CMPUT 498/501: Advanced Algorithms **Fall 2025**

Lecture 1 (Sep 3, 2025): Stable Matching, Greedy & Dynamic Programming

Lecturer: Mohammad R. Salavatipour *Scribe: Keven Qiu and Mohammad Salavatipour*

1.1 Introduction

We start by dicussing some classic algorithm design techniques, namely greedy and dynamic progrmaming.

1.2 Stable Matching

This is a classic problem, also known as stable marriage, that was introduced in a paper by Gale and Shapley in 1962 (two economists). The original question was posed in the setting of assignment college admission process but in practice it arises in many settings. Here we consider the problem in the setting of assignment of hospitals and medical students.

Suppose we have n doctors and n hospitals that need to be matched: each doctor to exactly only hospital and each hospital be given exactly one doctor. Such an assignment is called a **matching**. Each doctor d has a preference (orderd) list of hospitals from the most desirable to the least desirable. Similarly, each hostpital h has an ordered list preference of all the doctors.

Definition 1 *A pair (h, d) for hospital h and doctor d is called unsable if h prefers d over the doctor d' that is assigned to it and d prefres h over the hospital s/he is assigned to.*

For example consider having 3 hospitals $\{h_1, h_2, h_3\}$ and doctors $\{d_1, d_2, d_3\}$ with the following preference lists:

	$1st$	$2nd$	$3rd$
h_1	d_1	d_2	d_3
h_2	d_2	d_1	d_3
h_3	d_1	d_2	d_3

	$1st$	$2nd$	$3rd$
d_1	h_2	h_1	h_3
d_2	h_1	h_2	h_3
d_3	h_1	h_2	h_3

Hospitals' preference list and doctors' lists

Suppose we match them in the following way: (h_1, d_3), (h_2, d_2), and (h_3, d_1). Then the pair h_1, d_2 is unstable since h_1 prefers d_2 over d_3 (we say $d_2 > d_3$) and also d_2 prefers h_1 over h_3 (so $h_1 > h_3$). It is clear to see that if there is an unsable pair then h, d would prefer to break their current assignment and be paired up.

Definition 2 *A matching M is stable if it does not have any unstable pair.*

In the context of "stable marriage" we have n men and n women with their preference lists and the goal is to find a matching of these n men to n women so that there is no unstable pair. It is not immediately obvious that given the preference lists of doctors and hospitals as input there always exists a stable matching. National resident matching program (NRMP) was centralized to match medical school students to hospitals. In 1952 it began to fix unraveling of offer dates. Originally the algorithm used was called the “Boston Pool” algorithm. Here we discuss an algorithm due to Gale-Shapely that proves the existence of a stable matching and also finds one (efficiently).

Algorithm 1 Gale-Shapely Deferred Acceptance Algorithm

Input: Preference list for hospitals and doctors.
Output: Find a stable matching M.

```
1: $M = \emptyset$
2: while there is an unmatched hospital $h$ do
3:     $h$ offers to the next doctor on its list it has not made an offer before
4:     if $d$ has no job then Add $(h, d)$ to $M$
5:     if $d$ has job with $h'$ and $h' > h$ then Do nothing
6:     if $d$ has job with $h'$ and $h > h'$ then
7:         Remove $(h', d)$ from $M$ and add $(h, d)$ to $M$
8: return $M$
```

So each hospitals goes through their list (once) and propose in that order. It is not clear that M at the end is a (perfect) matching, let alone a stable matching.

Lemma 1 *At the end, M is a matching.*

The proof follows from the following two simple observations:

Observation 1: Once a doctor gets a job, then they never become jobless (since they only abandon a hospital h to go to another one h' that is higher in their list).

Observation 2: No hospital is matched to more than one doctor.

So at the end, if there is a hospital h that is no matched to any doctor, then there is a doctor d that is not matched to any hospital. But this means h has never proposed to d while h must have gone through its entire list (a contradiction).

Next we show why M must be a stable matching. Note that in this algorithm hospitals go ”down” in their preference lists and doctors go ”up” in their list (i.e. a doctor changes only to a better hospital).

Lemma 2 *M is stable at the end.*

Proof. By way of contradiction suppose that there is an unstable pair: say (d, h) and (d', h') are paired in M while for d: $h' > h$ and for h': $d > d'$.

