CMSC 451: Lecture 5
Greedy Algorithms for Scheduling

Greedy Algorithms: Before discussing greedy algorithms in this lecture, let us explore the general concept of greedy optimization algorithms. In an *optimization problem*, we are given an *input* and asked to compute a *discrete structure*, subject to various *constraints*, in a manner that either minimizes cost or maximizes some *objective function*. Such problems are extremely common in computation. Given an optimization problem, a fundamental question is whether it can be solved efficiently (as opposed to a brute-force enumeration of all possible solutions). If so, what approach should be used?

In most optimization algorithms the final structure is based on a series of selections. A simple design technique for optimization problems is based on a *greedy* approach, which builds up a solution by repeatedly selecting the *best alternative* in each step. (In particular, each choice is made without regard as to how this choice may interfere with later choices, and once a choice is made, it is never revoked.) When applicable, this method can lead to very simple and efficient algorithms. Even if a greedy algorithm does not yield the optimal solution, it sometimes produces a good approximation or a starting point for more sophisticated search algorithms.

Interval Scheduling: In this lecture, we discuss a number of problems motivated by applications in resource scheduling. In all instances we have one or more resources and a collection of requests to use these resources. We want to schedule all or some of these requests, subject to the limitations of our resources. As an example, suppose you work for the Parks and Recreation division of your community, and people want to reserve time at picnic tables at a local park. You want to write an algorithm to assign picnic tables to requests.

Our first problem is called *interval scheduling*. We are given a set $R = \{r_1, \ldots, r_n\}$ of n *activity requests* that are to be scheduled to use some resource. Each request r_i is associated with a given *start time* s_i and a given *finish time* f_i. For example, request indicates the desire to use the one picnic table in your park. (You work for a really cheap city!) Given that two groups cannot use the same picnic table at the same time, the objective is to grant as many of the requests as possible, but only one group can use the picnic area at a time.

We say that two requests r_i and r_j *conflict* if their start-finish intervals overlap, that is,

$$[s_i, f_i] \cap [s_j, f_j] \neq \emptyset.$$

(Note that by this criteria, we do not allow finish time of one request to overlap the start time of another one, but this is easily remedied in practice by shaving a tiny time period off the finish times.) Here is a formal problem definition.

Interval scheduling problem: Given a set R of n requests with start-finish times $[s_i, f_i]$ for $1 \leq i \leq n$, determine a subset of R of maximum cardinality consisting of requests that are mutually non-conflicting.

An example of an input and a possible optimal solution is shown in Fig. 1 (another optimal solution is $\{5, 6, 8\}$). Notice that goal here is to maximize the *number* of requests that are

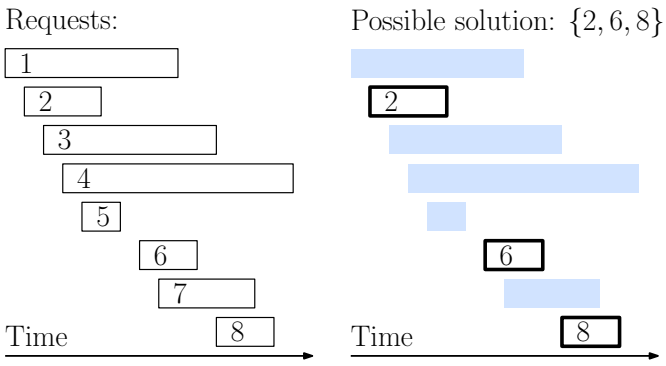

Fig. 1: An input and possible solution to the interval scheduling problem.

scheduled. There are many other alternatives, such as maximizing *utilization*, the total time that the resource is being used.

What is the best way to greedily schedule the largest number of requests? Here are a number of seemingly reasonable approaches that *do not* guarantee an optimal solution:

Earliest Activity First: Repeatedly select the request with the earliest start time, provided that it does not overlap any of the previously scheduled requests.

Shortest Activity First: Repeatedly select the request with the smallest duration ($f_i - s_i$), provided that it does not conflict with any previously scheduled requests.

