Week 4 - Wednesday

COMP 4500

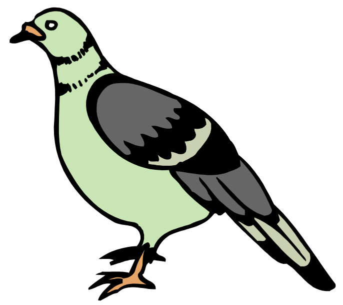
Last time

- What did we talk about last time?
- Topological sort
- Greedy algorithms
- Interval scheduling

Questions?

Logical warmup

- Two friends who live 36 miles apart decide to meet and start riding their bikes towards each other.
  - They plan to meet halfway.
  - Each is riding at 6mph.
  - One of them has a pet carrier pigeon who starts flying the instant the friends start traveling.
  - The pigeon flies back and forth at 18mph between the friends until the friends meet.
- How many miles does the pigeon travel?

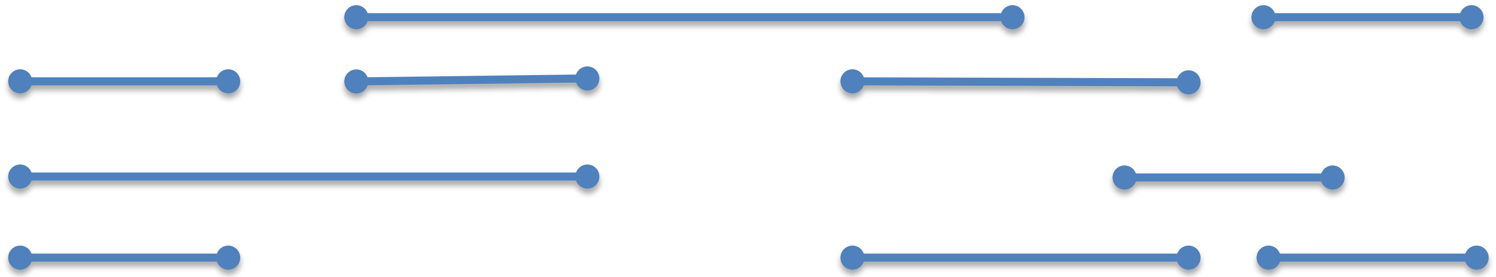


Scheduling all intervals

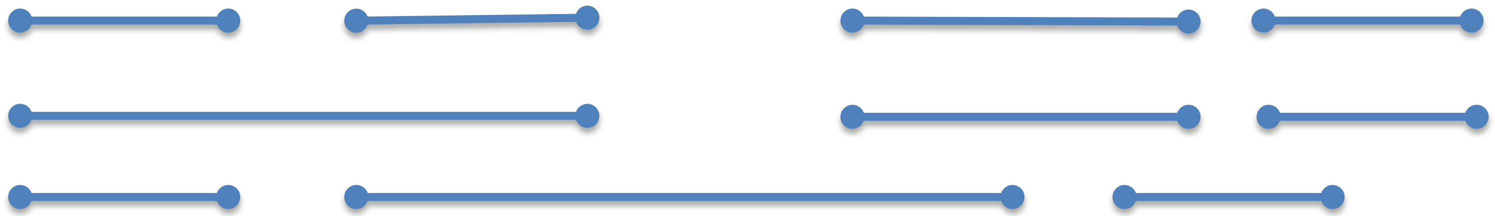
- What if all intervals need to be scheduled?
  - Example: Lectures are intervals and resources are classrooms
  - Example: Roasting pigs are intervals and resources are fire pits
- The problem is to find the minimum number of resources needed to satisfy all intervals
- The book calls this problem **interval partitioning**

Interval partitioning: visualization

- Intervals to schedule (arranged unhelpfully):



- Intervals to schedule (arranged helpfully):



- The depth ***d*** of a set of intervals is the maximum number that pass a single point on the time-line

