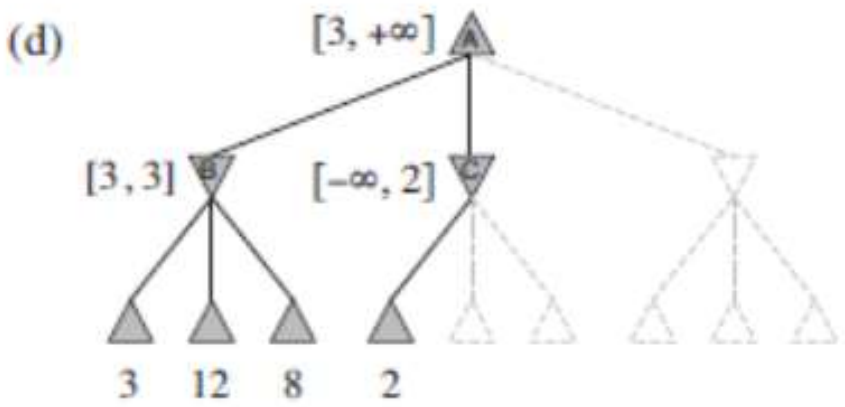
Optimal decisions— Alpha-beta pruning

[max_value, min_value]

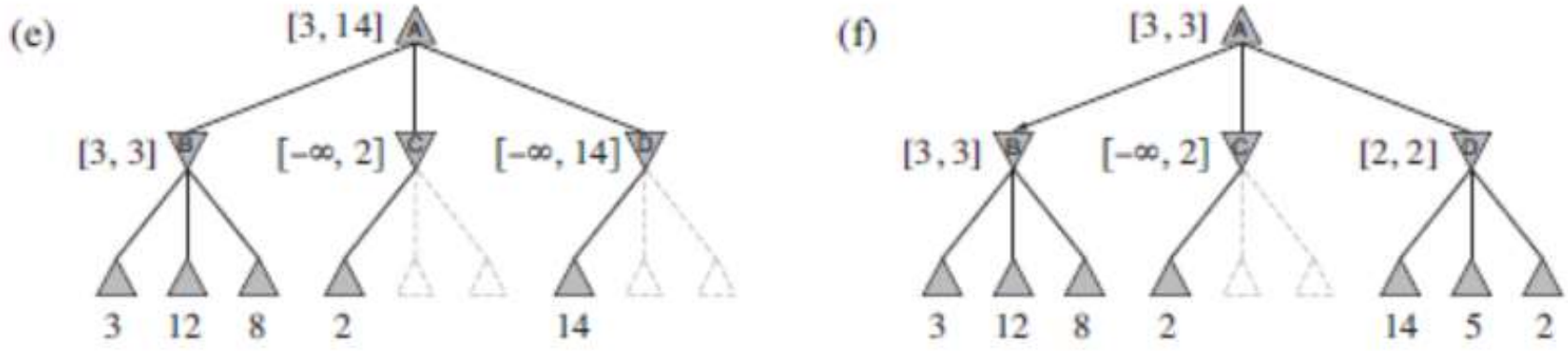


Optimal decisions— Alpha-beta pruning





Optimal decisions— Alpha-beta pruning



Alpha-beta pruning can reduce 2 nodes

Optimal decisions— Alpha-beta pruning

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v
```

```
function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v          (pruning these actions)
    α ← MAX(α, v)                   (update alpha for pruning)
  return v
```

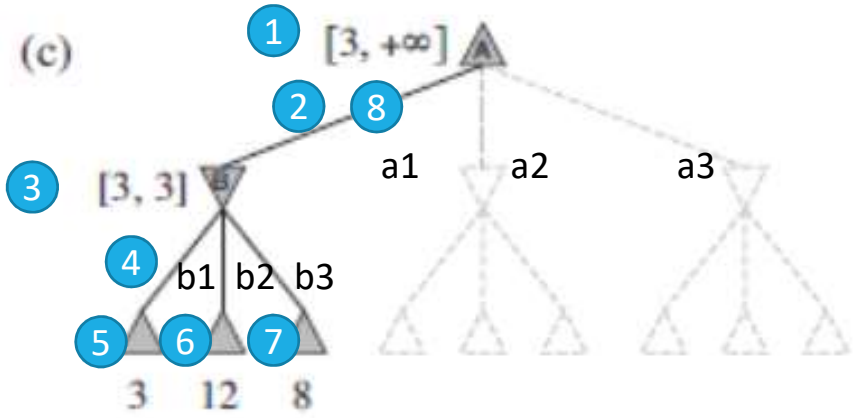
```
function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v          (pruning these actions)
    β ← MIN(β, v)                   (update beta for pruning)
  return v
```

Optimal decisions— Alpha-beta pruning

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a), α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

(A) (A) (a1 a2 a3) (α=3) (B) (b1 b2 b3) (β=3)

1 5 6 7 2 3 8 4



Optimal decisions— Alpha-beta pruning

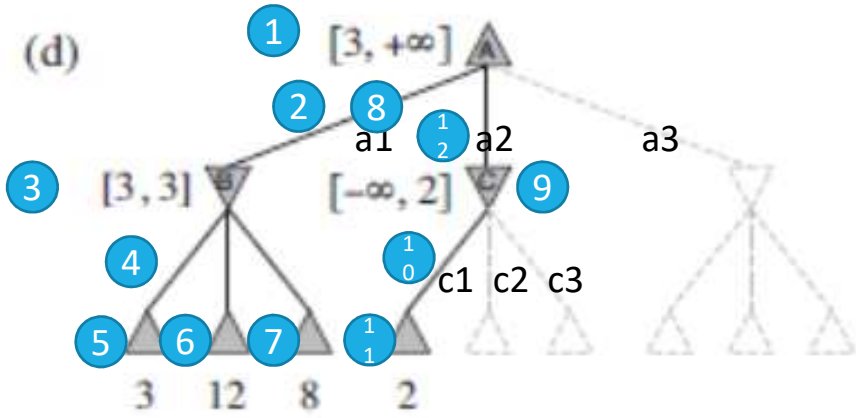
```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a), α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

