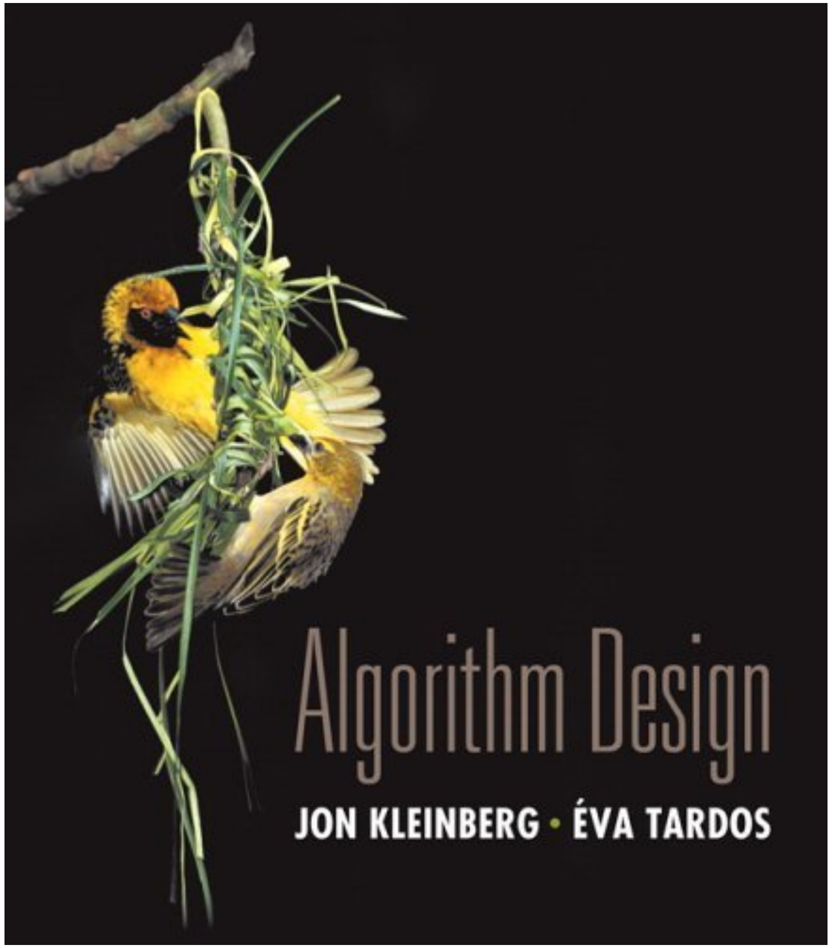


Greedy Algorithms

Greedy Algorithms

An algorithm is greedy if it builds a solution in small steps, choosing a decision at each step myopically [=locally, not considering what may happen ahead] to optimize some underlying criterion.

It is easy to design a greedy algorithm for a problem. There may be many different ways to choose the next step locally.

What is challenging is to produce an algorithm that produces
either an optimal solution,
or a solution close to the optimum.

Proving that the Greedy Solution is Optimal

Approaches to prove that the greedy solution is as good or better as any other solution:

1) prove that it stays ahead of any other algorithm
   e.g. Interval Scheduling

2) exchange argument (more general): consider any possible solution to the problem and gradually transform into the solution found by the greedy solution without hurting its quality.
   e.g. Scheduling to Minimize Lateness

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 solution must have a certain value. Then show that your algorithm always achieves this bound.

Example Problems

Interval Scheduling

Interval Partitioning

Scheduling to Minimize Lateness

Shortest Paths in a Graph (Dijkstra)

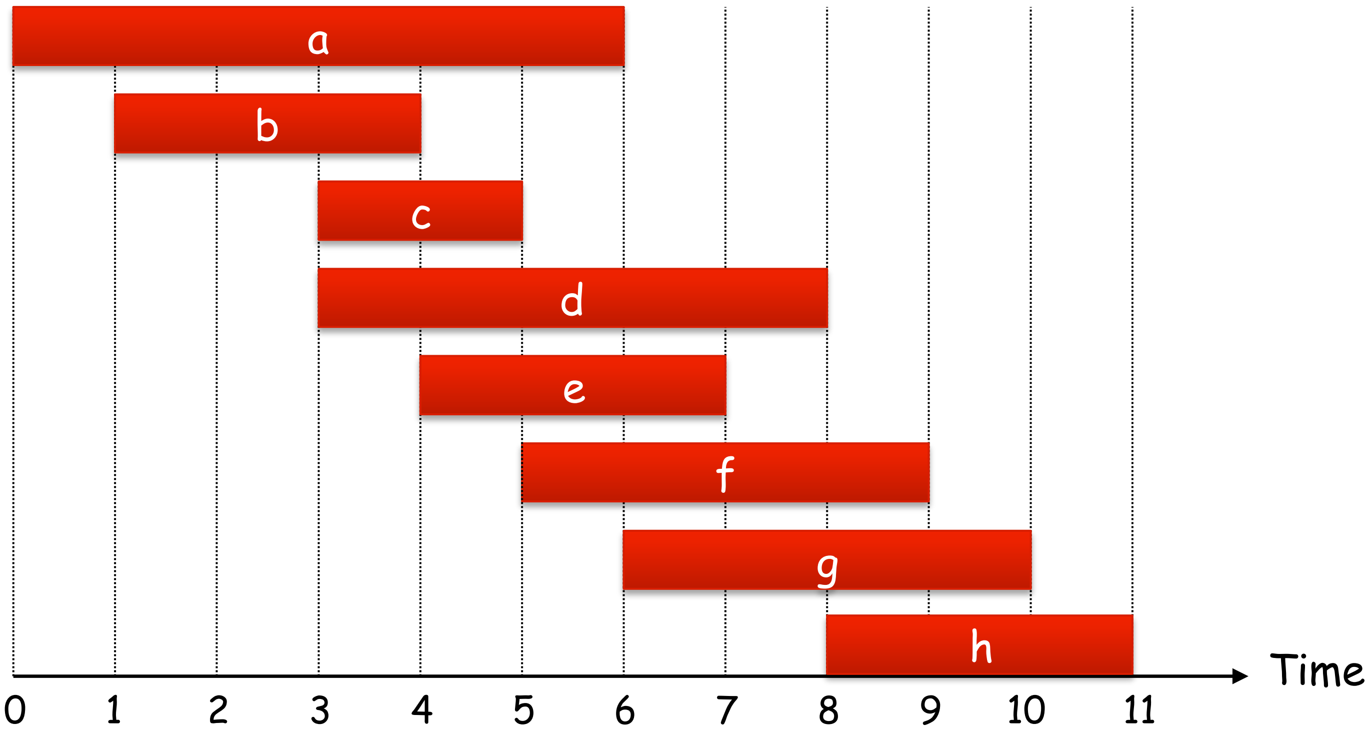
The Minimum Spanning Tree Problem
    Prim's Algorithm, Kruskal's Algorithm

Huffman Codes and Compression

Interval Scheduling

Interval scheduling.

- Job j starts at s_j and finishes at f_j.
- Two jobs compatible if they don't overlap.
- Goal: find maximum subset of mutually compatible jobs.



Interval Scheduling: Greedy Algorithms

Greedy template. Consider jobs in some order. Take each job provided it's compatible with the ones already taken.

- [Earliest start time] Consider jobs in ascending order of start time s_j.
- [Earliest finish time] Consider jobs in ascending order of finish time f_j.
- [Shortest interval] Consider jobs in ascending order of interval length $f_j - s_j$.
- [Fewest conflicts] For each job, count the number of conflicting jobs c_j. Schedule in ascending order of conflicts c_j.

