Take-home final

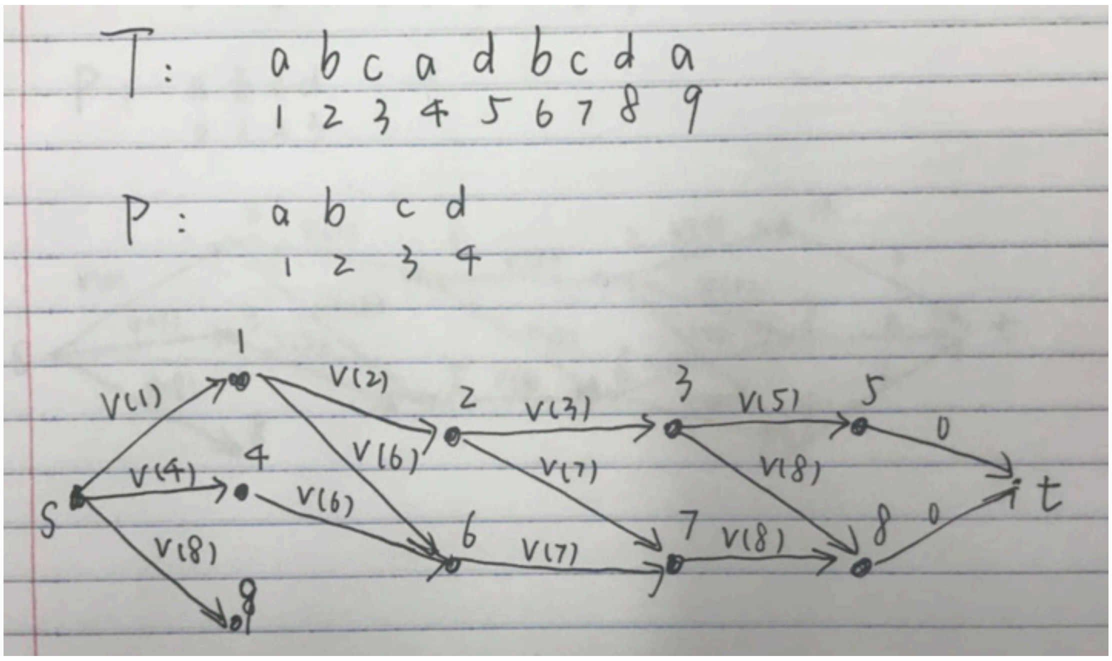
Wenjun Zhang

A53218995

Problem 1: Sequence matching

Algorithm description:

We can convert this problem into finding the largest weighted path in a DAG. To begin with, we can construct a graph $G = (V, E)$. For the vertex set, we first define two nodes s and t as the start and end point. There will be k layers between s and t. For layer i, there are all the possible nodes j such that $t_j = p_i$. For the edge set, we first link s to all nodes in the first layer, and link all the nodes in the last layer to t. Then, between two consecutive layer, j' in the previous layer link to j in the next layer if $j > j'$. Next, we assign all the edges (j, t) with weight 0, and for the other edges, we assign edge (j', j) with weight $v(j)$. An example graph is shown as below.



After constructing the graph, we need to get the topological order for all the nodes. Next, we can use the Dynamic Programming to solve this problem. We define $DP(j)$ as the largest weighted path from s to j, with base case $DP(s)$ = 0 and the recurrence is as below:

$$DP(j) = max_{(j',j)\in E}\{DP(j) + weight(j', j)\}$$

where $DP(j)$ represents the longest weighted path from s to j, and j' are all nodes that have a directed edge to j and are in front of j in Topological sorting order.

$DP(t)$ is the maximum sum that we are looking for. And we can iterate from $DP(t)$ to $DP(s)$ to find the path, which will be the sequence we want.

Proof of correctness:

To prove the correctness of the algorithm, we first prove that the graph we construct is a DAG. Since combining s and t, there are $k+2$ layers in this graph, and edges only exist from the previous layer to the next layer, there won't be a circle in this graph.

Next, we prove that the largest weighted path in this DAG represents the sequence we want in this question. Since the value $v(i)$ are all non-negative, the maximum weight path in the graph must be from s to t. And for each layer between s and t, only nodes j' such that $t_{j'} = p_i$ are in layer i, so they are candidates for p_i. And because j' only links to j in the next layer where $j' < j$, this path is bounded that the node index number in the next layer will always be larger than the previous one, which corresponds to the request that $1 \le i_1 \le i_2 \le \cdots \le i_k \le n$. For edges (s, j) and (j', j), the weight of each edge is assigned with $v(j)$, which is the value of using that node j. In addition, all the edges (j, t) are weighted 0 so there won't be additional weights put into the path.

