Module Summary: A Journey Through Shortest Path Algorithms

This module provided a comprehensive exploration of algorithms for finding the shortest path in a graph. My understanding progressed from a simple, greedy approach to more robust methods capable of handling complex scenarios, all underpinned by the powerful paradigm of **Dynamic Programming**.

Single-Source Shortest Path (SSSP)

My journey began with **Dijkstra's algorithm**, which I learned is a "greedy" method for finding the shortest path from a single source node to all other nodes. Its core logic involves iteratively selecting the "not-sure" node with the smallest estimated distance, updating its neighbors, and marking it as "sure." I now understand that Dijkstra's efficiency is heavily dependent on the data structure used to manage the nodes, ranging from a straightforward but slow O(n2) with arrays to a much faster O(nlogn+m) with Fibonacci heaps. A key takeaway for me was its major limitation: **Dijkstra's algorithm cannot handle negative edge weights**.

This limitation was addressed by the **Bellman-Ford algorithm**. I learned that its fundamental difference is that instead of greedily picking one node, it systematically relaxes *every* edge in the graph for $n-1$ iterations. This methodical approach allows it to correctly handle negative edge weights. I also found its ability to detect negative cycles—by running one final iteration to see if distances still improve—to be a powerful feature. The trade-off is its slower time complexity of O(nm), making it less ideal for dense graphs without negative weights.

The Power of Dynamic Programming (DP)

The introduction to Bellman-Ford served as my gateway to understanding **Dynamic Programming**. I now see DP as a strategy for solving complex problems by breaking them down into smaller, simpler pieces. The two defining characteristics that I'll look for in the future are:

1. **Optimal Substructure**: An optimal solution to the overall problem can be built from the optimal solutions of its subproblems.
2. **Overlapping Subproblems**: The same subproblems are encountered and solved repeatedly.

The Fibonacci sequence was a perfect introductory example. The naive recursive solution was exponential because it recalculated the same values over and over. By storing the results of subproblems (using a technique I learned is called **memoization** in a top-down approach, or by simply building a table in a bottom-up approach), the complexity was reduced to linear time. I realized Bellman-Ford is a perfect example of bottom-up DP, where the shortest path after i iterations is built upon the solution from i−1 iterations.

All-Pairs Shortest Path (APSP)

Finally, I moved beyond single-source problems to finding the shortest path between *all pairs* of nodes. While I could run Bellman-Ford n times, I learned there's a more elegant DP solution: the **Floyd-Warshall algorithm**. I grasped its core concept of iteratively considering each node k as a potential intermediate point in the path between any two nodes u and v. The update rule, D(k)[u,v]=min(D(k−1)[u,v],D(k−1)[u,k]+D(k−1)[k,v]), clearly demonstrated the optimal substructure at play. With a clean, three-loop structure, its O(n3) complexity is easy to understand, and it offers a simpler implementation than running Dijkstra n times, with the added benefit of handling negative weights.

Overall, this module connected the dots between specific graph algorithms and the broader, powerful concept of Dynamic Programming, giving me a much deeper understanding of how to approach and solve optimization problems.

Task 2:

```
FUNCTION GreedyColouring(graph):
  // graph is represented by a list of nodes and their neighbours
  // e.g., graph = { 'A': ['B', 'C'], 'B': ['A'], 'C': ['A'] }

  // 1. Initialize a dictionary to store the color of each node
  colors = {}
  FOR each node in graph:
    colors[node] = None  // 'None' signifies the node is not yet coloured

  // 2. Iterate through each node to assign a color
  FOR each node in graph:
    // a. Find the colors already used by its neighbours
    neighbour_colors = set()
    FOR each neighbour of node:
      IF colors[neighbour] IS NOT None:
        add colors[neighbour] to neighbour_colors

    // b. Find the smallest available color (0, 1, or 2)
    assigned_color = None
    FOR color_option in [0, 1, 2]:
      IF color_option IS NOT IN neighbour_colors:
        assigned_color = color_option
        BREAK // Stop searching once the smallest color is found

    // Assign the found color to the current node
    colors[node] = assigned_color

  // 3. Return the final color assignments
  RETURN colors
```

Does the approach always produce an optimal colouring? When does it fail?

No, this greedy approach **does not always produce an optimal colouring**. An optimal colouring uses the minimum number of colors possible for a given graph (known as the graph's chromatic number). This algorithm can often use more colors than the optimal number.

