Austin Youngren

CS-320-001

Lab 4: Longest Path in a Torus

02/29/2024

Algorithm

The algorithm I created for finding the longest path in a torus utilizes iteration and recursion to obtain a solution. To help with code cleanliness, bug reduction, bug identification, and reducing repeated code blocks, I created six subroutines: is_not_list(elem), get_rows_and_cols(torus), get_num_elems(torus), get_neighbors(torus, i, j), get_all_trails(torus, i, j, ind_matrix, v_indices), and get_longest_trail(torus). is_not_list(elem) is used to ensure that the torus received is a 2-D array by checking if the first row of the torus is a list, where elem is most commonly torus[0]. get_rows_and_cols(torus) obtains and returns the number of rows and columns in the torus as two variables, then get_num_elems(torus) is used to obtain the total number of elements in the torus; these two subroutines serve to make the code clean in other functions since these values are needed frequently.

The callee routine of the algorithm, longest_path, serves as a security checkpoint for the received torus. Here, the torus is checked for the value of None, if it is an empty list, if the dimensions of the torus are correct, and if the torus has at least two elements before calling get_longest_path(). If any of these checks are True, an empty list is returned. Then, once get_longest_path() returns a trail, after the algorithm's execution, that trail's length is checked to ensure it has at least 2 elements. If it does not, an empty list is returned, otherwise it returns that trail.

get_longest_trail(torus) ensures get_all_trails() has the information it needs to evaluate the torus while tracking the length and the indices of the element with the longest trail. This is also where a matrix of value -1, with the same dimensions as the torus, is created for memoization. The nested for-loops call get_all_trails() for every element, but once every element has been visited, the longest trail is returned within the nested for loops. We do this by checking if v_indices is equal to the number of elements in the torus. The number of visited elements is tracked with a list containing a single element of starting value 0. get_longest_trail() passes the torus, i—the current row, k—the current column, ind_matrix, and v_indices as parameters to get_all_trails() with each iteration of the inner loop.

get_all_trails(torus, i, j, ind_matrix, v_indices) is the workhorse of the algorithm and utilizes recursive calls to the neighbors of the current element (i, j), to obtain the trail for the current element and all its neighboring elements. The subroutine starts by checking the current element index within the ind_matrix. If that value is not -1, it will return the trail stored at that index instead of searching. If that value is -1, v_indices is incremented to note we have visited a new index. The neighbors are then retrieved by calling get_neighbors(), a subroutine that implements cyclic functionality for gathering the neighbors on the other side of the matrix boundaries for the current element at (i, j). Each neighbors' index is then sent as the i and j parameters in the recursive call. Additionally, get_all_trails(torus, i, j, ind_matrix, v_indices) is where the memoization of the algorithm occurs by storing the longest path of an element, the list temp_trail, in the corresponding index (i, j) of ind_matrix, replacing the value -1.

Approaches, Issues, & Complexity

I struggled with quite a few issues for this assignment which led to multiple approaches that centered around the same iterative and recursive concept, resulting in 4 different versions of the code. Having all four helped me pinpoint where I was falling short in my thought processes for retrieving all possible paths. The first struggle I ran into was ensuring my recursive call was returning the intended trail. Starting out, this issue resulted in the algorithm producing inconsistent results, especially since I had not implemented memoization within the approach. Due to this, I created another version to experiment with how and where I initialized the trail. This helped me gain a better understanding of tracking the path during recursive calls through comparing the two files as changes were made.

After struggling to gain consistency in the algorithm, I created a third file where I implemented memoization. This presented a new form of thinking, though I was better prepared for it due to my experiments with tracking the trail in the previous implementations; allowing me to finally create an algorithm that consistently returned the expected trail. With the code in working order, my next roadblock was the code complexity, my algorithm was $O(N^2)$ at worst. For a while, the best worst-case complexity I could achieve was $O(m*n^2)$ which left me perplexed.

Eventually, I decided to create a variable to track the number of visited indices through the recursive calls to create a consistent loop break condition. The variable was initialized to zero and would only be incremented once we reached a new index (something I could now track due to implementing memoization) in the matrix. Once the variable's value was equal to the number of elements in the matrix, the code would return the longest matrix from within the nested for-loops, making the complexity **O(N).**