Agenda

We've done

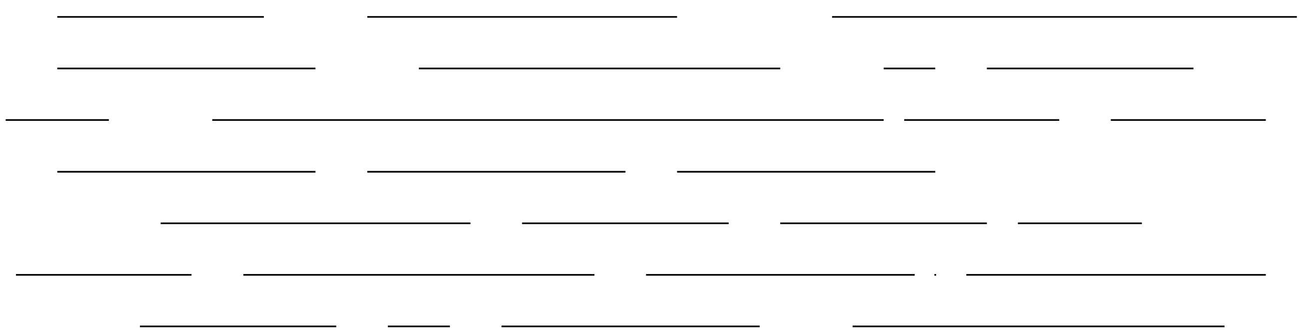
- Growth of functions
- Asymptotic Notations ($O, o, \Omega, \omega, \Theta$)
- Recurrence relations and a few methods of solving them
- Divide and Conquer

Now

- Designing Algorithms with the Greedy Method

Interval Scheduling – Problem Definition

- Scheduling requests on a single resource (a class room, a processor, etc.)
- **Input:**
    - a set $\mathcal{R} = \{R_1, \dots, R_n\}$ of n requests to be scheduled
    - R_i represented by the time interval $[s_i, f_i)$
- **Output:** a set of as many non-overlapping intervals as possible



Attempts at Greedy Choices

Note: after first interval is chosen, remove all conflicting intervals and then recurse

1. Select interval that starts earliest
2. Select shortest interval
3. Select interval that conflicts with fewest other intervals
4. Select interval that ends earliest

Greedy Algorithm - Recursive Implementation

Greedy-Interval-Scheduling($\mathcal{R}$)

```
if $\mathcal{R}$ is empty then
    Return $\emptyset$
else
    Let $R \in \mathcal{R}$ be the request with earliest finishing time
    Let $\mathcal{R}' \leftarrow (\mathcal{R}$ minus all requests overlapping with $R)$
    Return $\{R\} \cup$ Greedy-Interval-Scheduling($\mathcal{R}'$)
end if
```

Greedy Algorithm - A Better Implementation

Greedy-Interval-Scheduling($\mathcal{R}$)

```
if $\mathcal{R}$ is empty then
    Return $\emptyset$
end if
Sort intervals in increasing order of finishing times
// i.e. $f_1 \leq f_2 \leq \cdots \leq f_n$
$C \leftarrow \{R_1\}$ // select the first request
$j \leftarrow 1$      // record the last chosen request
for $i \leftarrow 2$ to $n$ do
    if $f_j \leq s_i$ then
        $C \leftarrow C \cup \{R_i\}$ // add $R_i$ to the output set
        $j \leftarrow i$              // record the last chosen request
    end if
end for
Return $C$
```

Proving Correctness

- Induct on $|\mathcal{R}|$ that our algorithm returns an optimal solution.
- Base case. When $|\mathcal{R}| = 1$, easy!
- Induction hypothesis. Suppose our algo is good when $|\mathcal{R}| \leq n - 1$.
- Induction step. Consider $|\mathcal{R}| = n$.
    - Claim 1: by the induction hypothesis, $C' = C - \{R_1\}$ is optimal for the sub-problem $\mathcal{R}'$
    - Hence,

$$\text{cost}(C) = 1 + \text{cost}(C') = 1 + \text{OPT}(\mathcal{R}') \tag{1}$$

    - Claim 2: there exists an optimal solution O containing the greedy choice (the first interval R_1)
    - Claim 3: $O' = O - \{R_1\}$ is an optimal solution for the sub-problem $\mathcal{R}'$
    - Thus,

$$\text{OPT}(\mathcal{R}) = \text{cost}(O) = 1 + \text{cost}(O') = 1 + \text{OPT}(\mathcal{R}') \tag{2}$$

    - Conclusion: (1) and (2) imply $\text{cost}(C) = \text{OPT}(\mathcal{R})$

Analysis

- Running time: $O(n \log n)$, where n is the number of intervals
- Space: $O(n)$ (Quick-sort swap elements in place, the output set of intervals is just an array.)