(A) (a2 a3)

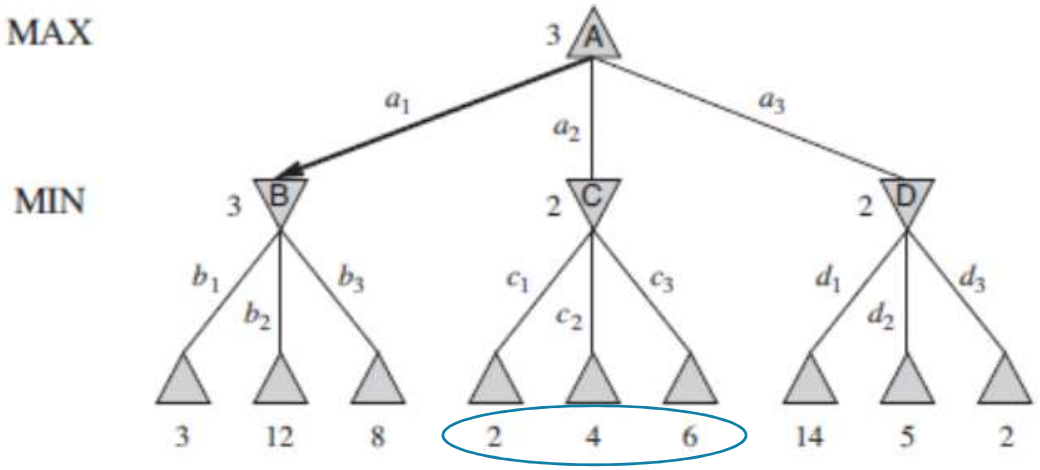
(3,∞)

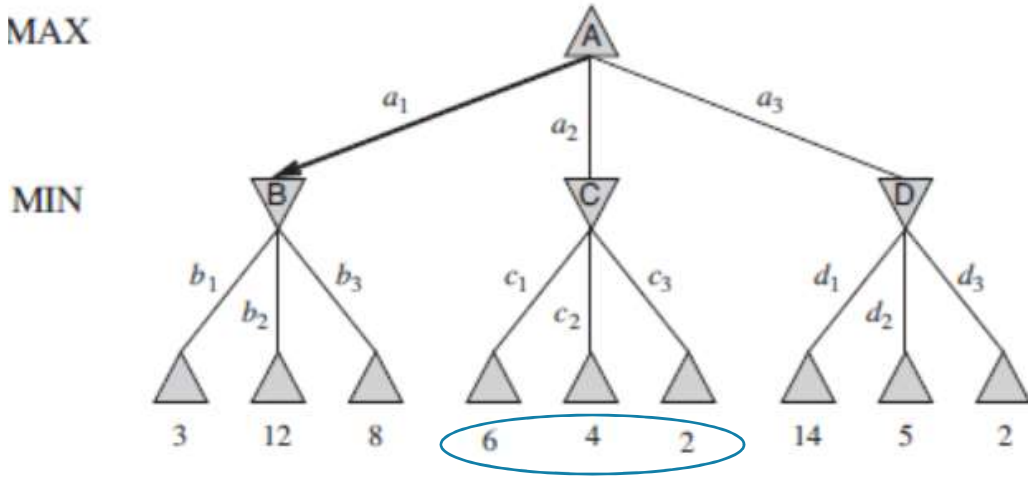
(C) (c1 c2 c3)

(2<α)



Optimal decisions— Alpha-beta pruning





The performance of alpha-beta pruning depends on the **ordering of nodes**

```
function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v          (2<α)
    β ← MIN(β, v)
  return v
```

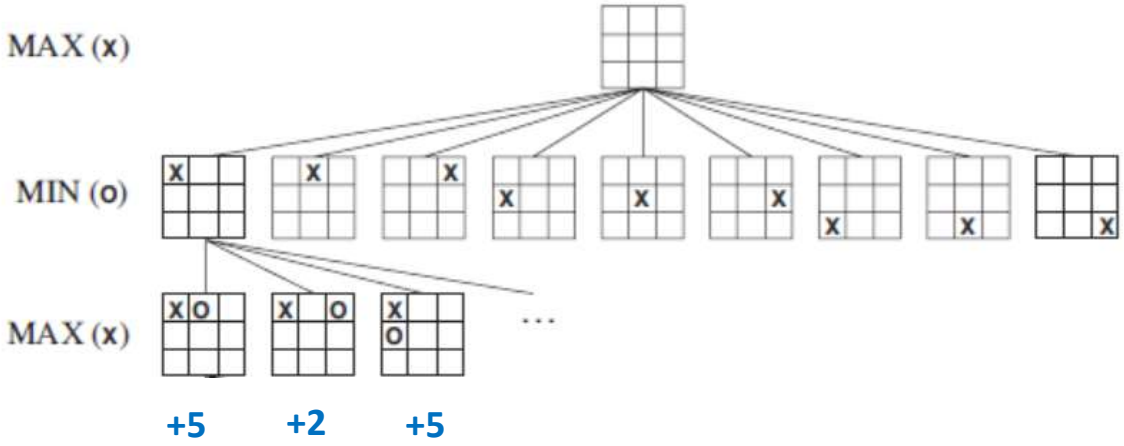
```
function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v          (α=3)
    β ← MIN(β, v)
  return v
```

Imperfect real-time decisions

- The minimax algorithm generates the entire search space, which is infeasible for most of games.
- Alpha-beta pruning can discard parts of search space. However, the search space is still huge and the pruning performance depends on the ordering of nodes.
- Hence, finding an imperfect decision is a way to play real-time games.
- Shannon proposed a heuristic evaluation function to state in the search, effectively turning ***nonterminal nodes*** into ***terminal leaves***.

