CSCI 4470 Algorithms

Part VI Graph Algorithms Notes

- 20 Elementary Graph Algorithms
- 21 Minimum Spanning Trees
- 22 Single-Source Shortest Paths
- **23 All-Pairs Shortest Paths**
- 24 Maximum Flow
- 25 Matchings in Bipartite Graphs

Chapter 23: All-Pairs Shortest Paths

```
23 All-Pairs Shortest Paths
  23.1 Shortest paths and matrix multiplication
  23.2 The Floyd-Warshall algorithm
  23.3 Johnson's algorithm for sparse graphs
```

Before All-Pairs Shortest Paths (APSP)

		Sparse G	Dense G
$V \times$ Dijkstra	$V \times O(E + V\log(V))$	$O(V^2\log(V))$	$O(V^3)$
$V \times$ Bellman-Ford	$V \times O(V \cdot E)$	$O(V^3)$	$O(V^4)$
D.P.	$O(V^3\log(V))$	$O(V^3\log(V))$	$O(V^3\log(V))$

Four Approaches for All-Pairs Shortest Paths

- `Bellman-Ford` , and `Dijkstra's` can solve the SSSP problems, therefore, both can solve All-Pairs Shortest Paths (APSP) Problems as well.

1. Bellman-Ford in APSP:
Application in **All-Pairs Shortest Paths (APSP)**: Efficient but Costly (Expensive)

- Prefer `Bellman-Ford` for graphs with negative edges not negative cycles, otherwise use `Dijkstra's` .
- **Negative Cycle Detection**: `Bellman-Ford` can detect negative cycles, a crucial feature when dealing with graphs that may have negative edge weights.
- **Bellman-Ford Time Complexity in APSP**: $O(V^2 \cdot E) \approx O(V^4)$ when $E \propto V^2$.
  - In APSP using Bellman-Ford, time complexity is $O(V^2 \cdot E)$. In dense graphs where every vertex connects to all others, $E \approx V^2$. Substituting E with V^2 gives $O(V^2 \cdot V^2) = O(V^4)$.

2. Dijkstra's in APSP:

- **Dijkstra's Time Complexity in APSP**: $O\left(V^2 + VE\log(V)\right) \approx O\left(V^3\log(V)\right)$.
- **Optimizations**: Optimizations like **Fibonacci Heaps** can improve `Dijkstra's` time complexity to $O(V \cdot \log(V) + E)$.

- **Path Reconstruction**: Both can storing predecessor information can help in reconstructing the actual shortest paths after running the algorithms.
- **Algorithm Choice**: The choice between `Bellman-Ford` and `Dijkstra's` can significantly affect performance, especially in large graphs.
- **Parallel Processing**: Some versions of these algorithms can be parallelized to improve performance on modern multi-core processors.

3. D.P in APSP, D.P. and Floyd-Warshall

3.1 Dynamic Programming Solution in APSP:

- Dynamic Programming Solution, First see a less efficient, use correspondence to matrix multiply to make the approach more efficient
- **Repeated Squaring of the Adjacency Matrix**: This method involves repeatedly squaring the graph's adjacency matrix to find shortest paths. While not as widely used due to its complexity and computational requirements, it's a DP approach that applies matrix multiplication principles. The complexity can be $O(V^3 \log V)$ with optimized matrix multiplication algorithms.

3.2 Floyd-Warshall in APSP:

- Another Dynamic Programming Approach
- **Floyd-Warshall Algorithm**: The most well-known DP algorithm for APSP, particularly effective for graphs with negative weights but no negative cycles. Its time complexity is $O(V^3)$.

4. Johnson's in APSP:

- For sparse graphs
- Uses Dijkstra's algorithm
- Requires edge reweighting if any edge weights are negative
- **Johnson's Algorithm**: Although not purely a dynamic programming approach, it cleverly combines Dijkstra's algorithm and the Bellman-Ford algorithm. Johnson's algorithm reweights the edges to enable the use of Dijkstra's algorithm, which is more efficient for sparse graphs. The overall complexity is $O(V^2 \log V + VE)$.

All Pairs Shortest Paths (APSP)

Given a directed graph $G(V, E)$

- The task is to find the shortest paths between all pairs of vertices, like $\delta(v_1, v_2), \delta(v_1, v_3), \delta(v_2, v_3)$.
- Edge weights are denoted by $w : E \to \mathbb{R}$, where $|V| = n$ is the number of vertices, and vertices are labeled as $1, 2, ..., n$.

Goal: Find $d_{ij} = \delta(i, j) \forall i, j$, where d_{ij} represents the shortest path from vertex i to vertex j. This is typically represented in an $n \times n$ matrix form.

Floyd-Warshall vs. Dynamic Programming with Matrix Multiplication in APSP:

- **Floyd-Warshall**:
  - Uses iterative updates for shortest paths.
  - Does not use matrix multiplication.
  - Time complexity: $O(V^3)$.
  - Ideal for systematic shortest path refinement.
- **Dynamic Programming with Matrix Multiplication**:
  - Relies on matrix multiplication to compute shortest paths.
  - Involves raising the adjacency matrix to successive powers.
  - Time complexity varies, potentially $O(V^3 \log V)$ with optimized techniques.

    - More theoretical, less practical for large graphs.
- Both methods apply dynamic programming to APSP but differ in techniques: iterative updates vs. matrix operations.

Dynamic Programming with Matrix in APSP

Recursive Relation

- Define l_{ij}^m as the shortest path from i to j using at most m edges.
- Base Case: l_{ij}^0 is 0 if $i = j$, and ∞ otherwise.

