香港科技大学（广州）
THE HONG KONG
UNIVERSITY OF SCIENCE AND
TECHNOLOGY (GUANGZHOU)

Design and Analysis of Algorithms

Jing Tang | DSAA 2043 Fall 2024

Shortest Paths

Paths in graphs.

A ***shortest path*** from u to v is a path of minimum weight from u to v.
The ***shortest-path weight*** from u to v is defined as:

$$\delta(u, v) = \min\{w(p) : p \text{ is a path from } u \text{ to } v\}.$$

Note: $\delta(u, v) = \infty$ if no path from u to v exists.

Dijkstra's algorithm

$d[s] \leftarrow 0$
for each $v \in V - \{s\}$
    do $d[v] \leftarrow \infty$
$S \leftarrow \varnothing$
$Q \leftarrow V$ ⊳ Q is a priority queue maintaining $V - S$, keyed on $d[v]$

Let us review the process of Dijkstra with an example

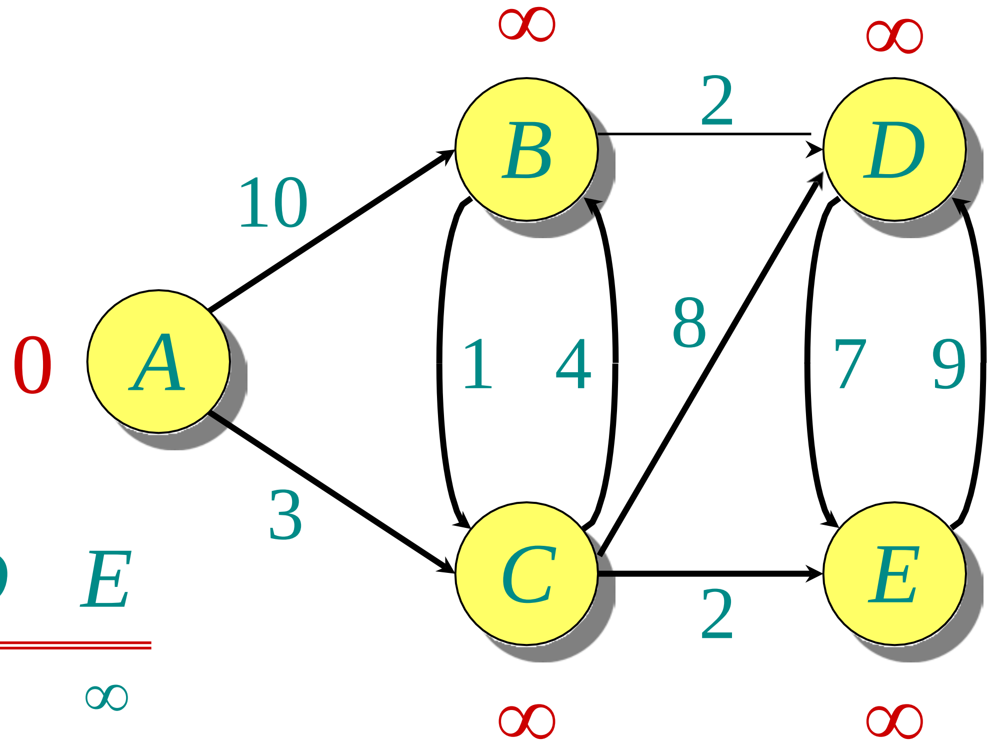
    $S \leftarrow S \cup \{u\}$
    for each $v \in Adj[u]$
        do if $d[v] > d[u] + w(u, v)$
            then $d[v] \leftarrow d[u] + w(u, v)$

relaxation step

Implicit DECREASE-KEY

Example of Dijkstra's algorithm

Initialize:

