FOUNDATIONS OF ALGORITHM DESIGN AND MACHINE LEARNING

ALL-PAIRS SHORTEST PATH IN A GRAPH (Additional)

The All-Pairs Shortest Path (APSP) problem involves finding the shortest paths between all pairs of vertices in a given directed weighted graph. There are several possible cases for the graph, and different approaches exist depending on the graph's characteristics.

Given a weighted graph G=(V,E), where V represents the set of vertices and E represents the set of weighted edges, we can determine the shortest paths between all pairs of vertices. The approach to solving the APSP problem depends on the characteristics of the graph, leading to four cases:

1. A Directed Acyclic Graphs(DAG) with real-valued edge weights (no cycles).
2. A graph with strictly positive edge weights.
3. A graph with real-valued edge weights (including negatives) but no negative-weight cycles.
4. A graph with arbitrary edge weights, potentially containing negative cycles.

Case 1: Directed Acyclic Graphs (DAGs)

Given a Directed Acyclic Graphs **DAG**, compute the shortest paths between all pairs of vertices. Since DAGs contain no cycles. Extending the recursive DFS approach to handle all pairs via topological sorting and dynamic programming.

Pseudocode for All-Pairs Shortest Paths in DAGs

Extending the recursive DFS approach to handle all pairs via topological sorting and dynamic programming.

```
# Initialization
1. Initialize a 2D cost matrix where:
  - cost[u][v] = ∞ for all u, v ∈ V
  - cost[u][u] = 0 for all u ∈ V
  - For each edge (u, v) with weight w:
    cost[u][v] = w  # Initialize direct edges

2. Topologically sort nodes into "topo_order"

# Modified DFS-based Path Relaxation
Function DFSP_AP(u, current, visited, cost):
  If visited[current]:
    return
  Mark visited[current] = True

  For each successor v of current:
    # Update FIRST before recursion
    new_cost = cost[u][current] + weight(current, v)
    If new_cost < cost[u][v]:
      cost[u][v] = new_cost  # Update path cost

    DFSP_AP(u, v, visited, cost)  # Then recurse

# Main Execution
For each node u in topo_order:
  Initialize visited array to False for all nodes
  DFSP_AP(u, u, visited, cost)  # Compute paths from u
```

Pseudocode Explanation

1. **Topological Sorting**:
    - Ensures nodes are processed in an order where all predecessors of a node are handled before the node itself.
2. **Dynamic Programming via DFS**:
    - For each node u (acting as the source), recursively traverse its successors.
    - Update cost[u][v] for every reachable node v using the formula:

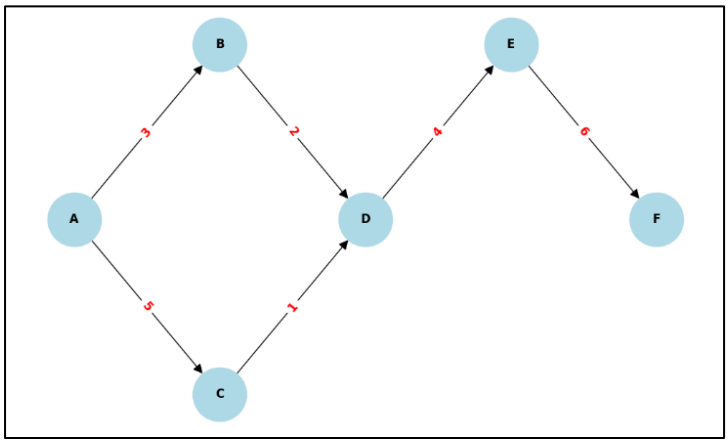
      $\text{cost}[u][v] = \min(\text{cost}[u][v], \text{cost}[u][\text{current}] + \text{weight}(\text{current}, v))$

Time complexity of **O(|V| *(|V| * |E|)) = O($|V|^2$ + |V| * |E|)**

Example:

Edges:

- A → B (3)
- A → C (5)
- B → D (2)
- C → D (1)
- D → E (4)
- E → F (6)



Topological Order: [A, B, C, D, E, F]

1. Initialization

- Initialize a **cost matrix** cost[u][v] where:
  - cost[u][u] = 0 (cost to self is zero)
  - All other entries are set to infinity (∞), since initially, no paths are known.

2. Process Node A as Source

- **Call DFSP_AP(A, A, visited, cost)** to compute paths from node A to all other nodes.

