EC504 ALGORITHMS AND DATA STRUCTURES
FALL 2020 MONDAY & WEDNESDAY
2:30 PM - 4:15 PM

Prof: David Castañón, dac@bu.edu

GTF: Mert Toslali, toslali@bu.edu

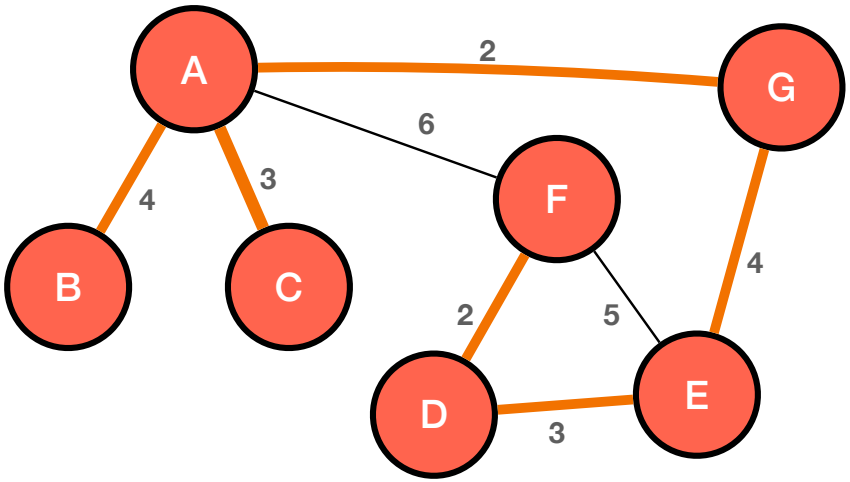
Haoyang Wang: haoyangw@bu.edu

Christopher Liao: cliao25@bu.edu

Minimum Spanning Trees

- A **minimum spanning tree** (MST) of an undirected weighted graph (V, E) with weights $w(\cdot)$:

  Connected subgraph (V, E') which is a tree and for which $\sum_{\{i,j\} \in E'} w(i, j)$ is minimized



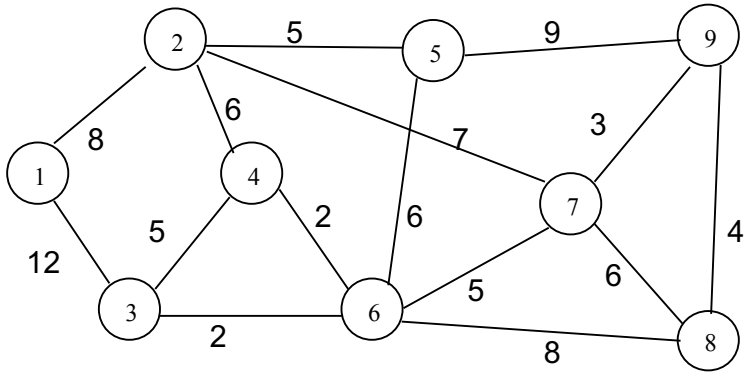
Kruskal's Algorithm

- Sort edges by weight in ascending order
- Start with empty set T (note: it is promising)
- For each edge e in sorted list
  - If adding edge e to T does not create cycle in $(V, T \cup e)$
  - …add it to MST: $T = T \cup \{e\}$
  - Claim: T is now promising set with one more edge
- Stop when you have $\#V - 1$ edges in T

Kruskal Runtime

- O(|V|) for iterating through vertices
- O(|E|log|E|) for sorting edges
- O(|E|×1) for iterating through edges and merging clouds with path compression
- O(|V|+|E|log|E|+|E|×1)
  - = O(|V|+|E|log|E|)
- O(|V|+|E|log|E|)
  - Better than simple $O(|V|^3)$ without disjoint sets

Example



Disjoint components: Minimum Spanning Tree

1 2 3 4 5 6 7 8 9 Empty

1) Add {3,6}

Disjoint components: Minimum Spanning Tree

3 1 2 4 5 7 8 9 3--6

|

6

2) Add {4,6}

3 1 2 5 7 8 9 3--6--4

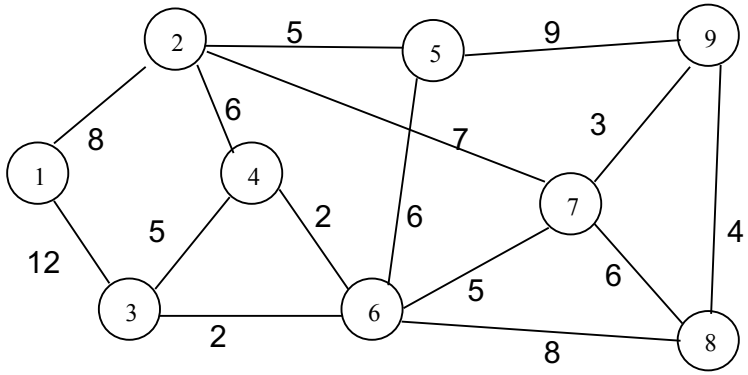
/ \

6 4

Sorted edges

{3,6}, {4,6},{7,9},{8,9},{3,4},{6,7}, {2,5},{2,4}{5,6},{7,8}, {2,7},{1,2}, {6,8},{5,9},{1,3}

Example



Disjoint components:

3) Add {7,9}

3 7 1 2 5 8

/ \ |

6 4 9

4) Add {8,9}

3 7 1 2 5

/ \ / \

6 4 9 8

Minimum Spanning Tree

3--6--4 7--9

3--6--4 7--9--8

Sorted edges

{3,6}, {4,6},{7,9},{8,9},{3,4},{6,7},
{2,5},{2,4}{5,6},{7,8}, {2,7},{1,2},
{6,8},{5,9},{1,3}

Example

Disjoint components:

5) Discard {3,4} (cycle); add {6,7}

```
   3       1  2   5

 /  | \

6  4   7

      /  \

     9    8
```

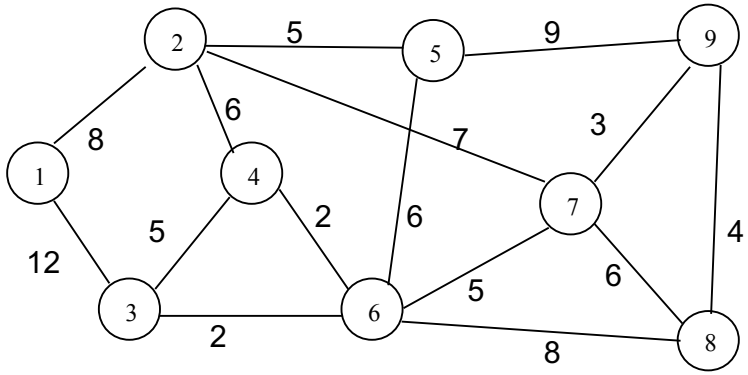
6) Add {2,5}

```
   3       1  2

 /  | \         \

6   4   7        5

      /  \

     9    8
```

Minimum Spanning Tree

```
3--6--4

       \

        7--9--8
```

```
3--6--4       2--5

       \

        7--9--8
```



Sorted edges

{3,6}, {4,6},{7,9},{8,9},{3,4},{6,7}, {2,5},{2,4}{5,6},{7,8}, {2,7},{1,2}, {6,8},{5,9},{1,3}

Example

Disjoint components:

7) Add {2,4}

```
  ( 3   )       1
 / | \   \
6  4   7   2
     / \   \
    9   8   5
```

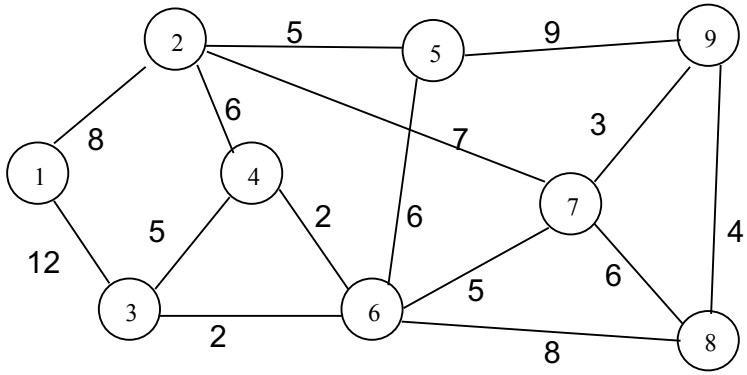
8) Discard {5,6},{7,8}, {2,7}. Select arc {1,2} . DONE

```
   (    3      )
 / | \   \  \ \  \
6  4   7   2  1  5   8
     /
    9
```

Minimum Spanning Tree

```
3--6--4--2--5
      \
      7--9--8
```

```
3--6--4--2--5
      \     \
       \     1
       7--9--8
```



Sorted edges

{3,6}, {4,6},{7,9},{8,9},{3,4},{6,7}, {2,5},{2,4}{5,6},{7,8}, {2,7},{1,2}, {6,8},{5,9},{1,3}

Different MST Algorithm: Prim-Jarmik

- Traverse G = (V,E) starting at any node
  - Maintain priority queue of nodes (e.g. binary heap, Fibonacci heap)
  - set priority to weight of the cheapest edge that connects them to MST
- Un-added nodes start with priority ∞
- At each step
  - Add the node with lowest cost to MST
  - Update ("relax") neighbors as necessary
- Stop when all nodes added to MST

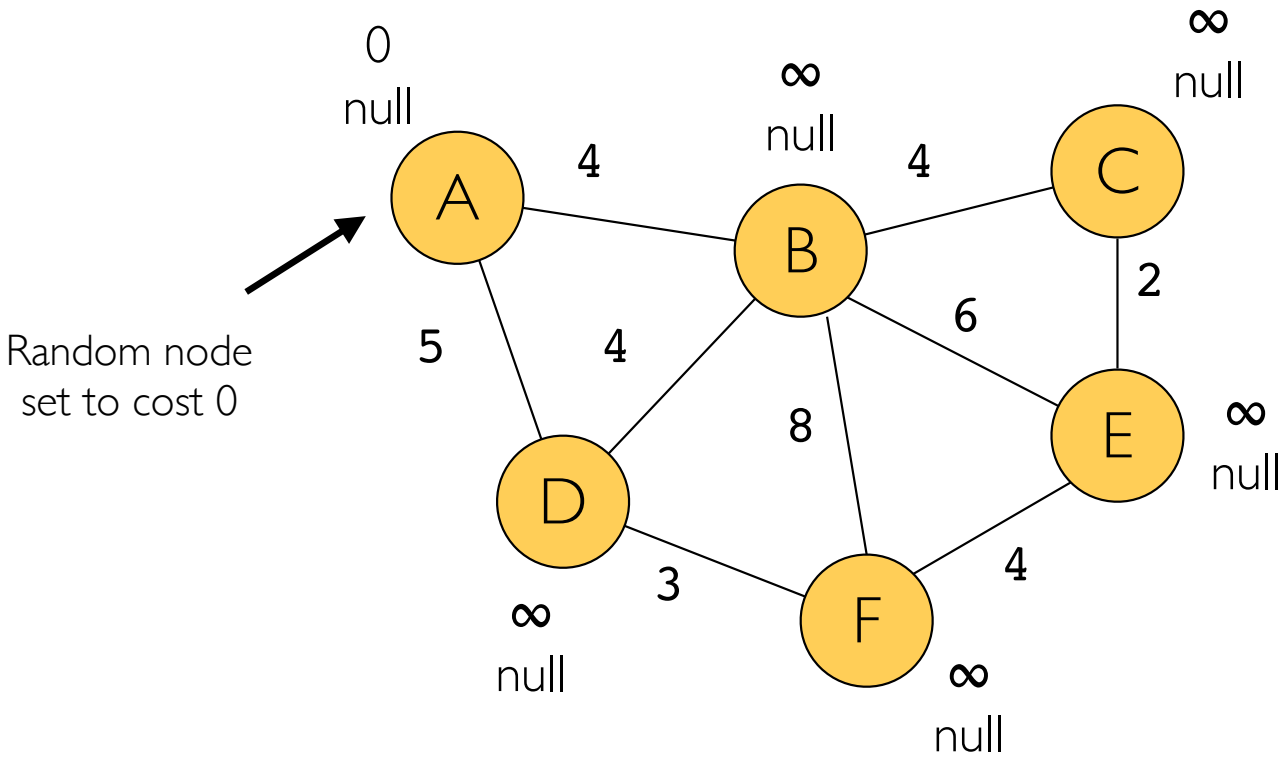
Different MST Algorithm: Prim-Jarnik

- Traverse G = (V,E) starting at any vertex v
- Form priority queue PQ of nodes (e.g. binary heap, Fibonacci heap)
  - Set v.dist = 0, v′.dist = ∞, v′ different from v. dist will be the key used in the priority queue
  - Set parent of all vertices, v.pred to NULL; set $T = \varnothing$
- At each step:
  - Remove min from priority queue: call it v. If v.pred not null, add {v,v.pred} to T
  - For all neighbors v′ of v: if v′ is in PQ and $w(v, v') <$ v′.dist:
    - Decrease v′.dist to $w(v, v')$, update v′.pred = v
- Stop when PQ is empty

