CMSC 27200 - Problem Set 5

Sohini Banerjee

February 16, 2024

Note: I have received a one-day extension from Professor Potechin.

1a

k	**s**	**a**	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**t**
0	0	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	0	4	5	8	∞	∞	∞	∞	∞	∞
2	0	3	5	8	1	9	∞	∞	13	∞
3	0	3	5	8	0	9	10	17	3	20
4	0	3	5	6	0	9	10	7	2	10
5	0	3	5	6	0	7	10	6	2	9
6	0	3	5	6	0	7	8	6	2	9
7	0	3	5	4	0	7	8	6	2	9
8	0	3	5	4	0	5	8	6	2	9
9	0	3	5	4	0	5	6	6	2	9
10	0	3	5	2	0	5	6	6	2	9

1b

Vertices with correct shortest path: s, a, b, d, g, h, t

Vertices with incorrect shortest path: c, e, f

The algorithm fails for nodes in a cycle (that is reachable from the starting vertex s) such that the sum of weights are negative. In this case, the sum of weights involving vertices c, e, f are negative. On the 10th iteration, we see that the weight for vertex c has updated, which is a warning sign for failure. This is true because with a vertex of n nodes, a valid path should not have more than $n-1$ edges since each vertex should be visited a maximum of once. Since there is a negative cycle of c, e, f, they do not have a shortest walk because we can make the walks arbitrarily negative.

2

We consider the following subproblem: What is the maximum happiness we can attain from time slot 1 to time slot j such that we are at venue i at time slot j?

Definition. Let $f(i, j)$ be the maximum happiness we can attain from time slot 1 to time slot j such that we are at venue i at time slot j. Let $g(i, j)$ be the previous venue in our schedule such that happiness is maximized at time slot j, and let $g(i, j) = \emptyset$ if $j = 1$ (i.e. we are at time slot 1, so

we cannot have previously been at a venue).

Definition. Let S be a valid schedule where the schedule consists of a sequence of venues visited at some time slots, in which there is a time slot when traveling between different venues.

Algorithm

- Base Case:
  - $f(i,1) = h_{i1}$ and $g(i,1) = \emptyset$ for all $i \epsilon [k]$
  - $n \geq 2$: $f(i,2) = h_{i2} + h_{i1}$, $g(i,2) = i$ for all $i \epsilon [k]$
- Recurrence Relation: $n \geq 3$: $f(i,j) = max\{f(i,j-1) + h_{ij}, max_{\forall i' \neq i}\{f(i',j-2) + h_{ij}\}\}$
  - Suppose we are at venue i at time slot j. If coming from venue i at time slot $j-1$ maximizes happiness, let $g(i,j) = i$. Otherwise, coming from venue i' at time slot $j-2$ (and traveling during time slot $j-1$) maximizes happiness at time slot j, so let $g(i,j) = i'$.
- Solution: The maximum happiness by the end of the festival occurs at time slot n and is given by $max_{\forall i \epsilon [k]}\{f(i,n)\}$. We can reconstruct the schedule by finding the venue i_{max} that maximizes the happiness, getting the previous venue using $g(i_{max}, n)$, and doing so until we have covered all the time slots. More specifically, if g gives a previous venue that is different from the optimal venue at time slot j, we know to check the previous venue at time slot $j-2$. If g gives a previous venue that is the same as the optimal venue at time slot j, we know to check the previous venue at time slot $j-1$. We do this until we have reached time slot $j = 1$.

Pseudocode

```
# venue i, time slot j
# k venues, n time slots

function find_maximum_happiness():
    # dp[i][j] = maximum happiness by time slot j at venue i
    # prev[i][j] = previous venue before reaching venue i at time slot j

    set dp = [[0] * (n + 1) for i in range(k + 1)]
    set prev = [[None] * (n + 1) for i in range(k + 1)]

    for i in range(1, k + 1), do:
        set dp[i][1] = H[i][1]

    if n >= 2, do:
        for i in range(1, k + 1):, do:
            set dp[i][2] = H[i][2] + H[i][1]
            set prev[i][2] = i

    for j in range(3, n + 1), do:
        for i in range(1, k + 1), do:
            if dp[i][j - 1] + H[i][j] > dp[i][j], do:
                set dp[i][j] = dp[i][j - 1] + H[i][j]
                set prev[i][j] = i
```

```
            for i_prime in range(1, k + 1):
                if i_prime != i:
                    if dp[i_prime][j - 2] + H[i][j] > dp[i][j], do:
                        set dp[i][j] = dp[i][j - 1] + H[i][j]
                        set prev[i][j] = i_prime

    set max_happiness = 0
    set final_venue = None
    for i in range(1, k + 1), do:
        if dp[i][n] > res, do:
            set max_happiness = dp[i][n]
            set final_venue = i
    set schedule = reconstruct_schedule(final_venue)
    return max_happiness, schedule
```

Proof

First, we will prove that $f(i,j) \geq max\{f(i,j-1) + h_{ij}, max_{\forall i' \neq i}\{f(i',j-2) + h_{ij}\}\}$.