Optimization Problems

- Optimization problems: find an optimal solution among a large set of feasible solutions
  - 0-1 KNAPSACK: A robber found n items in a store, the ith item is worth v_i dollars and weighs w_i pounds ($v_i, w_i \in \mathbb{Z}^+$), he can only carry W pounds. Which items should he take?
  - TRAVELING SALESMAN (TSP): find the shortest route for a salesman to visit each of the n given cities once, and return to the starting city.
- Typically, a feasible solution is a combinatorial structure (graph, set, etc.) composed of smaller "building blocks".
- The combination of "building blocks" need to satisfy some conditions for the structure to be feasible

Greedy Algorithms

- It is hard to define what a “greedy” algorithm is
- Roughly: at each iteration, select a “locally optimal” building block

Example

01-KNAPSACK At each iteration, select the most valuable item that he could still carry

Example

01-KNAPSACK At each iteration, select the item with the most value per pound that he could still carry

- Easy to construct examples where both greedy strategies lead to non-optimal solutions

---

In General Terms

- A correct algorithm always returns an optimal solution. (“Correct algorithm” is somewhat of a misnomer. In general, an algorithm is correct if it always returns solutions as we intended it to return. The intention need not be optimality.)
- To prove that a greedy algorithm is incorrect, present **one** counter example.
- Note: an incorrect greedy algorithm may still give optimal solutions, depending on the inputs.
- To prove that a greedy algorithm is correct, there are two basic strategies (among others):
  - Induction
  - Exchange argument

Proving Correctness Using Induction – Strategy 1

- Examples: INTERVAL SCHEDULING, HUFFMAN CODING
- Often applicable to recursive greedy algorithms of the form
  1. select a locally optimal building block b (greedy choice)
  2. recursively construct a solution S' to suitably defined a sub-problem,
  3. combine b with S' to obtain the final solution $S = b \cup S'$
- Proof strategy
  1. By induction, show that $\text{cost}(S') = \text{OPT}(\text{ sub-problem })$, leading to
     $$\text{cost}(S) = \text{cost}(b) + \text{cost}(S') = \text{cost}(b) + \text{OPT}(\text{sub-problem}).$$
  2. Show that there is an optimal solution O containing b.
  3. Show that $O' = O - b$ is optimal for the sub-problem, implying
     $$\text{cost}(O) = \text{cost}(b) + \text{cost}(O') = \text{cost}(b) + \text{OPT}(\text{sub problem}).$$
  4. Conclude that
     $$\text{cost}(S) = \text{OPT}(\text{problem}).$$

Important Note

Do not interpret $\cup, +, -$ literally! Their meanings are problem specific.

Proving Correctness Using Induction – Strategy 2

- Induct that every “greedy step” produces a (growing) part of a globally optimal solution.
- Example: SINGLE-SOURCE SHORTEST PATHS.

Other strategies

There are other types of inductions too. Problem dependent.

Proving Correctness Using the Exchange Argument

1. Let S be the solution returned by the greedy algorithm
2. Let O be any optimal solution
3. Prove that O can be turned into S by gradual modifications without sacrificing the optimality of any solution along the way
    - 3.1 Gradual modification: remove some building block b from O, add another building block b' to obtain $O' = O - b + b'$
    - 3.2 Show that O' is feasible and cost(O') = cost(O)
    - 3.3 Show that, after a certain number of steps O becomes S
        - Often done by showing O' is "closer" to S in some specific sense

Examples:

- SCHEDULING TO MINIMIZE LATENESS,
- MINIMUM SPANNING TREES,
- OPTIMAL CACHING

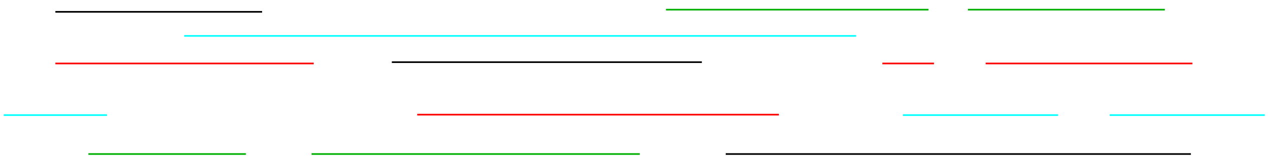
An abstraction

Strategy abstracted for a large class of problems using matroids

---

Scheduling All Interval – Problem Definition

- Scheduling all requests on as few resources as possible
- **Input:**
    - a set $\mathcal{R} = \{R_1, \dots, R_n\}$ of n requests to be scheduled
    - R_i represented by the time interval $[s_i, f_i)$
- **Output:** a partition of $\mathcal{R}$ into as few sets as possible, such that intervals in each set do not overlap.