2.1: Visit Node A

- Mark **A** as visited.
- **Successors of A**: B, C.
- Update the paths from **A** to **B** and **C**:
  - cost[A][B] = min(∞, 0 + 3) = 3
  - cost[A][C] = min(∞, 0 + 5) = 5

2.2: Recurse into Node B

- Mark **B** as visited.
- **Successor of B**: D.
- Update path from **A** to **D**:
  - cost[A][D] = min(∞, cost[A][B] + 2) = 3 + 2 = 5

2.3: Recurse into Node C

- Mark **C** as visited.
- **Successor of C**: D.
- Update path from **A** to **D**:
  - cost[A][D] = min(5, cost[A][C] + 1) = min(5, 5 + 1) = 5 (no change since the previous path was shorter)

2.4: Recurse into Node D

- Mark **D** as visited.
- **Successor of D**: E.
- Update path from **A** to **E**:
  - cost[A][E] = min(∞, cost[A][D] + 4) = 5 + 4 = 9

2.5: Recurse into Node E

- Mark **E** as visited.
- **Successor of E**: F.
- Update path from **A** to **F**:
  - cost[A][F] = min(∞, cost[A][E] + 6) = 9 + 6 = 15
  -

After processing all nodes in the topological order (A → B → C → D → E → F), the final cost matrix will be:

	A	B	C	D	E	F
A	0	3	5	5	9	15
B	∞	0	∞	2	6	12
C	∞	∞	0	1	5	11
D	∞	∞	∞	0	4	10
E	∞	∞	∞	∞	0	6
F	∞	∞	∞	∞	∞	0

Case 2: Graphs with Positive Edge Weights

Given a graph where all edge weights are **positive**, compute the shortest paths between all pairs of vertices. Since there are no negative weights, **Dijkstra's algorithm** is the most efficient approach.

Modified Best-First Search for All-Pairs Shortest Paths (APSP) in Positive-Weighted Graphs. Extending the single-source BFSW algorithm to compute shortest paths between all pairs of vertices. Apply **Dijkstra's Algorithm** from each node using a priority queue for optimization.

```
# Initialize a cost matrix for all pairs
cost_matrix = { u: { v: ∞ for v in V } for u in V }

for each source u in V:
  visited = { v: 0 for v in V }  # Track processed nodes for this source
  cost = { v: ∞ for v in V }    # Shortest paths from u to all v
  cost[u] = 0              # Distance from u to itself is 0
  OrQ = PriorityQueue()       # Ordered by current shortest path cost
  OrQ.insert(u, 0)          # Start with the source

  while OrQ is not empty:
    j = OrQ.remove_min()     # Node with smallest current cost
    if visited[j] == 1:
      continue          # Skip already visited nodes
    visited[j] = 1

    # Update costs for all neighbors of j
    for each k in succ(j):
      new_cost = cost[j] + C[j][k]
      if new_cost < cost[k]:
        cost[k] = new_cost
        OrQ.insert_or_update(k, new_cost)

  # Store results for this source
  for v in V:
    cost_matrix[u][v] = cost[v]

return cost_matrix
```

Pseudo code Explanation

1. **Initialization**:

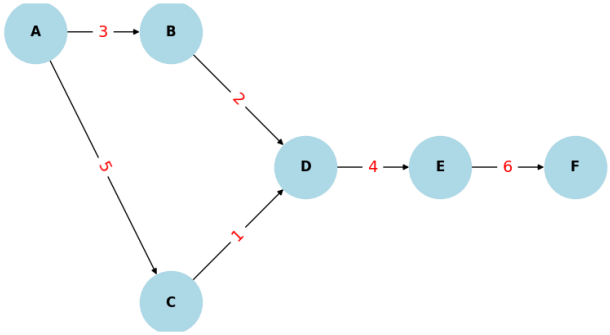
- A 2D matrix cost_matrix stores shortest distances for all pairs (u, v).
- For each source u, reset visited and cost arrays.

2. **Priority Queue Processing**:
   - Nodes are expanded in order of increasing path cost from u (greedy approach).
   - When a node j is dequeued, update costs for its neighbors via edge relaxation:
     cost[*k*]=min(cost[*k*],cost[*j*]+*C*[*j*][*k*])
3. **Termination**:
   - The algorithm processes all nodes reachable from u, not just a single target.
   - Results for u are stored in cost_matrix[u][*].

