CMSC 451: Lecture 7
Greedy Algorithms for Scheduling

Tuesday, Sep 19, 2017

Reading: Sects. 4.1 and 4.2 of KT. (Not covered in DPV.)

Interval Scheduling: We continue our discussion of greedy algorithms with a number of problems motivated by applications in resource scheduling. Our first problem is called *interval scheduling*. We are given a set R of n *activity requests* that are to be scheduled to use some resource. Each activity has a given *start time* s_i and a given *finish time* f_i. For example, these may represent bids to use a picnic area in a neighborhood park. The Department of Parks and Recreation would like to grant as many of the bids as possible, but only one group can use the picnic area at a time.

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

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

(We do not allow finish time of one request to overlap the start time of another one, but this is easily remedied in practice.) Here is a formal problem definition.

Interval scheduling problem: Given a set R of n activity 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 two possible (optimal) solutions is given in Fig. 1. Notice that goal here is to maximize the *number* of activities that are granted (as opposed, say to some other criterion, like maximizing the total time that the resource is being used).

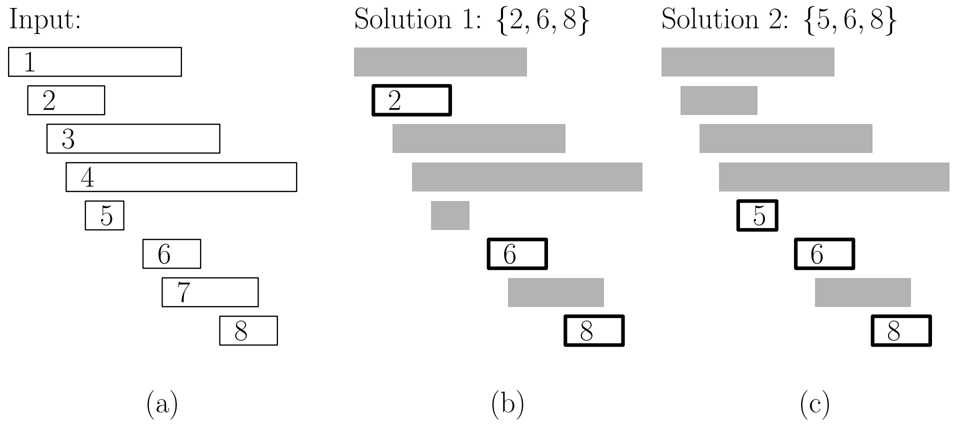


Fig. 1: An input and two possible solutions to the interval scheduling problem.

How do we schedule the largest number of activities on the resource? There are a number ideas on how to proceed. As we shall see, there are a number of seemingly reasonable approaches that *do not* guarantee an optimal solution.

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

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

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

As an exercise, show (by producing a counterexample) that each of the above strategies may not generate an optimal solution.

If at first you don't succeed, keep trying. Here, finally, is a greedy strategy that does work. Since we do not like activities that take a long time, let us select the activity that finishes first and schedule it. Then, we skip all activities that conflict with this one, and schedule the next one that has the earliest finish time, and so on. Call this strategy *Earliest Finish First* (EFF). The pseudo-code is presented in the code-block below. It returns the set S of scheduled activities.

Greedy Interval Scheduling

```
greedyIntervalSchedule(s, f) {      // schedule tasks with given start/finish times
    sort tasks by increasing order of finish times
    S = empty                       // S holds the sequence of scheduled activities
    prev_finish = -infinity         // finish time of previous task
    for (i = 1 to n) {
        if (s[i] > prev_finish) {   // task i doesn't conflict with previous?
            append task i to S      // ...add it to the schedule
            prev_finish = f[i]      // ...and update the previous finish time
        }
    }
    return S
}
```

An example is given in Fig. 2. The start-finish intervals are given in increasing order of finish time. Activity 1 is scheduled first. It conflicts with activities 2 and 3. Then activity 4 is scheduled. It conflicts with activities 5 and 6. Finally, activity 7 is scheduled, and it interferes with the remaining activity. 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.

Correctness: Let us consider the algorithm's correctness. First, observe that the output is a valid schedule in the sense that no two conflicting tasks appear in the final schedule. This is because we only add a task if its start time exceeds the previous finish time, and the previous finish time increases monotonically as the algorithm runs.