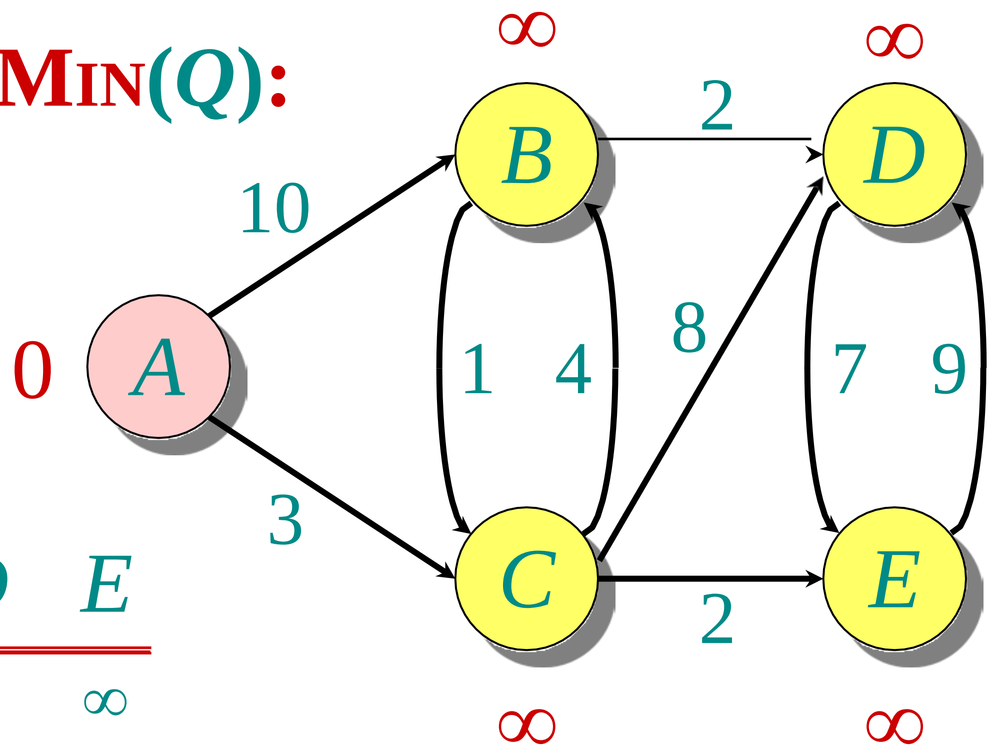


Q:	A	B	C	D	E
	0	∞	∞	∞	∞

S: {}

Example of Dijkstra's algorithm

"A" ← **EXTRACT-MIN(Q):**

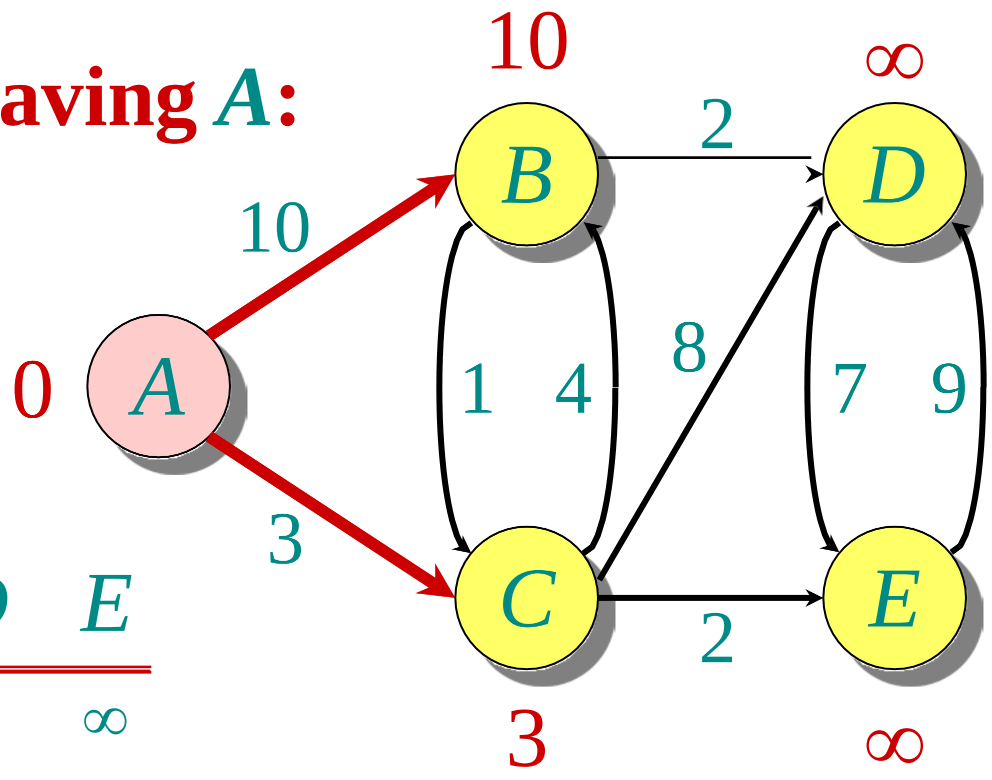


Q:	A	B	C	D	E
	0	∞	∞	∞	∞

S: { A }

Example of Dijkstra's algorithm

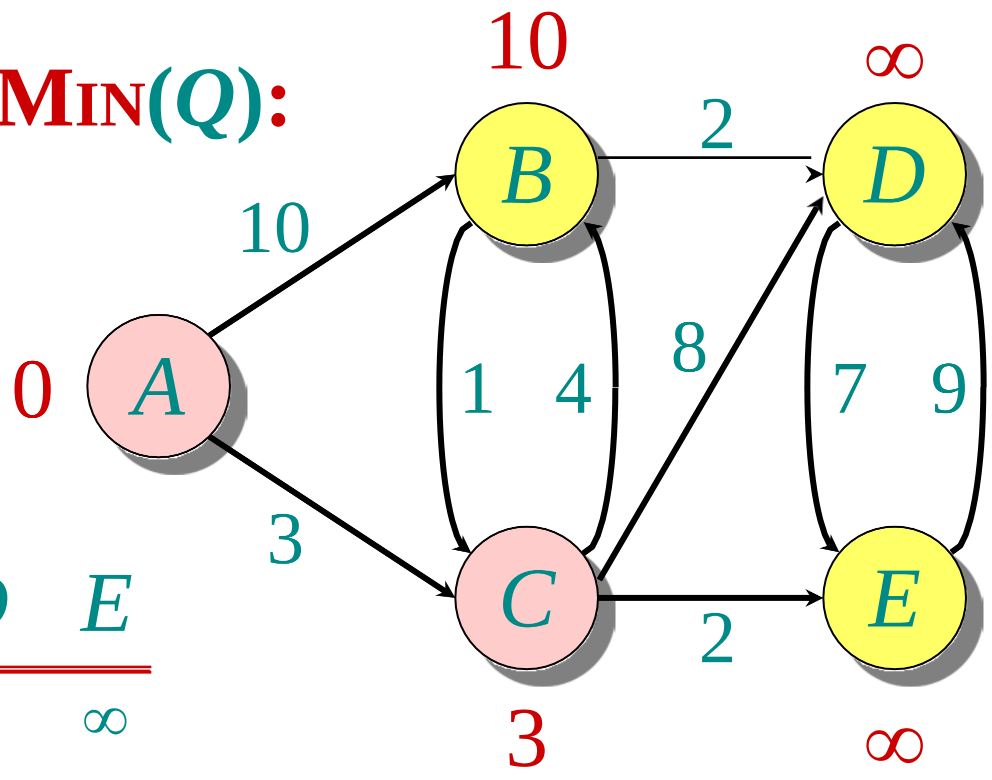
Relax all edges leaving A:



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A }

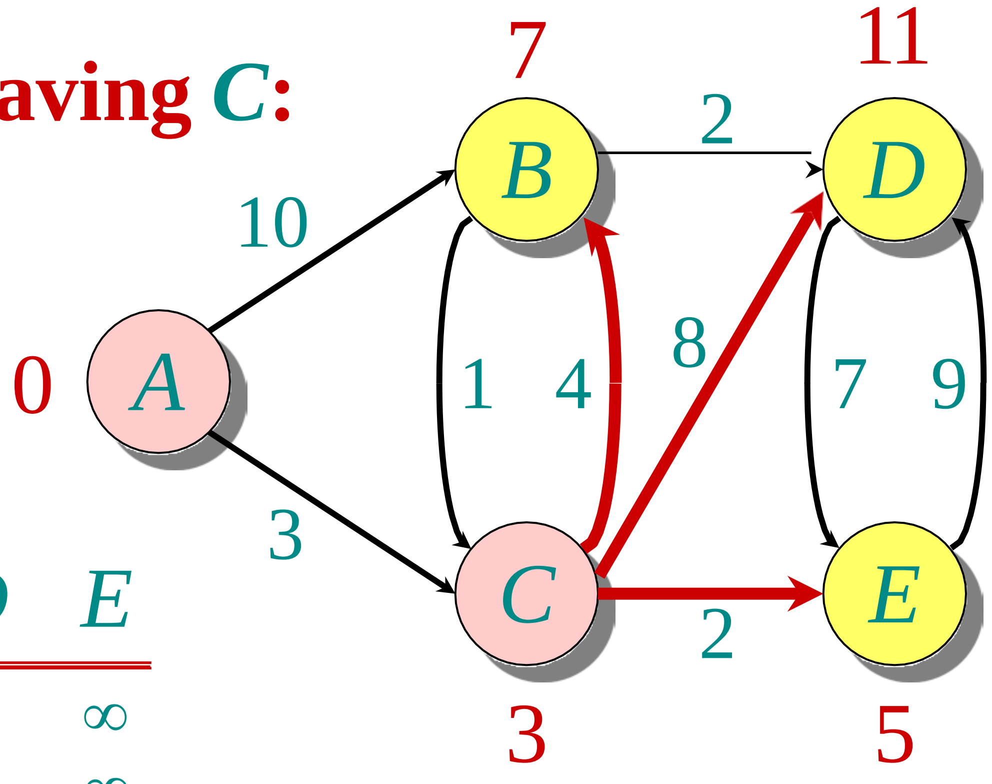
“*C*” ← EXTRACT-MIN(Q):



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A, C }

Relax all edges leaving *C*:

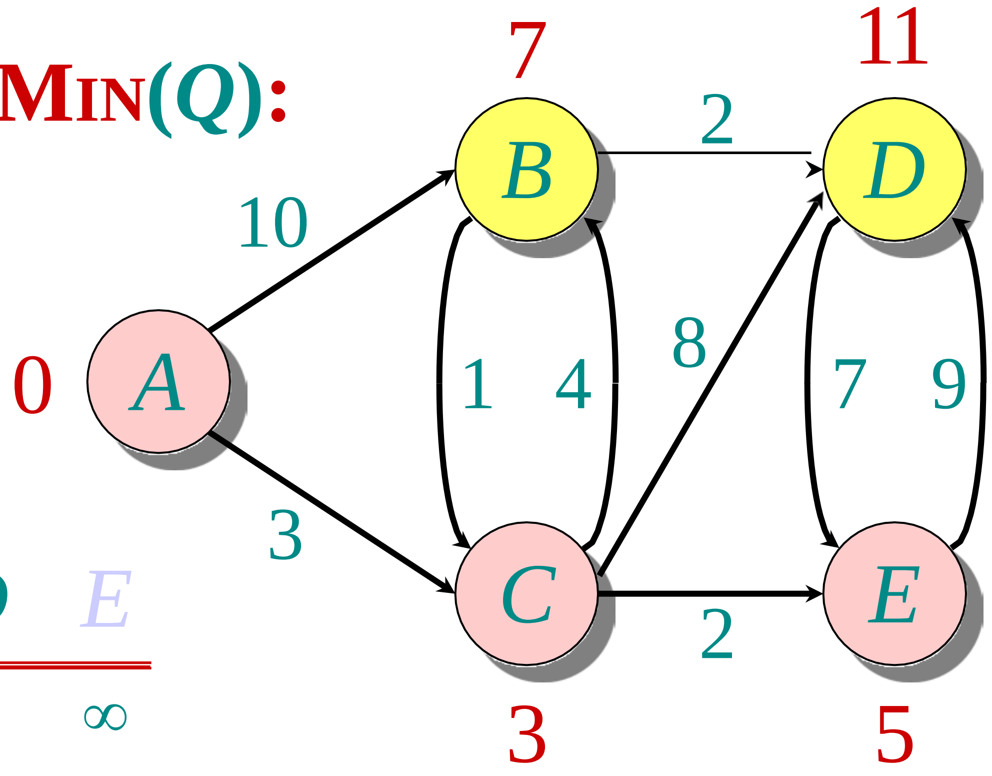


Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { *A*, *C* }

Example of Dijkstra's algorithm

"E" ← EXTRACT-MIN(Q):