Example Execution

Graph:
Graph
Edges: A→B (3), A→C (5), B→D (2), C→D (1), D→E (4), E→F (6)

Processing Source A:



1. Initialize cost[A][A] = 0.
2. Expand A → update B (3), C (5).
3. Expand B → update D (3+2=5).
4. Expand C → update D (5→5+1=6 → discard, keep 5).
5. Expand D → update E (5+4=9).
6. Expand E → update F (9+6=15).

Final cost_matrix[A][*]

	A	B	C	D	E	F
A	0	3	5	5	9	15

Similar we can compute the cost_matrix for B, C, D, E, F.

The time complexity of the modified Best-First Search (BFSW) algorithm for **All-Pairs Shortest Paths (APSP)** in a graph with **positive edge weights** depends on

the priority queue (OrQ) implementation. Here's the breakdown in terms of nodes (V) and edges (E):For each source node *u*, the algorithm performs:

1. **Priority Queue Operations**:
    - Each node is inserted and removed once from the queue.
    - Total operations: **O(ElogV)** (with a binary heap).
2. **Edge Relaxations**:
    - Each edge is processed once.
    - Total operations: **O(E).**

Thus, for **one source**, the complexity is:**O(E+ElogV)**

All-Pairs (Aggregate Over All Sources)

Since the algorithm runs for **all N sources**, the total complexity is: **$O(V \cdot (E+ElogV))=O(VE+V^2logV)$.**

Case 3: Graphs with Negative Weights (No Negative Cycles)

Given a graph with **negative edge weights but no negative weight cycles**, compute the shortest paths between all pairs of vertices. Since Dijkstra's algorithm fails with negative weights, we use **Floyd-Warshall** or **Matrix Multiplication-Based Methods**.

1.Matrix Multiplication-Based Methods.

Using a recursive matrix multiplication approach, this algorithm computes the shortest paths between all pairs of vertices in a graph with negative edge weights but no negative weight cycles.

Pseudocode

```
Function MatrixMultiplicationAPSP(G):
  Input: G = (V, E), where V is the set of vertices and E is the set of edges
      C[i][j] = weight of edge (i, j) or ∞ if no edge exists
  Output: D[i][j] = shortest path from vertex i to vertex j

  n = |V|  # Number of vertices

  # Initialize base case for D[i][j][1]
  For each i in V:
    For each j in V:
      If i == j:
        D[i][j][1] = 0  # Distance from a vertex to itself is 0
      Else:
        D[i][j][1] = C[i][j]  # Distance is the weight of the edge (i, j) or ∞

  # Compute D[i][j][2^k] recursively
  k = 1
  While 2^k < n:
    For each i in V:
      For each j in V:
        D[i][j][2^k] = ∞
        For each m in V:
          D[i][j][2^k] = min(D[i][j][2^k], D[i][m][2^(k-1)] + D[m][j][2^(k-1)])
    k = k + 1

  # Final solution is D[i][j][n-1]
  Return D[i][j][n-1] for all i, j in V
```

Explanation of Pseudocode

1. **Initialization**:
    - The base case is defined as D[i,j,1]
        - D[i,j,1]=0 if i==j (distance from a vertex to itself is zero).
        - D[i,j,1]=C[i,j] if i≠j(direct edge cost or infinity if no direct edge exists).
2. **Recursive Definition**:
    - For each power of two (2^k), compute D[i,j,2^k] using the formula:D[i,j,2k]=min m∈V {D[i,m,2^{k-1}]+D[m,j,2^{k-1}]}

- This means that the shortest path from i to j using at most 2^k edges is computed by combining paths through an intermediate vertex m.

3. **Iterative Computation**:
    - Start with k=1 and double the number of edges considered at each step until $2^k \geq n$.
4. **Final Solution**:
    - The final shortest path between all pairs of vertices is stored in D[i,j,n−1] which considers all possible paths.

Time Complexity

- **Outer Loop**: Runs for log(V) iterations (as k doubles at each step).
- **Inner Loops**: For each pair of vertices (i,j), we iterate over all intermediate vertices m.
- Total complexity:**$O(V^3 \cdot log(V))$**

Key Observations

1. The algorithm uses dynamic programming principles and repeatedly updates the distance matrix.
2. The time complexity is dominated by the triple nested loop for matrix multiplication:
    - Time Complexity: **$O(V^3 logV)$*.***
3. It works efficiently for graphs with negative edge weights but no negative weight cycles.

2. Floyd-Warshall.

This algorithm computes the shortest paths between all pairs of vertices in a graph with **negative edge weights but no negative weight cycles**. It uses a recursive dynamic programming approach based on the Floyd-Warshall algorithm.

