ALGORITHMS PART III

Sid Chi-Kin Chau

[Lecture 7]

What is Dynamic Programming

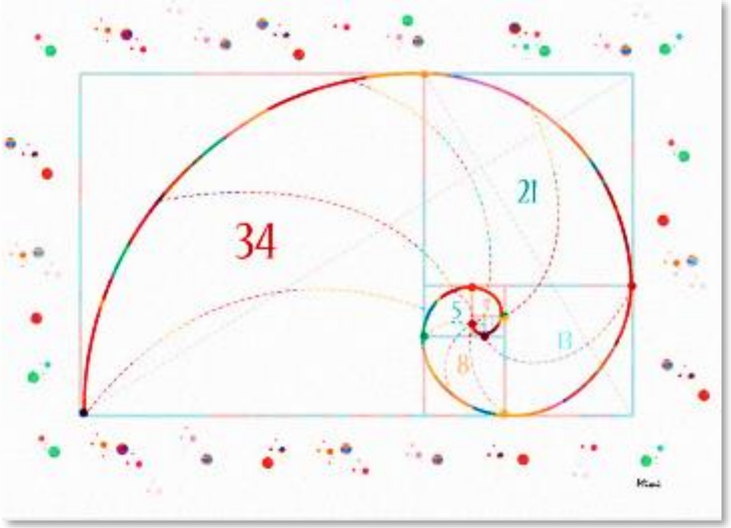
- Dynamic programming (DP) is a general technique
  - Powerful algorithmic design technique using recursion and memorization
  - A class of seemingly exponential-time problems may have a polynomial-time solution via DP
  - Particularly for optimization (min/max) problems (e.g., shortest paths)
  - “Programming” is not related to particular programming language
- Dynamic programming does not always guarantee efficiency
  - DP ≈ “controlled brute force”
- When designed properly, dynamic programming can be efficient
  - Memorization stores the results of expensive calls in the cache
  - DP ≈ recursion + memorization

Goals of This Lecture

- Apply dynamic programming to solve the following problems
  - Fibonacci numbers
  - Shortest paths
  - Minimum edit distance
  - Tetris
- See how efficiency of dynamic programming in each of these problems
  - From polynomial-time to exponential-time solutions

Fibonacci Numbers

- Fibonacci numbers = (1, 1, 2, 3, 5, 8, 13, 21, 34, …)
- Fibonacci numbers are often observed in nature
  - Shell, plant
- They also give elegant patterns
  - Architecture, art
- Recurrence equation of Fibonacci numbers:
  - $F_n = F_{n-1} + F_{n-2}$
  - $F_2 = F_1 = 1$



Fibonacci Numbers

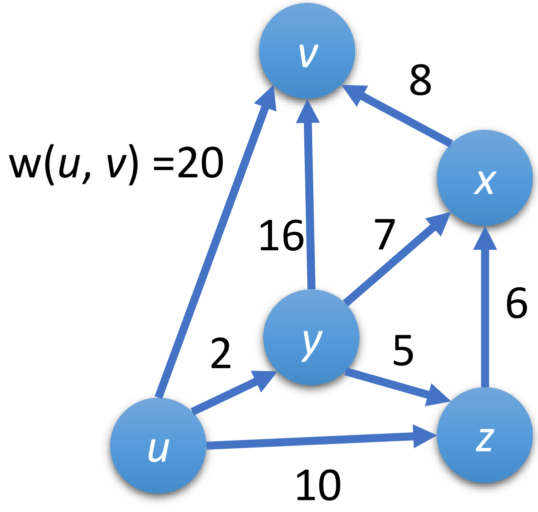
- Recurrence equation of Fibonacci numbers:
  - $F_n = F_{n-1} + F_{n-2}$
  - $F_2 = F_1 = 1$
  - Running time of direct computation:
    - $T(n) = T(n-1) + T(n-2) + O(1)$
      $\geq 2\,T(n-2) + O(1) \geq O(2^{n/2})$
- There are redundant computations
  - Can be improved by memorization in dynamic programming
  - Then running time is $T(n) = O(n)$ because of only n non-memorized calls

```
Fib[n]
// By direct computation
If n ≤ 2 Then
    Return f ← 1
Else
    Return f ← Fib[n-1] + Fib[n-2]
```

```
Fib[n]
// By dynamic programming
If memo[n] ≠ null Then
    Return memo[n] // memorized call
Else If n ≤ 2 Then
    f ← 1
Else // non-memorized call
    f ← Fib[n-1] + Fib[n-2]
    memo[n] ← f
    Return f
```