Interval Scheduling: Greedy Algorithms

Greedy template. Consider jobs in some order. Take each job provided it's compatible with the ones already taken.

breaks earliest start time

breaks shortest interval

breaks fewest conflicts

Interval Scheduling: Greedy Algorithm

Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it's compatible with the ones already taken.

```
Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

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

Implementation. O(n log n), due to the sorting operation

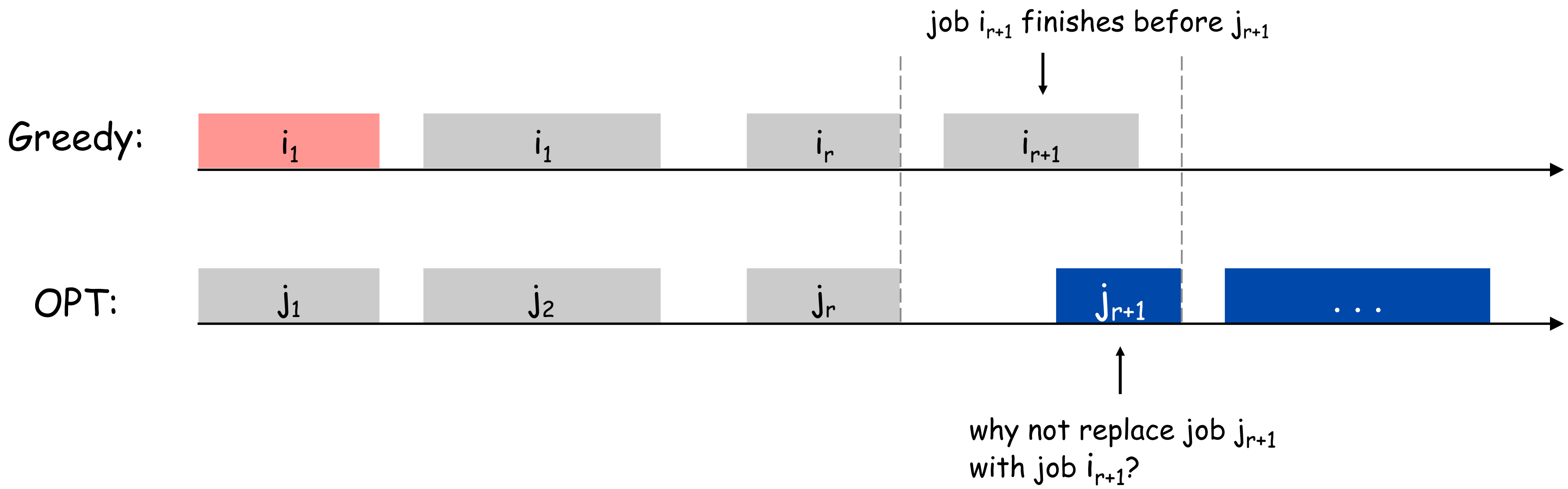
- Remember job j* that was added last to A.
- Job j is compatible with A if $s_j \geq f_{j^*}$.

Interval Scheduling: Analysis

Theorem. Greedy algorithm is optimal.

Pf. (by contradiction)

- Assume greedy is not optimal, and let's see what happens.
- Let $i_1, i_2, \ldots i_k$ denote set of jobs selected by greedy.
- Let $j_1, j_2, \ldots j_m$ denote set of jobs in the optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.

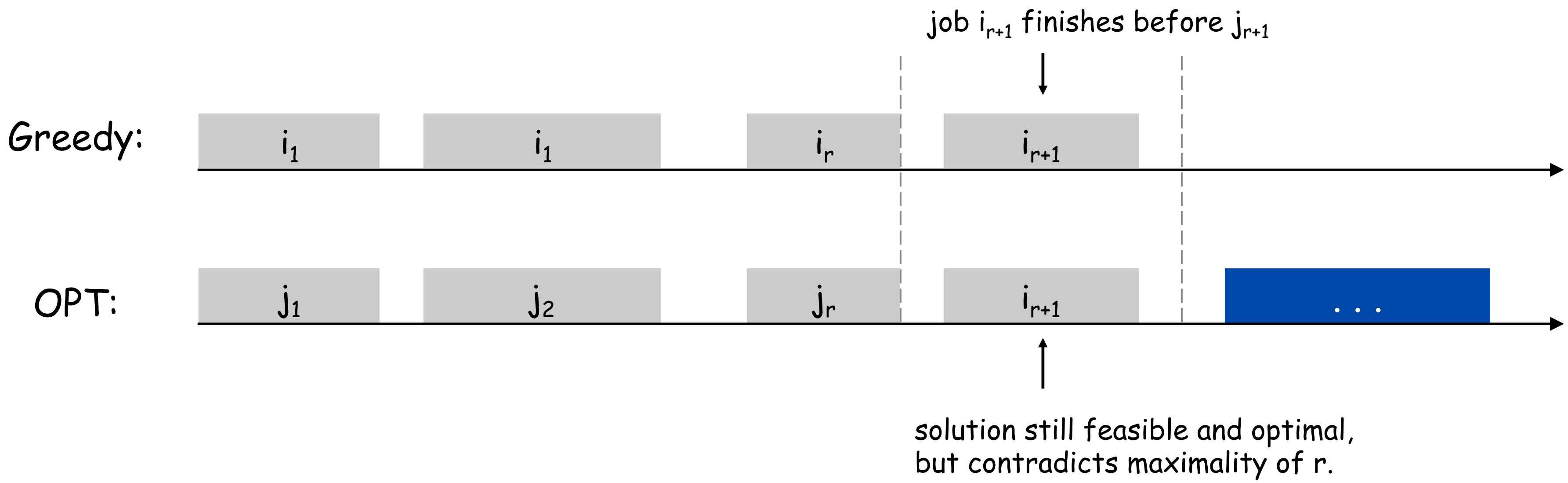


Interval Scheduling: Analysis

Theorem. Greedy algorithm is optimal.

Pf. (by contradiction)

- Assume greedy is not optimal, and let's see what happens.
- Let $i_1, i_2, \ldots i_k$ denote set of jobs selected by greedy.
- Let $j_1, j_2, \ldots j_m$ denote set of jobs in the optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.



4.1 Interval Partitioning

Interval Partitioning

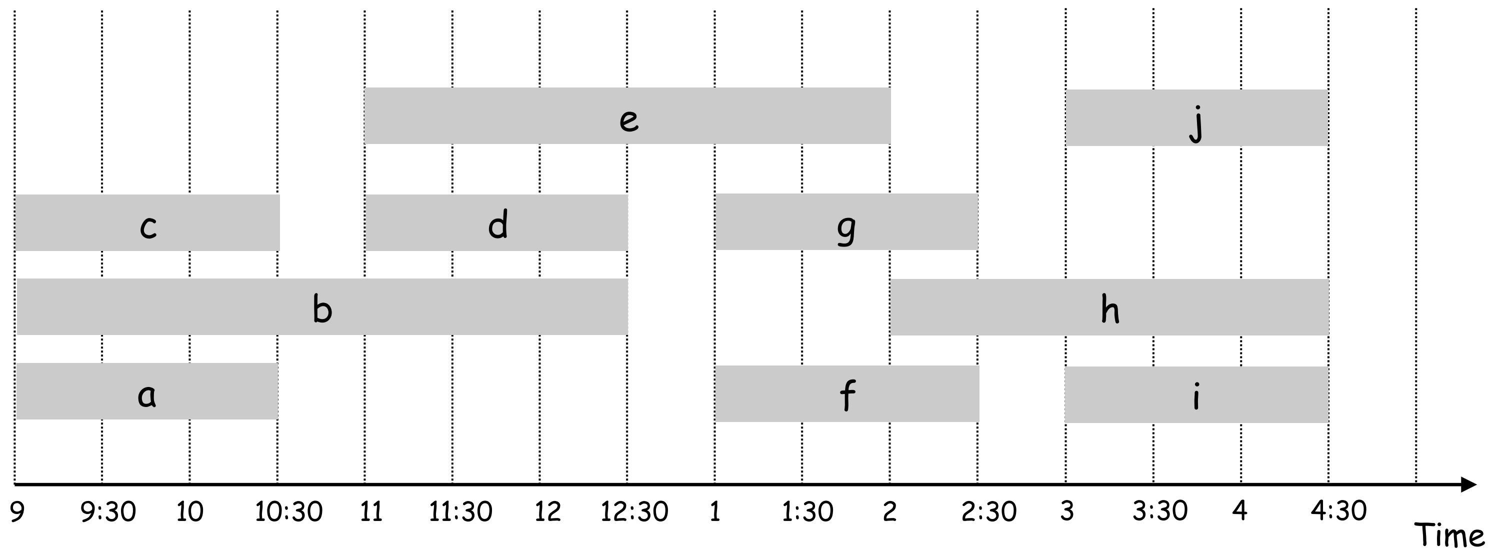
Interval partitioning.

Aim: Schedule all the requests by using as few resources as possible.

Example: Classroom Scheduling

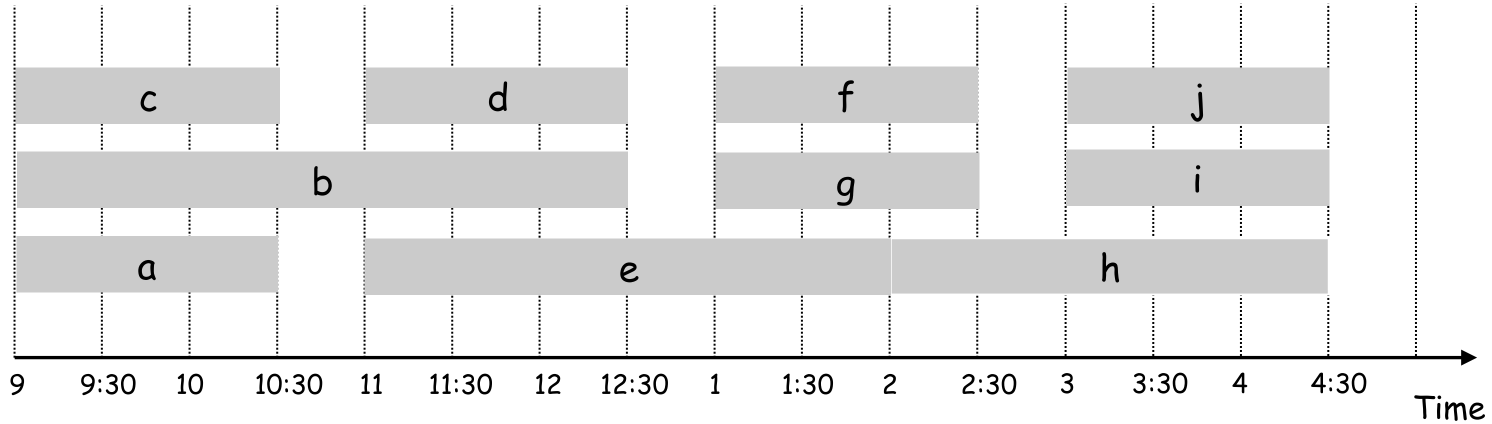
- Lecture j starts at s_j and finishes at f_j.
- Goal: find minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.

Ex: This schedule uses 4 classrooms to schedule 10 lectures.



Interval Partitioning

Ex: This schedule uses only 3.



Interval Partitioning: Lower Bound on Optimal Solution

Def. The depth of a set of intervals is the maximum number that pass over any single point on the time-line.

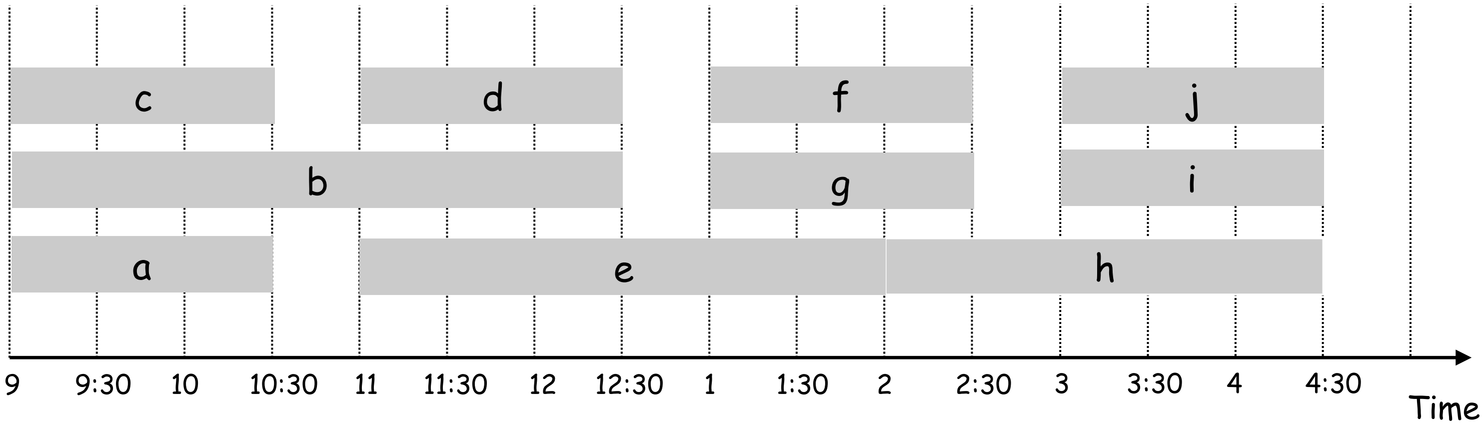
Key observation. Number of classrooms needed $\geq$ depth.

Ex: Depth of schedule below = 3 $\Rightarrow$ schedule below is optimal.

a, b, c all contain 9:30

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

R. May not be.



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 s1 ≤ s2 ≤ ... ≤ sn.
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. $O(n \log n)$.