Lowest Conflict Activity First: Repeatedly select the request that conflicts with the smallest number of remaining requests, provided that it does not conflict with of the previously scheduled requests. (Note that once an request is selected, all the conflicting requests can be effectively deleted, and this affects the conflict counts for the remaining requests.)

As an exercise, show (by producing a counterexample) that each of the above strategies may fail to generate an optimal solution for a given set of requests.

Earliest Finish First: If at first you don't succeed, keep trying. Here, finally, is a greedy strategy that does work. Intuitively, whenever we start a request, we want it to end as soon as possible to allow other requests to begin. This suggests that, among the non-conflicting requests, we repeatedly select the request that finishes first. Call this strategy *Earliest Finish First* (EFF). The pseudo-code is presented in the code-block below. It returns the set S of scheduled requests.

An example is illustrated in Fig. 2, where the requests are numbered in finish-order. Activity 1 is scheduled first. It conflicts with requests 2 and 3. Then request 4 is scheduled. It conflicts with requests 5 and 6. Finally, request 7 is scheduled, and it interferes with the remaining requests. The final output is $\{1, 4, 7\}$. Note that this is not the only optimal schedule. $\{2, 4, 7\}$ is also optimal.

The algorithm's correctness will be shown below. The running time is dominated by the $O(n \log n)$ time needed to sort the jobs by their finish times. After sorting, the remaining steps can be performed in $O(n)$ time.

Interval Scheduling by Earliest Finish First

```
greedy-intveral-schedule(s, f) {    // schedule requests using earliest-finish-first
    sort requests by increasing order of finish times
    S = empty                       // S holds the sequence of scheduled requests
    prevFinish = -infinity          // finish time of previous request
    for (i = 1 to n) {
        if (s[i] > prevFinish) {    // request i doesn't conflict with previous?
            append request i to S      // ...add it to the schedule
            prevFinish = f[i]       // ...and update the previous finish time
        }
    }
    return S
}
```

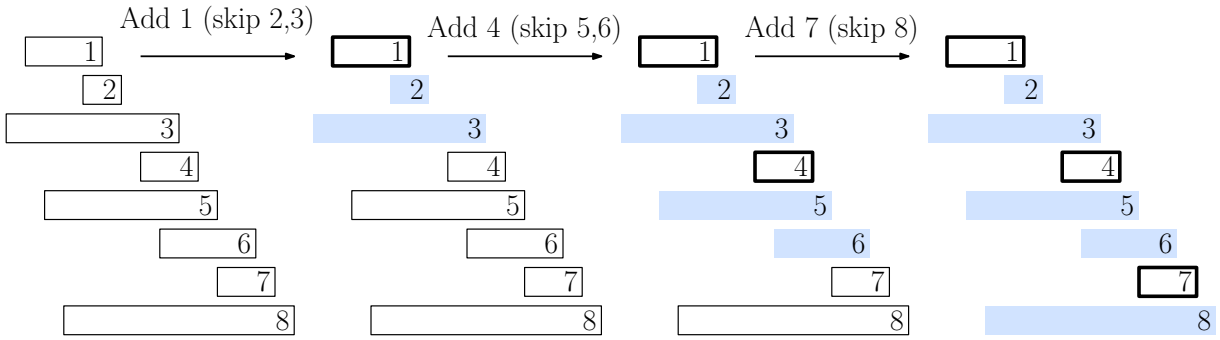

Fig. 2: An example of the greedy algorithm for interval scheduling. The final schedule is $\{1, 4, 7\}$.

Correctness: Let us consider the algorithm's correctness. First, observe that because we only schedule a request if it finishes after the previous one, the final schedule has no conflicting requests and hence is *feasible*.

To establish *optimality*, we use a common approach for greedy algorithms. Suppose towards a contradiction that the final schedule is not optimal. Then there must be some *first decision* where the algorithm differs from the optimal solution. We show that optimal solution can be modified to match the greedy solution at this decision point, without affecting its global quality. By applying this argument inductively, it follows that the greedy solution is as good as any optimal solution, thus it is optimal.

