Algorithms

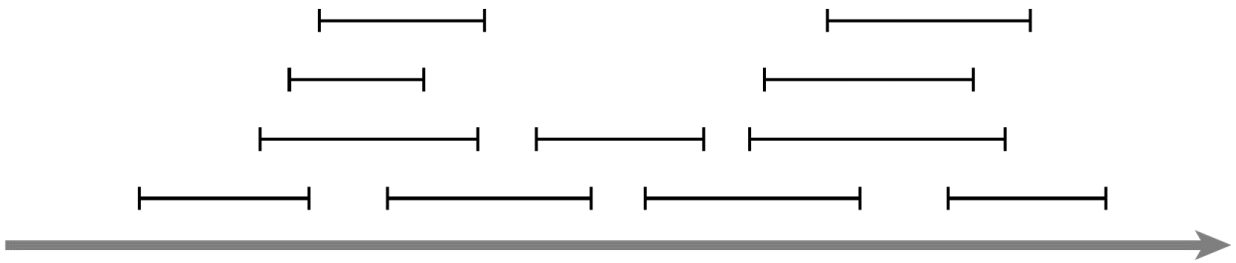
Algorithmic Techniques

We will discuss three basic and broad algorithmic techniques in this course.

- Greedy
- Divide and conquer
- Dynamic Programming

Interval Scheduling

Given a set of jobs $J_1, \cdots, J_n$ where J_i has a starting time $s(i)$ and ending time $t(i)$, how do we pick the largest number of jobs that are pairwise non-overlapping?



Greedy Strategy. Pick jobs one by one *greedily* and whenever a job is picked remove all jobs incompatible with it.

Example greedy stragegies. Always pick a job that

- has the earliest start time
- has the shortest duration
- has the fewest incompatibilities/conflicts
- has the earliest finish time

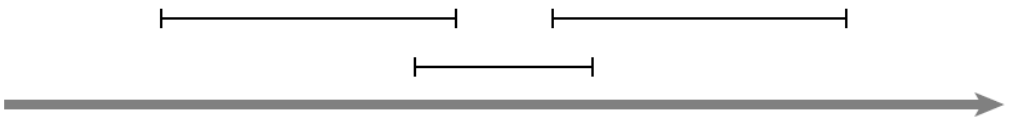
Interval Scheduling

Greedy Strategy. Pick jobs one by one *greedily* and whenever a job is picked remove all jobs incompatible with it.

Always pick a job that

- has the earliest start time *Bad!*

- has the shortest duration *Not Optimal!*

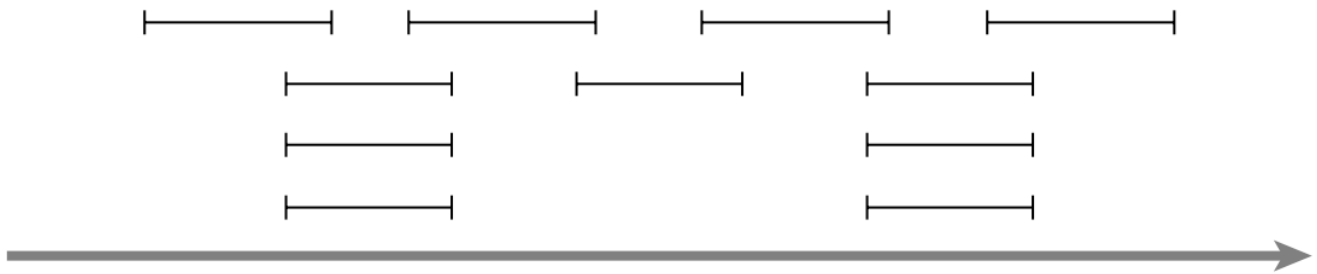


Interval Scheduling

Greedy Strategy. Pick jobs one by one *greedily* and whenever a job is picked remove all jobs incompatible with it.

Always pick a job that

- has the fewest incompatibilities/conflicts *Still not optimal!*

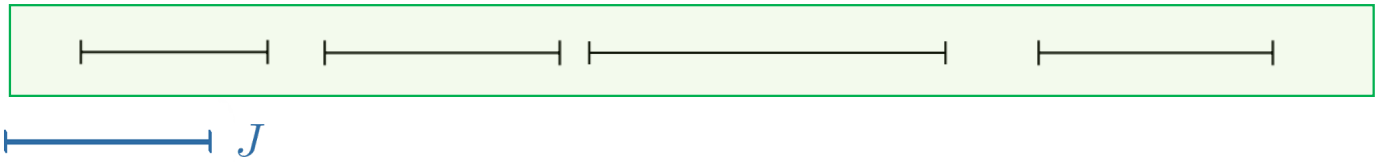


- has the earliest finish time *Optimal!* *Proof?*

Interval Scheduling

Lemma. The greedy algorithm that repeatedly picks a job with the earliest finish time and removes all jobs incompatible with it, yields ***an*** optimal solution.