- For each classroom k, maintain the finish time of the last job added.
- Keep the classrooms in a priority queue.

Interval Partitioning: Greedy Analysis

Observation. Greedy algorithm never schedules two incompatible lectures in the same classroom.

Theorem. Greedy algorithm is optimal.

Pf.

- Let d = number of classrooms that the greedy algorithm allocates.
- Classroom d is opened because we needed 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, we have d lectures overlapping at time $s_j + \varepsilon$.
- Key observation $\Rightarrow$ all schedules use $\geq$ d classrooms. ▪

Designing the Algorithm

Sort the intervals by their start times, breaking ties arbitrarily

Let $I_1, I_2, \ldots, I_n$ denote the intervals in this order

For j = 1, 2, 3, …,n

    For each interval I_i that precedes I_j in sorted order and overlaps it

        Exclude the label of I_i from consideration for I_j

    Endfor

    If there is any label from {1, 2, …, d} that has not been excluded

    then

        Assign a nonexcluded label to I_j

    else

        Leave I_j unlabeled

    Endif

Endfor

4.2 Scheduling to Minimize Lateness

We have a single resource and a set of n requests to use the resource for an interval of time. Each request has a deadline, d, and requires a contiguous time interval of length, t, but willing to be scheduled at any time before the deadline.

Aim: Minimizing the lateness

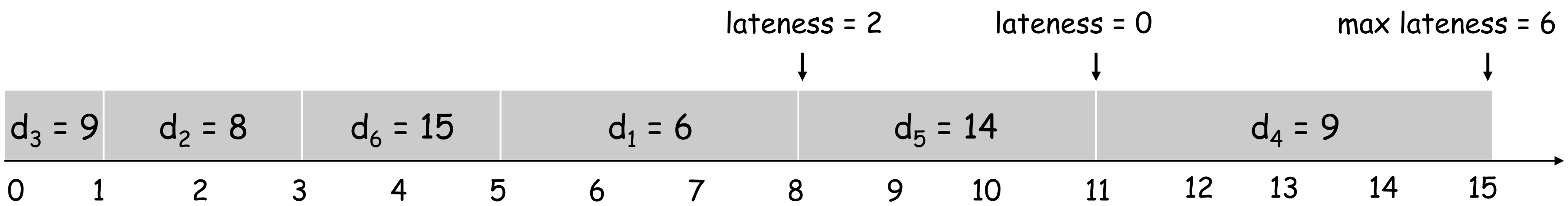
Scheduling to Minimizing Lateness

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: $\ell_j = \max \{ 0, f_j - d_j \}$.
- Goal: schedule all jobs to minimize maximum lateness $L = \max \ell_j$.

Ex:

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



Minimizing Lateness: Greedy Algorithms

Greedy template. Consider 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 $d_j - t_j$.

Minimizing Lateness: Greedy Algorithms

Greedy template. Consider jobs in some order.

- [Shortest processing time first] Consider jobs in ascending order of processing time t_j.

	1	2
t_j	1	10
d_j	100	10

counterexample

- [Smallest slack] Consider jobs in ascending order of slack $d_j - t_j$.

	1	2
t_j	1	10
d_j	2	10

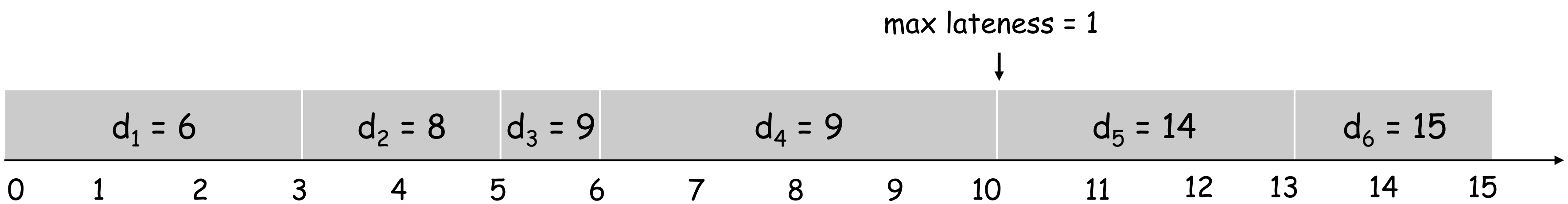
counterexample

Minimizing Lateness: Greedy Algorithm

Greedy algorithm. Earliest deadline first.

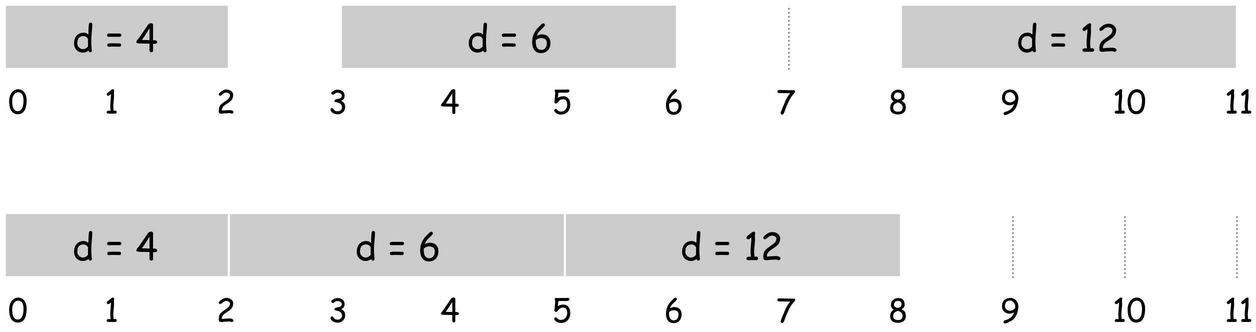
```
Sort n jobs by deadline so that d_1 ≤ d_2 ≤ … ≤ d_n