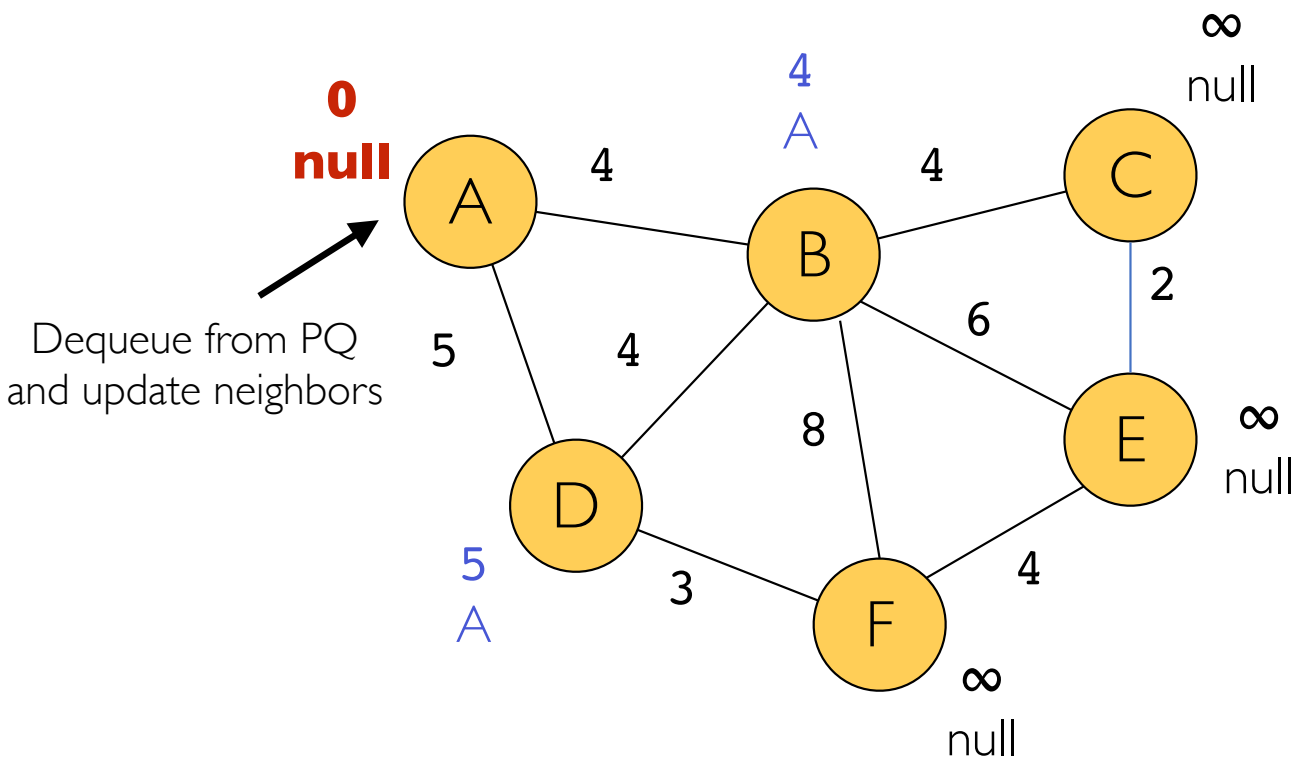
Pseudo-code

```
function prim(G):
   // Input: weighted, undirected graph G with vertices V
   // Output: list of edges in MST
   for all v in V:
      v.cost = ∞
      v.prev = null
   s = a random v in V // pick a random source s
   s.cost = 0
   MST = []
   PQ = PriorityQueue(V) // priorities will be v.cost values
   while PQ is not empty:
      v = PQ.removeMin()
      if v.prev != null:
         MST.append((v, v.prev))
      for all incident edges (v,u) of v such that u is in PQ:
         if u.cost > (v,u).weight:
            u.cost = (v,u).weight
            u.prev = v
            PQ.decreaseKey(u, u.cost)
  return MST
```

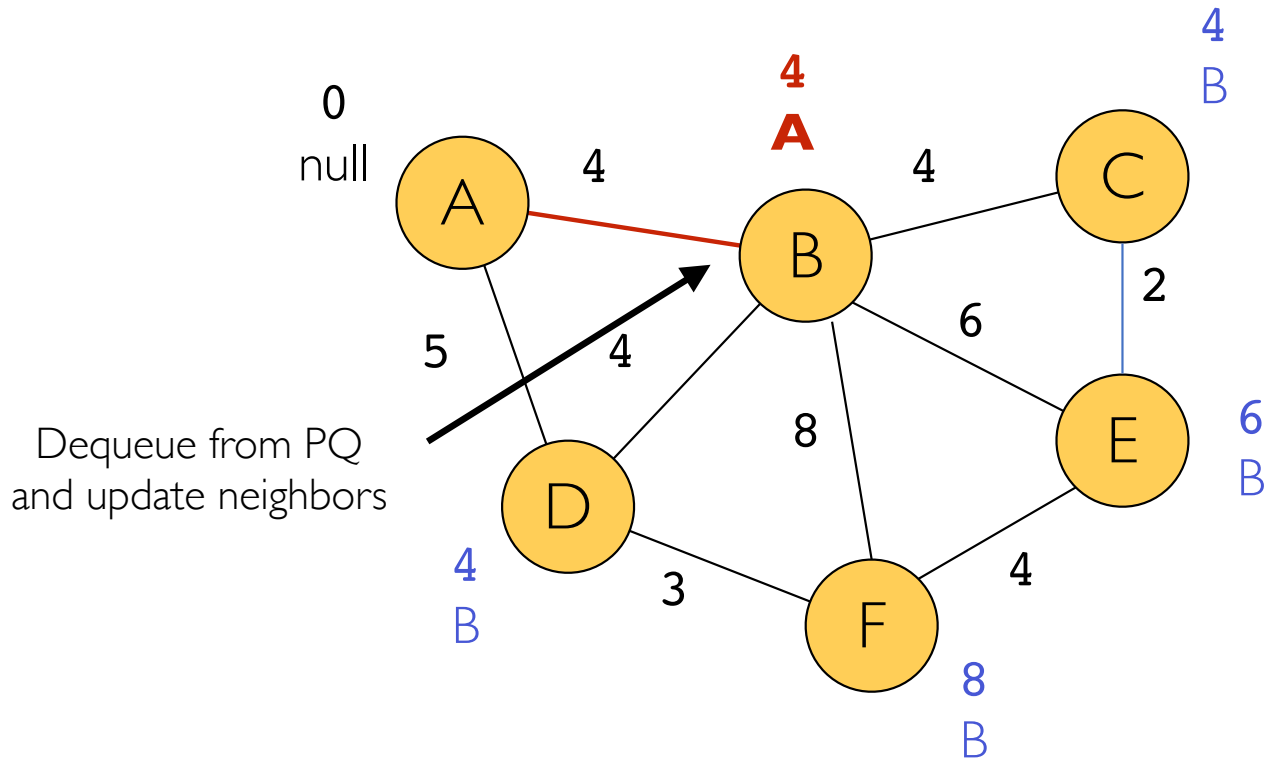
Example



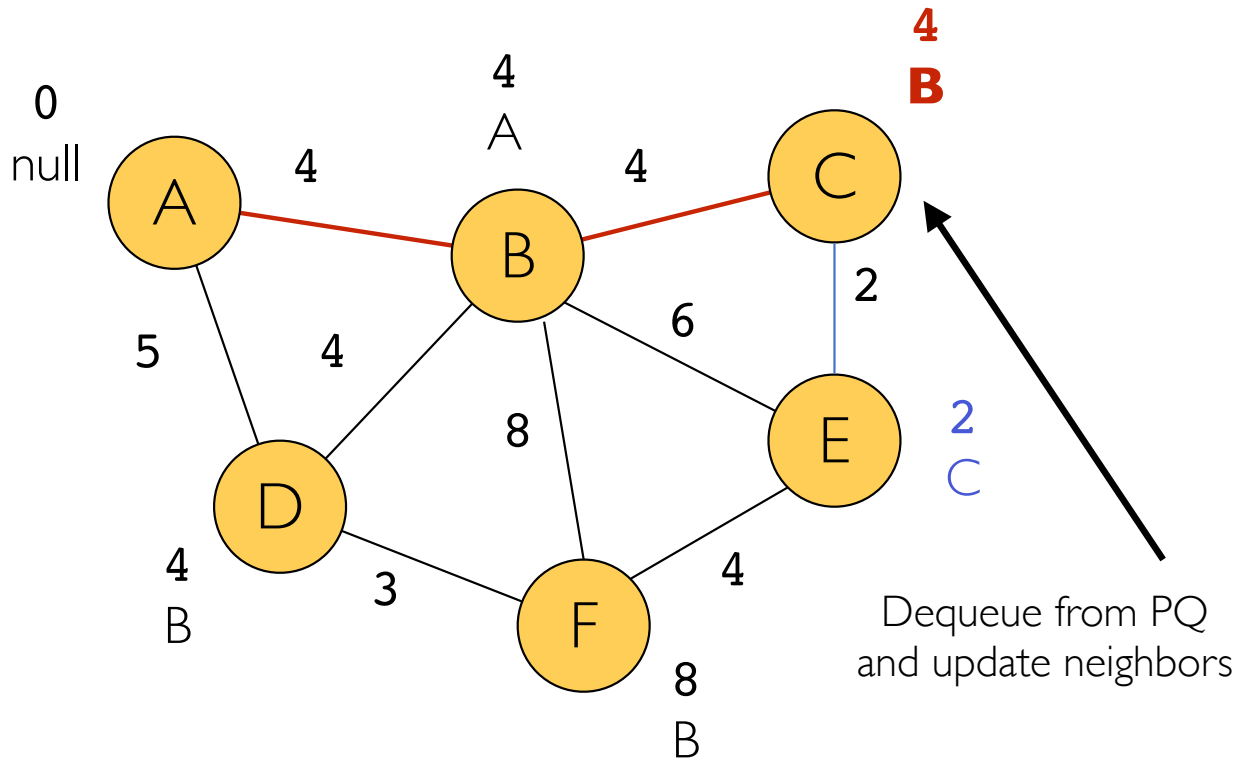
PQ = [(0,A),(∞,B),(∞,C),(∞,D),(∞,E),(∞,F)]


0
null
Dequeue from PQ
and update neighbors
A
B
C
D
E
F
4
A
∞
null
4
4
5
4
6
2
8
∞
null
5
A
3
4
∞
null

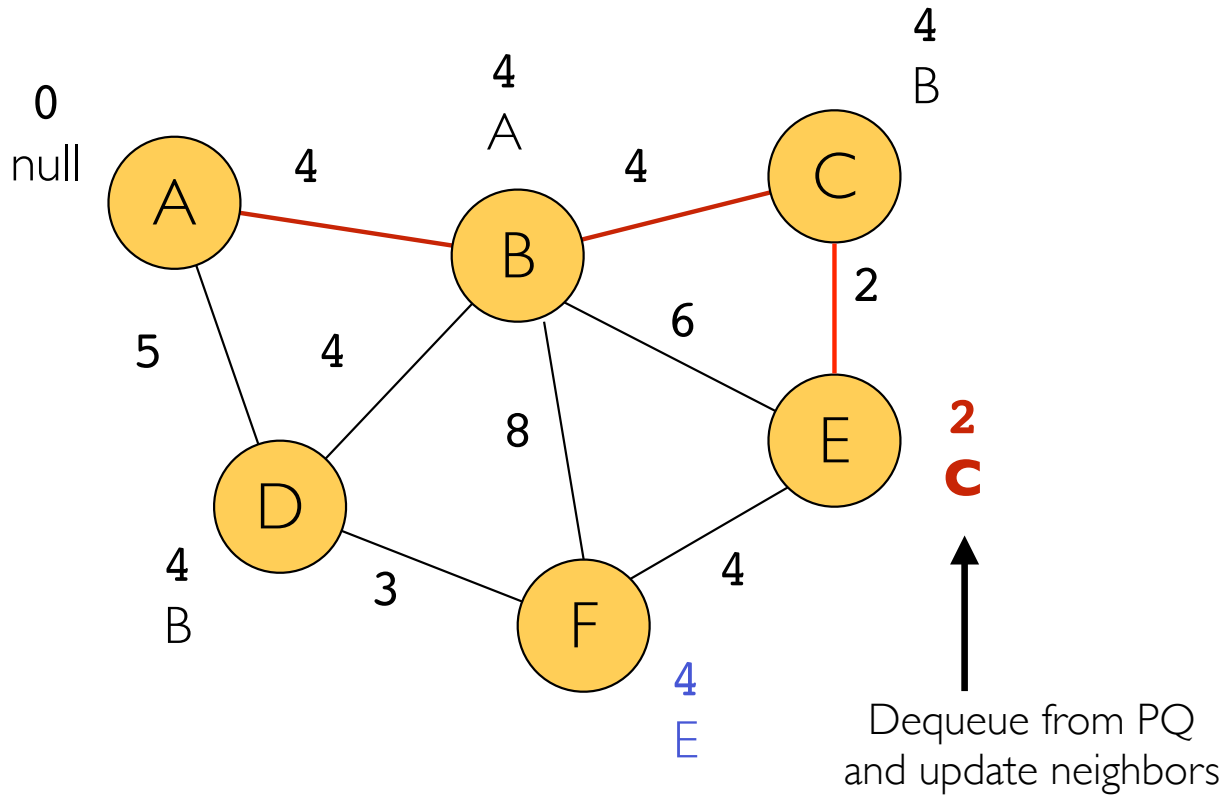
PQ = [(4,B),(5,D),(∞,C),(∞,E),(∞,F)]