Imperfect real-time decisions

- Shannon proposed a heuristic evaluation function to state in the search, effectively turning ***nonterminal nodes*** into ***terminal leaves***.



How to decide the evaluation function?

Function approximation

- The evaluation function can be approximated as a weighted **linear** function.

$$h(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s)$$

- The evaluation function can be approximated by a **nonlinear** function such as neural networks (e.g., AlphaGo).
- If the features can represent the probability of winning, the robot will make good decisions. However, the decision could not be optimal.

Function approximation

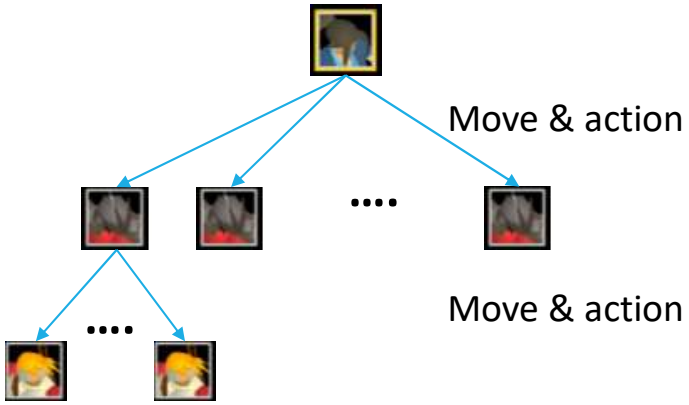
- EX: Tactics Core
- The robot computes 2 steps minimax.
- Finding 5 features of heuristic function?

$$h(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_5 f_5(s)$$

f_1 : The decreased HP of enemies

Function approximation

- Initial state: The game is set up at the start.
- Player(s): The player has the move in a state.
- Actions(a): Returns the set of possible movies in a state.
- Result(s,a): The result of a move.
- Terminal-Test(s): Return 1 or 0 for the termination.
- Utility(s,p): The final numeric value for the termination state.
  - Win: +1, Loss: -1, draw: 0



Function approximation

- In 1997, IBM Deep Blue beat the world chess champion after a six-game match.
  - 2 wins for IBM
  - 1 win for the champion
  - 3 draws

- Deep Blue can search for 30 billion positions per move and depth 14 routinely.
  - Run alpha-beta search
  - Evaluation function: 8000 features (hand crafting and database)
  - Custom VLSI processors and paralleling computation
- Humans: computers cannot beat humans in GO games!

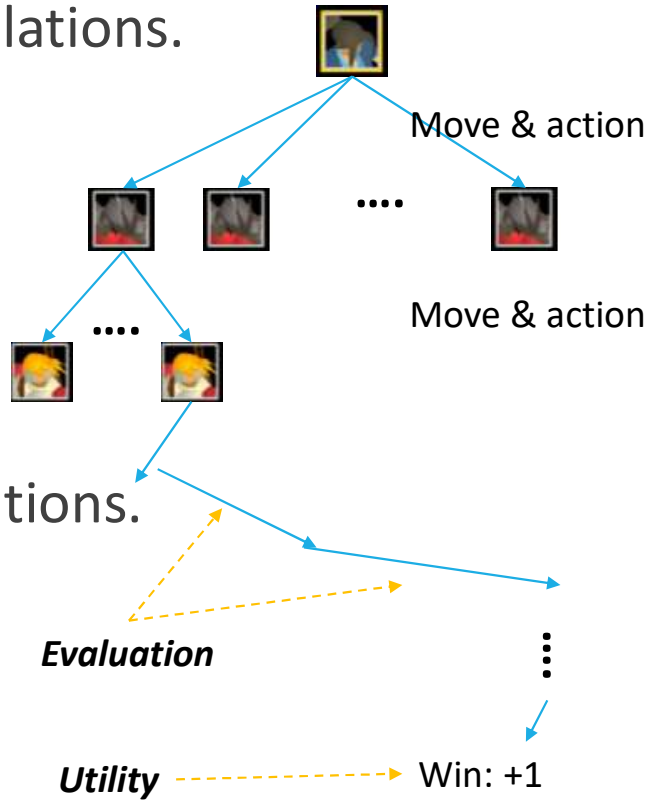
https://www.ibm.com/ibm/history/ibm100/us/en/icons/deepblue/

Function approximation

- Shannon proposed a heuristic evaluation function to state in the search, effectively turning ***nonterminal nodes*** into ***terminal leaves***.
- The evaluation functions could not represent the utility, which is +1, or -1. The robot has to search for the terminal nodes to get the utility.
- To evaluate a decision is good or not, we need an evaluation function.
- To know a decision is good or not, we need a utility function.
- Could we utilize both of ***evaluation*** and ***utility functions***?

Monte-Carlo tree search

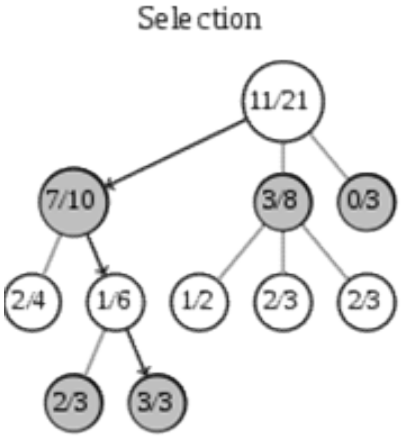
- Monte-Carlo tree search (MCTS) is a probabilistic algorithm to search for the utility outcome via random simulations.
- MCTS includes 4 steps:
  - Selection
  - Expansion
  - Simulation
  - Backpropagation
- Given fixed budget (e.g., time), MCTS find solutions.

