EE360C: Algorithms

Greedy Algorithms

Spring 2017

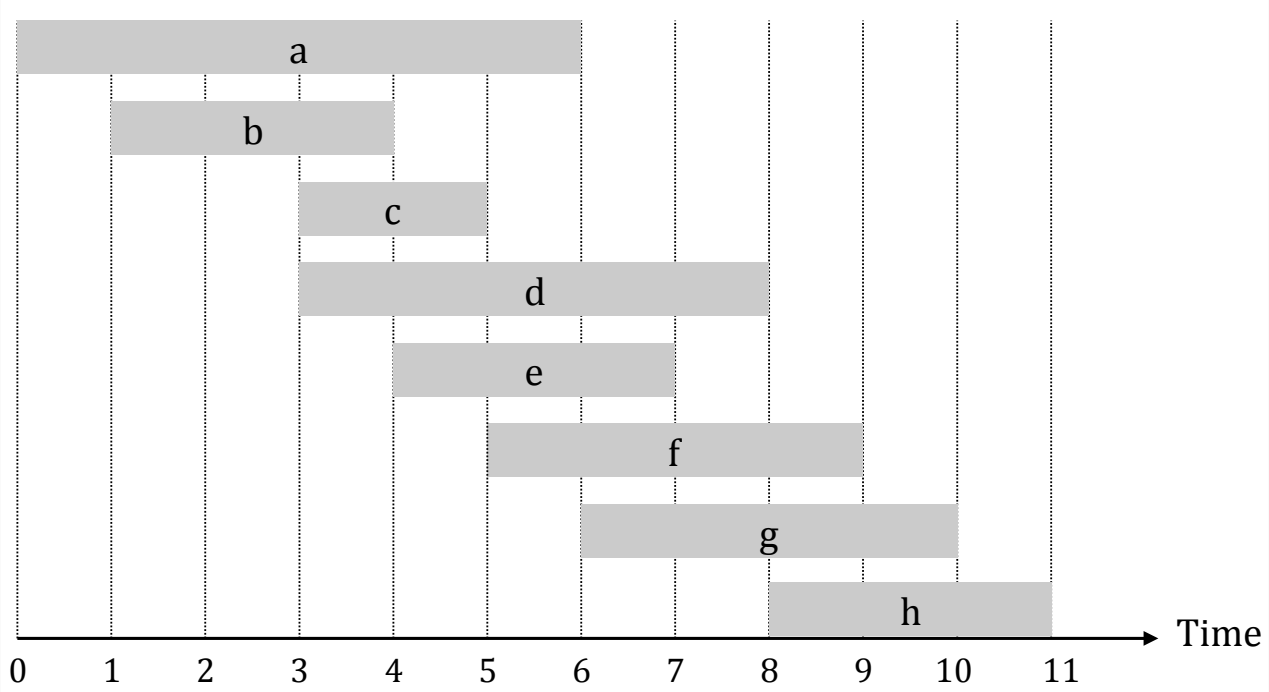
Department of Electrical and Computer Engineering
University of Texas at Austin

Interval Scheduling

Interval Scheduling: The Problem

Interval Scheduling

- Job j starts at s_j and finishes at f_j
- Two jobs are compatible if they do not overlap
- The goal is to find the maximum subset of mutually compatible jobs



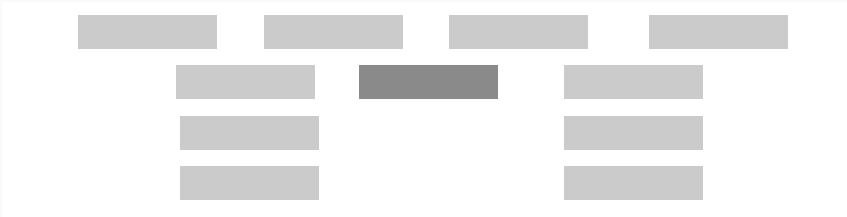
Interval Scheduling: a Greedy Choice

Consider the jobs in some order. Take each job if it is compatible with the ones already taken.

- **Earliest start time** Consider jobs in order of start time s_j

- **Shortest interval** Consider jobs in order of interval length $f_j - s_j$

- **Fewest conflicts** Consider jobs in order of number of conflicts



- **Earliest finish time** Consider jobs in order of finish time f_j

Interval Scheduling: Greedy Algorithm

Greedy Algorithm

Consider jobs in increasing order of finish time. Take each job provided that it is compatible with the ones already taken.

```
Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.

jobs selected
A ← φ
for j = 1 to n {
   if (job j compatible with A)
      A ← A ∪ {j}
}
return A
```

Implementation

We can implement this is $O(n \log n)$ running time.

