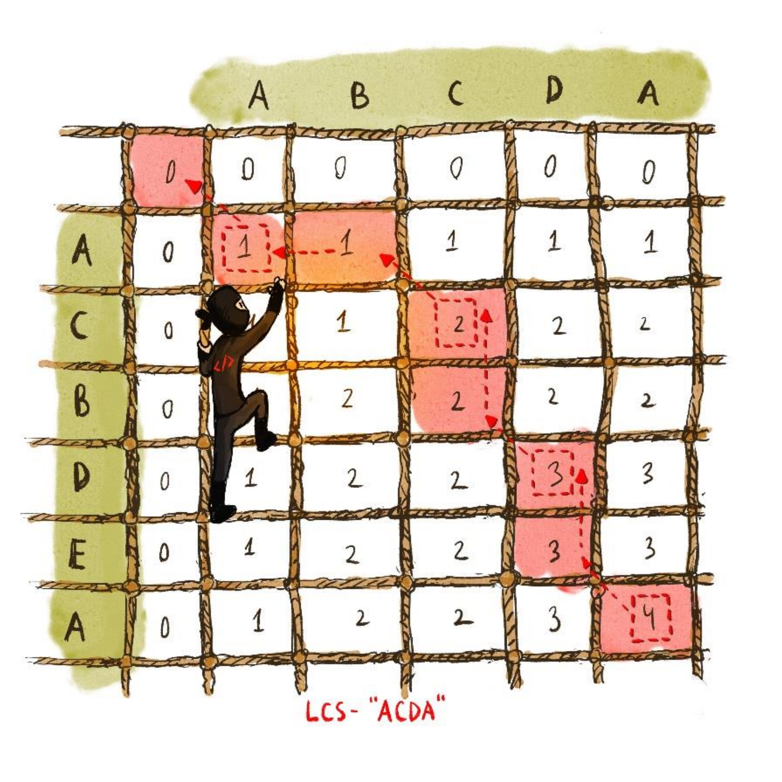
CS218: Design And Analysis Of Algorithms

Dynamic Programming V

DP for Games

Yan Gu



Game theory

- Game theory is the study of mathematical models of strategic interaction among rational decision-makers

- Happens in many fields including social science, computer science, economics, …

- Make decisions to maximize your benefits

Example - NIM

- Two players take turns to remove some pebbles from a pile
- Can take 1, 2, or 3 pebbles
- The player taking the last pebble wins
- What happens if we have 6 pebbles?
- What about 17? 20?

A winning strategy for the first player

- If $n \bmod 4 = r \neq 0$
- Take r pebbles first
- Then if the second player takes x pebbles, you take 4-x pebbles
- Always keep the total number of pebbles a multiply of 4

- Always wins!

What if $n = 4p$?

- If n is a multiply of 4, the first player does not have a winning strategy because the second player has a winning strategy

- Whatever the first player does, the second player can keep the number of pebbles a multiply of 4

- If the second player always plays rationally, the first player cannot win

OK this is simple… what happens if it's not 1, 2, 3?

- What if you can only take 1, 2 or 4 pebbles at a time?
- If there are 10 pebbles, who will win?
- If you are the first player, what is your best strategy?
- What if you can only take a[1], a[2], a[3], …, a[m] pebbles at a time, for some input a[1..m]?

Assume we can take 1, 2, or 4 pebbles
What happens if you are left with x pebble(s)

$x = 1$:
- take them all, win! ☺

$x = 2$:
- take them all, win! ☺

$x = 3$:
- no matter how many you take, player 2 can take all the rest, lose ☹

$x = 4$:
- take them all, win! ☺

$x = 1$, win!
$x = 2$, win!
$x = 3$, lose…
$x = 4$, win!

Assume we can take 1, 2, or 4 pebbles
What happens if you are left with x pebble(s)

$x = 5$:

- Take 2 pebbles
- The player 2 is left with 3 pebbles
- No matter how many pebbles player 2 takes, you take all the rest, you win! ☺

$x = 1,$ win!
$x = 2,$ win!
$x = 3,$ lose…
$x = 4,$ win!
$x = 5,$ win!

Assume we can take 1, 2, or 4 pebbles
What happens if you are left with x pebble(s)

$x = 6$:

- If you take 1 pebble, player 2 will be left with 5 pebbles, s/he can just take another 2 pebbles to make you lose ☹
- If you take 2 pebbles, player 2 will be left with 4 pebbles, s/he will take all and you lose ☹
- If you take 4 pebbles, player 2 will be left with 2 pebbles, s/he will take all and you lose ☹
- Whatever you do you will lose ☹

$x = 1,$ win!
$x = 2,$ win!
$x = 3,$ lose…
$x = 4,$ win!
$x = 5,$ win!
$x = 6,$ lose…

Assume we can take 1, 2, or 4 pebbles
What happens if you are left with x pebble(s)

$x = 7$:

- Just take 1 pebble. Player 2 will be left with 6 pebbles
- We've proved just now that whatever s/he does, s/he will lose …
- So you win! ☺
- (you can also take 4 and leave 3 to player 2, and you will also win)

$x = 1$, win!
$x = 2$, win!
$x = 3$, lose…
$x = 4$, win!
$x = 5$, win!
$x = 6$, lose…
$x = 7$, win!

