Austin Youngren

Lab 6: Breadth-First Reflection

CS 320-001

03/29/2024

Algorithm, Approach, & Complexity

For my algorithm I chose to take the iterative approach. During my time at Front Range Community College, we built breadth-first search algorithms many times for trees and graphs. So, the understanding and coding the basic approach for a breadth first search was quite intuitive for me. I thought about attempting a recursive approach, similar to previous assignments but ran into to many blockades when creating the pseudocode.

My first approach functioned similarly to the final product of my algorithm, but it utilized the GraphEL function adjacent_vertices() instead of the incident() function. My clouded thought process grouped the two functions to be the same thing because I was visualizing a simple graph. Though functionable, this initial approach presented a problem concerning the assignment's required algorithm complexity, as it ran at O(n^2) rather than O(n+m). After further familiarizing myself with the edgegraph.py file, I decided to adjust the algorithm to use the GraphEL class function, incident(), for the inner-loop rather than adjacent_indices(). After the implementation of incident, and a few adjustments to the existing code, I was able to achieve the complexity of O(n+m) with fewer lines of code.

In the creation of my algorithm, I built one subroutine to aid in the bfs() functionality. The subroutine is check_cases(), which takes the variable 'graph' and 'start' as parameters. The primary function for check_cases() is to ensure that the passed parameters are valid inputs for algorithm execution. check_cases() will return a Boolean value, True, if either of the parameter variables, 'graph' or 'start' are assigned the value of None. Additionally, the subroutine would return True if the vertex assigned to 'start' is not one of the vertices assigned to 'graph'. This check also ensures that graph is not empty, as 'start' will never be a vertex in an empty graph. If both 'graph' and 'start' are valid input variables, check_cases() will return False.

The bfs() routine starts by calling check_cases() within a conditional. If check_cases() returns True, bfs() will return an empty list to its caller. Otherwise, bfs() will begin the breadth first search of the graph by creating three empty lists: 'ordered', 'queue', and 'visited'. The starting vertex, 'start' is then appended to both the 'queue' and 'visited' list before entering the outer while-loop. 'ordered' will serve as the list containing the ordered vertices that is to be returned to the caller function, 'queue' will serve as the list containing the vertices that are to be evaluated, and 'visited' will serve as the list containing vertices that have been passed during our pathing of the graph.

The outer while-loop will continue to iterate for as long as the list 'queue' is not empty. Within the while-loop, the first element of 'queue' is removed using pop(0), which ensures the removal of the oldest vertex within the queue. Then, that vertex element is assigned to the variable 'current'. 'current' is then appended to 'ordered' before entering the inner for-loop.

The inner for-loop iterates for each incident edge of the vertex, 'current.' The two ends associated with the current edge of an iteration are assigned to the variable 'neighbors'. A conditional is then used to assign the vertex that is not 'current' to the variable 'neighbor'. Afterward, another conditional check if the vertex neighbor has been visited prior. If it has not been visited, it is then appended to both 'visited'

and ‘queue’ before exiting the inner for-loop. Once the algorithm exits the inner for-loop, the process will repeat for all remaining vertices in the graph. Then, once the algorithm finishes with the evaluation of all vertices, the list, ‘ordered’, is returned to bfs()’s caller function.