UNIVERSITY OF WASHINGTON, BOTHELL

CSS 549: ALGORITHM DESIGN AND ANALYSIS

PROBLEM SET 1

Sumit Hotchandani
Sharan Vaitheeswaran

1

The statement is true. In an instance where hospital h ranks student s top on its preference list and vice-versa, consider the following scenarios:

1. If both hospital h and student s are unmatched, and hospital makes the first proposal to s. In this case, s will accept the proposal from h and will reject any subsequent proposal, since it has already paired with its top match.

2. If s is already paired with another hospital h', and hospital h makes a proposal to student. In this case, s will unmatch with h' and pair up with h as it is higher on its preference list.

Alternatively, if there exists a perfect matching set with the pair (h', s) and (h, s'). Then since both h and s prefer each other to their current match, so this match would be unstable.

2

The algorithm is very similar to the base version of the Gale-Shapley algorithm, wherein a student s and a hospital h are both either committed or free. Instead of a direct match, we will be maintaining an array $slots$ for each hospital, with size of the array equal to the number of available slots in that hospital. This array will store the students that are matched with that hospital and we will also maintain a counter to track filled slots.

Algorithm 1 Gale-Shapley Generalized Algorithm

```
instantiate an array of slots for each hospital, with the size of the array equal to the number of available slots in that hospital.
while there is a hospital h has available slots do
    h offers a position to the next student s on its preference list
    if s is free then
        s accepts the offer
    else if s is matched with hospital h' then
        if s prefers h' to h then
            s remains matched to h'
        else s breaks match with h' and matches with h
            available slots at h' increase by 1
            available slots at h decrease by 1
        end if
    end if
end while
```

Proof of Stability: This algorithm finds a stable matching because it follows the principles of the Gale-Shapley algorithm:

1. For the first instability, per the algorithm if hospital h prefers student s' to s then h would propose to s' before it proposes to s and student s' if unmatched up to that point would accept the proposal and have a position at one of the hospitals, thus invalidating the scenario mentioned in the instability.

2. For the second instability, if hospital h is paired with student s, which causes the instability. This is only possible if h offers a position to s as a last resort because it prefers s' over s. And s' must prefer some other hospital h' to h as otherwise it must have rejected it's existing match to match with h which it prefers over all other hospitals. Thus, per the algorithm, this instability is a contradiction and cannot occur in any instance.

Worst-case running time: The worst-case running time of the algorithm is $O(mn)$, where m is the number of hospitals and n is the number of residents. In each iteration, each hospital processes proposals from at most n residents (hospital preference list length). There can be at most m iterations (number of hospitals). The outer loop runs until all hospitals have filled all their slots which in the worst-case scenario requires m iterations, and in each iteration a hospital may need to propose to each resident once before all slots are filled which requires at most n operations.

Therefore, the overall complexity is $O(mn)$.

3

The functions in ascending order of growth rate are as follows:

1. $f_3(n) = 2^{logn}$
2. $f_7(n) = nlogn$
3. $f_5(n) = n(logn)^4$
4. $f_2(n) = n^{5/3}$
5. $f_1(n) = n^2 logn$
6. $f_9(n) = n^{2.5}$
7. $f_6(n) = n^{logn}$
8. $f_8(n) = 10^n$
9. $f_4(n) = 2^{4n}$

4

1. **Basis Step:** Consider $n = 2$
   If a graph with n vertices contains $\frac{n(n-1)}{2}$ edges, then graph with 2 vertices will have $\frac{2(2-1)}{2} = 1$ edges.

2. **Inductive Hypothesis:** Let's assume for $n = k$, the graph contains $\frac{k(k-1)}{2}$ edges

3. **Inductive Conclusion:** Consider $n = k+1$, the graph will contain $\frac{(k+1)(k+1-1)}{2} = \frac{k(k+1)}{2}$ edges

   To prove this, let's use the induction hypothesis that a graph with k vertices contains $\frac{k(k-1)}{2}$ edges. Now if 1 more node is added to this graph which is connected to all other nodes, then k edges will be added.

   $$\frac{k(k-1)}{2} + k = \frac{k^2-k}{2} + \frac{2k}{2} = \frac{k^2+k}{2} = \frac{k(k+1)}{2}.$$

   Thus, a graph with n edges contains $\frac{n(n-1)}{2}$ edges.

5

We use a directed acyclic graph (DAG) to represent the precedence relationships between pets, where each vertex in the DAG represents a pet and an edge from P_i to P_j indicates that P_i died before P_j was born

1. For each fact "P_i passed away before P_j was born", add a directed edge from P_i to P_j in the DAG.

2. For each fact "P_i and P_j lived at the same time", check if there is already a path from P_i to P_j or P_j to P_i in the DAG. If not, add both edges (P_i -> P_j and P_j -> P_i) to create a two-way relationship.

3. Perform a topological sort on the DAG. If a cycle is detected during the sort, it means there are conflicting relationships between pets (e.g., P_i dying before P_j and P_j dying before P_i). This indicates inconsistency in the facts.

4. If no cycle is detected, then all the facts can be true simultaneously. The DAG represents a consistent timeline of the pets' life spans.

Explanation:

- A DAG efficiently represents the precedence relationships between pets. Paths in the DAG represent lifespan overlaps, and cycles indicate contradictions.
- Topological sort efficiently detects cycles in the DAG. If a cycle exists, it implies conflicting information and inconsistency in the facts.

Worst-Case running time:
Here the assumption is that the input set contains at least one instance of both facts.

- ***DAG construction:*** $O(m)$ time to iterate through and process each fact.
- ***Topological sort:*** $O(n+m)$ time, where n is the number of vertices (pets) and m is the number of edges (relationships).

Hence the worst case running time would be $O(n+m)$

6

The algorithm uses a greedy strategy to minimize the number of routers:

Algorithm 2 Greedy Sorting Algorithm

```
house_locations ← {h_1, h_2, ...h_n}        ▷ initialize a list with distances of all houses
sort house_locations in increasing order
router_locations ← {0}                      ▷ initialize a list to store router locations
current_coverage ← 100   ▷ initialize a variable with the range of router from current router location
for h_i in house_locations do
    if h_i <= current_coverage then
        continue
    else
        append h_i to router_locations
        current_coverage ← h_i + 100
    end if
end for
```

Explanation:

1. Initially all house distances from the street are added to a list *house_locations*. This list is sorted in increasing order, using either merge sort or quick sort, to ensure house closest to the origin are considered first.

2. Another list *router_locations* is initialized to store all locations where routers will be placed. The first element added to the list is 0, meaning that the first router is placed at the beginning of the street.

3. Variable *current_coverage* is initialized to 100, representing the range of router from its current location.

4. We then iterate over each hose h_i in the sorted *house_locations* list:

   (a) If house h_i is within *current_coverage* of last placed router, then we move onto the next house.

   (b) However, if house h_i is beyond *current_coverage* of last placed router then we place a new router at the location of current house by adding distance of h_i to the list *router_locations* and update *current_coverage* to cover the next 100 units from the newly added router location

The result is a list of router locations that cover all houses with the minimum number of routers. This algorithm is greedy because at each step it makes the locally optimal choice: it places a router at the farthest house it can cover. This greedy strategy results in a global optimum, covering all houses with the minimum number of routers.

Worst-case running time:

- ***Sorting:*** $O(nlogn)$ time using a comparison-based sorting algorithm like merge sort or quick sort.
- ***Placing routers:*** $O(n)$ time, iterating through the sorted list.
- ***Total worst-case:*** $O(nlogn)$ as the sorting operation