Design and Analysis of Algorithms
Review

Si Wu

School of CSE, SCUT

cswusi@scut.edu.cn

TA: Wenhao Wu (1565865638@qq.com)
Yi Liu (1337545838@qq.com)

Topics

- **Algorithm Analysis**
- **Recurrence**
- **Divide-and-Conquer**
- **Greedy Algorithm**
- **Linear Programming**
- **Dynamic Programming**

Review:
Algorithm Analysis

O-notation

***O(g(n))* = {*f(n)*: There exist positive constants *c* and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$}**

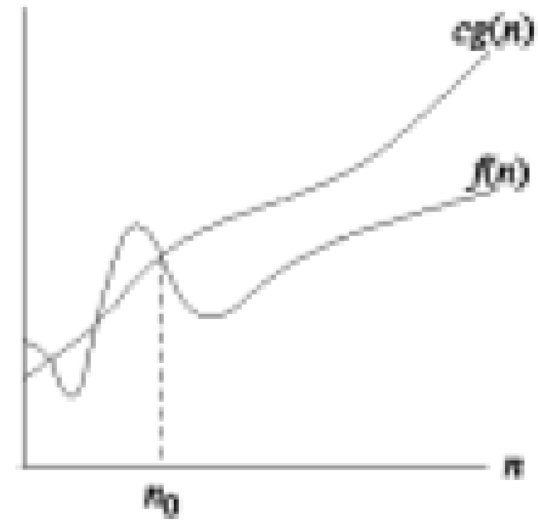
--*O(.)* is used to asymptotically upper bound a function.

--*O(.)* is used to bound worst-case running time.

Ex. $f(n) = 32n^2 + 17n + 1$

- $f(n)$ is $O(n^2)$
- $f(n)$ is also $O(n^3)$
- $f(n)$ is neither $O(n)$ nor $O(nlgn)$

Typical usage. **Insertion-Sort makes $O(n^2)$ compares to sort n elements.**



O-notation

Ex.

- $1/3n^2 - 3n \in O(n^2)$

Because $1/3n^2 - 3n \leq cn^2$ **if** $c \geq 1/3 - 3/n$ **which holds for** $c = 1/3$ **and** $n > 1$.

- $k_1n^2 + k_2n + k_3 \in O(n^2)$

Because $k_1n^2 + k_2n + k_3 \leq (k_1 + |k_2| + |k_3|)n^2$ **and for** $c > k_1 + |k_2| + |k_3|$ **and** $n \geq 1$, $k_1n^2 + k_2n + k_3 \leq cn^2$.

- $k_1n^2 + k_2n + k_3 \in O(n^3)$

As $k_1n^2 + k_2n + k_3 \leq (k_1 + |k_2| + |k_3|)n^3$ **(upper bound).**

Ω-notation

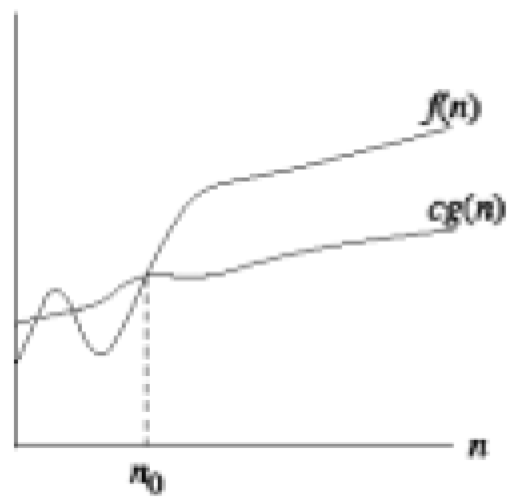
***Ω(g(n))* = {*f(n)*: There exist positive constants *c* and n_0 such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0$}**

- **We use *Ω*-notation to give a lower bound on a function.**

Ex. $f(n) = 32n^2 + 17n + 1$

- $f(n)$ **is both** $\Omega(n^2)$ **and** $\Omega(n)$
- $f(n)$ **is neither** $\Omega(n^3)$ **nor** $\Omega(n^3 lgn)$

Typical usage. Any compare-based sorting algorithm requires $\Omega(nlgn)$ **compares in the worst case.**



Ω-notation

Ex.

- $1/3n^2 - 3n \in \Omega(n^2)$

Because $1/3n^2 - 3n \geq cn^2$ if $c \leq 1/3-3/n$ which holds for $c = 1/6$ and $n > 18$.

