Pat Wongwiset
nw9ca
Postlab11.pdf
11/30/17

Complexity Analysis

In pre-lab part, topological.cpp file contained 2 classes: Vertex and Graph. These classes were created to build topological sort for the given graph which must be directed acyclic graph. The complexity Analysis for the program would be as followed such that V = the number of vertices in the graph, and E = the number of edges in the graph:

Method	Time Complexity	Space Complexity
Vertex()	A vertex had two constant fields, so the time complexity = **θ(1).**	A vertex had two constant fields, so the space complexity = **θ(1).**
add_vertex (adding a vertex in the graph)	There existed two loops for checking added vertex and filling weight (connection from a vertex to others in the graph) to 0 in case that added = true. Thus, the time complexity = θ(V) + θ(V) = **θ(V).**	When adding a vertex, there existed 'added' boolean variable and 2 strings in the first for-loop. After that, if added = true, the vertex was push_back() in the vertices vector and allocated weight that connected to other vertex to be 0. Due to declared variables and weight declaration from new vertex to all vertices, the space complexity = θ(1) + θ(V) = **θ(V)**. Even though two strings were declared inside the loop, it would be destroyed in each iteration. Therefore, the space complexity in that specific part was not θ(V).
Graph()	When creating a graph, the method looped to initialize all values in vertices = NULL. Then, the constructor would define a 2-dimensional array for edges. The time complexity = θ(V) + θ(V^2) = **θ(V^2).**	A graph contained 2 integers, 1 vector, and 1 matrix. Thus, the space complexity = 2*θ(1) + θ(V) + θ(V^2) = **θ(V^2).**
add_edge (adding an edge in the graph)	The method called add_vertex two times, find index of those vertices, and then change the weight at edges[row (src)][col (dest)] =1. The time complexity = θ(V) + θ(V) + θ(V) + θ(V) = **θ(V).**	The method added two given vertices into the graph by add_vertex(). Then, it initialized two variables by calling atIndex(). The space complexity = **θ(1)** from size of constant variables. The other parts were already allocated space in other methods.

Method	Time Complexity	Space Complexity
atIndex (checking an index of the given vertex)	The method looped over vertices list to find the name of vertex that matches to passing parameter, so the time complexity = **θ(V).**	An integer i was declared at first. It then initialized two variables overtime inside the loop. Thus, the space complexity = **θ(1)** from size of constant variables.
weight (checking if an edge was created between 2 vertices)	The method called atIndex() two times to find location of src and dest, so the time complexity = θ(V) + θ(V) = **θ(V).**	Two integers were declared from this method and returned integer from the matrix [v,w], so the space complexity = **θ(1)** from size of constant variables.
topsort (check if the graph is topological and then print out the order)	Since the worst possible case happened when the method looped to every vertices. All of the edges were unique, so the time complexity = **θ(V + E).**	The queue and three declared variables were in this method, so the space complexity = θ(V) + θ(1) = **θ(V)** from the queue's data structure.

The program ran in main(). It created a graph with a finite size which caused time and space complexity = $\boldsymbol{\theta(V^2)}$**,** and then read the file by looping until the end of the program. The program created two vertices and an edge inside the loop, so the time and space complexity = $\boldsymbol{\theta(E*V)}$**.** After that, the program called topsort() which had time and space complexity = $\boldsymbol{\theta(V + E)}$ and $\boldsymbol{\theta(V)}$ respectively.

Therefore, since E might be bigger than V, the program has total time and space complexity = $\boldsymbol{\theta(E*V)}$ and $\boldsymbol{\theta(V^2)}$ respectively.

