Graphs and Algorithms

(Typed up in a nice word document so if there are mistakes, please feel free to correct them!)

1a) Topological sort order is the reverse of the exit order

Topological sort order: 1, 3, 2, 5, 6, 4, 7

Order entered: 1, 2, 4, 7, 5, 6, 3

Order exited: 7, 4, 6, 5, 2, 3, 1

Chart, line chart

Description automatically generated

1b) Order added: A, H, E, F, D

Chart, line chart

Description automatically generated

Shortest path is AHED, with total cost 7

1ci) For each node we add 4 to count. For each node we also add 5 for each arc incident on that node. Since each arc is incident on two nodes, we add 10 for each arc in the graph (assuming a simple graph)

So count = 4n + 10m = O(n + m)

1cii) The main program adds 3 to count for each node in the graph. Then iterate is always called on unvisited nodes (because it is always under the condition “if not visited[a]” or similar) and the main program attempts to call it for every node in the graph exactly once, and so we add 3 + 2 = 5 to count (again) for each node in the graph.

And, similar to part (i) we add 2 to count for each arc incident on each node, so add 4 for each arc in the graph.

This gives count = 3n + 5n + 4m = 8n + 4m = O(n + m)

1d) Pseudocode for the routine:

M[0] = 0

found[0] = true

for i = 1 to n:

M[i] = c [i]

found[i] = (c[i] != 0)

for j = 1 to i:

if found[i – j] and found[j]:

if not found[i]:

M[i] = M[i – j] + M[j]

else:

M[i] = min(M[i], M[i – j] + M[j])

found[i] = true

if not found[n]:

return “not found”

return M[n]

2ai) Order added: A, B, E, F, C, D, G

Chart, line chart

Description automatically generated

2aii) 2 distinct MSTs. All arcs of weight 3 or less are used in the above MST. If any of these arcs were to be removed and another arc of the graph that isn’t in the MST above were to be included instead, that arc would have a greater weight than the removed arc, thus increasing the weight of the tree (so the result would not be an MST). Thus, arcs BE, EF, BC, AB, and DE (all arcs of weight 3 or less) must be in any MST.

The only node left to connect after including those mandatory arcs is G. this can be done by including arcs AG, FG or EG. The weight of EG is greater than that of the other two, so either AG or FG must be included. Thus, we have only two distinct MSTs of the graph (depending on whether AG or FG is included).

Alternatively, a shorter answer is:

There are two distinct MSTs as you could swap arc AG with FG which has the same weight, and so the total weight of the tree is unchanged.

2bi) We have a = 4, b = 2, f(n) = 8n. Thus E = logba = log24 = 2

Let ε = 1

Then we have

F(n) = 8n = Θ(n1) = Θ(nE – ε)

So, by the Master Theorem, we have:

T1(N) = Θ(nE) = Θ(n2)

2bii) We have a = 8, b = 4, f(n) = 2nlogn. Thus E = log48 = 3/2 = 1.5

We have logn = O(nm) for any m > 0, so 2nlogn = O(n1.25), since 2n = O(n)

Let ε = ¼

Then we have f(n) = 2nlogn = O(n1.25) = O(nE – ε)

So by the Master Theorem,

T2(n) = Θ(nE) = Θ(n3/2)

Alternative Explanation:

T2(n) = 8T2(n/4) + 2nlogn  
So a = 8, b = 4, E = log(8)/log(4) = 1.5  
f(n) = 2nlogn = Θ(nlogn)

For n > 0, n1.5 > nlogn, so there must exist some ε > 0 such that O(n1.5-ε) = f(n)  
By case 3 of the master theorem, T2(n) = Θ(n1.5) as there exists some ε > 0 such that f(n) = O(nE - ε)

2c) No comparisons for the first element. Since the first n/2 elements are sorted, we have 1 comparison each for the next n/2 – 1 elements.

Then n/2 comparisons for the next element (i.e. 1 in the given example), and n/2 + 1 comparisons for the final n/2 – 1 elements.

So 0 + 1(n/2 – 1) + n/2 + (n/2 + 1)(n/2 – 1)

= n/2 – 1 + n/2 + n2/4 – 1

= n2/4 + n – 2 comparisons

2di) Define another problem VER-INDN as: given an undirected graph G, a node x of G, k >= 1 and a set I of nodes of G, is I an independent set of G of size >= k containing x?

To determine VER-INDN(G, x, k, I):

1. Check size(I) >= k
2. Check x ∈ I
3. Check no two nodes of I are adjacent

1 and 2 can obviously be checked in p-time. 3 can also be checked in p-time by iterating through the adjacency list of G and for each node in I checking that no node that it is adjacent to in G is also a member of I. this VER-INDN is in P.

Also, clearly (by definition of VER-INDN), we have:

INDN(G, x, k) iff ∃I . VER-INDN(G, x, k, I)

And also clearly there exists a polynomial p such that |I| <= p(|G, x, k|) since we have I ⊆ G.

Thus INDN is in NP.

2dii) We show that for all decision problems D ∈ NP, D <= INDN.

We have that IND is NP-complete, so for all D ∈ NP, D <= IND.

So it suffices to show IND <= INDN since by transitivity of <=, if D <= IND and IND <= INDN, D <= INDN.

To show IND <= INDN we show that for some p-time computable function f,

IND(G, k) iff INDN(f(G, k))

We define f as:

f(G, k) = (G’, x, k’)

where

nodes(G’) = nodes(G) ∪ {x}

k’ = k + 1

x is a new node

So f adds a node to G that is not adjacent to any other node, and adds 1 to k. Clearly f is p-time computable.

Assume IND(G, k). Then G has an independent set of size >= k. Thus, if (G’, x, k’) = f(G, k), then G’ also has an independent size of set k’ = k + 1, with the extra node being x, which can always be included in an independent set as it is adjacent to no other nodes. Thus IND(G, k) implies INDN(f(G, k)).

Assume INDN(f(G, k)). Then G ∪ {x} has an independent set of size >= k + 1 containing x, the node added by f. Thus, we also must have an independent set of size >= k of the graph G. Hence we have INDN(f(G, k)) implies IND(G, k).

So IND(G, k) iff INDN(f(G, k)), so IND <= INDN, and INDN is NP-complete.