Updated Solution Q2 & Q4

Shreyas Athreya Venkatesh

9th February 2025

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.

Answer)

- The solution works by creating an intersection graph
- Traverse all the graphs simultaneously, level by level using breadth-first search, starting with the set of all initial nodes.
- Then apply a shortest path from Q3 on the intersection graph to find the shortest path valid in all graphs

Algorithm 1: Creating Common Graph

```
Input: List of graphs graph_list
Initialize:
initial_node ← G_1.initial_vertex;
current_nodes ← {v_i | v_i ∈ G_i, ∀G_i ∈ graph_list} ;        // Tracks all nodes of a level
Queue q ← (current_nodes, initial_node);
final_vertices ← {} ;                      // Tracks final vertices of the common graph
visited ← {} ;                             // Tracks all visited node combinations
destination_reached ← false ;              // Tracks if any valid path is found
while q not empty do
    (current_nodes, curr_int_node) ← q.pop();
    if ∃ vertex ∈ current_nodes[0], vertex ∈ final_vertices(G_1) then
        Add curr_node to final_vertices;
    end
    common_labels ← findCommonEdgeLabels(current_nodes) ;   // Find common labels and
     corresponding next-level nodes at the given level
    if common_labels is empty and destination_reached = false then
        return nullptr;
    end
    else if common_labels is empty and destination_reached = true then
        break;
    end
    for each (label, next_nodes) in common_labels do
        if next_nodes is visited then
            continue;
        end
        Mark current_nodes as visited;
        Create new_node ;                  // New node in common graph
        Create edge with label ;           // New edge in common graph
        Enqueue (next_nodes, new_node);
    end
end
Create new graph ;                         // initialize all remaining data members
Apply shortest path on the new graph;
return shortest path that is valid in all graphs;
```

- Set the initial node from G_1, prepare queues for nodes, final vertices, visited nodes, and a flag for destination reachability.
- Use a queue to process nodes level-by-level, checking for final vertices in G_1.
- Identify common labels across all graphs.
- Create new nodes and edges in the intersection graph,
- Mark visited nodes to avoid cycles.
- Apply the shortest path algorithm on the intersection graph.

Assumptions:

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

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 2: Modified* Creating Common Graph

Input: List of graphs *graph_list*

Initialize:

$initial\_node \leftarrow G_1.initial\_vertex$;

$nodes\_in\_queue \leftarrow \{v_i \mid v_i \in G_i, \forall G_i \in graph\_list\}$; `// Tracks all nodes of a level`

Queue $q \leftarrow (nodes\_in\_queue, initial\_node)$;

$final\_vertices \leftarrow \{\}$; `// Tracks final vertices of the common graph`

$visited \leftarrow \{\}$; `// Tracks all visited node combinations`

$destination\_reached \leftarrow$ `false` ; `// Tracks if any valid path is found`

Mark *nodes_in_queue* as visited;

while *q not empty* **do**

    $(current\_nodes, curr\_int\_node) \leftarrow q.pop()$;

    if $\exists\, v \in current\_nodes[0],\ v \in final\_vertices(G_1)$ **then**

        Add *curr_int_node* to *final_vertices*;

    end

    $common\_labels \leftarrow findCommonEdgeLabels(current\_nodes)$; `// Find common labels and corresponding next-level nodes at the given level`

    if *common_labels is empty* ***and*** $destination\_reached = false$ **then**

        return `nullptr`;

    end

    else if *common_labels is empty* ***and*** $destination\_reached = true$ **then**

        break;

    end

    for *each* $(label, next\_nodes)$ *in common_labels* **do**

        if *next_nodes is visited* **then**

            continue;

        end

        if $\prod_{\forall edge \in label} semiring\_label \neq 0$ **then**

            Compute the $\sum_{\forall\, \text{edge} \in \text{label}}$ semiring_sum for each label, **such that** the Idempotent Law is maintained; i.e., **if** $semiring\_sum_{current} = semiring\_sum_{running}$ **then**

                do nothing;

            end

            else

                add to $\text{semiring\_sum}_{\text{running}}$;

            end

            Mark *current_nodes* as visited;

            Create *new_node* ; `// New node in common graph`

            Create edge with *label* && *semiring_sum* ; `// New edge in common graph`

            Enqueue $(next\_nodes, new\_node)$;

        end

    end

end

Create new graph ; `// initialize all remaining data members`

Apply shortest path on the new graph;

return shortest path that is valid in all graphs;

- 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.

- Apply semiring-aware shortest path algorithm from Question 3

Note

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