The traveling.cpp in the in-lab part constituted two functions: cmputeDistance() and printRoute() to solve a salesperson problem that needs the "best" path. The program computed every possible paths and then chose the best path from that. The complexity Analysis for the program would be as followed such that V = the number of traveling cities, and n = the number of all cities:

Function/Method	Time Complexity	Space Complexity
MiddleEarth()	The method added all cities inside the world by vector. Then, it located all cities by adding and computing a position inside another loop. After that, it would compute the distance between all cities and placed it in 2-dimensional array. Finally, the program created hash_table for cities and index of it in the vector. Thus, the time complexity = θ(n) + θ(n) + θ(n) + $\theta(n^2)$ + θ(n) = **$\boldsymbol{\theta(n^2)}$.**	There are 3 vectors, a 2-dimensional array, and a hash_table, so the space complexity = θ(n) + θ(n) + θ(n) + $\theta(n^2)$ + θ(n) = **$\boldsymbol{\theta(n^2)}$.**

Function/Method	Time Complexity	Space Complexity
getItinerary():returning the list of cities traveled to	The method created new vector and loop V times, so the time complexity = **θ(V).**	Since there exists a vector, the space complexity = **θ(V).**
computeDistance(): computing a total distance between two cities	The method had a while loop that iterated through every traveling cities. It also called get Distance(), which has θ(1) time complexity, in order to add to the total distance. Thus, time complexity = **θ(V).**	There were 2 integers declared inside the method, so the space complexity = **θ(1).**
printRoute(): print Route from given start and traveling cities	The method had a while loop iterating through every traveling cities, and then printed the cities with '->'. Thus, time complexity = **θ(V).**	There was an integer declared inside the method, so the space complexity = **θ(1).**

When the program ran in main(), it planted passing parameters into the program with constant time and space complexity. The program created MiddleEarth 'me' which had $\boldsymbol{\theta(n^2)}$ space and time complexity. dests vector was created to find cities that traveled to by calling getItinerary(), so the time and space complexity = **θ(V).** Then, the program called sort() for dest which caused **θ(VlogV)** time complexity. To find the best path, I used permutation to observe every possible ways and reseted the result vector to current permutation when min (total distance) was smaller than that of the recent result vector. Since the result vector acted as a pointer, so the program did not have to initiate all value by looping inside. Thus, the time complexity = **θ(V!).** Finally, the program called printRoute(), which had **θ(V)** time complexity, and printed min.

Therefore, the total time complexity = **θ(V!)** and space complexity = $\boldsymbol{\theta(n^2)}$ since n may be larger than V.

Acceleration Techniques

1. Approximation Technique:Minimum Spanning Tree (MST)

This technique uses the triangle-Inequality which explained that a directly edge between two vertices will be the shortest path compared to a total distance which was connected by more than 2 vertices. When constructing a MST and defined a node as a root, the program goes in pre-order and added a root as the end of the order. The 'full walk' will go to all possible vertices in pre-order and it may repeat the same node. The total cost of full walk is at most twice larger than that of MST due to the MST structure constrained every edges to be visited at most 2 times. Also, the minimum cost of MST is the cost that connects all vertices, so the best possible Salesman Traveling is always less than that from MST. Thus, we could say that the MST is **2-approximate algorithm** since it guaranteed that the runtime/cost is at most twice from that of best possible output.

2. Exact Acceleration Technique: Dynamic Programing

To find the minimum cost (i.e. distance), the program adds recursive relation as a sub-problem such that C(S, i) = cost minimum in set x starting at 1 to i. The program then finds minimum cost (C(S,i) from all subset from size 2 to i. There is $O(2^n*n)$ time complexity for a sub-problem in one vertex. Thus, the total time complexity = $\boldsymbol{O(2^n*n^2)}$ which is much faster than

O(n!). For example, if n = 1000, the calculated worst time execution will be 1.07151*10^304 which is better than infinity from O(n!).

3. Approximation Technique: Christofide's Algorithm

The program creates an MST graph T from input graph G. Then, it takes vertices that have odd degree in T to G*. From the fact that 'any graph that has even number of vertices with odd degree', we could define that G* has even number of vertices. From the fact of minimum matchings in Polynomial time, G* is a complete weighted graph with even number, we could find a minimal perfect matching will be found in polynomial time. Since the previous fact, there are two matching paths. Thus, the optimization for path matching is 0.5*(optimal test). The optimal have one match for a vertex from all possible vertices, so the runtime can be approximately 1.5*(optimal test) when combining the path from G* to T. Therefore, this technique has proven to be 1.5-approximate algorithm, so it guaranteed that the runtime/cost is at most 1.5 times from that of best possible output.

http://www.geeksforgeeks.org/travelling-salesman-problem-set-1/
http://www.cs.tufts.edu/comp/260/Old/lecture4.pdf