0
null
A
4
A
B
4
4
C
4
B
2
6
5
4
Dequeue from PQ
and update neighbors
8
D
E
6
B
4
B
3
F
4
8
B

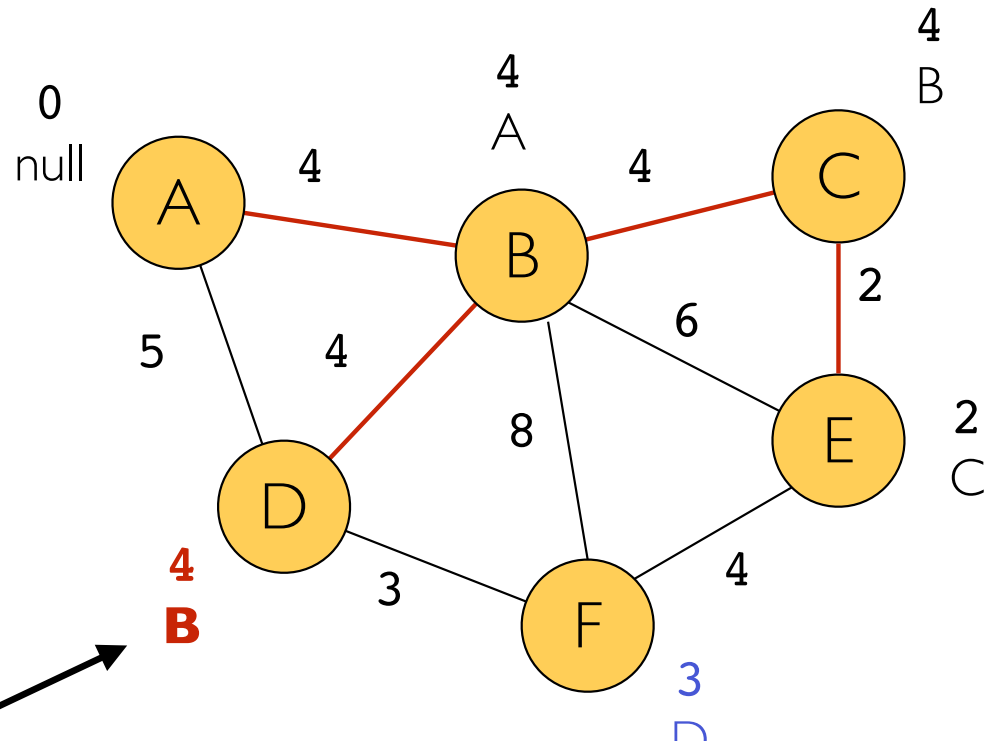
```
PQ = [(4,C),(4,D),(6,E),(8,F)]
```


0
null
A
4
A
B
4
4
B
C
2
5
4
6
8
D
E
2
C
4
B
3
F
4
8
B
Dequeue from PQ
and update neighbors

```
PQ = [(2,E),(4,D),(8,F)]
```

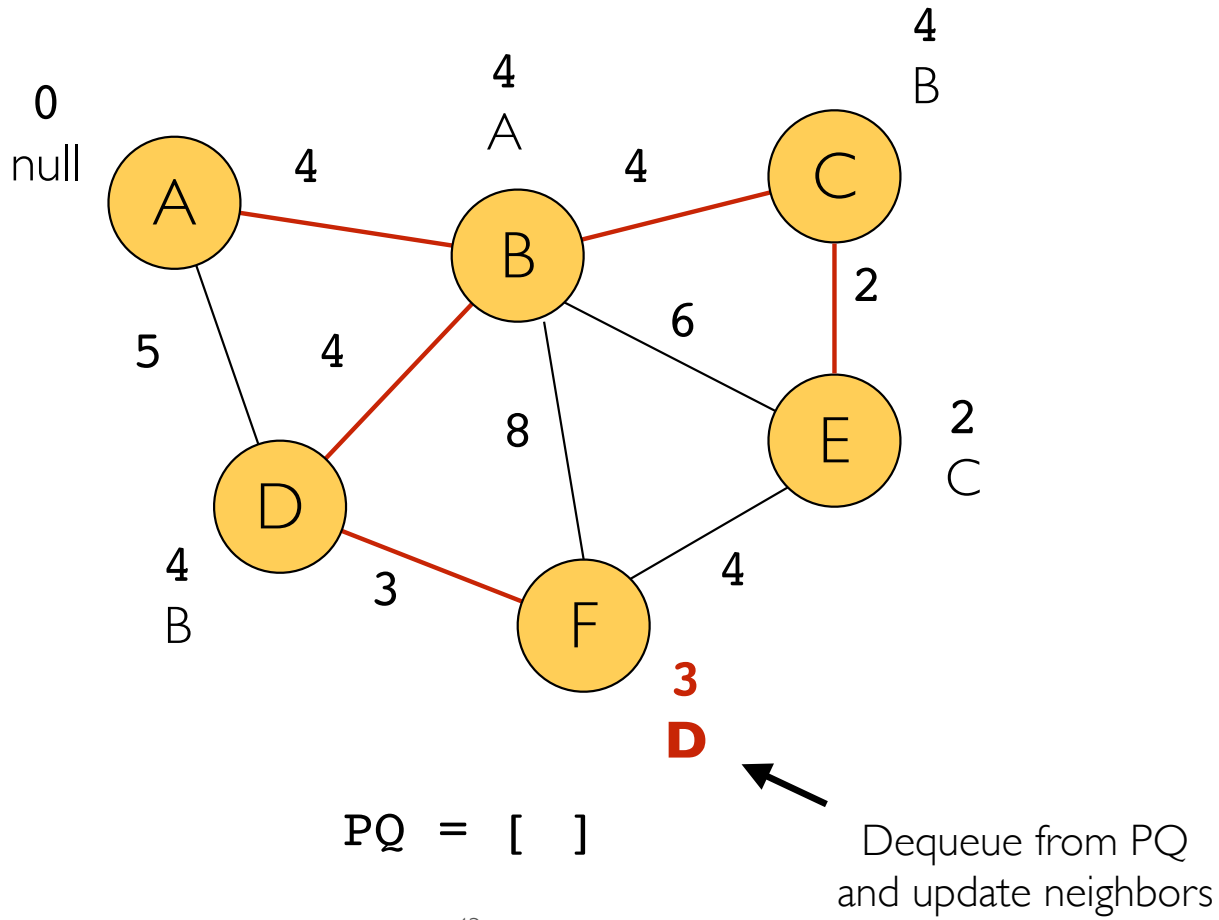

0
null
A
4
A
B
4
4
B
C
2
5
4
6
8
D
E
2
C
4
B
3
F
4
4
E
Dequeue from PQ
and update neighbors

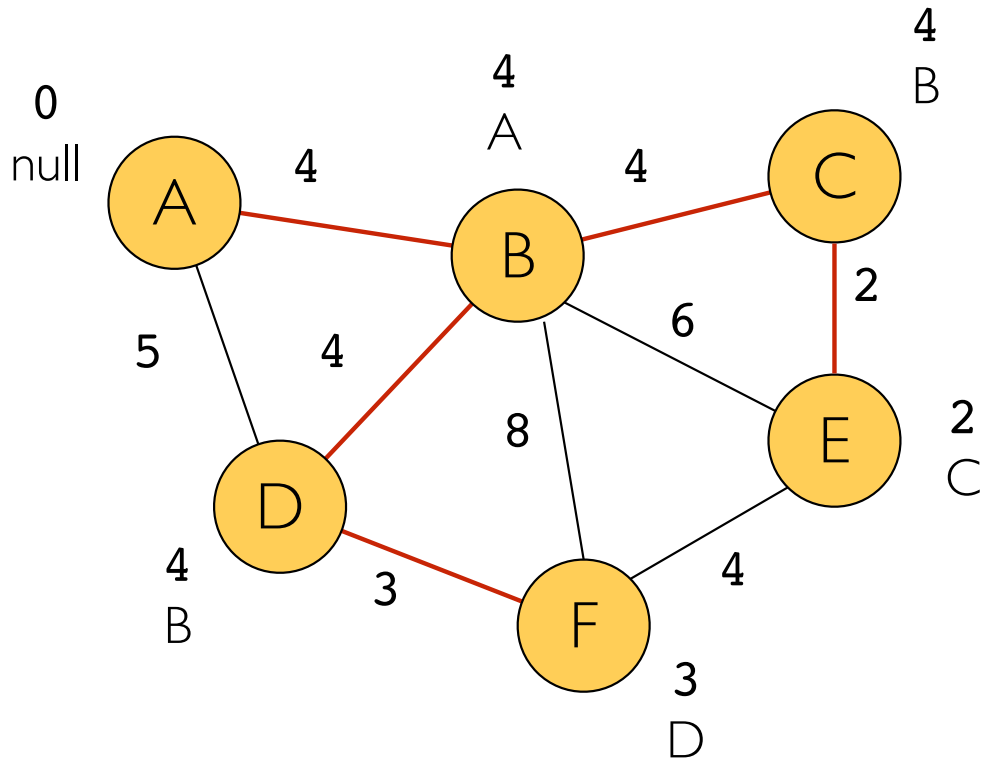
```
PQ = [(4,D),(4,F)]
```



Dequeue from PQ
and update neighbors

```
PQ = [(3,F)]
```


0
null
A
4
A
B
4
B
C
4
4
4
5
4
6
2
8
D
E
2
C
4
B
3
F
4
3
D
PQ = [ ]
Dequeue from PQ
and update neighbors


0
null
A
4
A
B
4
4
B
C
5
4
6
2
8
D
E
2
C
4
B
3
F
4
3
D

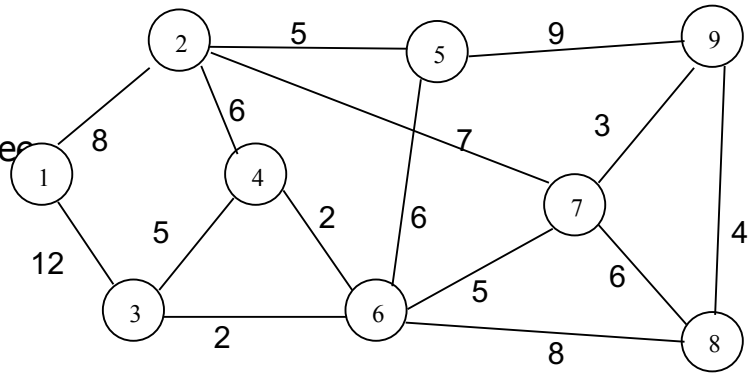
Proof of Correctness

- Let T be current tree at iteration step
- Claim: T is promising set
  - True initially, as $T = \varnothing$
  - Define B = vertices in T: t any stage, T is a tree over a subset of nodes $B \subset V$
  - Let v be min vertex in PQ; then {v.parent, v} leaves B
    - v has parent in B, and v is still in PQ, so it is not in B
    - If v in PQ, v.dist is set by min weight edge leaving B
  - Thus, {v.pred, v} is min-weight edge leaving B
  - Hence, $T \cup \left\{ \{v.pred, v\} \right\}$ is promising set after remove min extracts v