Case 1: h' never offered to d. This cannot happen since d is higher on the list of h'.

Case 2: h' made an offer to d: if d rejected then d must be matched to something higher and since it will not go down in his/her list it can't be matched to h later. If d accepted h' then it will not switch to h (or anything lower than h').

■

This matching is in favor of hospitals. We can do the algorithm based on preference lists of doctors instead.

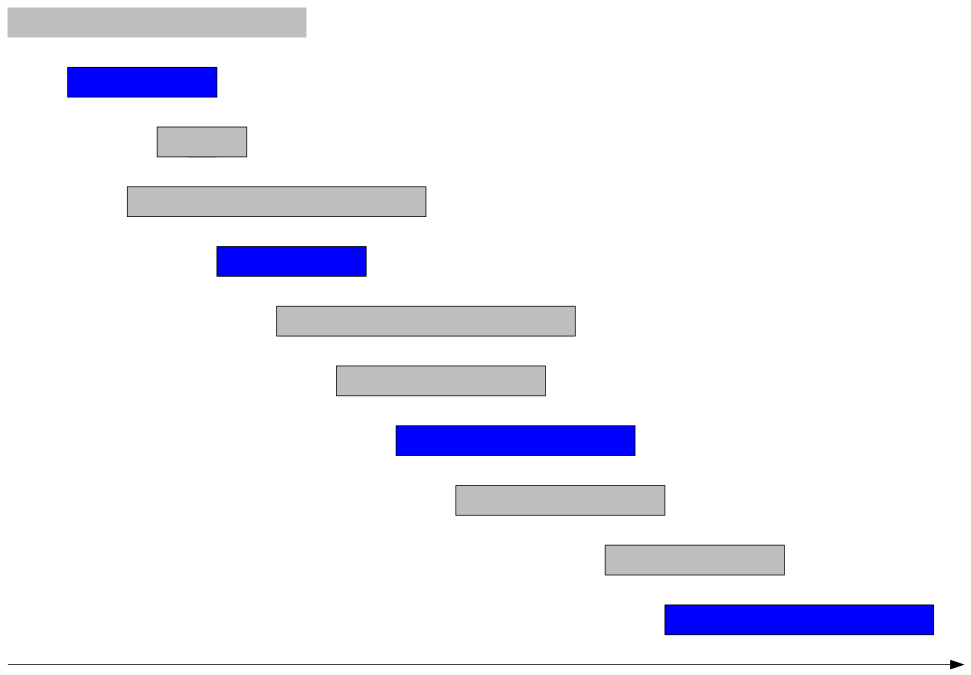


Figure 1.1: A set of jobs; blue ones are compatible.

1.3 Greedy Algorithms

Greedy algorithms are used for optimization problems (e.g. the coin change problem, shortest paths in weighted graphs or computing a minimum spanning tree, many scheduling problems). Some characteristics of these algorithms are: decisions are made locally and often never changed; they are usually efficient/fast and the proofs are often based on induction and uses an exchange argument. However, they have very limited applicability (i.e. for most problems the greedy methods don't work). Below we see some examples for which this method works well.

1.3.1 Interval Scheduling

Scheduling problems are among the most well studied (and applied) problems in optimizations. Suppose we are given a set of n jobs J, with each job j having a start time s_j and finish time f_j. We say two jobs are compatible if their intervals don't overlap (so they could be run on a single processor that can run one job at a time). The goal is to find a largest set of compatible jobs (see Figure 1.1).

There are many ways one may try to design a greedy algorithm for this. For instance, one may try to sort the jobs based on s_j, or duration (i.e. $f_j - s_j$) and greedily pick them as long as they are compatible with the selected ones so far; or sor based on f_j and do this.

If one tries to sort based on s_j (small to large) then it will fail because the example in Figure 1.2.

Sorting by interval length (from smaller to larger) will fail as shown in Figure 1.3

It turns out if we sort by finish time (smaller to larger) then it gives us the optimal schedule. In other words at any given time, pick the job that is feasible (with the jobs selected so far) and has the earliest finish time.

