SHANGHAI JIAO TONG UNIVERSITY

# Algorithm Analysis HW:3

Author: *519030910100 Huangji Wang F1903004*

Course: *Fall 2021, AI2615: Algorithm*

Date: *December 1, 2021*

**Problem 1 (25 points)**

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.

This problem asks us to determine the existence of a **perfect matching**, in other words it asks us to divide nodes into pairs with edges in the tree. The main idea is to firstly store the input tree and reversed tree. Then, find the level of tree traversal and mark each node in order of access in the input tree. And then explore the reversed tree by the order. If each node can find a parent that doesn't belong to another node, then the tree has a perfect matching. Otherwise, the tree doesn't have any perfect matching.

The main structure of the algorithm goes like:

1. **Initialize**

   - Store the input tree and reserved tree.

   - Do level traversal of the tree and get the order from biggest to smallest.

2. **Explore by order**

   - If the node is *single*, check it's son in reserved tree (i.e. parent in input tree).

   - If it has no son, then end the algorithm and return **impossible**.

   - If its son is *not single*, then end the algorithm and return **impossible**.

   - Else, set its son *not single* and continue exploration.

Time complexity analysis: $O(V)$ for level traversal. $O(V)$ for exploration. Total time complexity: $O(V)$. Thus this is a linear-time algorithm.

Correctness proof of the greedy idea:

1. Base step: $\emptyset$ is in an OPT.

2. Assumptions: the selected $k-1$ pairs are in an OPT.

3. Induction: after selecting the $k$-th pair, we are still in an OPT.

   Proof:

   Suppose we don't select the $k$-th pair and there are n nodes in the tree. Then, it means that if we give up the edge between the node that has $index = n - 2 * (k-1)$ and its parent, then we can have more pairs. . However, in the input tree, its children have already make pairs. This means that this node cannot be made pair and leads to impossible. For the k-th pair we want to select, it can at least make sure this node cannot be given up. Thus, the method leads to lower pairs and yields a contradiction!

4. Conclusion: after selecting the last pair, it is in an OPT. Nothing can be added, so it is OPT.

**Problem 2 (25 points)**

Consider you are a driver, and you plan to take highways from $A$ to $B$ with distance $D$. Since your cars 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 supports 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) (5 points) Determine whether it is possible to reach $B$ from $A$.

(b) (20 points) Minimized the gas cost for reaching $B$.

**Suppose all the gas stations are sorted by the order of $d_i$ from smallest to biggest.**

(a) For the (a) problem, we should check whether the distance between adjacent gas stations is larger than the capacity $C$. If there exist at least one distance larger than $C$, then it is impossible to reach B from A. Otherwise, it is possible.

The main structure of the algorithm goes like:

1. **Initialize**

   - We don't need to sort $d_i$ from smallest to biggest. No initialization.

2. **Explore**

   - Compute adjacent distance $dist[i] = d_i - d_{i-1}, i \in [1, n]$. $dist[n+1] = D - d_n$.

   - If $dist[i] > C$, end the algorithm and return **impossible**.

   - If $dist[i] \leq C$ for all $i \in [1, n+1]$, return **possible**.

Time complexity analysis: $O(n)$ for adjacent distance calculation and judgement.

(b) Firstly we can make sure that it is possible to reach B from A based on problem(a).

The main idea for this problem is to check whether the price in the i-th gas station is smaller than the next station. If it is smaller, continue to compare and buy more. Otherwise, we just buy $d_i$ gas. In brief, we want to buy the cheap gas as much as possible and buy the expensive gas if necessary(It's the greedy idea). The main structure of the algorithm goes like(0-th gas station means place **A** and $(n+1)$-th gas station means place **B**):

1. **Initialize**

   - Take $p_i$ and $d_i$ as a pair. Don't need sorting.

   - Compute adjacent distance $dist[i] = d_i - d_{i-1}, i \in [1, n]$. $dist[n+1] = D - d_n$.

   - Notation: $max\_dis$: the maximum distance I can reach. Set $max\_dis = 0$.

   - Notation: $cost$: the total gas cost I will pay. Set $cost = 0$.

2. **Explore until reach the $(n+1)$-th gas station B**

   - We are in the $i$-th gas station now.