- Remember job $j*$ that was most recently added to A
- Job j is compatible with A if $s_j \geq f_{j*}$

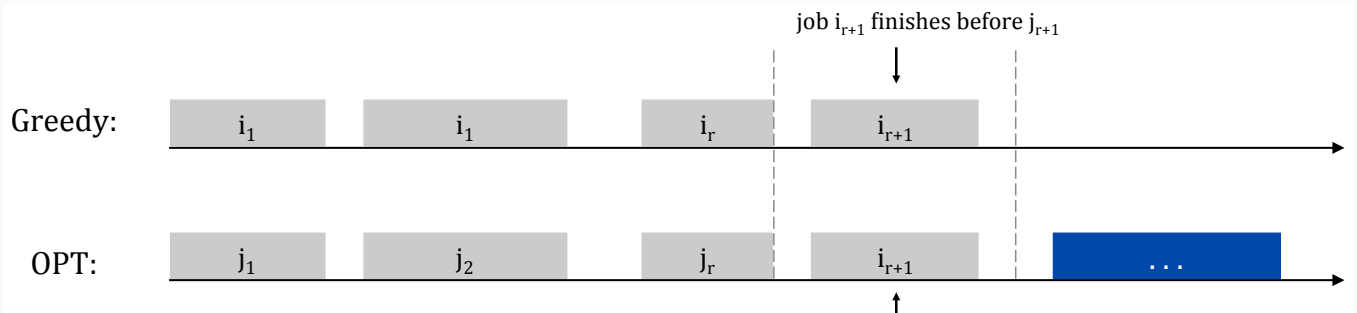
Interval Scheduling: Analysis

Theorem

The greedy algorithm is optimal.

Proof (by contradiction)

- Assume the greedy algorithm is not optimal.
- Let $i_1, i_2, \ldots i_k$ denote the set of jobs selected by the greedy algorithm.
- Let $j_1, j_2, \ldots j_m$ denote the set of jobs in the optimal solution, where $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.
- So job i_{r+1} in the greedy solution finishes before job j_{r+1} in the optimal solution.
- We can replace job j_{r+1} with job i_{r+1} without affecting the optimality of the remaining solution.



Interval Scheduling: Analysis II

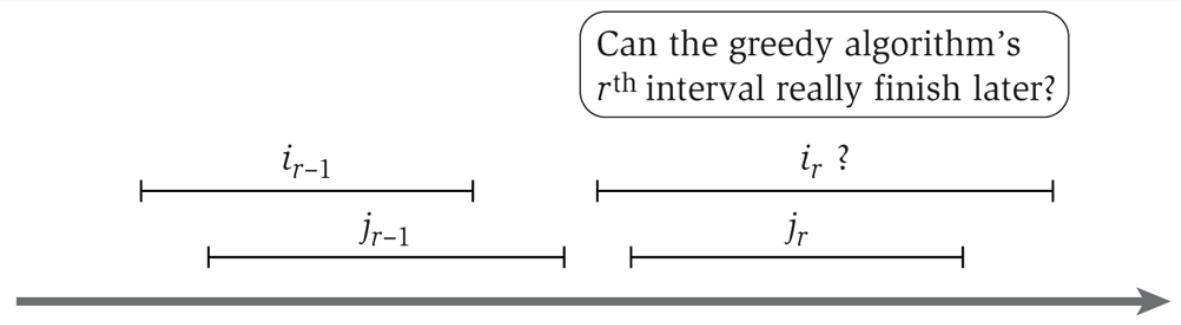
Greedy Algorithm Stays Ahead of Optimal Solution

When i counts indices in the greedy solution and j counts indices in the previous solution, for all indicies $r \leq k$, $f_{i_r} \leq f_{j_r}$.

Proof (by induction)

Base case: $r = 1$; the claim is clearly true, since the greedy algorithm selects the interval with the earliest finish time.

Inductive step: Let $r > 1$, and assume the claim is true for $r - 1$. But by the definition of the greedy algorithm, at each step, the algorithm will choose the next non-conflicting job with the earliest finish time. Because $f_{i_{r-1}} \leq f_{j_{r-1}}$, this selected job's finish time cannot possibly be later than the finish time for the next selected job in the optimal solution.



Interval Scheduling: Complications

This was the simplest framing of the problem. Additional potential considerations:

- the algorithm doesn't know all of the requests *a priori* — online algorithms
- different requests may have different weights — weighted interval scheduling

Interval Partitioning

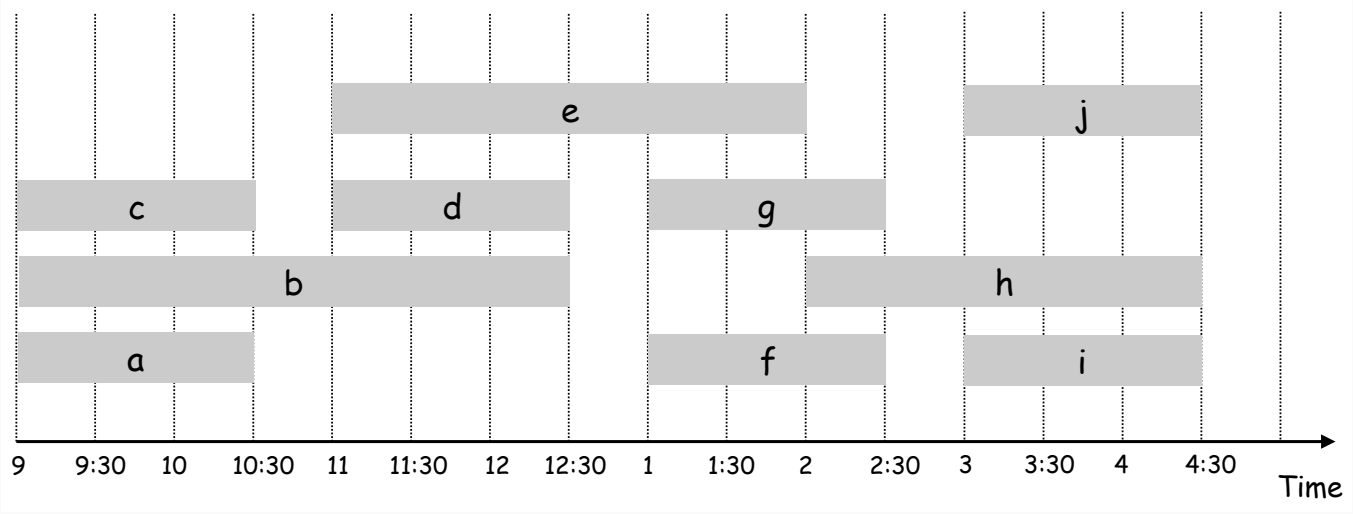
Interval Partitioning

Consider the problem in which one must schedule *all* requests using the fewest "resources" (e.g., processors, classrooms)