Shortest Paths

- Given a directed graph (V, E)
  - Each directed edge (u, v) has a weight $w(u, v)$
  - Let $\delta(s, v)$ be the total weight of the shortest path from node s to node v in E
- Shortest paths are useful for many applications
  - Telematic navigation
  - Communication networks
  - Logistic and transportation
  - Planning and scheduling
- Shortest path is the first dynamic programming problem by Bellman

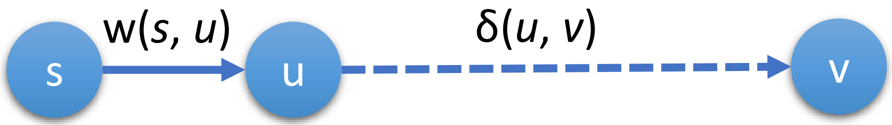


Shortest Paths

- Shortest path can be found iteratively from neighbours
  - Let SP(u, v) be the shortest path from u to v
  - Then SP(s, v) is the lowest-cost path concatenating edge (s, u) and SP(u, v)
  - SP(s, v) can be found based on its total weight that satisfies

$$\delta(s, v) = \min\{w(s, u) + \delta(u, v) \mid (s, u) \in E\}$$

- Find shortest paths by memorized DP
  - $\delta_0(s, v) \leftarrow \infty$ for $s \neq v$ (base case)
  - $\delta_k(s, v) \leftarrow \min\{w(s, u) + \delta_{k-1}(u, v) \mid (s, u) \in E\}$

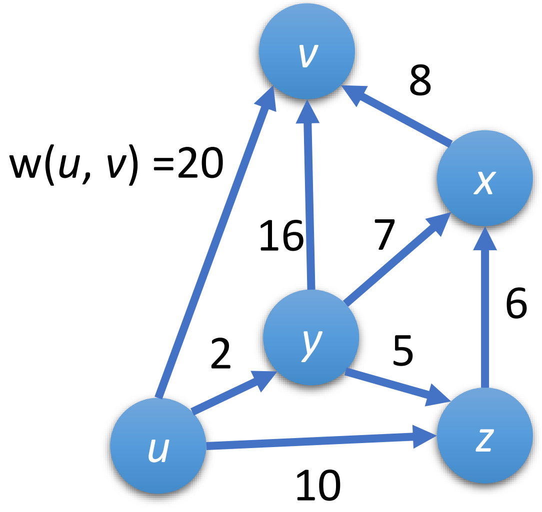


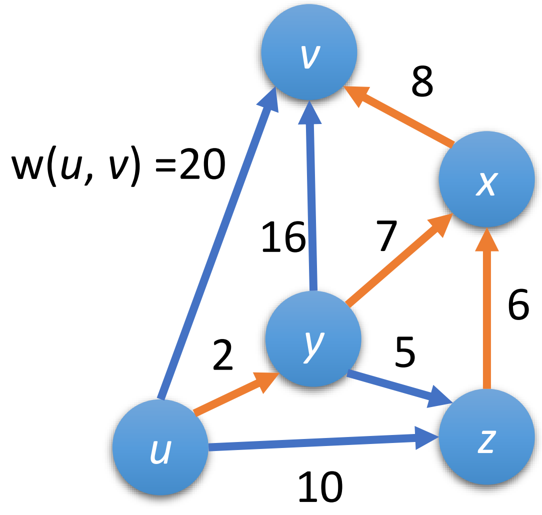

```
ShortestPath[V,E,v]
// Shortest path by dynamic program
δ_0(v,v) ← 0     // Initialization
δ_0(s,v) ← ∞ for all s ≠ v

// memorized DP in multiple iterations
For k = 1 to |V|
      δ_k(s,v) ← min{w(s,u)+δ_k-1(u,v)|(s,u)∈E}
      for all s ≠ v

// Return all shortest paths {SP(s, v)}
Return {SP(s,v)|δ_k(s,v)=w(u,v)+δ_k(u,v), s∈V}
```

