Problem 1

Given a connected graph, G = (V, E) and a specific vertex u ∈ V, suppose a DFS rooted at u returns a tree T that contains all vertices in G. Also, suppose that a BFS rooted at u returns the same tree, T.

Claim

G = T (i.e., G does not contain any edges that are not also in T).

Proof

Assume, for contradiction, that G ≠ T. Then, by definition, there exist some vertices x, y in T such that (x, y) is an edge in G that is not in T. Since T is a DFS tree, by theorem 3.7, this implies that one of x or y is an ancestor of the other in T. However, T is also a BFS tree, which implies that x and y are at most one layer apart in T (by theorem 3.4). Since either x must precede y in T, or y must precede x, and they can be no more than a single layer apart, one of x or y must be the parent of the other. Since BFS always connects direct children with their parent node, the edge (x, y) must be in T. This contradicts our assumption that (x, y) ∉ T and that G ≠ T.

Problem 2

The problem asks for an algorithm that either returns an 2n-tuple of temporally sorted birth and death events, or else reports that no such sequence of events exists. Based on the manner in which the problem was phrased, it's quite clear that the solution should return a topological ordering of the alleged events (structured as a digraph), if possible. If the digraph is acyclic, there will be at least one way to topologically sort the nodes (events). If the digraph contains a cycle, then no topological ordering exists, and it follows that the data are inconsistent.

Solution

The set of 2n nodes (events), structured as a digraph G, can be partitioned into two sets, B and D, that contain nodes corresponding to births and deaths, respectively. Each individual $i \in P$ has corresponding nodes $i_b \in B$ and $i_d \in D$, plus a directed edge (i_b, i_d) in G to signify that i's birth preceded i's death. For each individual $i \in P$, we must examine the remaining n-1 individuals to determine the placement of edges. For some $j \neq i \in P$, if i and j were alive at the same time, add the edges (i_b, j_d) and (j_b, i_d) to signify that i was born before j died, and that j was born before i died. If i died before j was born, add the edge (i_d, j_b) to indicate the order of events. Once all individuals in P have been examined, we can attempt to topologically sort G by identify nodes that have $\delta_{in} = 0$.

```
// Assume that events are partitioned into sets B and D as described above.
For each i in P:
    Add edge (i_b, i_d).
    For each j ≠ i in P:
        If i and j were alive at the same time,
            add edges (i_b, j_d) and (j_b, i_d).
        Else if i died before j was born,
            add edge (i_d, j_b).
        Else j died before i was born,
            add edge (j_d, i_b).

// Now, we can attempt to topologically sort G.
Declare a tuple T to hold topologically sorted nodes.
Declare a set S to hold all nodes with δ_in = 0.
Declare a set W to hold δ_in(v) for all nodes in G.
For each node v in G:
    Add δ_in(v) to W.
    If δ_in(v) = 0, add v to S.
For each node v in S:
    For each edge (v, w):
        Remove edge (v, w), decrement δ_in(w).
        If δ_in(w) = 0, add w to S.
    Remove v from G and S, add v to T.

// Check if G has been topologically sorted.
If T has 2n elements, then G has been topologically sorted.
    Output T.
Else, G has a cycle.
    Thus, the data are internally inconsistent.
```

Correctness

By theorems 3.20 and 3.18, G has a topological ordering if and only if G is a directed acyclic graph. Similarly, the order of events described to the ethnographers in exercise 12 are internally consistent if and only if they can be temporally sorted. Since the above algorithm structures G as a directed graph and then attempts to find a topological ordering of G, it essentially checks whether or not the provided data are internally consistent.

Time Complexity

Constructing the digraph takes $n(n-1) = \mathcal{O}(n^2)$ time. Initially identifying all nodes with no incoming edges and filling W takes $\mathcal{O}(n)$ steps in the worst case. Then, deleting m edges from G takes an additional $\mathcal{O}(m)$ steps. Finally, outputting T takes $2n = \mathcal{O}(n)$ steps. Thus, the entire algorithm runs in $n^2 + 2n + m = \mathcal{O}(n^2)$ time.

Problem 3

Claim

The greedy algorithm that packs boxes (in the order of arrival) until a truck is full before loading the next truck is the solution to the truck packing problem described in [KT], problem 4.3. Let n be the number of trucks used to ship k boxes. The greedy solution will always use the fewest trucks to ship k boxes.

Proof

We can proof the optimality of the solution described in [KT], problem 4.3 by induction, showing that our greedy solution always 'stays ahead' of any proposed optimal solution.