Assume we can take 1, 2, or 4 pebbles What happens if you are left with x pebble(s)

$x = 8$:

- Win! Take 2 pebbles to make x = 6

$x = 9$

- Lose..
  - If you take 1, the other player see 8: win
  - If you take 2, the other player see 7: win
  - If you take 4, the other player see 5: win

$x = 10$

- Win! Take 1 pebble to make x = 9

$x = 1$, win!
$x = 2$, win!
$x = 3$, lose…
$x = 4$, win!
$x = 5$, win!
$x = 6$, lose…
$x = 7$, win!

Assume we can take 1, 2, or 4 pebbles
What happens if you are left with x pebble(s)

- **Let $f[i] = win$ if i is a winning state**
  - When you see i pebbles, you have a winning strategy)
- **Let $f[i] = lose$ if i is a losing state**
  - When you see i pebbles, player 2 has a winning strategy, so whatever you do, you may lose, if player 2 is smart enough)

$\boldsymbol{x = 0,}$ **lose!**
$x = 1,$ win!
$x = 2,$ win!
$x = 3,$ lose.
$x = 4,$ win!
$x = 5,$ win!
$x = 6,$ lose…
$x = 7,$ win!

winning state

losing state

- **$f[i] = win$ iff. at least one of $f[i-1]$, $f[i-2]$ or $f[i-4]$ is “lose”.**
  - You take the corresponding number of pebbles, leaving a losing state to player 2
- **$f[i] = lose$ iff. all $f[i-1]$, $f[i-2]$ and $f[i-4]$ are “win”.**
  - Then whatever you do, player 2 is winning!

Assume we can take 1, 2, or 4 pebbles
What happens if you are left with x pebble(s)

- Let $f[i] = win$ if i is a winning state, and $f[i] = lose$ if i is a losing state

- $\boldsymbol{f[i] = win}$ **iff. at least one of** $\boldsymbol{f[i - a[j]]}$ **is "losing".** Then you take the corresponding number of pebbles, leaving a losing state to player 2

- $\boldsymbol{f[i] = lose}$ **iff. all** $\boldsymbol{f[i - a[j]]}$ **are "winning".** Then whatever you do, player 2 is winning!

- Can be generalize to any set of numbers to take
- Boundary: $f[0] = lose$
  - if you are left with 0 pebbles, it means player 2 has won!

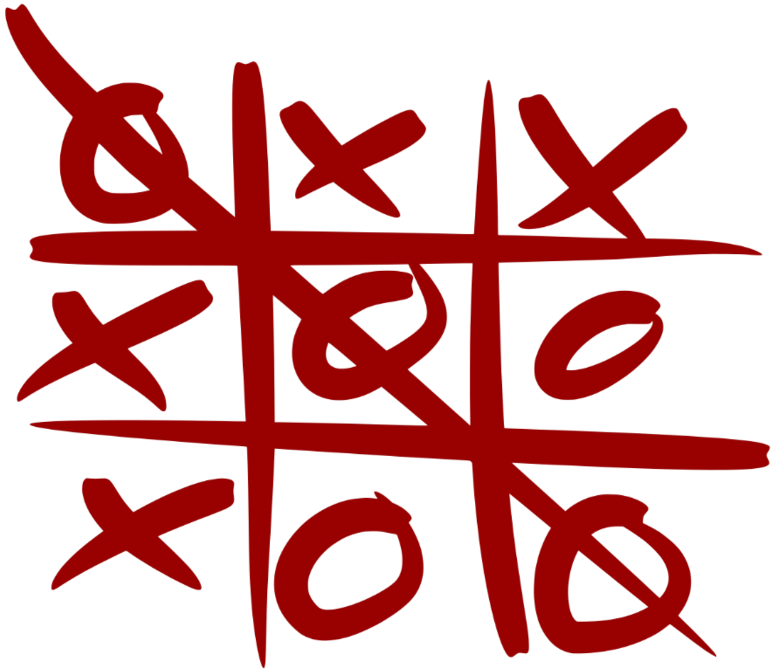
NIM – algorithm

- **if you can only take a[1], a[2], a[3], …, a[m] pebbles at a time, for some input a[1..m], given initial number of pebbles n, decide who will win**
- **Assume one of a[1..m] is 1 so that the game can finish**

```
Initialize f[0..n]
f[0] = lose;
for i = 1 to n {
  f[i] = lose;
  for j = 1 to m {
    //if taking a[j] pebbles makes player 2 lose
    if (i>=a[j] && f[i-a[j]] = lose) {
      f[i] = win; break;
    }
  } // if none of a[j] makes f[i-a[j]] lose, f[i] will stay "lose"
}
if (f[n] is win) output "player 1 wins"; else output "player 2 wins";
```

In homework 4

- Tic-tac-toe
- State is more complicated
- Decide who will win



Impartial games

- Two players must alternate turns until a final state is reached
- A winner is chosen when one player may no longer change position or make any operation
- There must be a finite number of operations and positions for both players
- All operations must be able to be done by both sides
- No action in the game may be reliant on chance

