```java
import java.util.*;

public class MyCITS2200Project implements CITS2200Project {

    /*
     * Indexing by strings can be messy and inefficient, so we instead assign each
     * vertex a unique integer ID between in the range. This ID will serve as an
     * index into the adjacency list, allowing us to find a vertex's list of
     * neighbours in constant time. To allow us to convert back and forth between;
     * the string and integer representations of our vertices, we introduce a list
     * of strings that can be indexed efficiently by vertex ID, and a map from the
     * vertex URL to its ID.
     *
     * Create original adjacency list for the graph, and an additional transposed
     * adjacency list. Kosaraju's algorithm is based on the observation that the
     * SCCs in the original graph are the same as those in the transpose graph (that
     * is, the graph with all edges reversed).
     */

    /**
     * Allows us to lookup a page Page-URL by Page-ID
     */
    private final ArrayList<String> idToURL = new ArrayList<>();

    /**
     * Allows us to lookup a Page-ID by Page-URL
     */
    private final LinkedHashMap<String, Integer> urlToID = new LinkedHashMap<>();

    /**
     * Original adjacency list for the graph
     */
    private final ArrayList<List<Integer>> originalList = new ArrayList<>();

    /**
     * Transposed adjacency list for the graph
     */
    private final ArrayList<List<Integer>> transposedList = new ArrayList<>();

    /**
     * Simply add an entry to the adjacency list to represent the new edge.
     *
     * @param urlFrom From
     * @param urlTo   To
     */
    @Override
    public void addEdge(String urlFrom, String urlTo) {
        // Add vertices if necessary
        addVertex(urlFrom);
        addVertex(urlTo);

        // Add edges to both adjacency lists
        int from = urlToID.get(urlFrom), to = urlToID.get(urlTo);

        // Original order
        originalList.get(from).add(to);

        // Transposed order
        transposedList.get(to).add(from);
    }

    /**
     * Adding an edge to the graph requires us to first make sure both vertices
     * exist in the graph. The function checks if a vertex exists using our urlToID
     * map, and adds it to the graph if it does not.
     *
     * @param vertex URL
     */
    private void addVertex(String vertex) {
        if (!urlToID.containsKey(vertex)) {
            // Add for looking up
            idToURL.add(vertex);
            urlToID.put(vertex, urlToID.size());

            // Add for listing
            originalList.add(new ArrayList<>());
            transposedList.add(new ArrayList<>());
```

```java
        }
    }

    /******************************************************************************/
    // Question 1: Shortest Path
    /******************************************************************************/

    /**
     * When the BFS has finished, our array will hold the length of the shortest
     * path from our source to each vertex it can reach, or the original value of
     * the array if no such path exists.
     *
     * @param urlFrom From
     * @param urlTo   To
     * @return distance
     */
    @Override
    public int getShortestPath(String urlFrom, String urlTo) {
        // Running BFS through the ID of the URL
        int[] result = breadthFirstSearch(urlToID.get(urlFrom));

        // Return the relevant shortest path
        return result[urlToID.get(urlTo)];
    }

    /**
     * We can find the lengths of these shortest paths by performing a Breadth First
     * Search (BFS), which enumerates vertices according to the number of edges they
     * are away from our starting vertex. By maintaining an array of distances from
     * our starting vertex, we can fill this array in as we perform our BFS.
     *
     * @param source Starting point
     * @return Array of distances
     */
    private int[] breadthFirstSearch(int source) {
        // Distances from the source root to each vertex
        Queue<Integer> queue = new LinkedList<>();
        int[] visited = new int[idToURL.size()];

        // Mark all the vertices as not visited, -1 by default
        Arrays.fill(visited, -1);

        // The source root is set to 0
        visited[source] = 0;
        queue.add(source);

        // BFS Ordering
        while (!queue.isEmpty()) {
            // Retrieve and remove the head of this queue
            int current = queue.remove();
            for (int next : originalList.get(current)) {
                if (visited[next] == -1) {
                    // Add the distance to the Array
                    // +1 to negate the initial -1 fill value
                    visited[next] = visited[current] + 1;

                    // Add it to our order
                    queue.add(next);
                }

            }
        }
        return visited;
    }

    /******************************************************************************/
    // Question 2: Hamiltonian Path
    /******************************************************************************/

    /*
     * The editorial highlighted how bitshifting can be used to speed or code up,
     * this was a rather complicated topic and as such we relied heavily on other
     * reference implementations (report references). The output is seemingly
     * correct based on the small integer graph given as a sample.