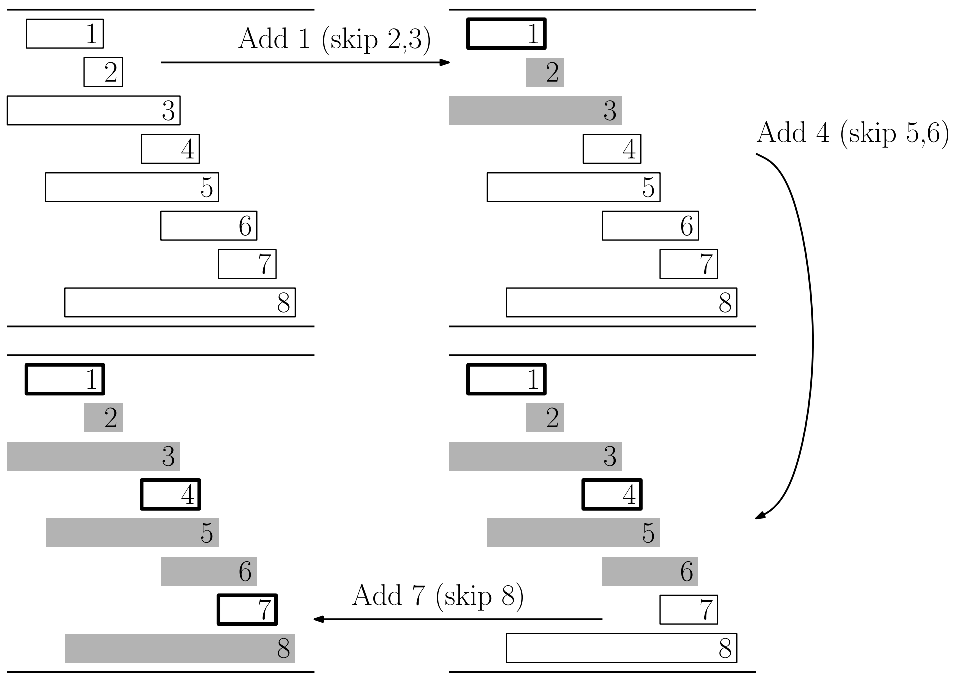


Fig. 2: An example of the greedy algorithm for interval scheduling. The final schedule is $\{1, 4, 7\}$.

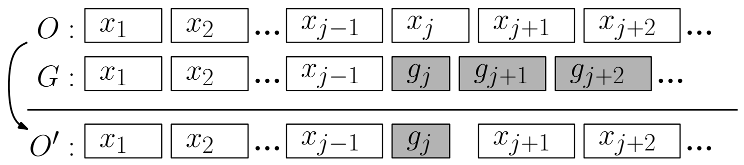
Second, we consider optimality. The proof's structure is worth noting, because it is common to many correctness proofs for greedy algorithms. It begins by considering an arbitrary solution, which may assume to be an optimal solution. If it is equal to the greedy solution, then the greedy solution is optimal. Otherwise, we consider the first instance where these two solutions differ. We replace the alternate choice with the greedy choice and show that things can only get better. Thus, by applying this argument inductively, it follows that the greedy solution is as good as an optimal solution, thus it is optimal.

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

Proof: Let $O = \langle x_1, \ldots, x_k \rangle$ be the activities of an *optimal solution* listed in increasing order of finish time, and let $G = \langle g_1, \ldots, g_{k'} \rangle$ be the activities 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 activities as the greedy schedule, and hence there is a first index j 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 $g_j \neq x_j$. (Note that $k \geq j$, since otherwise G would have more activities than O, which would contradict O's optimality.) The greedy algorithm selects the activity with the earliest finish time that does not conflict with any earlier activity. Thus, we know that g_j does not conflict with any earlier activity, and it finishes no later than x_j finishes.



O : x_1 x_2 … x_{j-1} x_j x_{j+1} x_{j+2} …
G : x_1 x_2 … x_{j-1} g_j g_{j+1} g_{j+2} …
O' : x_1 x_2 … x_{j-1} g_j x_{j+1} x_{j+2} …

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 activities 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 activities. It follows that G is optimal.