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



Minimizing Lateness: No Idle Time

Observation. There exists an optimal schedule with no idle time (no "gaps" between the scheduled jobs).

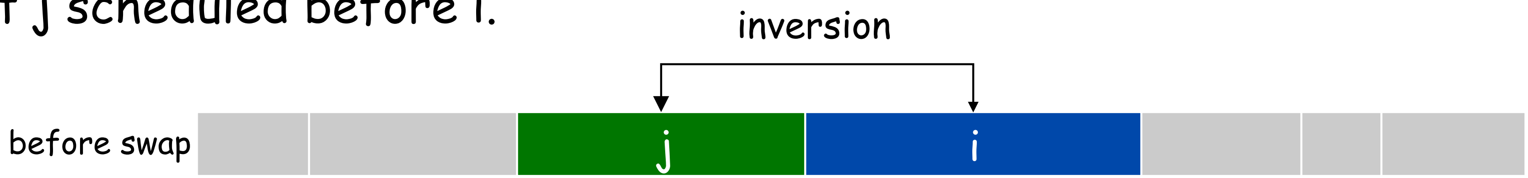


Observation. The greedy schedule has no idle time.

This is good since the aggregate execution time can not be smaller. We must check if it satisfies "minimum latenes."

Minimizing Lateness: Inversions

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

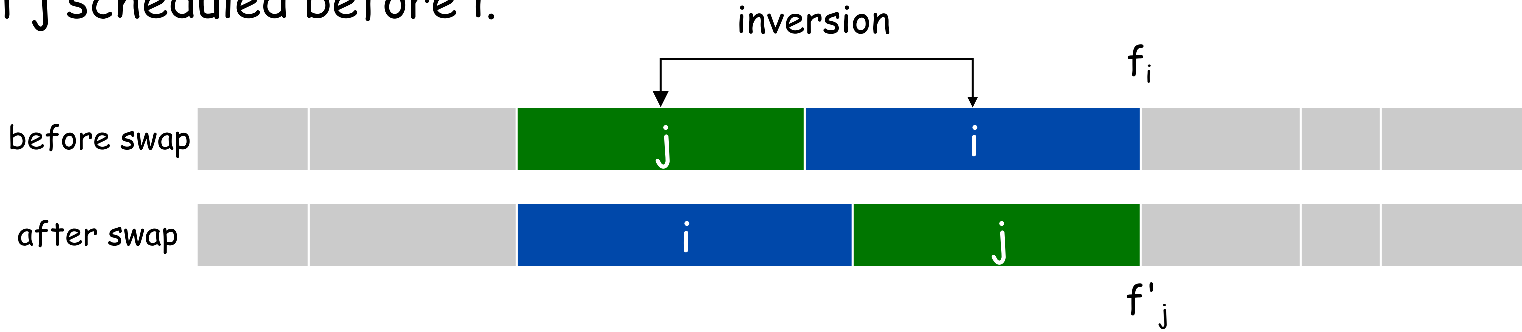


Observation. Greedy schedule has no inversions.

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

Minimizing Lateness: Inversions

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



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

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

- $\ell'_k = \ell_k$ for all $k \neq i, j$
- $\ell'_i \leq \ell_i$
- If job j is late:

$$
\begin{aligned}
\ell'_j &= f'_j - d_j && \text{(definition)} \\
&= f_i - d_j && (j \text{ finishes at time } f_i) \\
&\leq f_i - d_i && (i < j) \\
&\leq \ell_i && \text{(definition)}
\end{aligned}
$$

Minimizing Lateness: Analysis of Greedy Algorithm

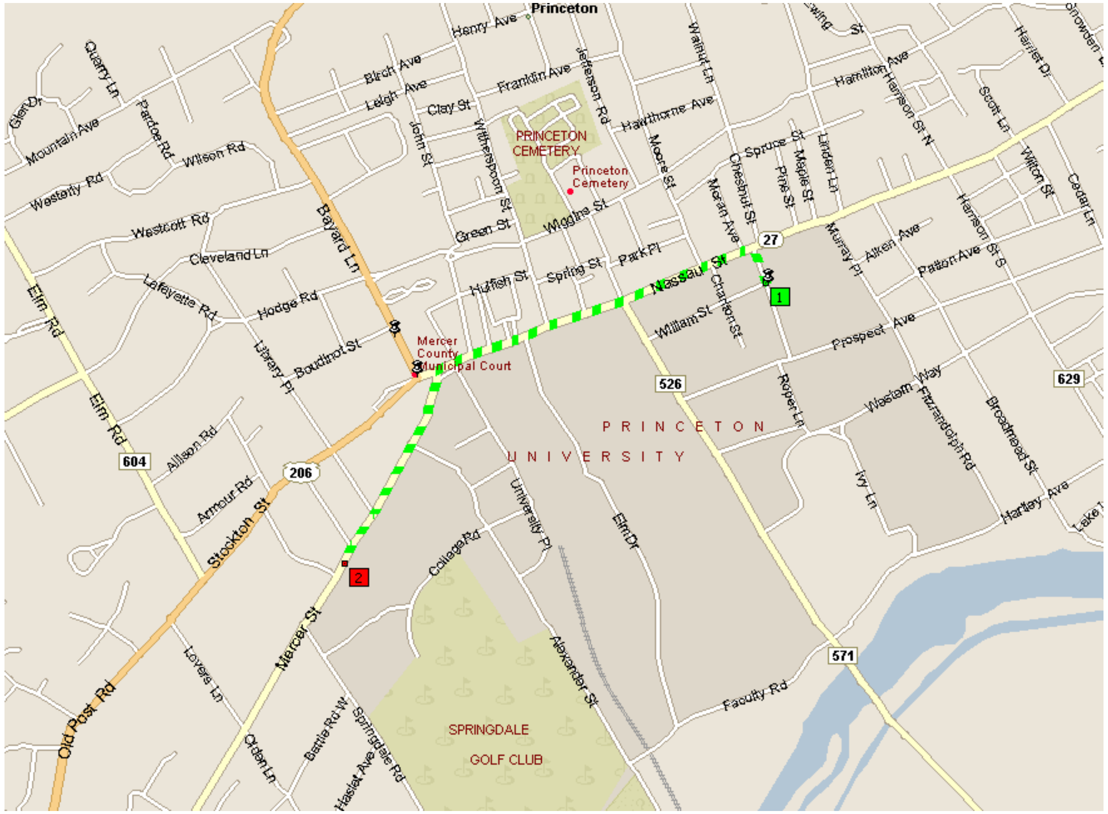
All schedules with no inversions and no idle time has the same maximum lateness.

Theorem. Greedy schedule S is optimal.