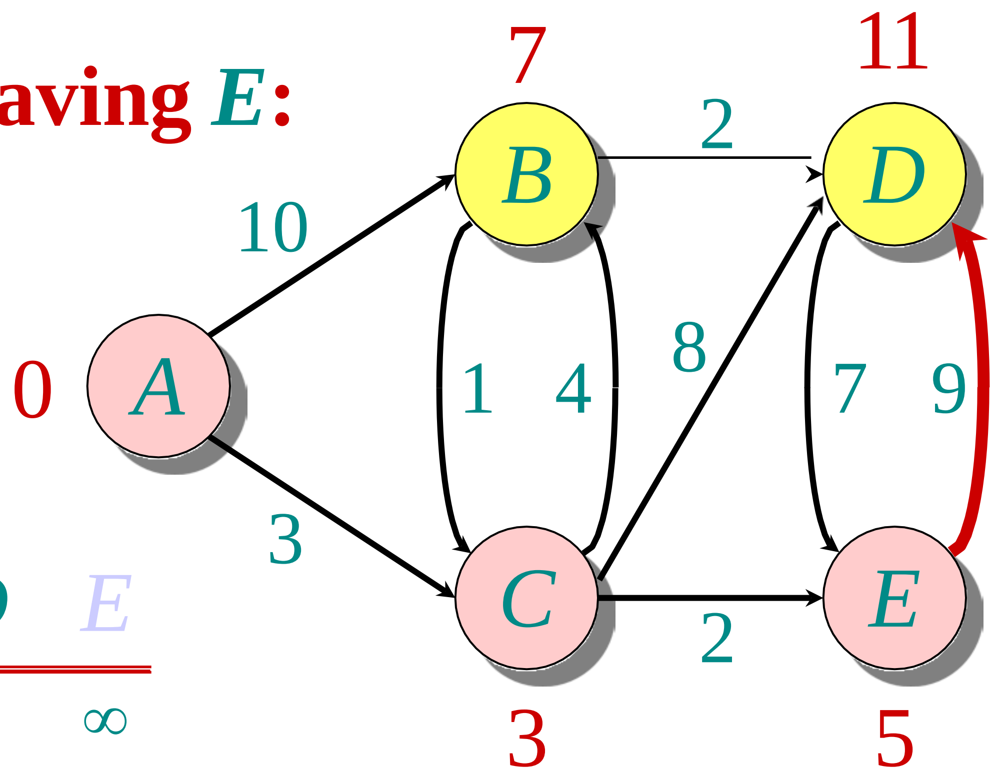


Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { A, C, E }

Example of Dijkstra’s algorithm

Relax all edges leaving *E*:

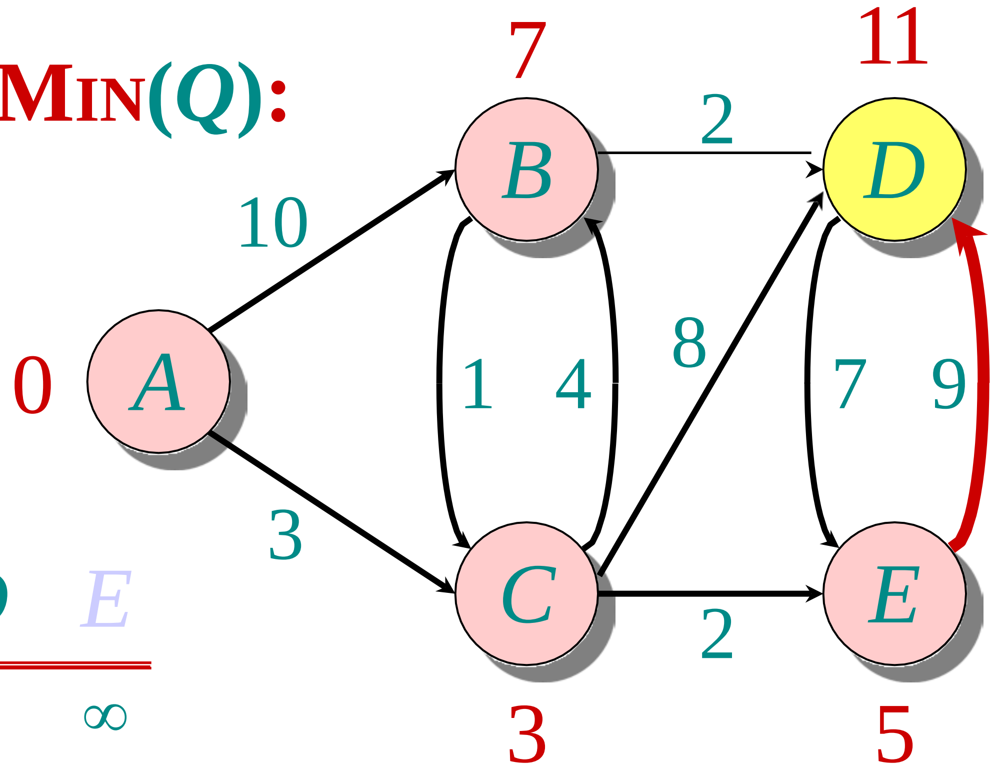


Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { *A*, *C*, *E* }

Example of Dijkstra's algorithm

“B” ← EXTRACT-MIN(Q):