The Sprague–Grundy theorem

- Every impartial game is equivalent to a one-heap game of NIM, or to an infinite generalization of NIM
- Every game can be represented by a NIMBER (NIM number) as a non-negative integer
- A algebraic system whose addition operation combines multiple heaps to form a single equivalent heap in NIM, usually by MEX ("**m**inimum **ex**cluded value“, e.g., MEX({0,1,3,4})=2)
- NIMBERs can be combined: e.g., the NIMBER for a two-pile NIM game of NIMBERs a and b is $a \operatorname{xor} b$

Impartial games

- Two players must alternate turns until a final state is reached
- A winner is chosen when one player may no longer change position or make any operation
- There must be a finite number of operations and positions for both players
- All operations must be able to be done by both sides
- No action in the game may be reliant on chance

Extensions to partisan games

- Development for Deep Blue began in 1985 at CMU, later funded by IBM
- It won the world champion Garry Kasparov in the six-game rematch by 3½–2½ in 1997
- The computer program was similar to what we have described in this class, but of course, with some additional ideas

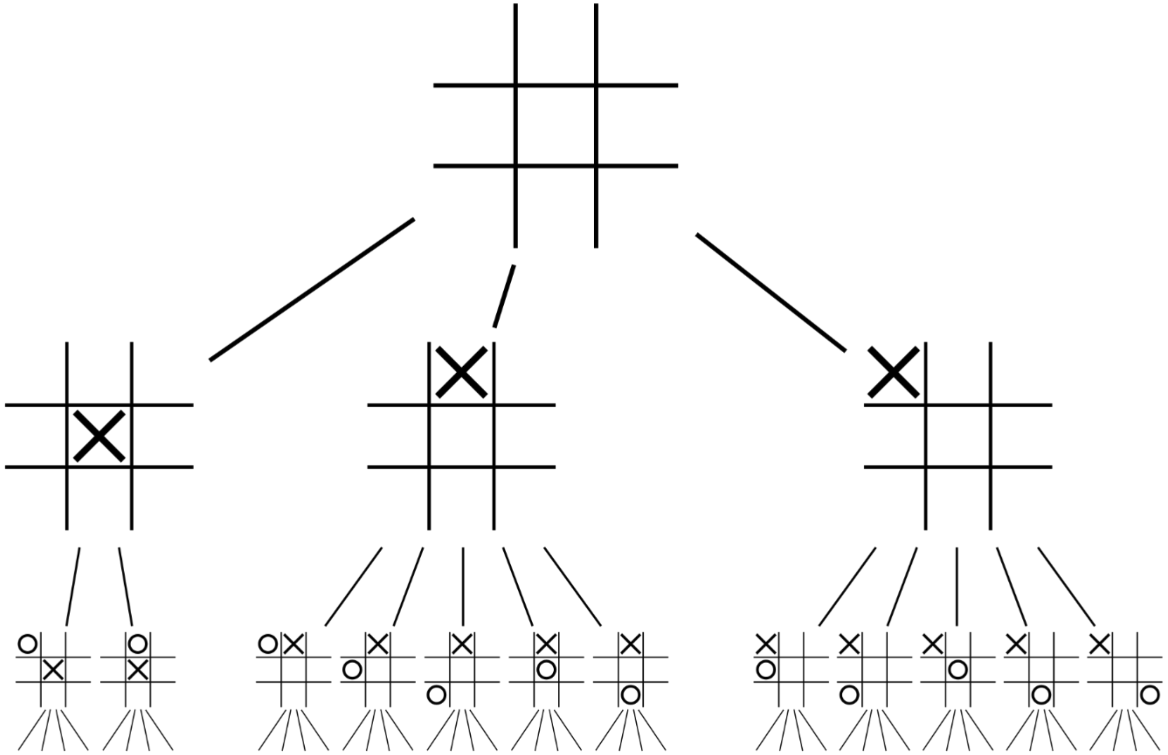
Let’s first revisit the NIM solution

- Can we still enumerate and list the states of ongoing chess boards?

```
Initialize f[0..n]
f[0] = lose;
for i = 1 to n {
  f[i] = lose;
  for j = 1 to m {
    //if taking a[j] pebbles makes player 2 lose
    if (i>=a[j] && f[i-a[j]] = lose) {
      f[i] = win; break;
    }
  } // if none of a[j] makes f[i-a[j]] lose, f[i] will stay “lose”
}
if (f[n] is win) output “player 1 wins”; else output “player 2 wins”;
```