Runtime Analysis

- Initializing nodes with distance and previous pointers is O(|V|); putting nodes in PQ is O(|V|)
- While loop runs |V| times
  - removing vertex from PQ is O(log|V|)
  - So O(|V|log|V|)
- For loop (in while loop) runs |E| times in total
  - Determining whether v' is in PQ: O(1) if we build index into PQ (need to find location; not easy!)
  - Decreasing vertex's key in the PQ is log|V| (binary heap), or amortized to $O(1)$ if we use Fibonacci or rank-pairing heaps
  - So O(|E|) in complex data or O(|E| log|V|)
- Overall runtime
  - O(|V| + |V|log|V| + |E|)
    - = O(|E| + |V|log|V|) best

Example



Priority Q

(1,0)

(2,M) (3,M)

(4,M) (5,M) (6,M) (7,M)

(8,M) (9,M)

1) Scan 1, reduce key of 2, 3; remove 1

(2,8)

(9,M) (3,12)

(4,M) (5,M) (6,M) (7,M)

(8,M)

Minimum Spanning Tree

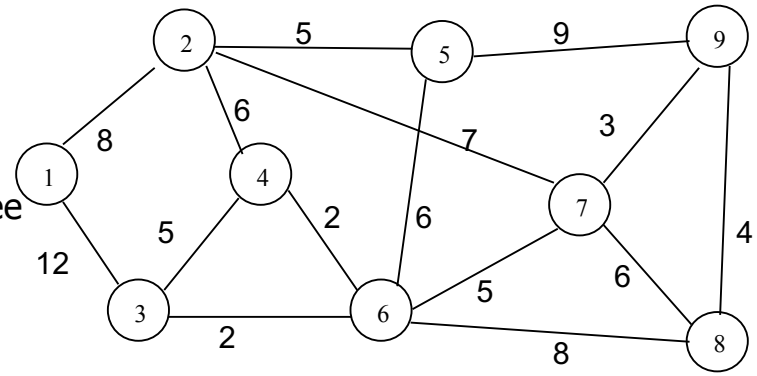
Empty

1

Sorted edges

{3,6}, {4,6},{7,9},{8,9},{3,4},{6,7},
{2,5},{2,4}{5,6},{7,8}, {2,7},{1,2},
{6,8},{5,9},{1,3}

Example



2) Scan 2, reduce key of 4,5,7. Remove 2.

Priority Q

(5,5)

(4,6) (7,7)

(9,M) (8,M) (6,M) (3,12)

Minimum Spanning Tree

1—2

3) Scan 5, reduce key of 6, 9; remove 5

(4,6)

(9,9) (6,6)

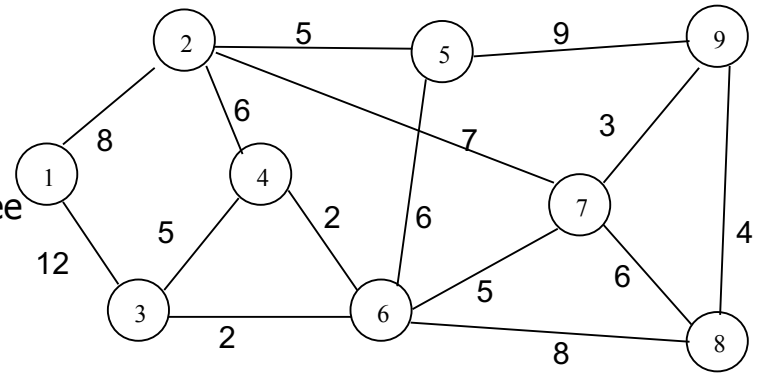
(3,12) (8,M) (7,7)

1—2—5

Sorted edges

{3,6}, {4,6},{7,9},{8,9},{3,4},{6,7},
{2,5},{2,4}{5,6},{7,8}, {2,7},{1,2},
{6,8},{5,9},{1,3}

Example



4) Remove 4, reduce key of 3,6.

Priority Q

(6,2)

(3,5) (7,7)

(9,9) (8,M)

5) Remove 6, reduce key of 3, 7, 8;

(3,2)

(9,9) (7,5)

(8,8)

—

Minimum Spanning Tree

1—2—5

|

4

1—2—5

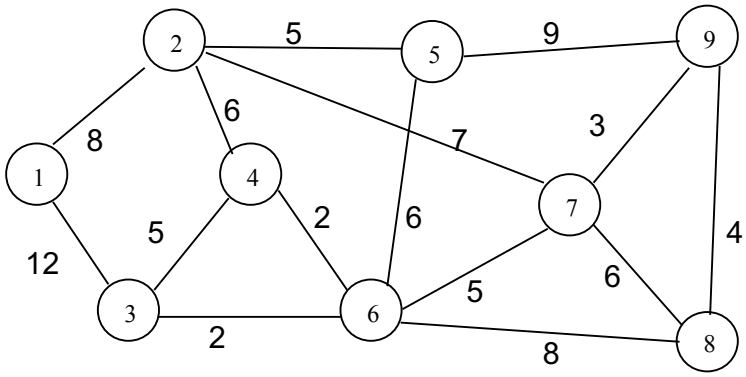
|

4 — 6

Sorted edges

{3,6}, {4,6},{7,9},{8,9},{3,4},{6,7},
{2,5},{2,4}{5,6},{7,8}, {2,7},{1,2},
{6,8},{5,9},{1,3}

Example



6) Remove 3.

Priority Q

(7,5)

(9,9) (8,8)

5) Remove 7, reduce key of 8,9;

(9,3)

(8,6)

6) Remove 9, reduce key of 8

(8,4)

7) Remove 8. DONE

Minimum Spanning Tree

1—2—5

|

4 — 6 — 3

1—2—5

| /— 7

4 — 6 — 3

1—2—5

| /— 7— 9— 8

4 — 6 — 3

Sorted edges

{3,6}, {4,6},{7,9},{8,9},{3,4},{6,7}, {2,5},{2,4}{5,6},{7,8}, {2,7},{1,2}, {6,8},{5,9},{1,3}

Summary

- Initializing nodes with distance and previous pointers is O(|V|); putting nodes in PQ is O(|V|)
- While loop runs |V| times
  - removing vertex from PQ is O(log|V|)
  - So O(|V|log|V|)
- For loop (in while loop) runs |E| times in total
  - Determining whether v' is in PQ: O(1) if we build index into PQ (need to find location; not easy!)
  - Decreasing vertex's key in the PQ is log|V| (binary heap), or amortized to $O(1)$ if we use Fibonacci or rank-pairing heaps
  - So O(|E|) in complex data or O(|E| log|V|)
- Overall runtime
  - O(|V| + |V|log|V| + |E|)
    - = O(|E| + |V|log|V|) best

Interesting Facts

- Cayley's theorem: The complete graph on n nodes has n^{n-2} spanning trees
- Suppose that you change the cost of every edge in G as follows. For which is every MST in G an MST in G' (and vice versa)?
  - c'(e) = c(e) + 17.
  - c'(e) = 17 × c(e).
  - Assume c(e) >0; then, c'(e) = $\log_{17} c(e)$
  - All of the above.
- The highest weight edge in every cycle cannot be part of an MST
  - Reverse algorithm: throw out highest weight edges in cycles

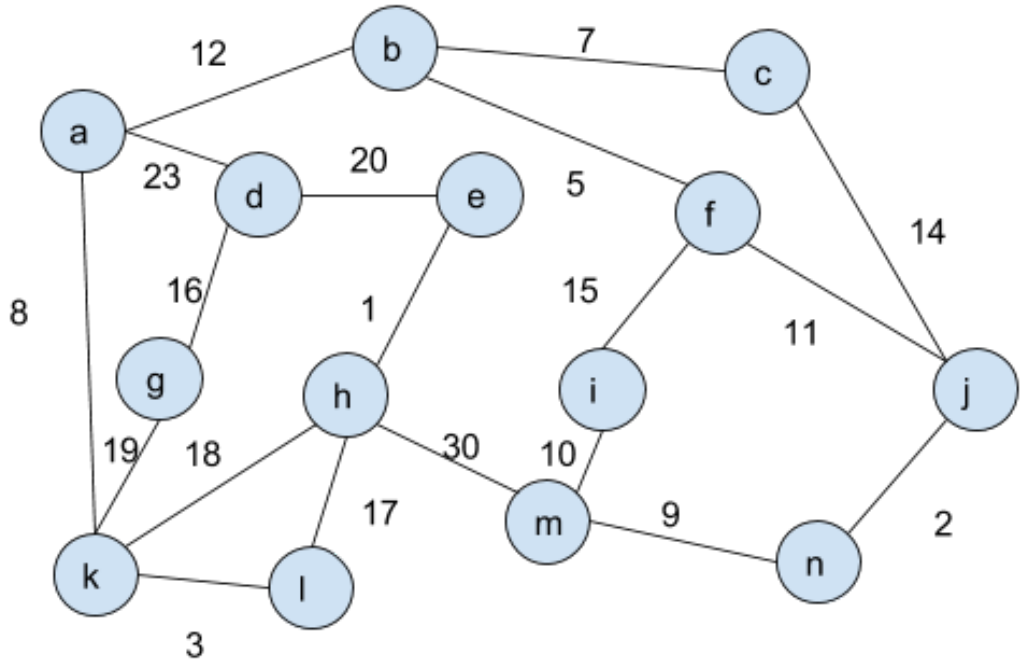
Applications

- MST is fundamental problem with diverse applications.
  - Cluster analysis.
  - Max bottleneck paths.
  - Real-time face verification.
  - LDPC codes for error correction.
  - Image registration with Renyi entropy.
  - Find road networks in satellite and aerial imagery.
  - Model locality of particle interactions in turbulent fluid flows.
  - Reducing data storage in sequencing amino acids in a protein.
  - Autoconfig protocol for Ethernet bridging to avoid cycles in a network.
  - Approximation algorithms for NP-hard problems (e.g., TSP, Steiner tree).
  - Network design (communication, electrical, hydraulic, computer, road).

Borůvka's Algorithm

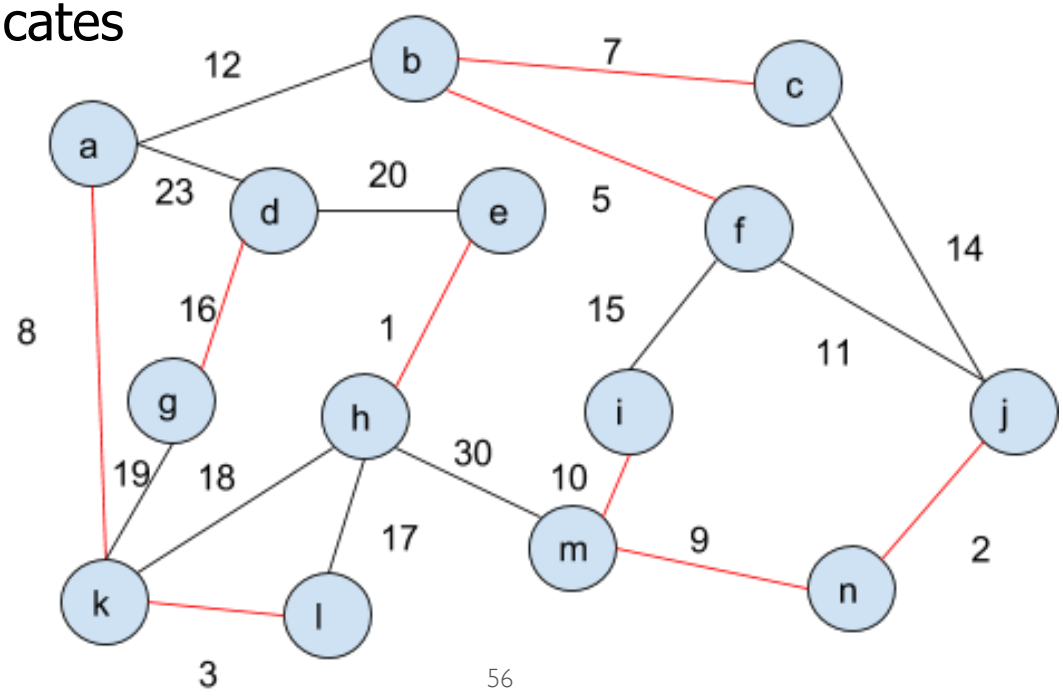
- Earliest MST algorithm: 1926. Application: design of power grid
- For **every** connected set in a forest,
  - Select smallest weight edge that leaves connected set
  - Add it to the MST
  - But don't add it twice if same edge selected by two connected sets
- In principle, merges at least half of the trees at each time: O(log(n)) iterations
    - Easy to parallelize
- Each pass is O(#E)