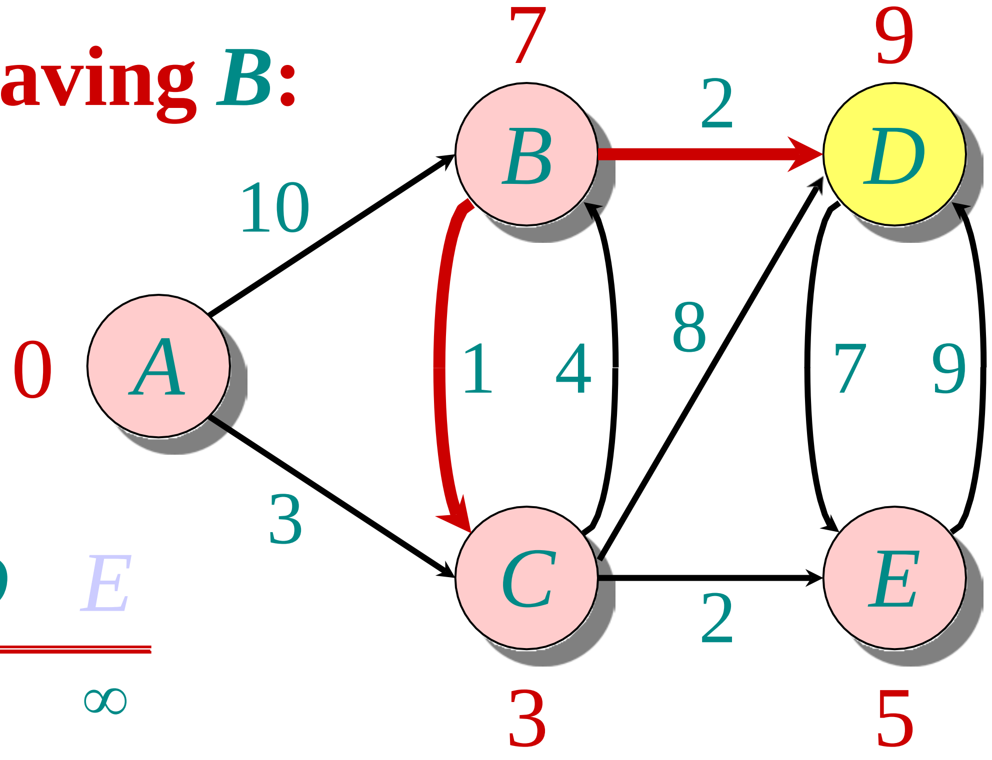


Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E, B }

Example of Dijkstra's algorithm

Relax all edges leaving *B*:

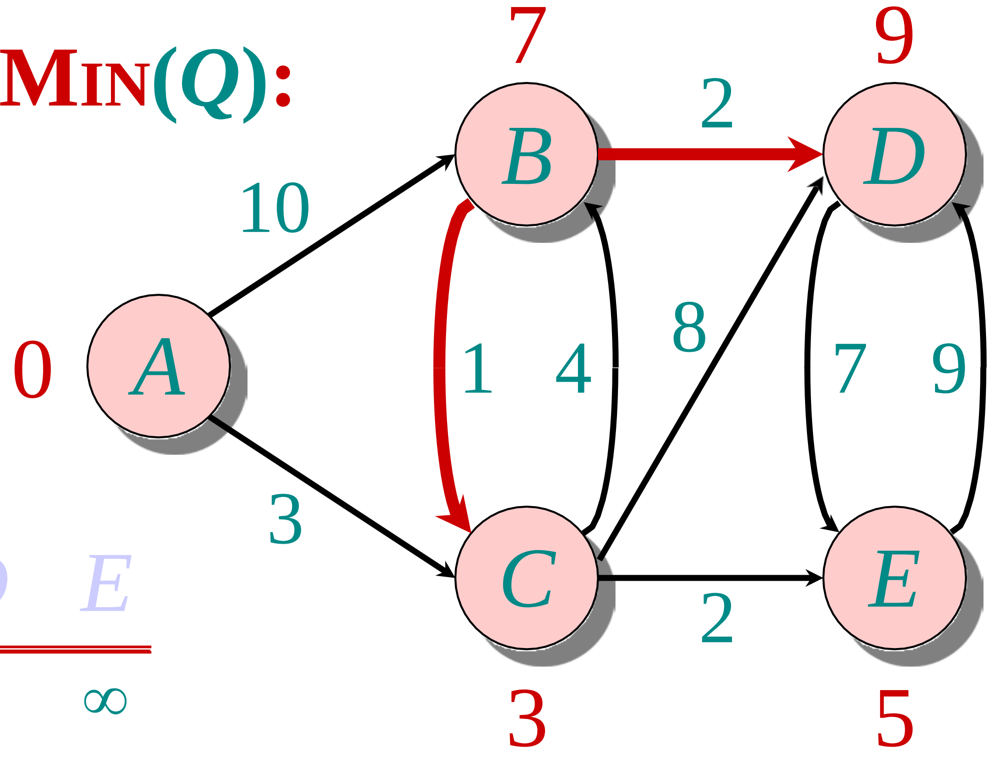


Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: { A, C, E, B }

Example of Dijkstra’s algorithm

“D” ← EXTRACT-MIN(Q):