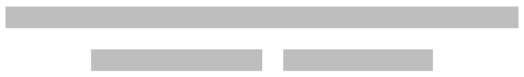


Figure 1.2: Bad example for sorting based on s_j

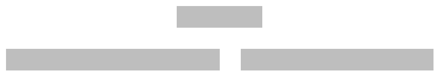


Figure 1.3: Bad example for sorting based on $f_j - s_j$

First, note that the time complexity of this algorithm is $O(n \log n)$: it takes this time to sort them. Also in order to check if each job being considered is compatiple with the schedule so far we only need to keep track of the finish time of the last job added and compare that with the start time of the current job (in $O(1)$ time).

Theorem 1 *The greedy algorithm given finds an optimum schedule.*

Proof. By way of contradiction assume the greedy is not optimal. Let $i_1, \ldots, i_k$ denote set of jobs selected by greedy and $j_1, \ldots, j_m$ denote set of jobs in an optimal solution with $i_1 = j_1, \ldots, i_r = j_r$ for largest possible value of r. Clearly $m > k$. Now j_{r+1} in optimum must finish after i_{r+1} because i_{r+1} has the earliest finish time among all the jobs that are compatible with $i_1, \ldots, i_r$. So we can replace j_{r+1} in optimum with i_{r+1}. This does not interfere with the jobs scheduled later as they all start after the finish time of j_{r+1} and hence after i_{r+1}. This creates another optimum with more common jobs with Greedy, contradicting the definition of r.

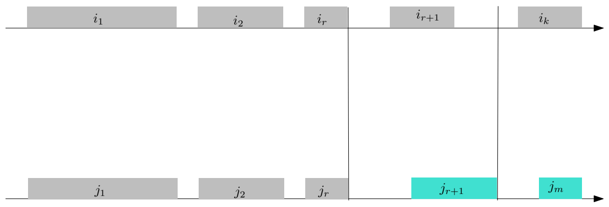


Figure 1.4: Job j_{r+1} can be swapped with i_{r+1} in optimum

■

In fact one can prove by induction that for any $m \geq 1$, after our algorithm completes m intervals, an optimum has completed $\leq m$ intervals.

One can ask an alternative question: what if we have to schedule all the jobs but on possibly more than one machine, while using minimum number of machines?

Exercise: Think of another greedy algorithm for this variant that minimizes the number of machines on which all the jobs can be scheduled.

1.3.2 Minimum Lateness Schedule

Let's consider another scheduling problem that tries to scheules all the jobs while minimizing the lateness of them. In this problem we are given a set of n jobs $J : j_1, \ldots, j_n$ each having a length t_i (processing time) and deadline d_i. If job i starts at time s_i then finishes at time $f_i = s_i + t_i$ and will have lateness $\ell_i = \max\{0, f_i - d_i\}$. The goal is, find an ordering of the jobs to run on a machine (and start times for them) that minimizes $\max_i\{\ \ell_i\}$. We can solve this by greedy. Sort by deadline, i.e. $d_1 \leq d_2 \leq \cdots \leq d_n$. (see example in Figure 1.5)

Algorithm 2 Greedy Interval Scheduling

```
1: Sort jobs based on finish time, $f_1 \leq f_2 \leq \cdots \leq f_n$
2: $S = \emptyset$
3: for $i = 1$ to $n$ do
4:     if $[s_i, f_i]$ does not conflict with anything in $S$ then
5:         Add $i$ to $S$
6: return $S$
```

jobs	1	2	3	4	5	6
t_j	2	3	1	1	3	4
d_j	5	8	12	1	8	11

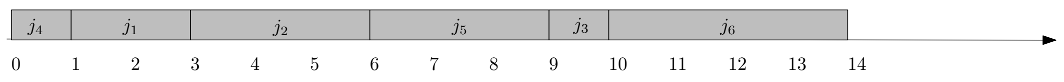


Figure 1.5: A schedule for 6 jobs. Jobs 5 and 6 have lateness of 1 and 3 in this schedule, respectively.

First, note that in any schedule, if there is idle time between two scheduled jobs we can start the latter job right after the finish time of the former without increasing the lateness. Therefore we can assume that:

Observation 1 *There is an optimal solution with no idle time.*

In a schedule S an *inversion* is a pair of jobs i and j where j comes before i but i has a deadline earlier than deadline of j. The following two observations are immediate.