Attempts at Greedy Choices

- Think of each set of non-overlapping intervals as a color
  - Colors are represented by integers (color 1, color 2, etc.)
  - Partitioning becomes coloring
  - Two conflicting intervals need different colors
- Possible strategies
  1. Consider intervals one at a time, assign to a new interval the least non-conflicting integer.
  2. Sort intervals by starting times, then use strategy 1.

A lower bound

Let d be the maximum number of mutually overlapping intervals, then we need at least d colors.

Greedy Algorithm

We will use only d colors: $[d] = \{1, \dots, d\}$

Greedy-Interval-Partitioning($\mathcal{R}$)

```
Sort requests by their starting times, breaking ties arbitrarily
// now $s_1 \leq s_2 \leq \cdots \leq s_n$
for $j = 1$ to $n$ do
    for each $R_i$ preceding $R_j$ and overlaps $R_j$ do
        Exclude color of $R_i$ from consideration for $R_j$
    end for
    if there is any color in $[d]$ available then
        Use that color for $R_j$
    else
        Leave $R_j$ un-colored
    end if
end for
```

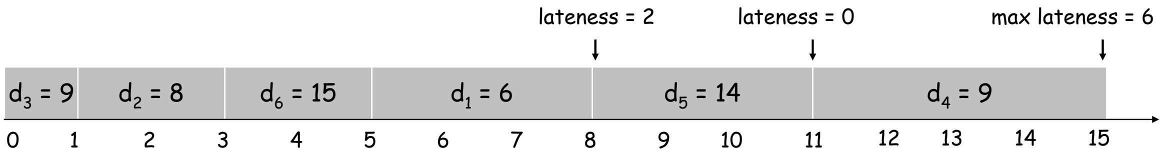
Proof of Correctness

- Let J be the set of intervals overlapping R_j
- Then, $J \cup \{R_j\}$ is a mutually-overlapping set of intervals
- Thus $|J| \leq d - 1$.
- Thus, there is always an available color for R_j
- Since we used only d colors, our algorithm is optimal.

Scheduling to Minimize Lateness – Problem Definition

- **Input:** n jobs; job J_i has duration t_i and deadline d_i
- **Output:** a schedule on a single machine to minimize the maximum lateness.
- Lateness is the amount of time a job is late compared to its deadline, and is 0 if the job is on-time.

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

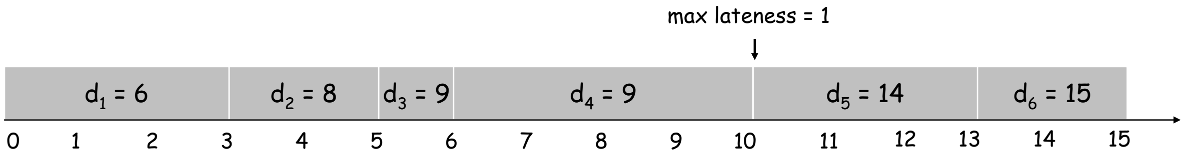


Attempts at Greedy Choices

1. Shortest processing time first
2. Shortest slack time first (slack time is $d_i - t_i$)
3. Earliest deadline first

Greedy Algorithm – Earliest Deadline First

```
Scheduling-Minimize-Lateness($\mathbf{t}, \mathbf{d}$)
Sort jobs so that $d_1 \le d_2 \le \cdots \le d_n$
$f = 0$
for $i = 1$ to $n$ do
    Assign job $J_i$ to the time interval $s_i = f, f_i = f + t_i$
    $f \leftarrow f + t_i$
end for
```



Proof of Correctness

- A schedule has an inversion if $d_i > d_j$ yet J_i is scheduled before d_j
- The machine has idle time if it's free for a while between some two jobs

