SHANGHAI JIAOTONG UNIVERSITIY

Assignment 3 Suggested Answer

Course: *AI2615 - Algorithm Design and Analysis*
Due date: *Dec 1, 2021*

Problem 1

Give a linear-time algorithm that takes as input a tree and determines whether it has a *perfect matching*: a set of edges that touch each node exactly once.

Answer referred by Yichen Tao . Here is the algorithm:

Algorithm 1 Check perfect matching

```
Require: A tree $T = (V, E)$.
Ensure: Whether $T$ has a perfect matching and the prefect matching $M$ if it has one.
  $Match \leftarrow \emptyset$
  while $E$ is not empty do
    for All leaf nodes $v_L$ in $V$ do
      Find the parent of $v_L$ : $v_P$
      $M \leftarrow M + \{(v_L, v_P)\}$
      Remove all edges connecting to $v_P$ from $E$.
      Remove $v_L$ and $v_P$ from $V$.
    end for
  end while
  if $V$ is empty then
    return $True$ & $M$
  else
    return $False$
  end if
```

Proof of Correctness. Notice that if the tree has a perfect matching, the number of nodes should be even. Since the algorithm deletes two nodes at a time, it will always return *False* when the number of nodes is odd. So we only consider the case when the number of nodes is even, and denote the number of nodes by $2n$. We prove the correctness by induction over n.

Base case. When $n = 1$, the tree consists only of a root and a leaf node, and the algorithm is obviously correct.

Induction hypothesis. The algorithm is correct for a tree with $2(n-1)$ nodes.

Induction step. We need to prove that the algorithm works for a tree T with $2n$ nodes. Denote the tree after removal by T', which the algorithm can make correct decision about according to the induction hypothesis. If T has a perfect matching M, we should have $(v_L, v_p) \in M$, for otherwise there is no node able to match with v_L. So $M - (v_L, v_p)$ is a perfect matching of T', which can be correctly obtained according to induction hypothesis. If T does not have a perfect matching, the algorithm surely cannot find one.

Time complexity. Since each node and edge is deleted at most once, the time complexity of the algorithm is $O(|V| + |E|)$.

Problem 2

Consider you are a driver, and you plan to take highways from A to B with distance D. Since your car's tank capacity C is limited, you need to refuel your car at the gas station on the way. We are given n gas stations with surplus supply, they are located on a line together with A and B. The i-th gas station is located at d_i that means the distance from A to the station, and its price is p_i for each unit of gas, each unit of gas exactly support one unit of distance. The car's tank is empty at the beginning and so you can assume there is a gas station at A. Design efficient algorithms for the following tasks.

(a) Determine whether it is possible to reach B from A.

(b) Minimize the gas cost for reaching B.

Answer Referred by Yuhao Zhang. We solve two sub questions together, i.e., if we can reach B, we will output the minimum cost, otherwise, we will output "impossible". We assume that the input is given in the ascending order of the distance, otherwise, we can sort them first that additionally costs $O(n \log n)$ time.

The algorithm is built on a greedy approach. We keep driving stations by stations from A to B, Whenever we are at a station i with G unit of gas in the tank, we determine how much we need to refuel by the following steps.

1. First, we find the nearest cheaper gas station j (if exists) after i. (i.e., we find the first gas station j after i such that $p_j < p_i$.) If j does not exist, then we manually fix $j = n + 1$, and $d_j = D$, to represent the location of B, we can also view B as the $(n + 1)$-th station with distance D and the cheapest price.
2. If $d_j \leq d_i + C$, we refuel the tank to $d_j - d_i$ and pay $\max\{d_j - d_i - G, 0\} \cdot p_i$.
3. Otherwise, if $d_j > d_i + C$, we refuel the tank to C and pay $(C - G) \cdot p_i$.

Correctness of The Greedy. Following the greedy approach above, we prove that we can reach B with the minimized cost if it is possible. At first, if our algorithm can not reach B, then it must fail to reach the next station at a station i. Following the greedy approach, we will fill up the tank in this case, and thus $i + 1$ should be more than C apart from i. Hence, reaching the station $i + 1$ is impossible.

