Algorithms: Floyd-Warshall

Learning objective: Students will derive an efficient algorithm for finding the shortest paths between all pairs of vertices in a directed, weighted graph.

1 Consider the *adjacency matrix* in Model 1: it describes a weighted directed graph. (You may wish to draw the graph if doing so helps you to more clearly visualize the graph.)

Examining the graph, list all pairs of vertices for which there is a path through an intermediate vertex that is cheaper than tracing the edge from the first to the second vertex. For example, it is cheaper to travel from v_1 to v_2 through intermediate vertex v_4 (total length: 2 ($v1$ to v_4) + 1 (v_4 to v_2) = 3) than directly from v_1 to v_2 (edge weight: 6).

Note that the graph is not symmetric; the shortest path from v_1 to v_2 need not be the same as the shortest path from v_2 to v_1.

Be sure to list the pair, the intermediate vertex, and the total cost of the path with the intermediate vertex. You should find a total of five pairs.

2 Find an instance where taking a path that goes through a second intermediate vertex is cheaper than both (a) the path going through just one intermediate vertex and (b) tracing the single edge that connects the starting and ending vertices. There is only one such instance.

Model 1: Adjacency Matrix

	v_1	v_2	v_3	v_4
v_1	-	8	7	2
v_2	3	-	3	6
v_3	8	3	-	2
v_4	6	1	5	-

Figure 1: An adjacency matrix for a weighted directed graph G. The value at $M[i, j]$ corresponds to the weight of edge (v_i, v_j). In other words, nodes on the left are originating vertices. Nodes on the top are destination vertices.

3 Inspired by your answers to Questions 1 and 2, fill in the base case and recursive cases of the algorithm below. The algorithm finds the shortest path distance between any pair of vertices for a graph with n vertices.

The parameter `g` refers to the graph being explored, and `g.edge_weight(i, j)` returns the weight of the edge that connects v_i to v_j in graph g. The `start` and `end` parameters represent the indices of the vertices between which we seek the shortest path. The k parameter represents the maximum vertex index under consideration as an intermediate node. When $k = 0$, no intermediate nodes will be considered; only the edge weight from `start` to `end` can be used. When $k = 1$, v_1 can be considered. When $k = 2$, v_1 and v_2 can be considered, and so forth.

```
shortest_path_distance(Graph g, int start, int end, int k)
  if k == 0
    return (                                              )
  else
    let distance_ignoring_vertex_k =
       shortest_path_distance(g,     ,     ,    )
    let distance_using_vertex_k =
       shortest_path_distance(g,     ,     ,    )
       + shortest_path_distance(g,     ,     ,    )
    if (                                             )
      return distance_ignoring_vertex_k
    else
      return distance_using_vertex_k
```

4 List every distinct base-case recursive call encountered when calling the algorithm on Model 1 with the call `shortest_path_distance(model_1, 1, 4, 4)`.

5 To prove the algorithm from Question 3 correctly finds the shortest path from `start` to `end`, we need to use *strong induction*[1] due to its multiple distinct recursive calls. What would all the base cases be for a correctness proof by strong induction?

6 How many base cases would there be?

[1] In weak induction, we assume that $P(k)$ is true in order to prove $P(k+1)$. In strong induction, we assume that our inductive hypothesis holds for *all* values preceding k. Said differently, we assume that each $P(i)$—from our base case up until $P(k)$—is true (e.g., $P(1), P(2), \ldots, P(k)$ all hold) in order to prove that $P(k+1)$ is true.

7 What would be the inductive hypothesis?

8 What would we need to prove in the inductive step?

9 Complete the proof by strong induction that this algorithm finds the shortest path from `start` to `end`.

10 Write a recurrence for the asymptotic time complexity of the algorithm you wrote in Question 3.

11 Estimate the asymptotic time complexity of your algorithm based on the recurrence from Question 10.

12 How might the algorithm (or adjacency matrix) from Question 3 be modified to handle missing edges?

13 If we were to rewrite the algorithm from Question 3 as a bottom-up dynamic programming algorithm (e.g., left-to-right, row-major order, etc.), how many dimensions would the table need to have? What would each dimension represent?

14 Write pseudocode for a bottom-up dynamic programming version of the algorithm. (This is known as the Floyd-Warshall algorithm.)

15 What is the asymptotic time complexity of the algorithm from Question 14?

16 How might we augment the dynamic programming table so that we can reconstruct the path between two vertices?

17 Under what circumstances is Dijkstra's algorithm preferable to the Floyd-Warshall algorithm? Why?

18 Under what circumstances is the Floyd-Warshall algorithm preferable to Dijkstra's algorithm? Why?