Example

- Start with every vertex in a separate connected set, partial MST empty
- 

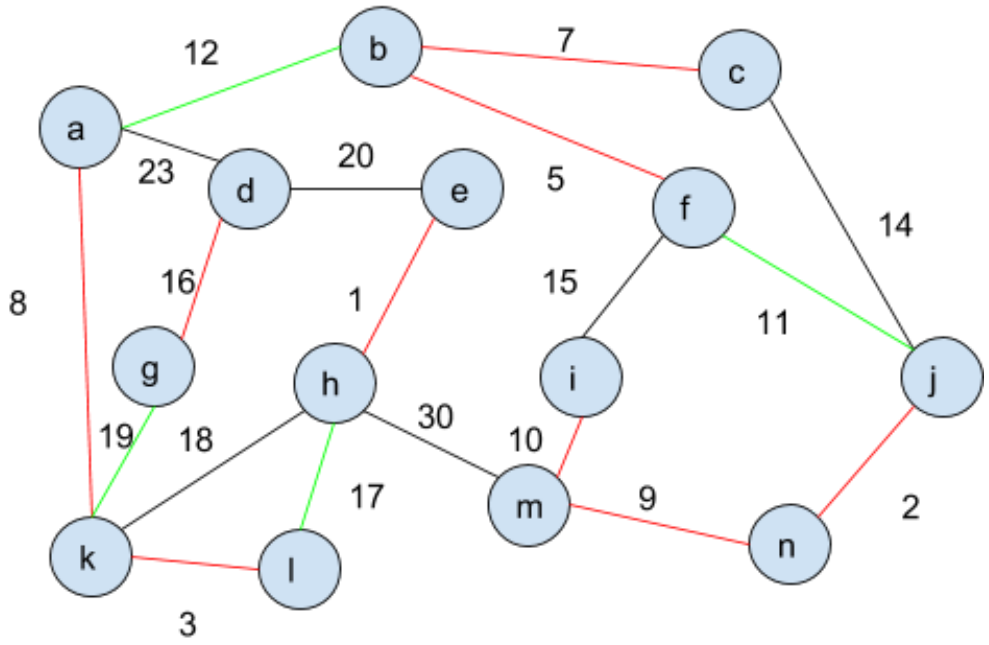
Example

- Find smallest weight edge connected to each set
  - Add all these edges to the partial MST
  - Avoid duplicates



Example

- Find smallest weight edge connected to each connected set
  - Add all these edges to the partial MST: Done!

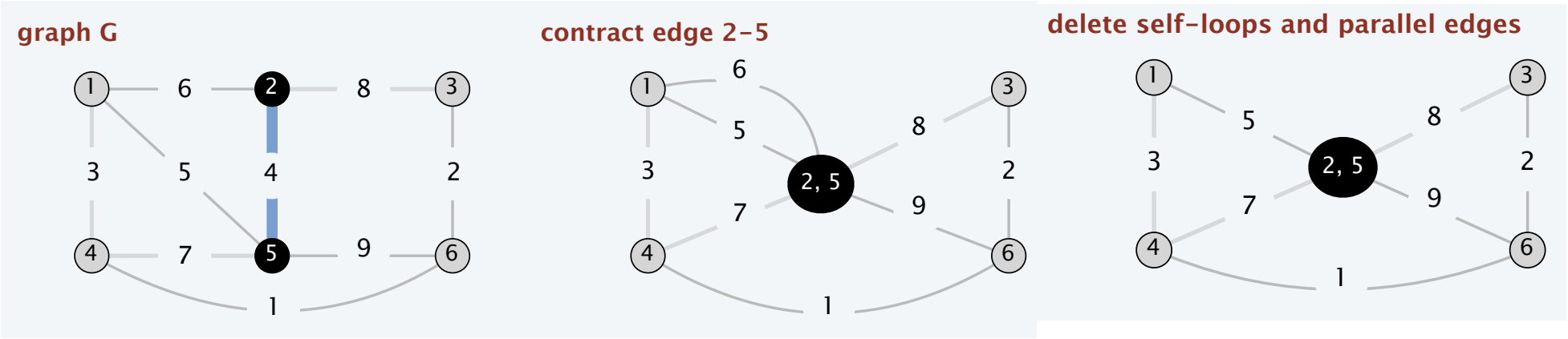


Borůvka's Algorithm

- Why does this work?
  - Can add edges sequentially and maintain promising set structure
  - Let v $\in$ V be any vertex in G. The minimum spanning tree for G must contain an edge (v, w) that is the minimum weight edge incident on v.
- What is hard to compute?
  - A bit of work to do: For each pass, need to avoid edges that form loops
    - Can do in O($\log^*(\#V)$) with disjoint set data structure, or other tricks (graph contraction)
- Total complexity O($\#E \log(\#V)$)

Graph Contraction

- After each phase, contract each connected component to a single super-vertex
  - Delete self-loops and parallel edges (keeping only cheapest one)
- Borůvka iteration becomes: take cheapest edge incident to each super-vertex



Graph Contraction: O(#E)

- Step 1: Let F be current edges in partial MST
  - Compute the n′ connected components in (V, F) as super-vertices V′
  - Let id[u] = super-vertex index of vertex u
  - For each e = {u,v} $\in$ E, add an edge e′ = {id[u],id[v]} to E′ if and only if id[u] differs from id[v], and set w(e′) = w(e)
- Step 2: Remove parallel edges
  - For each set {e′: e′ = {j,k} for j, k in V′}, keep only the edge with minimum weight
  - Can do this efficiently using a lexicographic sort (not comparison-based)

Further Work

- Hybrid: Combine Borůvka's and Prim-Jarnik, with n = #V, m = #E

  Run Borůvka's for r phases: reduces super-vertices to $\leq \frac{n}{2^r}$, O(mr)

  Run Prim-Jarnik on contracted graph G': $O(m + \frac{n}{2^r} \log(\frac{n}{2^r}))$

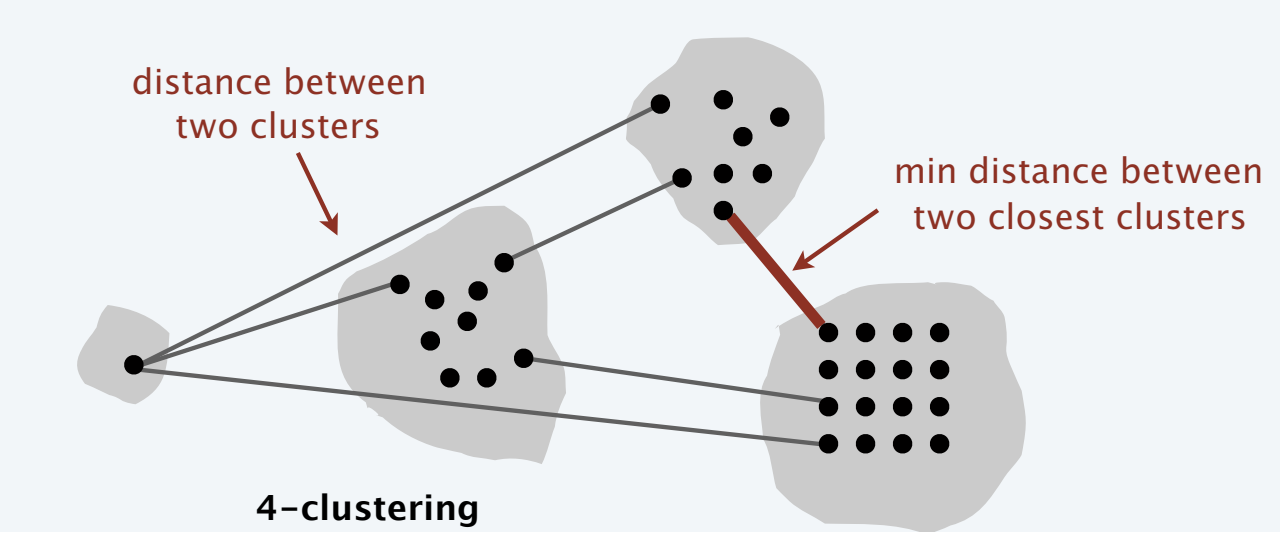
- Choose r to balance complexity of both: r = $\log_2(\log_2(n))$
- Total complexity:

  O(m $\log_2(\log_2(n))$ + $m + \frac{n}{\log_2(n)} \log(\frac{n}{\log_2(n)})$) = O(m $\log_2(\log_2(n))$)

- Much work trying to get it down to O(m)

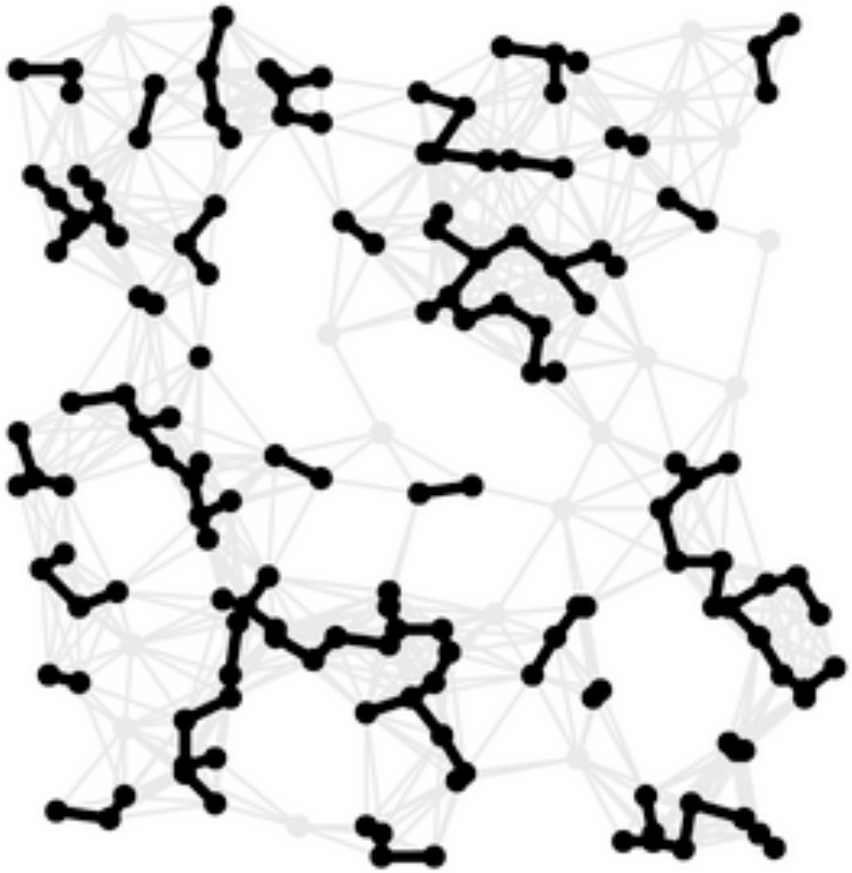
Application: Clustering

- Given a weighted graph G = (V,E), where weight is distance, want to create k clusters such that we maximize the minimum distance between any pair of vertices in clusters



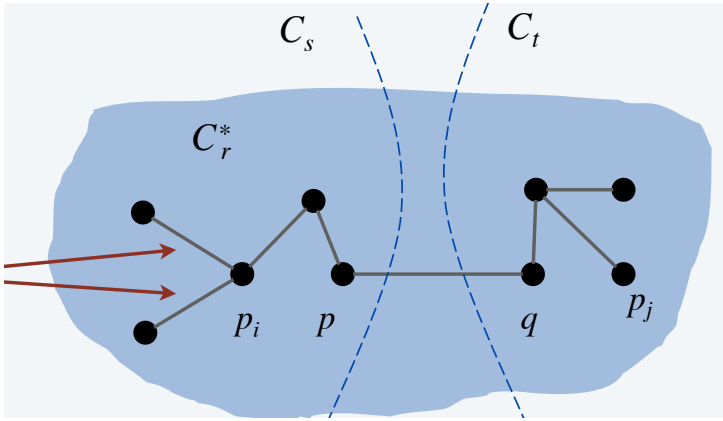
Solution

- Approach 1: Run Kruskal's algorithm until there are only k connected components left
- Approach 2: Run any MST algorithm, and delete the longest k-1 edges in MST



Why does this work?