Q:	A	B	C	D	E
	$\boxed{0}$	∞	∞	∞	∞
		10	$\boxed{3}$	∞	∞
		7		11	$\boxed{5}$
		$\boxed{7}$		11	
				$\boxed{9}$	

S: { A, C, E, B, D }

Analysis of Dijkstra

$$\text{Time} = \Theta(|V|)\cdot T_{\text{EXTRACT-MIN}} + \Theta(|E|)\cdot T_{\text{DECREASE-KEY}}$$

Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total
array	$O(\|V\|)$	$O(1)$	$O(\|V\|^2)$
binary heap	$O(\lg\|V\|)$	$O(\lg\|V\|)$	$O(\|E\|\lg\|V\|)$
Fibonacci heap	$O(\lg\|V\|)$ amortized	$O(1)$ amortized	$O(\|E\| + \|V\| \lg \|V\|)$ worst case

The heap-optimized Dijkstra algorithm is far better than the naive Dijkstra algorithm in most cases

However, for dense graph ($\mathrm{m} \approx n^2$), naïve Dijkstra algorithm works better.

Bellman-Ford algorithm

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

- (-) Slower than Dijkstra's algorithm

Dijkstra can't handle negative edges because it's based on greedy, and the "current best" may not be the "final best" with negative edges.

- (+) Can handle negative edge weights.
  - Can be useful if you want to say that some edges are actively good to take, rather than costly.
  - Can be useful as a building block in other algorithms.

Basic idea:

Instead of picking the u with the smallest d[u] to update, just update all of the u's simultaneously.

Bellman-Ford algorithm

香港科技大学(广州) THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

Bellman-Ford(G,s):

- d[v] = ∞ for all v in V
- d[s] = 0
- **For** i=0,…,|V|-1:
  - **For** u in V:
    - **For** v in u.neighbors:
      - d[v] ← min(d[v], d[u] + edgeWeight(u,v))

Instead of picking u cleverly, just update for all of the u's.

Compare to Dijkstra:

- **While** there are **not-sure** nodes:
  - Pick the **not-sure** node u with the smallest estimate **d[u].**
  - **For** v in u.neighbors:
    - d[v] ← min(d[v], d[u] + edgeWeight(u,v))
  - Mark u as **sure**.

Bellman-Ford algorithm

- We are actually going to change this to be less smart.
- Keep n arrays: $d^{(0)}, d^{(1)}, \ldots, d^{(n-1)}$

Bellman-Ford*(G,s):

- $d^{(i)}[v] = \infty$ for all v in V, for all i=0,…,|V|-1
- $d^{(0)}[s] = 0$
- **For** i=0,…,|V|-2:
  - **For** u in V:
    - **For** v in u.neighbors:
      - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v]\ ,\ d^{(i+1)}[v],\ d^{(i)}[u] + \text{edgeWeight}(u,v))$
- Then dist(s,v) = $d^{(n-1)}[v]$

Slightly different than the original Bellman-Ford algorithm, but the analysis is basically the same.

Pros and cons of Bellman-Ford

- Running time: $O(|V||E|)$ running time
  - For each of |V| steps we update m edges
  - Slower than Dijkstra
- However, it's also more flexible in a few ways.
  - Can handle negative edges
  - If we constantly do these iterations, any changes in the network will eventually propagate through.

How Bellman-Ford deals with negative cycles

- If there are no negative cycles:
  - Everything works as it should.
  - The algorithm stabilizes after |V|-1 rounds.
  - Note: Negative ***edges*** are okay!!
- If there are negative cycles:
  - Not everything works as it should…
    - it couldn't possibly work, since shortest paths aren't well-defined if there are negative cycles.
  - The d[v] values will keep changing.
- Solution:
  - Go one round more and see if things change.
    - If so, return NEGATIVE CYCLE ☹

Shortest Path

Single-source shortest paths

- Nonnegative edge weights
  - Dijkstra's algorithm: $O(|E| + |V| \lg|V|)$
- General
  - Bellman-Ford algorithm: $O(|V||E|)$

All-pairs shortest paths

- Nonnegative edge weights
  - Dijkstra's algorithm $|V|$ times: $O(|V||E| + |V|^2 \lg|V|)$
- General
  - Floyd-Warshall algorithms: $\Theta(|V|^3)$.

Dynamic Programming

Dynamic Programming

- Dynamic Programming is an algorithm design technique for *optimization problems:* often minimizing or maximizing.

- Like divide and conquer, DP solves problems by combining solutions to sub-problems.

- Unlike divide and conquer, sub-problems are not independent.
  - Sub-problems may share sub-sub-problems.