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

- Can assume S* has no idle time.
- If S* has no inversions, then S = S*.
- If S* has an inversion, let i-j be an adjacent inversion.
  - swapping i and j does not increase the maximum lateness and strictly decreases the number of inversions
  - this contradicts definition of S* ▪

4.4 Shortest Paths in a Graph



shortest path from Princeton CS department to Einstein's house

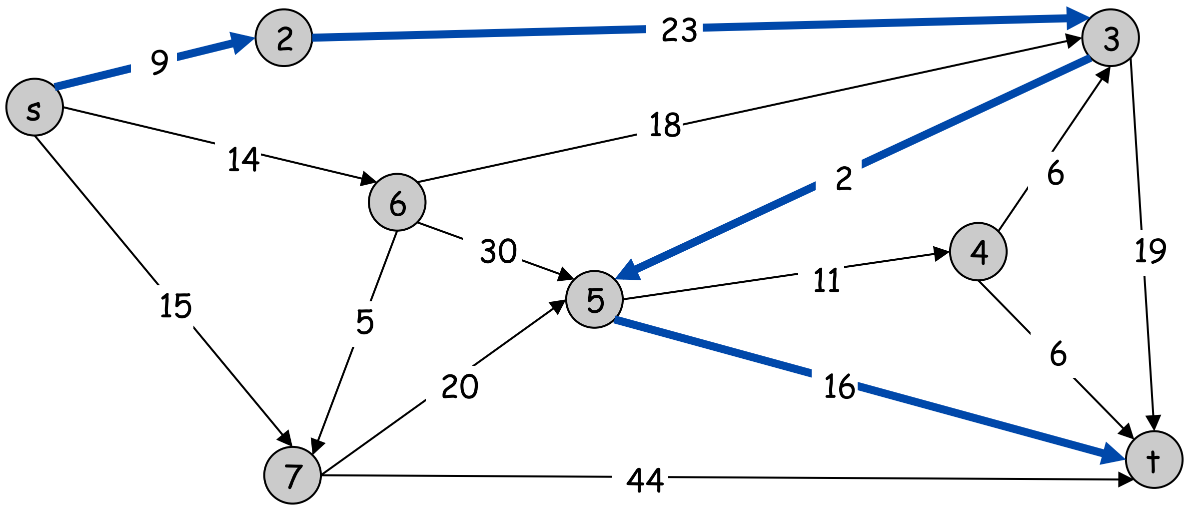
Shortest Path Problem

Shortest path network.

- Directed graph $G = (V, E)$.
- Source s, destination t.
- Length ℓ_e = length of edge e.

Shortest path problem: find shortest directed path from s to t.

cost of path = sum of edge costs in path



Cost of path s-2-3-5-t
= 9 + 23 + 2 + 16
= 48.

Dijkstra's Algorithm

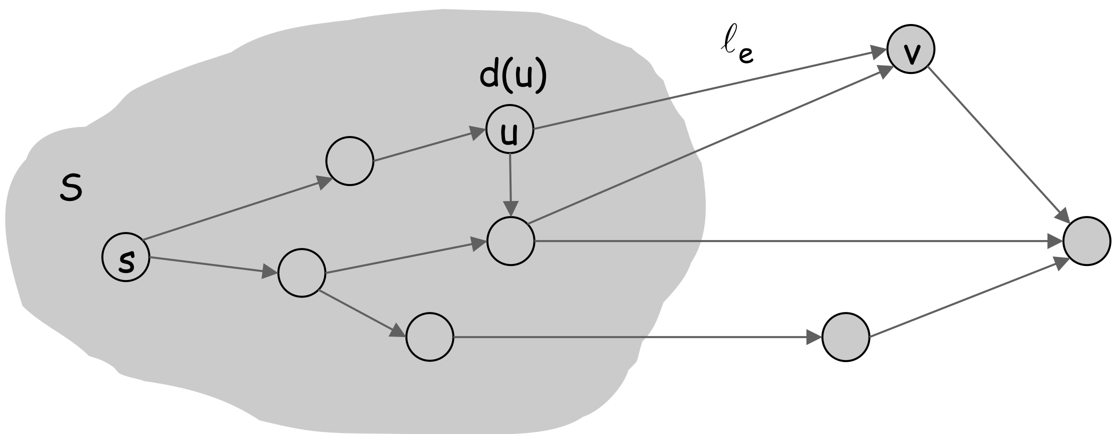
Dijkstra's algorithm.

- Maintain a set of explored nodes S for which we have determined the shortest path distance d(u) from s to u.
- Initialize S = { s }, d(s) = 0.
- Repeatedly choose unexplored node v which minimizes

$$\pi(v) = \min_{e = (u,v)\,:\, u \in S} d(u) + \ell_e ,$$

shortest path to some u in explored part, followed by a single edge (u, v)

add v to S, and set d(v) = π(v).



Dijkstra's Algorithm

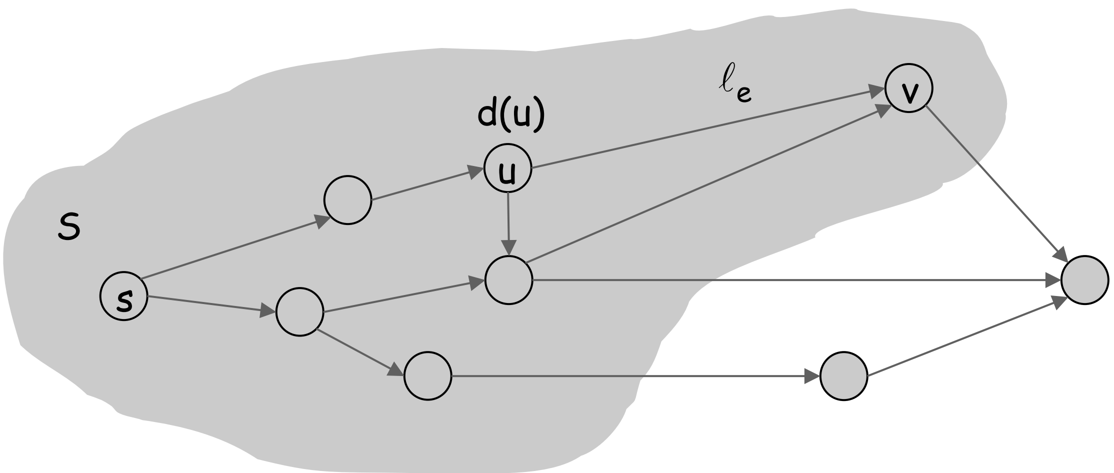
Dijkstra's algorithm.

- Maintain a set of explored nodes S for which we have determined the shortest path distance d(u) from s to u.
- Initialize S = { s }, d(s) = 0.
- Repeatedly choose unexplored node v which minimizes

$$\pi(v) = \min_{e = (u,v) \,:\, u \in S} d(u) + \ell_e,$$

shortest path to some u in explored part, followed by a single edge (u, v)

add v to S, and set d(v) = π(v).



Dijkstra's Algorithm: Proof of Correctness

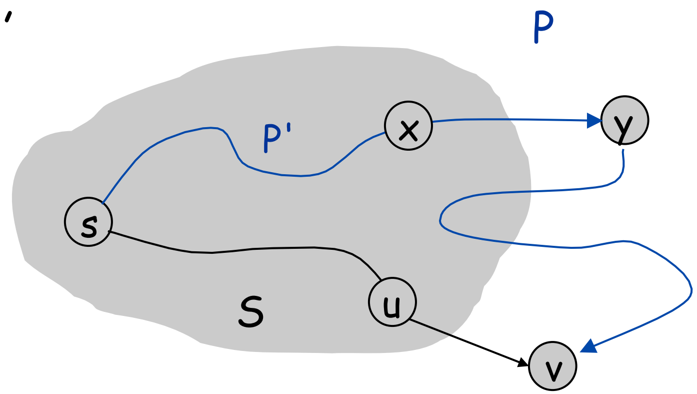
Invariant. For each node $u \in S$, $d(u)$ is the length of the shortest s-u path.

Pf. (by induction on $|S|$)

Base case: $|S| = 1$ is trivial.

Inductive hypothesis: Assume true for $|S| = k \geq 1$.

- Let v be 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 $\pi(v)$.
- Consider any s-v path P. We'll see that it's no shorter than $\pi(v)$.
- Let x-y be the first edge in P that leaves S, and let P' be the subpath to x.
- P is already too long as soon as it leaves S.



$$\ell(P) \underbrace{\geq}_{\text{nonnegative weights}} \ell(P') + \ell(x,y) \underbrace{\geq}_{\text{inductive hypothesis}} d(x) + \ell(x,y) \underbrace{\geq}_{\text{defn of } \pi(y)} \pi(y) \underbrace{\geq}_{\text{Dijkstra chose v instead of y}} \pi(v)$$

Dijkstra's Algorithm: Implementation

For each unexplored node, explicitly maintain $\pi(v) = \min_{e = (u,v) : u \in S} d(u) + \ell_e$.