Pseudocode

```
Function FloydWarshallRecursive(G):
  Input: G = (V, E), where V is the set of vertices and E is the set of edges
      C[i][j] = weight of edge (i, j) or ∞ if no edge exists
  Output: F[i][j][n] = shortest path from vertex i to vertex j

  n = |V|  # Number of vertices

  # Initialize base case for F[i][j][0]
  For each i in V:
    For each j in V:
      If i == j:
        F[i][j][0] = 0  # Distance from a vertex to itself is 0
      Else:
        F[i][j][0] = C[i][j]  # Distance is the weight of the edge (i, j) or ∞

  # Compute F[i][j][k] recursively for k = 1 to n
  For k = 1 to n:
    For each i in V:
      For each j in V:
        F[i][j][k] = min(F[i][j][k-1], F[i][k][k-1] + F[k][j][k-1])

  # Final solution is F[i][j][n]
  Return F[i][j][n] for all i, j in V
```

Explanation of Pseudocode

1. **Initialization**:
    - The base case F[i,j,0] represents the shortest paths when no intermediate vertices are allowed.
    - If i==j, the distance is zero (F[i,j,0]=0).
    - Otherwise, F[i,j,0]is the direct edge weight C[i,j] or infinity (∞) if no direct edge exists.
2. **Recursive Formula**:
    - F[i,j,k] represents the shortest path from vertex I to vertex j using only the first k vertices as intermediate nodes.
    - The recursive formula is:$F[i,j,k]=\min(F[i,j,k-1],F[i,k,k-1]+F[k,j,k-1])$
    - This means that the shortest path from i to j either:

- Does not pass through vertex k*k* (use F[i,j,k−1]), or
- Passes through vertex k (use F[i,k,k−1]+F[k,j,k−1]).

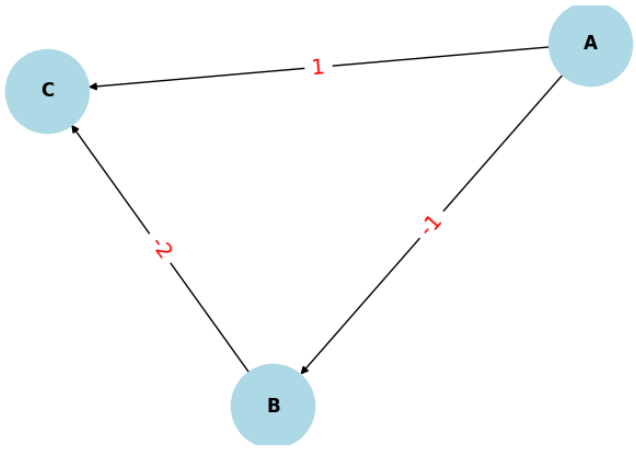
3. **Final Solution**:
    - After processing all vertices as possible intermediates (k=n), the final matrix F[i,j,n] contains the shortest paths between all pairs of vertices.

Key Observations

1. The algorithm systematically considers all possible paths by incrementally adding intermediate nodes.
2. It handles negative edge weights but requires no negative weight cycles.
3. The time complexity is cubic **($O(V^3)$)**, making it efficient for dense graphs.

Example

1. **Vertices**: A, B, C
2. **Edges (Directed)**:
    - A → B with weight **-1**
    - B → C with weight **-2**
    - A → C with weight **1**



Graph Properties

3. **Contains negative-weight edges**
4. **No negative cycles** (The cycle A → B → C → A is hypothetical and does not exist in this directed acyclic graph)

Floyd-Warshall Execution

1: Initial Distance Matrix

	A (0)	B (1)	C (2)
A	0	-1	1
B	∞	0	-2
C	∞	∞	0

2: After Intermediate Node B

- Update paths using B as an intermediate node:
    - A → C:
      min(1,(−1)+(−2))=−3\min(1, (-1) + (-2)) = -3min(1,(−1)+(−2))=−3

	A (0)	B (1)	C (2)
A	0	-1	-3
B	∞	0	-2
C	∞	∞	0

3: Final Distance Matrix

(Same as Step 2 since no further updates occur)

	A (0)	B (1)	C (2)
A	0	-1	-3
B	∞	0	-2
C	∞	∞	0

Key Observations

1. **Negative Edge Handling**: Floyd-Warshall correctly handles negative weights (e.g., A → C improves from **1** to **-3** via B).
2. **Cycle Detection**: No negative cycles exist. If C → A had weight **3**, the cycle A → B → C → A would sum to **0** (non-negative).
3. **Time Complexity**: O ($|V|^3$) = O(3^3) = 27operations for this example.