- We will prove that $f(i,j) \geq f(i,j-1) + h_{ij}$. By definition, there exists a schedule S such that S ends at venue i at time slot $j-1$, so $f(S) \geq f(i,j-1)$. We can stay at venue i during time slot j and gain happiness h_{ij}, so $f(i,j) = f(S) + h_{ij} \geq f(i,j-1) + h_{ij}$. Thus, we have shown that $f(i,j) \geq f(i,j-1) + h_{ij}$.
- We will prove that $f(i,j) \geq f(i',j-1) + h_{ij}$ such that $i' \neq i$. By definition, there exists a schedule S such that S ends at venue i' at time slot $j-2$, so $f(S) \geq f(i,j-2)$. We can travel during time slot $j-1$ and stay at venue i during time slot j and gain happiness h_{ij}, so $f(i,j) = f(S) + h_{ij} \geq f(i,j-2) + h_{ij}$. Thus, we have shown that $f(i,j) \geq f(i,j-2) + h_{ij}$ such that $i' \neq i$.
- Taken together, this shows that $f(i,j) \geq max\{f(i,j-1) + h_{ij}, max_{\forall i' \neq i}\{f(i',j-2) + h_{ij}\}\}$.

Second, we will prove that $f(i,j) \leq max\{f(i,j-1) + h_{ij}, max_{\forall i' \neq i}\{f(i',j-2) + h_{ij}\}\}$.

- By definition, there exists a schedule S' such that S' ends at venue i at time slot j There are two cases to consider.
- Case 1: Suppose that S' was at the same venue during time slot $j-1$ and j. We will prove that $f(i,j) \leq f(i,j-1) + h_{ij}$. In this case, there exists a schedule S such that S ends at venue i at time slot $j-1$ and has maximum happiness $f(i,j-1)$, so $f(S) \leq f(i,j-1)$. From here, S' consists of S and then stays at venue i at time slot j to gain happiness h_{ij}, so $f(S) = f(S') + h_{ij} \leq f(i,j-1) + h_{ij}$. This means $f(i,j) \leq f(i,j-1) \leq max\{f(i,j-1) + h_{ij}, max_{\forall i' \neq i}\{f(i',j-2) + h_{ij}\}\}$.
- Case 2: Suppose that S' was at a different venue during time slot $j-2$ and j. We will prove that $f(i,j) \leq f(i',j-2) + h_{ij}$ such that $i' \neq i$. In this case, there exists a schedule S such that S ends at venue i' at time slot $j-2$ and has maximum happiness $f(i',j-2)$, so $f(S) \leq f(i',j-2)$. From here, S' consists of S and then travels during time slot $j-1$ and stays at venue i at time slot j to gain happiness h_{ij}, so $f(S) = f(S') + h_{ij} \leq f(i',j-2) + h_{ij}$. This means $f(i,j) \leq f(i',j-2) \leq max\{f(i,j-1) + h_{ij}, max_{\forall i' \neq i}\{f(i',j-2) + h_{ij}\}\}$.
- Taken together, this shows that $f(i,j) \leq max\{f(i,j-1) + h_{ij}, max_{\forall i' \neq i}\{f(i',j-2) + h_{ij}\}\}$.

Therefore, $f(i,j) = max\{f(i,j-1) + h_{ij}, max_{\forall i' \neq i}\{f(i',j-2) + h_{ij}\}\}$ for $j \geq 3$.

Runtime

There are $O(kn)$ subproblems to solve because consider the maximum happiness at all k venues at all n time slots. Furthermore, solving each problem is $O(k)$ because to find some $f(i, j)$, we have to consider coming from all possible venues, either the same venue at time slot $j - 1$ or a different venue at time slot $j - 2$. Furthermore, updating the previous venue for time slot j is done alongside checking each previous venue. Reconstructing the final schedule takes $O(n)$ time since we have to consider the venue at a maximum of n different time slots, and it takes $O(1)$ to recover the previous venue given the venue at a current time slot. Therefore, the time complexity of this problem is dominated by the dynamic programming part and is $O(k^2n)$.

3a

Definition. Let $e.children$ be the employees that employee e is an immediate supervisor of.

Definition. An employee v is an immediate subordinate of employee u if there exists an edge from u to v. More generally, an employee w is a subordinate of employee u if there exists a path from u to w.