Interval Partitioning

- Lecture j starts at s_j and finishes at f_j.
- Goal: find the minimum number of classroosms to schedule all lectures so that no two occur at the same time in the same room.

This schedule uses 4 classrooms to schedule 10 lectures:



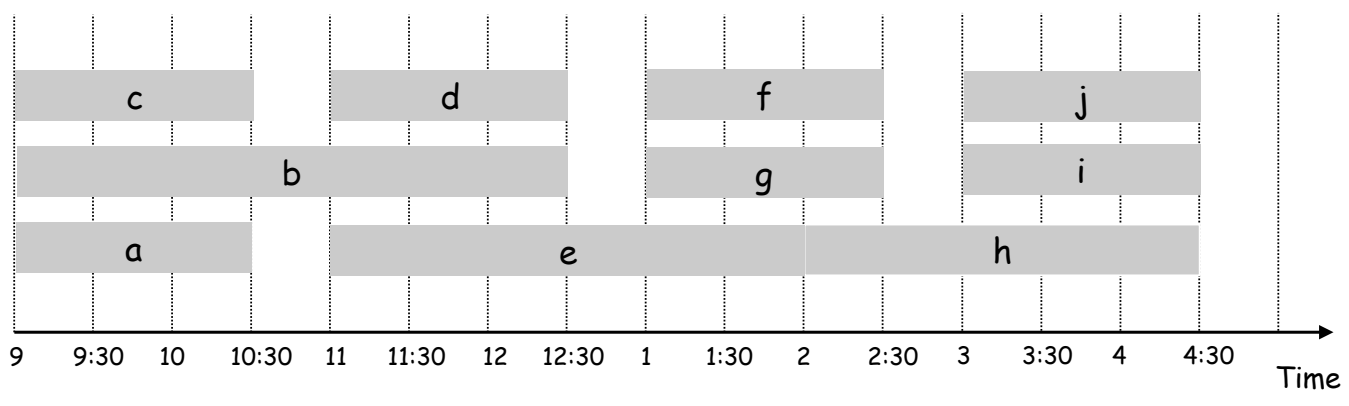
Interval Partitioning

Consider the problem in which one must schedule *all* requests using the fewest "resources" (e.g., processors, classrooms)

Interval Partitioning

- Lecture j starts at s_j and finishes at f_j.
- Goal: find the minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.

This schedule uses 3 classrooms to schedule 10 lectures:



Interval Partitioning: Lower Bound

Definition

The depth of a set of open intervals is the maximum number that contain any given time.

Key Observation

The number of classrooms needed $\geq$ depth

Example

The depth of the schedule on the previous slide was three; the schedule is optimal.

Question

Does there always exist a schedule equal to the depth of intervals?

Interval Partitioning: Greedy Algorithm

Greedy Algorithm

Consider lectures in increasing order of start time; assign lecture to any compatible classroom.

```
Sort intervals by starting time so that s_1 ≤ s_2 ≤ ... ≤ s_n.
d ← 0   ← number of allocated classrooms

for j = 1 to n {
   if (lecture j is compatible with some classroom k)
      schedule lecture j in classroom k
   else
      allocate a new classroom d + 1
      schedule lecture j in classroom d + 1
      d ← d + 1
}
```

Implementation

- For each classroom k, maintain the finish time of the last job added.
- Keep the classrooms in a priority queue.
- Running time: $O(n \log n)$

Interval Partitioning: Greedy Analysis

Observation

The greedy algorithm never schedules two incompatible lectures in the same classroom.

Theorem

The greedy algorithm is optimal.

Proof

- Let d be the number of classrooms the greedy algorithm uses.
- Classroom d is opened because we need to schedule a job, say, j, that is incompatible with all $d - 1$ other classrooms.
- Since we sorted by start time, all these incompatibilities are caused by lectures that start no later than s_j.
- Thus, at time $s_j + \epsilon$, we have d overlapping lectures.
- Key observation: all schedules use $\geq d$ classrooms.