Proof. Consider any optimal solution.



Observation. Let J be the job with the earliest finish time among all jobs.

J has a finish time that is either the same or earlier than the finish time than the first job in the optimal solution.

In either case, replacing the first job in the optimal solution by J yields ***an*** optimal solution. So, we can safely include J in our solution.

We now recursively apply the argument to the jobs compatible with J. ■

Interval Scheduling

Another job scheduling problem.

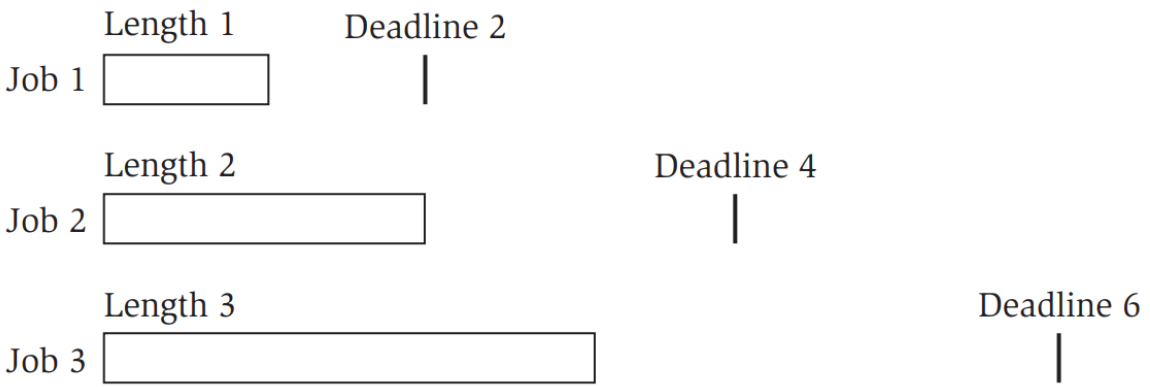
There are n jobs that we need to schedule on a machine.

Job i takes time t_i and has a deadline of d_i.

If f_i is the time we finish job i then we are late by $\ell_i = \max(0, f_i - d_i)$.

Goal. Minimize the maximum lateness $L = \max_i \ell_i$.

Example.



Solution. Execute jobs in the order $1, 2, 3$. $L = 0$.

Interval Scheduling

Which greedy strategy shall we use?

- Shortest duration first *Not optimal!*

  Consider two jobs: job 1 with duration $t_1 = 1$ and deadline $d_1 = 20$ and job 2 with duration $t_2 = 10$ and deadline $d_2 = 10$.

  We should not ignore the deadlines!

- Minimum *slack time* $s_i = d_i - t_i$ first *Not optimal!*

  Consider two jobs: job 1 with duration $t_1 = 1$ and deadline $d_1 = 2$ and job 2 with duration $t_2 = 10$ and deadline $d_2 = 10$.

- Earliest deadline first *Optimal!*

Interval Scheduling

Lemma. The greedy algorithm that executes the jobs in the non-decreasing order of their deadlines yields ***an*** optimal solution.

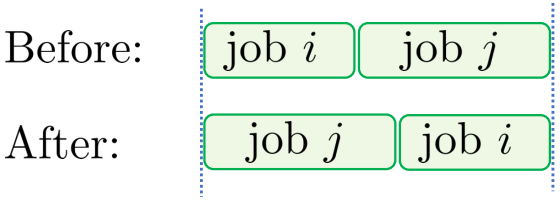
Proof. Consider a schedule where job j is scheduled right after job i and $d_j < d_i$.

Observation. Exchanging the order of the jobs i and j does not increase $\max(\ell_i, \ell_j)$ and therefore doesn't increase the maximum lateness L.

Why?

The lateness of job j can only decrease after the exchange.

The lateness of job i can increase but is still smaller than the earlier lateness of job j.