Claim: The EFF strategy yields an optimal solution to interval scheduling.

Proof: Let $O = \langle x_1, \ldots, x_k \rangle$ be the requests of an *optimal solution* listed in increasing order of finish time. (There may be many such solutions, and we may take O to be any of them.) Let $G = \langle g_1, \ldots, g_{k'} \rangle$ be the requests of the EFF solution similarly sorted. If $G = O$, then we are done. Otherwise, observe that since O is optimal, it must contain at least as many requests as G. Hence, there must be a first index j, $1 \leq j \leq k'$, where these two schedules differ. That is, we have:

$$\begin{aligned} O &= \langle x_1, \ldots, x_{j-1}, x_j, \ldots \rangle \\ G &= \langle x_1, \ldots, x_{j-1}, g_j, \ldots \rangle, \end{aligned}$$

where $g_j \neq x_j$.

The greedy algorithm repeatedly selects the request with the earliest finish time that does not conflict with any earlier request. Thus, we know that g_j does not conflict with any earlier request, and it finishes no later than x_j finishes (see Fig. 3).

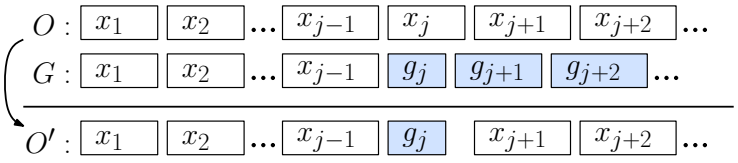

Fig. 3: Proof of optimality for the greedy schedule.

Consider the modified "greedier" schedule O' that results by replacing x_j with g_j in the schedule O (see Fig. 3). That is, $O' = \langle x_1, \ldots, x_{j-1}, g_j, x_{j+1}, \ldots, x_k \rangle$. Clearly, O' is a valid schedule, because g_j finishes no later than x_j, and therefore it cannot create any new conflicts. This new schedule has the same number of requests as O, and so it is at least as good with respect to our optimization criterion.

By repeating this process, we will eventually convert O into G without ever decreasing the number of requests. Therefore, G is optimal.

Interval Partitioning: One shortcoming with interval scheduling is that some requests are not satisfied. Next, let us consider a variant where all the requests can be satisfied. Instead of a single resource, we have an infinite number of possible exclusive resources to use, and we want to schedule *all* the requests using the *smallest* number resources. (The Department of Parks and Recreation can truck in as many picnic tables as it likes, but there is a cost for each table.)

As before, we are given a collection $R = \{r_1, \ldots, r_n\}$ of n requests, each with a start and finish time $[s_i, f_i]$. The objective is to find the smallest number d, such that it is possible to partition R into d disjoint subsets $R_1, \ldots, R_d$, such that the requests of R_j are mutually nonconflicting, for each j, $1 \leq j \leq d$.

We can view this as a *coloring problem*. In particular, we want to assign "colors" to the requests such that two overlapping requests must have different colors. (In our example, each picnic table has its own color. Two overlapping requests must be assigned to different tables, that is, different colors.) If this is done using at most d colors, the result is called a *d-coloring* of the requests. Scheduling all the requests to the minimum number of resources is equivalent to finding the smallest d, such that it is possible to d-color the activity requests.

We refer to the subset of requests that share the same color as a *color class*. The requests of each color class are assigned to the same resource. (For example, in Fig. 4(a) we give an example with $n = 12$ requests and in (b) show a 3-coloring. Thus, the six requests labeled 1 can be scheduled in one picnic table, the three requests labeled 2 can be put in a second table, and the three requests labeled 3 can be put in a third table.)