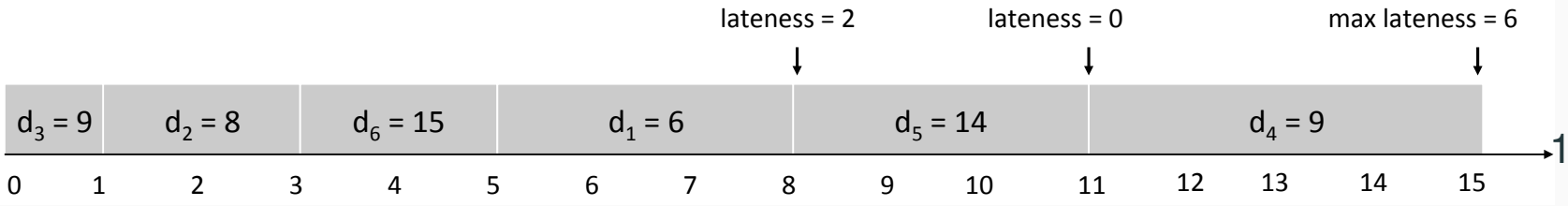
Scheduling to Minimize Lateness

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:

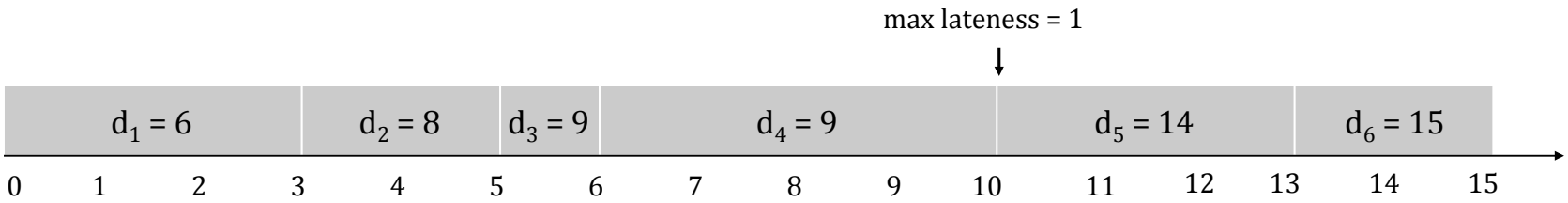
- **Shortest processing time first.** Consider jobs in ascending order of processing time t_j.
- **Earliest deadline first.** Consider jobs in ascending order of deadline d_j.
- **Smallest slack.** Consider jobs in ascending order of slack $d_j - t_j$.

Can you construct counter examples for any of these options?

Greedy Algorithm

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

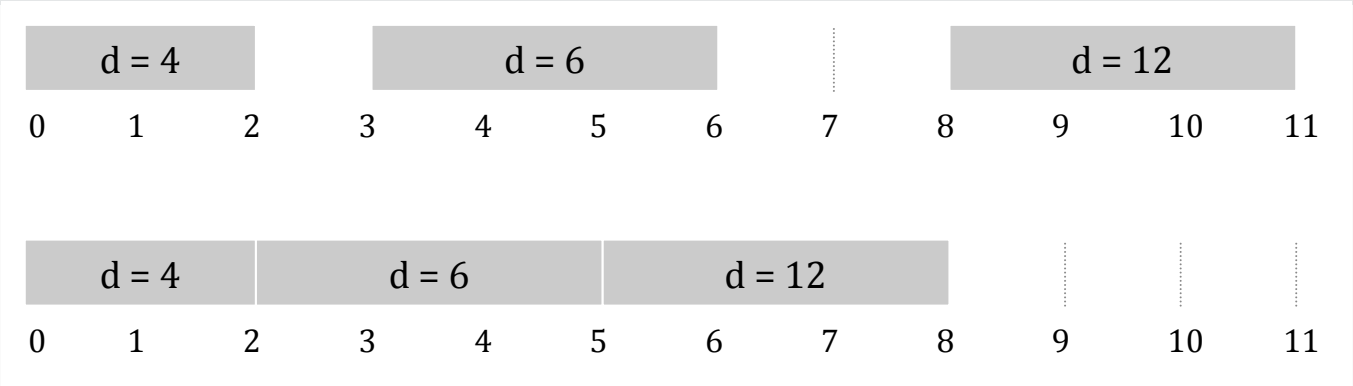
$t \leftarrow 0$
for j = 1 to n
  Assign job j to interval $[t, t + t_j]$
  $s_j \leftarrow t$, $f_j \leftarrow t + t_j$
  $t \leftarrow t + t_j$
output intervals $[s_j, f_j]$



No Idle Time

Observation 1

There exists an optimal schedule with no idle time.



Observation 2

The greedy schedule has no idle time.

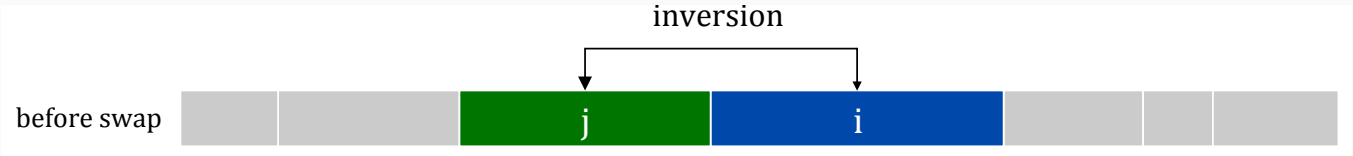
Process

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.

Observation

If a schedule (with no idle time) has an inversion, it has one with a pair of inverted jobs scheduled consecutively.

Inversions (cont.)

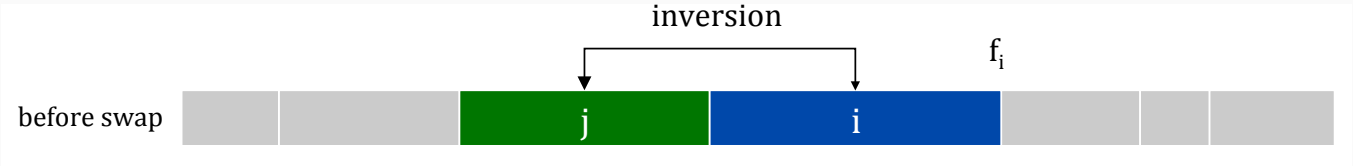
Claim

Swapping two adjacent, inverted jobs reduces the number of inversions by one and does not increase the maximum lateness.