Shortest Paths: Example

k	$\delta_k(v, v)$	$\delta_k(x, v)$	$\delta_k(y, v)$	$\delta_k(z, v)$	$\delta_k(u, v)$
0	0	∞	∞	∞	∞
1	0	8	16	∞	20
2	0	8	~~16~~ (15)	14	~~20~~ (18)
3	0	8	15 (~~19~~)	14	18 (~~24~~)
4	0	8	15	14	~~18~~ (17)
5	0	8	15	14	17

(new path)



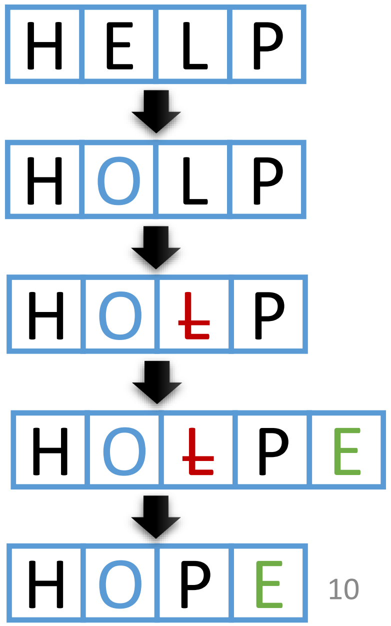


Steps for Dynamic Programming

0. Original problem
   - e.g., find SP(s, v) or compute $\delta(s, v)$
1. Define subproblems
   - e.g., $\delta(u, v)$ where u is a neighbour of s
2. Guess (part of solution)
   - e.g., $w(s, u) + \delta(u, v)$
3. Relate subproblem solutions
   - e.g., $\delta(s, v) = \min\{w(s, u) + \delta(u, v) \mid (s, u) \in E\}$
4. Recurse + memorize
   - Build DP table bottom-up check subproblems acyclic/topological order
   - e.g., $\delta_k(s, v) \leftarrow \min\{w(s, u) + \delta_{k-1}(u, v) \mid (u, v) \in E\}$

Edit Distance

- Used for DNA comparison, plagiarism detection, etc.
- Given two strings x and y, what is the cheapest possible sequence of character edits to transform x into y?
    - Character edits:
        - Insert a new character c into x
        - Delete a character c from x
        - Replace a character c in x by c': c → c'
- Cost of edit depends only on characters c, c'
    - For example in DNA, common mutation C → G has low cost
- Edit distance is the total cost of a sequence of edits