Claim 1

There is an optimal schedule with no idle time

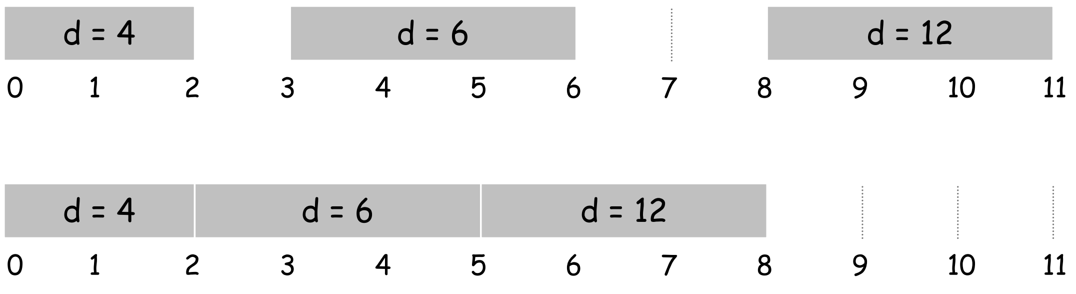
Claim 2

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

Claim 3

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

Proof of Claim 1

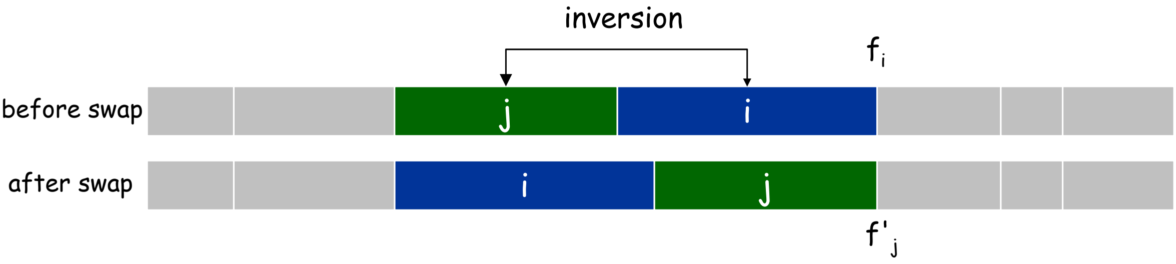


Proof of Claim 2

- Schedules with no inversions and no idle times might differ in ordering of jobs with the same deadline
- Among jobs with the same deadline, say d_i, the last one has the maximum lateness, which does not depend on the ordering of these jobs

Proof of Claim 3

- Let O be any optimal schedule with no idle time
- If O has an inversion, there is an inversion where jobs i and j are next to each other in the schedule
- Swap i and j does not change optimality, yet reduces the number of inversions by 1
- Max number of inversions is a polynomial in n (how many?), thus the method terminates in polynomial time



Single Source Shortest Paths – Problem Definition

- $G = (V,E)$, a path is a sequence of vertices $P = (v_0, v_1, \dots, v_k)$, where $(v_i, v_{i+1}) \in E$, and no vertex is repeated
- A walk is the same kind of sequence with repeated vertices allowed
- If $w : E \to \mathbb{R}$, then $w(P) = w(v_0 v_1) + \cdots + w(v_{k-1} v_k)$.

Single Source Shortest Paths Problem

Given a directed graph $G = (V,E)$, a source vertex $s \in V$, and a weight function $w : E \to \mathbb{R}^+$.
Find a shortest path from s to each vertex $v \in V$

Question

What if the graph is undirected?

Representing Shortest Paths

How do we represent shortest paths from s to each vertex $v \in V$?

Lemma

If $P = (s, \dots, u, v)$ is a shortest path from s to v, then the part of P from s to u is a shortest path from s to u.

Shortest Path Tree

For each $v \in V$, maintain a pointer $\pi[v]$ to the previous vertex along a shortest path from s to v. For the rest,

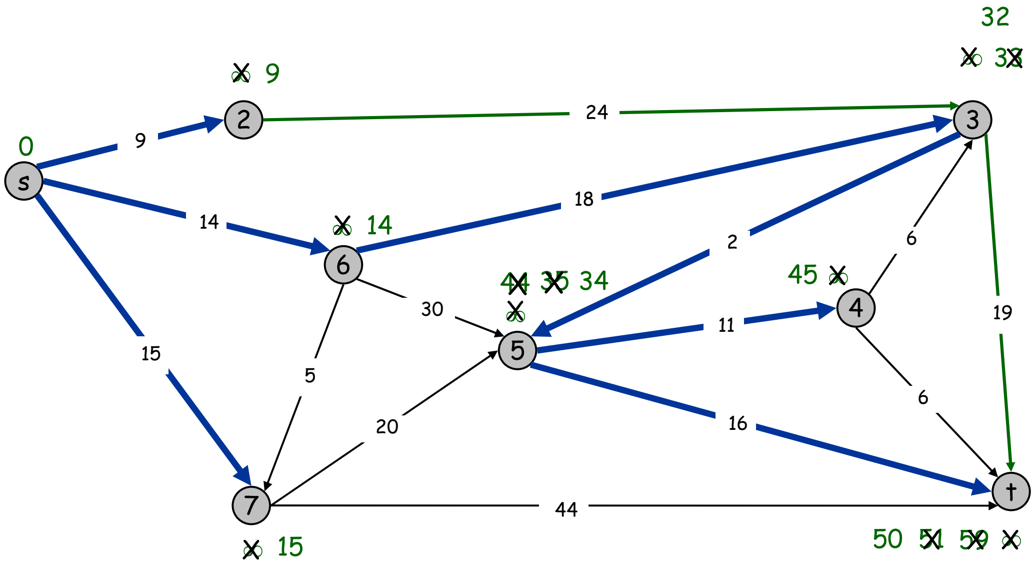
$$\begin{aligned} \pi[s] &= \text{NIL} \\ \pi[v] &= \text{NIL if } v \text{ is not reachable from } s \end{aligned}$$

