# EE360C: Algorithms

## Greedy Algorithms
## Part 2/4

Summer 2019

Department of Electrical and Computer Engineering
University of Texas at Austin

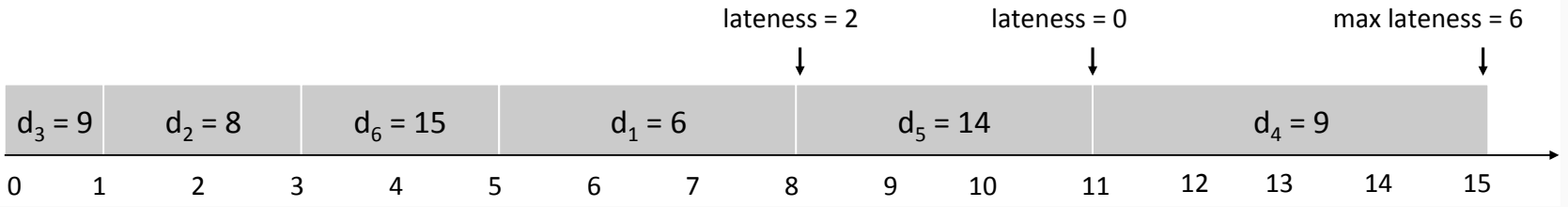
# Scheduling to Minimize Lateness

# Scheduling to Minimize Lateness

## Minimizing Lateness Problem

- Single resource processes one job at a time
- Job $j$ requires $t_j$ units of processing time and is due at time $d_j$
- If $j$ starts at time $s_j$, it finishes at time $f_j = s_j + t_j$
- Lateness = $l_j = \max\{0, f_j - d_j\}$
- Goal: Schedule all jobs to minimize maximum lateness $L = \max l_j$

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| $t_j$ | 3 | 2 | 1 | 4 | 3 | 2 |
| $d_j$ | 6 | 8 | 9 | 9 | 14 | 15 |



# Greedy Template: Greedy Choice

**Greedy Choice**

We have to consider the jobs in some order:

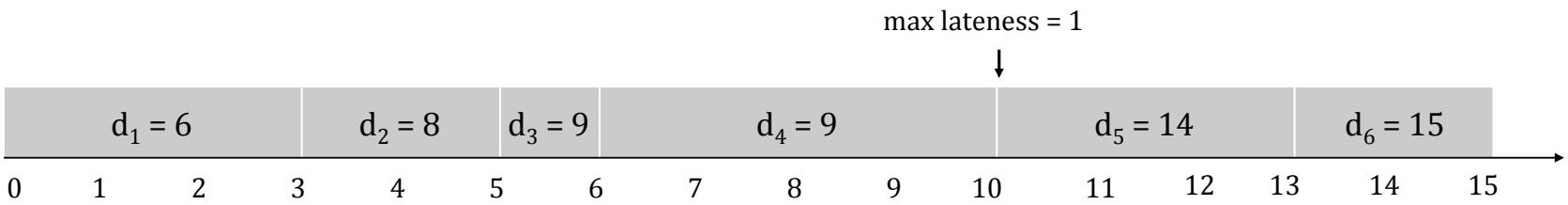
- **Shortest processing time first.** Consider jobs in ascending order of processing time $t_j$.
- **Earliest deadline first.** Consider jobs in ascending order of deadline $d_j$.
- **Smallest slack.** Consider jobs in ascending order of slack $d_j - t_j$.

Can you construct counter examples for any of these options?