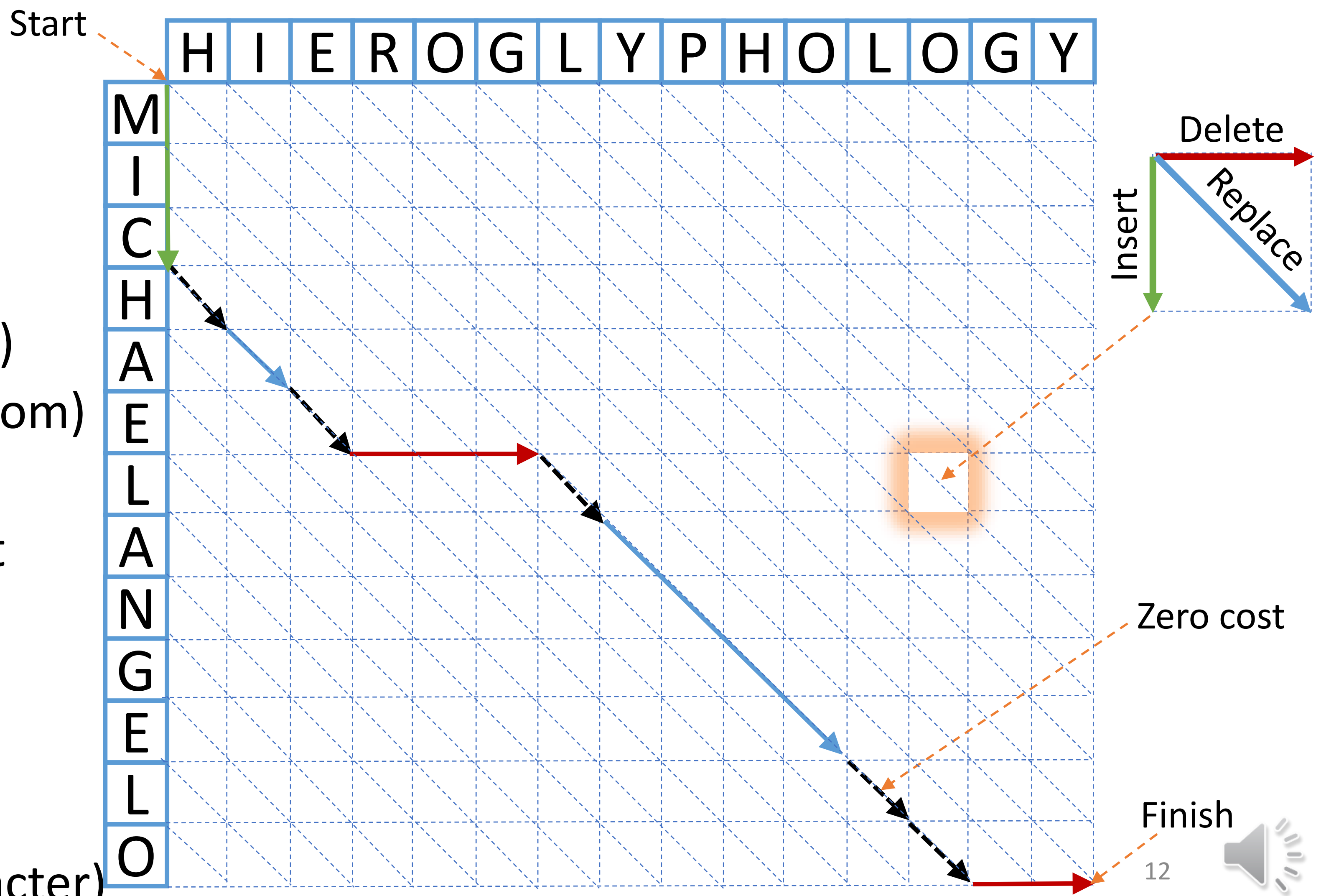
Edit Distance: Example

- Edit distance = Total cost of edits
  - No cost for the same character in both strings
- If insertion and deletion cost 0.5 and replacement costs 1, the minimum edit distance equivalent to finding the longest common subsequence
  - Subsequence is sequential but not necessarily contiguous
  - Example
    - H I E R O G L Y P H O L O G Y vs. M I C H A E L A N G E L O
    - The longest common subsequence is HELLO
    - Edit distance

      = insertion cost (1.5) + deletion cost (2.5) + replace cost (5) = 9

Edit Distance: Dynamic Programming

- Finding the minimum edit distance is equivalent to finding shortest path
- Construct a directed graph
  - From start (leftmost top)
  - To finish (rightmost bottom)
  - Deletion cost
    = horizontal edge cost
  - Insertion cost
    = vertical edge cost
  - Replacement cost
    = diagonal edge cost
    (zero cost for same character)