Then, we prove the optimality of the cost. Let $g(i)$ be the refueling unit of gas at station i in our algorithm, $G(i) = \{g(1), g(2), ..., g(i)\}$ be the set of strategy until i, we prove that $G(i)$ is a subset of one optimal strategy by induction. Then, it implies that $G(n)$ is one of the optimal strategy. We start from the base case that $G(0) = \emptyset$ is a subset of one optimal solution, then we assume that the strategy before i (i.e., $G(i-1)$) is a subset of one optimal solution, we prove that if we refuel $g(i)$ at station i (i.e., $G(i) = G(i-1) \cup g(i)$), we are still in a subset of one optimal solution. Suppose not, there is an optimal solution contain as a subset $G'(i) = G(i-1) \cup g'(i)$, where $g'(i) \neq g(i)$. Consider the two choice made by our algorithm at i:

- At first, we discuss the case that the first cheaper station j (or $j = n+1$) has distance $d_j \le d_i + C$. If $g'(i) > g(i)$, consider the optimal strategy $G'(n) \supset G'(i)$, and let us do the following modification: decrease $g'(i)$ to $g(i)$ and increase $g'(j)$ to $g'(j) + g(i) - g'(i)$. It is easy to check that the strategy after modification is still feasible because we can still reach j with $g(i)$ refueling at i, and the total cost will decrease because p_j is cheaper than p_i, which is a contradiction. If $g'(i) < g(i)$, there should not be enough gas to reach j only by $G'(i)$, so that the optimal strategy $G'(n) \supset G'(i)$ should refuel some gas after i and before j to support the car to reach j, and we have $\sum_{k=i}^{j-1} \ge g(i)$. Then, we can increase $g'(i)$ to $g(i)$ and decrease totally $g(i) - g'(i)$ unit of refueling among $g(k)$ from $i+1$ to $j-1$. We can easily check that it is still a feasible strategy, with at least the same cost as $G'(n)$ because all p_k can not be cheaper than i. Thus, we have that the modified strategy is still an optimal strategy, and it is a super set of $G(i) = G(i-1) \cup g(i)$.

- Then, we discuss the case that the next cheaper station j is not reachable ($d_j > d_i + C$). In this case, we fill up the tank, so the only possible case is $g'(i) < g(i)$. Now because we need reach the location $d_i + C < D$, similar as before, we should have $\sum_{k=i}^{j'} g(k) \ge g(i)$ where j' is the farthest reachable station from i with $d_{j'} \le d_i + C$. Then, consider the optimal strategy $G'(n) \supset G'(i)$, we can increase $g'(i)$ to $g(i)$ and decrease totally $g(i) - g'(i)$ unit of refueling among $g(k)$ from $i+1$ to j'. The modified strategy is still feasible with no larger cost because all p_k is not cheaper than p_i. It means the modified strategy is also optimal, and it is a super set of $G(i) = G(i-1) \cup g(i)$.

The Running Time. Following the greedy approach, we can simply formalize the algorithm. We let the car move from the first station, determine how much gas we should refuel, and move to the next. In each round, we should at most pay $O(n)$ time to find j, so the time complexity is $O(n^2)$ totally.

Improve the running time by priority queue. The slowest part in the previous analysis is that we need $O(n)$ time to find the first j that is cheaper than i. We improve this part by priority queue. We start from the end (i.e., $i = n$), and we view $n+1$ as the cheapest station of all, so we have $f[i] = n+1$. Then we calculate $f[i]$ from n to 1 with a priority queue. The priority queue stores the potential nearest cheaper stations in ascending order of the distance from A. It is easy to check that if a station j has larger distance than j', and p_j is not cheaper than $p_{j'}$, then j is impossible to play as the nearest cheaper station for those stations before j'. So the priority queue sorted by ascending order of the distance is also sorted by the descending order of the price. When we calculate $f[i]$, we should find from the first station in the queue, until the first one that is cheaper than i, and it is exactly $f[i]$. At the same time, when we pass these stations in the queue that are not cheaper than i, it means that we can pop them from the list because they all have larger distance than i. After removing, we push i into the queue as the first station. At the end, because each station can be at most pushed into the list once, be passed and popped once, we can calculate all $f[i]$ in totally $O(n)$ time.

