Bellman-Ford - Lab Reflection
Ryan Blocker
April 5th, 2024
CS320

Code: This was the `main.py` code I uploaded to Zybooks. My code was never able to pass the sanity test for some reason after one time. It kept saying I had the wrong function name however I couldn't find the error. This is the first lab where the Zybooks tests fully stumped me even though my code worked perfectly when I tested it. I have provided my test code below but feel free to test it yourself.

```
def bellman_ford(graph, start, end):
    if graph is None or start is None or end is None:
        return []
    if not graph.get_vertex(start) or not graph.get_vertex(end):
        return []
    if len(graph.vertices()) == 1:
        return [start]

    distances = {vertex.name: float('inf') for vertex in graph.vertices()}
    predecessors = {}
    distances[start] = 0

    for vertex in range(len(graph.vertices()) - 1):
        for edge in graph.edges():
            if edge.get_value() <= 0:
                return []
            u = edge.tail().name
            v = edge.head().name
            weight = edge.get_value()

            if distances[u] + weight < distances[v]:
                distances[v] = distances[u] + weight
                predecessors[v] = u

    if start == end:
        return [start]

    path = []
    current_vertex = end
    while current_vertex is not None:
        path.append(current_vertex)
        if current_vertex == start:
            break
        current_vertex = predecessors[current_vertex]
```

```
   if path[-1] != start:
       return []

   return path[::-1]
```

Testing: Here is my test code I used to test my file below:

```
from main import *
from edgegraph import *

#large graph test
graph = GraphEL()
vertices = ['A', 'S', 'E', 'D', 'C', 'B']
for vertex in vertices:
   graph.add_vertex(VertexEL(vertex))

edges = [
   ('e1', 'A', 'S', 10),
   ('e2', 'A', 'E', 8),
   ('e3', 'E', 'D', 1),
   ('e4', 'D', 'C', 1),
   ('e5', 'C', 'B', 2),
   ('e6', 'S', 'C', 2),
   ('e7', 'B', 'S', 1),
   ('e8', 'D', 'S', 4),
]

for edge_id, tail, head, weight in edges:
   edge = EdgeEL(edge_id, graph.get_vertex(tail),
graph.get_vertex(head))
   edge.set_value(weight)
   graph.add_edge(edge)
start_vertex = 'A'
end_vertex = 'C'
path = bellman_ford(graph, start_vertex, end_vertex)
print(f"Shortest path from {start_vertex} to {end_vertex}: {path}")

# test with only one vertices
graph = GraphEL()
vertices = ['A']
for vertex in vertices:
   graph.add_vertex(VertexEL(vertex))

edges = [
   ('e1', 'A', 'A', 1)
```

```
]

for edge_id, tail, head, weight in edges:
   edge = EdgeEL(edge_id, graph.get_vertex(tail),
graph.get_vertex(head))
   edge.set_value(weight)
   graph.add_edge(edge)
start_vertex = 'A'
end_vertex = 'A'
path = bellman_ford(graph, start_vertex, end_vertex)
print(f"Shortest path from {start_vertex} to {end_vertex}: {path}")
```

Output: The output from this test file on my end shows this:

```
Shortest path from A to C: ['A', 'E', 'D', 'C']
Shortest path from A to A: ['A']
```

Approach: My code failed the online tests in each submission. However, all of the submissions were based on the pseudo code from the textbook we were given. I start by testing edge cases, checking if the graph has a non-start or non-end, and if the start or end is not in the graph. If there is only one vertex, I return the start vertex in a list. Then, following the pseudo code, I initialize a distances list with every value set to infinity. I also create a predecessors dictionary and initialize start as the current vertex in the distances list.

In the relaxation loop, I iterate through every vertex in the graph. In the sub for loop, I iterate through every edge in the graph. First, I check if the edge is less than zero, as guided by the lab guidelines. If it is, I return an empty list. Then, I perform the relaxation check by comparing the current path with the new shorter path and updating the predecessors dictionary accordingly. If the check is equal, I return the start and its own list output. To get the proper output, I create a path list and a vertex that I set as the end. Then, I loop through the path, appending the current vertex while going backwards, and finally return the path.

Modularity: I have sectioned off the algorithm well. The first half handles edge cases, and then I move into the relaxation portion of the algorithm. Finally, I focus on returning the output in the proper format based on the lab guidelines. For increased readability, I set the variable names u, v, and weight to the appropriate edge methods to obtain their values for the relaxation if statement.

Big-O Complexity: The Bellman-Ford function you provided has a time complexity of O(|V| * |E|), where |V| is the number of vertices and |E| is the number of edges in the graph. This is due to the nested loops, with the outer loop iterating |V| - 1 times and the inner loop iterating over all edges for each outer iteration.

Implementation of Bellman-Ford: My function embodies the Bellman-Ford algorithm by initializing distances, performing edge relaxation to progressively find shorter paths, and attempting to reconstruct the shortest path. It iterates through all edges |V| - 1 times, updating distances to reflect the shortest paths found, consistent with Bellman-Ford's methodology. However, it modifies the standard algorithm by including an early termination for non-positive edge weights and lacks explicit negative cycle detection.