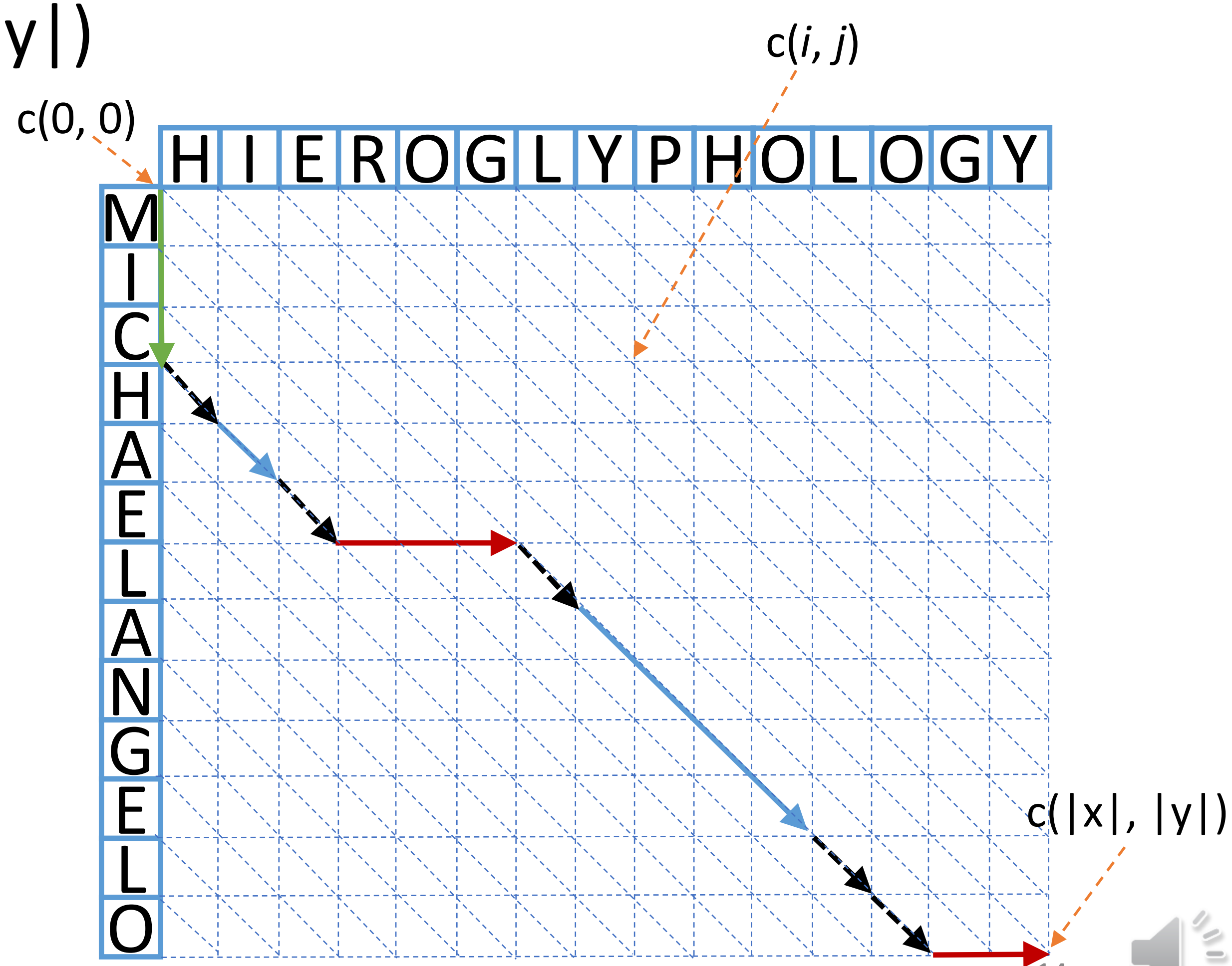


Edit Distance: Dynamic Programming

- More general problems for multiple strings/sequences
  - Suffix/prefix/substring subproblems
  - Multiply state spaces
  - Still polynomial for a constant number of strings
- Given strings x and y, let x[*i*] and y[*j*] be the *i*-th and *j*-th characters of x and y, respectively
- Guess whether, to turn x into y, following one of the 3 choices:
  - Deleting x[*i*] incurs a cost $Cost_{del}(x[i])$
  - Inserting y[*j*] incurs a cost $Cost_{ins}(y[j])$
  - Replacing x[*i*] by y[*j*] incurs a cost $Cost_{rep}(x[i],y[j])$
    - $Cost_{rep}(x[i],y[j]) = 0$, when $x[i] = y[j]$

Edit Distance

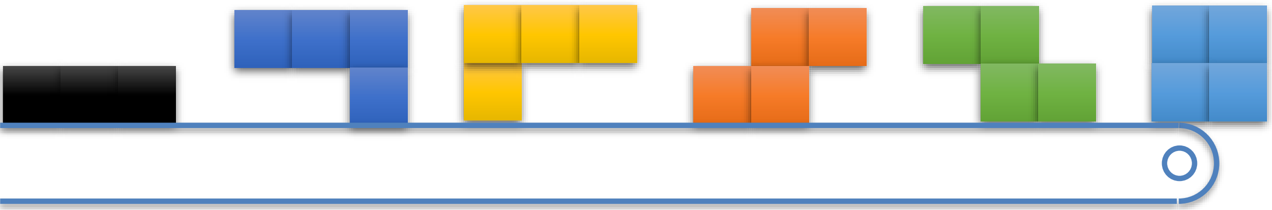
- $c(i, j)$ is min cost from (i, j) to $(|x|,|y|)$
- Recurrence: $c(i, j)$ = minimum of:
  - $Cost_{del}(x[i]) + c(i+1, j)$ if $i < |x|$,
  - $Cost_{ins}(y[j]) + c(i, j+1)$ if $j < |y|$,
  - $Cost_{rep}(x[i],y[j]) + c(i+1, j+1)$
    if $i < |x|$ and $j < |y|$
- Set $c(|x|,|y|) = 0$
- Directed graph of the table:
  - Top to bottom OR right to left
  - Linear space of states of table size
  - Total running time = $O(|x| \cdot |y|)$