Finally, after we spend $O(n)$ time to create the array of $f[i]$, we can easily use $O(1)$ time to find j in each round i with the help of $f[i]$. The total time is bounded by $O(n)$.

Problem 3

The problem is concerned with scheduling a set J of jobs $j_1, j_2, \ldots, j_n$ on a single processor. In advance (at time 0) we are given the earliest possible start time s_i, the required processing time p_i and deadline d_i of each job j_i. Note that $s_i + p_i \leq d_i$. It is assumed that s_i, p_i and d_i are all integers. A schedule of these jobs defines which job to run on the processor over the time line, and it must satisfy the constraints that each job j_i starts at or after s_i, the total time allocated for j_i is exactly p_i, and the job finishes no later than d_i. When such a schedule exists, the set of jobs is said to be feasible. We allow a preemptive schedule, i.e., a job when running can be preempted at any time and later resumed at the point of preemption. For example, suppose a job j_i has start time 6, processing time 5 and deadline 990, we can schedule j_i in one interval, say, $[6, 11]$, or over two intervals $[7, 8]$ and $[15, 19]$, or even three intervals $[200, 201]$, $[300, 301]$, $[400, 403]$.

(*a*) Give a set of jobs that is not feasible, *i.e.*, no schedule can finish all jobs on time.

(*b*) Design an efficient algorithm to determine whether a job set J is feasible or not. Let D be the maximum deadline among all jobs, *i.e.*, $D = \max_{i=1}^{n} d_i$, you should analyze the running time in terms of n and D. (Tips, you need to prove that if the input job set J is feasible, then the algorithm can find a feasible schedule for J.)

(*c*) Design an efficient algorithm with the running time only in terms of n.

Answer referred by Haoxuan Xu.

(*a*) $(s_1, p_1, d_1) = (0, 5, 6)$, $(s_2, p_2, d_2) = (0, 5, 6)$. Obviously it's not feasible, because they both start at 0, and their deadlines are 6, but the total processing time needed is $5 + 5 = 10 > 6$, so it's not feasible.

(*b*) The algorithm is similar to the one in question 2. We use an array T with size D to maintain that the allocation of time, $T[t] = 1$ meaning that there is one job allocated at time t, $T[t] = 0$ meaning that there is no job allocated at time t.

First of all, we need to sort the jobs in ascending order according to the deadlines, and initialize the array T with 0. Then, take out jobs from the sorted array in order. For example, we now take out a job j_i with s_i, p_i, d_i. Then we begin from s_i to d_i to check that whether there are p_i time are unallocated. More specifically, we can use an index t_i initialized with s_i, current time allocated ct_i initialized with 0, continuously checking that whether $T[t_i]$, if $T[t_i] = 1$, $t_i += 1$, if $T[t_i] = 0$, set $T[t_i] = 1$, $t_i += 1$, $ct_i += 1$. If we find $ct_i == p_i$, we break and continue to check the next job. If we find $t_i > d_i$, then the set J is not feasible and return false. If all jobs are checked to finish, we return true.

Time Complexity. The firsr part of sorting J takes $O(nlogn)$. The second part of checking one job takes at most $O(d_i - s_i) = O(D)$. So checking every jobs takes $O(nD)$ at most. In conclusion, the time complexity of the algorithm is $O(nlogn + nD)$.