     *
     * We can speed up our code by using arrays of primitives (it's likely to have
     * to better memory layout than a list of objects) and operating on bitmasks
     * directly.
```

```java
     *
     * Java's â€‹ Math.pow()â€‹ function is not constant time, but is rather logarithmic
     * in the power. This introduced an O (log n) factor that is not present when
     * using bit-shifts.
     */

    /**
     * The left operands value is moved left by the number of bits specified by the
     * right operand.
     *
     * @param source number of bits
     * @return result
     */
    private int leftShift(int source) {
        return (1 << source);
    }

    /**
     * Check if the bit is set.
     *
     * Binary AND Operator copies a bit to the result if it exists in both operands.
     *
     * @param left  operand
     * @param right leftShifted int
     * @return result
     */
    private int checkBitSet(int left, int right) {
        return (left & (1 << right));
    }

    /**
     * Sets the bit that corresponds to the right value.
     *
     * Binary OR Operator copies a bit if it exists in either operand.
     *
     * @param left  operand
     * @param right leftShifted int
     * @return result
     */
    private int checkBitCorresponds(int left, int right) {
        return (left | (1 << right));
    }

    /**
     * Implementation of the algorithm given by Bellman, Held, and Karp which uses
     * dynamic programming to check whether a Hamiltonian Path exists in a graph.
     *
     * @return Hamiltonian path or null
     */
    @Override
    public String[] getHamiltonianPath() {
        // Set the size of the graph and square of the size of the graph
        int graphSize = idToURL.size(), graphSizePow = leftShift(idToURL.size());

        // Because the graph is unweighted we can use a boolean array
        boolean[][] dpSet = new boolean[graphSizePow][graphSize];

        // Mark the subset containing only the vertices as true
        Arrays.fill(dpSet[graphSizePow - 1], true);

        // Iterate over the subsets of our graph
        iterateSubsets(dpSet, graphSize, graphSizePow);

        // Return empty or return our path
        return (Objects.requireNonNull(reconstructPath(dpSet, graphSize, graphSizePow))).toArray(new String[0]);
    }

    /**
     * Iterate over subsets of our graph. We can represent these subsets as a
     * bitset, using each binary digit in an integer to represent whether the
     * corresponding vertex is in the set or not, we can store the answer to each
     * question as it is computed, meaning we will never have to recompute an answer
     *
     * @param dpSet        Set to store results
     * @param graphSize    Size of the graph
     * @param graphSizePow Squared size of the graph
     */
```

```java
    private void iterateSubsets(boolean[][] dpSet, int graphSize, int graphSizePow) {
        // The loop iterates over all the subsets of the vertices
        // We subtract twice so that we do not fill the starting bit
        for (int mask = graphSizePow - 1 - 1; mask > 0; mask--) {
            // Check which of the vertices are present in subset
            for (int lastVertex = 0; lastVertex < graphSize; lastVertex++) {
                // Check if it is the last vertex present in the mask
                if (checkBitSet(mask, lastVertex) > 0) {
                    // For every lastVertex present in mask
                    for (int nextVertex : originalList.get(lastVertex)) {
                        // Present in mask and check for neighbours of last Vertex
                        if (checkBitSet(mask, nextVertex) == 0) {
                            // For every nextVertex check if cell is true or not
                            if (dpSet[checkBitCorresponds(mask, nextVertex)][nextVertex]) {
                                // Whether there is a path that visits each vertex in the subset
                                // exactly once and ends at nextVertex
                                dpSet[mask][lastVertex] = true;

                                // Stop iteration as we have found a path
                                break;
                            }
                        }
                    }
                }
            }
        }
    }

    /**
     * Iterate over the solutions from iterateSubsets() and reconsruct the
     * hamiltonian path. The corresponding URLs of path are then saved as a String
     * and returned if a path exists, if not it returns null.