Interval Partitioning: Next, let us consider a variant of the above problem. In interval scheduling, we assumed that there was a single exclusive resource, and our objective was to schedule as many nonconflicting activities as possible on this resource. Let us consider a different formulation, where instead we have an infinite number of possible exclusive resources to use, and we want to schedule *all* the activities 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, so it wants to keep the number small.)

As before, we are given a collection R of n activity 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 events 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 activities such that two conflicting activities must have different colors. (In our example, the colors are rooms, and two lectures at the same time must be assigned to different class rooms.) Our objective is to find the minimum number d, such that it is possible to color each of the activities in this manner.

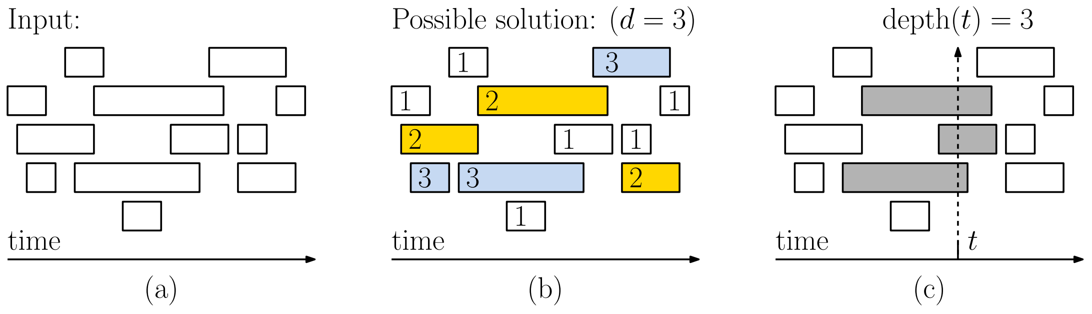


Fig. 4: Interval partitioning: (a) input, (b) possible solution, and (c) depth.

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

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 by increasing order of start times. We then assign each request the smallest color (possibly a new color) such that it conflicts with no other requests of this color class. The algorithm is presented in the following code block.