Interval partitioning algorithm

- Sort intervals by their start times
- Let $\boldsymbol{I}_1$, $\boldsymbol{I}_2$, …, $\boldsymbol{I}_n$ be the ordered intervals
- For $\boldsymbol{j}$ = 1, 2, 3, …, $\boldsymbol{n}$
  - For each interval $\boldsymbol{I}_i$ that precedes $\boldsymbol{I}_j$ in sorted order and overlaps it
    - Exclude the label of $\boldsymbol{I}_i$ from consideration for $\boldsymbol{I}_j$
  - If there is any label from {1, 2, …, $\boldsymbol{d}$} that has not been excluded
    - Assign that label to $\boldsymbol{I}_j$
  - Else
    - Leave $\boldsymbol{I}_j$ unlabeled

Interval partitioning correctness

- **Claim:** In our algorithm, every interval will be assigned a label, and no two overlapping intervals will receive the same label.
- **Proof:** Consider interval $\boldsymbol{I_j}$, and suppose there are $\boldsymbol{t}$ intervals earlier in the sorted order that overlap it. These $\boldsymbol{t}$ intervals with $\boldsymbol{I_j}$ form $\boldsymbol{t} + 1$ intervals that pass over a common point on the time-line. Thus, $\boldsymbol{t} + 1 \leq \boldsymbol{d}$ and $\boldsymbol{t} \leq \boldsymbol{d} - 1$. Thus, there must be at least one label left to be assigned to $\boldsymbol{I_j}$.

Proof continued

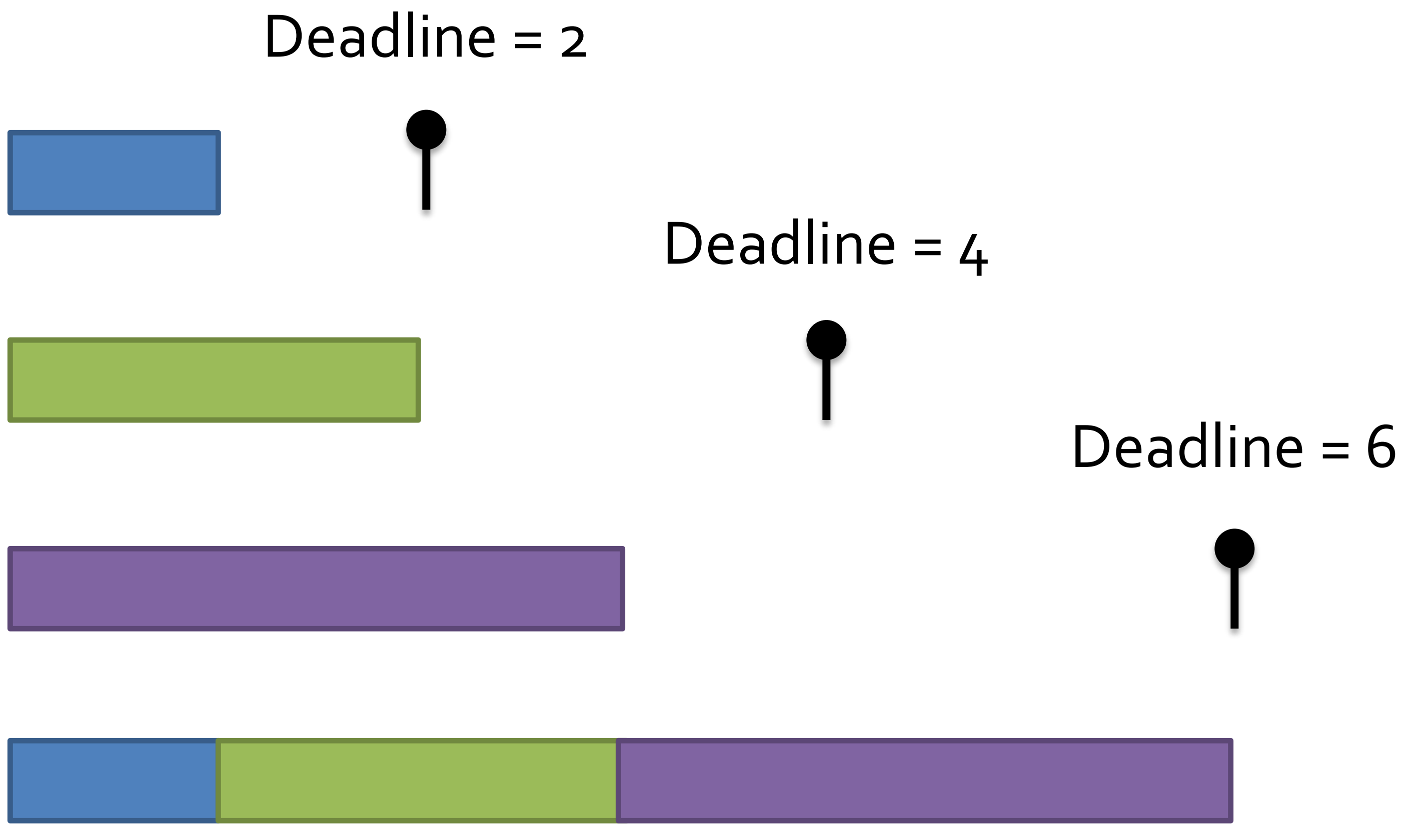
- To show that no two overlapping intervals are assigned the same label, consider two intervals ***I*** and ***I'*** that overlap. Suppose that ***I*** precedes ***I'*** in the sorted order. When ***I'*** is considered by the algorithm, ***I*** is in the set of intervals whose labels are excluded. ■