# Greedy Algorithm

Sort n jobs by deadline so that $d_1 \le d_2 \le \ldots \le d_n$

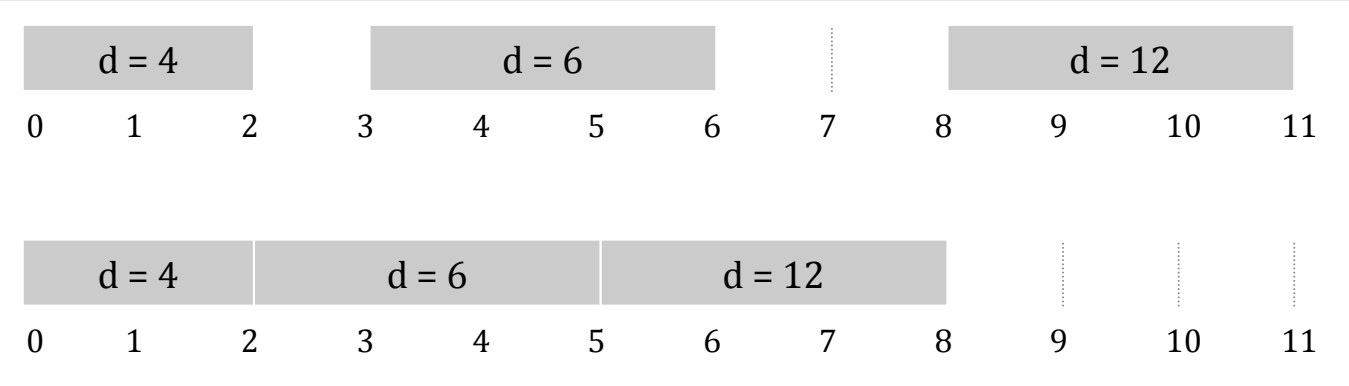
$t \leftarrow 0$
for j = 1 to n
  Assign job j to interval $[t, t + t_j]$
  $s_j \leftarrow t$, $f_j \leftarrow t + t_j$
  $t \leftarrow t + t_j$
output intervals $[s_j, f_j]$



# No Idle Time

## Observation 1

There exists an optimal schedule with no idle time.




## Observation 2

The greedy schedule has no idle time.

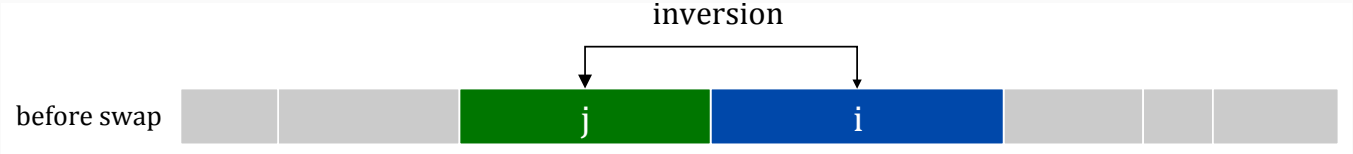
## Process

Consider an optimal schedule *O*. Gradually modify *O*, preserving its optimality at each step, but eventually transforming it into a schedule *A* that our greedy algorithm would give us. This is an exchange argument.

# Inversions

## Definition

An inversion in schedule $S$ is a pair of jobs $i$ and $j$ such that $d_i < d_j$ but $j$ scheduled before $i$.




## Observation

A greedy schedule has no inversions.

# Inversions (cont.)

**Claim**

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

**Proof**

- If two different schedules have no inversions or idle time, then they differ in the order in which jobs that have identical deadlines can be scheduled.
- Let such a deadline be $d$.
- Jobs with deadline $d$ are scheduled consecutively after jobs with earlier deadlines and after jobs with later deadlines.
- Among jobs with deadline $d$ only the last one has the greatest lateness, and this lateness does not depend on the order of the jobs.

# Exchange Argument

## Claim

There is an optimal solution that has no inversions and no idle time.

## Proof

- Already claimed there is an optimal solution with no idle time.

**(a)** If $\mathcal{O}$ has an inversion, then there is a pair of jobs $i$ and $j$ such that $j$ is scheduled immediately after $i$ and has $d_j < d_i$.

- Consider an inversion in which job $a$ is scheduled sometime before a job $b$, and $d_a > d_b$.
- If we advance in the scheduled order of jobs from $a$ to $b$ one at a time, there has to come a point where the deadline we see decreases for the first time.
- This corresponds to a pair of consecutive jobs that form an inversion.