   - Repeat comparing the price $p_i$ with the next station.

   - Until it is not cheaper or cannot reach with remained capacity $C'$.

   - Buy enough gas with price $p_i$. Update the remained capacity $C'$.

   - Go to **the next** gas station with the help of the gas.

Time complexity analysis: $O(n)$ for checking whether it is possible to reach B from A. $O(n^2)$ for exploration until reach the $(n+1)$-th gas station. Total time complexity: $O(n^2)$.

Correctness proof of the greedy idea:

(a) Base step: $\emptyset$ is in an OPT.

(b) Assumptions: the selected $k-1$ buying strategies are in an OPT.

(c) Induction: after selecting the $k$-th buying strategy, we are still in an OPT.

Proof:

Suppose we don't select the $k$-th strategy. Then, there are two situations.

i. It means if we don't need to compare the gas price between the current station(denoted as $i$-th) and one next station(may be $j$-th, $j > i$), then we can get cheaper cost. Suppose the end station of the $k$-th strategy is $t$-th station. The cost should be at least $p_i * (d_j - d_i) + p_j * (d_t - d_j)$ However, if we choose the $k$-th strategy, it has $p_i * (d_t - d_i)$, which is much smaller than this method. This method leads to more expensive cost and yields a contradiction!

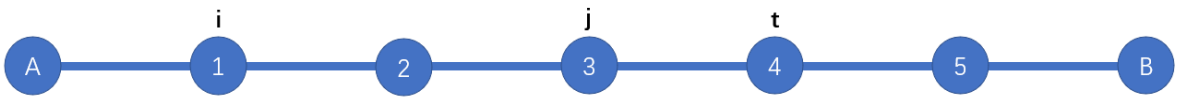


Figure 1: Explanation for i.

ii. It means if we add much gas of high cost in the $k$-th strategy, then we can get cheaper cost. Suppose the end station of this method is $t$-th station, the beginning is $i$-th station and the break point of $k$-th strategy is $j$-th station. However, it will produce cost: $p_i * (d_t - d_i)$. For the $k$-th strategy, the cost is $p_i * (d_j - d_i) + p_j * (d_t - d_j)$. This method leads to more expensive cost and yields a contradiction!


Figure 2: Explanation for ii.

(d) Conclusion: after selecting the last strategy, it is in an OPT. Nothing can be added, so it is OPT.

Actually, the selection **comparing** method has the time complexity of $O(n)$. However, we don't need to iterate all the stations after the $i$-th station because we can save all the situations by pre-treatments. We can use the monotone queue to record all the smallest distance in C. We can do reverse iteration over the original gas station and build the monotone queue using the sliding window method. We should do reverse iterations because the station at the back will influence the station at the front. The sliding window method is almost the same as the problem **11618 Interesting Queue** in acmOJ(**Click here** for detail information). Using this method can get the nearest smallest station at O(1). The algorithm structure of this pre-treatments goes like:

1. **Initialize**

   - Construct array $b[n]$. $b[i]$ means the smallest station in the interval $[d_i, d_i + C]$.

   - Construct an empty queue $q$ that is used to find the smallest station.

2. **Explore from the end to the beginning**

   - We are in the $i$-th gas station now.

   - Check the front value of $q$. If it is over $C$, we pop it from front.

   - Repeat check the rear value of $q$. If it is larger than $p_i$, pop it from rear.

   - Push $p_i$ into the q and update $b[i] = q.front()$

There is one example for the monotone queue. Suppose the distance between each station is 10 and the capacity $C = 30$, the prices are presented below:

$$75, 60, 80, 100, 120, 50, 70$$

The algorithm goes like:

| 75 | 60 | 80 | 100 | 120 | 50 | 70 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| q | contents in array b | | | | | | | | | Explanation: larger pop, smaller keep |
| 70 | | | | | | | | | 70 | No number, input 70 |
| 50 | | | | | | | | 50 | 70 | 70>50, pop 70, push 50 |
| 50 | 120 | | | | | | 50 | 50 | 70 | 50<120, push 120 |
| 50 | 100 | | | | | 50 | 50 | 50 | 70 | 120>100, 50<100, pop 120, push 100 |
| 50 | 80 | | | | 50 | 50 | 50 | 50 | 70 | 100>80, 50<80, pop 100, push 80 |
| 60 | | | | 60 | 50 | 50 | 50 | 50 | 50 | 50:outofrange,80>60,pop 50,80,push 60 |
| 60 | 75 | | 60 | 60 | 50 | 50 | 50 | 50 | 50 | 60<75, push 75 |

Constructing $b$ array will cost $O(n)$ while each iteration call cost $O(1)$. Therefore, we can find the smallest station and not over the capacity within $O(n)$. It means that the total complexity is $O(n)$.