Game tree search

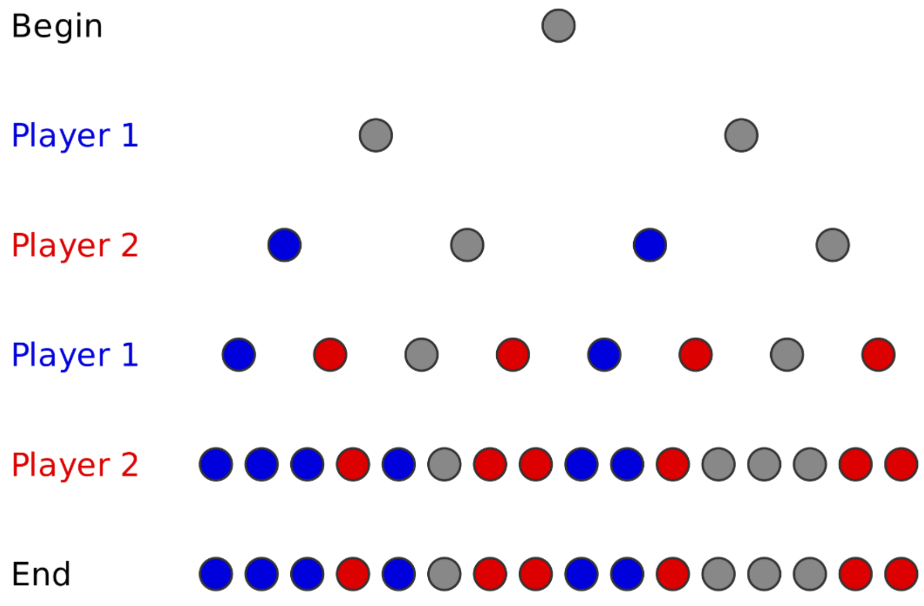
- **Instead of storing all states, you can only search those reachable states, but the logic remains the same**



```
Play (state s) {
  if (s is a final state) return win/loss;
  for j = 1 to m { // for all possible moves
    s' = the state after the j-th move
    if (Play(s') = lose)
      return win;
  }
  return lose; // if none of the moves makes s' lose, s will stay "lose"
}
```

Two-player game tree search

```
Player-1 (state s) {
  if (s is a final state) return win/loss;
  for j = 1 to m { // for all possible moves
    s' = the state after the j-th move
    if (Player-2(s') = win) // for player 1
      return win;
  }
  return lose;
}
Player-2 (state s) {
  if (s is a final state) return win/loss;
  for j = 1 to m { // for all possible moves
    s' = the state after the j-th move
    if (Player-1(s') = lose) // for player 1
      return lose;
  }
  return win;
}
```



Heuristic-based search

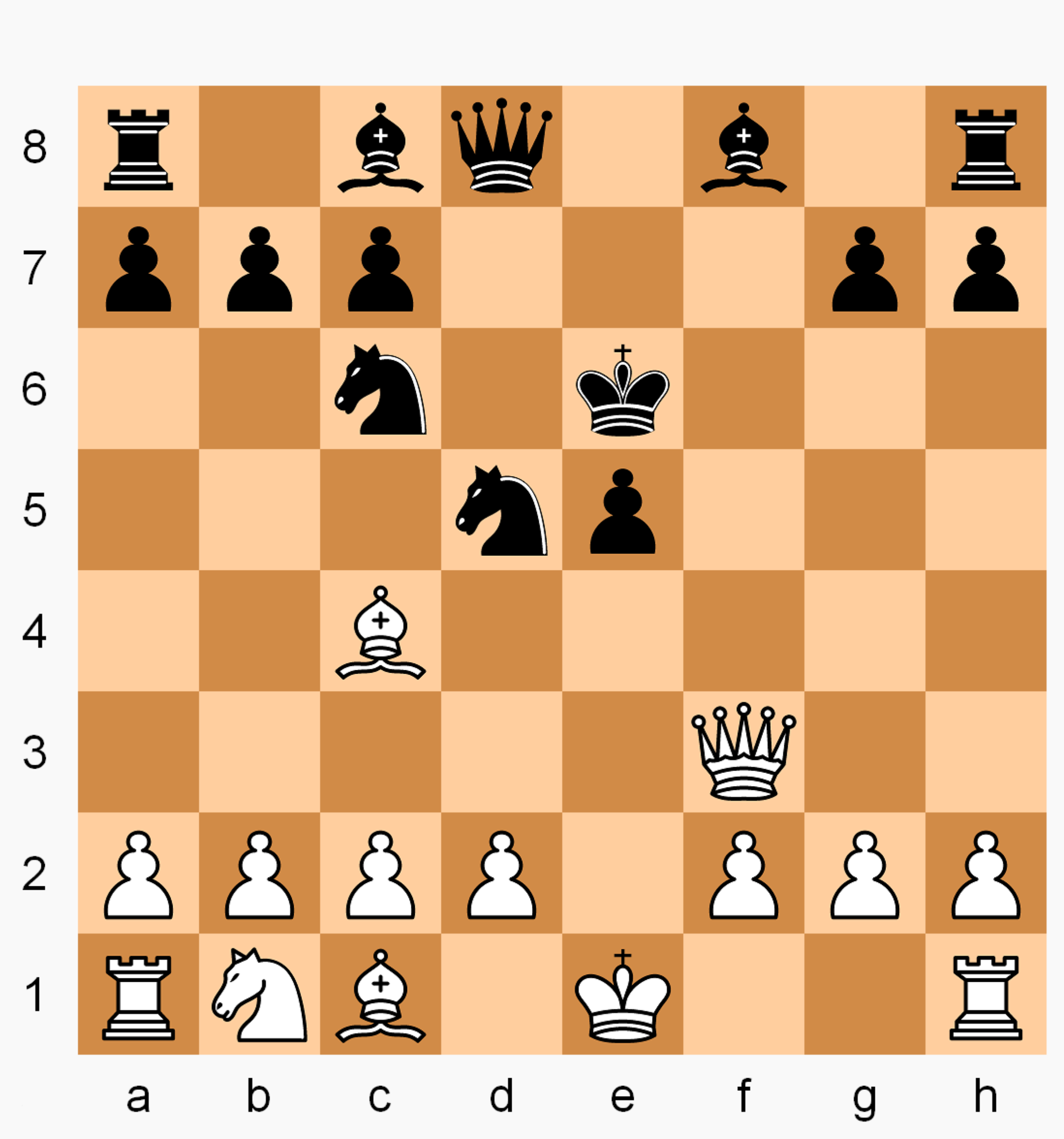
```
Player-1 (state s) {
  if (s is a final state) return win(1)/loss(-1)/draw(0);
  if (s is a deep enough) return h(s); // between 1 and -1
  for j = 1 to m { // for all possible moves
    s' = the state after the j-th move
    if (Player-2(s') = win)
      return win;
  }
  return lose;
}
Player-2 (state s) {
  if (s is a final state) return win(1)/loss(-1)/draw(0);
  if (s is a deep enough) return h(s); // between 1 and -1
  for j = 1 to m { // for all possible moves
    s' = the state after the j-th move
    if (Player-1(s') = lose)
      return lose;
  }
  return win;
}
```