Greedy Interval Partitioning

```
greedyIntervalPartition(s, f) {      // schedule tasks with given start/finish times
    sort requests by increasing start times
    for (i = 1 to n) do {            // classify the ith request
        E = emptyset                 // E stores excluded colors for activity i
        for (j = 1 to i-1) do {
          if ([s[j],f[j]] overlaps [s[i],f[i]]) add color[j] to E
        }
        Let c be the smallest color NOT in E
        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 never assigns the same color to two conflicting activities. This is due to the fact that the inner for-loop eliminates the colors of all preceding conflicting tasks from consideration. Thus, the algorithm produces a valid coloring. The question is whether it produces an optimal coloring, that is, one having the minimum number of distinct colors.

To establish this, we will introduce a helpful quantity. Let t be any time instant. Define $\text{depth}(t)$ to be the number of activities whose start-finish interval contains t (see Fig. 4(c)). Given an set R of activity requests, define $\text{depth}(R)$ to be the maximum depth over all

[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 event (start or finish), the index of the activity (a number $1 \leq i \leq n$), and the time of the event (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 $2n$ events can be processed in $O(1)$ time. We leave the implementation details as an exercise. The total running time to sort the records is $O((2n)\log(2n)) = O(n \log n)$, and the total processing time is $2n \cdot O(1) = O(n)$. Thus, the overall running time is $O(n \log n)$.

possible values of t. Since the activities that contribute to depth(t) conflict with one another, clearly we need at least this many resources to schedule these activities. Therefore, we have the following:

Claim: Given any instance R of the interval partitioning problem, the number of resources needed is at least depth(R).

This claim states that, if d denotes the minimum number of colors in any schedule, we have $d \geq \text{depth}(R)$. This does not imply, however, that this bound is necessarily achievable. But, in the case of interval partitioning, we can show that the depth bound is achievable, and indeed, the greedy algorithm achieves this bound.

Claim: Given any instance R of the interval partitioning problem, the number of resources produced by the greedy partitioning algorithm is at most depth(R).

Proof: It will simplify the proof to assume that all start and finish times are distinct. (Let's assume that we have perturbed them infinitesimally to guarantee this.) We will prove a stronger result, namely that at any time t, the number of colors assigned to the activities that overlap time t is at most depth(t). The result follows by taking the maximum over all times t.

To see why this is true, consider an arbitrary start time s_i during the execution of the algorithm. Let $t^- = s_i - \varepsilon$ denote the time instant that is immediately prior to s_i. (That is, there are no events, start or finish, occurring between t^- and s_i.) Let d denote the depth at time t^-. By our hypothesis, just prior to time s_i, the number of colors being used is at most the current depth, which is d. Thus, when time s_i is considered, the depth increases to $d+1$. Because at most d colors are in use prior to time s_i, there exists an unused color among the first $d+1$ colors. Therefore, the total number of colors used at time s_i is $d+1$, which is not greater than the total depth.

To see whether you really understand the algorithm, ask yourself the following question. Is sorting of the activities essential to the algorithm's correctness? For example, can you answer the following questions?

- If the sorting step is eliminated, is the result necessarily optimal?
- If the tasks are sorted by some other criterion (e.g., finish time or duration) is the result necessarily optimal?

Scheduling to Minimize Lateness: Finally, let us discuss a problem of scheduling a set of n tasks where each task is associated with a *execution time* t_i and a *deadline* d_i. (Consider, for example, the assignments from your various classes and their due dates.) 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.

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

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

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

An example is shown in Fig. 5. The input is given 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.

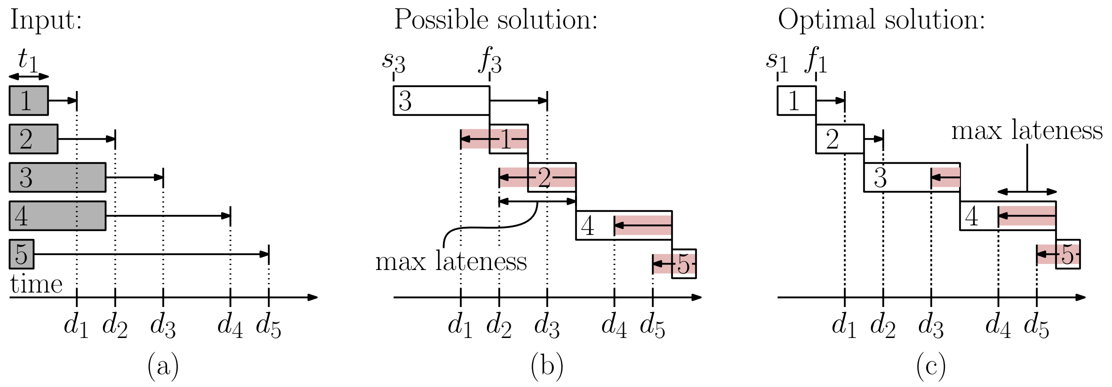


Fig. 5: Scheduling to minimize lateness.

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.

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

```
greedySchedule(t, d) {          // schedule given execution times and deadlines
    sort tasks by increasing deadline (d[1] <= ... <= d[n])
    f_prev = 0                  // f is the finish time of previous task
    for (i = 1 to n) do {
        assign task i to start at s[i] = f_prev  // start next task
        f_prev = 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.

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 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

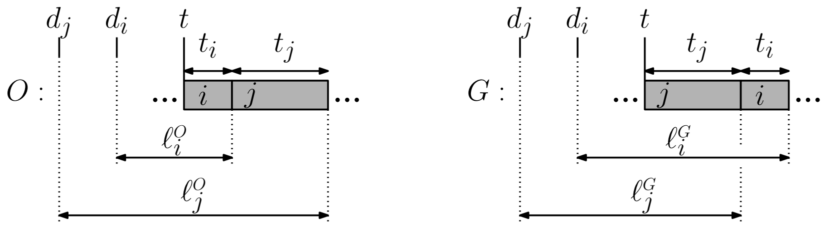


Fig. 6: Optimality of the greedy scheduling algorithm for minimizing lateness.

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

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

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

Claim: The greedy scheduling algorithm minimizes maximum lateness.