- Next node to explore = node with minimum $\pi(v)$.
- When exploring v, for each incident edge e = (v, w), update

$$\pi(w) = \min \{ \ \pi(w), \ \pi(v) + \ell_e \ \}.$$

Efficient implementation. Maintain a priority queue of unexplored nodes, prioritized by $\pi(v)$.

See: 04demo-dijkstra.ppt

PQ Operation	Dijkstra	Array	Binary heap	d-way Heap	Fib heap †
Insert	n	n	log n	$d \log_d n$	1
ExtractMin	n	n	log n	$d \log_d n$	log n
ChangeKey	m	1	log n	$\log_d n$	1
IsEmpty	n	1	1	1	1
Total		n^2	m log n	$m \log_{m/n} n$	m + n log n

† Individual ops are amortized bounds

Algorithm-Dijkstra

```
function Dijkstra ( L[1..n, 1..n] ): array [2..n]
//Finds the lenght of the shortest path from node1 to each of the other nodes
of the graph with n nodes
//Input: L(i,j): Length of the edge between vertices i and j
array D[2..n]

{initialization}
C <- {2, 3, 4, …, n}
S <- {1}
for i <-2 to n do
        D[i] <- L[1, i]
 repeat (n-2)  times
        v <- some element of C minimizing D[v]
        C <- C\{v}
        S <- S U {v}
        for each (w member-of C) do
                D[w] <- min (D[w], D[v]+L[v,w])
endrepeat
return D
```

Complexity:

$\Theta(n^2)$

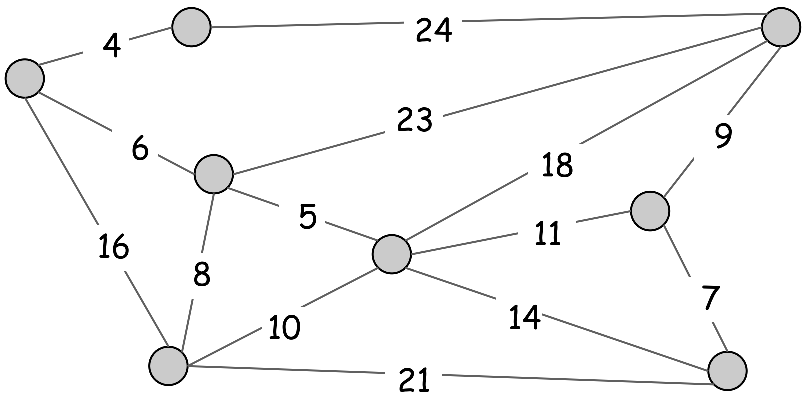
two nested loops!

n is representing number of nodes in the graph

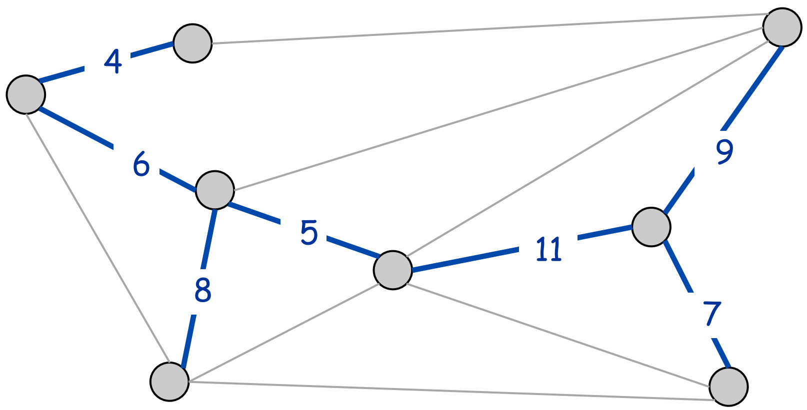
4.5 Minimum Spanning Tree

Minimum Spanning Tree

Minimum spanning tree. Given a connected graph $G = (V, E)$ with real-valued edge weights c_e, an MST is a subset of the edges $T \subseteq E$ such that T is a spanning tree whose sum of edge weights is minimized.



$G = (V, E)$



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

Cayley's Theorem. There are n^{n-2} spanning trees of K_n.

can't solve by brute force

Applications

MST is fundamental problem with diverse applications.

- Network design.
  - telephone, electrical, hydraulic, TV cable, computer, road
- Approximation algorithms for NP-hard problems.
  - traveling salesperson problem, Steiner tree
- Indirect applications.
  - max bottleneck paths
  - LDPC (low density parity check) codes for error correction
  - image registration with Renyi entropy
  - learning salient features for real-time face verification
  - reducing data storage in sequencing amino acids in a protein
  - model locality of particle interactions in turbulent fluid flows
  - autoconfig protocol for Ethernet bridging to avoid cycles in a network
- Cluster analysis.

Greedy Algorithms

Kruskal's algorithm. Start with $T = \phi$. Consider edges in ascending order of cost. Insert edge e in T unless doing so would create a cycle.

Prim's algorithm. Start with some root node s and greedily grow a tree T from s outward. At each step, add the cheapest edge e to T that has exactly one endpoint in T.

Reverse-Delete algorithm. Start with T = E. Consider edges in descending order of cost. Delete edge e from T unless doing so would disconnect T.

Remark. All three algorithms produce an MST.

DEMOS

Kruskal:
http://www.unf.edu/~wkloster/foundations/KruskalApplet/KruskalApplet.htm

Prim:
http://www-b2.is.tokushima-u.ac.jp/~ikeda/suuri/dijkstra/PrimApp.shtml?demo3

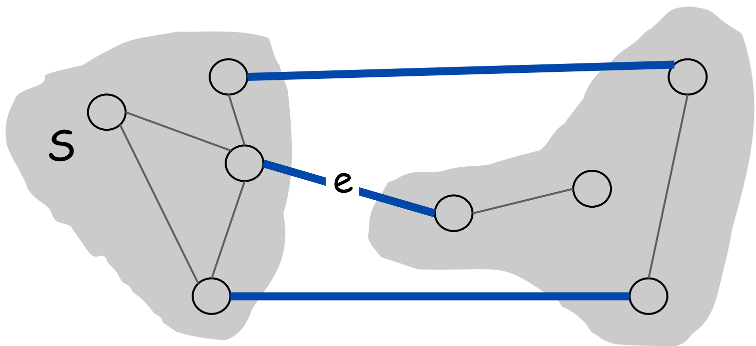
See: 04Greedy_Demo_PrimKruskal.ppt

Greedy Algorithms

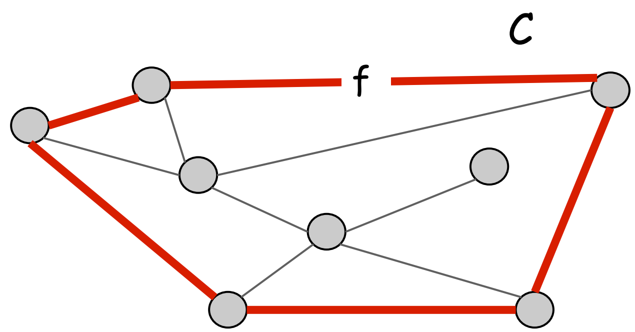
Simplifying assumption. All edge costs c_e are distinct.

Cut property. Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then the MST contains e.

Cycle property. Let C be any cycle, and let f be the max cost edge belonging to C. Then the MST does not contain f.



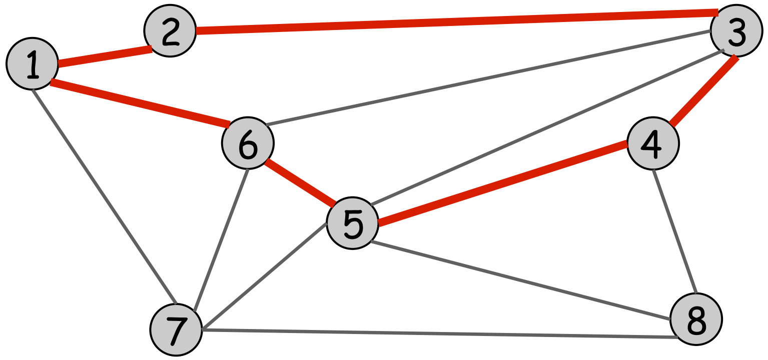
e is in the MST



f is not in the MST

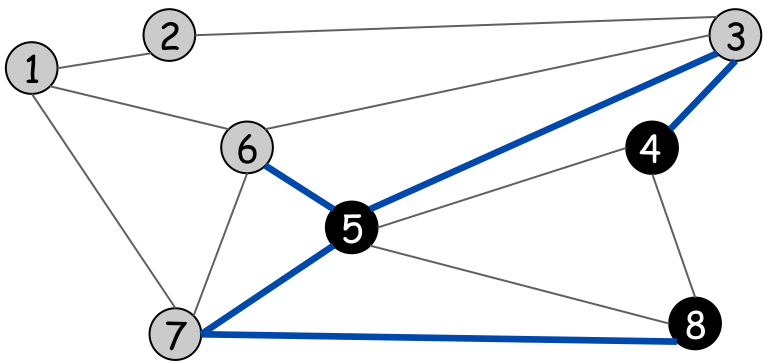
Cycles and Cuts

Cycle. Set of edges the form a-b, b-c, c-d, …, y-z, z-a.