Proof

Let l be the lateness before the swap, and let l' be it after.

- $l'_k = l_k$ for all $k \neq i, j$ and $l'_i \leq l_i$
- If job j is late:
  - $l'_j = f'_j - d_j$ (definition)
  - $l'_j = f_i - d_j$ (j finishes at time f_i)
  - $l'_j \leq f_i - d_i$ ($i < j$)
  - $l'_j \leq l_i$



Analysis of Greedy Algorithm

Theorem

Greedy schedule S is optimal.

Proof

Define S^* to be an optimal schedule that has the fewest number of inversions, and let's see what happens.

- We can assume that S^* has no idle time.
- If S^* has no inversions, then $S = S^*$. (There is only one schedule with no idle time and no inversions.)
- If S^* has an inversion, then let *i*-*j* be an adjacent inversion.
  - Swapping *i* and *j* does not increase the maximum lateness and strictly decreases the number of inversions.
  - Therefore the swapped schedule is at least as good as S^* (it's also optimal) and has fewer inversions.

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_π 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_π is the set of edges induced by the π values for the elements of V_π:

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

Practically, we maintain $v.\pi$ for each v and create a chain of predecessors that runs back along the shortest path to s. Conceptually, π 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 $v \in G.V$
2 **do** $v.d \leftarrow \infty$
3 $v.\pi \leftarrow$NIL
4 $s.d \leftarrow 0$

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 $d[v] \geq \delta(s, v)$ 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.

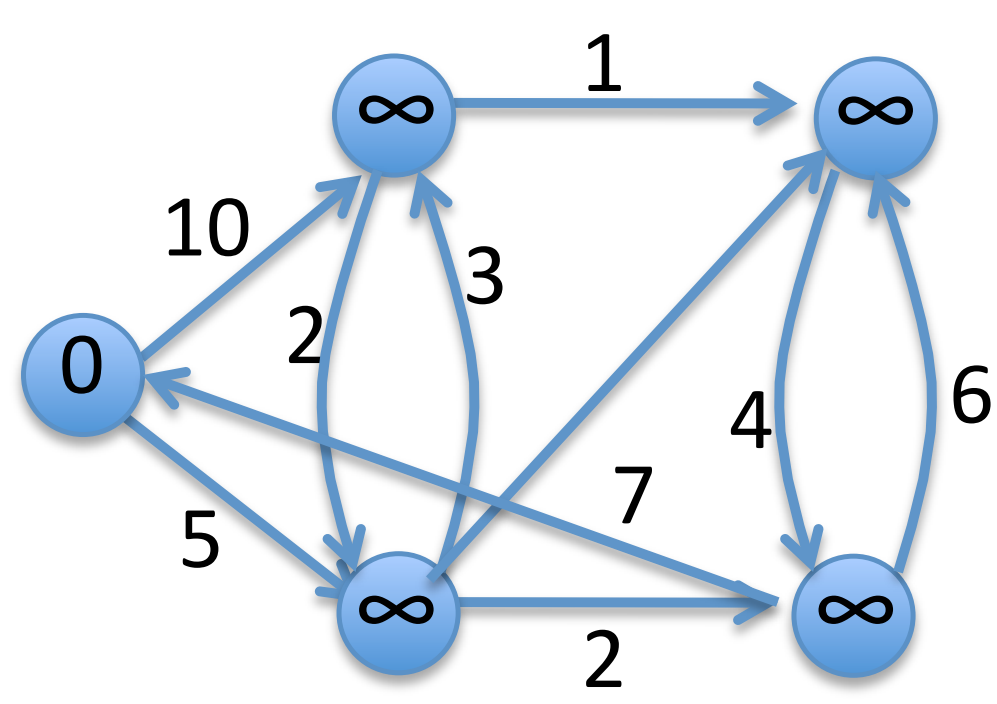
Dijkstra's Algorithm

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 $O(E \lg V)$)

Minimum Spanning Trees

Minimum Spanning Trees

Given a connected undirected graph, $G = (V, E)$ where each edge $(u, v) \in E$ is associated with a weight $w(u, v)$ specifying the cost of the edge, find an acyclic subset $T \subseteq E$ that connects all of the vertices in V and whose total weight:

$$w(T) = \sum_{(u,v)\in T} w(u, v)$$

is minimized.

This is a common problem:

- interconnect a set of pins in electronic circuitry using the least amount of wire
- find the least latency paths in a network

Minimum Spanning Tress (cont.)