ℓ_j before exchange: $s + t_i + t_j - d_j$

ℓ_i after exchange: $s + t_i + t_j - d_i$

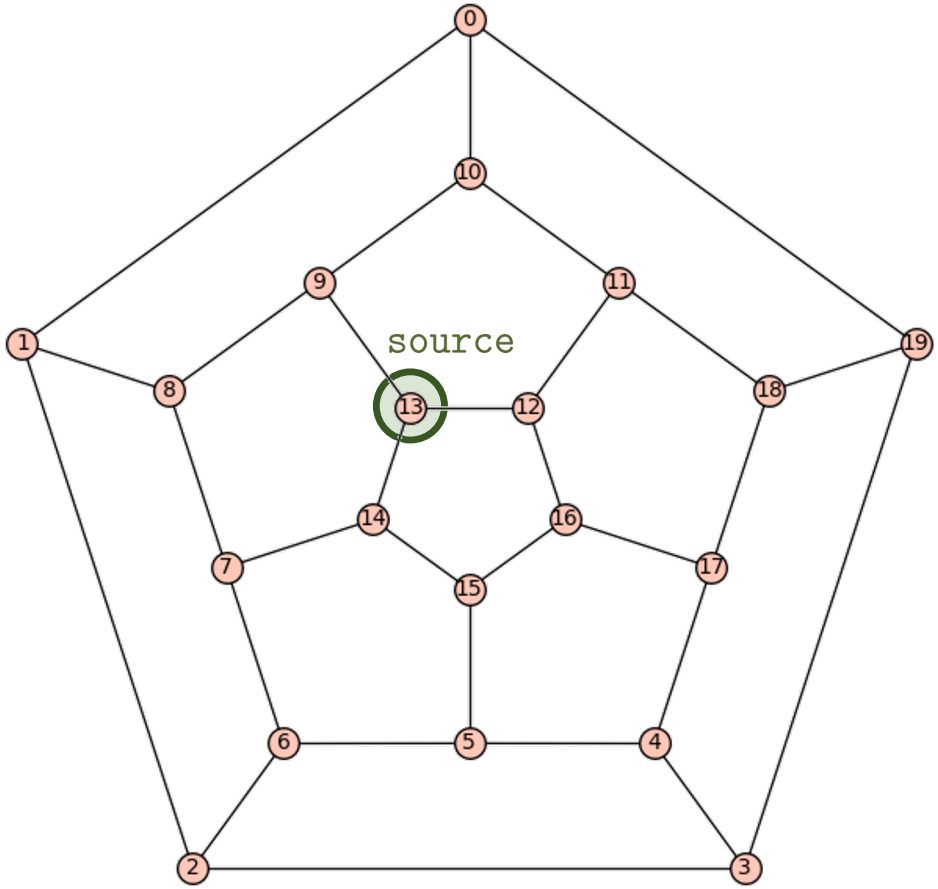
smaller since $d_i > d_j$

■

Single Source Shortest Paths (SSSP)

How do we find the shortest path from a given node in a graph to all other nodes?

BFS!



Single Source Shortest Paths (SSSP)

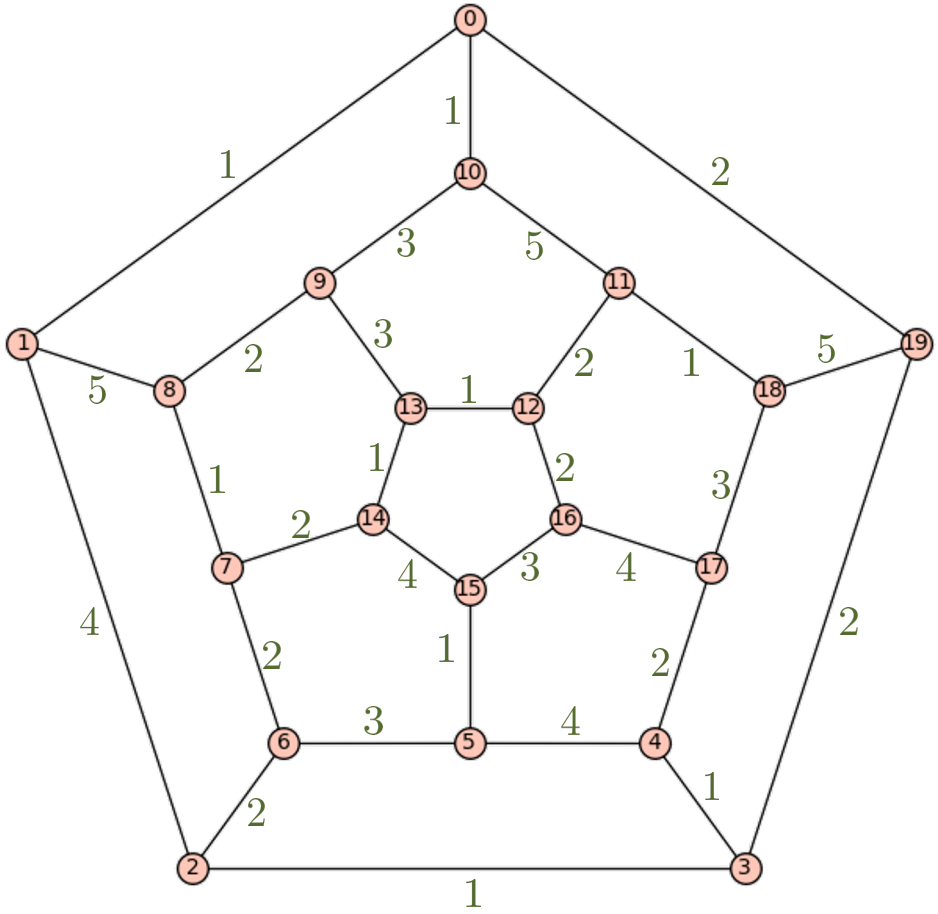
What if we had non-negative weights on the edges indicating its "length"?