Notes:

- There could be multiple shortest paths to the same vertex
- The representation gives one set of shortest paths

Shortest Path Tree

S = { s, 2, 3, 4, 5, 6, 7, t }
PQ = { }



Dijkstra's Algorithm

- $d[v]$: current estimate of the weight of a shortest path to v
- $\pi[v]$: pointer to the previous vertex on the shortest path to v

```
DIJKSTRA$(G, s, w)$
Set $d[v] \leftarrow \infty$ and $\pi[v] \leftarrow$ NIL, for all $v$
$d[s] \leftarrow 0$; $S \leftarrow \{s\}$ // $S$ is the set of explored nodes
while $S \neq V$ do
    Choose $v \notin S$ with at least one edge from $S$ for which
        $d'(v) = \min_{(u,v)\in E, u\in S} \{d[u] + w(u, v)\}$
    is as small as possible. Let $u \in S$ be the vertex realizing the minimum $d'(v)$
    $S \leftarrow S \cup \{v\}$; $d[v] \leftarrow d'(v)$, $\pi[v] \leftarrow u$
end while
```

Better Implementation

- $d[v]$: current estimate of the weight of a shortest path to v
- $\pi[v]$: pointer to the previous vertex on the shortest path to v

INITIALIZE-SINGLE-SOURCE(G, s)

```
for each $v \in V(G)$ do
    $d[v] \leftarrow \infty$
    $\pi[v] \leftarrow$ NIL
end for
$d[s] \leftarrow 0$
```

RELAX(u, v, w)

```
if $d[v] > d[u] + w(u, v)$ then
    $d[v] \leftarrow d[u] + w(u, v)$
    $\pi[v] \leftarrow u$
end if
```

Better Implementation – Priority Queues

A priority queue is a data structure that

- maintains a set S of objects
- for each $s \in S$, $key[s] \in \mathbb{R}$

Two types: min-priority queue and max-priority queue

Min-Priority Queue – denoted by Q

- INSERT(Q, x): insert x into Q
- MINIMUM(Q): returns element with min key
- EXTRACT-MIN(Q): removes and returns element with min key
- DECREASE-KEY(Q, x, k): change $key[x]$ to k, where $k \leq key[x]$

Using Heap, Min-PQ can be implemented so that:

- Building a Q from an array takes $O(n)$
- Each of the operations takes $O(\lg n)$

Better Implementation

```
DIJKSTRA($(G, s, w)$)
INITIALIZE-SINGLE-SOURCE($(G, s)$)
$S \leftarrow \emptyset$        // set of vertices considered so far
$Q \leftarrow V(G)$ // $\forall v$, $key[v] = d[v]$ after initialization
while $Q$ is not empty do
    $u \leftarrow$ EXTRACT-MIN($(Q)$)
    $S \leftarrow S \cup \{u\}$
    for each $v \in Adj[u]$ do
        RELAX($(u, v, w)$)
    end for
end while
```

Running Time

PQ Op.	Dijkstra	Array	Bin. Heap	d-way Heap	Fib. Heap
INSERT	n	n	$\lg n$	$d \log_d n$	1
EXR-MIN	n	n	$\lg n$	$d \log_d n$	$\lg n$
DEC-KEY	m	1	$\lg n$	$\log_d n$	1
IS-EMPTY	n	1	1	1	1
Total		n^2	$m \lg n$	$m \log_{m/n} n$	$m + n \lg n$

Correctness of Dijkstra's Algorithm

For each u, let P_u be the path from s to u in the shortest path tree returned by Dijkstra's algorithm.

Theorem

Consider the set S at any point in the execution of the algorithm. For each vertex $u \in S$, the path P_u is a shortest s–u path

Proof.