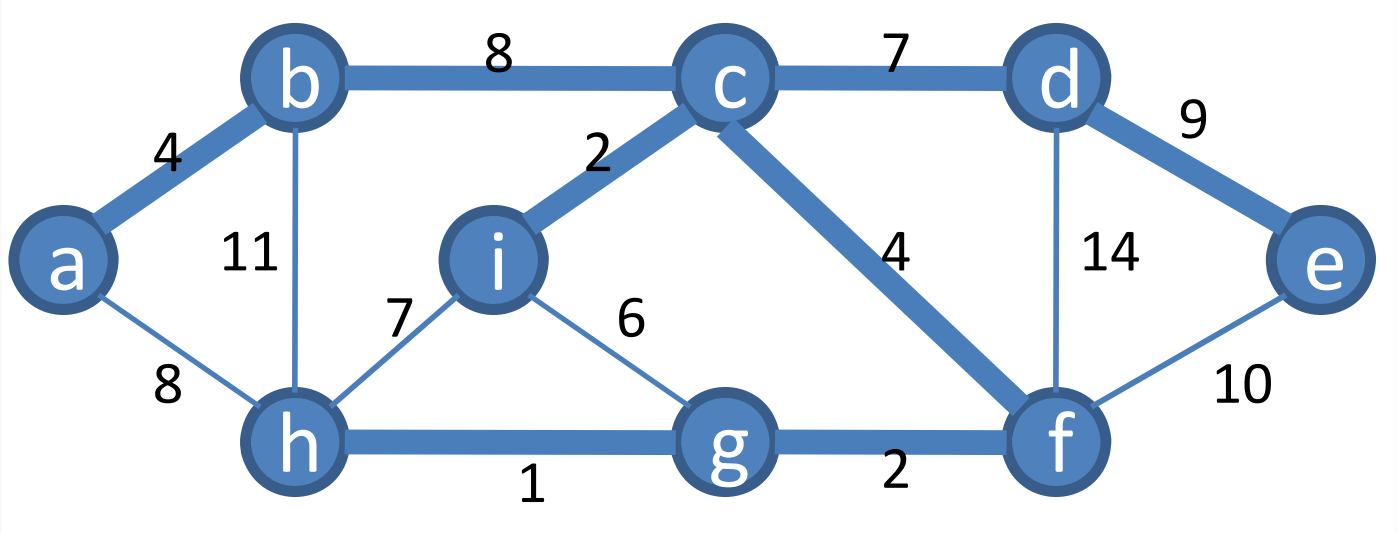
The resulting tree is a *spanning tree* since it touches all of the nodes in G

- we want the **minimum-weight-spanning tree** (or just **minimum spanning tree**)

We look at two algorithms to compute the minimum spanning tree. Both are greedy algorithms

- *greedy* refers to the fact that they make the "best" choice at this moment
- greedy algorithms are not always optimal (more later)
- our two minimum spanning tree algorithms *do* end up minimizing the total weight in the spanning tree

An Example MST



Growing a Minimum Spanning Tree

First, let's develop a generic algorithm for computing the minimum spanning tree (MST) by gradually growing the tree

Given a connected, undirected graph, $G(V, E)$ and a weight function $w : E \to \mathbf{R}$, find the minimum spanning tree of G.

Our approach is to grow a set A of edges that will eventually constitute our minimum spanning tree. What should our loop invariant be?

- Prior to each iteration, A is a subset of *some* minimum spanning tree

Growing a Minimum Spanning Tree (cont.)

Let a *safe edge* be any edge that can be added to A while maintaining the invariant.

At each step, we select any safe edge (u, v) and add it to A so that A becomes $A \cup \{(u, v)\}$

GENERIC-MST(G, w)

```
1  A ← ∅
2  while A does not form a spanning tree
3        do find an edge (u, v) that is safe for A
4           A ← A ∪ {(u, v)}
5  return A
```

Generic MST Loop Invariant

Remember our loop invariant:

- Prior to each iteration, A is a subset of some minimum spanning tree

Can you prove it true for this generically stated algorithm?

- **Initialization**: initially A is the empty set; it's a subset of the minimum spanning tree
- **Maintenance**: because the edge that we add is selected as a safe edge, it's guaranteed to maintain the invariant when it's added to A
- **Termination**: upon termination, A is a subset of a minimum spanning tree by the loop invariant, and A is a spanning tree by the negation of the guard on the while loop, so A must be a minimum spanning tree upon termination

The question is: how do we find a safe edge?

A Cut of an Undirected Graph

A **cut** $(S, V - S)$ of an undirected graph $G = (V, E)$ is a partition of V

- an edge (u, v) **crosses** a cut if one of u and v is in S and the other is in $V - S$
- a cut **respects** a set A of edges if no edges in A **cross** the cut
- an edge is a **light edge** crossing a cut if its weight is the minimum of all of the edges that cross the cut

Finding Safe Edges

Theorem

Let $G = (V, E)$ be a connected undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, let $(S, V - S)$ be any cut of G that respects A, and let (u, v) be a light edge crossing $(S, V - S)$. Then (u, v) is a safe edge for A.

Proof of the Safe Edge Theorem

Suppose that T is some minimum spanning tree that includes A but does not contain (u, v). We can construct an equivalent minimum spanning tree T' that includes $A \cup \{(u, v)\}$, demonstrating that (u, v) is safe for A. (u, v) forms a cycle with the edges on the path from u to v in T. u and v are on opposite sides of the cut $(S, V - S)$; because T is a spanning tree, there must be at least one edge in T that crosses the cut. Call it (x, y). $(x, y) \notin A$ because the cut respects A. Removing (x, y) and replacing it with (u, v) forms a new spanning tree $T' = T - \{(x, y)\} + \{(u, v)\}$. But is it a *minimum* spanning tree? (u, v) is a light edge crossing the cut $(S, V - S)$, so $w(u, v) \leq w(x, y)$. But T was a minimum spanning tree, so it must be that $w(u, v) = w(x, y)$, and $w(T') = w(T)$.