     *
     * @param dpSet        Set to store results
     * @param graphSize    Size of the graph
     * @param graphSizePow Squared size of the graph
     * @return Resulting path
     */
    private ArrayList<String> reconstructPath(boolean[][] dpSet, int graphSize, int graphSizePow) {
        // Iterate over all the vertices
        for (int vertex = 0; vertex < graphSize; vertex++) {
            // Check if the cell is true or not
            if (dpSet[leftShift(vertex)][vertex]) {
                // Save the vertex value
                int currentVertex = vertex;

                // Store our String result
                ArrayList<String> result = new ArrayList<>();

                // Set the mask
                int mask = leftShift(vertex);

                // Add the URL of our starting vertex
                result.add(idToURL.get(currentVertex));

                // Iterate over the subsets
                while (mask != (graphSizePow - 1)) {
                    // For every currentVertex present in mask
                    for (int nextVertex : originalList.get(currentVertex)) {
                        // Present in mask and check for neighbours of nextVertex
                        if (checkBitSet(mask, nextVertex) == 0) {
                            // Check if the cell is true or not
                            if (dpSet[checkBitCorresponds(mask, nextVertex)][nextVertex]) {
                                // Set the mask
                                mask = checkBitCorresponds(mask, nextVertex);

                                // Overwrite our original currentVertex
                                currentVertex = nextVertex;

                                // Break out of the current iteration
                                break;
                            }
                        }
                    }
                    // Add the URL of our currentVertex to our result
                    result.add(idToURL.get(currentVertex));
                }
                // Return URL of the vertices in the path
```

```java
                return result;
            }
        }
        // If there is no such path return null
        return null;
    }

    /*******************************************************************************/
    // Question 3: Strongly Connected Components
    /*******************************************************************************/

    /**
     * Returns the set of vertices that can all reach each other. This implies that
     * a vertex belongs to exactly one SCC, which may even be just that vertex.
     *
     * @return The strongly connected component or components
     */
    @Override
    public String[][] getStronglyConnectedComponents() {
        // ArrayList to store our result
        ArrayList<Stack<Integer>> result = new ArrayList<>();

        // Execute the algorithm
        kosajaruAlgorithm(result);

        // Create 2D Array to store required format
        // This array can be jagged and does not need to be filled with null
        String[][] components = new String[result.size()][];

        // Return our formatted result
        return kosajaruResult(result, components);
    }

    /**
     * Formats Kosajaru's algorithm to the required result format.
     *
     * Partition the vertices of the graph into the SCCs that contain them.