Heuristic-based search

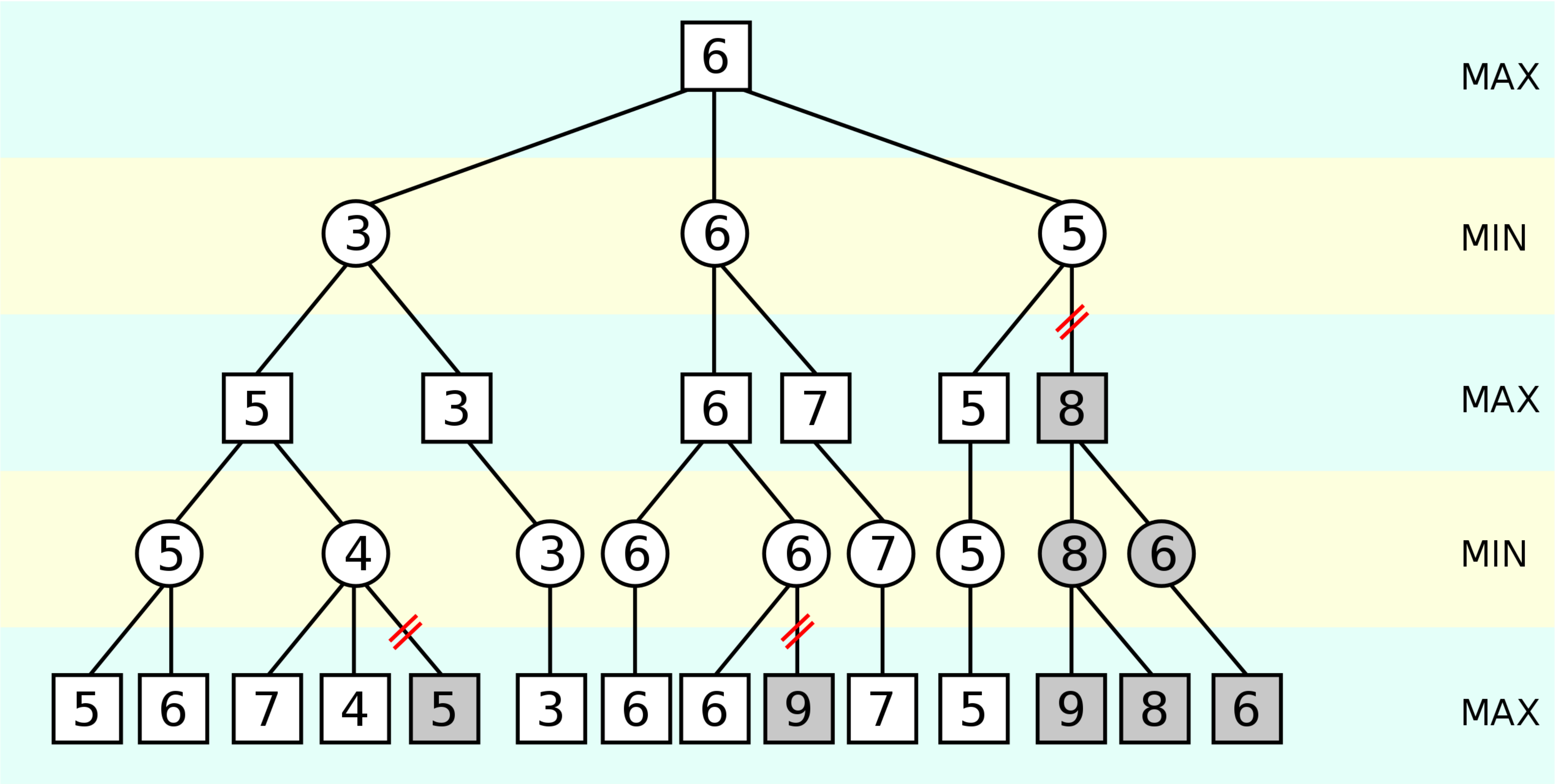
```
Player-1 (state s) {
  if (s is a final state) return win(1)/loss(-1)/draw(0);
  if (s is a deep enough) return h(s);
  r = -1;
  for j = 1 to m { // for all possible moves
    s' = the state after the j-th move
    r = max(r, Player-2(s'));
}
  return r;
}
Player-2 (state s) {
  if (s is a final state) return win(1)/loss(-1)/draw(0);
  if (s is a deep enough) return h(s);
  r = 1;
  for j = 1 to m { // for all possible moves
    s' = the state after the j-th move
    r = min(r, Player-1(s'));
  }
  return r;
}
```

