Solutions

Shreyas Athreya Venkatesh

3rd February 2025

Question 1

Imagine a graph $G = (V, E)$. Furthermore, there is a mapping $L_E : E \rightarrow L$ from edges to a set of *labels* L. There is a designated initial vertex $v_0 \in V$, and a set of *final* vertices $V_f \subseteq V$. Output the labels from the shortest path from v_0 to any $v_f \in V_f$. Please code this solution up.

Assumptions:

- The graph is unweighted/unit weight.
- Vertices are unique (no repetition of vertex IDs).

The **solution** involves:

- Performing a Breadth-First Search (BFS) traversal from the source to all destinations to find all valid paths.
- Identifying the shortest path among all the valid paths.

Algorithm 1: Path Finder / Breadth-first Search

Input : Graph with nodes and edges, initial vertex v_0, final vertices V_f
Output: Paths from v_0 to any $v_f \in V_f$

Clear `path_to_final_vertices`;
Initialize queue q with $(v_0, \{\})$;
while q ***not empty*** **do**
    $(current\_node, path) \leftarrow q.\texttt{front}()$;
    $q.\texttt{pop}()$;
    if $current\_node \in V_f$ **then**
        Add *path* to `path_to_final_vertices`;
    end
    foreach *neighbor connected to current_node* **do**
        if *neighbor not visited or is a final vertex* **then**
            Add the edge between *current_node* and neighbor to the current path;
            Add the neighbor and updated path to the queue for further exploration;
        end
    end
end

Breadth-first Search Explanation

- Construct the graph with the given initial conditions.

- Push {node, path_traversed} pairs into a queue starting from the initial node.
- For each element in the queue:
  - Check if the current node is a final vertex; if yes, add the path to `path_to_final_vertices`.
  - Track visited nodes to avoid cycles.
  - For unvisited neighbors or final vertices, find the connecting edge, update the path, and push to the queue.

Algorithm 2: Find Shortest Paths

```
Input  : Paths from v_0 to final vertices in path_to_final_vertices
Output: Shortest paths in minimum_paths
Set shortest_size to size of the first path in path_to_final_vertices;
Add the first path to minimum_paths;
for each path in path_to_final_vertices do
    if path size < shortest_size then
        Update shortest_size;
        Add path to minimum_paths;
    end
    else if path size == shortest_size then
        Add path to minimum_paths;
    end
end
```

Shortest Path Explanation

- Iterate through all valid paths collected from BFS.
- Initialize with first element of the valid paths container `minimum_paths`.
- For each path:
  - If the path is shorter, update `minimum_paths`.
  - If the path length is equal to the shortest, add it to `minimum_paths`.

Question 2

Imagine the same question as Question 1, but you are given multiple graphs $G_1, \ldots, G_n$. If $\mathcal{P}_i$ is the labels from all paths from initial to final for graph G_i, output the shortest labels in $\mathcal{P}_0 \cap \ldots \cap \mathcal{P}_n$. In other words, output the shortest labels that correspond to a valid path in every graph. Try to find a way to reduce this problem to Question 1. Please code this solution up.

This question has been solved with **two solutions**, involving:

1. **String Matching:** (Based on Intuition)
   Extract label sequences from all valid paths in each graph, find their intersection, and select the shortest common sequence.

2. **Constructing a New Graph by Common Edges:** (Based on the Hint)
   The key observation is that edge labels that are common to all graphs must be present in the resultant graph Then build a new graph using edges common to all graphs, find all possible paths, and choose the shortest one.

Algorithm 3: String Intersection of All Paths

```
Input  : List of sets all_paths
Output: List of common elements
Set common_label_sequences to the first element in all_paths;
for each set in all_paths do
  | Update common_label_sequences by taking the intersection with the current set;
end
return common_label_sequences;
Find and return the shortest string among common_label_sequences;
```

[Method 1] String Matching Approach

- Extract label sequences from all valid paths in each graph, treating them as strings.
- Convert these sequences into sets to represent the unique paths for each graph.
- Find the intersection of these sets to identify common path sequences across all graphs.
- Select the shortest string from the intersected sequences as the final solution.

Algorithm 4: Creating Common Graph

```
Input  : List of graphs graphList
Output: Intersection graph newGraph
Extract common label from each graph ;
Map common labels to their corresponding node pairs using the first graph to create the new graph
  label_to_nodes;
Initialize unique_nodes and edges_with_labels from label_to_nodes;
//these containers are created to match the constructor of the Graph class for initilization of the
  //new graph
for each common label do
    Add corresponding nodes to unique_nodes;
    Add edge with label to edges_with_labels;
end
Create newGraph using unique_nodes, edges_with_labels, the source node, and final vertices from
  the first graph;
return newGraph;
```

[Method 2] Constructing a New Graph by Common Edges/Labels

- Identify all edge labels from each graph.
- Find the intersection of edge labels common to all graphs.
- Map the common labels to their corresponding nodes using one of the graphs.
- Create a new graph using these common edges and associated nodes.
- Perform a path search on the new graph to find all valid paths.
- Select the shortest path from the valid paths.