**Problem 3 (25 points)**

The problem is concerned with scheduling a set $J$ of jobs $j_1$, $j_2$, ..., $j_n$ 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) (5 points) Give a set of jobs that is not feasible, i.e., no schedule can finish all jobs on time.

(b) (20 points) 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) (Bonus: 5 points) Design an efficient algorithm with the running time only in terms of $n$.

(a) The table below shows one example that is not feasible.

| $j_i$ | $s_i$ | $p_i$ | $d_i$ |
|---|---|---|---|
| $j_1$ | 0 | 10 | 15 |
| $j_2$ | 5 | 10 | 15 |

Table 1: The set of jobs that is not feasible

(b) The key idea for this problem is to sort all the jobs by $s_i$ from smallest to biggest(if the same, sort $d_i$ from smallest to biggest). Then, try to fill all the jobs with possible processing time. If there is no schedule place to fill with, then this job set $J$ is not feasible. Otherwise, it is feasible.

The main structure of the algorithm goes like:

1. **Initialize**

- Initialize time array with size $|D|$. $D[i] = 0$ implies there is no job at time $i$.

- Take $s_i$, $p_i$ and $d_i$ as one triple for all $i \in [1, n]$.

- Sort $s_i$ from smallest to biggest. If $s_i = s_j$, sort $d_i$ the same way.

2. **Explore until we find the schedule**

   - Choose the time $i$ that is not used($D[i] = 0$). Do iteration over the sorted jobs.

   - Find the first job $j$ that can fit in the space($s[j] \leq i \leq d[j] and p[j] > 0$).

   - Update $D[i] = 1$, $p_j = p_j - 1$.and continue to explore the next time $i = i + 1$.

3. **Result**:

   - If all the works are allocated correct space, return **feasible**.

   - Else, return **not feasible**.

Time complexity analysis: $O(D)$ for time array initialization. $O(n \log n)$ for initial sorting. There are $|D|$ time sizes and each time allocation needs at most $n$ iterations to find the correct job, leading to $O(nD)$. At last, we do iteration over the jobs to check feasible situations, costing $O(n)$. Total time complexity: $O(n(D + \log n + 1)) = O(n(D + \log n))$

Correctness proof of the greedy idea:

(a) Base step: $\emptyset$ is in an OPT.

(b) Assumptions: the selected $k - 1$ allocations are in an OPT.

(c) Induction: after selecting the $k$-th allocation, we are still in an OPT.

Proof:

Suppose we don't select the $k$-th allocation. Then, it means if we don't need to choose the $k$-th earliest finish time job, then we can get more jobs in the schedule. This means next job we choose(denote as $j$) will have the property that $s_j > s_k$ or $s_j = s_k \&\& d_j > d_k$. Then, we will talk about these two situations.

(i) $s_j = s_k \&\& d_j > d_k$

Actually, this condition is the same as the condition in **Greedy Homework Scheduling** in **Slide 10**. If we don't choose the $k$-th job, the $j$-th job we choose cannot have more jobs because if there is one job(denoted as $t$) that can be inserted into $j$-th and $k$-th jobs, then this choose method can lower the number of jobs, which leads to wrong condition and yields a contradiction! Thus,

when two jobs have the same start time, the closest deadline job should be finished first.

(ii) $s_j > s_k$

When $s_j > s_k$, there may be empty spaces in the interval $[s_k, s_j]$ and it can contain one part of one job. If we choose $j$-th job, then it may lower the number of jobs, which yields a contradiction!

(d) Conclusion: after selecting the last allocation, it is in an OPT. Nothing can be added, so it is OPT.