In general, coloring problems are hard to solve efficiently (in the sense of being NP-hard). However, due to the simple nature of intervals, it is possible to solve the interval-partitioning problem quite efficiently by a simple greedy approach. First, we sort the requests in increasing

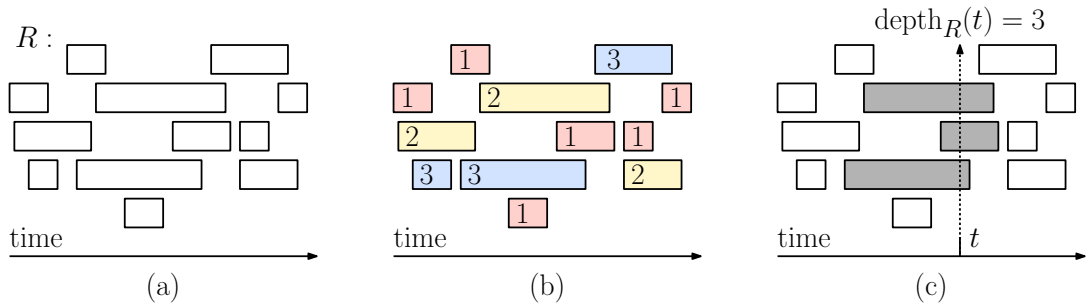

Fig. 4: Interval partitioning: (a) the requests R, (b) a possible 3-coloring, and (c) $\text{depth}_R(t)$.

order of start times. Whenever we encounter a new request, we assign it to the smallest color (possibly a new color) such that this color has not been assigned to any overlapping request. The algorithm is presented in the following code block.

Greedy Interval Partitioning

```
greedy-interval-partition(s, f) {   // interval partition schedule
    sort requests by increasing start times
    for (i = 1 to n) do {           // classify the ith request
        X = emptyset                // X stores excluded colors for request i
        for (j = 1 to i-1) do {
          if ([s[j],f[j]] overlaps [s[i],f[i]]) add color[j] to X
        }
        Let c be the smallest color NOT in X
        color[i] = c
    }
    return color[1...n]
}
```

The solution given in Fig. 4(b) comes about by running the above algorithm. With it's two nested loops, it is easy to see that the algorithm's running time is $O(n^2)$. If we relax the requirement that the color be the smallest available color (instead allowing any available color), it is possible to reduce this to $O(n \log n)$ time with a bit of added cleverness.[1]

Correctness: Let us now establish the correctness of the greedy interval partitioning algorithm. We first observe that the algorithm generates a *feasible* output, since it takes care to never assign the same color to two overlapping requests.

To establish that the algorithm is *optimal*, we will employ another widely used technique in greedy algorithm analysis. We will identify a statistic based on the given instance. We

[1] Rather than have the for-loop iterate through just the start times, sort both the start times and the finish times into one large list of size $2n$. Each entry in this sorted lists stores a record consisting of the type of entry (start or finish), the index of the request (a number $1 \le i \le n$), and the time of the request (either s_i or f_i). The algorithm visits each time instance from left to right, and while doing this, it maintains a stack containing the collection of *available colors*. It is not hard to show that each of the $2n$ entries can be processed in $O(1)$ time. We leave the implementation details as an exercise. Given $O(n \log n)$ time to sort the start and finish times, the total processing time is $2n \cdot O(1) = O(n)$. Thus, the overall running time is $O(n \log n)$.

will show (a) that this statistic is a *lower bound* on the size of *any* feasible solution, and (b) that our algorithm's output is *upper bounded* by this statistic. It follows that our algorithm's output is optimal.

In our case, this statistic, called *depth*, is defined as follows. Given a set R of time intervals, and letting t denote any time instant, define $\text{depth}_R(t)$ to be the number of intervals of R that contain t (see Fig. 4(c)). Next, define $\text{depth}(R)$ to be the maximum depth over all possible values of t:

$$\text{depth}(R) \;=\; \max_{t \geq 0} \text{depth}_R(t).$$

Since the requests that contribute to $\text{depth}_R(t)$ conflict with one another, clearly we need at least this many resources to schedule these requests. Therefore, we have the following lower bound on the number of colors.