Scheduling to minimize lateness

- Consider a problem with requests that are not fixed in time
- Instead, each request $\boldsymbol{i}$ has a deadline $\boldsymbol{d_i}$ and requires time $\boldsymbol{t_i}$ using the resources
- If the finish time $\boldsymbol{f}(\boldsymbol{i}) > \boldsymbol{d_i}$, its lateness $\boldsymbol{l_i} = \boldsymbol{f}(\boldsymbol{i}) - \boldsymbol{d_i}$
- If the finish time $\boldsymbol{f}(\boldsymbol{i}) \leq \boldsymbol{d_i}$, its lateness $\boldsymbol{l_i} = 0$
- One goal we could have is to minimize the maximum lateness of any given job
  - We don't care about the sum of the lateness, just the single job that is the **most** overdue

Scheduling example

- Job 1:
- Job 2:
- Job 3:
- Solution:
- Lateness = 0



Designing the algorithm

- Which request do we schedule next?
- Shortest jobs first?
  - No, we could have short jobs with late deadlines
- Jobs with the least slack time ($\boldsymbol{d_i} - \boldsymbol{t_i}$)
  - No, consider Job 1 with $\boldsymbol{t_1} = 1$ and $\boldsymbol{d_1} = 2$ and Job 2 with $\boldsymbol{t_2} = 10$ and $\boldsymbol{d_2} = 10$
- Sort by order of increasing deadlines?
  - Surprisingly, yes! Length of job doesn't matter

Minimizing lateness algorithm

- Sort the jobs in increasing order of deadlines
- For simplicity, relabel jobs and deadlines so that $\boldsymbol{d_1} \leq \boldsymbol{d_2} \leq \ldots \boldsymbol{d_n}$
- Set $\boldsymbol{f} = \boldsymbol{s}$
- For $\boldsymbol{i} = 1, 2, \ldots, \boldsymbol{n}$
  - Assign job $\boldsymbol{i}$ to the interval from $\boldsymbol{s}(\boldsymbol{i}) = \boldsymbol{f}$ to $\boldsymbol{f}(\boldsymbol{i}) = \boldsymbol{f} + \boldsymbol{t_i}$
  - Set $\boldsymbol{f} = \boldsymbol{f} + \boldsymbol{t_i}$

Observations

- **Idle time** is time when no jobs are scheduled but there are still jobs left (gaps)
- There is an optimal schedule with no idle time
- We will use an **exchange argument** to transform the optimal schedule $\boldsymbol{O}$ into the schedule $\boldsymbol{A}$ we produce
- We say that a schedule has an **inversion** if a job $\boldsymbol{i}$ with deadline $\boldsymbol{d_i}$ is scheduled before a job $\boldsymbol{j}$ with an earlier deadline $\boldsymbol{d_j} < \boldsymbol{d_i}$
  - Our algorithm produces a schedule with **no** inversions