# Exchange Argument (contd.)

## Proof

**(b)** After swapping $i$ and $j$ we end up with a schedule with one less inversion.

- When we swap $i$ and $j$ the inversion is fixed and no new inversions are created.

**(c)** The new swapped schedule has a maximum lateness no larger than that of $\mathcal{O}$.

- Proof on white board.

# Analysis of Greedy Algorithm

**Claim**

The schedule $A$ produced by the greedy algorithm has optimal maximum lateness $L$.

**Proof**

- By last claim, there is an optimal schedule with no inversions and idle time.
- All schedules with no inversions and no idle time have the same lateness.
- Therefore the greedy algorithm schedule $A$ is optimal.

# Recap: Greedy Analysis Strategies

**Greedy algorithm stays ahead**

Show that after each step of the greedy algorithm, its solution is at least as good as any other algorithm's.

**Exchange argument**

Gradually transform any solution to the one found by the greedy algorithm without hurting its quality.

**Structural**

Discover a simple "structural" bound asserting that every possible optimal solution must have a certain value. Then show that your algorithm always achieves this bound.

# Shortest Paths

# Shortest Paths

Given a graph and its vertices and edges, how do we find the shortest path from one vertex to another?

- Clearly, enumerating all of the possible paths, summing the weights of the edges, and taking the minimum is excessively expensive.

This is another pervasive problem:

- networking algorithms
- maps and distances traveled
- scheduling algorithms
- or anything else where the weights can be costs, time, distances, etc.

# Shortest Path Problem

## The shortest-paths problem

Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbf{R}$, define the weight of a path $p = \langle v_0, v_1, \ldots v_k \rangle$ to be the sum of the weights of its constituent edges:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

We define the **shortest path weight** from $u$ to $v$ as:

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{if there is a path from u to v} \\ \infty & \text{otherwise} \end{cases}$$

A **shortest path** from $u$ to $v$ is any path $p$ with weight $w(p) = \delta(u, v)$.

# Variants of the Shortest Path Problem

We'll focus first on the **single-source shortest-paths problem** (i.e., find the shortest paths given a specific source vertex), but others exist:

- **single-destination shortest-paths problem**: find the shortest paths to a given destination from all other vertices
- **single-pair shortest-path problem**: given $u$ and $v$, find the shortest path from $u$ to $v$
- **all-pairs shortest-paths problem**: find a shortest path for every pair of vertices

# Shortest Paths Substructure

The algorithms we'll discuss often rely on the fact that a shortest path contains within it other shortest paths.

**Lemma 24.1**

Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbf{R}$, let $p = \langle v_0, v_1, \ldots v_k \rangle$ be a shortest path from $v_0$ to $v_k$ and, for any $i$ and $j$ such that $1 \leq i \leq j \leq k$, let $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ be the subpath of $p$ from $v_i$ to $v_j$. Then $p_{ij}$ is a shortest path from $v_i$ to $v_j$.

**Prove this by contradiction.**

# Gotcha: Negative-Weight Edges

At times, it makes sense for edges to have negative weights.

- if there are no cycles in the graph that contain negative weights, then the shortest path weights are well-defined
- if there are cycles that contain negative edges, the shortest path is not well-defined (since if I found one, I could always find one smaller by going around again)
- if there is such a negative edge cycle on a path from $u$ to $v$, we define $\delta(u, v) = -\infty$

Some algorithms assume that all edge weights are non-negative (so they're basically preceded by a phase that checks this); others can handle negative weights.