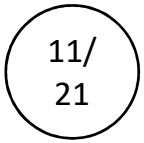


https://en.wikipedia.org/wiki/Monte_Carlo_tree_search

Monte-Carlo tree search

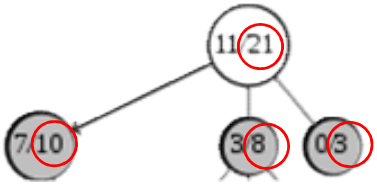
- Notations of MCTS:



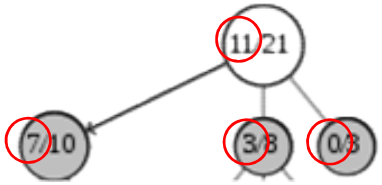


Player 1,
The total number of simulated games: 21
The number of winning in this subtree: 11

Player 2,
The total number of simulated games: 10
The number of winning in this subtree: 7



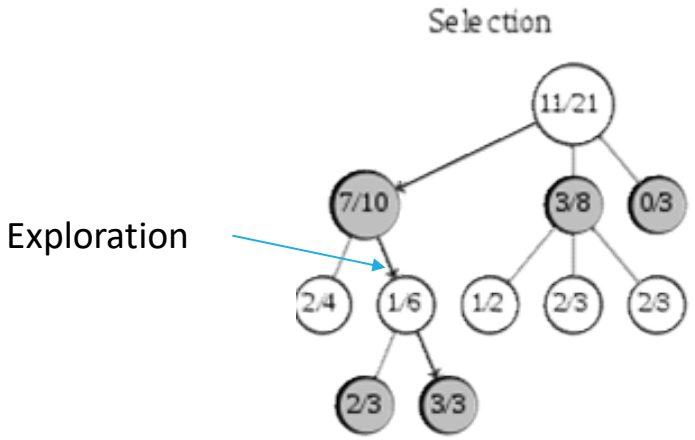
21=10+8+3



21=11+7+3+0

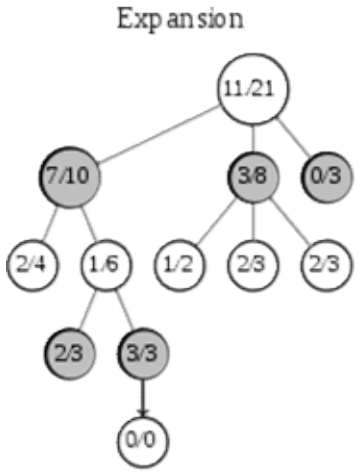
Monte-Carlo tree search

- *Selection*: start from the **root** (*R)* and select successive child nodes until a **leaf** (*L)* node is reached. The root is the current game state and a leaf is any node from which no simulation (playout) has yet been initiated. The selection step needs to face a trade-off between exploration and exploitation.



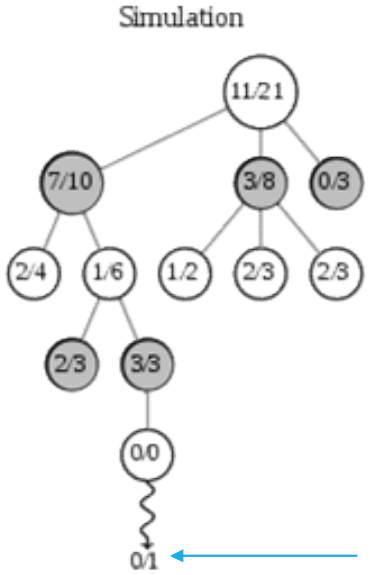
Monte-Carlo tree search

- *Expansion*: unless (*L)* ends the game decisively (e.g., win/loss/draw) for either player, create one or more child nodes and choose node (*C*) from one of them. Child nodes are any valid moves from the game position defined by (*L)*.



Monte-Carlo tree search

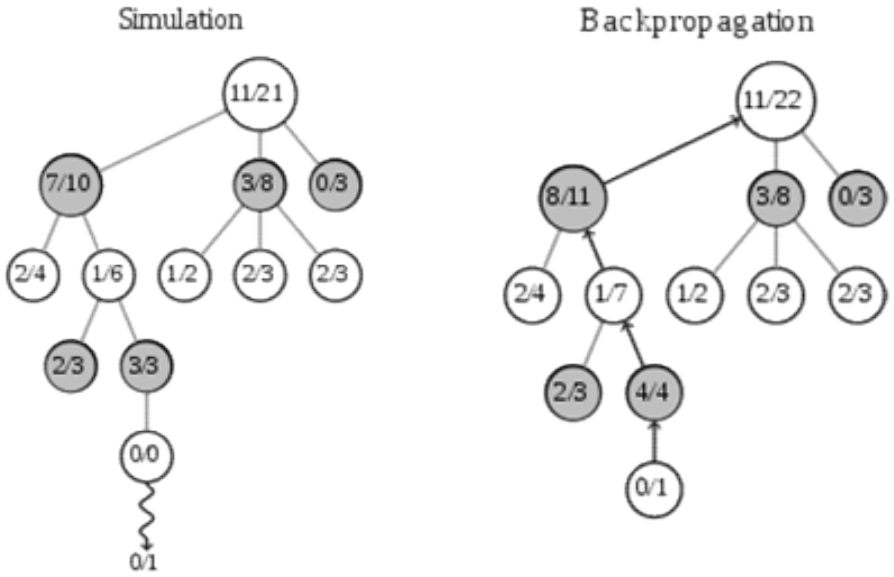
- *Simulation*: complete one random playout from node *C*. This step is sometimes also called playout or rollout. A playout may be as simple as choosing uniform random moves (or using an evaluation function) until the game is ended (for example in chess, the game is won, lost, or drawn).