Observation 2 *The earliest-deadline-first schedule is the unique idle-free schedule with no inversions.*

Observation 3 *If an idle-free schedule has an inversion, then it has an adjacent inversion. (Think of it like sorting, if it is not sorted, two adjacent items are in the wrong order.)*

The key observation that helps us to complete the proof is the following. If we exchang two adjacent inverted jobs i and j it reduces the number of inversions by 1 and does not increase the maximum lateness. To see this suppose i, j form an inversion with finish times f_j and $f_i = f_j + t_i$ and lateness ℓ_j and ℓ_i. Suppose we swap i, j. Let ℓ'_i, ℓ'_j be the new lateness of these jobs. Note that the lateness of other jobs do not change and $\max\{\ell'_i, \ell'_j\} \leq \max\{\ell_i, \ell_j\}$

Theorem 2 *Earliest deadline-first schedule S is optimal.*

Proof. Define S^* to be an optimal schedule with the fewest inversions. We can assume S^* has no idle time. There are two cases:

1. S^* has no inversions: Then $S = S^*$.
2. S^* has an inversion: Let i, j be an adjacent inversion. Exchanging jobs i and j decreases the number of inversions by 1 without increasing the maximum lateness. This contradicts the fewest inversion assumption of S^*.

■

1.4 Dynamic Programming

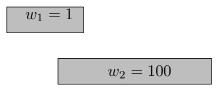
Dynamic programming (DP) is one of the most powerful techniques in design of efficient algorithms. It can sometimes be quite involved and intricate. The basic principle is to break the problem into smaller subproblems, then solving the subproblems and storing the values or solutions to these subproblems into a table, and using these partial solutions recursively to solve bigger subproblems. It often involves some optimization as well. Perhaps the most important step is to identify the proper subproblem and define a table to store the values of the subproblems. Later on we will see how DP can be used to design approximation algorithms for problems that are otherwise difficult to solve exactly.

1.4.1 Weighted Interval Scheduling

As a first example we look at a generalization of the first scheduling problem,

Weighted Interval Scheduling Problem: In this problem we are given a set J of n jobs each with a start time s_i, finish time f_i, and w_i as weight of job i. The goal is to find a subset of compatible jobs (i.e. non-overlapping) with maximum total value.

We saw that greedy works well when $w_i = 1$ for all jobs. This is not the case with weights. See Figure 1.6 for bad examples for two natural greedy algorithms.



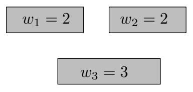


Figure 1.6: *a*) For the instance on the left earliest-finish-time first fails; *b*) for the instance on the right largest w_i first fails.

Consider the jobs in increasing order of finish time; $f_1 \leq f_2 \leq f_n$. For each job j, let $p(j)$ be the largest index $i < j$ such that job i does not overlap with j ($p(j) = 0$ if no such job exists). For now assume our goal is to find the *value* of an optimum solution (instead of the schedule itself). Let opt[j] denote the max weight of a schedule that uses only a subset of jobs in $\{1, \ldots, j\}$. Our goal is to compute opt[n]. It is easy to see that opt[1] $= w_1$.

When considering job j,

1. If optimum of the first j jobs does not include j.

$$\text{opt}[j] = \text{opt}[j-1]$$

2. j If j belongs to the best schedule of the first j jobs, the it must be the last in that schedule and we have to find the best schedule of jobs $1, \ldots, p(j)$.

$$\text{opt}[j] = w_j + \text{opt}[p(j)]$$

We do not know which of the two cases, so opt[0] = 0 and

$$\text{opt}[j] = \max \begin{cases} \text{opt}[j-1] \\ \text{opt}[p(j)] + w_j \end{cases}$$

We can fill in the table using this recurrence as follows:

Algorithm 3 Weighted-Interval-DP

```
Sort jobs such that $f_1 \leq \cdots \leq f_n$
Compute $p(j)$ for each $j$
$O[0] = 0$
**for** $i = 1$ to $n$ **do**
    $O[i] = \max\{O[i-1], w_i + O[p(i)]$
**return** $O[n]$
```

This computes the optimum in $O(n \log n)$. To find the actual schedule, we can trace back the selection of jobs. More specifically, starting at $O[n]$ we check whether $O[n] = O[n-1]$ or not. If not then n belongs to optimum and we jump to $p(n)$ and check $O[p(n)]$. Else (if $O[n] = O[n-1]$), it means n is not part of the optimum and we continue from $O[n-1]$.