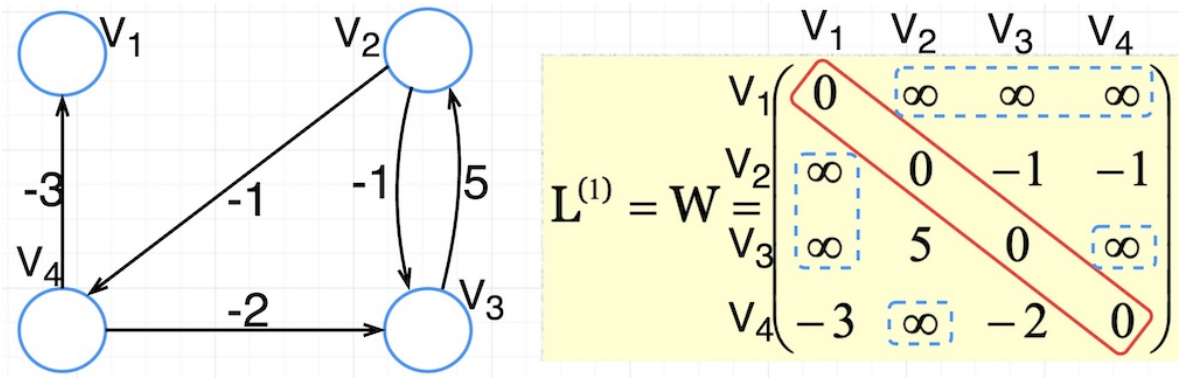
$$l_{ij}^0 = \begin{cases} 0 & \text{if } i = j, \\ \infty & \text{otherwise.} \end{cases}$$

- i.e. when $m = 0$, l_{ij}^0 is the weight of the shortest path between vertex i and vertex j with no edges
- $L^{(1)}$ is the adjacency matrix of the graph G (we use only one edge).
- $L^{(2)}$ is the weights of the shortest paths between i and j if we use up to two edges.
- $L^{(3)}$ is the weights of the shortest paths between i and j if we use up to three edges.

Step 1: Graph Representation to Matrix

- Represent the graph $G(V, E)$ using an adjacency matrix W.

Step 2: Initialization Initialize Diagonal and Non-Edge Entries

- Set diagonal entries $W_{ii} = 0$ for all vertices i and non-edge entries $W_{ij} = \infty$ for non-edges.
- 

Step 3: Sub-problem Definition

- Sub-problems l_{ij}^m represent shortest paths from i to j with at most m edges.

Matrix Multiplication Transition Function:

- The transition function is expressed as a matrix multiplication:

$$L^{(m)} = L^{(m-1)} \times W$$

    - Here, $L^{(m)}$ is the matrix of shortest paths using at most m edges.
    - This function implicitly updates the shortest paths in each iteration.

Iterative Process:

- Iteratively compute $L^{(m)}$ for $m = 1, 2, \ldots, n-1$ using the matrix multiplication transition function.

Result:

- The final matrix $L^{(n-1)}$ contains the shortest path distances between all pairs of vertices.

Complexity:

- The time complexity depends on the matrix multiplication algorithm used, typically $O(V^3 \log V)$ with optimized methods.

Application:

- Effective for dense graphs and where the graph's edge weights are subject to change.

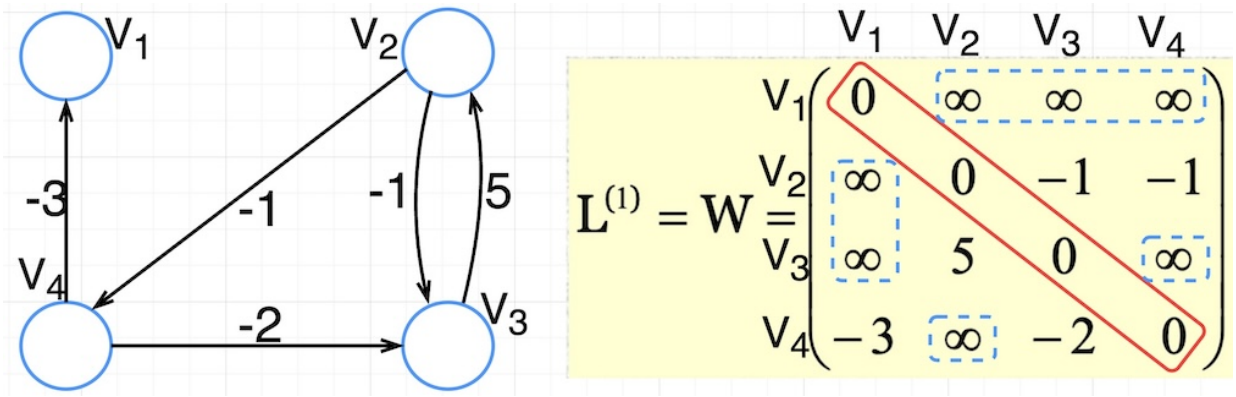
DP for APSP Example, Given the graph $G(V, E)$, $V = \{V_1, V_2, V_3, V_4\}$,
$E = \{(V_2, V_3, -1), (V_2, V_4, -1), (V_3, V_2, 5), (V_4, V_3, -2), (V_4, V_1, -3)\}$,

- Edges are in the $i \rightsquigarrow j$, maximum length of shortest path between vertex i and j is $(n-1)$

Step 1: Initialization: $l_{ij} = w_{ij}$, Apply the formula to calculate $l_{ij}^{(1)} = w$

- Initialize l_{ij} to w_{ij}, the weight from vertex i to vertex j. If no direct edge exists, set it to ∞. Self-loops (l_{ii}) are 0.