     *
     * @param result     The ArrayList to store our result
     * @param components 2D Array for storing formatted result
     * @return The strongly connected components
     */
    private String[][] kosajaruResult(ArrayList<Stack<Integer>> result, String[][] components) {
        // Add the components
        for (int distinct = 0; distinct < result.size(); distinct++) {
            // Set our number
            components[distinct] = new String[result.get(distinct).size()];

            // Set the size
            int size = result.get(distinct).size();

            // Add the strongly connected components
            for (int connected = 0; connected < size; connected++) {
                // Set the mapped URL of our components
                components[distinct][connected] = idToURL.get(result.get(distinct).get(connected));
            }
        }

        // Return the url of our components
        return components;
    }

    /**
     * Performs Kosajaru's algorithm
     *
     * Compute the set of all vertices a vertex can reach by performing a DFS
     * starting at that vertex. Doing this is in both the original graph and the
     * transpose graph is sufficient to compute the SCC to which this vertex
     * belongs, Kosaraju's algorithm uses a property of the DFS order through the
     * original graph in order to ensure that the DFS through the transpose graph
     * only explores this intersection
     *
     * @param result Keep track of the result
     */
    private void kosajaruAlgorithm(ArrayList<Stack<Integer>> result) {
        // Marks all as not visited by default
        boolean[] visited = new boolean[idToURL.size()];

```

```java
        // Create a stack for the order
        Stack<Integer> order = new Stack<>();

        // Iterate through the original list
        for (int i = 0; i < idToURL.size(); i++) {
            if (!visited[i]) {
                // Fill the position order
                depthFirstSearch(i, visited, true, order);
            }
        }

        // Mark as not visited by default
        visited = new boolean[idToURL.size()];

        // Iterate through the transposed list
        while (!order.isEmpty()) {
            int current = order.pop();
            if (!visited[current]) {
                Stack<Integer> component = new Stack<>();
                // Find the SCC
                depthFirstSearch(current, visited, false, component);
                result.add(component);
            }
        }
    }

    /**
     * DFS through the transpose graph starting from the last vertex in post-order,
     * and any vertices it visits must be part of its SCC, as any vertex that is
     * earlier in the post-order that our starting vertex can reach must therefore
     * have been able to reach and be reached from this starting vertex in the
     * original graph. We can then DFS again from the next highest vertex in the
     * post-order that is not yet visited in order to find all its SCC, and so on
     * until we have found all the SCCs.
     *
     * @param current  Starting position
     * @param visited  Boolean Array to keep track of visits
     * @param original Original or transposed list
     * @param stack    Stack to hold result
     */
    private void depthFirstSearch(int current, boolean[] visited, boolean original, Stack<Integer> stack) {
        // Mark current as visited
        visited[current] = true;

        // Pick the list
        ArrayList<List<Integer>> currentList = original ? originalList : transposedList;

        // DFS the required list
        for (int next : currentList.get(current)) {
            if (!visited[next]) {
                depthFirstSearch(next, visited, original, stack);
            }
        }

        // Add to results
        stack.add(current);
    }

    /*******************************************************************************/
    // Question 4: Graph Centers
    /*******************************************************************************/

    /**
     * Using BFS for computing the lengths of the shortest paths to each vertex
     *
     * @return The center or centers
     */
    @Override
    public String[] getCenters() {
        return centers().toArray(new String[0]);
    }

    /**
     * The radius of a graph is the smallest eccentricity of any vertex.
     *
     * A center is a vertex whose eccentricity is the radius.
     *
     * The simplest way to find the center of the graph is to find the all-pairs
```

```java
     * shortest paths and then picking the vertex where the maximum distance is the
     * smallest.
     *
     * @return List of center or centers
     */
    private ArrayList<String> centers() {
        // Resulting center or centers
        ArrayList<String> result = new ArrayList<>();

        // Storing the minimum eccentricity
        int minimum = idToURL.size(), url = 0, urlSize = idToURL.size();

        // Check each URL
        while (url < urlSize) {
            // Set default eccentricity for iteration
            int eccentricity = -1;

            // Using BFS for computing the lengths of the shortest paths to each vertex
            for (int vertex : breadthFirstSearch(url)) {
                // Look for the most distant vertex from the URL
                if (eccentricity < vertex) {
                    // Check for -1 because our BFS returns -1 by default
                    // Ignore error highlighting
                    if (vertex > -1) {
                        eccentricity = vertex;
                    }
                }
            }

            // Vertices with minimum eccentricity
            if (eccentricity < minimum) {
                // Check if it isn't the default set value
                if (eccentricity > -1) {
                    // Update the minimum
                    minimum = eccentricity;

                    // Reset the results
                    result = new ArrayList<>();
                }
            }

            // Add to the center if it's the same
            if (eccentricity == minimum) {
                result.add(idToURL.get(url));
            }

            url++;
        }

        // Return our ArrayList with the result
        return result;
    }

}
```