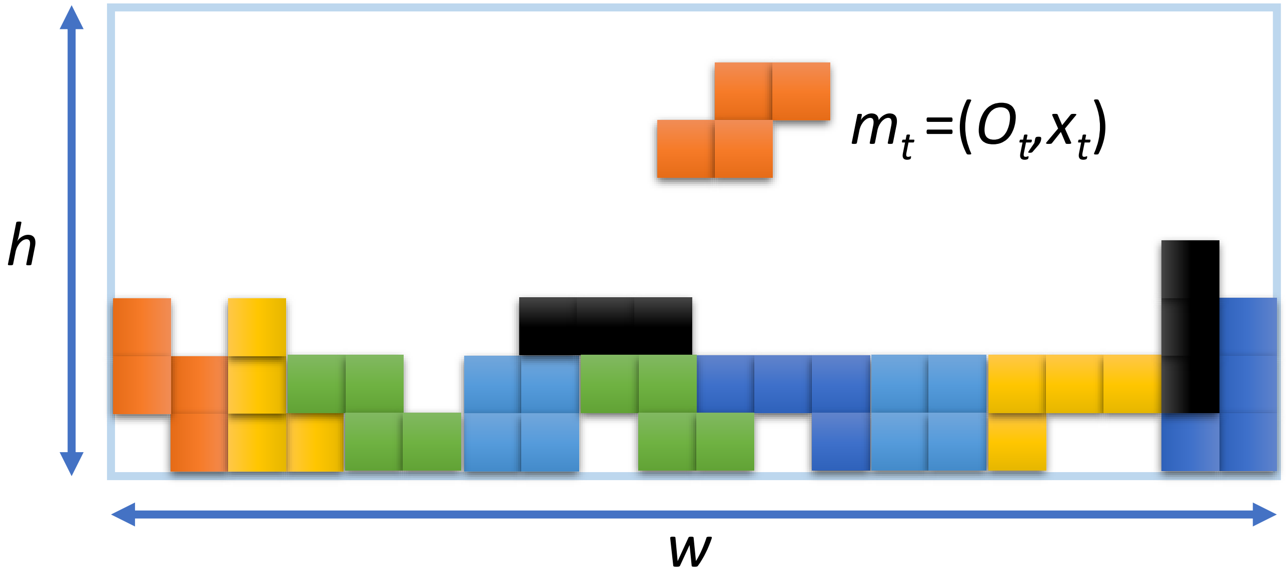


Tetris

- There is an empty board of small width w
- Given a sequence of n Tetris pieces
- For the t-th piece, decide its move m_t
  - Orientation O_t (rotate by 90°, 180°, 270°)
  - X-coordinate x_t (in $\{1,\ldots, w\}$)
- Then must drop piece till it hits something
- Full rows do not clear
- Goal:
  - To survive, namely, stay within height h