Assumptions:

- The graph is unweighted/unit weight.
- Vertices are unique (no repetition of vertex IDs).
- Common edges/labels contain the same vertex pair across all graphs.
- Any edge that is common across all graphs should be present the new graph

Question 3

Imagine the same question as Question 1, but we also have a commutative semiring R where each edge is labelled by an element of the semiring $R_E : E \to R$. Find a path $e_1 \dots e_n$ from v_0 to some $v_f \in V_f$ such that

$$e_1 \cdot \dots \cdot e_n \neq 0.$$

The semiring should satisfy the property that $e + e = e$.
Answer)

- The question now maps the earlier label to a new component of the commutative semiring.
- We proceed by modifying the previous search algorithm, by maintaining the product of new labels and checking the condition for non-zero product

Algorithm 5: Modified* Path Finder / Breadth-first Search

```
Input  : Graph with nodes and edges, initial vertex v_0, final vertices V_f
Output: Paths from v_0 to any v_f ∈ V_f
Initialize queue q with (v_0, {}, 1, 0);
//where 1 is the product and 0 is the initial sum of semiring labels
while q not empty do
    (current_node, path, product, sum) ← q.front();
    q.pop();
    if current_node ∈ V_f then
        Add (path, product, sum) to path_to_final_vertices;
    end
    foreach neighbor connected to current_node do
        if neighbor not visited or is a final vertex then
            Add the edge between current_node and neighbor to the current path;
            Calculate the new product for the path;
            Update the running sum of semiring labels:
            if running sum == semiring label then
                //Apply idempotent law: a + a = a;
                do nothing
            end
            else
                Add the new label to the running sum;
            end
            if the new product ≠ 0 then
                Add the neighbor, updated path, updated product, and updated sum to the queue for
                  further exploration;
            end
        end
    end
end
```

- Construct the graph with the given initial conditions.
- Push {`node, path_traversed, product = 1, sum = 0`} pairs into a queue starting from the initial node, where:
  - `product` represents the cumulative semiring product of labels along the path.
  - `sum` represents the cumulative sum of semiring labels along the path.

- For each element in the queue:
  - Check if the current node is a final vertex; if yes, add the `(path, product, sum)` to `path_to_final_vertices`.
  - Track visited nodes to avoid cycles.
  - For unvisited neighbors or final vertices:
    - Find the connecting edge and update the path.
    - Calculate the new product for the path.
    - Update the running sum of semiring labels:
      - If the running sum already contains the semiring label, apply the idempotent law $a+a=a$ (i.e., do nothing).
      - Otherwise, add the new semiring label to the running sum.
    - If the new product is non-zero, add the neighbor, updated path, updated product, and updated sum to the queue for further exploration.

Algorithm 6: Modified* Find Shortest Paths

```
Input  : Paths from v_0 to final vertices in pair{path_to_final_vertices, product, sum}
Output: Shortest paths in minimum_paths
Set shortest_size to the size of the first path in pair{path_to_final_vertices, product, sum};
Add the first (product, path, sum) pair to minimum_paths;
for each (product, path, sum) in pair{path_to_final_vertices, product, sum} do
    if path size < shortest_size then
        Update shortest_size;
        Clear minimum_paths;
        Add (product, path, sum) to minimum_paths;
    end
    else if path size == shortest_size then
        if running sum differs from existing paths then
            Consider the trade-off of the running_sum based on the mapping function;
            Add (product, path, sum) to minimum_paths;
        end
        else
            Apply idempotent law: a + a = a;
        end
    end
end
```

Note:

- If the paths are the same but the running products or sums differ, we need to consider a trade-off. The path selection should depend on the mapping function chosen, which might prioritize either shorter paths, higher semiring products, or sums based on the problem context.
- Another property to consider is the idempotent behavior when performing the addition of semiring labels in the shortest paths. If labels repeat, apply $a+a=a$.

- Iterate through all valid paths, their corresponding products, and running sums collected from BFS.
- Initialize the shortest path length, product, and sum as the first element of `minimum_paths`, which now stores `(product, path, sum)` triples.
- For each `(product, path, sum)` triple:
  - If the path is shorter, update `minimum_paths` with the new `(product, path, sum)`.
  - If the path length is equal to the shortest, check the running sum:
    * If the running sum or product differs, consider it based on the mapping function.
    * If the sums are the same, apply the idempotent law $a + a = a$.
    * Add the `(product, path, sum)` to `minimum_paths` if it meets the criteria.

Question 4

Imagine the same as Question 3, but now we want to find a shared path, like we did in Question 2. However, we also want the product of *all* the semiring labels to not be zero. In other words, for each edge labelling, we will do a full product of all elements.

The semiring should satisfy the property that $e + e = e$.

Answer:

- In Question 4, we extend the intersection model from Question 2 by filtering edges based on semiring constraints. We then construct a new graph and apply the semiring-aware path-finding algorithm from Question 3.

Algorithm 7: Modified* Creating Common Graph

```
Input  : List of graphs graphList
Output: Intersection graph newGraph
Extract label sets from each graph and find their intersection to get common labels;
for each common label do
    Check if the product of the semiring labels across all graphs is non-zero;
    if product is non-zero then
        Initialize semiring_sum to 0;
        for each semiring label associated with the common edge do
            if label already exists in semiring_sum then
                Apply idempotent law: a + a = a (do nothing);
            end
            else
                Add label to semiring_sum;
            end
        end
    end
    Map common labels to their corresponding node pairs using the first graph;
end
Initialize unique_nodes and edges_with_labels from label_to_nodes;
//These containers are required for initializing the new graph with the Graph class constructor
for each common label do
    Add corresponding nodes to unique_nodes;
    Add edge with label and computed semiring_sum to edges_with_labels;
end
Create newGraph using unique_nodes, edges_with_labels, the source node, and final vertices from
  the first graph;
return newGraph;
```

- Extract labels from both graphs and calculate their intersection to identify common edges.
- For each common label, verify whether the product of semiring labels from both graphs is non-zero.
- If a non-zero product prevails, compute the sum of semiring labels.
- Construct the new graph using the filtered edges, respective semiring sums, and node mappings from the first graph.

Note

For Questions 3 and 4, I have not included any code, as they do not explicitly require coding the solution.