All schedules with no inversions and no idle time have the same maximum lateness

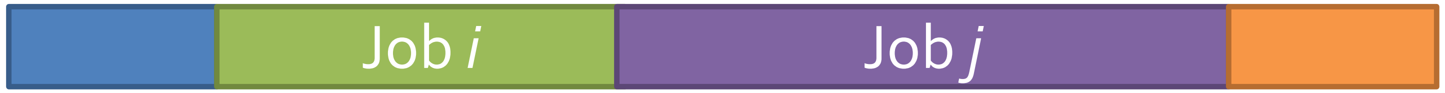
- Proof:
  - If two different schedules have neither inversions nor idle time, they might not produce the same order of jobs, but they can only differ in the order in which jobs with identical deadlines are scheduled.
  - Consider such a deadline $\boldsymbol{d}$. In both schedules, the jobs with deadline $\boldsymbol{d}$ are all scheduled consecutively (after all jobs with earlier deadlines and before all jobs with later deadlines). Among the jobs with deadline $\boldsymbol{d}$, the last one has the greatest lateness, and this lateness does not depend on the order of the jobs. ■

There is an optimal schedule with no inversions and no idle time

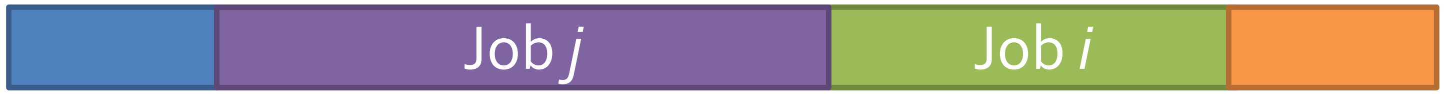
- Proof:
  - We know there is an optimal schedule ***O*** with no idle time.
  - If ***O*** has an inversion, then there is a pair of jobs ***i*** and ***j*** such that ***j*** is scheduled right after ***i*** and has $d_j < d_i$.
  - If there were no inversions, all of the deadlines would be in order, but the fact that there is an inversion means that some point will be reached where a job has an earlier deadline than the job before it. Let that job be ***i*** and the previous job be ***j***.
  - If we swap ***i*** and ***j***, we get a schedule with one fewer inversion.

Illustration of jobs *i* and *j*

Original schedule ***O*** (before swapping)



Schedule ***O'*** (after swapping)



Nothing other than ***i*** and ***j*** are affected

Proof continued

- This new swapped schedule has a maximum lateness no larger than that of ***O***.
- Why?
- Since there are no gaps, the only things affected are jobs ***i*** and ***j***. (Nothing before or after changes.)
- Job ***j*** is finishing earlier, so its lateness will not increase.
- We will use a prime (') to differentiate quantities in the swapped schedule.
- Now, $f'(i) = f(j)$ and remember that $d_i > d_j$
- $l_i' = f(j) - d_i < f(j) - d_j = l_j$
- The old maximum lateness $L \geq l_j > l_i'$
- Thus, the new maximum lateness L' will not be greater

■

Our greedy algorithm is optimal

- Claim:
  - The schedule ***A*** produced by the greedy algorithm has optimal maximum lateness.
- Proof:
  - The previous proof shows that we can construct a schedule without inversions as good as any optimal one with inversions.
  - The proof before that says that all schedules with no inversions and no idle time have the same maximum lateness.
  - Since our algorithm finds a schedule without inversions and no idle time, it must be optimal. ■

Three-Sentence Summary of Shortest Paths and Minimum Spanning Tree

Shortest Paths

Shortest path set up

- Directed graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ with start node $\boldsymbol{s}$
- Assume that there is a path from $\boldsymbol{s}$ to every other node (although that's not critical)
- Every edge $\boldsymbol{e}$ has a length $\boldsymbol{l_e} \geq 0$
- For a path $\boldsymbol{P}$, length of $\boldsymbol{P}$ $\boldsymbol{l}(\boldsymbol{P})$ is the sum of the lengths of the edges on $\boldsymbol{P}$
- We want to find the shortest path from $\boldsymbol{s}$ to every other node in the graph
- An undirected graph is an easy tweak