Further optimizations example: α-β pruning

- “No need to search for silly moves”
- We don’t need to search too much for “Queen f5”
- For the max-player, if a current state cannot be a new max, then the search on this branch can be skipped



α-β pruning example



Other optimizations

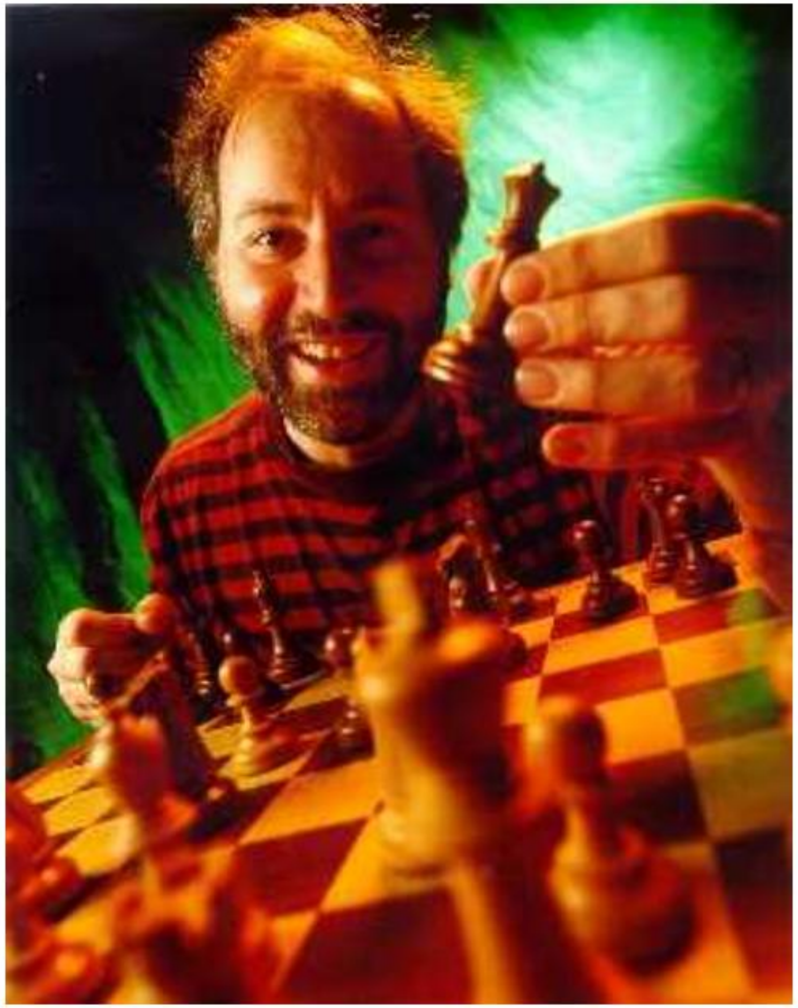
- Parallelism is desperately needed to search deep trees
- Chess playing is one of the Cilk's motivating goals
- (Picture: Thinking Machine CM-5, fastest supercomputer at that time)
- Lots of CS techniques are designed in this process

Anecdotes

Charles E. Leiserson's MIT Homepage

Charles E. Leiserson is Professor of Computer Science and Engineering at MIT.
Engineering and Computer Science (EECS). He was selected as a Margaret Mac
and former Associate Director and Chief Operating Officer of MIT's Computer S
Computation Group (TOC), and head of the Lab's Supertech Research Group. Pr
professional societies: ACM, AAAS, SIAM, and IEEE.