Finally, we prove that the Dynamic Programming method can return the largest weighted path in this DAG. We first assume $DP(j)$ represents the largest weighted path from s to j and a base case $DP(s) = 0$. For the recurrence,

$$DP(j) = max_{(j',j)\in E}\{DP(j) + weight(j', j)\}$$

Since we traverse the nodes in the vertex set in Topological order, if j' is in front of j in Topological sorting order and have a directed edge to j, $DP(j')$ will have a value, and the recurrence will consider all the candidates that have a directed edge to j and return the maximum. So $DP(j)$ will be the largest weighted path to j, and $DP(t)$ will be the largest weighted path in the DAG.

Time complexity:

There are at most $O(nk)$ nodes and $O(n^2k)$ edges in the graph, so the construction of the graph takes $O(n^2k)$. Topological sort takes $O(|V| + |E|) = O(n^2k)$. Besides, the DP traversal takes at most $O(n^2)$ in each layer. Therefore, the time complexity of the algorithm is

$$T(n^2k)$$

Problem 2: Covering points with gain

Algorithm description:

We can solve this problem by applying a two-dimensional Dynamic Programming method. Firstly, sort $\{x_1, x_2, \dots, x_n\}$ in non-decreasing order. Then we also sort the intervals based on the following rules:

1. If $f_{i+1} > f_i$, interval $[s_i, f_i]$ is placed in front of $[s_{i+1}, f_{i+1}]$.
2. If $f_{i+1} = f_i$ and $s_{i+1} \geq s_i$, interval $[s_i, f_i]$ is placed in front of $[s_{i+1}, f_{i+1}]$.

We define $DP[i][j]$ as the maximum gain from the interval $[s_1, f_1]$ to $[s_i, f_i]$, considering the point set is $\{x_1, x_2, \dots, x_j\}$. The base case are $DP[0][j] = 0$ for $0 \leq 1 \leq n$. Assuming that for $[s_{i+1}, f_{i+1}]$, there are k_{i+1} points in $\{x_1, x_2, \dots, x_j\}$ that are within the interval, the recurrence is shown as below:

$$DP[i+1][j] = \max\left(DP[i][j], DP[i][\max(0, j-k_{i+1})] + \sum_{\substack{x_l \in \{x_1, \dots, x_j\} \\ x_l \in [s_{i+1}, f_{i+1}]}} w_l - c_{i+1}\right)$$

We can traverse the $DP[i][j]$ row by row from left to right. When we start to meet the point $x_{j'} > f_i$, the rest of $DP[i][j]$ in this row is computed as

$$DP[i][j] = DP[i][j'-1] \qquad for\, j' \leq j \leq n.$$

We finally output $DP[d][n]$ as the maximum gain of all intervals. And we can look back from $DP[d][n]$ to $DP[0][0]$ to find which intervals are used.

Proof of correctness:

We first define $DP[i][j]$ as the maximum gain until interval $[s_i, f_i]$ that cover points until x_j. The base case are $DP[0][j] = 0$ for $0 \leq 1 \leq n$, meaning the gain until any point when using no interval is zero.

Next, we are going to prove the recurrence. When a new interval $[s_{i+1}, f_{i+1}]$ comes, we will have two options: to use this interval or not to use. If we choose not to use this interval, the maximum gain until intervals $[s_{i+1}, f_{i+1}]$ using j sorted points will be the same as $DP[i][j]$. If we choose to use this interval, then this interval at most cover k_{i+1} consecutive x_l points. So we should extract the points that this interval covers, and use the previous i intervals to cover the previous points. There will be $j - k_{i+1}$ or no points that the previous intervals need to cover if $j - k_{i+1} \leq 0$. In this case,

$$DP[i+1][j] = DP[i][\max(0, j-k_{i+1})] + \sum_{\substack{x_l \in \{x_1, \dots, x_j\} \\ x_l \in [s_{i+1}, f_{i+1}]}} w_l - c_{i+1}$$

Because $DP[i][j]$ is a maximizer, we choose the larger one of the two terms to be $DP[i][j]$. In addition, when we start to meet the point $x_{j'} > f_i$, then the range until interval $[s_i, f_i]$ cannot cover the point $x_{j'}$ and those after $x_{j'}$, so we assign

$$DP[i][j] = DP[i][j' - 1] \qquad for\, j' \leq j \leq n.$$

Because we traverse $DP[i][j]$ row by row and from left to right, there will always be a value for every $DP[i][j]$. At the end, $DP[d][n]$ represents the maximum gain until the last interval that tries to cover all the points, and it corresponds to the solution of the problem.