Correctness. We need to prove that if the input J is feasible, then the algorithm can find a feasible schedule. But we can also prove that if the algorithm return False and find that the input J is not feasible, then J is really infeasible, there is no possible schedule to finish them, and if the algorithm return True, obviously there

Algorithm 2 FeasibleJobSet Algorithm

```
Input: Job set $J$, $J[i] = (s_i, p_i, d_i)$.
 1: Sort $J$ according to deadlines $d$ in ascending order using quick sort algorithm, get
    the maximum deadline $D$.
 2: Initialize array $T$ of size $D$ with 0.
 3: for all $j_i$ in $J$ do
 4:    $ct_i \leftarrow 0, t_i \leftarrow s_i$
 5:    while True do
 6:       if $T[t_i] = 0$ then
 7:          $ct_i \leftarrow ct_i + 1$, $T[t_i] = 1$.
 8:       end if
 9:       $t_i \leftarrow t_i + 1$.
10:       if $ct_i = p_i$ then
11:          break
12:       end if
13:       if $t_i > d_i$ then
14:          return False
15:       end if
16:    end while
17: end for
18: return True
```

is a possible schedule and we can get it by setting $T[t_i]$ from $\{0, 1\}$ to allocated job id and unallocated flag like -1.

Now, we prove the former statement about situation that returning False. When it returns False at job j_i and its $t_i > d_i$, then which means every time unit before d_i is allocated but there is still some time need to finish j_i. The only way to finish j_i is that we set $p_i - ct_i$ time units before d_i to finish j_i instead of their origin allocated jobs. Suppose the origin allocated job is j_j, whose deadline is earlier than j_i, $d_j \leq d_i$. If we do so, then d_j can only be scheduled after d_i, but its deadline is earlier, so we cannot do the change. In conclusion, there is no way to finish j_i before d_i, and the job set J is really infeasible. Now, we finish the proof of the correctness.

(*c*) I think the main cost is that for one time unit t, there might be many jobs which checks for it, so we need to improve it.

The first way in my mind is that t_i shouldn't add 1 once and check again, it should jump to the next available time unit. So, we might borrow the idea or data structure for malloc function in memory management, use a explicit free list to store all free time units, each continuous free time period is abstracted into a whole, i.e. a struct containing the start of previous free time period, the start of the next free time period, the size of itself. But the time complexity is a little bit complex I'm not sure that it's $O(nlogn)$.

The second way is that we directly allocate time unit instead of jobs and checking each time unit, and this idea is from Shuyang Jiang and Yichen Tao. We use a time t to denote current time, and maintain two priority queues q_s, q_d, one for all unmeet jobs(j_i with $s_i > t$) order by the start time, one for all hung jobs(j_i with $s_i < t$) but priority is lower than the current scheduled job order by the deadline. Then we initialize the current time with 0, and continue to do so, suppose we have a job j_c in scheduling, and get job j_i from q_s and next start time ns, then we add the job with later deadline to q_d, and have the new current job j'_c, then we allocate all time before ns to j'_c, if it can be finished before ns, then continue to get a job from j_j from q_d, and allocate its time until we get ns, then we get a new job from q_s, repeat to do so.

Time Complexity. The time for selecting one job is at most $O(logn)$, and we select each jobs $O(n)$, so the total time complexity is $O(nlogn)$.

Problem 4

Makespan Minimization. Given m identical machines and n jobs with size $p_1, p_2, \ldots, p_n$. How to find a feasible schedule of these n jobs on m machines to minimize the *makespan*: the maximal completion time among all m machines?
Recall that we have introduced two greedy approaches in the lecture.

- **GREEDY:** Schedule jobs in an arbitrary order, and we always schedule jobs to the earliest finished machine.
- **LPT:** Schedule jobs in the decreasing order of their size, and we always schedule jobs to the earliest finished machine.

We have proved that GREEDY is 2-approximate and LPT is 1.5-approximate, can we finish the following tasks? (Please write down complete proofs.)

(a) Complete the proof that LPT is 4/3-approximate, (*i.e.* LPT $\leq 4/3 \cdot$ OPT).
(b) Prove that GREEDY is $(2 - 1/m)$-approximate, (*i.e.* GREEDY $\leq (2 - 1/m) \cdot$ OPT).
(c) Prove that LPT is $(4/3 - 1/3m)$-approximate, (*i.e.* LPT $\leq (4/3 - 1/3m) \cdot$ OPT).

Answer referred by Yichen Tao.

(a) Without loss of generality, suppose that $p_1 > p_2 > \ldots > p_n$, and that the last finished job is p_n, since if there are jobs that finishes last but is not scheduled last, deleting them will only make the case better. We consider two cases here: $p_n \leq \frac{1}{3} \cdot$ OPT and $p_n > \frac{1}{3} \cdot$ OPT.