Cycle C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1

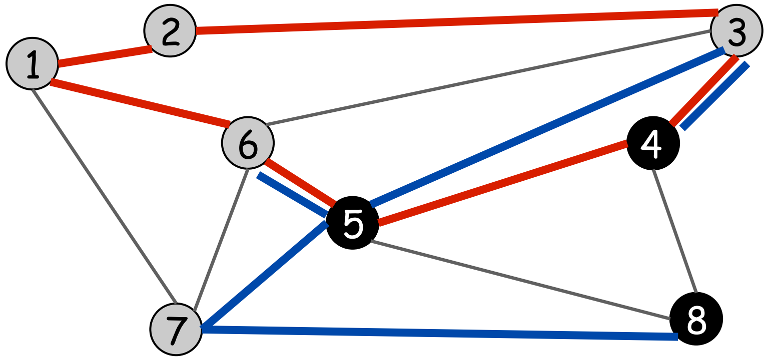
Cutset. A cut is a subset of nodes S. The corresponding cutset D is the subset of edges with exactly one endpoint in S.



Cut S = { 4, 5, 8 }
Cutset D = 5-6, 5-7, 3-4, 3-5, 7-8

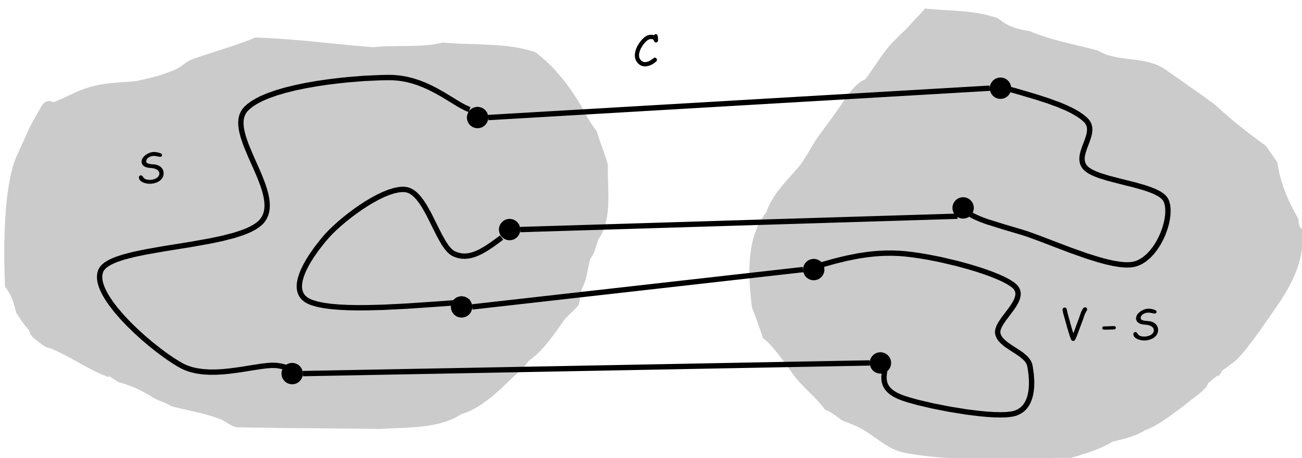
Cycle-Cut Intersection

Claim. A cycle and a cutset intersect in an even number of edges.



Cycle C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1
Cutset D = 3-4, 3-5, 5-6, 5-7, 7-8
Intersection = 3-4, 5-6

Pf. (by picture)

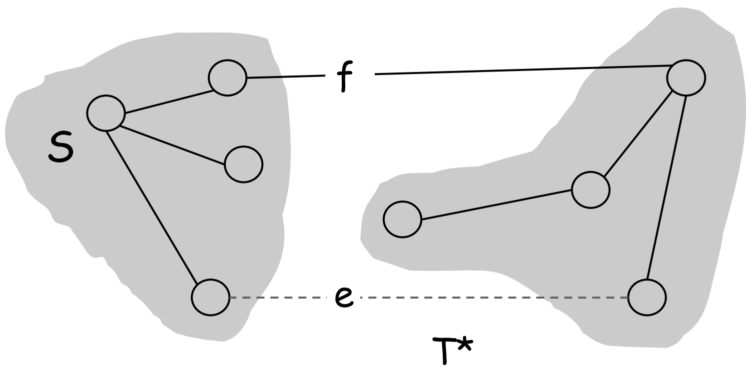


Simplifying assumption. All edge costs c_e are distinct.

Cut property. Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then the MST T* contains e.

Pf. (exchange argument)

- Suppose e does not belong to T*, and let's see what happens.
- Adding e to T* creates a cycle C in T*.
- Edge e is both in the cycle C and in the cutset D corresponding to S
  $\Rightarrow$ there exists another edge, say f, that is in both C and D.
- $T' = T^* \cup \{ e \} - \{ f \}$ is also a spanning tree.
- Since $c_e < c_f$, cost(T') < cost(T*).
- This is a contradiction. ▪

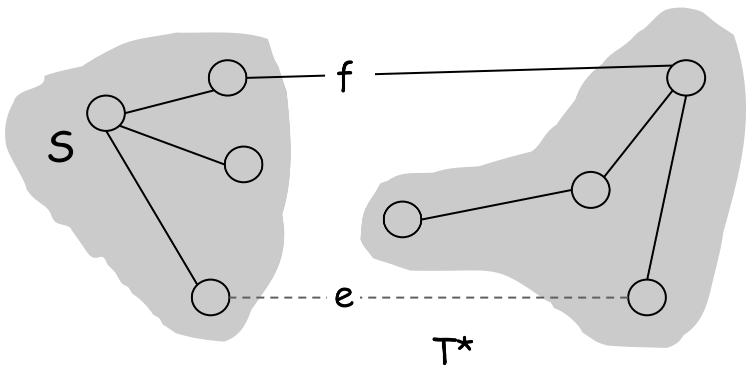


Simplifying assumption. All edge costs c_e are distinct.

Cycle property. Let C be any cycle in G, and let f be the max cost edge belonging to C. Then the MST T* does not contain f.

Pf. (exchange argument)

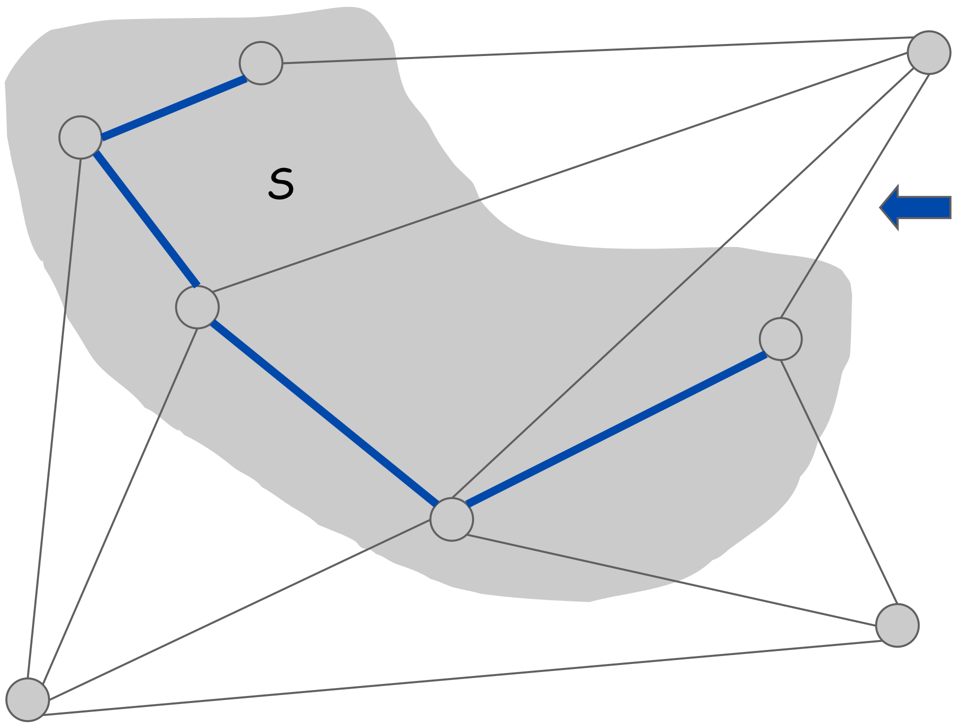
- Suppose f belongs to T*, and let's see what happens.
- Deleting f from T* creates a cut S in T*.
- Edge f is both in the cycle C and in the cutset D corresponding to S $\Rightarrow$ there exists another edge, say e, that is in both C and D.
- $T' = T^* \cup \{e\} - \{f\}$ is also a spanning tree.
- Since $c_e < c_f$, cost(T') < cost(T*).
- This is a contradiction. ▪



Prim's Algorithm: Proof of Correctness

Prim's algorithm. [Jarník 1930, Dijkstra 1957, Prim 1959]

- Initialize S = any node.
- Apply cut property to S.
- Add min cost edge in cutset corresponding to S to T, and add one new explored node u to S.