Time complexity:

The two sorting each takes $O(d \log d)$ and $O(n \log n)$. We need to traverse the whole DP matrix once, so here it is $O(nd)$. In each iterations, we need $O(1)$ comparison and $O(n)$ to compute the sum of gain of points. Hence, the time complexity of the algorithm is

$$T(n) = n^2 d$$

Problem 3: Round-robin tournament

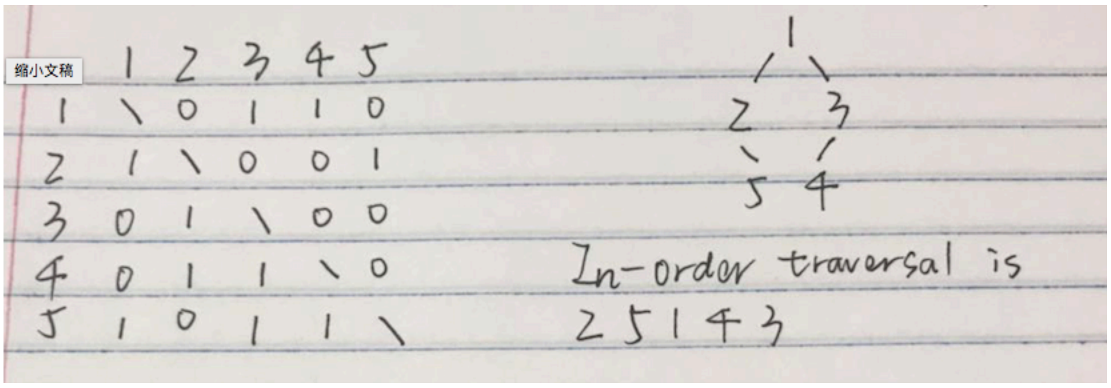
Algorithm description:

At first I want to try Topological sorting or divide and conquer method to solve this problem, but this problem is not necessarily a DAG and I couldn't figure out a divide and conquer algorithm to solve this problem in $O(n \log n)$. Here it's an algorithm with best case time $O(n \log n)$ but worst case in $O(n^2)$.

We can use binary tree and In-order traversal to solve this problem. Assume the players are marked with $p_1, p_2, \dots, p_n$. We traverse the players from p_1 to p_n and build a graph mentioned as below:

1. We define p_1 as the root of a binary tree.
2. At each iteration we meet a new player p_i, we begin by comparing p_i with the root p_j. If p_i defeats p_j, we assume p_j as the left child of p_j and we continue to compare p_i with p_j. Similarly, if p_j defeats p_i, we assume p_j as the right child of p_j and we continue to compare p_i with p_j.
3. We stop the last step until we reach the leaves of the tree. If p_i defeats this leaf p_j, we assign p_i as the left child of p_j. If p_i is defeated by this leaf p_j, we assign p_i as the right child of p_j.
4. After assigning p_i as a leaf of the binary tree, we continue to compare the next player p_{i+1} until we traverse all the players.

A possible binary tree is shown as below. Finally, we can return the In-order traversal of this binary tree as a solution to the problem.



	1	2	3	4	5
1	\	0	1	1	0
2	1	\	0	0	1
3	0	1	\	0	0
4	0	1	1	\	0
5	1	0	1	1	\

In-order traversal is
25143

Proof of correctness:

We can prove the correctness by looking at the rules we set to build the binary tree. According to the rules, we can ensure that the players in the left subtree of a node p_j all defeat p_j, and the all the players in the right subtree of a node p_j are defeated by p_j. We traverse all the players and we build a binary tree with n nodes, so the In-order traversal can ensure there are n players. Since the order of In-order traversal is left, middle, right, we can assure that the In-order traversal corresponds to a feasible solution of the problem.

We can also prove the correctness by assuming In-order traversal is not a feasible solution. In this case we will have a player in the left subtree is defeated by the root p_j, which contradicts the fact that the player who is defeated by the root p_j be put into the right subtree.

