Lab 11 Post Lab Hasan Khan, hk4cd

Time complexities

Prelab Complexity:

- First while loop of this program inserts the course names into the set course. O(n) complexity
- Nested for-loop populates the adjacency matrix (aMatrix) with 0's, thus O(n^2) complexity
- Second while loop runs through the courses file again and fills the aMatrix with edge mapping representations, thus having a O(n) time complexity. However, within this loop, the find() method is used to find the correct course given an integer (because we are using a map). Each call to find runs through the provided map using a simple for loop, and is O(n). Thus, the overall complexity for this while loop is O(n) * O(n) = O(n^2)
- Nested for-loop creates the inDegree and haveEdge vectors, O(n^2) time complexity
- Next for-loop inserts any courses with inDegree of 0 into the visited queue, O(n) time complexity
- In the last while loop, there if a for loop which checks the children of each course, with a time complexity of O(n), which is called n times (because it is called on each course). Thus, the complexity is O(n^2)
- TOTAL COMPLEXITY: O(n^2) + O(n^2) + O(n) + O(n^2) + O(n) → O(3n^2) → O(n^2) overall

In Lab Complexity:

- Time complexity of next_permutation -> O(n!)
- Time complexity of computeDistance -> single for loop that runs through the dests size; linear time complexity O(n)
- Time complexity of printRoute -> single for loop that runs through the dests size; linear time complexity O(n)
- While loop time complexity: calls computeDistance *next_permutation* amount of times; thus, time complexity is O(n*n!).
- TOTAL COMPLEXITY: O(n) + O(n*n!) → O((n*n!)

Space Complexity (size in parentheses, in unit bytes)

Prelab:

- 4 string read ins (s1, s2, s3, s4) (each string is 6 char long, and each char takes 2 bytes, so 12 * 4 = 48 bytes)
- size of map and set – the map links each course to a corresponding integer ((12* # of courses) + (4 * #of courses)), the set contains a set of all the courses (12 * #of courses)
- int numNodes, key1, key 2, currRow, currCol, infix, courser (28 bytes)
- aMatrix size (4 * (courses.size()^2) bytes)
- Vector <int> inDegree, haveEdge (4* numNodes amount of bytes)
- Queue <int> visisted (4* numNodes amount of bytes)

In-Lab:

- Size of the middle Earth object
- Dests (variable on the number and city string sizes)
- finalRoute (equal to dests in terms of space allocation)
- startCity – assuming start city is Dunharrow, (2* 9 = 18 bytes)
- tempDistance, minDistance, temp (4 * 3 = 12 bytes)

Acceleration Techniques (first two are heuristic, last is exact algorithm)

Nearest Neighbor – the nearest neighbor heuristic simply picks the nearest unvisited city as the next edge for the route. While this algorithm may pick a relatively short route, there are graphs where this it could pick a non-optimal route. The time complexity is O (log|v|), because the set of nearest city to choose from for each successive chosen city is reduced on each iteration (it grows smaller). This complexity is far smaller than the brute force O(n!) method, but doesn't promise the optimal route. A comparison of all complexity times is included on the next page.

Pairwise exchange, 2-opt – start with a random solution/route through the graph. Then, iteratively swaps 2 edges so that each now connect to the other's original endpoint. See the hypothetical visual below:

Distance: 150 *Distance after swap: 100*

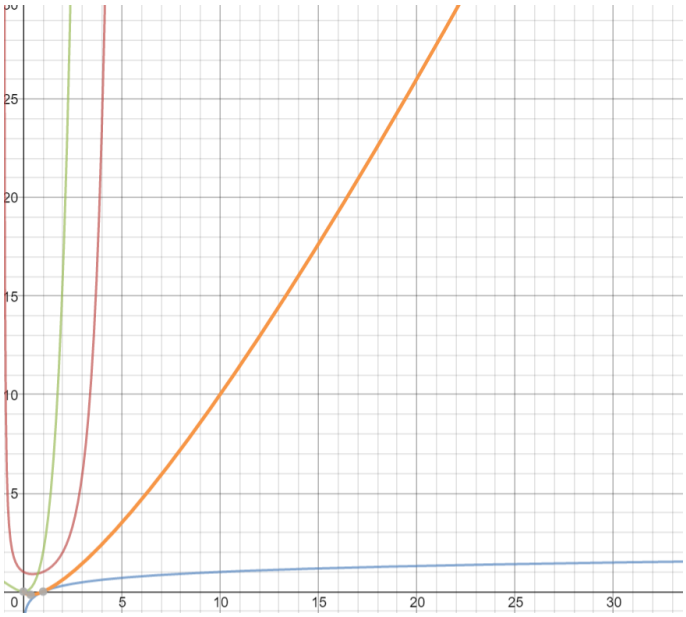
As the iteration continues, the total distance slowly decreases as the swaps slowly optimize the route. The time complexity for this heuristic is O (n log(n)), which a huge improvement over the brute force O(n!).

Reference: https://www.youtube.com/watch?v=SC5CX8drAtU

Held Karp Algorithm - This algorithm breaks down the route into sub routes, represented as sets, and finds the minimum distances of each sub routes, starting at the smallest sub route and working its way up to bigger sub routes until the entire tree graph's minimum route distance is calculated. The solution to the entire route consists of an equation that itself consists of smaller sub route equations, all of which are calculated recursively. The time complexity is: O (2^n * n^2). Note that on the graph, this equation looks slower than O(n!), but for large inputs O(n!) becomes slower.

Reference: https://www.youtube.com/watch?v=-JjA4BLQyqE

Time complexities comparison:



Blue = O(log n)
Red = O (n!)
Orange = O (n log n)
Green = O(n^2 * 2^n)