Given Graph $G(V, E)$ and Adjacency Matrix W: For Calculating of $L^{(m)}$:

- 

Step 1: Initialization $L^{(0)}$

- $L^{(0)} = W$ (as given above).

$$W = \begin{bmatrix} 0 & \infty & \infty & \infty \\ \infty & 0 & -1 & -1 \\ \infty & 5 & 0 & \infty \\ -3 & \infty & -2 & 0 \end{bmatrix}$$

Step 2: Calculate $L^{(1)}$

$$L^{(1)} = w = \begin{bmatrix} 0 & \infty & \infty & \infty \\ \infty & 0 & -1 & -1 \\ \infty & 5 & 0 & \infty \\ -3 & \infty & -2 & 0 \end{bmatrix}$$

Calculate $L_{1,1}^{(1)}$: The calculation of $L_{1,1}^{(1)}$ involves the dot product of the first row of $L^{(0)}$ and the first column of W:

- $L_{1,1}^{(1)} = (L_{1,1}^{(0)} \times W_{1,1}) + (L_{1,2}^{(0)} \times W_{2,1}) + (L_{1,3}^{(0)} \times W_{3,1}) + (L_{1,4}^{(0)} \times W_{4,1})$
- Plugging in the values: $L_{1,1}^{(1)} = (0 \times 0) + (\infty \times \infty) + (\infty \times \infty) + (\infty \times -3)$
- Since terms like $\infty \times \infty$ and $\infty \times -3$ are treated as ∞, the calculation becomes: $L_{1,1}^{(1)} = 0 + \infty + \infty + \infty$

- The minimum of these values is 0, so $L_{1,1}^{(1)} = 0$

Calculate $L_{1,2}^{(1)}$: $L_{1,2}^{(1)}$ is calculated as:

- $L_{1,2}^{(1)} = (L_{1,1}^{(0)} \times W_{1,2}) + (L_{1,2}^{(0)} \times W_{2,2}) + (L_{1,3}^{(0)} \times W_{3,2}) + (L_{1,4}^{(0)} \times W_{4,2})$
- Plugging in the values: $L_{1,2}^{(1)} = (0 \times \infty) + (\infty \times 0) + (\infty \times 5) + (\infty \times \infty)$
- In matrix multiplication, terms like $\infty \times \infty$ and $\infty \times 5$ are considered as ∞. $L_{1,2}^{(1)} = \infty + \infty + \infty + \infty$
- The minimum value here is ∞ since all terms are ∞. $L_{1,2}^{(1)} = \infty$.

Calculate $L_{1,3}^{(1)}$: $L_{1,3}^{(1)}$ is calculated as:

- $L_{1,3}^{(1)} = (L_{1,1}^{(0)} \times W_{1,3}) + (L_{1,2}^{(0)} \times W_{2,3}) + (L_{1,3}^{(0)} \times W_{3,3}) + (L_{1,4}^{(0)} \times W_{4,3})$
- Plugging in the values: $L_{1,3}^{(1)} = (0 \times \infty) + (\infty \times -1) + (\infty \times 0) + (\infty \times -2)$
- Terms involving ∞ are considered as ∞, except for $\infty \times -1$ and $\infty \times -2$ which are indeterminate.
- The minimum value here is indeterminate, so we consider it as ∞.
- the result for $L_{1,3}^{(1)}$: $L_{1,3}^{(1)} = \infty$

Calculate $L_{1,4}^{(1)}$: $L_{1,4}^{(1)}$ is calculated as:

- $L_{1,4}^{(1)} = (L_{1,1}^{(0)} \times W_{1,4}) + (L_{1,2}^{(0)} \times W_{2,4}) + (L_{1,3}^{(0)} \times W_{3,4}) + (L_{1,4}^{(0)} \times W_{4,4})$
- Plugging in the values: $L_{1,4}^{(1)} = (0 \times \infty) + (\infty \times -1) + (\infty \times \infty) + (\infty \times 0)$
- As before, terms involving ∞ are considered as ∞, except for $\infty \times -1$ which is indeterminate.
- The minimum value here is also indeterminate, so we consider it as ∞.
- the result for $L_{1,4}^{(1)}$: $L_{1,4}^{(1)} = \infty$

- $L_{2,1}^{(1)}$: The original entry is ∞. Now, consider paths through each vertex:
  - Through V_1: ∞ (no path from V_2 to V_1 to V_1).
  - Through V_2: ∞ (self-loop, ignored).
  - Through V_3: $L_{2,3}^{(0)} + W_{3,1} = -1 + \infty = \infty$ (no path from V_3 to V_1).
  - Through V_4: $L_{2,4}^{(0)} + W_{4,1} = -1 + (-3) = -4$.
  - The minimum of these is -4, so $L_{2,1}^{(1)} = -4$.
- Similarly, we calculate for each pair (i, j).

Step 3: Calculate $L^{(2)}$

$$L^{(2)} = \begin{bmatrix} 0 & \infty & \infty & \infty \\ -4 & 0 & -3 & -1 \\ \infty & 5 & 0 & 4 \\ -3 & 3 & -2 & 0 \end{bmatrix},$$

Calculations for $l_{i,j}^{(2)}$ by using $L^{(1)}$, bassically is $l_{i,j}^{n-1}$

$$l_{i,j}^{(2)} = min\left\{l_{i,j}^{(1)}, min\{l_{i,k}^{(1)} + w_{k,j}\}\right\}$$
$$l_{i,j}^{(2)} = \min\{l_{i,j}^{(1)}, \min\{l_{i,2}^{(1)} + w_{2,j}\}\}$$

- $l_{ij}^{(2)} = \infty$,
- $l_{2,1}^{(1)} + w_{1,1} = \infty + 0$,
- $l_{2,2}^{(1)} + w_{2,1} = 0 + \infty$,
- $l_{2,3}^{(1)} + w_{3,1} = (-1) + \infty$,
- $l_{2,4}^{(1)} + w_{4,1} = (-1) + (-3) = -4$,

- $\pi . l_{2,1}^{(1)} = 4$
- Here, we consider paths that include at most two intermediate vertices. We update the matrix similarly by considering paths through up to two intermediate vertices.
- For example, $L_{2,1}^{(2)}$ will consider paths from V_2 to V_1 that can go through V_1 and V_2, or V_1 and V_3, etc., and take the minimum of all such paths.