Hence, we prove the correctness of this algorithm.

Time complexity:

The worst case is to meet new player who always defeat or be defeated by the previous players. In this case the time complexity is

$$T(n) = O\big(1 + 2 + \cdots + (n-1)\big) = O(n^2)$$

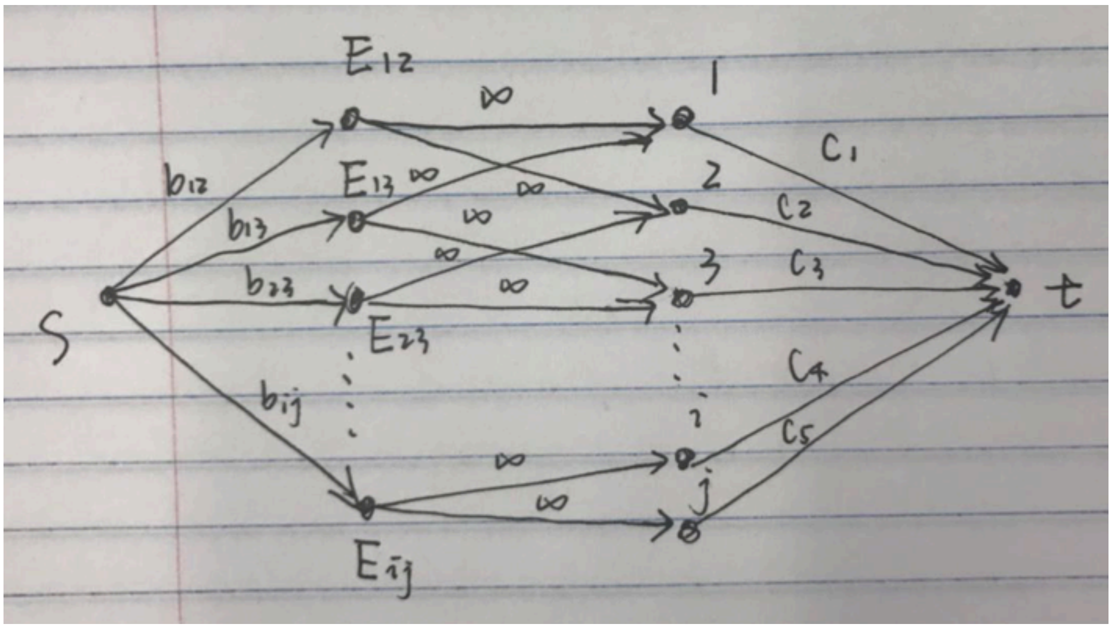
The best case of this problem is that the binary tree is a complete binary tree. In this case the maximum times of comparison is $O(\log n)$, so the time complexity is

$$T(n) = O(n \log n)$$

Problem 4: Cellular network (Revised problem)

Algorithm description:

We can use min-cut to solve this problem. We first construct a network $G' = (V', E')$. For the vertex set V', it includes a source s and sink t, and the vertices in the provided undirected vertex set V. In addition, for every edge $(i, j) \in E$, a node E_{ij} is included in V. For the edges set E', we first link s to all the E_{ij} nodes in V', then we link every E_{ij} to nodes i and j. Besides, we also link every node in V to the sink t. For the capacities of the edges, we define $c(s, E_{ij}) = b_{ij}$, $c(E_{ij}, i) = c(E_{ij}, j) = \infty$ and $c(i, t) = c_i$. An example network is shown as below:



The capacities can be irrational, so we need to apply Preflow-push method to find the min cut. Suppose the min cut (S, T) where the source *s* belongs to S and the destination *t* belongs to T, then the set of all the vertices i in S will be the subset we want.

Proof of correctness:

To prove the correctness of the algorithm, we need to show that the min cut solution corresponds to the solution of the original problem.

In this problem, we are looking for a set A such that $\sum_{i,j \in A} b_{ij} - \sum_{i \in A} c_i$ is maximized. We assume $C = \sum_{i,j \in V} b_{ij}$, then this problem is equivalent to looking for the minimized $C - \sum_{i,j \in A} b_{ij} + \sum_{i \in A} c_i$.