# Representing Shortest Paths

Usually when we construct the shortest paths, we want more than their weights, we want to know the actual paths, too. We maintain the **predecessor subgraph** $G_\pi = (V_\pi, E_\pi)$. $V_\pi$ is the set of vertices in $G$ that end up having non-NIL predecessors (and the source, $s$):

$$V_\pi = \{v \in V : v.\pi \neq nil \cup s\}$$

$E_\pi$ is the set of edges induced by the $\pi$ values for the elements of $V_\pi$:

$$E_\pi = \{(v.\pi, v) \in E : v \in V_\pi - \{s\}\}$$

Practically, we maintain $v.\pi$ for each $v$ and create a chain of predecessors that runs back along the shortest path to $s$. Conceptually, $\pi$ gives us a tree rooted at $s$ that is a subgraph of $G$, contains all vertices reachable from $s$ with their shortest paths.

# Initialization

For each vertex, we maintain $v.d$ to be an upper estimate on the weight of the shortest path from $s$ to $v$.

INITIALIZE-SINGLE-SOURCE$(G, s)$

```
1  for each vertex v ∈ G.V
2        do v.d ← ∞
3           v.π ←NIL
4  s.d ← 0
```

# Relaxation

**Relaxation** of an edge $(u, v)$ tests whether we can improve the shortest path known to $v$ by going through $u$, and, if so, updating $v.d$ and $v.\pi$.

RELAX$(u, v, w)$

1 **if** $v.d > u.d + w(u, v)$
2 **then** $v.d \leftarrow u.d + w(u, v)$
3 $v.\pi \leftarrow u$

# Properties of Shortest Paths

- **Triangle Inequality:** For any edge $(u, v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$
- **Upper-bound property:** We always have $v.d \geq \delta(s, v)$ for all $v \in V$, and once $v.d$ reaches $\delta(s, v)$, it never changes
- **No-path property:** If there is no path from $s$ to $v$, then we always have $v.d = \delta(s, v) = \infty$
- **Convergence property:** If $s \rightsquigarrow u \rightarrow v$ is a shortest path, and if $u.d = \delta(s, u)$ prior to relaxing $(u, v)$, then $v.d = \delta(s, v)$ after
- **Path-relaxation property:** If $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a shortest path from $v_0 = s$ to $v_k$ and the edges of $p$ are relaxed in order, then $v_k.d = \delta(s, v_k)$, regardless of what other relaxations are performed, even if they are interleaved.
- **Predecessor-subgraph property:** Once $v.d = \delta(s, v)$ for all $v \in V$, the predecessor subgraph is a shortest paths tree rooted at $s$.