Simulation step can run via parallel computing!

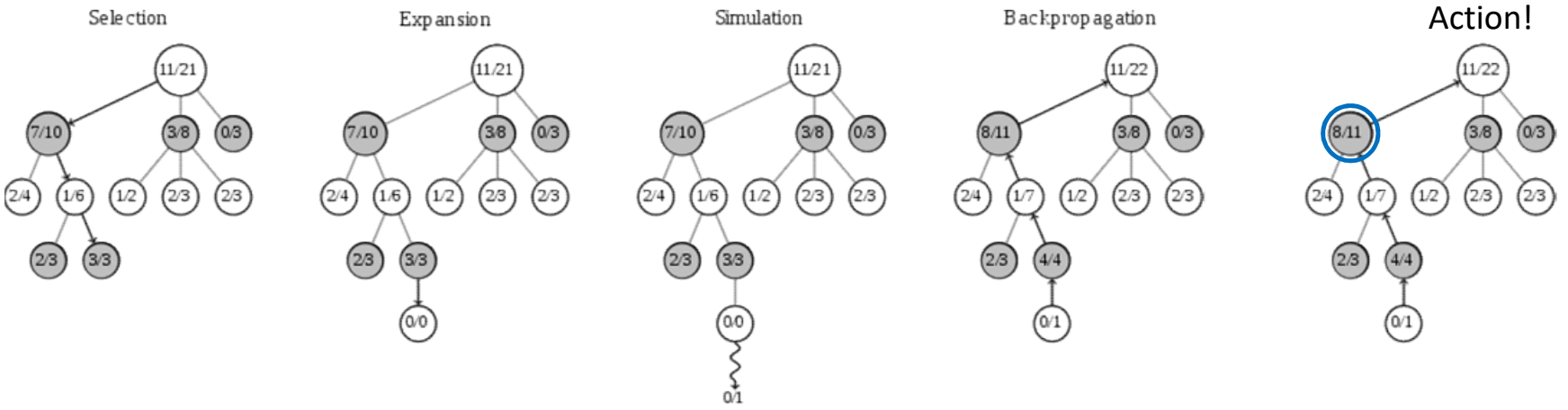
Monte-Carlo tree search

- *Backpropagation*: use the result of the playout to update information in the nodes along the path from *bottom* to *root*.



Monte-Carlo tree search

- Selection → Expansion → Simulation→ Backpropagation
- Given fixed budget, MCTS finds solutions and actions. Then, the robot changes the tree root and runs MCTS again until it wins or loses.



Monte-Carlo tree search

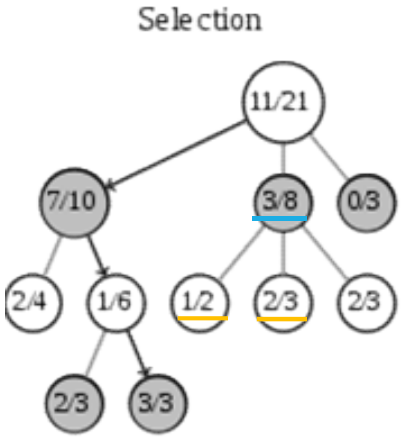
- In the selection step, the robot has the dilemma of ***exploration and exploitation***. Exploitation is to select the best action so far. Exploration is to select some uncertain actions.
- The upper confidence bound applied to tree (UCT) is one approach to decide which node should be expanded.

$$UCT(node) = \underbrace{\frac{W(node)}{n(node)}}_{\text{exploitation}} + \underbrace{C\sqrt{\frac{\ln(n(parent\ node))}{n(node)}}}_{\text{exploration}}$$

W : the number of wins for the node

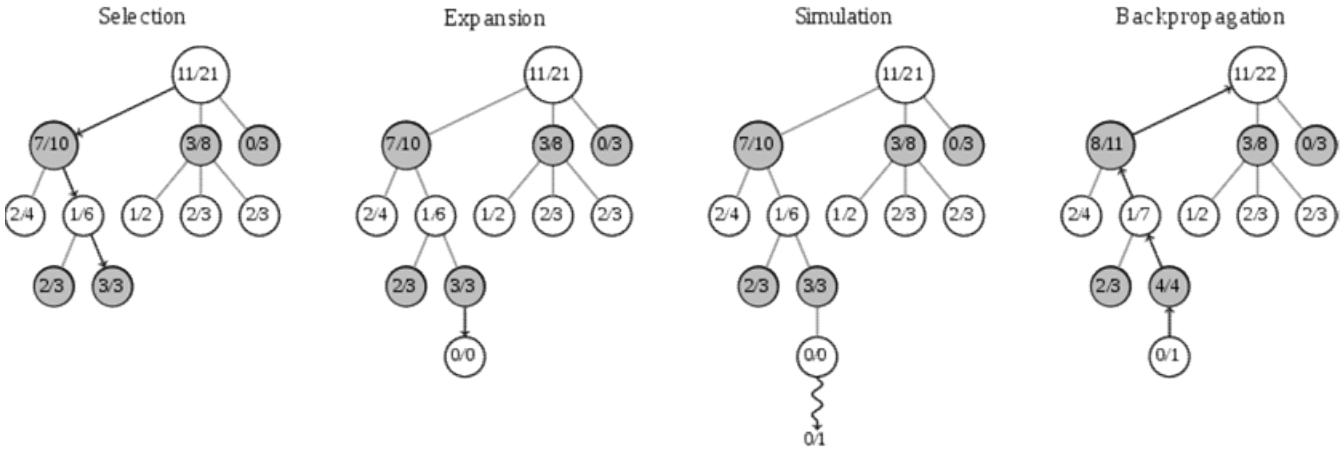
n : the number of simulations for the node

C : the exploration parameter



Monte-Carlo tree search

- MCTS includes 4 steps:
  - Selection: select the leaf via maximal winning values or UCT
  - Expansion: expand a node
  - Simulation: simulate games via random actions or evaluation functions
  - Backpropagation: update the w/n rate of the path



EX: AlphaGo

- Why Go is a challenging game for robots?

$$Chess: b \approx 35, d \approx 80, b^d \approx 10^{80}$$

$$Go: b \approx 250, d \approx 150, b^d \approx 10^{170}$$

- The size of GO search tree is more than the number of atoms in the universe!