Claim: Given any set R of intervals, for any d-coloring, we have $d \geq \text{depth}(R)$.

Next, we show that our algorithm achieves this as an upper bound.

Claim: Given any set R of intervals, the greedy partitioning algorithm generates a d-coloring, where $d \leq \text{depth}(R)$.

Proof: It will simplify the proof to assume that all start and finish times are distinct. (We can always perturb them infinitesimally to guarantee this.) We will prove a stronger result, namely that at any time t, the number of colors in use at time t is at most $\text{depth}_R(t)$. The claim follows by taking the maximum over all times t.

Assume that requests are sorted by start times. Suppose towards a contradiction that there is some time t such that greedy uses more than $\text{depth}_R(t)$ colors. The first such time must be the start time s_i of some request. Consider the time t immediately prior to s_i. Let d denote the number of colors used by greedy at time t. Since s_i is the first violation, we have $d \leq \text{depth}_R(t)$. When s_i is seen, the depth increases by $+1$, so $\text{depth}_R(s_i) = \text{depth}_R(t) + 1$. Since the excluded set X used in the algorithm has d elements, there is an unused color among the first $d+1$ colors, which implies that the algorithm uses at most $d+1$ colors at time s_i. Thus, we are using at most $d + 1 \leq \text{depth}_R(s_i)$ colors, which contradicts the hypothesis that more than $\text{depth}_R(s_i)$ colors were needed at this point.

The above algorithm may seem utterly obvious. But, to see whether you really understand it, consider the following question. If the algorithm was identical, but the intervals were sorted according to some different criterion (not start times), would the result still be feasible? still be optimal?

Scheduling to Minimize Lateness: Finally, let us discuss a problem of scheduling a set of n tasks T where each task is associated with an *execution time* t_i and a *deadline* d_i. The objective is to schedule the tasks, no two overlapping in time, such that they are all completed before their deadline. If this is not possible, define the *lateness* of the ith task to be amount by which its finish time exceeds its deadline. The objective is to minimize the maximum lateness over all the tasks. (As an exmple, consider the assignments given to you from your

various classes. You know how long each assignment takes, and you know when the deadline is. You lose points whenever assignments are late.)

More formally, given the execution times t_i and deadlines d_i, the output is a set of n starting times, $S = \{s_1, \ldots, s_n\}$, for the various tasks. Define the the finish time of the ith task to be $f_i = s_i + t_i$ (its start time plus its execution time). The intervals $[s_i, f_i]$ must be pairwise disjoint. The *lateness* of the ith task is the amount of time by which it exceeds its deadline, that is, $\ell_i = \max(0, f_i - d_i)$. The *maximum lateness* of S is defined to be

$$L(S) \;=\; \max_{1 \le i \le n} \max(0, f_i - d_i) \;=\; \max_{1 \le i \le n} \ell_i.$$

The overall objective is to compute S that minimizes $L(S)$.

Consider the instance shown in Fig. 5(a), where the execution time is shown by the length of the rectangle and the deadline is indicated by an arrow pointing to a vertical line segment. A suboptimal solution is shown in Fig. 5(b), and the optimal solution is shown in Fig. 5(c). The width of each red shaded region indicates the amount by which the task exceeds its allowed deadline. The longest such region yields the maximum lateness.