- Theorem: Let C* = $\{C_1^*, \ldots, C_k^*\}$ denote the clustering formed by deleting the k – 1 longest edges of an MST. Then, C* is a k-clustering of max spacing
  - Let d^* be min distance in C*. Consider any other clustering C = $\{C_1, \ldots, C_k\}$
  - Let v, w be in the same cluster in C* (e.g. C_1^*) but not in C
  - Then, there is an edge $\{p, q\}$ on v-w path in C_1^*, such that p in C_j, q in C_k, and j differs from k
  - $w(p, q) \leq d^*$ because it was added by Kruskal, so min distance in C $\leq d^*$

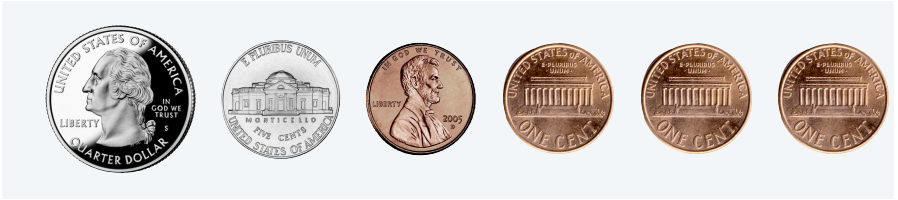


MHT Algorithms are Greedy Algorithms

- Greedy algorithm: an algorithm that builds a solution adding an element a a time that is the locally optimal choice at that time
  - Uses a simple rule e.g. MST adds edge with minimum weight across a cut
  - No backtracking (!!!)
  - Greedy algorithms are not always optimal
  - Special classes of problems can be solved to optimality by greedy algorithms

Example: Coin Changing

- Given U. S. currency denominations { 1, 5, 10, 25, 100 }, devise a method to pay amount using fewest coins



- Greedy Algorithm
  - At each iteration, add largest value coin that does not exceed amount
  - e.g. for $2.89,

- Is it optimal? Will this yield fewest coins for any amount?

Greedy Coin Changing in US

- For US denominations, optimal solution must satisfy:
  - Number of pennies ≤ 4 (if not, can reduce coins with a nickel)
  - Number of nickels ≤ 1 (otherwise, use dime)
  - Number of quarters ≤ 3 (otherwise, use Liberty $)
  - Number of nickels + dimes ≤ 2 (otherwise, use quarters)

Greedy Coin Changing Optimal in US

- Prove by induction: Let c_k = value of coin k, x = amount to be paid
  - Assume greedy is optimal for $x < c_k$
  - If $c_k \leq x < c_{k+1}$, optimal solution must include at least one of coin k

Otherwise, must match c_k with 1 lesser value coins, and it is not possible with US coins)

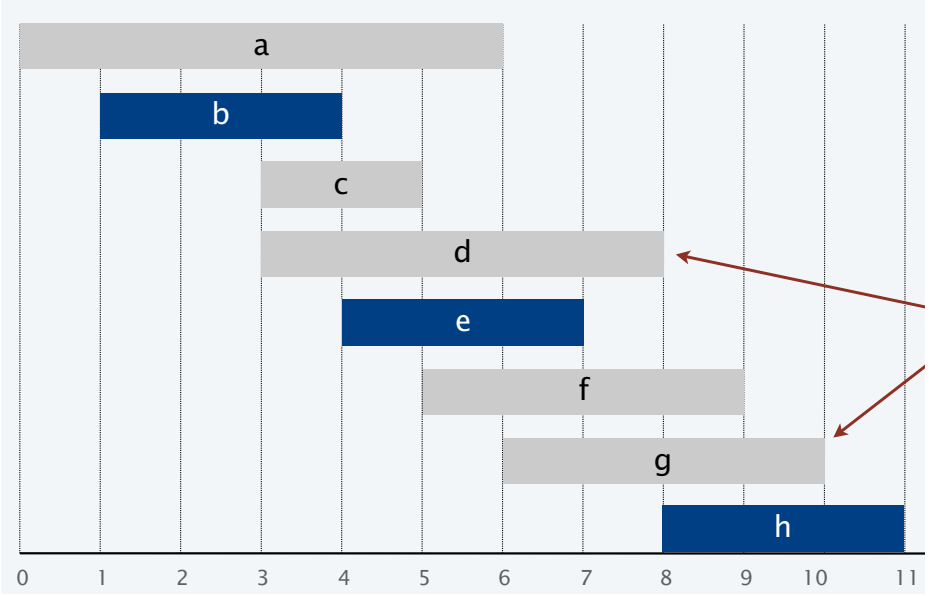
  - Thus, optimal to use 1 of coin k, and then solve for x-c_k

k	c_k	all optimal solutions must satisfy	max value of coin denominations $c_1, c_2, \ldots, c_{k-1}$ in any optimal solution
1	1	$P \leq 4$	–
2	5	$N \leq 1$	4
3	10	$N + D \leq 2$	4 + 5 = 9
4	25	$Q \leq 3$	20 + 4 = 24
5	100	*no limit*	75 + 24 = 99

- NOTE: not true for other denominations (e.g. stamps)

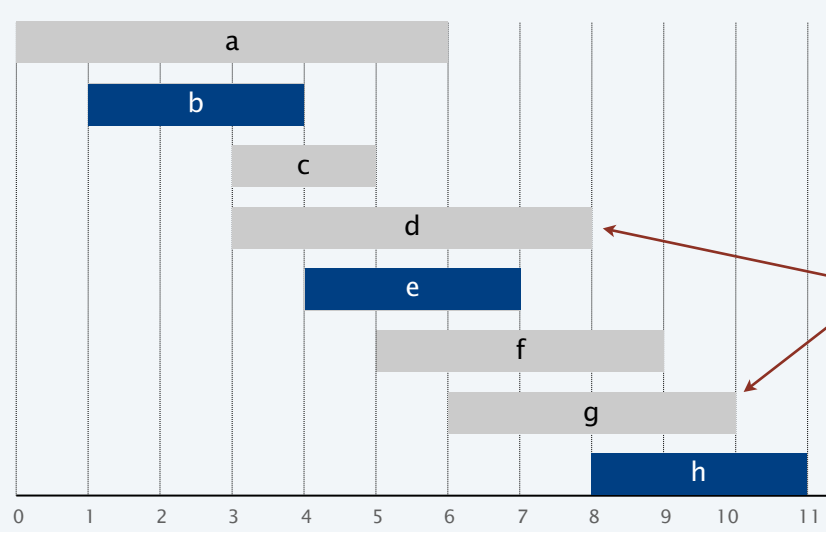
Interval Scheduling

- Jobs {1, 2, . . . N} occur at fixed times, with start times s(j), f(j) for job j
- Single processor
- 2 jobs are compatible if they don't overlap
- Goal: Maximize number of jobs scheduled that are compatible



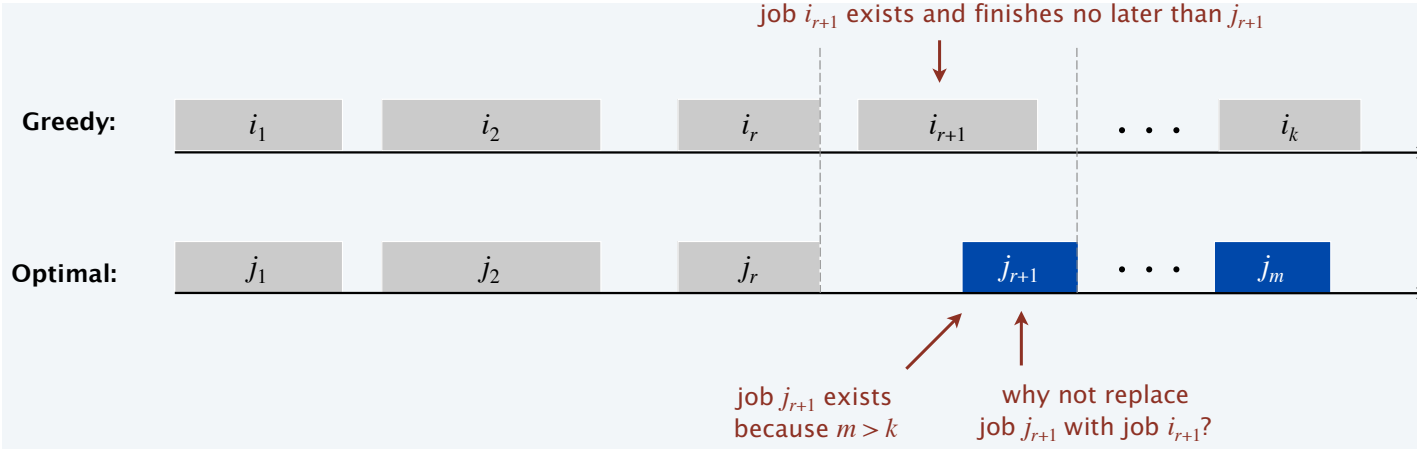
Interval Scheduling

- Quiz: Which is best? Sequence jobs by using an index, then add to schedule if compatible?
  - Earliest start time? (a,b,c,d,e,f,g,h)
  - Earliest finish time? (b,c,a,e,d,f,g,h)
  - Shortest interval? (c,b,e,h,f,g,d,a)
  - Least overlap? (h,b,c,g,a,e,f,d)



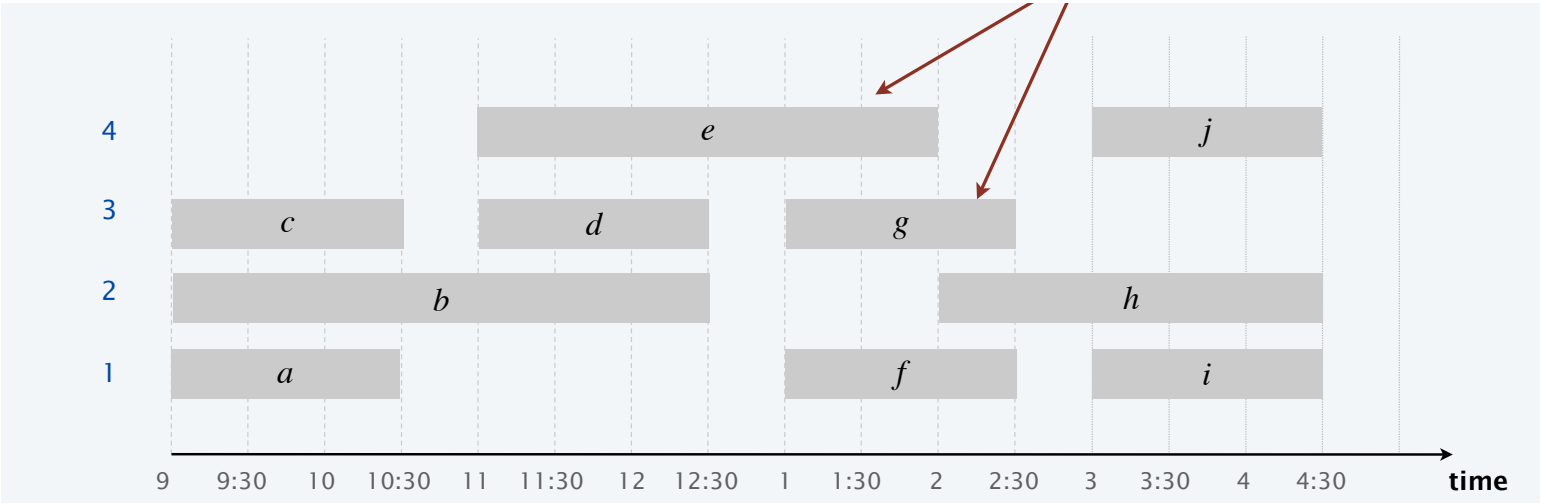
Interval Scheduling: Earliest finish time optimal

- Prove by contradiction
  - Let $i_1, i_2, \ldots, i_k$ denote set of jobs selected by EFT, and let $j_1, j_2, \ldots, j_m$ denote set of jobs in an optimal solution
  - Let r be largest value where $i_1 = j_1, \ldots, i_r = j_r$ (schedules agree)
  - Note: i_{r+1} finishes no later than j_{r+1} (because of EFT choice)
  - So, can replace j_{r+1} with i_{r+1} and still be optimal for rest of schedule J —> contradiction
  -



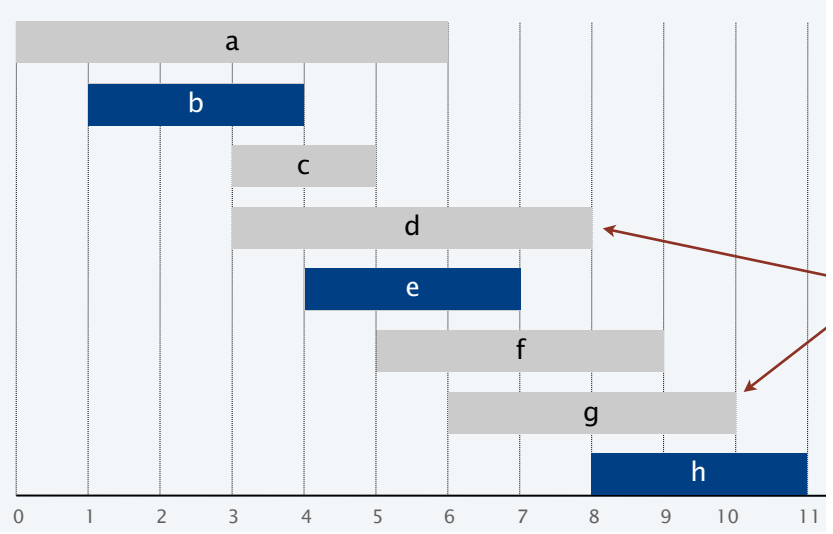
Scheduling Jobs on Parallel Processors

- Jobs {1, 2, . . . N} occur at fixed times, with start times s(j), f(j) for job j
- What is minimum number of parallel processors so we can schedule all jobs without conflict?
- The registrar problem: jobs are classes with start times, end times, and processors are classrooms (a bit more complicated if you add class sizes)



Parallel Job Scheduling

- Quiz: Which is best? Sequence jobs by using an index, then assign to first available processor?
  - Earliest start time? (a,b,c,d,e,f,g,h)
  - Earliest finish time? (b,c,a,e,d,f,g,h)
  - Shortest interval? (c,b,e,h,f,g,d,a)
  - Least overlap? (h,b,c,g,a,e,f,d)
  - ????