Induction on $|S|$. ☐

Analysis of Dijkstra's Algorithm

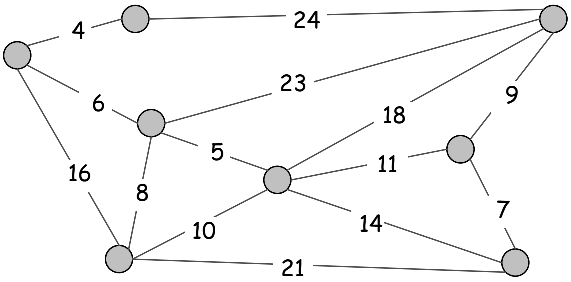
Let $n = |V(G)|$, and $m = |E(G)|$

- INITIALIZE-SINGLE-SOURCE takes $O(n)$
- Building the queue takes $O(n)$
- The while loop is done n times, so EXTRACT-MIN is called n times for a total of $O(n \lg n)$
- For each u extracted, and each v adjacent to u, RELAX(u, v, w) is called, hence totally $|E|$ calls to RELAX were made
- Each call to RELAX implicitly implies a call to DECREASE-KEY, which takes $O(\lg n)$; hence, totally $O(m \lg n)$-time on DECREASE-KEY

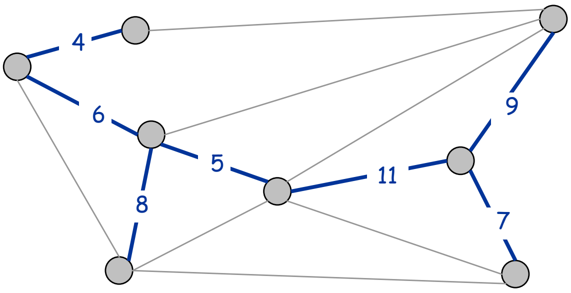
In total, we have $O((m + n) \lg n)$, which could be improved using FIBONACCI-HEAP to implement the priority queue

Minimum Spanning Tree – Problem Definition

- Input: a connected graph $G = (V, E)$, edge cost $c : E \to \mathbb{R}^+$
- Output: a spanning tree T of G, i.e. a connected sub-graph with no cycle which spans all vertices.



G = (V, E)



T, $\Sigma_{e \in T} c_e = 50$

Attempts at Greedy Choices

- (Kruskal) Start with $T = \emptyset$. Consider edges in ascending order of costs. Add edge e into T unless e completes a cycle in T.
- (Prim) Start from any vertex s of G. Grow a tree T from s. At each step, add the cheapest edge e with exactly one end in T
- (Reverse Delete) Start with $T = E$. Consider edges in descending order of costs. Remove e from T unless doing so disconnects T

Amazingly

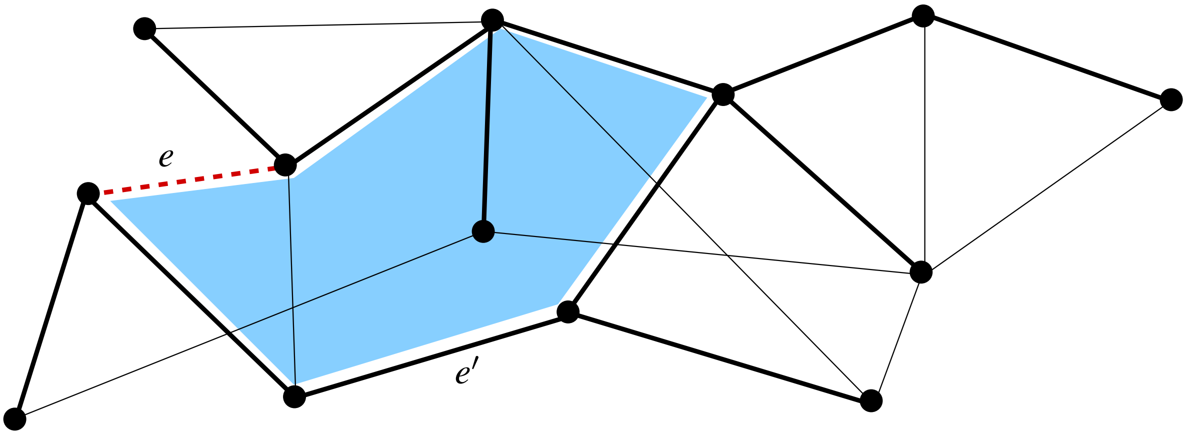
All three attempts are good greedy algorithms

Proof of Correctness – An Exchange Lemma

Lemma (Exchange Lemma)

Let T be any minimum spanning tree of G. Let e be any edge of G with $e \notin T$. Then,

- *e forms a cycle with some edges of T,*
- *all edges on this cycle has cost at least $c(e)$*



$c(e) \geq c(e')$

Correctness of Kruskal's and Prim's Algorithms

Kruskal is correct

Let $e_1, \dots, e_{n-1}$ be edges of the tree that Kruskal algorithm selects, in that order. Prove by induction that, for each $i \in \{1, \dots, n-1\}$ there exists an MST containing $e_1, \dots, e_i$.