Daniel Dominic Kaplan Sleator

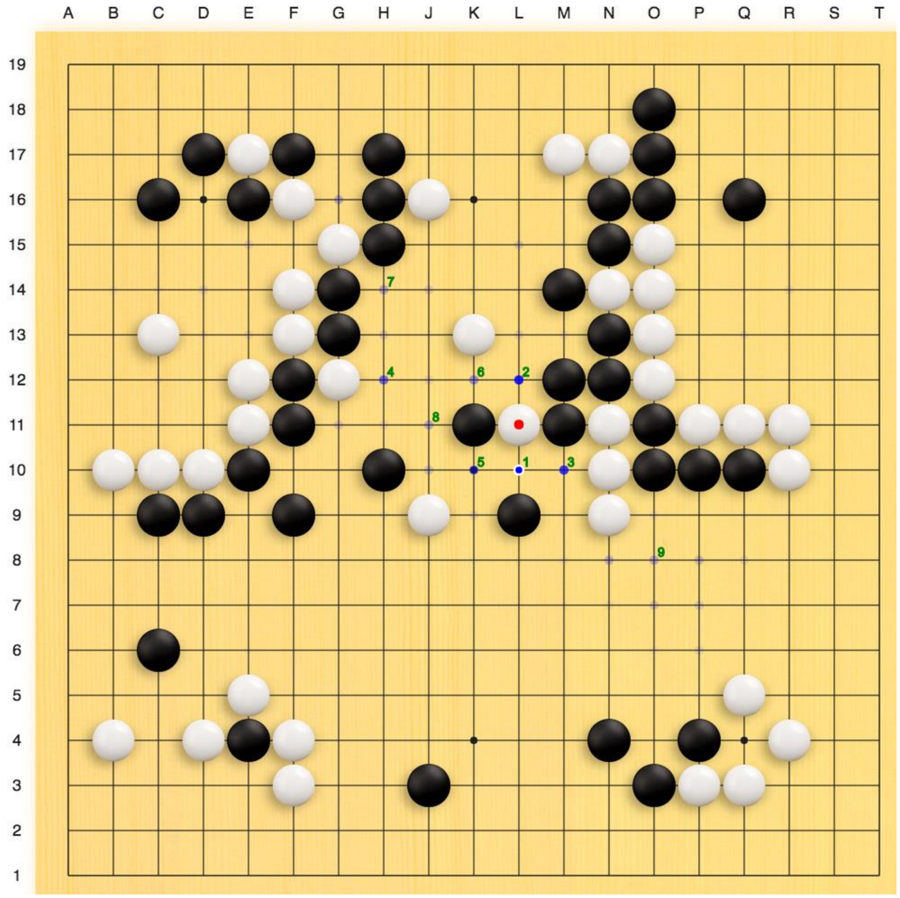


Professor of Computer Science
sleator@cs.cmu.edu

After 20 years: AlphaGo

- **Observation: since the game board of Go is much larger**

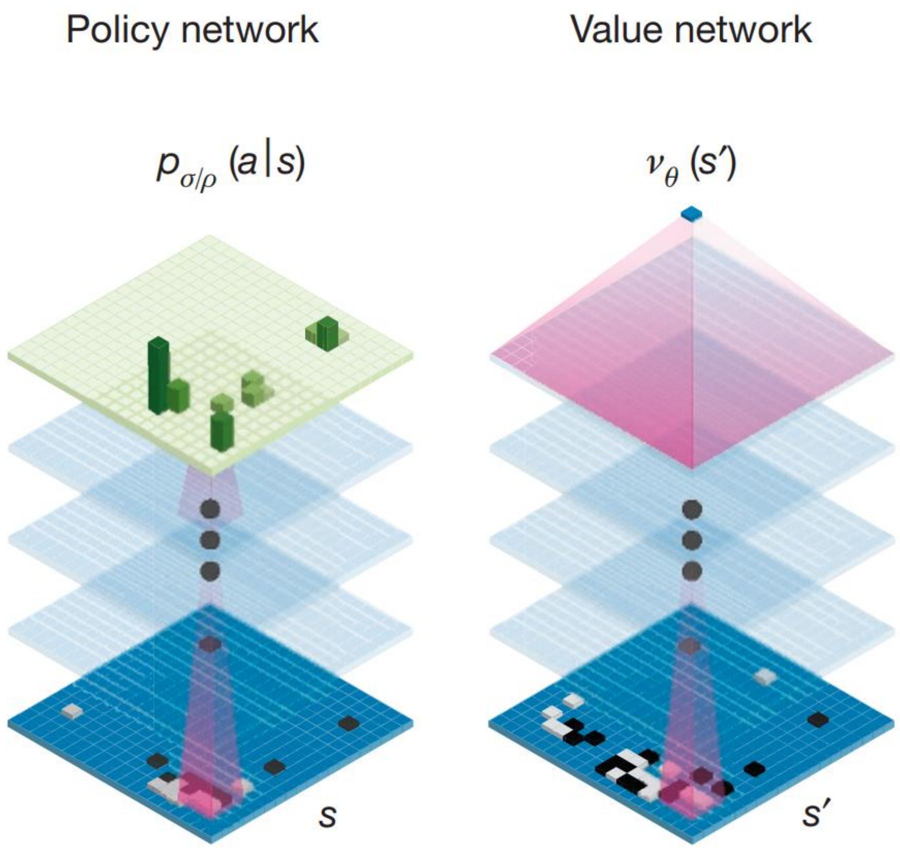
- **AlphaGo beat Ke Jie in 2017 summer, with the additional support of:**
  - Evaluation neural network (value network)
  - Monte Carlo tree search



After 20 years: AlphaGo

- **Observation: since the game board of Go is much larger**

- **AlphaGo beat Ke Jie in 2017 summer, with the additional support of:**
  - Evaluation neural network (value network)
  - Monte Carlo tree search
  - Reinforcement learning



Combinatorial game theory

- Two(or more)-player games vs. one player puzzle
- “traditional” games vs. “economic” games
- Perfect information vs. partial information
- Deterministic vs. randomness included
- Can draw vs. always a winner

Summary

DP for games

- **Define “winning state” and “losing state”**
- **A state is “winning” if you can take some action to make it a “losing state”**
  - As long as there exists such an action
- **A state is “losing” if whatever action you take, the next state is a “winning state”**
  - All possible next states must be “losing”, so you have no way to win

- **Compute “winning/losing” for all states**

Summary for Dynamic Programming

Dynamic Programming (DP)