Step 4: Calculate $L^{(3)}$

- Now, we consider paths that include at most three intermediate vertices. This process is similar to the previous steps, where we consider all possible paths with up to three intermediates and update the matrix accordingly.

Conclusion:

- Each step $L^{(m)}$ refines the shortest paths by considering more intermediate vertices.
- The final matrix $L^{(n-1)}$ gives the shortest path distances between all vertex pairs considering paths through up to $n-1$ intermediates.

PrEcEDENcE MATRIX
While computing $L^{(k)}$, also compute $\pi^{(k)}$

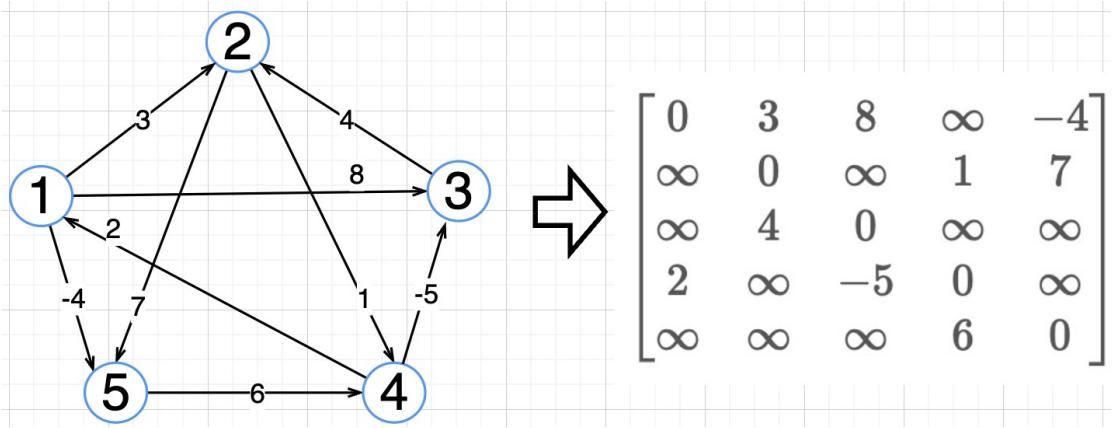
- If $l_{ij}^{(m)} = l_{ij}^{(m-1)}$ then $\pi_{ij}^{(K)} = \pi_{ij}^{(k-1)}$
- If ($l_{ij}^{(m)} = l_{ij}^{(m-1)} + w_{ij}$ then $\pi_{ij}^{(m)} = k$

$$L^{(3)} = \begin{bmatrix} 0 & \infty & \infty & \infty \\ -4 & 0 & -3 & 15 \\ \infty & 5 & 0 & 4 \\ -3 & 3 & -2 & 0 \end{bmatrix},$$

$$A^3 = \begin{bmatrix} 0 & 3 & 5 & 6 \\ 7 & 0 & 2 & 3 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix}, \quad A^4 = \begin{bmatrix} 0 & 3 & 5 & 6 \\ 5 & 0 & 2 & 3 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix}$$

Example APSP using DP and Matrix Multiplication

- No direct edge between i and j the, $w[i,j] = \infty$



$$\begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

Step 1: Initialization: $l_{ij} = w_{ij}$, Apply the formula to calculate $L_{ij}^{(1)} = w$

$$L^{(1)} = \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

Shortest Path (SP) Algorithms

	BFS	Dijk's	BM-Ford	FI-WS
Complexity	$O(V+E)$	$O((V+E)\log(V))$	$O(VE)$	$O(V^3)$
Recommended graph size	Large	Large/ Medium	Medium/ Small	Small
Good for APSP?	Only works on unweighted graphs	Ok	Bad	Yes
Can detect negative cycles?	No	No	Yes	Yes
SP on graph with weighted edges	Incorrect SP answer	Best algorithm	Works	Bad in general
SP on graph with unweighted edges	Best algorithm	Ok	Bad	Bad in general

Floyd-Warshall in APSP (Dynamic Programming)

Overview:

- Floyd-Warshall algorithm uses Dynamic Programming (DP) for APSP, finding shortest paths in a graph between all vertex pairs.
- The algorithm iteratively updates the shortest path estimates until they converge to the actual shortest paths.

Floyd-Warshall Algorithm:

1. Initialization:

- Create a matrix D of dimensions $n \times n$, where n is the number of vertices, and initialize it such that $D_{ij}^{(k)}$ is the weight of a shortest path from i to j with intermediate vertices, or ∞ if there is no edge. thus, $\delta(i,j) = D_{ij}^{k}$. $D_{ij}^{(0)} = a_{ij}$

2. Iteration:

- For each vertex k from 1 to n, update the matrix D using the following rule for each pair of vertices i and j:

$$D_{ij}^{k} = \min\left\{D_{ij}^{(k-1)}, D_{ik}^{(k-1)} + D_{kj}^{(k-1)}\right\}$$

3. Result:

- The final matrix D contains the shortest path distances between all pairs of vertices.

- Time Complexity: $O(n^3)$, efficient for dense graphs.

Algorithm Steps:

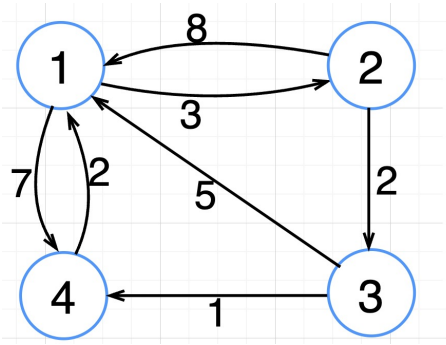
1. Initialize a matrix $D^{(0)}$ with dimensions $n \times n$, where n is the number of vertices.
2. Set $D_{ij}^{(0)} = \infty$ for $i \neq j$ and $D_{ii}^{(0)} = 0$.
3. For each edge (u, v) with weight $w(u, v)$, set $D_{uv}^{(0)} = w(u, v)$.
4. For $k = 1$ to n, update the matrix $D^{(k)}$ from $D^{(k-1)}$ using the recurrence relation:

$$D_{ij}^{(k)} = \min\left(D_{ij}^{(k-1)}, D_{ik}^{(k-1)} + D_{kj}^{(k-1)}\right)$$

Conclusion:

- The final matrix $D^{(n)}$ contains the shortest path distances between all pairs of vertices.
- The time complexity of the algorithm is $O(n^3)$, which makes it efficient for dense graphs.