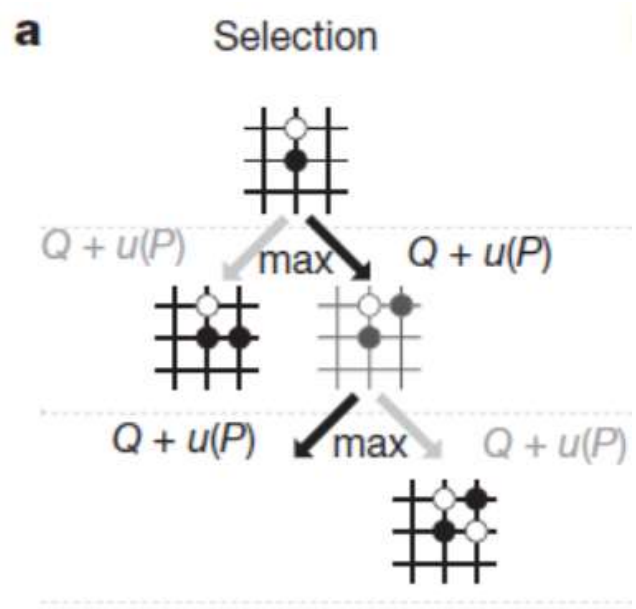
EX: AlphaGo

- AlphaGo adopts MCTS and deep reinforcement learning (DRL). In 2016, AlphaGo beat the best human player in Go!
- MCTS: search for actions
- DRL: learn value (s) and policy (a) functions via deep neural networks
- In 2015, DeepMind's AlphaGo beat European Champion 5:0
- In 2016, AlphaGo beat World Champion (Lee Sedol) 4:1
- In 2017, AlphaGo Zero beat AlphaGo 100:0
- After winning Go Games, Deepmind announced to win StarCraft II. In 2019, AlphaStar beats the best human players.

David Silver, et. al, "Mastering the game of Go with deep neural networks and tree search," Nature, 2016.

EX: AlphaGo

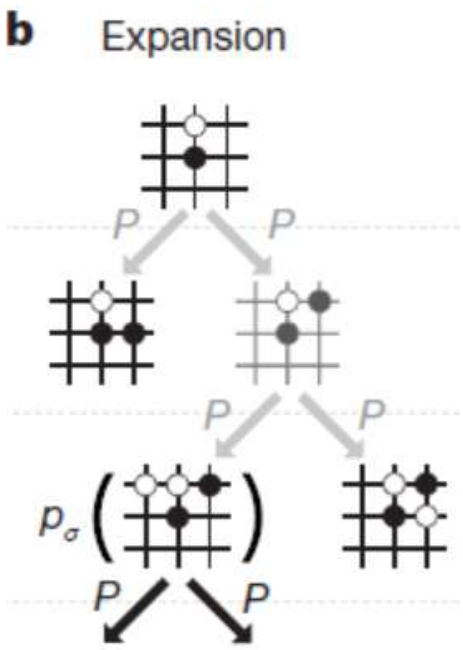
- Selection

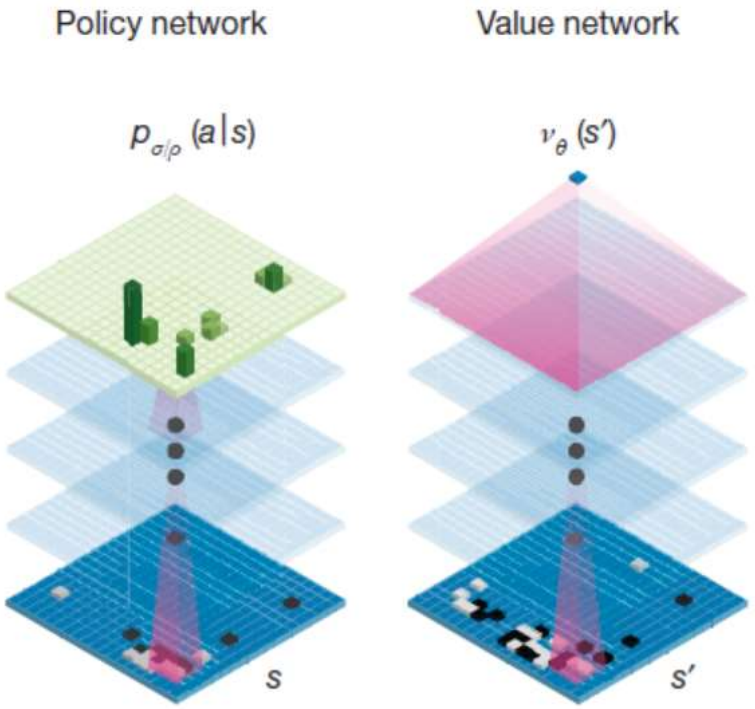


- P(s,a): prior distribution
- Q(s,a): action value
- Q+u(P) is an evaluation function for selection