There are three types of edges in G', where $c(E_{ij}, i) = \infty$, so the min-cut will not cross any (E_{ij}, i) edge, or it will lead to infinite cut value. Therefore, the min-cut only involves the other two types of edges, where we have $c(s, E_{ij}) = b_{ij}$ and $c(i, t) = c_i$. Suppose the min-cut is (S, T) where the source *s* belongs to S and the destination *t* belongs to T. Then the min-cut value is

$$cut(S,T) = \sum_{E_{ij} \notin S} b_{ij} + \sum_{i \in S} c_i = C - \sum_{E_{ij} \in S} b_{ij} + \sum_{i \in S} c_i$$

Since the min-cut doesn't involve any (E_{ij}, i) edge, E_{ij} must be in the same set with node i and j. Hence, if we assume A' is the set of all the vertices i in S, we can rewrite the above equation as

$$cut(S,T) = C - \sum_{i,j \in A'} b_{ij} + \sum_{i \in A'} c_i$$

Since this is the min cut, we cannot achieve a cut that has a smaller value, A' is exactly the set we are looking for. Therefore, if the min cut is (S, T) where the source *s* belongs to S and the destination *t* belongs to T, the set of all the vertices i in S will be the subset we want.

Time complexity:

We first need to construct the network, which has vertices $|V'| = O(|V| + |E|)$ and edges $|E'| = O(|E| + 2|E| + |V|) = O(|V| + |E|)$. So the construction takes $O(|V| + |E|)$. We use FIFO preflow-push algorithm, whose time complexity is $O(|V'|^3)$, so it takes $O((|V| + |E|)^3)$. Thus, the total complexity of the algorithm is

$$T(n) = O((|V| + |E|)^3).$$

Problem 6: Approximation algorithm

Suppose the optimal solution is OPT, and the sum of all the c_i is $2L$, which can be written as $2L = \sum_{i=1}^{n} c_i$. Since, we are looking for the bigger sum of the partitioned set, it is trivial to know that $OPT \geq L$.

The algorithm first chooses k largest c_i's and find the optimal partition of these integers. We can without loss assume that $c_1, c_2, \ldots, c_k$ are the k largest c_i's. Suppose the two sets are obtain using this algorithm are S_1 and S_2, and for convenience, we write $w(S_j) = \sum_{i \in S_j} c_i$, and we can also without loss suppose $w(S_1) \geq w(S_2)$.

Let l be the last one that is put into S_1 that makes $w(S_1) - c_l \leq w(S_2)$. By adding $w(S_1)$ to both sides, we have

$$2w(S_1) - c_l \leq 2L \qquad \rightarrow \qquad w(S_1) \leq L + \frac{c_l}{2}$$

There are two periods that c_l can be put into S_1. If it happens during the period when we partition the k largest c_i's, then S_1 is the optimal solution. We can prove this by looking at the description of the algorithm. We first partition the k largest c_i's and obtain the optimal solution. In the second period when we partition the rest elements, there is no such element that can make $w(S_1)$ minus its value to be smaller than $w(S_2)$. According to the algorithm, we only put the element to a set if the sum in that set is smaller. Hence, no element is put into S_1 in the second period. Since $w(S_1) \geq w(S_2)$ and S_1 is derived by find the optimal solution in the first period, S_1 is the optimal solution, and the approximation ratio is 1 in this case.

If c_l is put into S_1 in the second period, then we know that

$$c_l \leq c_i \qquad for\ 1 \leq i \leq k$$

Hence, there are at least $k+1$ c_i with value no smaller than c_l in the whole set, and we can write it as

$$(k+1)c_l \leq 2L \qquad \rightarrow \qquad \frac{c_l}{2L} \leq \frac{1}{k+1}$$

So the approximation ratio can be written as

$$\frac{w(S_1)}{OPT} \leq \frac{w(S_1)}{L} \leq \frac{L + \frac{c_l}{2}}{L} \leq 1 + \frac{c_l}{2L} = \frac{k+2}{k+1} < \frac{k+3}{k+1}$$

We first need to choose the k largest c_i's. We can solve this by traversing the whole set once, so it takes $O(n)$. Next, we are going to partition the k largest c_i's, which will take $O(2^k)$ by checking all the possibilities. Next, we simply traverse the rest set and partition them by checking which set has larger sum, so it takes $O(n-k)$. Therefore, the whole algorithm takes time complexity $O(2^k+n)$.

As mentioned all above, we prove that this algorithm with $O(2^k+n)$ time complexity outputs a sum that is within $(k+3)/(k+1)$ fraction of the optimal output.