Floyd-Marshall Example from Video:

- 

$$A^0 = W = \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & \infty \\ 5 & \infty & 0 & 1 \\ 2 & \infty & \infty & 0 \end{bmatrix}$$

For $A^{(1)}$, only selected first row and column, diagonal keeps in 0s.

$$A^1 = \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & & \\ 5 & & 0 & \\ 2 & & & 0 \end{bmatrix}, \quad A^1 = \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & \infty & 0 \end{bmatrix}$$

$$A^{(1)}[2,3] = \min\{A^{(0)}[2,1], A^{(0)}[2,1] + A^{(0)}[1,3]\} = 2 < 8 + \infty$$
$$A^{(1)}[2,4] = \min\{A^{(0)}[2,4], A^{(0)}[2,1] + A^{(0)}[1,4]\} = \infty > 8 + 7(15)$$
$$A^{(1)}[3,2] = \min\{A^{(0)}[3,2], A^{(0)}[3,1] + A^{(0)}[1,2]\} = \infty > 5 + 3(8)$$
$$A^{(1)}[3,4] = \min\{A^{(0)}[3,4], A^{(0)}[3,1] + A^{(0)}[1,4]\} = 1 < 5 + 7(12)$$
$$A^{(1)}[4,2] = \min\{A^{(0)}[4,2], A^{(0)}[4,1] + A^{(0)}[1,2]\} = \infty > 2 + 3(5)$$
$$A^{(1)}[4,3] = \min\{A^{(0)}[4,3], A^{(0)}[4,1] + A^{(0)}[1,3]\} = \infty = 2 + \infty$$

For $A^{(2)}$, only selected 2nd row and column, diagonal keeps in 0s.

$$A^2 = \begin{bmatrix} 0 & 3 & & \\ 8 & 0 & 2 & 15 \\ & 8 & 0 & \\ & 5 & & 0 \end{bmatrix}, \quad A^2 = \begin{bmatrix} 0 & 3 & 5 & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix}$$

$$A^{(2)}[1,3] = \min\{A^{(1)}[1,3], A^{(1)}[1,2] + A^{(1)}[2,3]\} = \infty > 3 + 2(5)$$
$$A^{(2)}[1,4] = \min\{A^{(1)}[1,4], A^{(1)}[1,2] + A^{(1)}[2,4]\} = 7 < 3 + 15(18)$$
$$A^{(2)}[3,1] = \min\{A^{(1)}[3,1], A^{(1)}[3,2] + A^{(1)}[2,1]\} = 5 < 8 + 3(11)$$
$$A^{(2)}[3,4] = \min\{A^{(1)}[3,4], A^{(1)}[3,2] + A^{(1)}[2,4]\} = 1 < 8 + 15(23)$$

$A^{(2)}[4,1] = \min\{A^{(1)}[4,1], A^{(1)}[4,2] + A^{(1)}[2,1]\} = 2 < 3 + 5(8)$
$A^{(2)}[4,3] = \min\{A^{(1)}[4,3], A^{(1)}[4,2] + A^{(1)}[2,3]\} = \infty < 5 + 2(7)$

For $A^{(3)}$, only selected 3rd row and column, diagonal keeps in 0s.

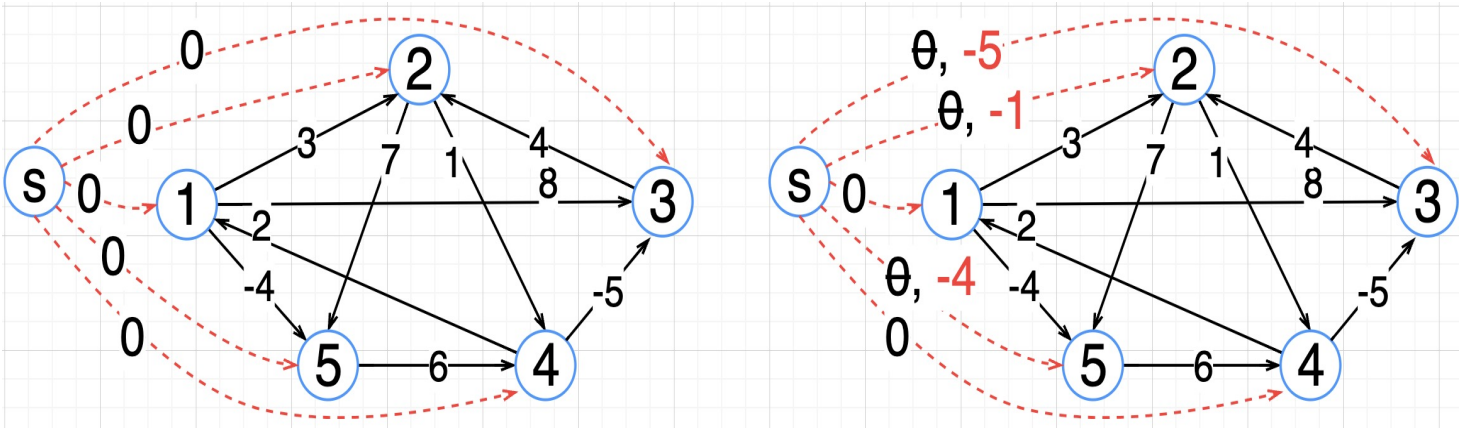
$$A^3 = \begin{bmatrix} 0 & & 5 & \\ & 0 & 2 & \\ 5 & 8 & 0 & 1 \\ & & 7 & 0 \end{bmatrix}, \quad A^3 = \begin{bmatrix} 0 & 3 & 5 & 6 \\ 7 & 0 & 2 & 3 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix}$$