EX: AlphaGo

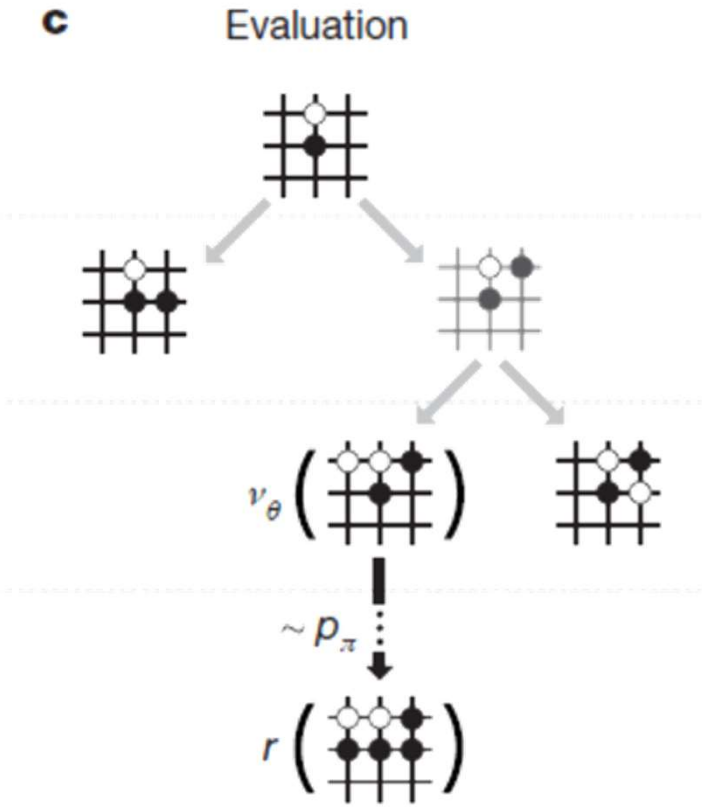
- Expansion





EX: AlphaGo

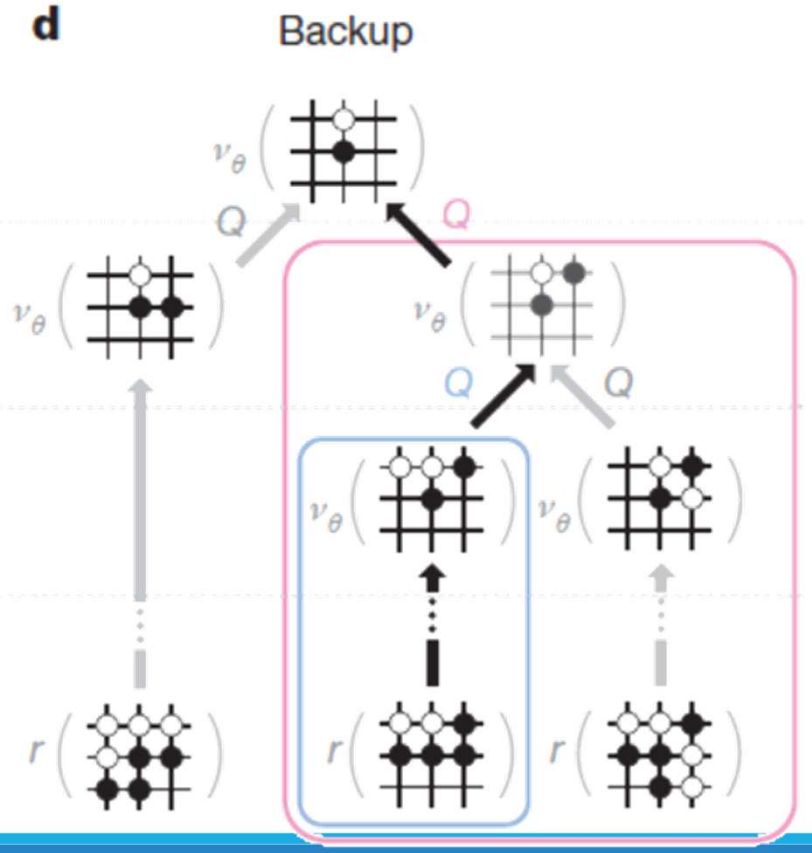
- Simulation



- Run multiple simulations with **parallel** computers.
- Parts of simulations run with **value networks**. The others run with the **outcome of the game** (+1 or -1)