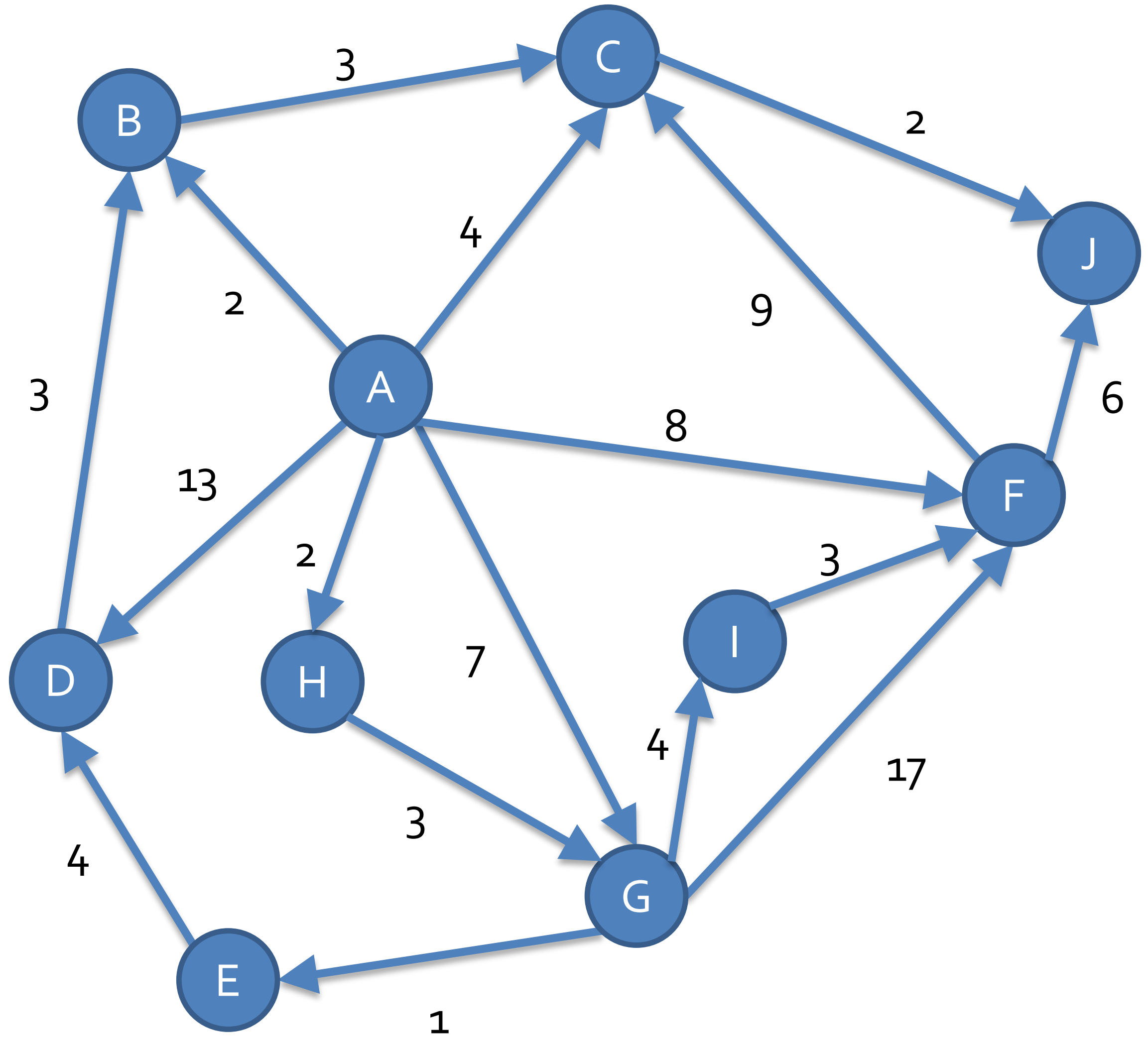
Designing the algorithm

- Let's first look at the **length** of the paths, not the actual paths
- We keep set ***S*** of vertices to which we have determined the true shortest-path distance
  - ***S*** is the explored part of the graph
- Then, we try to find the shortest new path by traveling from any node in the explored part ***S*** to any node ***v*** outside
- We update the distance to ***v*** and add ***v*** to ***S***
- Then, continue