The success and optimality of the algorithm are highly dependent on the **order in which the nodes are processed**. A different node order can produce a completely different and potentially better (or worse) result.

Example of Sub-optimal Colouring:

Consider a graph that is **2-colorable** (bipartite). Our algorithm might unnecessarily use a third color due to an unlucky node ordering.

Let's say we have a central node A connected to four other nodes B, C, D, and E, which form a square (B-C-D-E-B). This graph is 2-colorable.

- **Optimal colouring:** color(A)=0, color(B,D)=1, color(C,E)=0. This uses only 2 colors.
- **Greedy colouring (sub-optimal):** If we process the nodes in the order A, B, C, D, E:
    1. color(A) = **0**
    2. color(B) (neighbour A=0) = **1**
    3. color(C) (neighbours A=0, B=1) = **2** // Unnecessarily uses the third color
    4. color(D) (neighbours A=0, C=2) = **1**
    5. color(E) (neighbours A=0, B=1, D=1) = **2**

The greedy choice at step 3 resulted in using **3 colors** for a graph that only needed **2**.

The algorithm **fails to find a valid colouring** when a node's neighbours have already used all 3 available colors. For example, in a complete graph with 4 nodes (K4), where every node is connected to every other node, the fourth node processed will have neighbours colored 0, 1, and 2, making it **un-colourable**.

Why is your approach greedy?

This approach is **greedy** because it makes a **locally optimal choice** at each step without considering the global consequences. 🧐

For each node, the algorithm asks a simple, short-sighted question: "What is the smallest-numbered color I can use right now?" It immediately picks that color (e.g., preferring 0 over 1) and moves on. It **never reconsiders** a past decision.

It doesn't strategize or look ahead to see if picking a different color (e.g., 1 instead of 0) might prevent problems later on and lead to a better overall solution. This "take the first and best option now" strategy, without a broader plan, is the defining characteristic of a greedy algorithm.

Task 3:

```
FUNCTION IsBipartite(graph):
 // graph is represented by a list of nodes and their neighbours
 // e.g., graph = { 'A': ['B', 'C'], 'B': ['A', 'D'], ... }

 // 1. Initialize a dictionary to store colors. 'None' means uncolored.
 colors = {}
 FOR each node in graph:
   colors[node] = None

 // 2. Iterate through all nodes to handle disconnected graphs
 FOR each node in graph:
   // If a node is uncolored, it's part of a new component. Start BFS from it.
   IF colors[node] IS None:
     // Start a new BFS for this component
     queue = new Queue()

     colors[node] = 0  // Start coloring with color 0
     queue.enqueue(node)

     WHILE queue is not empty:
       u = queue.dequeue()

       FOR each neighbour v of u:
         IF colors[v] IS None:
           // If neighbour is uncolored, color it with the opposite color
           colors[v] = 1 - colors[u]
           queue.enqueue(v)
         ELSE IF colors[v] == colors[u]:
           // CONFLICT: An edge connects two nodes of the same color.
           // The graph contains an odd-length cycle.
           RETURN False // Not bipartite

 // 3. If the loops complete without any conflicts, the graph is bipartite
 RETURN True
```

Testing and Results

To verify the algorithm, we test it on one bipartite and one non-bipartite graph.

Test 1: Bipartite Graph (A Square)

A graph is bipartite if it has no odd-length cycles. A square is an even-length cycle (4-cycle), so it should be bipartite.

- Graph:
    1. Nodes: A, B, C, D
    2. Edges: (A,B), (B,C), (C,D), (D,A)
    3. Execution Trace:
    4. Start BFS at A. colors[A] is set to 0.
    5. Neighbors of A (B and D) are colored 1.
    6. Neighbors of B (C) and D (C) are colored 0.
    7. The algorithm completes without finding any two adjacent nodes with the same color.
- Expected Result: True
- Actual Result: The algorithm will return True. ✅

Test 2: Non-Bipartite Graph (A Triangle)

A triangle is a 3-cycle (an odd-length cycle), which is the classic example of a non-bipartite graph.