If $p_n \leq \frac{1}{3} \cdot$ OPT, the analysis is similar to that of the 3/2-approximation. Suppose p_n starts at time t according to LPT schedule. We can know that all machines are busy before the time t, so total workload $\sum p_i$ should be no less than tm. Hence we have OPT $\geq t$, and ALG $= t + p_n \leq$ OPT $+ \frac{1}{3} \cdot$ OPT $= \frac{4}{3} \cdot$ OPT.

Otherwise, when $p_n > \frac{1}{3} \cdot$ OPT we can prove an even stronger conclusion: LPT $=$ OPT. We prove this by contradiction. Suppose job with size p_l is the first scheduled job violating OPT. Then before scheduling this job, each machine should have at least one and at most two jobs scheduled, since all jobs are larger than $\frac{1}{3} \cdot$ OPT. Suppose there are i machines with one job scheduled on, and these jobs should have their sizes larger than OPT $- p_l$. We can these jobs "*long jobs*" and the other jobs "*short jobs*". There are $(m - i)$ machines with two short jobs, and p_l is still there waiting to be scheduled, so the number of short jobs is $2(m - i) + 1$ if we do not consider the job after the l-th.

To this point, we find that it is impossible to satisfy OPT with any schedule even considering only the first l jobs. The *i long jobs* would have taken up i machines because the sizes of all other jobs are no less than p_l. There are only $(m-i)$ machines left. As the job sizes are all larger than $\frac{1}{3}\cdot\text{OPT}$, each machine can handle at most 2 short jobs. Thus the left machines can only handle $2(m-i)$ *short jobs*. However, there are $2(m-i)+1$ short jobs, which leads to contradiction. Hence, our assumption is false, and the conclusion is proved.

(b) Again, suppose that the last finished job is the one with cost p_n. Denote the starting time of p_n by t.

We can prove that $t \le \frac{1}{m}\sum_{i=1}^{n-1} p_i$. $\frac{1}{m}\sum_{i=1}^{n-1} p_i$ is the time when all machines finish the first $(n-1)$ jobs at the same time. Then when the finishing time of the m machines are not the same, the shortest one should be shorter than $\frac{1}{m}\sum_{i=1}^{n-1} p_i$. According to our algorithm, the n-th job will be assigned to this earliest finishing machine, so t cannot be larger than $\frac{1}{m}\sum_{i=1}^{n-1} p_i$.

Thus, $\text{GREEDY} = t + p_n \le \frac{1}{m}\sum_{i=1}^{n-1} p_i + p_n = \frac{1}{m}\sum_{i=1}^{n} p_i + (1-\frac{1}{m})p_n$.
We then apply two obvious facts: $\text{OPT} \ge \frac{1}{m}\sum_{i=1}^{n} p_i$; $\text{OPT} \ge p_n$. Hence, we can get $\text{GREEDY} \le (2-\frac{1}{m})\text{OPT}$.

(c) Again, suppose that $p_1 > p_2 > \ldots > p_n$, and that the last finished job is p_n. Two cases are considered here: $p_n \le \frac{1}{3}\cdot\text{OPT}$ and $p_n > \frac{1}{3}\cdot\text{OPT}$.

When $p_n \le \frac{1}{3}\cdot\text{OPT}$, similar to the analysis in (b), we have $\text{LPT} = t + p_n \le \frac{1}{m}\sum_{i=1}^{n-1} p_i + p_n = \frac{1}{m}\sum_{i=1}^{n} p_i + (1-\frac{1}{m})p_n \le \text{OPT} + (1-\frac{1}{m})\cdot\frac{1}{3}\text{OPT} = (\frac{4}{3}-\frac{1}{3m})\cdot\text{OPT}$.

In another case, when $p_n > \frac{1}{3}\cdot\text{OPT}$, it can be asserted that LPT gives the optimal solution according to our analysis in (a).

To sum up, LPT is $(\frac{4}{3}-\frac{1}{3m})$-approximate.