EX: AlphaGo

- Backpropagation



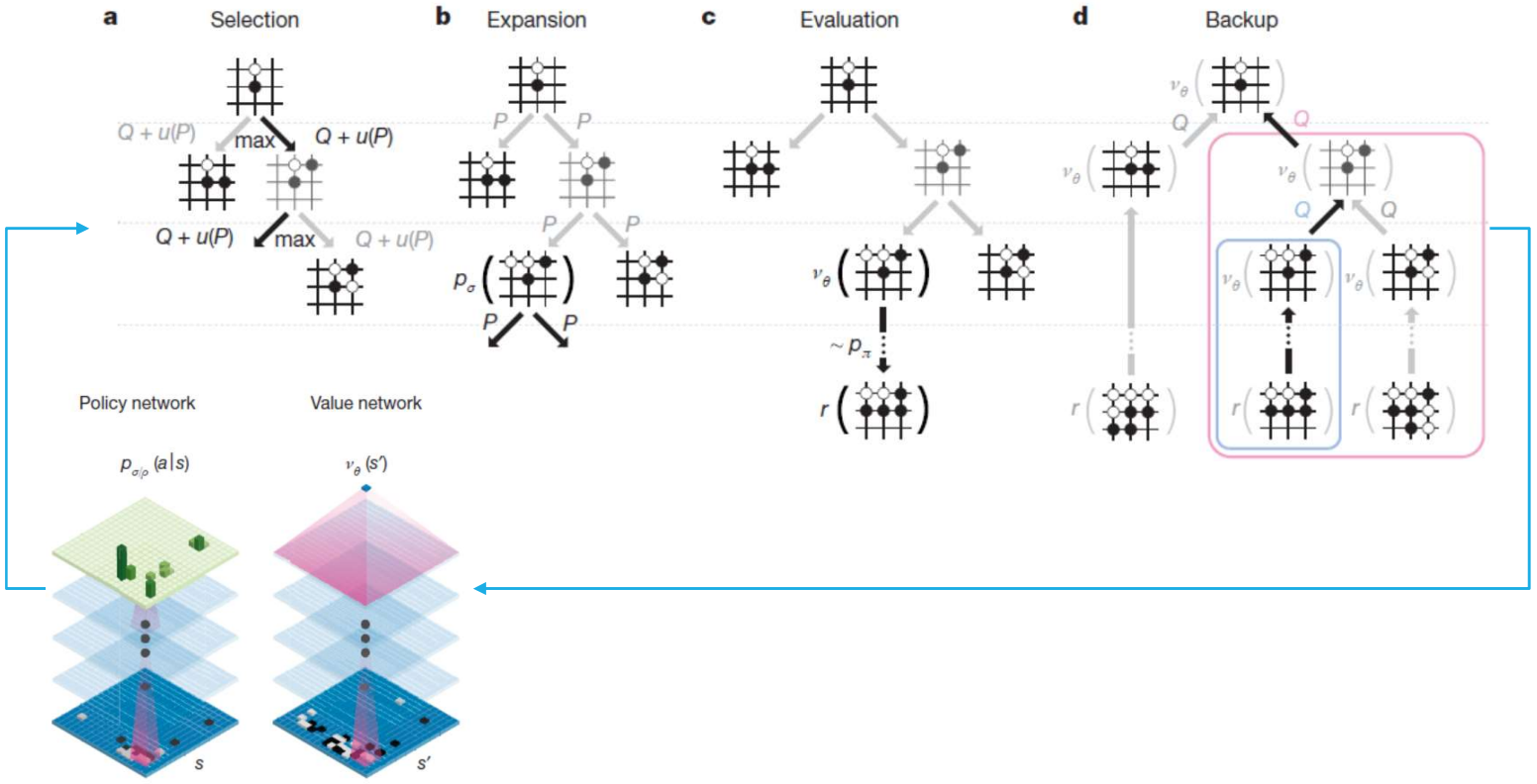
$Q(s,a)$: action value function

$v_\theta(s,a)$: value function

r : the winner function

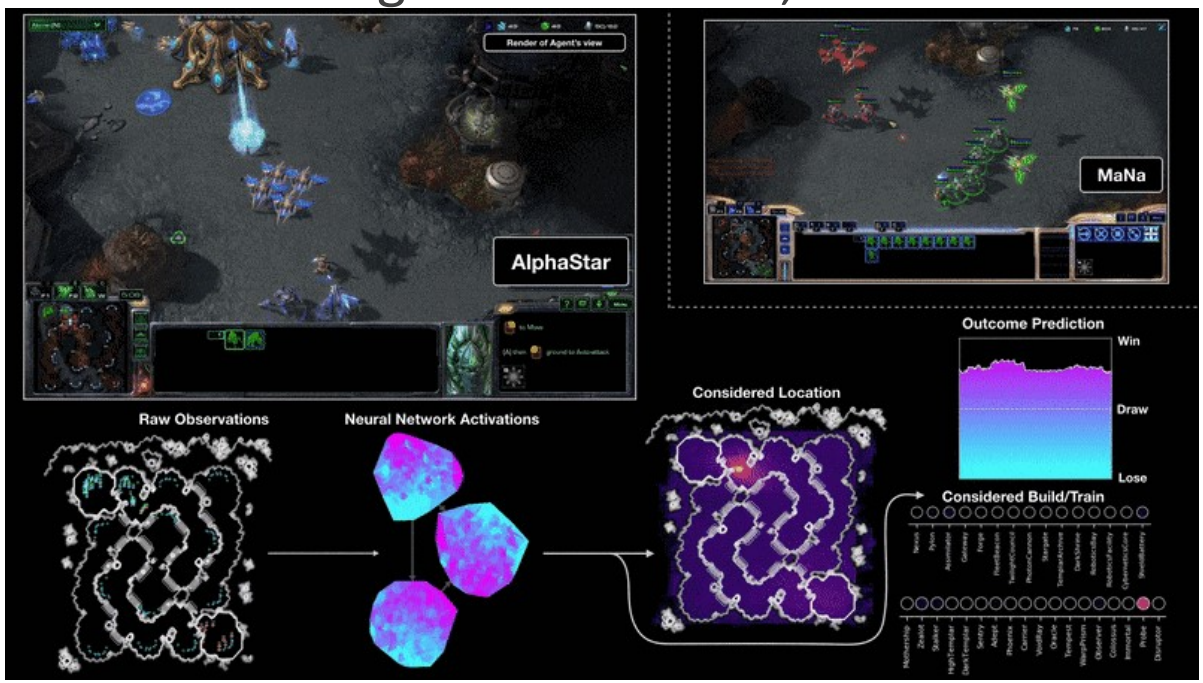
EX: AlphaGo

- Selection → Expansion → Simulation→ Backpropagation



EX: AlphaGo → AlphaStar

- The Challenge of StarCraft II, 2019



https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/?fbclid=IwAR0oUtPE9az5V1kKXlSNJOncd1Ys-VP1r3PVICOV5gb0mI39uMowI6SURl8
https://www.youtube.com/watch?v=cUTMhmVh1qs

Conclusion

- The search space of games is too huge to find optimal solutions (e.g., minimax and alpha-beta pruning). MCTS and function approximation provide feasible approaches to find a suboptimal action within fixed budget.
- MCTS+DRL conquer these games, which humans were good at. It means that robots are better than humans in any domains? No, currently they are good at something on the simulators.
- The states of aforementioned games are observable. For example, the robot knows the position of its chess or its army. In the real world, it is not easy to know the states. It's a ***perception*** problem. We will talk about how to deal with perception problems via Bayes theorem (proposed in 1763).

Q&A