- Graph:
    1. Nodes: A, B, C
    2. Edges: (A,B), (B,C), (C,A)
    3. Execution Trace:
    4. Start BFS at A. colors[A] is set to 0.
    5. Neighbor B is colored 1.
    6. Neighbor C is also colored 1.
    7. When the algorithm processes node B, it looks at its neighbor C. It finds that colors[B] is 1 and colors[C] is also 1.
    8. Since adjacent nodes B and C have the same color, a conflict is detected.
- Expected Result: False
- Actual Result: The algorithm will return False. ✅

Task 4:

```
FUNCTION BellmanFord(graph, source):
```

```
// graph is represented by a list of vertices and a list of edges
// where each edge is a tuple: (u, v, weight)
 // 1. Initialize distances and predecessors
distances = {}
predecessors = {}
FOR each vertex in graph.vertices:
  distances[vertex] = Infinity
  predecessors[vertex] = None
 distances[source] = 0

// 2. Relax all edges |V| - 1 times
// A simple shortest path can have at most |V| - 1 edges
num_vertices = length(graph.vertices)
FOR i from 1 to num_vertices - 1:
  FOR each edge (u, v, weight) in graph.edges:
    // If a shorter path to v is found through u, update it
    IF distances[u] + weight < distances[v]:
      distances[v] = distances[u] + weight
      predecessors[v] = u

// 3. Check for negative-weight cycles
// If we can still relax an edge, a negative cycle must exist
FOR each edge (u, v, weight) in graph.edges:
  IF distances[u] + weight < distances[v]:
    // Negative cycle found!
    RETURN "Error: Graph contains a negative-weight cycle."

// 4. If no negative cycle, return the results
RETURN (distances, predecessors)
```

Test Cases and Demonstration

Here are two test cases that demonstrate the algorithm's functionality as required.

Test 1: Graph with Negative Weights (No Cycle)

This test demonstrates that the algorithm correctly computes shortest paths in a graph with negative edge weights but no negative-weight cycles.

- Graph Definition:
  - Vertices: {S, A, B, E}
  - Edges: (S, A, 4), (S, B, 2), (A, E, 3), (B, A, -3), (B, E, 1)
  - Function Call: BellmanFord(graph, source='S')
- Expected Result: The algorithm should successfully complete and return the shortest distances. The path from S to A should be S -> B -> A with a total weight of 2 + (-3) = -1. The returned distances should be:
  - S: 0
  - A: -1
  - B: 2
  - E: 1
- This demonstrates the correct handling of negative weights. ✅

Test 2: Graph with a Negative-Weight Cycle

This test demonstrates that the algorithm can successfully detect and report the presence of a negative-weight cycle.

- Graph Definition:
  - Vertices: {A, B, C}
  - Edges: (A, B, 2), (B, C, 3), (C, A, -6)
  - Function Call: BellmanFord(graph, source='A')
- Expected Result: The algorithm should detect that the cycle A -> B -> C -> A has a total weight of 2 + 3 + (-6) = -1. In the final check (step 3 of the pseudocode), it will find that the distances can still be reduced, triggering the cycle detection logic.
  The function should return the error message: "Error: Graph contains a negative-weight cycle." ⚠️
  This demonstrates that the cycle detection is working correctly. ✅

Task 5:

```
FUNCTION FloydWarshall(graph):
```

```
// graph is represented by an adjacency matrix with weights

// graph[i][j] is the weight of the edge from node i to node j

// Use Infinity for no direct edge, and 0 for diagonals (i to i)

 num_nodes = number of nodes in graph

 // 1. Initialize distance and next_node matrices

// 'dist' will store the shortest path lengths

// 'next_node' will be used to reconstruct the paths (for extra credit)

dist = copy(graph)

next_node = create_matrix(num_nodes, num_nodes)



FOR i from 0 to num_nodes - 1:

  FOR j from 0 to num_nodes - 1:

    IF i == j OR dist[i][j] IS NOT Infinity:

      next_node[i][j] = j

    ELSE:

      next_node[i][j] = None



// 2. Main DP loop to find all-pairs shortest paths

// Consider each node 'k' as a potential intermediate node

FOR k from 0 to num_nodes - 1:

  FOR i from 0 to num_nodes - 1:

    FOR j from 0 to num_nodes - 1:

      // If path i -> k -> j is shorter than the current i -> j path

      IF dist[i][k] + dist[k][j] < dist[i][j]:
        dist[i][j] = dist[i][k] + dist[k][j]

        // Update the path: to get from i to j, first go towards k

        next_node[i][j] = next_node[i][k]

 // 3. Detect negative-weight cycles

// A negative cycle exists if the distance from any node to itself is negative

FOR i from 0 to num_nodes - 1:

  IF dist[i][i] < 0:

    RETURN "Error: Negative-weight cycle detected."



// 4. Return the resulting matrices

RETURN (dist, next_node)
```