1. First, suppose for contradiction that the graph contains a cycle. This means that there exists a path starting from a subordinate employee to a higher-level one. However, since each edge only connects a higher-level employee to a lower-level employee, once we reach a lower-level employee in the graph, we can no longer end up at a higher-level employee. This makes the graph acyclic.
2. Second, all employees have an edge from its supervisor and/or to its children, and there exists a path from the CEO to every employee. This makes the graph connected.

Therefore, a connected acyclic directed graph is a directed tree.

3b

Algorithm

- Preprocessing: Suppose there are n total employees. Number the employees from $[1, n]$.
- Stored Data: Let $days$ be a 1-based array of size n such that $days[i]$ gives the minimum number of days for all subordinates of employee i (supervised directly or indirectly) to receive the message. Let $order$ by a 1-based array of size n such that $order[i]$ gives the list of $e.children$ in non-increasing order by $days$.
- Initialization: Let $days[i] = \infty$ for all $i \epsilon [n]$.
- Recursive Step: We will pass an employee e to the recursive function. Start by passing the CEO to the recursive function.
  - Base Case: If $e.children = \emptyset$, set $days[e] = 0$.
  - Otherwise, do the following:
    - Call the recursive function for each employee in $e.children$ (in any order).
    - Sort $e.children$ in non-increasing order of $days[c]$ where c is in $e.children$, and let $order[e]$ be this sorted list. So, $days[c_1] \geq days[c_2] \geq ... \geq days[c_m]$ where m is the size of $e.children$.
    - Set $days[e] = max(days[c_i] + i)$ where $i \epsilon [m]$.

Pseudocode

```
function find_minimum_days(employee e):
    if count(e.children) = 0, do:
        set days[e] = 0
    else, do:
        for c in e.children, do:
            find_minimum_days(c)
        set order[e] = sort_by_days(e.children)
        set days[e] = 0
        for i in range(1, count(order[e]) + 1), do:
            set days[e] = max(days[e], days[order[e][i]] + i)
```

Explanation

Base Case: Suppose employee e has no children. Then, there are no subordinates to distribute the message to, so $days[e] = 0$.

Recursive Case:

- Overview: We solve this problem bottom-up by solving the problem for all subordinates or employee e before solving the problem for employee e. More specifically, we use the solutions for the employee e's immediate subordinates to solve the problem for employee e.
- We can use an exchange argument to justify sorting immediate subordinates in non-increasing order of minimum days to notify their subordinates.
  - Suppose employee e has some immediate subordinates c_k and c_{k+1} such that e notifies c_k before c_{k+1} where $days[c_k] < days[c_{k+1}]$. It takes employee e $days[c_k] + k$ days to notify c_k and its subordinates and $days[c_{k+1}] + (k+1)$ days to notify c_{k+1} and its subordinates. Therefore, considering c_k and c_{k+1}, it takes $max(days[c_k] + k, days[c_{k+1}] + (k+1)) = k + max(days[c_k], days[c_{k+1}] + 1)$ days to notify c_k, c_{k+1}, and their subordinates. Since $days[c_k] < days[c_{k+1}]$, $k + max(days[c_k], days[c_{k+1}] + 1) = k + days[c_{k+1}] + 1$.
  - Now, we apply an exchange argument to notify c_{k+1} before c_k. In this case, it takes $k + max(days[c_{k+1}], days[c_k] + 1)$ days to notify c_k, c_{k+1}, and their subordinates. Since $days[c_k] < days[c_{k+1}]$, then $days[c_k] + 1 \leq days[c_{k+1}]$, so $k + max(days[c_{k+1}], days[c_k] + 1) = k + days[c_{k+1}]$.
  - Prior to the exchange, it took $k + days[c_{k+1}] + 1$ days to notify and after the exchange, it took $k + days[c_{k+1}]$ days to notify. Therefore, it is always optimal to notify an employee first if it takes more days to notify its subordinates than another employee. In order words, we should order immediate subordinates such that $days[c_k] \geq days[c_{k+1}]$.
- It takes $days[c_k] + k$ to notify the kth subordinate (after sorting). Intuitively, the first subordinate we notify takes $days[c_1]$ days to notify its subordinates, and we add an extra day to account for employee e notifying c_1. Similarly, the second subordinate we notify $days[c_2]$ days to notify its subordinates, and we add an extra day to account for employee e notifying c_2. However, employee e notified c_2 a day after c_1, c_3 a day after c_2, and so on. So, it takes i days for immediate subordinate c_i to receive the message (after employee e has received it) and $days[c_i]$ days to transmit it to its subordinates. As a result, the maximum days of an immediate subordinate of e to receive and notify its subordinates gives the minimum days of e to notify all its subordinates.