Case 4: Graphs with Negative Edge Cycles

Given a graph that **may contain negative weight cycles**, compute the shortest paths between all pairs of vertices. If a **negative cycle** exists, distances decrease indefinitely, making shortest paths undefined.

To compute all-pairs shortest paths in graphs that may contain negative weight cycles, the **Bellman-Ford algorithm** can be adapted.

```
function AllPairsBellmanFord(G):
  n = |V|
  // Initialize distance matrix
  dist = n x n matrix filled with ∞

  for each source s in V:
    // Single-source Bellman-Ford for source s
    dist[s][s] = 0
    for i = 1 to n-1:
      for each edge (u, v) in E:
        if dist[s][v] > dist[s][u] + C(u, v):
          dist[s][v] = dist[s][u] + C(u, v)

    // Negative cycle detection for source s
    has_negative_cycle = false
    for each edge (u, v) in E:
      if dist[s][v] > dist[s][u] + C(u, v):
        has_negative_cycle = true
        break

    // Propagate -∞ if negative cycle exists
    if has_negative_cycle:
      for each node v in V:
        if reachable_from_cycle(s, v):
          dist[s][v] = -∞

  return dist
```

Explanation of Pseudocode

1. **Matrix Initialization**
    - Creates a 2D distance matrix instead of a 1D array
    - dist[s][v] = shortest distance from source s to v
2. **Per-Source Execution**
    - Runs Bellman-Ford **once for each vertex** as the source
    - Total time complexity: $\mathbf{O(|V|^2|E|)}$

3. **Negative Cycle Handling**
    - Detects cycles per source using edge relaxation checks
    - Propagates $-\infty-\infty$ to all nodes reachable from a negative cycle via reachable_from_cycle() (implemented via BFS/DFS)

In summary, different approaches to the All-Pairs Shortest Path (APSP) problem are suited to distinct types of graphs, each with its own computational complexity.

Case 1 (Directed Acyclic Graphs - DAGs): The Recursive DFS Algorithm, which leverages the acyclic property of DAGs, provides an efficient solution by calculating all-pair paths at each node. This results in a time complexity of $\mathbf{O(|V|^2 + |V| * |E|)}$, making it particularly suitable for DAGs due to the absence of cycles.

Case 2 (Graphs with Positive Edge Costs): Dijkstra's Algorithm, adapted for all-source shortest paths by running it from each node as the source, gives a time complexity of $\mathbf{O(|V| * (|E| \log |V|))}$. This approach is optimal for graphs with positive edge weights, as Dijkstra's algorithm efficiently finds the shortest path using a priority queue.

Case 3 (Graphs with Negative Edges but No Negative Cycles): For graphs with negative edges, the Floyd-Warshall Algorithm and Matrix Multiplication offer viable solutions. Floyd-Warshall has a time complexity of $\mathbf{O(|V|^3)}$, while Matrix Multiplication provides a slightly faster $\mathbf{O(|V|^3 \log |V|)}$. Both are effective when negative edges are present but without negative weight cycles, offering comprehensive solutions for all pairs.

Case 4 (Graphs with Negative Cycles): The Bellman-Ford Algorithm, with a complexity of $\mathbf{O(|E| * |V|^2)}$, is suitable for graphs that may contain negative edge cycles. While slower than some of the other methods, Bellman-Ford is capable of detecting negative weight cycles and can handle graphs with such complexities.

Graph Type	Algorithm	Time Complexity
Directed Acyclic Graphs (DAGs)	Recursive DFS	$\mathbf{O(\|V\|^2 + \|V\| * \|E\|)}$
Graphs with Positive Edge Costs	Dijkstra's Algorithm (per node)	$\mathbf{O(\|V\| * (\|E\| \log \|V\|))}$
Graphs with Negative Edges (No Cycles)	Floyd-Warshall	$\mathbf{O(\|V\|^3)}$
	Matrix Multiplication	$\mathbf{O(\|V\|^3 \log \|V\|)}$
Graphs with Negative Cycles	Bellman-Ford	$\mathbf{O(\|E\| * \|V\|^2)}$

In conclusion, the choice of the algorithm depends on the type of graph being dealt with (DAG, positive weights, negative weights, or negative cycles) and the trade-off between time complexity and the graph's properties. Each algorithm provides an efficient method for tackling APSP based on the graph’s structure, offering solutions that balance performance and accuracy in varied contexts.