(c) For the initial greedy idea, we hope to fill all the blank space in each iteration. We find that the iteration over $|D|$ will cost more time than we think. Thus, we try to optimize the iteration method. We use one heap to help allocate the space of $|D|$. The main idea is: For each step, input all the jobs into the heap(sort by $d_i$) with the same start time. Then, compare current start time with the start time for the next job in the job list. The extra time between current start time and the start time for the next job is the job space to be allocated. Continue the process until the job list is empty. Then we do iteration in the remaining jobs and end the algorithm.

The main structure of the algorithm goes like:

1. **Initialize**

   - Initialize current time pointer = 0 and the heap(sort by $d_i$ from smallest to biggest).

   - Take $s_i$, $p_i$ and $d_i$ as one triple for all $i \in [1, n]$.

   - Sort $s_i$ from smallest to biggest. If $s_i = s_j$, sort $d_i$ the same way.

2. **Explore until the job list is empty**

   - Choose one batch jobs with same start time, remove them from the job list and put them into the heap.

   - If the job list is empty, end the exploration and turn to **Step 3**.

   - Else, calculate the extra time between current time and the start time of the first job in the job list.

   - Repeat allocating the extra time to the job on the top of the heap by updating the processing time(if one job is allocated enough time, pop it) until the heap is empty or lack of time or lack of space.

- If it is break due to lack of space, return not feasible.

- Else, update current time by current time = the start time of the removed jobs.

3. **Explore until the heap is empty**

   - Repeat allocating the extra time to the job on the top of the heap by updating the processing time(if one job is allocated enough time, pop it) until the heap is empty or lack of time or lack of space.

   - If it is break due to lack of space, return not feasible.

4. **Result**: return **feasible**.

Correctness proof of the greedy idea: The same as problem(b) and I don't prove it again.

Time complexity analysis: Initialize $O(n \log n)$. Exploring, removing and inputting jobs costs at most $O(n \log n)$ because the jobs can be only inputted and popped only once. And for each input operation there must be one update operation. Consider the update operation, there has two situations.

1. Some jobs are popped due to the update operation.

   Then, the update time is exactly the total popped time of these popped jobs, which is at most $O(n \log n)$.

2. One job is updated due to the update operation.

   Then, the update time is exactly $O(\log n)$ due to the inserting cost of the heap.

Therefore, the total time cost is $O(n \log n)$.

I use one example to help understand my algorithm.

**Example works to be done:**

- (1) 0 2 3
- (2) 0 2 8
- (3) 0 3 11
- (4) 3 2 7
- (5) 5 1 6
- End of time array is: 1 1 2 4 4 5 2 3 3 3

The step of my algorithm will goes like:

1. **Initialize**

   - current time = 0. The sorted job list is: (1)(2)(3)(4)(5).

2. **Explore until the job list is empty**

   [I] Put (1)(2)(3) into the heap because they all have the same start time.

   - current time = 0 and the first start time is 3. We have 3 time to allocate.

   - We can finish job (1). Pop (1). Finish one part of job (2). Update job (2) by 0 1 8.

   - Update current time = 3.

   [II] Put (4) into the heap because it has the unique start time 3.

   - current time = 3 and the first start time is 5. We have 2 time to allocate.

   - We can finish job (4). Pop (4).

   - Update current time = 5.

   [III] Put (5) into the heap because it has the unique start time 5.

   - The job list is empty. End **Step 2**.

3. **Explore until the heap is empty**

   - Allocate the space to job (5) (the top job of the heap) and pop (5).

   - Allocate the space to job (2) (the top job of the heap) and pop (2).

   - Allocate the space to job (3) (the top job of the heap) and pop (3).

4. **Result**: return **feasible**.

Therefore, this job list is feasible:

0(0 2 3)->2(0 2 8)->3(3 2 7)->5(5 1 6)->6(0 1 8)->7(0 3 11)->10(Feasible!)

**Problem 4 (25 points)**

**Makespan Minimization** Given m identical machines and $n$ jobs with size $p_1, p_2, ..., p_n$. How to find a feasible schedule of these $n$ jobs on $m$ machines to minimize the *makespan*: the maximized 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) (10 points) Complete the proof that LPT is 4/3-approximate,

(i.e., LPT $\leq$ 4/3OPT).