Prim is correct

Let $e_1, \dots, e_{n-1}$ be edges of the tree that Prim algorithm selects, in that order. Prove by induction that, for each $i \in \{1, \dots, n-1\}$ there exists an MST containing $e_1, \dots, e_i$.

Reverse-delete is correct

Let $e_1, \dots, e_{m-(n-1)}$ be edges that REVERSE-DELETE deleted during its execution, in that order. Prove by induction that, for each $i \in \{1, \dots, m-(n-1)\}$ there exists an MST not containing any of $e_1, \dots, e_i$.

Implementing Prim's Algorithm with a Priority Queue

- Similar to Dijkstra's algorithm, grow the tree from S
- For each unexplored v, maintain "attachment cost" $a[v] =$ cost of cheapest edge connecting S to v

MST-PRIM(G, w)

```
$a[v] \leftarrow \infty, \forall v \in V$;   $S \leftarrow 0$,   $Q \leftarrow \emptyset$
Insert all $v$ into $Q$
while $Q$ is not empty do
    $u \leftarrow$ EXTRACT-MIN$(Q)$;   $S \leftarrow S \cup \{u\}$
    for each $v$ such that $e = (u, v) \in E$ do
        if $w_e < a[v]$ then
            DECREASE-KEY$(Q, v, w_e)$
        end if
    end for
end while
```

Time: $O(n^2)$ with an array as Q, $O(m \lg n)$ with a binary heap.

Implementing Kruskal's Algorithm with Union-Find Data Structure

MST-Kruskal(G, w)

```
$A \leftarrow \emptyset$ // the set of edges of $T$
Sort $E$ in increasing order of costs // $c(e_1) \leq \cdots \leq c(e_m)$
for each vertex $v \in V(G)$ do
    MAKE-SET$(v)$
end for
for $i = 1$ to $m$ do
    // Suppose $e_i = (u, v)$
    if FIND-SET$(u) \neq$ FIND-SET$(v)$ then
        $A \leftarrow A \cup \{e_i\}$
        SET-UNION$(u, v)$
    end if
end for
```

- It is known that $O(m)$ set operations take $O(m \lg m)$.
- Totally, Kruskal's Algorithm takes $O(m \lg m)$.

Huffman Coding – Problem Definition

- 7-bit ASCII code for “abbccc” uses 42 bits
- Suppose we use ’0’ to code ’c’, ’10’ to code ’b’, and ’11’ to code ’c’: “111010000” - 9 bits
- To code effectively:
  - Variable codes
  - No code of a character is a prefix of a code for another: prefix code
  - The characters with higher frequencies should get shorter codes
- Prefix codes can be represented by binary trees with characters at leaves
- The binary trees have to be full if we want the code to be optimal (why?)
- The problem: given the frequencies, find an optimal full binary tree

More Precise Formulation

- **Input:**
  - C: the set of characters
  - Frequency $f(c)$ for each $c \in C$
- **Output:** an optimal coding tree T.
  Let $d_T(c)$ be the depth of a leaf c of T
  The total number of bits required is

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

We want to find T with the least $B(T)$

Huffman's Idea

HUFFMAN'S ALGORITHM

```
while there are two or more leaves in $C$ do
    Pick two leaves $x, y$ with least frequency
    Create a node $z$ with two children $x, y$, and frequency $f(z) = f(x) + f(y)$
    $C = (C - \{x, y\}) \cup \{z\}$
end while
```

Correctness of Huffman's Algorithm

Lemma

Let C be a character set, where each $c \in C$ has frequency $f(c)$. Let x and y be two characters with least 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

Lemma

Let T be a full binary tree representing an optimal prefix code for C. Let x and y be any leaves of T which share the same parent z. Let $C' = (C - \{x, y\}) \cup \{z\}$, with $f(z) = f(x) + f(y)$. Then, $T' = T - \{x, y\}$ is an optimal tree for C'.

Optimal Caching – Problem Definition

Setting

- A cache with capacity to store k items, $k < n$, initially full
- A sequence of n requests for items: $d_1, \dots, d_n$
- Cache hit: requested item already in cache when requested
- Cache miss: requested item not in cache, must evict some item to bring requested item into cache

Objective find an eviction schedule (which item(s) to evict and when) to minimize the number of evictions

Example: $k = 2$, initial cache ab, requests a, b, c, b, c, a, b

Cache content	a b	a b	c b	c b	c b	a b	a b
Requests	a	b	c	b	c	a	b

Attempts at Greedy Choices

- First In Last Out (FILO)
- First In First Out (FIFO)
- Evict item least frequently used in the past
- Evict item referenced farthest into the past (Least Recently Used – LRU)
- Evict item least frequently used in the future
- Evict item needed the farthest into the future (Les Belady's idea, 1960s): this works!