For $A^{(4)}$, only selected 4th row and column, diagonal keeps in 0s.

$$A^4 = \begin{bmatrix} 0 & & & 6 \\ & 0 & & 3 \\ & & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix}, \quad A^4 = \begin{bmatrix} 0 & 3 & 5 & 6 \\ 5 & 0 & 2 & 3 \\ 3 & 6 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix}$$

Johnson's Algorithm

Example



Steps to apply Johnson's Algorithm

Step 1: Compute a potential h for the graph G, which are the shortest paths from vertex s to all vertices

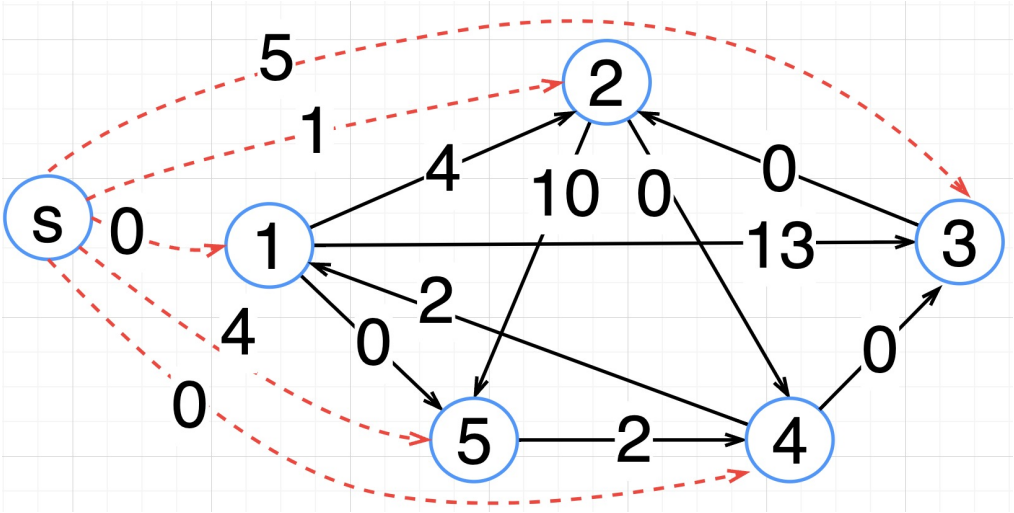
Iteration	(s,1)	(s,2)	(s,3)	(s,4)	(s,5)	(1,2)	(1,3)	(1,5)	(2,4)	(2,5)	(3,2)	(4,1)	(4,3)	(5,4)
1st iteration	√	√	√	√	√	√	√	**[√]**	√	√	√	√	**[√]**	√
2nd iteration	√	√	√	√	√	√	√	√	√	√	**[√]**	√	√	√

$h(1) = \delta(s,1) = 0$
$h(2) = \delta(s,2) = 0 + (-5) + 4 = -1$
$h(3) = \delta(s,3) = 0 + (-5) = -5$
$h(4) = \delta(s,4) = 0$
$h(5) = \delta(s,4) = 0 + (-4) = -4$

Step 2: Reweighting the new graph G where $\hat{w}(u,v) = w(u,v) + h(u) - h(v)$

$\hat{w}(1,2) = w(1,2) + h(1) - h(2) = 3 + (0) - (-4) = 4$
$\hat{w}(1,3) = w(1,3) + h(1) - h(3) = 8 + (0) - (-5) = 13$
$\hat{w}(1,5) = w(1,5) + h(1) - h(5) = -4 + (0) - (-4) = 0$
$\hat{w}(2,4) = w(2,4) + h(2) - h(4) = 1 + (-1) - (0) = 0$
$\hat{w}(2,5) = w(2,5) + h(2) - h(5) = 7 + (-1) - (-4) = 10$
$\hat{w}(3,2) = w(3,2) + h(3) - h(2) = 4 + (-5) - (-1) = 0$
$\hat{w}(4,1) = w(4,1) + h(4) - h(1) = 2 + (0) - (0) = 2$
$\hat{w}(4,3) = w(4,3) + h(4) - h(3) = -5 + (0) - (-5) = 0$
$\hat{w}(5,4) = w(5,4) + h(5) - h(4) = 6 + (-4) - (0) = 2$
$\hat{w}(s,1) = w(s,1) + h(s) - h(1) = 0 + (0) - (0) = 0$
$\hat{w}(s,2) = w(s,2) + h(s) - h(2) = 0 + (0) - (-1) = 1$
$\hat{w}(s,3) = w(s,3) + h(s) - h(3) = 0 + (0) - (-5) = 5$
$\hat{w}(s,4) = w(s,4) + h(s) - h(4) = 0 + (0) - (0) = 0$
$\hat{w}(s,5) = w(s,5) + h(s) - h(5) = 0 + (0) - (-4) = 4$

- Draw a new graph by using the reweighted values



Step 3: Compute all-pairs shortest paths dist′ with the new weighting.