(b) (10 points) Prove that GREEDY is (2−1/m)-approximate,

(i.e., GREEDY $\leq$ (2 −1/m)OPT).

(c) (5 points) Prove that LPT is (4/3−1/3m)-approximate,

(i.e., LPT $\leq$ (4/3 − 1/3m)OPT).

(a) When $n = 1$ then this is obviously true since $LPT = OPT \leq 4/3OPT$.

When $n$ is larger, we apply the similar method used to proof 2-approximation.

(a) Base step: $n = 1$ satisfies that $LPT = OPT \leq 4/3OPT$.

(b) Assumptions: the selected $k - 1$ jobs satisfy $LPT \leq 4/3OPT$.

(c) Induction: after selecting the $k$-th job, it still satisfies $LPT \leq 4/3OPT$.

Proof:

Denote the last finished job $p_j$ and the last chosen job $p_n$. We should consider two situations between $j$ and $n$.

(i) $j < n$

We can remove the $n$-th job and this schedule is still in the LPT because $p_j$ is the last finished job. Then there are $(n-1)$-th jobs in the LPT. By induction assumptions $LPT \leq 4/3OPT$ when there are $(n-1)$-th jobs in the LPT.

(ii) $j = n$

Notice that $ALG = t + p_n \leq OPT + p_n$. Next, we consider the relationship between $p_n$ and $1/3OPT$. If $p_n \leq 1/3OPT$ then this is obviously true. If $p_n > 1/3OPT$, since $p_n$ is the last chosen job and it is the smallest job, it means there are at most 2 jobs in each machine. By the proof procedure of 1.5-approximate LPT, we know LPT is optimal when there are at most 2 jobs in each machine. Thus this is also true.

(d) Conclusion: after selecting the last job, it is in an OPT. Nothing can be added, so it is OPT.

(b) Suppose the last job $p_n$ is chosen at time $t$. Thus, we can ensure that all the machines are busy before $t$ with the Greedy method. Then, $workload \geq tm \rightarrow OPT \geq t$. For the 2-approximate GREEDY, we find that $p_n \leq OPT$ and get $ALG = t + p_n \leq 2OPT$. We can optimize $t$. Note that workload is full before t, and the total work load must be larger than $tm + p_n$. Thus, OPT must be larger than $t + \frac{p_n}{m}$, which implies that $t + \frac{1}{m}p_n \leq OPT$.

Therefore, $t + p_n = t + \frac{1}{m}p_n + (1 - \frac{1}{m})p_n \leq OPT + (1 - \frac{1}{m})OPT = (2 - \frac{1}{m})OPT$

(c) In the same way, we can optimize $t$. Notice that the workload is full before t, and the total work load must be larger than $tm + p_n$. Thus, OPT must be larger than $t + \frac{p_n}{m}$, which implies that $t + \frac{1}{m}p_n \leq OPT$.

Therefore, if $p_n \leq 1/3OPT$, we have

$t + p_n = t + \frac{1}{m}p_n + (1 - \frac{1}{m})p_n \leq OPT + (1 - \frac{1}{m}) * 1/3OPT = (\frac{4}{3} - \frac{1}{3m})OPT$

If $p_n > 1/3OPT$, then this is optimal and thus $t + p_n = OPT \leq (\frac{4}{3} - \frac{1}{3m})OPT$

Above all,

$t + p_n = t + \frac{1}{m}p_n + (1 - \frac{1}{m})p_n \leq OPT + (1 - \frac{1}{m}) * 1/3OPT = (\frac{4}{3} - \frac{1}{3m})OPT$

**Problem 5**

How long does it take you to finish the assignment (include thinking and discussing)? Give a score (1,2,3,4,5) to the difficulty. Do you have any collaborators? Write down their names here.

I take 10 hours in total to finish the assignment(include thinking and discussing).

Difficulty: 4.

Collaborators: Zhiteng Li in Problem 1,3,4.

Thoughts: I know these proof problems are beneficial to our thoughts. However, I found it hard to build the proof structure. And I'm always worried about whether my meanings can be well understood by TAs based on my poor English.