The Generic Algorithm Again

What are we really doing in GENERIC-MST?

- at any point, $G_A = (V, A)$ is a forest, and each connected component in G_A is a tree
- any safe edge for A connects distinct components from G_A since $A \cup \{(u, v)\}$ must be acyclic

Corollary

Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, and let $C = (V_C, E_C)$ be a connected component (tree) in the forest $G_A = (V, A)$. If (u, v) is a light edge connecting C to some other component in G_A, then (u, v) is safe for A.

Kruskal's Algorithm

Kruskal's algorithm builds directly on GENERIC-MST. At each step, the algorithm looks at all of the edges connecting any two trees in the forest G_A and chooses the smallest one.

- it's a greedy algorithm because at each step, it just takes the smallest of all of the possibilities

We use the algorithm for computing the connected components of a graph.

- we use a *disjoint-set* data structure to maintain several disjoint sets of elements (initially, each vertex is in its own set)
- FIND-SET(u) returns a representative element from the set that contains u.
- we can use FIND-SET to determine whether two vertices u and v belong to the same tree

Kruskal's Algorithm (cont.)

MST-KRUSKAL(G, w)

```
1  A ← ∅
2  for each vertex v ∈ G.V
3        do MAKE-SET(v)
4  sort the edges in G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, in nondecreasing order by weight
6        do if FIND-SET(u) ≠ FIND-SET(v)
7              then A ← A ∪ {(u, v)}
8                   UNION(u, v)
9  return A
```

Kruskal's Algorithm Running Time

What's the running time?

- initializing A is $O(1)$
- initializing the disjoint sets in lines 2-3 is $O(V)$
- sorting the edges in line 4 is $O(E \lg E)$
- the for loop runs for each edge, and does some amount of work that can be proven to be less than $O(\lg E)$ for each edge; the running time for the loop is therefore $O(E \lg E)$

So the overall running time is: $O(E \lg E)$

Prim's Algorithm

Prim's algorithm differs in that it chooses an arbitrary vertex and grows the minimum spanning tree from there

A is always a single tree.

Prim's algorithm is greedy since it always picks the smallest edge that could grow the tree.

The key challenge is storing and recalling the edges to make it easy to find the right edge to add to A.

Prim's Algorithm (cont.)

We construct a min-priority queue Q that holds all of the vertices that are not yet in the minimum spanning tree under construction

- the priority of each vertex v in Q is the minimum weight of any edge connecting v to any vertex in the tree
- we also store $v.\pi$, the parent on the other end of this minimum weight edge

Prim's algorithm does not have to explicitly store A, it's just:

$$A = \{(v, v.\pi) : v \in V - \{r\} - Q\}$$

where r is the provided root vertex

Prim's Algorithm (cont.)

MST-PRIM(G, w, r)

```
 1  for each u ∈ G.V
 2       do u.key ← ∞
 3          u.π ← NIL
 4  r.key ← 0
 5  Q ← G.V
 6  while Q ≠ ∅
 7       do u ← EXTRACT-MIN(Q)
 8          for each v ∈ Adj[u]
 9               do if v ∈ Q and w(u, v) < v.key
10                    then v.π ← u
11                         v.key ← w(u, v)
```

Prim's Algorithm Running Time

What's the running time?

- intialization in lines 1-5 is $O(V)$ assuming we use a min-heap as our priority queue
- the **while** loop is executed $|V|$ times, and each EXTRACT-MIN call takes $O(\lg V)$ time, giving us a total here of $O(V \lg V)$
- the **for** loop is executed $O(E)$ times total, and the implicit call to DECREASE-KEY takes $O(\lg V)$ time

So the total is $O(V \lg V + E \lg V) = O(E \lg V)$, which is asympotically the same as Kruskal's since $|E| < |V|^2$.

Notes on the Greedy Strategy

The Greedy Strategy in General

As a general rule, we are pretty direct in applying the greedy method:

1. Cast the optimization problem into one in which we make a choice and are left with only a single subproblem to solve.
2. Prove that there is always an optimal solution to the original problem that makes the greedy choice (i.e., that the greedy choice is *safe*)
3. Demonstrate that combining a solution to the remaining subproblem with the greedy choice yields an optimal solution to the original problem

Identifying Greedy Problems

In general, problems that can be solved by a greedy approach exhibit:

- optimal substructure — if an optimal solution to a problem contains within it optimal solutions to subproblems
- greedy-choice property — a globally optimal solution can be arrived at by making a locally optimal choice

The Knapsack Problem

0-1 Knapsack

A thief robbing a store finds n items; the i^{th} item is worth v_i dollars and weighs w_i pounds (both integers). The thief wants to take as valuable a load as possible but can carry only W pounds. Which items should he take?

Fractional Knapsack

The problem is the same as the 0-1 Knapsack problem, but the thief can take fractions of items; he doesn't have to take all or none of a particular item

Fractional Knapsack Problem

Does the problem display the optimal substructure property?

Yes. Take w of item j; then we're left with finding an optimal solution for weight $W - w$ given the remaining $n - 1$ items plus $w_j - w$ of item j

Solution

- calculate v_i / w_i for each item (the value per pound)
- sort the items by value per pound
- take items from the front of the list until the knapsack is full

0-1 Knapsack Problem

Does the problem display the optimal substructure property?

Yes. If we take item j, we're left with finding an optimal solution for weight $W - w_j$ given items $\{1, 2, \ldots, j-1, j+1, \ldots n\}$.