Two Ways to Think and Implement DP

- Top down:
- Think of it like a recursive algorithm.
- To solve the big problem:
  - Recurse to solve smaller problems
    - Those recurse to solve smaller problems
      - etc..
- The difference from divide and conquer:
  - Keep track of what small problems you've already solved to prevent re-solving the same problem twice.
  - Aka, "**memoization**"

- Bottom up:
- For Fibonacci:
- Solve the small problems first
  - fill in F[0],F[1]
- Then bigger problems
- …
- Then bigger problems
  - fill in F[n-1]
- Then finally solve the real problem.
  - fill in F[n]

The Process of Applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the length of the longest common subsequence.
- **Step 3:** Use dynamic programming to find the length of the longest co

Let us review the LCS problem as an example.

- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual LCS.
- **Step 5:** If needed, code this up like a reasonable person.

Longest Common Subsequence

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

- Subsequence:
  - BDFH is a **subsequence** of ABCDEFGH
- If X and Y are sequences, a **common subsequence** is a sequence which is a subsequence of both.
  - BDFH is a **common subsequence** of ABCDEFGH and of ABDFGHI
- A **longest common subsequence**…
  - …is a common subsequence that is longest.
  - The **longest common subsequence** of ABCDEFGH and ABDFGHI is ABDFGH.

Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the length of the longest common subsequence.
- **Step 3:** Use dynamic programming to find the length of the longest common subsequence.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual LCS.
- **Step 5:** If needed, code this up like a reasonable person.

Step 1: Optimal substructure

Prefixes:

X	A	C	G	G	T		
Y	A	C	G	C	T	T	A

Notation: denote this prefix **ACGC** by Y_4

- Our sub-problems will be finding LCS's of prefixes to X and Y.
- Let C[i,j] = length_of_LCS(X_i, Y_j)

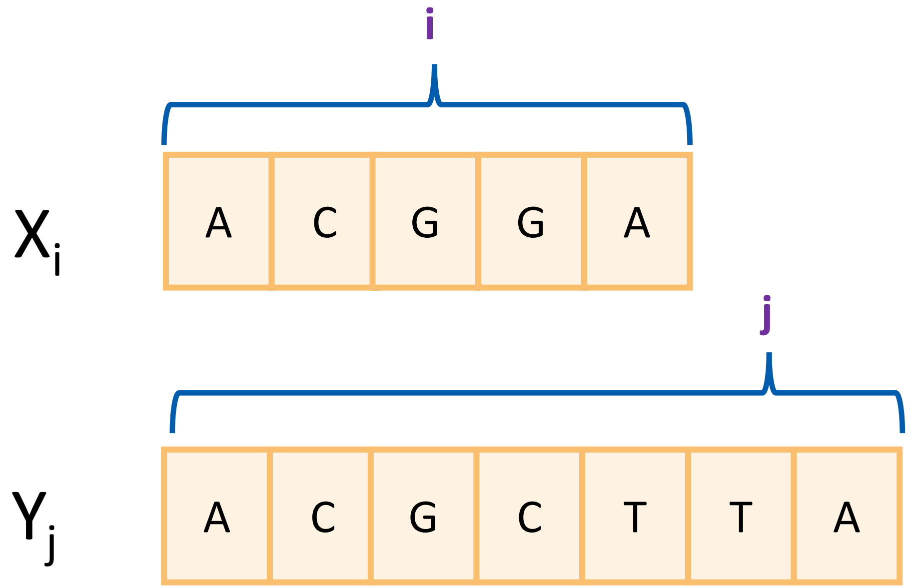
Examples: C[2,3] = 2
C[4,4] = 3

Recipe for applying Dynamic Programming

- Step 1: Identify optimal substructure.
- Step 2: Find a recursive formulation for the length of the longest common subsequence.
- Step 3: Use dynamic programming to find the length of the longest common subsequence.
- Step 4: If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual LCS.
- Step 5: If needed, code this up like a reasonable person.

Goal

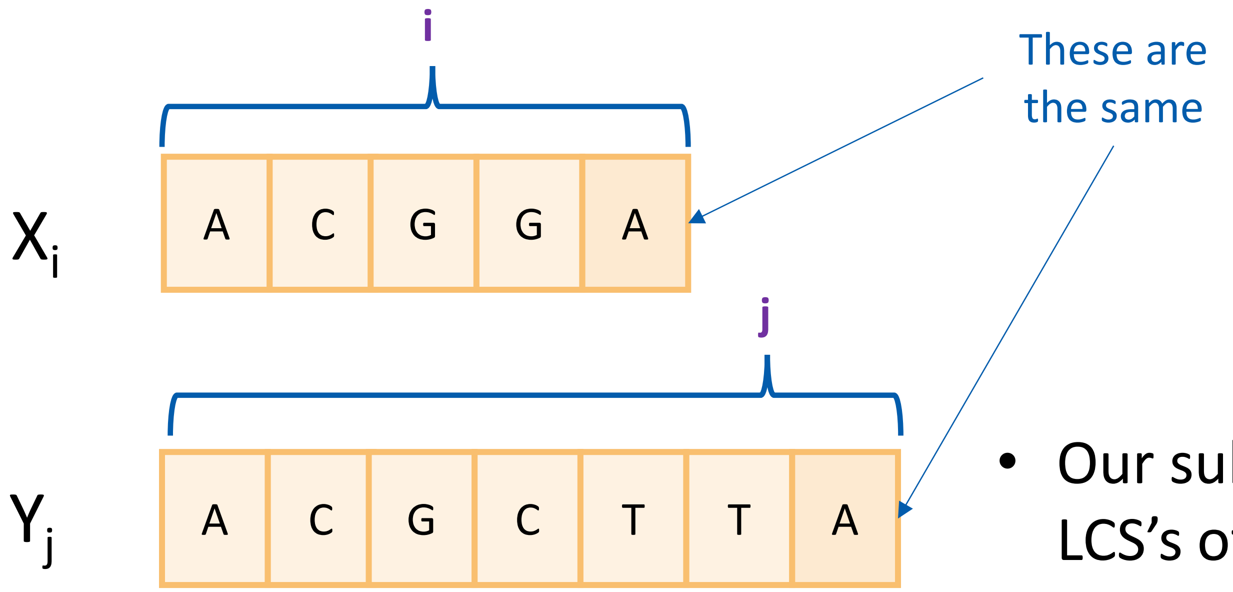
- Write C[i,j] in terms of the solutions to smaller sub-problems