- Step3 is calculated by Dijkstra's Algorithm
- To complete this part, need to finish this video https://youtu.be/058U6ZXPhDc

Notice:

- Johnson's Algorithm is efficient for sparse graphs
- Johnson's algorithm, effective on graphs with negative edges, operates with a time complexity of $O(VE + V^2 \log V)$. Johnson 算法適用於帶有負邊的圖，時間複雜度為 $O(VE + V^2 \log V)$。

Before Class

Dynamic Programming Approach for All-Pairs Shortest Paths (APSP)_

動態規劃方法用於所有對最短路徑 (APSP)

1. Graph Representation: 圖形表示

- The graph $G(V, E)$ is represented by adjacency matrix $W = (w_{ij})$ where vertices and edges are denoted by V and E respectively.
  圖 $G(V, E)$ 由鄰接矩陣 $W = (w_{ij})$ 表示，其中頂點和邊分別由 V 和 E 表示。

2. Diagonal Entries: 對角項

- All diagonal entries $w_{ii} = 0$ for $1 \leq i \leq n$, indicating zero weight from a vertex to itself.
  所有對角項 $w_{ii} = 0$ 對於 $1 \leq i \leq n$，表示從一個頂點到它自己的零權重。

3. Edge Weights: 邊權重

- Entry w_{ij} denotes the weight of edge (i, j) if $i \neq j$ and $i, j \in E$.
  項目 w_{ij} 表示邊 (i, j) 的權重，如果 $i \neq j$ 且 $i, j \in E$。

4. Non-Edge Entries: 非邊項目

- Entry $w_{ij} = \infty$ if $i \neq j$ and $(i, j) \notin E$, indicating no direct edge between vertices i and j.
  如果 $i \neq j$ 且 $(i, j) \notin E$，則項目 $w_{ij} = \infty$，表示頂點 i 和 j 之間沒有直接邊。

DP Programming Approach in APSP

1. Sub-problem Definition: 子問題定義

- Expression $P_{ij} = i \overset{P_{ik}}{\rightsquigarrow} k \overset{w_{kj}}{\rightsquigarrow} j$ and l_{ij}^m both decompose paths from `i` to `j` into smaller parts, aiding in solving the APSP problem.
  表達式 $P_{ij} = i \overset{P_{ik}}{\rightsquigarrow} k \overset{w_{kj}}{\rightsquigarrow} j$ 和 l_{ij}^m 都將從 `i` 到 `j` 的路徑分解成較小的部分，有助於解決 APSP 問題。

2. Optimal Substructure: 最優子結構

- Optimal solution for $i \overset{P_{ik}}{\rightsquigarrow} k$ contributes to solving $P_{ij}$and l_{ij}^m.
  $i \overset{P_{ik}}{\rightsquigarrow} k$ 的最優解有助於解決 P_{ij} 和 l_{ij}^m。

3. Intermediate Vertex: 中間頂點

- Intermediate vertex `k` enables finding shortest paths through iterative updates, akin to l_{ij}^m with at most m edges.
  中間頂點 `k` 通過反覆更新使得找到最短路徑成為可能，類似於最多包含 m 條邊的 l_{ij}^m。

4. Transition Function: 轉換函數

- Expressed as $D_{ij} = \min(D_{ij}, D_{ik} + w_{kj})$ or $l_{ij}^r = \min\{l_{ij}^{r-1}, \min\{l_{ik}^{r-1} + w_{kj}\}\}$ for iterative updates to distance matrix D.
  表示為 $D_{ij} = \min(D_{ij}, D_{ik} + w_{kj})$ 或 $l_{ij}^r = \min\{l_{ij}^{r-1}, \min\{l_{ik}^{r-1} + w_{kj}\}\}$ 用於反覆迭代以更新距離矩陣 D。
  - l_{ij}^m represents the shortest path from vertex i to vertex j with at most m edges.
    l_{ij}^m 表示從頂點 i 到頂點 j 的最短路徑，最多包含 m 條邊。
  - The goal is to compute the path length r, through a series of transitions, considering n vertices.
    目標是通過一系列轉換計算路徑長度 r，考慮 n 個頂點。
  - If $i \to k \to j$ is not the shortest path, then $l_{ij}^{(r)} = l_{ij}^{(r-1)}$.
    如果 $i \to k \to j$ 不是最短路徑，則 $l_{ij}^{(r)} = l_{ij}^{(r-1)}$。
  - $l_{ij}^r = \min\left\{l_{ij}^{r-1}, \min\{l_{ik}^{r-1} + w_{kj}\}\right\}$ is a transition function that attempts to find a shorter path by considering an additional edge.
    $l_{ij}^r = \min\left\{l_{ij}^{r-1}, \min\{l_{ik}^{r-1} + w_{kj}\}\right\}$ 是一個轉換函數，試圖通過考慮一個額外的邊來找到更短的路徑。
  - l_{ij}^{n-1} represents the shortest path from i to j considering $n-1$ edges, where n is the total number of vertices.
    l_{ij}^{n-1} 表示從 i 到 j 的最短路徑，考慮 $n-1$ 條邊，其中 n 是頂點的總數。

5. Solution Storage: 解決方案存儲

- Matrix D stores shortest path distances, updated iteratively, akin to l_{ij}^{n-1} for at most $n-1$ edges.
  矩陣 D 存儲最短路徑距離，並反覆更新，類似於最多包含 $n-1$ 條邊的 l_{ij}^{n-1}。

r **Explanation**, The variable r represents the number of edges in a particular path, and understanding its relation to the number of vertices n is crucial in analyzing and solving the All-Pairs Shortest Paths (APSP) problem using dynamic programming.

1. **Path Length Significance (路徑長度的重要性)**:
   - Calculating path length r helps in exploring all possible paths between pairs of vertices iteratively, aiding in finding the shortest path. 計算路徑長度 r 有助於反覆地探索頂點對之間的所有可能路徑，有助於找到最短路徑。
2. **Vertices-Edges Relation (頂點-邊關係)**:
   - In a graph, a path can have at most $n-1$ edges since each vertex, except the last one, contributes one edge to the path. 在圖中，一條路徑最多可以有 $n-1$ 條邊，因為每個頂點（除了最後一個）都為路徑貢獻了一條邊。
3. **Iterative Refinement (反覆精煉)**:
   - By considering paths of lengths $r = 0, 1, 2, ..., n-1$, we iteratively refine the solution, allowing the algorithm to build up the shortest paths incrementally. 通過考慮長度為 $r = 0, 1, 2, ..., n-1$ 的路徑，我們反覆精煉解決方案，使算法能夠逐步構建最短路徑。
4. **Sub-problem Dependency (子問題依賴性)**:
   - The shortest path with r edges depends on solutions to sub-problems with fewer edges, illustrating a dependency chain essential for dynamic programming. r 條邊的最短路徑取決於具有較少邊的子問題的解決方案，顯示出動態規劃所必需的依賴性鏈。

By comprehending the role of r and its connection to n vertices, we harness the power of dynamic programming to iteratively and efficiently solve the APSP problem.