- **DP is not an algorithm, but an algorithm design idea (methodology)**

- **DP works on problems with optimal substructure**

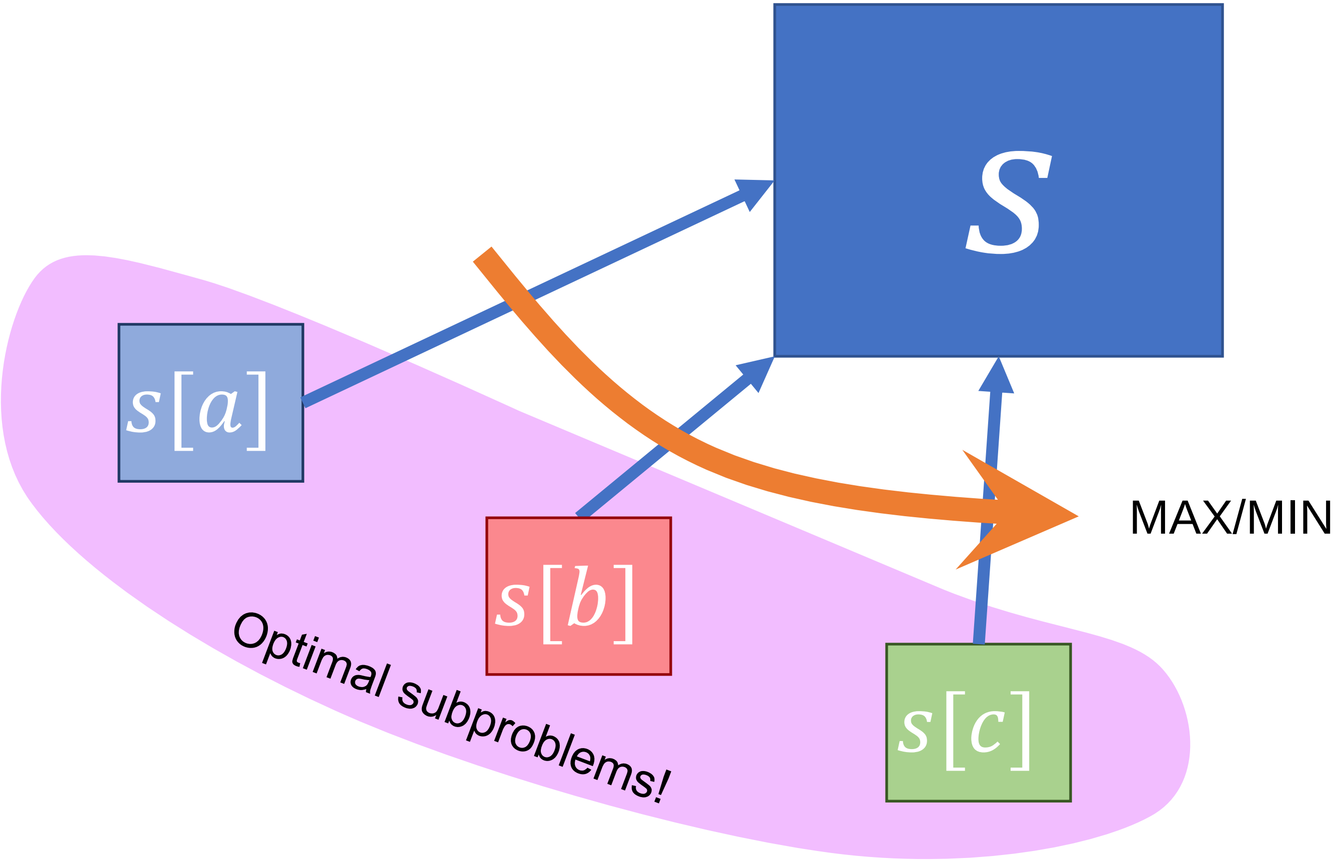
- **A DP recurrence of the states, with boundary cases**

- **We can convert a DP recurrence to a DP algorithm**
  - Recursive implementation: straightforward
  - Non-recursive implementation: faster, and easy to be optimized

What is dynamic programming?

$$s[i,j] = \begin{cases} s[i-1,j-1]+1 & : X[i] = Y[j] \\ max(s[i-1,j], s[i,j-1]) & : X[i] \neq Y[j] \end{cases}$$

- Decompose the problems into solutions of subproblems
- Try all possible last moves and find the optimal solution
- Memoize the solutions using an array



A high-level approach to design DP algorithms

- DP is not an algorithm, but an algorithm design idea (methodology)

- Ideas in the lectures
- Subproblems: a prefix of the problem
- Decisions: what is the possible “last move” (second last element)?
- Boundary: what is the end of the recursion?

Things to learn for dynamic programming

- Understand why dynamic programming makes an algorithm faster
- Understand the structure of dynamic programming
- Understand the classic DP algorithms and their variants
- Understand how to in general design DP algorithms
- Understand how to accelerate DP algorithms and apply to real-world applications

Key Points in Dynamic Programming (DP)

- **How to decide the states (subproblems)**
  - Captures the “key feature” in the problem
  - Consider the similarities to the classic problems covered in class

- **How to correctly write down the recurrences**
  - Decide the correct decision to make to compute the states (last element of a prefix)
  - No missing cases
  - Be specific about the boundary cases and what the answer is

- **How to correctly program a DP recurrences**
  - Top-down (memoization)
  - Bottom-up (using nested for-loops)

For the midterm exam

- Materials in this lecture will be covered
- Prepare your cheetsheet carefully
- Come to WCH 205/206 before 11am

Do not cheat!