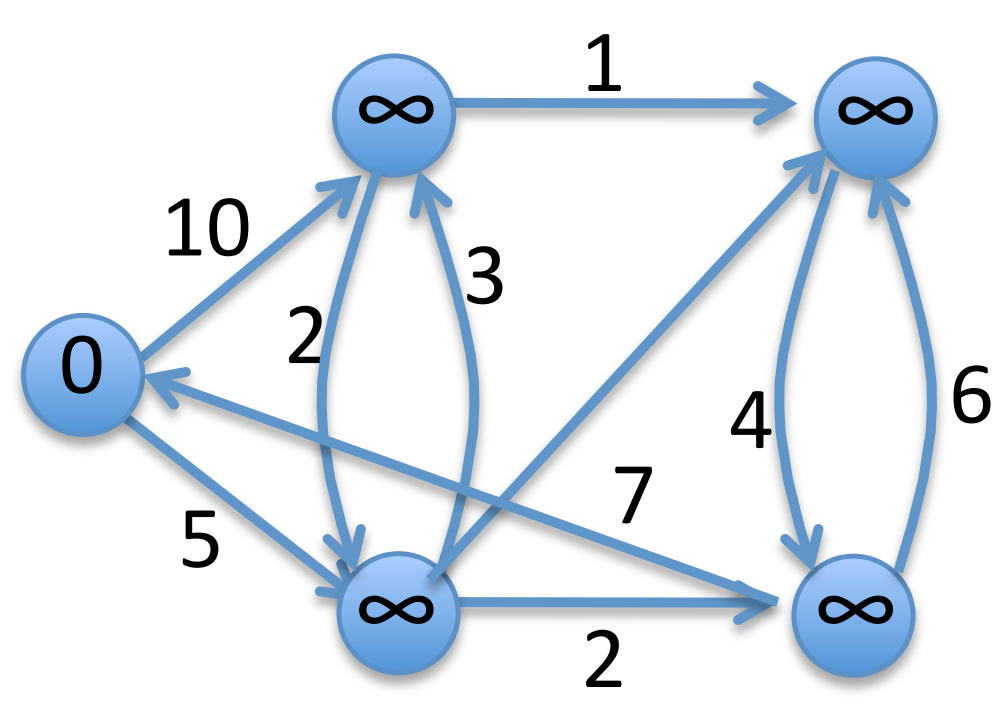
# Dijkstra's Algorithm

## The Intuition

Dijkstra's algorithm maintains a set $S$ of vertices to which the shortest path has been determined.

- the algorithm selects a new vertex $u$ with the minimum shortest path estimate of those left in $V - S$ to add to $S$
- then relax all of the edges leaving $u$ and repeat
- we use a minimum priority queue $Q$ of the vertices that are keyed by the $d$ estimates

# Dijkstra's Algorithm Example



# Dijkstra's Algorithm (cont.)

DIJKSTRA$(G, w, s)$

1 INITIALIZE-SINGLE-SOURCE$(G, s)$
2 $S \leftarrow \emptyset$
3 $Q \leftarrow G.V$
4 **while** $Q \neq \emptyset$
5 **do** $u \leftarrow$ EXTRACT-MIN$(Q)$
6 $S \leftarrow S \cup \{u\}$
7 **for** each vertex $v \in$ Adj$[u]$
8 **do** RELAX$(u, v, w)$

# Dijkstra's Correctness

## Claim

Dijkstra's algorithm terminates with $u.d = \delta(s, u)$ for all $u \in V$.

## Proof by Induction

Base case: $|S| = 1$ is trivial.
Inductive hypothesis: Assume true for $|S| = k \geq 1$

- Let $v$ be the next node added to $S$ and let $u$-$v$ be the chosen edge.
- The shortest $s$-$u$ path plus $(u, v)$ is an $s$-$v$ path of length $d(v)$.
- Consider any $s$-$v$ path $P$. We'll see that it is no shorter than $d(v)$.
- Let $x$-$y$ be the first edge in $P$ that leaves $S$, and let $P'$ be the subpath $s$-$x$.
- $P$ is already too long as soon as it leaves $S$:

$$w(P) \geq w(P') + w(x, y) \geq d(x) + w(x, y) \geq d(y) \geq d(v)$$

- The last step of the above is because the loop chose $v$ instead of $y$ as the next vertex to include.

# Dijkstra's Algorithm Running Time

Assume we store the $v.d$ values in an array of size $V$, indexed by $v$.

- initialization takes $O(V)$ time
- inserting into $Q$ and decreasing the key of an element of $Q$ both take $O(1)$ time (because we stored the keys in an array indexed by $v$)
- EXTRACT-MIN takes $O(V)$ time
- the **while** loop runs $O(V)$ times, each doing an EXTRACT-MIN, so it's $O(V^2)$ overall
- by aggregate analysis, the **for** loop runs $O(E)$ times, and each RELAX call takes $O(1)$ time
- so the overall running time is $O(V^2 + E) = O(V^2)$

# Dijkstra's Running Time (Min-Heap)

If the graph is sparse, we can improve the running time by storing the priority queue in a min-heap

- initialization takes $O(V)$
- the **while** loop runs $O(V)$ times, and EXTRACT-MIN takes $O(\lg V)$ time each time it runs giving $O(V \lg V)$ for the loop
- by aggregate analysis, the internal **for** loop runs twice for each edge (once for every entry in an adjacency list), and RELAX contains an implicit DECREASE-KEY, so the time for the inner loop is $O(E \lg V)$
- so the total running time is $O((V + E) \lg V)$ (or likely $O(E \lg V)$)

# Questions