Main program logic and output

```
// Main execution block
// Example graph with 4 nodes (0, 1, 2, 3)
graph = [
 [0, 5, Infinity, 10],
 [Infinity, 0, 3, Infinity],
 [Infinity, Infinity, 0, 1],
 [Infinity, Infinity, Infinity, 0]
]


// 2. Run the Floyd-Warshall algorithm
result = FloydWarshall(graph)


// 3. Check for errors and print results
IF result is a string (contains an error message):
 PRINT result
ELSE:
 // Unpack the results
 distance_matrix, next_node_matrix = result


 // 4. Print the final distance matrix
 PRINT "Shortest Distance Matrix:"
 FOR each row in distance_matrix:
   PRINT row
  PRINT "\n----------------------------------------\n"
  // 5. (Extra Credit) Print all-pairs shortest paths

 PRINT "All-Pairs Shortest Paths:"

 num_nodes = length(graph)

 FOR i from 0 to num_nodes - 1:

   FOR j from 0 to num_nodes - 1:

     path = ReconstructPath(i, j, next_node_matrix)

     PRINT "Path from " + i + " to " + j + ": " + path + " (Cost: " +
distance_matrix[i][j] + ")"
```

Task 6:

```
FUNCTION JohnsonsAlgorithm(graph):
```

```
// graph is represented by a list of vertices and edges
 // 1. Create a new graph with an extra source node 's'
new_graph = copy(graph)
s = new_node()
FOR each vertex v in new_graph.vertices:
  add_edge(s, v, 0) to new_graph.edges
 // 2. Run Bellman-Ford from 's' to get re-weighting values (h)
// This step also detects negative cycles in the original graph.
h = BellmanFord(new_graph, source=s).distances
IF BellmanFord reported a negative cycle:
  RETURN "Error: Original graph contains a negative-weight cycle."


// 3. Re-weight the original graph to remove negative edges
reweighted_graph = copy(graph)
FOR each edge (u, v, weight) in reweighted_graph.edges:
  new_weight = weight + h[u] - h[v]
  update_edge_weight(u, v, new_weight)


// 4. Run Dijkstra from every node on the re-weighted graph
all_pairs_distances = create_matrix(num_nodes, num_nodes)
FOR each vertex u in graph.vertices:
  // d_prime are the shortest paths in the re-weighted graph
  d_prime = Dijkstra(reweighted_graph, source=u).distances
  // 5. Compute the final distances by reversing the re-weighting

  FOR each vertex v in graph.vertices:

    all_pairs_distances[u][v] = d_prime[v] - h[u] + h[v]



// 6. Return the final matrix of all-pairs shortest paths

RETURN all_pairs_distances
```

Performance Comparison: Johnson's vs. Floyd-Warshall

This comparison analyzes the time complexity of both algorithms, which is crucial for deciding which one to use. Let n be the number of vertices and m be the number of edges.

- **Floyd-Warshall:** Has a consistent time complexity of **O(n3)**. Its performance is the same regardless of whether the graph is sparse or dense. It's also very simple to implement.
- **Johnson's Algorithm:** Has a time complexity of **O(nm+n2logn)** (assuming Dijkstra is implemented with a binary heap). Its performance depends heavily on the density of the graph.

Sparse Graphs

A sparse graph is one where the number of edges m is much smaller than n2 (e.g., m is close to n).

- **Johnson's:** The complexity becomes roughly O(ncdotn+n2logn)=O(n2logn).
- **Floyd-Warshall:** Remains O(n3).

Conclusion: For sparse graphs, **Johnson's algorithm is significantly faster**. This is its primary advantage. 🚀

Dense Graphs

A dense graph is one where the number of edges m is close to n2.

- **Johnson's:** The complexity becomes roughly O(ncdotn2+n2logn)=O(n3).
- **Floyd-Warshall:** Remains O(n3).

Conclusion: For dense graphs, both algorithms have a similar asymptotic performance. However, **Floyd-Warshall is generally preferred** because its implementation is much simpler and it doesn't have the overhead of the multiple steps required by Johnson's algorithm. ⚖️