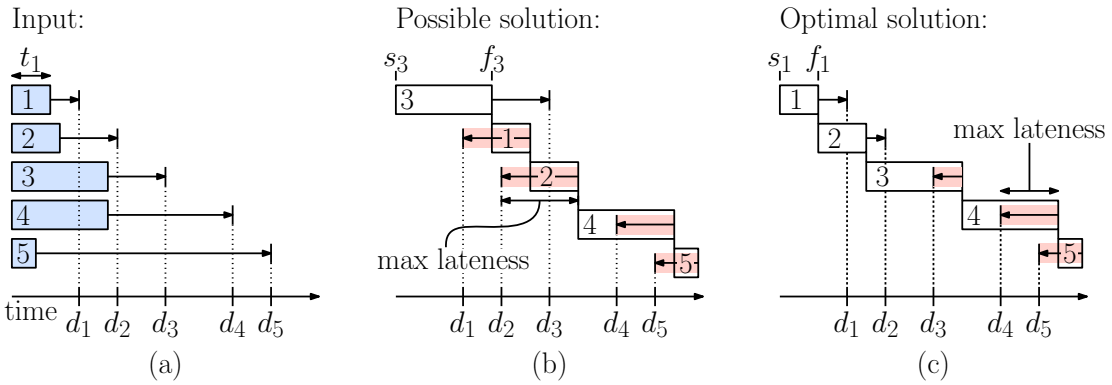

Fig. 5: Scheduling to minimize lateness.

Let us present a greedy algorithm for computing a schedule that minimizes maximum lateness. As before, we need to find a quantity upon which to base our greedy choices. Here are some ideas that *do not* guarantee an optimal solution.

Smallest duration first: Sort tasks by increasing order of execution times t_i and schedule them in this order.

Smallest slack-time first: Define the *slack time* of task x_i as $d_i - t_i$. This statistic indicates how long we can safely wait before starting a task. Schedule the tasks in increasing order of slack-time.

As before, see if you can generate a counterexample showing that each of the above strategies may fail to give the optimal solution.

Earliest Deadline First: So what is the right solution? The best strategy turns out to find the task that needs to finish first and get it out of the way. Define the *Earliest Deadline First*

(EDF) strategy work by sorting the tasks by their deadline, and then schedule them in this order. (This is counterintuitive, because it completely ignores part of the input, namely the running times.) Nonetheless, we will show that this is the best possible. The pseudo-code is presented in the following code block.

Greedy Schedule for Minimizing Lateness

```
greedy-lateness-schedule(t, d) { // schedule to minimize lateness
    sort tasks by increasing deadline (d[1] <= ... <= d[n])
    prevFinish = 0               // f is the finish time of previous task
    for (i = 1 to n) do {
        assign task i to start at s[i] = prevFinish  // start next task
        prevFinish = f[i] = s[i] + t[i]              // its finish time
        lateness[i] = max(0, f[i] - d[i])            // its lateness
    }
    return array s              // return array of start times
}
```

The solution shown in Fig. 5(c) is the result of this algorithm. Observe that the algorithm's running time is $O(n \log n)$, which is dominated by the time to sort the tasks by their deadline. After this, the algorithm runs in $O(n)$ time.

Correctness: It is easy to see that this algorithm produces a valid schedule, since we never start a new job until the previous job has been completed. We will show that this greedy algorithm produces an optimal schedule, that is, one that minimizes the maximum lateness. As with the interval scheduling problem, our approach will be to show that is it possible to "morph" any optimal schedule to look like our greedy schedule. In the morphing process, we will show that schedule remains valid, and the maximum lateness can never increase, it can only remain the same or decrease.

Claim: The EDF scheduling algorithm yields an optimal schedule for maximum lateness.

Proof: To begin, we observe that our algorithm has no *idle time* in the sense that the resource never sits idle during the running of the algorithm. It is easy to see that by moving tasks up to fill in any idle times, we can only reduce lateness. Henceforth, let us consider schedules that are "idle-free." Let G be the schedule produced by the greedy algorithm, and let O be any optimal idle-free schedule. If $G = O$, then greedy is optimal, and we are done.

