Review: Hallmark of a Greedy Algorithm

- Sort data according to some criteria
- Consider each piece of data in sorted order and make a local decision
- Result is globally optimal (if this problem is amenable to a greedy solution)
- Complexity is generally no better than $O(n \log n)$ due to the sort
- Important to prove that the solution is optimal

Scheduling to Minimize Lateness

- Single computer processes one job at a time.
- Job $j$ requires $t_j$ units of processing time
- Job $j$ has a deadline $d_j$ by which it must be done
- 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 while minimizing maximum lateness $L = \max_{1 \leq j \leq n} l_j$
Scheduling Example: Earliest Deadline First

Minimizing Lateness: No Idle Time

Observation. There exists an optimal schedule with no idle time.

Minimizing Lateness: No Idle Time

Observation. The greedy schedule has no idle time.
Minimizing Lateness: Inversions

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

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1 )</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>( t_2 )</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

Proof Strategies for Greedy Algorithms

- **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.

Shortest Path Problem

- **Directed graph** \( G = (V, E) \).
- **Source** s, destination t.
- **Length** \( l_e \) = length of edge e.
- \( l_e > 0 \) for all edges e.

**Shortest path problem:** (the Google Maps problem!) find shortest path from s to t.
Dijkstra's Algorithm: Implementation

Dijkstra's Algorithm (G, s) {
    S = {s}    // S is the set of explored nodes
    d(s) = 0    // d is the distance to the node from s

    while S ! = V {   // there are unexplored nodes
        select a node v from V-S with an edge from S for
        which the distance from s to v is the minimum of all
        paths to any node in V-S
        add v to S
        d(v) = minimum distance from s to v
    }
}

How do we implement this efficiently?

Dijkstra's Algorithm (G, s) {
    S = {s}    // S is the set of explored nodes
    d(s) = 0    // d is the distance to the node from s
    lastNode = s

    while S != V {   // there are unexplored nodes
        for each edge (lastNode, v) where v is in V-S {
            dist_to_v = d(lastNode) + (lastNode, v)
            if d'(v) is unknown {
                d'(v) = dist_to_v
                heap.addElement (v, d'(v))
            } else if dist_to_v < d'(v) {
                d'(v) = dist_to_v
                heap.changeKey(v, d'(v))
            }
        }
        lastNode = heap.extractMin()
        add lastNode to S
        d(lastNode) = d'(lastNode)
    }
}

Cost?