Dijkstra's algorithm

- Let $\boldsymbol{S}$ be the set of explored nodes
  - For each $\boldsymbol{u} \in \boldsymbol{S}$, we store a distance $\boldsymbol{d}(\boldsymbol{u})$
- Initially $\boldsymbol{S} = \{\boldsymbol{s}\}$ and $\boldsymbol{d}(\boldsymbol{s}) = 0$
- While $\boldsymbol{S} \neq \boldsymbol{V}$
  - Select a node $\boldsymbol{v} \notin \boldsymbol{S}$ with at least one edge from $\boldsymbol{S}$ for which $\boldsymbol{d'}(\boldsymbol{v}) = \min_{e=(u,v):u \in S} \boldsymbol{d}(\boldsymbol{u}) + \boldsymbol{l}_e$ is as small as possible
  - Add $\boldsymbol{v}$ to $\boldsymbol{S}$ and define $\boldsymbol{d}(\boldsymbol{v}) = \boldsymbol{d'}(\boldsymbol{v})$

Dijkstra's algorithm example



Consider S at any point during the algorithm

- Claim: For each $\boldsymbol{u} \in \boldsymbol{S}$, the path $\boldsymbol{P_u}$ is a shortest $\boldsymbol{s}$-$\boldsymbol{u}$ path
- Proof by induction on the size of $\boldsymbol{S}$:
  - Basis case: ($\boldsymbol{n}$ = 1)
  - $|\boldsymbol{S}|$ = 1 means $\boldsymbol{S} = \{\boldsymbol{s}\}$ and $\boldsymbol{d}(\boldsymbol{s}) = 0$
  - Clearly, that's the best distance to $\boldsymbol{s}$
  - Induction step: ($\boldsymbol{n} = \boldsymbol{k}$)
  - Assume that when $|\boldsymbol{S}| = \boldsymbol{k}$ for $\boldsymbol{k} \geq 1$, $\boldsymbol{S}$ contains the shortest paths for everything in $\boldsymbol{S}$

Proof continued

- Suppose we are increasing S to size k + 1 by adding node v. Let (u, v) be the final edge on our s-v path P_v.
- By the induction hypothesis, P_u is the shortest s-u path for all $u \in S$.
- Consider any other s-v path P. Since v is not in S, P must leave S somewhere. Let y be the first node on P that is not in S, and let $x \in S$ be the node just before y.

Proof continued

- On step $k + 1$, we could have added y, but we didn't. Thus, there is no path from s to y through x that is shorter than P_v.
- But the subpath of P up to y is such a path, so that subpath must be at least as long as P_v. Since edge lengths are nonnegative, P is at least as long as P_v.
- Thus, P_v must be the shortest path to v. ■

Reflections on Dijkstra's algorithm

- You can think of Breadth-First Search as a pulse expanding, layer by layer, through a graph from some starting node
- Dijkstra's algorithm is the same, except that the time it takes for the pulse to arrive is based not on the number of edges, but the lengths of the edges it has to pass through
- Because Dijkstra's algorithm expands from the starting point to whatever is closer, it grows like a blob
- There are algorithms that, under certain situations, can cleverly grow in the direction of the destination and will often take less time to find the path there

Running time for Dijkstra's algorithm

- For n nodes, the While loop runs $n - 1$ times
- In the worst case, we might need to look at all the edges to compute appropriate minima for each iteration
  - Yielding a running time of $O(mn)$
- It turns out that using a priority queue allows us to get the time down to $O(m \log n)$
  - Some deep data structure work has been done here, and I don't want to go into it
  - Read the literature if you want to implement the algorithm as fast as possible

Quiz

Upcoming

Next time…

- Finish Dijkstra's
- Minimum spanning trees
- Review

Reminders

- **Finish Assignment 2**
  - **Due Friday before midnight**
- Review chapters 1 through 3
- Exam 1 is next Monday
- **All office hours are canceled today**