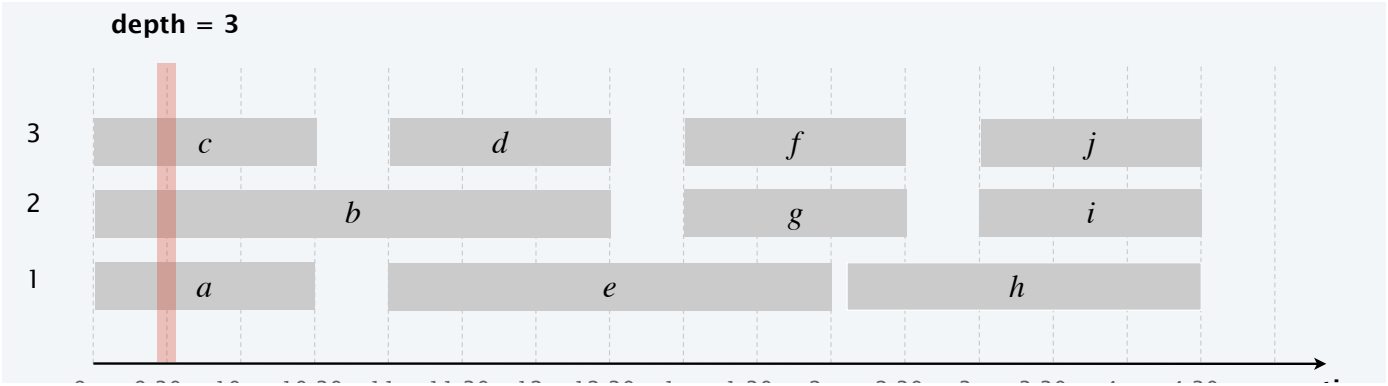


Optimal Algorithm

- Sequence by earliest start time: $s(1) \leq s(2) \leq \cdots \leq s(n)$
  - Let d = number of allocated classrooms
  - For j = 1 to n:
    - If processor available at s(j), assign s(j) to processor
    - Else increase d = d+1, assign s(j) to new processor
- Alternative, sequence by earliest finishing time…would also work
- O(n log(n)) —> need to sort

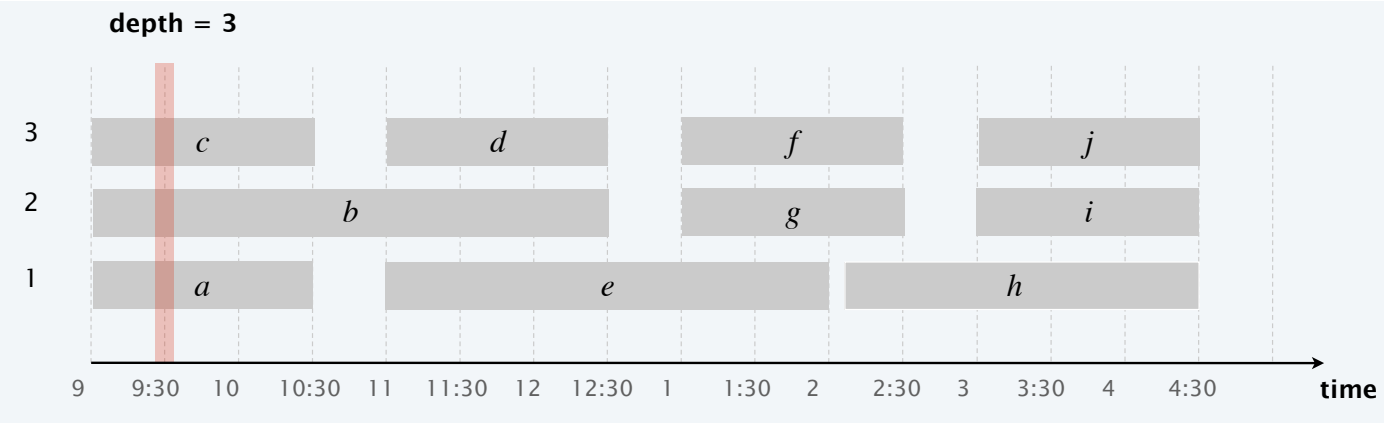
Concept: Depth of intervals

- Def. The depth of a set of open intervals is the maximum number of intervals that contain any given point.
- Key observation. Number of processors needed ≥ depth.
- Q. Does minimum number of processors needed always equal depth?
  - Yes! Moreover, earliest-start-time-first algorithm finds a schedule whose number of processors equals the depth



Optimality

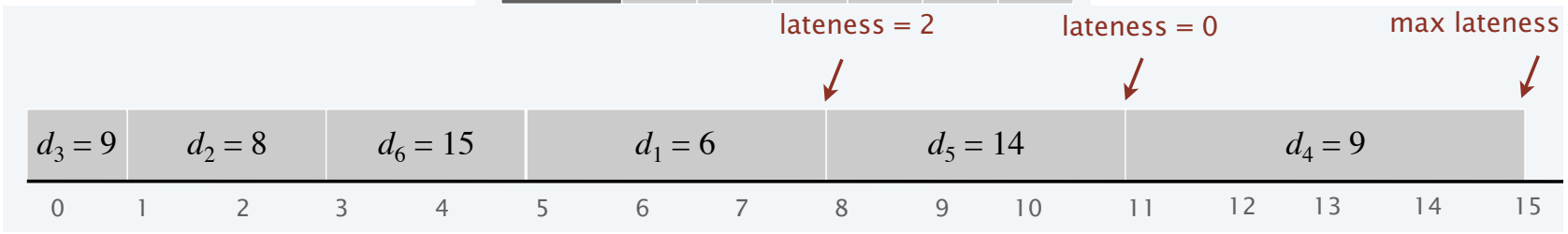
- If you must add a processor to increase d in earliest-start-time algorithm for job j at s(j), then depth is at least d+1 at time $s(j) + \epsilon$
- Key observation. Number of processors needed ≥ depth
- Thus, EST algorithm achieves lower bound on number of processors, hence it is at least as good as any other algorithm —> it is optimal



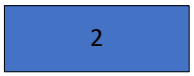
Scheduling to minimize lateness

- Jobs {1, 2, . . . N}, with required processing times t_j and deadlines d_j
- If job j starts at time s, it finishes at $f_j = s + t_j$
- Lateness measure $\ell_j = \max\{0, f_j - d_j\}$
- Goal: schedule all jobs to minimize maximum lateness $L = \max_{j=1,\ldots,N} \ell_j$

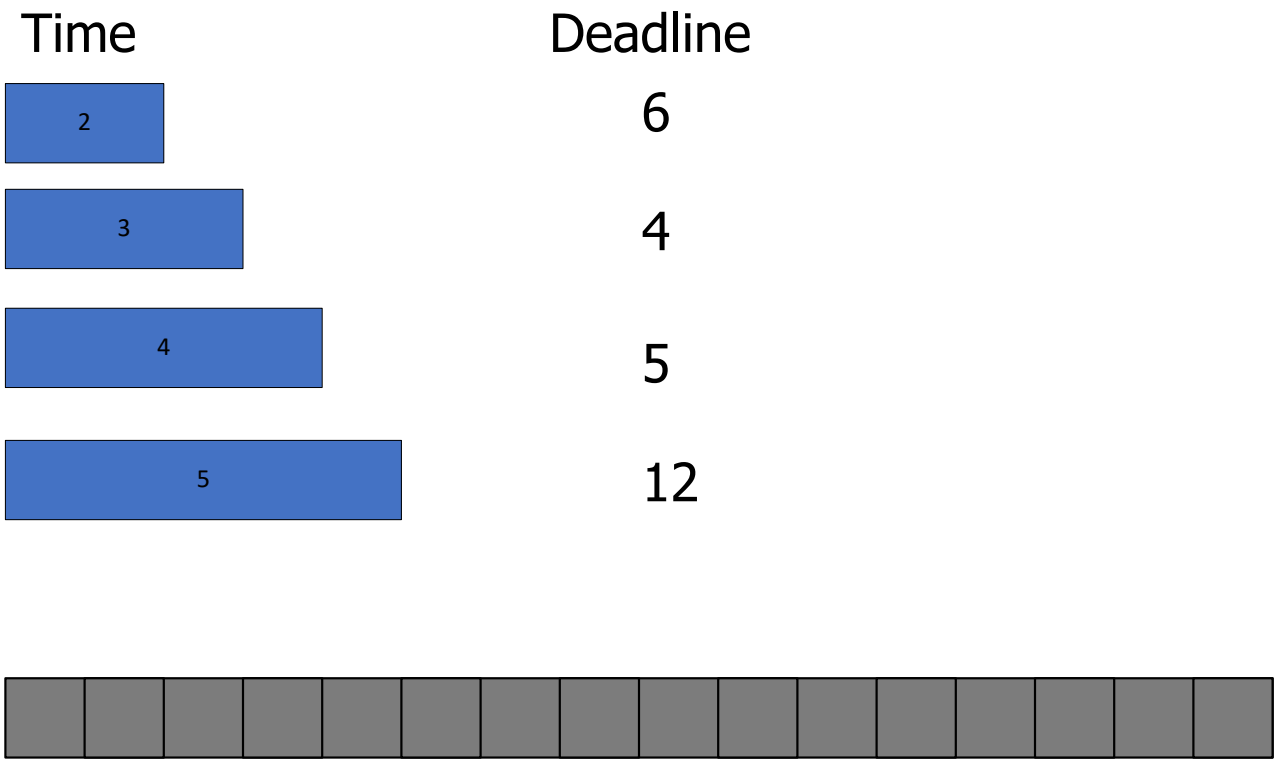
	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



Example

Time	Deadline
	2
	4
	Lateness 1
	Lateness 3

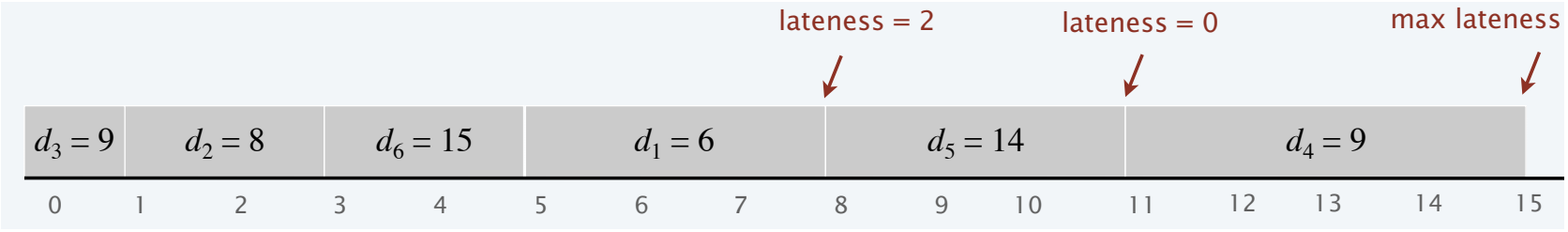
Determine the minimum lateness



Deadline Scheduling

- Quiz: Which is best? Sequence jobs by using an index?
  - Shortest processing time?
  - Earliest deadline first?
  - Increasing order of slack $d_j - t_j$?
  - Something else?