Runtime

- Preprocessing: Labeling the employees takes $O(n)$ time.
- Recursive Step:
  - Naively, there are n subproblems to solve for each employee e. Given the solutions to the subproblems of e, our algorithm has to sort all the children of e by *days*. Each employee has a maximum of n immediate subordinates, so sorting takes $O(n \log n)$ time, making the overall time complexity $O(n^2 \log n)$.
  - However, suppose $C = max(c_i)$ where c_i is the number of immediate subordinates employee i has. Furthermore, the overall time complexity is $\sum_{i=1}^{n} c_i \log(c_i)$ (since each subproblem involves sorting its immediate subordinates). $c_i \leq C$, so $\sum_{i=1}^{n} c_i \log(c_i) \leq \sum_{i=1}^{n} c_i \log(C) = log(C) \sum_{i=1}^{n} c_i$. Also, $C \leq n$ since an employee cannot have more immediate subordinates than the number of total employees. So, $log(C) \sum_{i=1}^{n} c_i \leq \log n \sum_{i=1}^{n} c_i = n \log n$.

Therefore, the overall time complexity is dominated by the recursive step and takes $O(n \log n)$ time.

4

Definition. Let $f(l, r)$ be the list of possible winners in all games between players p_l through p_r, inclusive.

Algorithm

- Base Case: $f(x, x) = [p_x]$ for all $x \epsilon [n]$
- Recurrence Relation: Let $f(l, r) = \cup_{x \epsilon [l+1, r]} \{compete(f(l, x-1), f(x, r))\}$ where *compete* simulates a game between every p_i where $l \leq i \leq x - 1$ and every p_j where $x \leq j \leq r$. More specifically, *compete* returns a list of all players who have won a game of p_i versus p_j.

Pseudocode

```
# dp[l][r] = [players who can win with players between l and r, inclusive]

function find_possible_winners():
    set dp = [[] * (n + 1) for i in range(n + 1)]

    for x in range(1, n + 1), do:
        set dp[x][x] = [p[x]]

    for right in range(1, n + 1), do:
        for left in range(r, 0, -1), do:
            set all_winners = []
            for x in range(l + 1, r + 1), do:
                set (all_winners += compete(dp[l][x - 1], dp[x][r]))
            set dp[left][right] = remove_duplicates(all_winners)

    return dp[1][n]
```

Explanation

Base Case: If each p_x is competing with nobody else, p_x is the only possible winner.

Recurrence Relation: We consider the final battle in the range $[l, r]$. There exists a split such that a player is one of two finalists, and one of the finalists must win the battle. To be a finalist, this player must have competed and been able to win against another set of players (or no player, if the group has one player).

We can consider the final battle of two groups A and B, such that group A has possible winners among $[l, x-1]$ and group B has possible winners among $[x, r]$, where x is splitting the final interval $[l, r]$ into two sub battles. Since $l + 1 \leq x \leq r$, we know that both group A and B have at least one player for every valid split. Furthermore, we are given that $f(l, x-1)$ and $f(x, r)$ give the possible winners for group A and B, respectively. For a player to win among $[l, r]$, it must be able to win some subgame and emerge as a finalist in the subgame, and also beat any other possible finalist in the other subgame. More specifically, subgame $[l, x-1]$ and $[x, r]$ represent the possible finalists coming from group A and B.

To find any possible finalist, we must simulate a game between every p_i in A and p_j in B since any player group A and any player from group B can be in the final battle of players $[l, r]$. Therefore, suppose some player p_i beats another player p_j (without loss of generality, apply same logic to p_j beating p_i). Since p_i can be a winner and p_j can be a winner in their respective subgames, there exists a series of games such that p_i and p_j are the finalists and p_i beats p_j, where p_i emerges as the winner of $[l, r]$. More specifically, if p_i is a possible winner of range $[l, x-1]$ and p_j is a possible winner of $[x, r]$, then after p_i wins battles among group A and p_j wins battles among group B, p_i and p_j will become adjacent players and one of them emerges as the winner of the entire range.

Finally, we consider all splits because we cannot make an assumption of which 2 players will competing against each other in the final round, so we consider all splits of group A and B. In particular, when there are adjacent players, the middle player might play the player to their left or right, so we must consider what happens when you play one particular side in the subgames.

Runtime

There are $O(n^2)$ subproblems to solve because have n options for each subarray bound (i.e. l can take n different indices and r can take n different indices). To solve each problem, we first consider n possible splits for $x \epsilon [l+1, r]$. To solve each split, we have to simulate n^2 games since $[l, x-1]$ and $[x, r]$ can each contain a maximum of n players. Therefore, it takes $O(n^3)$ time to solve each subproblem, making the overall time complexity $O(n^5)$.