Base case: n = 1

By definition the greedy solution will pack as many boxes as possible into the first truck, thus it's equal to the optimal solution for n = 1 trucks.

Inductive step:

Assume that the greedy solution is optimal for n = 1 to n-1 trucks. Then, in the $(n-1)^{th}$ case, if the greedy solution can fit j boxes into n-1 trucks, the proposed optimal solution is able to fit at most $i \leq j$ boxes into n-1 trucks.

n^{th} case:

Now, for the n^{th} truck, the greedy solution has $(k-j) \leq (k-i)$ boxes left to pack, while the proposed optimal solution has at least $(k-i) \geq (k-j)$ boxes left. Say that there are (k-i) boxes remaining: then, the greedy solution and optimal solution can both pack all of the boxes into the n^{th} truck. If there are $j \geq i > (k-i)$ boxes remaining, the optimal solution can only fit i boxes, whereas the greedy solution can fit at least the first i boxes, and possibly the first j boxes, if j ¿ i. Thus, the greedy solution is optimal for n = any number of trucks.

Problem 4

The problem asks us to decide on a schedule that minimizes completion time of a triathlon (consisting of swimming, biking, and running in the order stated), for all competitors. Only one competitor at a time may use the pool, and each competitor i has a projected swimming time, biking time, and running time, which are respectively denoted i_s, i_b, and i_r in the solution below.

Solution

An intuitive (naive) initial attempt at a solution might involve schedule competitors by swimming time, in ascending order (shortest i_s to longest). However, if the last competitor (and slowest swimmer) is also the slowest biker and slowest runner, the completion time of the triathlon will be sub-optimal: everyone else will be waiting for the last competitor to finish. However, if we send the last competitor out first, we can make more efficient use of our time, as other people will have time to use the pool and begin biking/running while the slowest (first) competitor meanders their way to the finish line. Ideally, we'd like the first competitor out to be the fastest swimmer, the slowest biker, and the slowest runner, but this is unlikely to be the case.

Instead, we can use a similar sort of logic (and the fact that multiple people can bike and run simultaneously), to deduce that the first competitor should be the one who has the longest combined running and biking times, and that the remaining competitors should be sent out in descending order of $i_b + i_r$ time. This way, we can maximize the number of people biking and running simultaneously. The final competitor will have the fastest running and biking time, giving them an opportunity to "catch up" to the other slower runners and bikers.

Proof

Imagine that the solution above is not optimal. Then, there exists some other optimal solution in which the competitors are not sent out in descending order of biking and running times. Suppose that in the optimal solution, some competitor k is sent out before some other competitor k+1, with biking/running time $k_b + k_r < (k+1)_b + (k+1)_r$ and swimming time $k_s = (k+1)_s$. Then, competitor k+1 would finish in $(k+1)_b + (k+1)_r + 2k_s$ minutes, while k would finish in $(k)_b + (k)_r + k_s$ minutes. In our solution, competitor k+1 goes first, taking $(k+1)_b + (k+1)_r + k_s$ minutes to complete the event, versus $(k)_b + (k)_r + 2k_s$ minutes for k. So competitor k takes k_s more minutes and competitor k+1 takes k_s fewer minutes in our solution versus an optimal solution, meaning that there is no difference between the two solutions.

Problem 5

Description

Describe an efficient algorithm for finding a Hamiltonian path in a DAG, and provide time complexity analysis.

Solution

Assume that we're given a directed acyclic graph, G = (V, E), and that we're using the topological sort algorithm described in [KT] page 103, with running time $\mathcal{O}(m+n)$, where m is the number of edges and n is the number of vertices.

```
Topologically sort G into a tuple T.
For each node v in T:
    If (v, v+1) ∈ E, continue to the next node.
    Else, no Hamiltonian path exists.
```

Correctness and Time Complexity

The topological sorting takes $\mathcal{O}(m+n)$ steps and examining adjacent edges between nodes in T takes $\mathcal{O}(n)$ steps. Thus, the entire algorithm runs in $\mathcal{O}(m+n)$ time. The algorithm is correct because a topological ordering arranges vertices in ascending order with respect to degree. Since a Hamiltonian path includes each vertex in G exactly once, if $v_1 -> v_2 -> ... -> v_n$, $0 = \delta_{in}(v_1) < \delta_{in}(v_2)$, so v_1 will precede v_2 in T. Then, according to the algorithm in [KT] page 103, we delete v_1 and all associated out-going edges, so that now $0 = \delta_{in}(v_2) < \delta_{in}(v_3)$, and thus v_2 precedes v_3. Continuing in this manner, we find that T follows the exact same order as the Hamiltonian path in G.