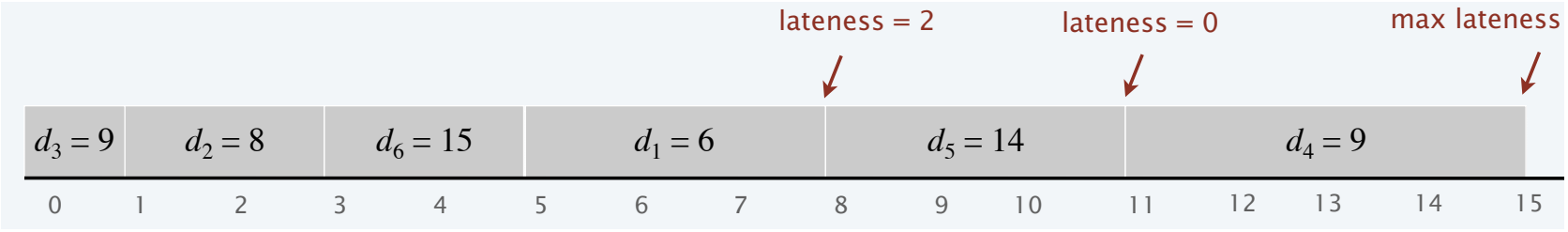
	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



Greedy algorithm: Earliest deadline first

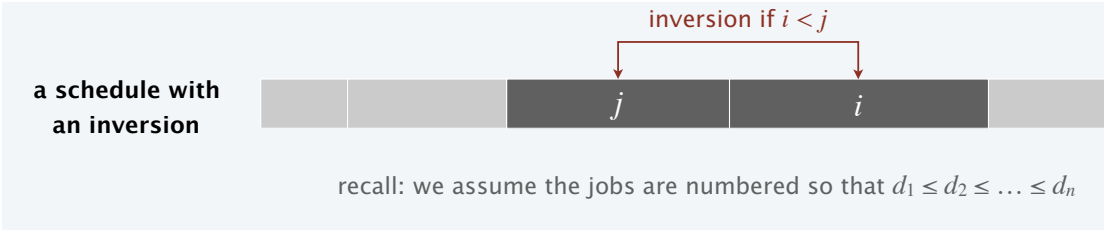
- Sort jobs by deadline, so $d_1 \leq d_2 \leq \cdots \leq d_N$
- Start with t = 0
- For j=1 to N:
  - Schedule job j in [t, t+d_j]
  - Increase t = t+d_j

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



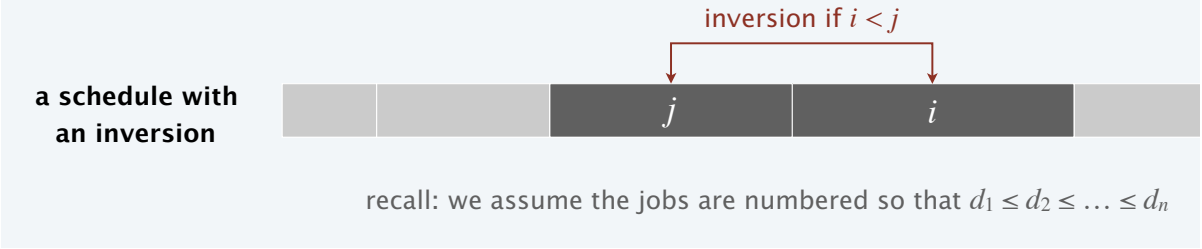
Observation: No idle time is optimal

- There exist an optimal schedule with no idle time (by contradiction, easy to perturb to eliminate idle time
  - Earliest deadline schedule has no idle time
- Definition: a schedule has an inversion if, for a particular pair j < k, job k is scheduled before job j
  - Note: if an inversion exists, then there must exist a pair of adjacent jobs in a schedule with inversion



  - Earliest deadline schedule has no inversions

Minimizing Lateness

- Given a schedule S with inversions, exchanging the order of an adjacent, inverted pair j,k does not increase maximum lateness
- Proof: Let ℓ_m be the lateness of job m before the exchange, and ℓ'_m the lateness after the exchange
  - $\ell'_m = \ell_m$ for all $m \neq j, k$
  - $\ell'_j \leq \ell_j$ because we have moved it up
  - If job k is now late, $\ell'_k = f'_k - d_k = f_j - d_k \leq f_j - d_j = \ell_j$ (earliest deadline)
  - So, max lateness is not increased
  - 

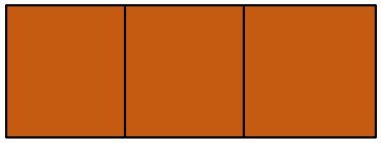
Optimality

- Earliest deadline first is optimal
- Proof:
  - Assume there is an optimal schedule that is not earliest deadline first that has the fewest number of inversions among optimal schedules
  - If number of inversions is 0, it is the EDF solution.
  - If not, there must be an adjacent inversion
  - Swapping the inversion does not increase the maximum lateness, and reduces the number of inversions, so we generate an optimal schedule which has less inversions
    - Contradiction

Optimal Caching

- Caching problem:
  - Maintain collection of items in local memory
  - Cache with capacity to store k items.
  - Sequence of m item requests $d_1, \ldots, d_m$
  - Cache hit: item in cache when requested. Cache miss: item not in cache when requested (must evict item)
- Goal: Minimize number of items that result in cache misses

Caching example



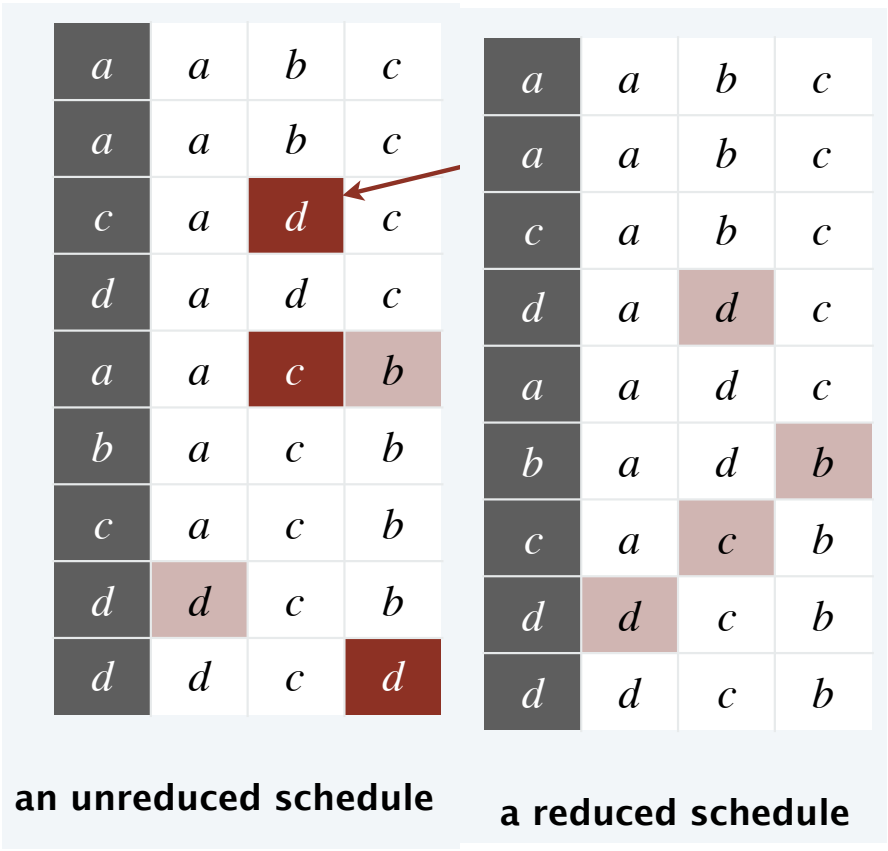
A, B, C, D, A, E, B, A, D, A, C, B, D, A

Optimal Caching

- Non-causal: if we know the sequence of requests $d_1, \ldots, d_m$, what is optimal replacement?
- Greedy algorithms
  - LIFO: evict most recent
  - FIFO: evict oldest
  - LRU: evict item with earliest recent access
  - LFU: evict least frequently requested

Optimal Caching

- Non-causal algorithm: evict item not requested until **farthest in future** (FF) sequence
- A reduced schedule is a schedule that brings an item d into the cache in step j only if there is a request for d in step j and d is not already in cache
- Given any unreduced schedule S, can transform it into a reduced schedule S′ with no more evictions



Farthest in the future algorithm

- Discard element used farthest in the future

A, B, C, A, C, D, C, B, C, A, D

Optimal Caching

- Theorem. FF is optimal eviction algorithm
  - Follows directly from the following invariant:
- There exists an optimal reduced schedule S that has the same eviction schedule as S_{FF} through the first j steps

  Pf. [by induction] Base case: j = 0 obvious.
  - Let S be reduced schedule that satisfies invariant through j steps; produce S′ that satisfies invariant after j + 1 steps.
  - Let d denote the item requested in step j + 1
  - Since S and S_{FF} agreed until j, they have the same cache contents before step j + 1.
  - **Case 1**: d is already in the cache. S′ = S satisfies invariant.
  - **Case 2**: d is not in the cache and S and SFF evict the same item. S′ = S satisfies invariant.

Optimal Caching - 2

- **Case 3**: d is not in the cache; S_{FF} evicts e; S evicts f ≠ e
  - begin construction of S′ from S by evicting e instead of f

- S′ agrees with S for first j + 1 steps; show that having item f in cache is no worse than having item e in cache
- let S′ behave the same as S until S′ is forced to take a different action (because either S evicts e; or because either e or f is requested)

Optimal Caching - 3

- **Case 3**: d is not in the cache; S_{FF} evicts e; S evicts f ≠ e
  - Let j' be the first step after j + 1 that S' must take a different action from S; let g denote the item requested in step j'

- Possibilities: a) g=e cannot happen, because in S_{FF} e was evicted first, so f is accessed earlier
- b) g = f. Since f is not in cache of S, f must evict e'.
  if e' = e, then have the same cache after S, S' so same performance
  if e' differs from e, make S' replace e' with e, and now S, S' have the same cache and performance
  S' can be transformed to reduced schedule that agrees with S for j+1 steps

Summary

- FF optimal for off-line cache optimization
  - Full sequence of requests known priori
- On-line approximation: LRU —> evict item whose last access was earliest
  - Like FF, but backwards in time…
- Strategies like FIFO and LIFO can be pretty bad

Other Scheduling Problems

1. Single server, N jobs

- Job j has processing times t(j)
- Select start time s(j) for j, when server is idle; finish time f(j) = s(j) + t(j)
- Objective: Schedule jobs to minimize total finishing time summed over jobs

$$\text{Min} \sum_{j=1}^{N} f(j)$$

- Optimal solution: Greedy algorithm
  - Sort jobs so $t(1) \leq t(2) \leq \cdots \leq t(N)$ and schedule back-to-back in that order

Other Scheduling Problems

- Proof by contradiction:

  Let P is an optimal schedule. Let p_j be job in position j in P

  Assume P is not ordered according to increasing processing time

  Find first j such that $t_{p_j} > t_{p_{j+1}}$ (must exist because of P ordering)

  Swap these in schedule to get schedule P′, and show P′ is better

  Note: finishing time of $p_1, \ldots, p_{j-1}$ and $p_{j+1}, \ldots, p_N$ is the same under P, P′

  Let T_{j-1} be finishing time p_{j-1}. Then,

  Under P, finishing times of p_j and p_{j+1} are: $T_{j-1} + t_{p_j}$, $T_{j-1} + t_{p_j} + t_{p_{j+1}}$

  Under P′, finishing times of p_j and p_{j+1} are: $T_{j-1} + t_{p_{j+1}}$, $T_{j-1} + t_{p_j} + t_{p_{j+1}}$

  Since $t_{p_j} > t_{p_{j+1}}$, P′ is better! So P cannot be optimal, contradiction.

Other Scheduling Problems

1. Single server, N jobs

- Job j has unit processing time, hard deadline d(j), Value V(j) if job completed on time
- Objective: Find sequence of jobs that will be scheduled to maximize value completed on time
- Concept: a set of jobs $\{j_1, \ldots, j_k\}$ is feasible $\Longleftrightarrow$ there is an ordering of these jobs where each job is completed before its deadline
  - If a set of jobs is feasible, scheduling them by earliest deadline first is sufficient —> simple to show

Other Scheduling Problems

- Determining the best feasible set is the hard problem
- Algorithm:
  - Initialize feasible set S as empty; sort the jobs by decreasing value V(j), into order $j_1, \ldots, j_N$
  - Initialize step k = 1
  - At step k, add job j_k to S, and see if S is still a feasible set
    - If S is not feasible, remove j_k from S
    - If k < N, increment k = k+1 and repeat
- Claim: this will be an optimal feasible set
  - Important: every job takes the same amount of time