Imagine starting a fire at the source vertex.

Think of the weights on the edges as the time it takes the fire to traverse that edge.

Then the time a vertex catches fire is precisely its shortest distance from the source.

At what times do the vertices catch fire if we start it at the node 13?



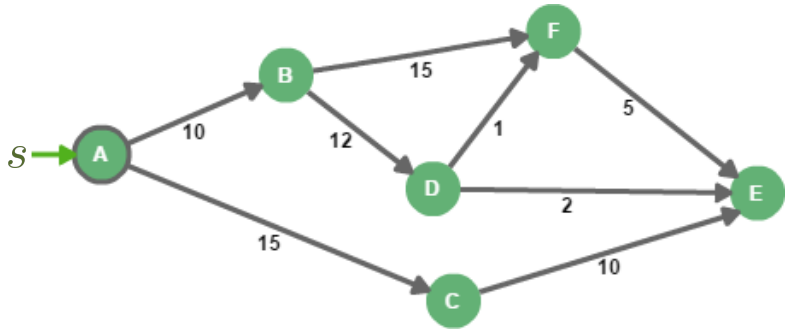
Single Source Shortest Paths (SSSP)

Observation. It suffices to find an algorithm for directed graphs.

An algorithm for directed graph can also be used for undirected graphs.

We are given a directed graph in which each edge e has a positive length $\ell(e)$.

We are also specified a *source* vertex s in the graph.



Goal. Find the shortest directed path from s to all other vertices in the graph.

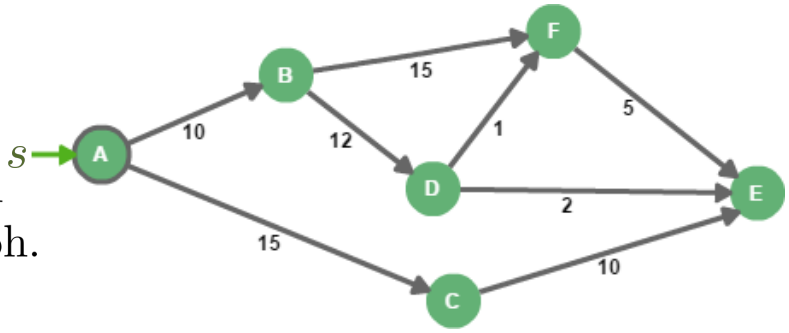
Would minimizing the next step work?

Single Source Shortest Paths (SSSP)

Observation. It suffices to find an algorithm for directed graphs.

An algorithm for directed graph can also be used for undirected graphs.

Goal. Find the shortest directed path from s to all other vertices in the graph.



Would minimizing the next step work?

No, better to think recursively!! (think of the case of path to E, if $\ell(CE)$ is 1 instead of 10)

Better to minimise the next path

Given $d[v]$, the length of the shortest $s \rightsquigarrow v$ path, for each vertex v, we can figure out the shortest path to any vertex from s.

Dijkstra's Algorithm for SSSP

At any point in time we maintain a set of vertices S for which we have already computed the shortest path distance from s.

Initially $S = \{s\}$ and $d[s] = 0$.

We find an edge $u \to v$ from a vertex $u \in S$ to a vertex $v \notin S$ which *minimizes* $d[u] + \ell(u \to v)$.