Implementation: Prim's Algorithm

Implementation. Use a priority queue ala Dijkstra.

- Maintain set of explored nodes S.
- For each unexplored node v, maintain attachment cost a[v] = cost of cheapest edge v to a node in S (a[v] = attachment cost).
- $O(n^2)$ with an array; $O(m \log n)$ with a binary heap.

```
Prim(G, c) {
   foreach (v ∈ V) a[v] ← ∞
   Initialize an empty priority queue Q
   foreach (v ∈ V) insert v onto Q
   Initialize set of explored nodes S ← φ

   while (Q is not empty) {
      u ← delete min element from Q
      S ← S ∪ { u }
      foreach (edge e = (u, v) incident to u)
          if ((v ∉ S) and (c_e < a[v]))
             decrease priority a[v] to c_e
}
```

Minimum SpanningTree-Prim's Algorithm (Detailed) (INCOMPLETE)

```
function Prim( L[1..n, 1..n] ): set of edges
//In this subprogram node 1 is selected as the arbitrary starting node
B <- {1}
T <- Ø
for i <- 2 to n do
        nearest[i] <- 1
        mindist[i] <- L[i,1]
repeat (n-1) times
        min <- infinity
        for j <- 2 to n do
                if (0 <= mindist[j] < min) then
                                min <- mindist[j]
                                k <- j
                T <- T U {(nearest[k],k)}
                mindist[k] <-  -1  //not to be considered again
                for j <- 2 to n do    //update the distances of the neighbouring nodes
                        if L[j,k] < mindist[j] then
                                        mindist[j] <- L[j,k]
                                        nearest[j] <- k
endrepeat
return T
```

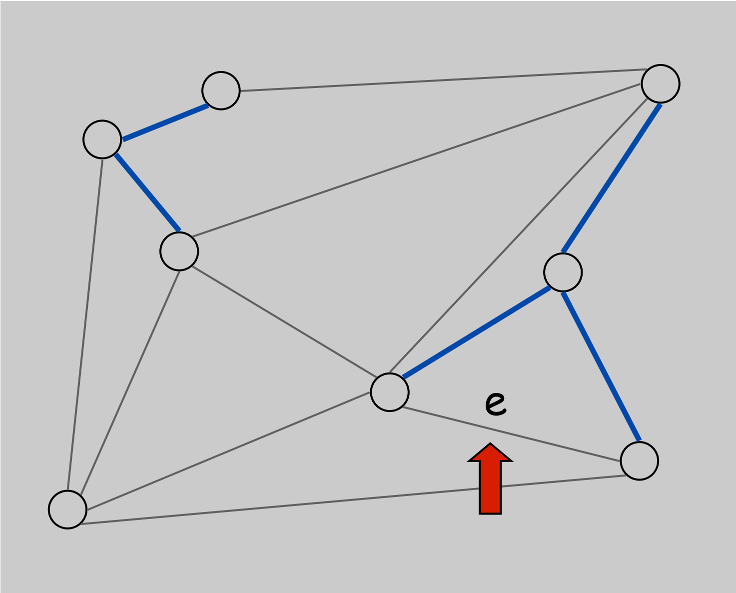
Complexity:

$\Theta(n^2)$: array representation of Q
$\Theta(m\log n)$: binary heap representation of Q

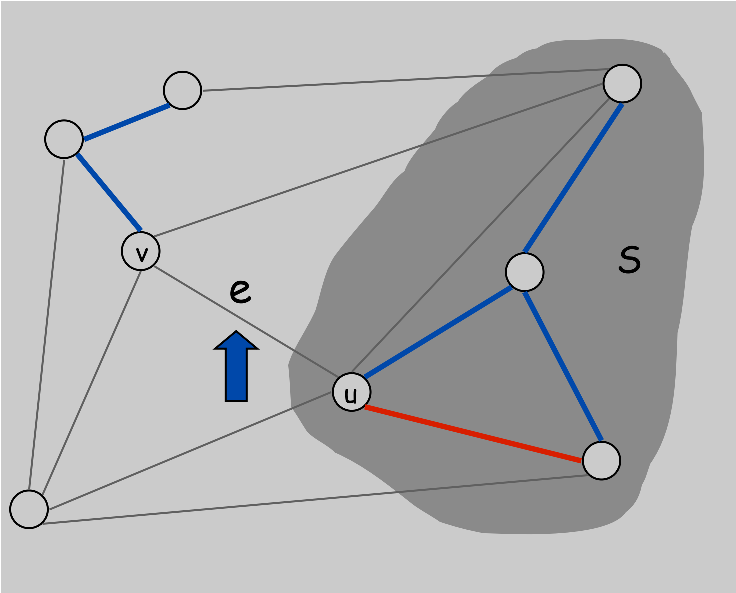
Kruskal's Algorithm: Proof of Correctness

Kruskal's algorithm. [Kruskal, 1956]

- Consider edges in ascending order of weight.
- Case 1: If adding e to T creates a cycle, discard e according to cycle property.
- Case 2: Otherwise, insert e = (u, v) into T according to cut property where S = set of nodes in u's connected component.



Case 1



Case 2

Implementation: Kruskal's Algorithm

An extremely slow growing fn Inverse Ackerman Fn

Implementation. Use the union-find data structure.

- Build set T of edges in the MST.
- Maintain set for each connected component.
- $O(m \log n)$ for sorting and $O(m \underbrace{\alpha(m, n)}_{\text{essentially a constant}})$ for union-find.

$m \le n^2 \Rightarrow \log m$ is $O(\log n)$

```
Kruskal(G, c) {
   Sort edges weights so that c1 ≤ c2 ≤ ... ≤ cm.
   T ← φ

   foreach (u ∈ V) make a set containing singleton u

   for i = 1 to m
      (u,v) = ei
      if (u and v are in different sets) {
         T ← T ∪ {ei}
         merge the sets containing u and v
      }
   return T
}
```

are u and v in different connected components?

merge two components

O(mlogm)

O(n)

m times

O(logn)

Union Find Data Structure

- Useful when we keep adding nodes.
- MakeUnionFind(S): returns a data structure on S where each node is in a separate set.
  - O(|S|)
- Find(u): returns the name of the set containing element u.
  - O(log(n))   (linked list repr).
- Union(A,B): merge sets A and B into a single set.
  - O(1) time (linked list repr.)
- Array and pointer implementation are possible. Pointer implementation is faster. See pages 152-155 of the book.

Minimum Spanning Tree-Kruskal's Algorithm

```
function Kruskal(G = <N, A>:graph; length:A ->  R+): set of edges
{initialization}
sort A by increasing length
n <- the number of nodes in N
T <- Ø
Initialize n sets, each containing a different element of N
{greedy loop}
repeat
        e <- (u,v)  :shortest edge has not yet considered
        ucomp <- find(u)
        vcomp <- find(v)
        if ucomp<>vcomp then
                merge(ucomp, vcomp)
                T <- T U {e}
until T contains n-1 edges
return T
```

Complexity:

- $\Theta(m\log m)$: sorting the edges where m represents the number of edges.
Actually $O(m\log n)$ since $(n-1) <= m <= n(n-1)/2$.

- $\Theta(n)$: initializing n disjoint sets.

- $\Theta(m)$: total complexity of find operations, since there can be at most 2m of them.

- $\Theta(n)$: total complexity of merge operations, since there can be at most (n-1) merge operations

Hence, $\Theta(m\log n)$ is the complexity of the algorithm.

Complexity:

$\Theta(mlogn)$: using union-find data structure with linked list implementation

NOTE

- SUBJECT NOT COVERED THIS YEAR, BUT INTERESTING:

- Optimal Caching
- Clustering
- Huffman Codes (Homework)