Austin Youngren

CS 320-001

Lab 7: Bellman-Ford Reflection

04/05/2024

Algorithm

When constructing the code for my algorithm, I chose to take an iterative approach because it seemed the more intuitive path. Due to this being a newer concept, and not wanting to accidentally implement a different algorithm, I referenced the Zybook pseudocode for the core of the algorithm. Additionally, given the workload I have in other classes, taking the path of least resistance seemed the best option to ensure the completion of all my assignments.

With the Zybook pseudocode, I was able to implement the core of the algorithm with ease, though I ran into larger issues when figuring out how that code returned the shortest path of ordered vertices. My subroutine, get_shortest_path(), went through many versions before properly returning the correct path. Many of my issues stemmed from one problem: having two equivalent edge values who shared a vertex as a tail. This created duplicated vertices within the shortest_path list. After a few versions, I realized I should be storing the indices of the vertex locations that consist of the shortest path rather than the vertices themselves during the core routine's execution. After making this change, the subroutine went through a few more versions before I could properly return a list without duplicates from get_shortest_path().

In the implementation of the algorithm I created four subroutines: check_cases(graph, start, end), get_graph_details(graph), get_edge_details(edge, vertices), and get_shortest_path(end, vertices, previous_verts).

check_cases() aids with code cleanliness and its function is to take the algorithm inputs 'graph', 'start', and 'end' as parameters and checks to see if they are valid inputs; checking if 'graph', 'end', and 'start' are not None and checking that 'start' and 'end' are vertices within the input graph. If any of them are not valid inputs, the subroutine returns True, otherwise the subroutine returns False.

get_graph_details() takes the input graph as a parameter and uses its information to create lists that will help with the implementation of the Bellman-Ford algorithm. It is largely a subroutine to aid in code cleanliness, too. Within the subroutine I create the variables: 'num_verts', which is assigned the value of the total number of vertices in the graph; the variable 'distances', which is a list of equal length to the number of vertices in the graph and contains the float value of infinity for each element; the variable 'vertices', which is a list containing all vertices of the graph; and the variable previous_verts, which is a list of equal length to the number of vertices in the graph, where each element contains the value None for each element.

get_graph_details() is a subroutine that takes an edge and the list, 'vertices,' as parameters. This subroutine is mainly used for code cleanliness. The subroutine first collects the two vertices connected to the input parameter, edge, and assigns the vertices to the variables 'u', the tail vertex of edge, and 'z', the head vertex of the edge. The indices associated with these two vertices, in the list 'vertices', are then found and assigned to the variables u_ind and z_ind. After, the edge weight is extracted and assigned to the variable edge_val. u_ind, z_ind, and edge_val are then returned to the caller routine.

Before explaining the get_shortest_path(), it would be best to cover the core routine, bellman-ford(). The start of this routine calls check_cases() in a conditional to check if the inputs are valid. If any of them are not valid, the routine will return an empty list. After the check, the routine calls get_graph_details() and assigns the returned values to the variables num_verts, 'distances', 'vertices', and previous_verts. Then, I find the associated index of 'start' within the list of vertices and assign the index value to the variable v_ind. v_ind is used to access the associated element in the list 'distances' which is assigned the value of zero to show there is 0 distance traveled at the start of the path.

The implementation then enters the outer for-loop. The loop iterates for the number of vertices in the graph minus one before entering the inner for-loop, which iterates for each edge outgoing from the current vertex. In a single iteration of the inner for-loop, the subroutine get_edge_details() is called and it's returned values are assigned to the variables u_ind, z_ind, and edge_val. If edge_val, the weight value of the edge, is zero or negative, the routine will return an empty list to the caller function. If not, the routine continues to another conditional which checks if the previous weight (distance[u_ind]), plus the current weight/distance (edge_val), is less than the weight of the alternate path (distances[z_ind]). For the first path option of every vertex, the value of distances[z_ind] will be infinity, so this conditional will always be true for such iterations. This conditional will only execute again for the same current vertex if a shorter distance is found. If the distance of the next edge is not equal to or shorter than the current distance, the conditional will be skipped, allowing the start of another loop cycle. Within the conditional, I assign the previous weight plus the current weight to distances[z_ind] to track the distance covered from start to the current vertex. Then, the index of the tail vertex within the vertices list, is stored in previous_verts[z_ind].

Once the algorithm leaves the loop structure, it executes the subroutine get_shortest_path() within the return statement. The subroutine get_shortest_path() takes the variables 'end', 'vertices', and 'previous_verts' as input parameters and is used to collect the vertices of the shortest path after the execution of the Bellman-Ford algorithm. In the subroutine an empty list, named shortest_path, is created to collect the vertices, and the variable, 'index,' is assigned the index location of 'end' in the list, 'vertices.' In the loop, the variable 'index' is used to access vertices in the vertices list to append them to shortest_path. After appending a vertex, 'index' is assigned the index value stored in previous_verts, which is the index of the next vertex in the shortest path. This continues for each value in previous_verts until the value in previous_verts[0], None, is assigned to 'index'. Because we collect these by backtracking from 'end,' we reverse the list, so the vertices are in start-to-end order. The subroutine then returns shortest_path to bellman_ford(), then bellman_ford returns that path to the original caller.