$C[i,j] = \text{length\_of\_LCS}(X_i, Y_j)$

Two cases

Case 1: X[i] = Y[j]

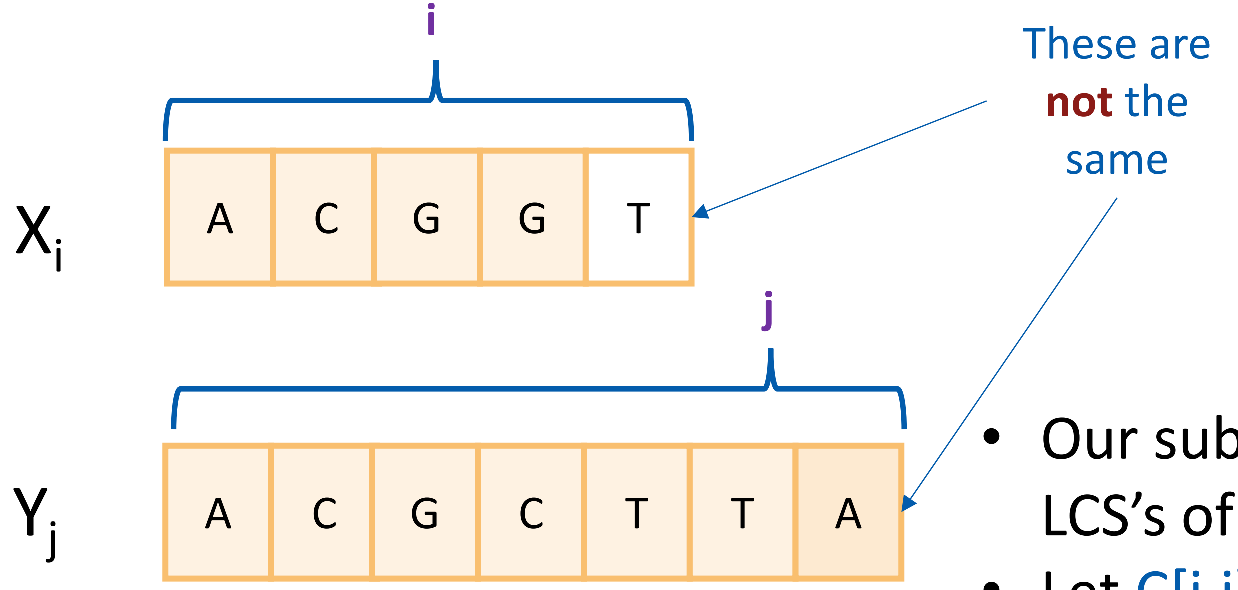


- Our sub-problems will be finding LCS's of prefixes to X and Y.
- Let C[i,j] = length_of_LCS(X_i, Y_j)

- Then C[i,j] = 1 + C[i-1,j-1].
  - because LCS(X_i,Y_j) = LCS(X_{i-1},Y_{j-1}) followed by $\boxed{A}$

Two cases

Case 2: X[i] != Y[j]



- Our sub-problems will be finding LCS's of prefixes to X and Y.
- Let C[i,j] = length_of_LCS(X_i, Y_j)

- Then C[i,j] = max{ C[i-1,j], C[i,j-1] }.
  - either LCS(X_i,Y_j) = LCS(X_{i-1},Y_j) and $\boxed{T}$ is not involved,
  - or LCS(X_i,Y_j) = LCS(X_i,Y_{j-1}) and $\boxed{A}$ is not involved,
  - (maybe both are not involved, that's covered by the "or").

Recursive formulation of the optimal solution

X_0 | |

Y_j | A | C | G | C | T | T | A |

Case 0

- $C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{\, C[i, j-1], C[i-1, j] \,\} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$

Case 1

X_i | A | C | G | G | A |

Y_j | A | C | G | C | T | T | A |

Case 2

X_i | A | C | G | G | T |

Y_j | A | C | G | C | T | T | A |

Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the length of the longest common subsequence.
- **Step 3:** Use dynamic programming to find the length of the longest common subsequence.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual LCS.
- **Step 5:** If needed, code this up like a reasonable person.

LCS DP

- **LCS**(X, Y):
  - C[i,0] = C[0,j] = 0 for all i = 0,…,m, j=0,…n.
  - **For** i = 1,…,m and j = 1,…,n:
    - **If** X[i] = Y[j]:
      - C[i,j] = C[i-1,j-1] + 1
    - **Else:**
      - C[i,j] = max{ C[i,j-1], C[i-1,j] }
  - Return C[m,n]

Running time: O(nm)

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\ \max\{ C[i, j-1], C[i-1, j] \} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0 \end{cases}$$

The End