Does the problem satisfy the greedy-choice property?

No. Consider a knapsack that can hold 50 pounds and a set of three possible items: one that weighs 10 pounds and is worth \$60, one that weighs 20 pounds and is worth \$100, and one that weighs 30 pounds and is worth \$120.

0-1 Knapsack Problem (cont.)

What happened?

By choosing greedily, we didn't fill up the knapsack entirely, resulting in a lower effective value per pound of the knapsack.

What we should have done was compare the value of the subproblem that included the greedy choice with the value of the subproblem that *didn't*.

Huffman Codes

Huffman Codes

Huffman coding is a variable length data compression scheme that takes advantage of the fact that some characters are more common than others

- characters that are more common get shorter codes
- avoid ambiguity by requiring that no codeword is a prefix of another

For example, if the letters to encode are a, b, c, d, e, f, and the codes are, respectively, 0, 101, 100, 111, 1101, 1100:

- encoding is easy; just look up the code for each letter and concatenate them
- decoding is not bad since the coding is “prefix-free”
- e.g., decode 001011101

Decoding Huffman Codes

To do decoding, we need a fairly efficient representation of the code book that we can traverse quickly

We use a full binary tree whose leaves are characters represented by the code

- the codeword for a character is given by the path from the root to the leaf corresponding to the character
- decoding just traverses the tree following the pattern until it reaches a leaf. That's a character, then start over at the top.

The "optimal" encoding is a full binary tree; if not, we could "splice" out a node that only had one child and get shorter paths from the root to the leaves in that subtree.

Huffman Codes and Greedy Programming

Great. What does this have to do with greedy algorithms?

Let C be a set of n characters, each with frequency $f(c_i)$ in some file. Construct the optimal codebook of C that minimizes the number of bits needed to represent the file.

Given the codetree T, the number of bits needed to encode the file is:

$$B(T) = \sum_{c \in C} f(c) d_T(c)$$

where $d_T(c)$ is the depth of c's leaf in T.

Given the Tree Structure

What if we knew the tree structure of the optimal code?

Suppose that u and v are leaves of T^* (an optimal prefix code tree), and $\text{depth}(u) < \text{depth}(v)$. Suppose that in a labeling of T^* corresponding to an optimal prefix code, leaf u is labeled with $y \in S$ and leaf v is labeled with $z \in S$. Then $f_y \geq f_z$.

Prove this.

So given the structure of the tree, how would you assign the elements of S to the leaves?

Where are the most frequent characters? Where are the least frequent characters?

Constructing a Huffman Code

Approach

Start with $|C|$ leaves; perform $|C| - 1$ merging operations to create tree

- use a min heap keyed on f to merge the two least frequent objects
- the result is a new object with frequency that's the sum of the frequencies of the merged objects

HUFFMAN(C)

```
1  n ← |C|
2  Q ← C
3  for i ← 1 to n − 1
4       do allocate a new node z
5          left[z] ← x ← EXTRACT-MIN(Q)
6          right[z] ← y ← EXTRACT-MIN(Q)
7          f[z] ← f[x] + f[y]
8          INSERT(Q, z)
9          return EXTRACT-MIN(Q)
```

Correctness of Huffman's Algorithm: Greedy Choice

Lemma

Let C be an alphabet in which each $c \in C$ has frequency $f[c]$. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.

Proof

T is a full binary tree, so it must have at least two "deepest" leaves that are siblings. Since x and y have the lowest frequencies, we can swap them into these two spots without increasing the total cost of encoding.

What does this have to do with the greedy choice?

By choosing x and y with the lowest frequencies, Huffman picks the least cost merge at each step.

Correctness of Huffman's Algorithm: Substructure

Lemma

Let C be an alphabet with $f[c]$ defined for each $c \in C$. Let x and y be characters in C with lowest frequencies. Let $C' = C - \{x, y\} \cup z$ where $f[z] = f[x] + f[y]$. Let T' be any optimal prefix code for C'. Then the tree T obtained by replacing the leaf node for Z in T' with an internal node having x and y as children is an optimal prefix code for C.

Proof of Huffman Substructure

Let $d_T(c)$ be the depth of c in T (i.e., the length of c's code). The cost with respect to all nodes other than x and y is unchanged from T to T'.
$d_T(x) = d_T(y) = d_{T'}(z) + 1$. So

$$\begin{aligned} f[x]d_T(x) + f[y]d_T(y) &= (f[x] + f[y])(d_{T'}(z) + 1) \\ &= f[z]d_{T'}(z) + (f[x] + f[y]) \end{aligned}$$

$B(T) = B(T') + f[x] + f[y]$. Rewriting, $B(T') = B(T) - f[x] - f[y]$.

Proof by Contradiction. Suppose T was not an optimal prefix code for C. Then there is some T'', $B(T'') < B(T)$. Because x and y have the smallest frequencies in C, x and y are siblings in T'', too (that's the greedy choice property). Construct T''' that is exactly T'' except that the subtree rooted at the parent of x and y is replaced by a node z, whose frequency is the sum of the frequencies of x and y.

$$\begin{aligned} B(T''') &= B(T'') - f[x] - f[y] \\ &< B(T) - f[x] - f[y] \\ &< B(T') \end{aligned}$$

But that's a contradition, since T' was supposed to be optimal for C', and now we've found something (T''') for the same C' that's better.

Questions