通過理解 r 的作用及其與 n 個頂點的關聯，我們利用動態規劃的力量來反覆且高效地解決 APSP 問題。

The path length r in the context of All-Pairs Shortest Paths (APSP) is crucial for understanding the properties and constraints of the graph in question. Here's why it's significant and how it relates to n vertices and $n-1, n-2, n-3,$ etc:

1. **Vertex Limit (頂點限制)**:
   - In a graph with n vertices, the maximum number of edges in any simple path is $n-1$. 在具有 n 個頂點的圖中，任何簡單路徑中的最大邊數是 $n-1$。
2. **Path Length Calculation (路徑長度計算)**:
   - Calculating r, the length of the path, helps in understanding the structure of the graph and the relationship between vertices. 計算 r，路徑的長度，有助於理解圖的結構和頂點之間的關係。
3. **Intermediate Vertices (中間頂點)**:
   - The values $n-1, n-2, n-3,$ etc., represent the inclusion of intermediate vertices in the path, which can potentially lead to shorter or more accurate paths. 值 $n-1, n-2, n-3,$ 等表示路徑中包含中間頂點，這可能導致更短或更準確的路徑。
4. **Cycle Detection (循環檢測)**:
   - If a path has n or more edges, it must contain a cycle. Therefore, understanding r in relation to n and $n-1$ helps in cycle detection which is crucial for certain graph algorithms. 如果路徑具有 n 或更多邊，則必須包含循環。因此，理解 r 與 n 和 $n-1$ 的關係有助於循環檢測，這對於某些圖算法至關重要。
5. **Algorithm Efficiency (算法效率)**:
   - Knowing r can guide the design of more efficient algorithms by understanding the graph's properties better, thereby improving algorithmic performance. 知道 r 可以通過更好地理解圖的屬性來指導更高效算法的設計，從而提高算法性能。

These points explain the necessity of calculating r and its relation to n vertices and the intermediate vertices count $n-1, n-2, n-3,$ etc., in a succinct and comprehensive manner.

Iterative Calculation (迭代計算)

3. **Calculating $l_{ij}^{(1)}$ (計算 $l_{ij}^{(1)}$)**:
   - Use the formula $l_{i,j}^{(1)} = \min\left\{l_{i,j}^{(0)}, \min\{l_{i,k}^{(0)} + w_{k,j}\}\right\}$ to find the entries of the next matrix $L^{(1)}$.
   - 使用公式 $l_{i,j}^{(1)} = \min\left\{l_{i,j}^{(0)}, \min\{l_{i,k}^{(0)} + w_{k,j}\}\right\}$ 找到下一個矩陣 $L^{(1)}$ 的條目。
4. **Calculating $l_{ij}^{(2)}$ (計算 $l_{ij}^{(2)}$)**:
   - Similarly, calculate $l_{ij}^{(2)}$ using the formula $l_{i,j}^{(2)} = \min\left\{l_{i,j}^{(1)}, \min\{l_{i,k}^{(1)} + w_{k,j}\}\right\}$ to find the entries of the matrix $L^{(2)}$.
   - 同樣地，使用公式 $l_{i,j}^{(2)} = \min\left\{l_{i,j}^{(1)}, \min\{l_{i,k}^{(1)} + w_{k,j}\}\right\}$ 計算 $l_{ij}^{(2)}$ 以找到矩陣 $L^{(2)}$ 的條目。

Precedence Matrix (優先矩陣)

5. **Computing Precedence Matrix ($\pi^{(k)}$) (計算優先矩陣 ($\pi^{(k)}$)**:
   - While computing $L^{(k)}$, also compute $\pi^{(k)}$.
   - 在計算 $L^{(k)}$ 時，也計算 $\pi^{(k)}$。
6. **Updating Precedence Matrix (更新優先矩陣)**:
   - If $l_{ij}^{(m)} = l_{ij}^{(m-1)}$, then $\pi_{ij}^{(k)} = \pi_{ij}^{(k-1)}$.
   - 如果 $l_{ij}^{(m)} = l_{ij}^{(m-1)}$，則 $\pi_{ij}^{(k)} = \pi_{ij}^{(k-1)}$。
   - If $l_{ij}^{(m)} = l_{ij}^{(m-1)} + w_{ij}$, then $\pi_{ij}^{(m)} = k$.
   - 如果 $l_{ij}^{(m)} = l_{ij}^{(m-1)} + w_{ij}$，則 $\pi_{ij}^{(m)} = k$。

Final Iteration (最終迭代)

7. **Calculating $L^{(3)}$ (計算 $L^{(3)}$)**:
   - Continue the iterative process to compute $L^{(3)}$, using the formula $l_{i,j}^{(3)} = \min\left\{l_{i,j}^{(2)}, \min\{l_{i,k}^{(2)} + w_{k,j}\}\right\}$.
   - 繼續迭代過程以計算 $L^{(3)}$，使用公式 $l_{i,j}^{(3)} = \min\left\{l_{i,j}^{(2)}, \min\{l_{i,k}^{(2)} + w_{k,j}\}\right\}$。

Through these steps, you're incrementally refining the matrix $L^{(m)}$ to find the shortest path between all pairs of vertices. The Precedence Matrix helps to keep track of the path information as you progress through the iterations.

通過這些步驟，您可以逐步完善矩陣 $L^{(m)}$ 以找到所有頂點對之間的最短路徑。優先矩陣有助於在您完成迭代時跟踪路徑信息。