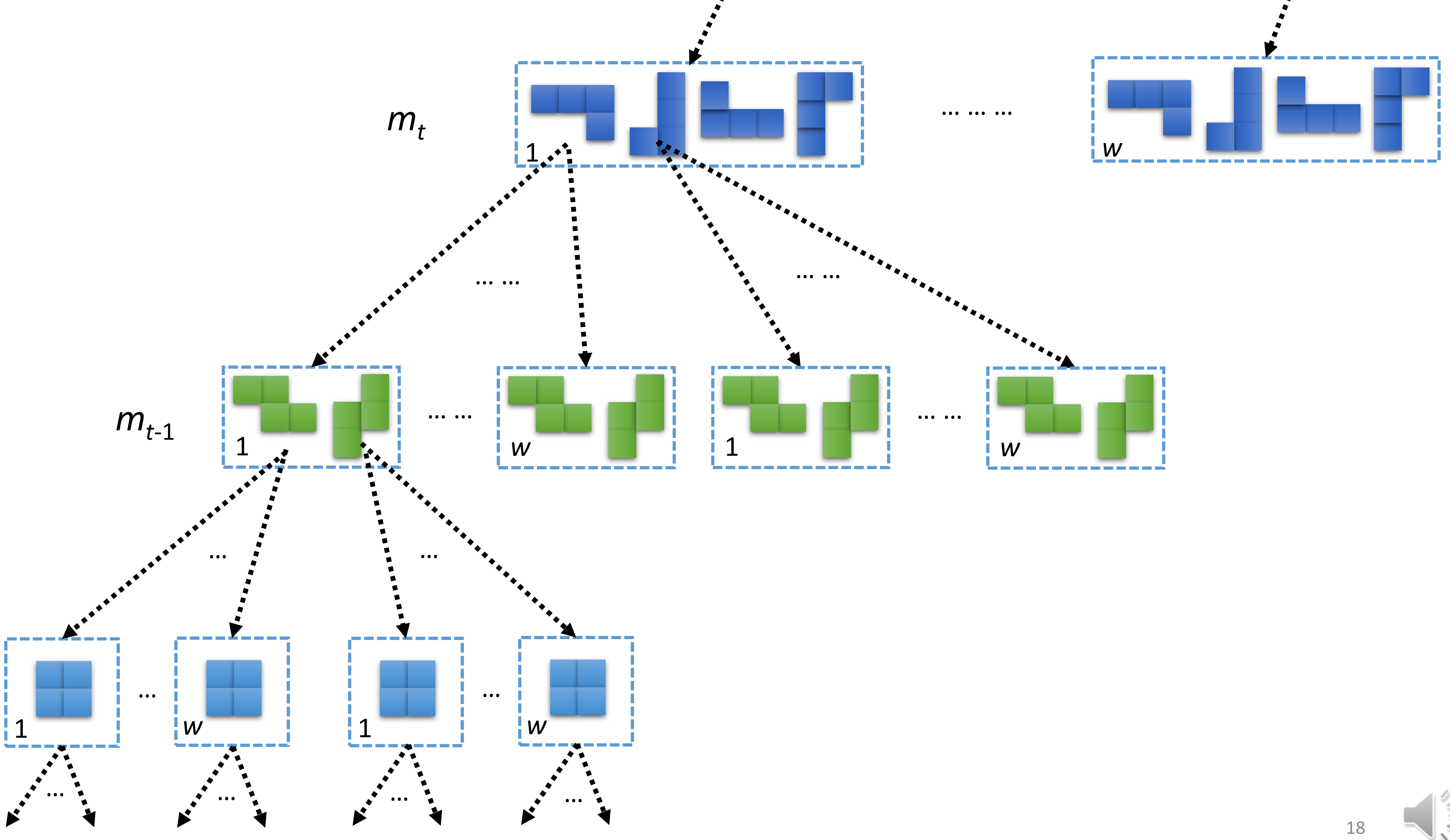
Tetris





Tetris

1. Subproblem: Survive in each column i:
   - The column occupancy heights $\mathbf{h}^t = (h^t_1, h^t_2, \dots, h^t_w)$ at time t
   - Define Height[t] to be the min height by adding the t-th piece
2. Recurrence:
   - At time t, the t-th piece is dropped
   - Height[t] = min(Height[t-1] + cost of a valid move m_t of the t-th piece in $\mathbf{h}^t$)
   - The number of moves of the t-th piece = $O(4^w)$
3. Construct a directed graph
   - Connect each valid move m_t of the t-th piece to every valid move m_{t-1} of the (t-1)-th piece
   - The cost of each move is the additional height incurred by the t-th piece


m_t
1
… … …
w
$m_{t\text{-}1}$
1
… …
w
1
… …
w
1
w
1
w

Summary

- Dynamic programming (DP) is a general technique
  - memorization stores the results of expensive calls in the cache
  - DP ≈ recursion + memorization
- Problems:
  - Fibonacci numbers
  - Shortest paths
  - Edit distance
  - Tetris

Reference

- Visualizations
  - https://www.cs.usfca.edu/~galles/visualization/DPFib.html
  - https://www.cs.usfca.edu/~galles/visualization/Dijkstra.html
  - https://www.cs.usfca.edu/~galles/visualization/DPLCS.html