Otherwise, O must contain at least one *inversion*, that is, at least one pair of tasks that have not been scheduled in increasing order of deadline. Let us consider the first instance of such an inversion. That is, let x_i and x_j be the first two consecutive tasks in the schedule O such that $d_j < d_i$. We have:

(a) The schedules O and G are identical up to these two tasks

(b) $d_j < d_i$ (and therefore x_j is scheduled before x_i in schedule G)

(c) x_i is scheduled before x_j in schedule O

We will show that by swapping x_i and x_j in O, the maximum lateness cannot increase. The reason that swapping x_i and x_j in O does not increase lateness can be seen intuitively

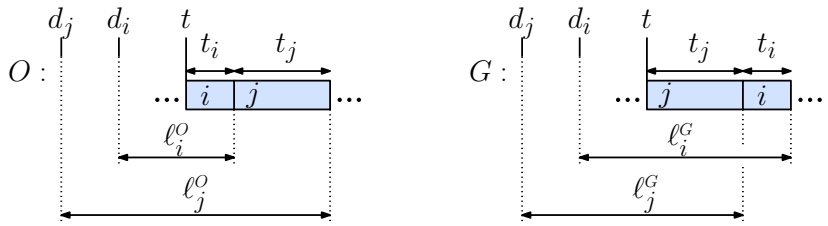

Fig. 6: Optimality of the greedy scheduling algorithm for minimizing lateness.

from Fig. 6. The lateness is reflected in the length of the horizontal arrowed line segments in the figure. It is evident that the worst lateness involves x_j in schedule O (labeled ℓ_j^O). Unfortunately, a picture is not a formal argument. So, let us see if we put this intuition on a solid foundation.

First, let us define some notation. The lateness of task i in schedule O will be denoted by ℓ_i^O and the lateness of task j in O will be denoted by ℓ_j^O. Similarly, let ℓ_i^G and ℓ_j^G denote the respective latenesses of tasks i and j in schedule G. Because the two schedules are identical up to these two tasks, and because there is no slack time in either, the first of the two tasks starts at the same time in both schedules. Let t denote this time (see Fig. 6). In schedule O, task i finishes at time $t+t_i$ and (because it needs to wait for task i to finish) task j finishes as time $t+(t_i+t_j)$. The lateness of each of these tasks is the maximum of 0 and the difference between the finish time and the deadline. Therefore, we have

$$\ell_i^O \;=\; \max(0, t+t_i-d_i) \qquad \text{and} \qquad \ell_j^O \;=\; \max(0, t+(t_i+t_j)-d_j).$$

Applying a similar analysis to G, we can define the latenesses of tasks i and j in G as

$$\ell_i^G \;=\; \max(0, t+(t_i+t_j)-d_i) \qquad \text{and} \qquad \ell_j^G \;=\; \max(0, t+t_j-d_j).$$

The "max" will be a pain to carry around, so to simplify our formulas we will exclude reference to it. (You are encouraged to work through the proof with the full and proper definitions.)

Given the individual latenesses, we can define the maximum lateness contribution from these two tasks for each schedule as

$$L^O \;=\; \max(\ell_i^O, \ell_j^O) \qquad \text{and} \qquad L^G \;=\; \max(\ell_i^G, \ell_j^G).$$

Our objective is to show that by swapping these two tasks, we do not increase the overall lateness. Since this in the only change, it suffices to show that $L^G \le L^O$. To prove this, first observe that, t_i and t_j are nonnegative and $d_j < d_i$ (and therefore $-d_j > -d_i$). Recalling that we are dropping the "max", we have

$$\ell_j^O \;=\; t+(t_i+t_j)-d_j \;>\; t+t_i-d_i \;=\; \ell_i^O.$$

Therefore, $L^O = \max(\ell_i^O, \ell_j^O) = \ell_j^O$. Since $L^G = \max(\ell_i^G, \ell_j^G)$, in order to show that $L^G \le L^O$, it suffices to show that $\ell_i^G \le L^O$ and $\ell_j^G \le L^O$. By definition we have

$$\ell_i^G \;=\; t+(t_i+t_j)-d_i \;<\; t+(t_i+t_j)-d_j \;=\; \ell_j^O \;=\; L^O,$$

and

$$\ell_j^G \;=\; t + t_j - d_j \;\leq\; t + (t_i + t_j) - d_j \;=\; \ell_j^O \;=\; L^O.$$

Therefore, we have $L^G = \max(\ell_i^G, \ell_j^G) \leq L^O$, as desired. In conclusion, we have the following.