Austin Youngren

CS 320-001

Lab 5: Random Placement

03/08/2024

Algorithm & Complexity

My solution to the Random Placement problem utilizes iteration as the approach for the algorithm. The worst-case complexity of this algorithm is O(3(L*S)) but after dropping constants, the complexity is O(L*S).

For my approach I created five subroutines to help with finding possible bugs and to improve readability of the code. These routines are get_map_dimensions(map), get_num_map_spots(map), check_row_len_inequality(map), get_free_spaces(map), and place_objects(map).

get_map_dimensions(), get_num_map_spots(), and check_row_len_inequality() all take the map as a parameter and are used to retrieve information about the map. get_map_dimensions() was created to reduce the repetition of retrieving the map dimensions within various functions and is built to detect 1-dimensional tuple maps if they are passed to the algorithm. get_num_map_spots() is a function used to obtain the total number of elements in the map and utilizes get_map_dimensions() functionality to handle 1-dimensional tuple maps. check_row_len_inequality() is a subroutine used to check that all rows in the map are the same length—ensuring the map is rectangular in shape (an edge case that may or may not be checked for, but is mentioned in instructions).

The core routine, placement(num_object, map), first checks that the num_object parameter is of type int whose value is greater than 0; if not, None is returned. After, the routine checks if the map parameter value is None, or if it is empty, None is returned if true. If the map passes this check, I then ensure the map has more than one element and more than one dimension; if not, None is returned.

After the conditionals, the subroutine get_free_spaces() is called within placement(). get_free_spaces() takes a 2-dimensional tuple map as a parameter and creates a new, empty list, free_spaces. Then, the subroutine loops through the elements of the map, checking for Boolean values of True. If an element value is True, the indices of that element are appended to a list variable free_spaces. The list free_spaces is then returned to the placement() routine.

Once the algorithm has returned from get_free_spaces(), there is a check that ensures that the number of free spaces in the map is more than 0 and if there are enough free spaces for the number of objects to be inserted; if not, None is returned. If so, the free_spaces list and num_objects are passed as parameters to the place_objects() subroutine.

place_objects() is where the randomness of the algorithm is utilized. The subroutine creates an empty list, placed_objs, and uses a for-loop that iterates for the number of objects that need to be placed. For each iteration, free_spaces is passed to the function random.choice() as a parameter. Then, the random.choice() function returns a randomly chosen element (tuple of indices) from free_spaces. After, the subroutine removes the chosen element from free_spaces, it appends it to placed_objs.

Once complete, placed_objs (the list of indices where the objects are to be placed) is returned to the placement() routine; then placement() returns that list to its callee, concluding algorithm execution.

Issues & Alternatives

Finding a solution to this problem was generally straight forward, especially after Lab 4. However, ensuring that all the algorithm's edge cases were considered took some time. After an hour or two of creating test cases for the map (resulting in checks for cases that may not be expected criteria), I realized I did not create checks concerning the num_objects parameter within the algorithm (other than ensuring there were enough spaces for the number of objects passed). Once I realized this, I created an if-statement for placement() that ensures the algorithm is not receiving negative, 0, or non-integer values for num_objects.

A smaller issue I ran into when starting was understanding the shape of map. At times, due to the instruction examples, I was thinking of the map in an almost three 3-dimensional manner, which made initial test creation a difficult process. After trying a few different map types, I realized where my confusion came from and corrected the issue.

Initially my first approach combined place_objects() and get_free_spaces() as a single subroutine. Eventually I split it in two so I could create a check for the free_spaces list before choosing random elements from it. This check helped to ensure that the list was compatible with the number of objects, all while allowing for an early return if the number of places weren't viable.