Reduced Schedules

Notes:

- At each step, we can evict and bring in as many items as we wish
- We can assume that the cache is always full

Reduced schedules:

- A schedule is reduced if it only brings in an item at the point when the item is requested (and missed)
- Every schedule S can be transformed into a reduced schedule S' with the same number of misses $\Rightarrow$ there is a reduced optimal schedule!

Transforming Schedules to Reduced Schedules

Evicted Item	.	.	e	.	.	.	.
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	.	.	d	.	.	.	.
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
Requests	.	.	x	.	.	.	d

- Say, d was inserted before needed, e was sacrificed
- If e is brought back in before d is requested $\Rightarrow$ miss
- If e is not brought back in before d is requested, e could have just remained, and bring d in when requested

Correctness of Farthest-in-Future

Let S_F be the schedule returned by Farthest-in-Future

We show by induction on j that

For every $j \geq 0$, there exists a reduced optimal schedule S which makes the same evictions as S_F through the first j steps.

- Base case: $j = 0$ is obvious.
- Let S be a reduced optimal schedule agreeing with S_F 'til step j
- Consider step $j + 1$: suppose d is requested, S_F evicts e, S evicts f
  - Define another S': S' evicts e, then mimics S as far as possible
  - The first time S' can't follow S, suppose g is requested
  - Case 1: $g \neq e$, $g \neq f$, S evicts e
  - Case 2: $g = f$, (2a) S evicts e, (2b) S evicts $e' \neq e$
  - Case 3: $g = e$ – impossible!
  - Thus, S' is optimal and agree with S_F till step $j + 1$

S and S_F

Evicted	.	.	e	.	.	.	.	.	.	.
S_F	.	.	.	.	.	.	.	.	.	.
	.	.	d	.	.	.	.	.	.	.
	.	.	.	.	.	.	.	.	.	.
	.	.	f	.	.	.	.	.	.	.
Requests	.	.	d	.	.	.	.	.	.	.

Evicted	.	.	f	.	.	.	.	.	.	.
S	.	.	.	.	.	.	.	.	.	.
	.	.	d	.	.	.	.	.	.	.
	.	.	.	.	.	.	.	.	.	.
	.	.	e	.	.	.	.	.	.	.
Requests	.	.	d	.	.	.	.	.	.	.

S and S', Case 1: $g \neq e, g \neq f$

Evicted	.	.	e	.	.	.	f	.	.	.
S'	.	.	.	.	.	.	.	.	.	.
	.	.	d	.	.	.	.	.	.	.
	.	.	.	.	.	.	.	.	.	.
	.	.	f	.	.	.	g	.	.	.
Requests	.	.	d	.	.	.	g	.	.	.

Evicted	.	.	f	.	.	.	e	.	.	.
S	.	.	.	.	.	.	.	.	.	.
	.	.	d	.	.	.	.	.	.	.
	.	.	.	.	.	.	.	.	.	.
	.	.	e	.	.	.	g	.	.	.
Requests	.	.	d	.	.	.	g	.	.	.

Case 2a: $g = f$, S evicts e

Evicted	.	.	e	.	.	.	.	.	.	.
S'	.	.	.	.	.	.	.	.	.	.
	.	.	d	.	.	.	.	.	.	.
	.	.	.	.	.	.	.	.	.	.
	.	.	f	.	.	.	f	.	.	.
Requests	.	.	d	.	.	.	f	.	.	.

Evicted	.	.	f	.	.	.	e	.	.	.
S	.	.	.	.	.	.	.	.	.	.
	.	.	d	.	.	.	.	.	.	.
	.	.	.	.	.	.	.	.	.	.
	.	.	e	.	.	.	f	.	.	.
Requests	.	.	d	.	.	.	f	.	.	.

Case 2b: $g = f$, S evicts $e' \neq e$

Evicted	.	.	e	.	.	.	e’	.	.	.
S'	.	.	.	.	.	.	.	.	.	.
	.	.	d	.	.	.	.	.	.	.
	.	.	.	.	.	.	e	.	.	.
	.	.	f	.	.	.	f	.	.	.
Requests	.	.	d	.	.	.	f	.	.	.

Evicted	.	.	f	.	.	.	e’	.	.	.
S	.	.	.	.	.	.	.	.	.	.
	.	.	d	.	.	.	.	.	.	.
	.	.	.	.	.	.	f	.	.	.
	.	.	e	.	.	.	e	.	.	.
Requests	.	.	d	.	.	.	f	.	.	.