- $k_1n^2+k_2n+k_3 \in \Omega(n^2)$

- $k_1n^2+k_2n+k_3 \in \Omega(n)$

Θ-notation

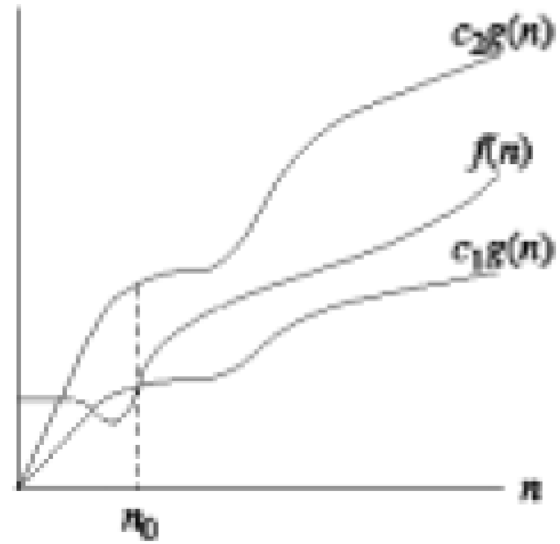
$\Theta(g(n)) = \{f(n)$: There exist positive constants c_1, c_2 and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0\}$

- We use Θ-notation to give a tight bound on a function.
- $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

Ex. $f(n) = 32n^2 + 17n + 1$

- $f(n)$ is $\Theta(n^2)$
- $f(n)$ is neither $\Theta(n)$ nor $\Theta(n^3)$

Typical usage. Merge-Sort makes $\Theta(nlgn)$ compares to sort n elements.



Θ-notation

Ex.

- $k_1n^2+k_2n+k_3 \in \Theta(n^2)$

- $6n\log n + \sqrt{n}\log^2 n = \Theta(n\log n)$

We need to find $c_1, c_2, n_0 > 0$ such that $c_1 n\log n \le 6n\log n + \sqrt{n}\log^2 n \le c_2 n\log n$ for $n \ge n_0$.

- ➢ $c_1 n\log n \le 6n\log n + \sqrt{n}\log^2 n \rightarrow c_1 \le 6 + \log n/\sqrt{n}$ **, which is true if we choose $c_1 = 6$ and $n_0 = 1$.**
- ➢ $6n\log n + \sqrt{n}\log^2 n \le c_2 n\log n \rightarrow 6 + \log n/\sqrt{n} \le c_2$**, which is true if we choose $c_2 = 7$ and $n_0 = 2$. This is because $\log n \le \sqrt{n}$ if $n \ge 2$. So $c_1 = 6$, $c_2 = 7$ and $n_0 = 2$ works.**

Useful Facts

- If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c > 0$, then $f(n)$ is $\Theta(g(n))$.
  By definition of the limit, there exists n_0 such that for all $n \geq n_0$
  $$\frac{1}{2}c \leq \frac{f(n)}{g(n)} \leq 2c$$
  Thus, $f(n) \leq 2cg(n)$ for all $n \geq n_0$, which implies $f(n)$ is $O(g(n))$.
  Similarly, $f(n) \geq \frac{1}{2}cg(n)$ for all $n \geq n_0$, which implies $f(n)$ is $\Omega(g(n))$.

- If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, then $f(n)$ is $O(g(n))$ but not $\Theta(g(n))$.

Review:
Recurrence

Induction

Induction used to prove that a statement T(n) holds for all integers n:

- Base case: prove T(0)
- Assumption: assume that T(n-1) is true
- Induction step: prove that T(n-1) implies T(n) for all *n>0*

Strong induction: when we assume T(k) is true for all $k \leq n - 1$ and use this in proving T(n)

Induction

The most general method:

Guess: the form of the solution.

Verify: by induction.

Ex. $T(n) = 4T(n/2) + bn$

Base case $T(1) = \Theta(1)$.

Guess $O(n^3)$. (Prove O and Ω separately.)

Assume that $T(k) \leq ck^3$ for $k < n$.

Prove $T(n) \leq cn^3$ by induction.

Example of Substitution

$$
\begin{aligned}
T(n) &= 4T\left(\frac{n}{2}\right) + bn \\
&\leq 4c\left(\frac{n}{2}\right)^3 + bn \\
&= \left(\frac{c}{2}\right)n^3 + bn \\
&= cn^3 - \left(\left(\frac{c}{2}\right)n^3 - bn\right) \\
&\leq cn^3
\end{aligned}
$$

$\boxed{T(k) \leq ck^3 \text{ for } k < n}$

For example, if $c \geq 2b$, then $\left(\frac{c}{2}\right)n^3 - bn \geq 0$.

Example of Substitution

Use algebraic manipulation to make an unknown recurrence similar to what you have seen before.

Ex. $T(n) = 2T(\sqrt{n}) + \log n$

Set $m = \log n$ and we have $T(2^m) = 2T(2^{m/2}) + m$

Set $S(m) = T(2^m)$ and we have $S(m) = 2S(m/2) + m$

$\rightarrow S(m) = O(m \log m)$

As a result, we have $T(n) = O(\log n \log \log n)$

A Useful Recurrence Relation

- $T(n) =$ max number of compares to Merge-Sort a list of size $\leq n$
- $T(n)$ is monotone nondecreasing.

Merge-Sort recurrence

$$T(n) \leq \begin{cases} 0, & if\ n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n, & otherwise \end{cases}$$

Solution. $T(n)\ is\ O(nlogn)$

Assorted proofs. We describe several ways to solve this recurrence. Initially we assume n is a power of 2 and replace "≤" with "=" in the recurrence.

Proof by Induction

If $T(n)$ satisfies the following recurrence, then $T(n)\ is\ O(nlogn)$.

$$T(n) = \begin{cases} 0, & if\ n = 1 \\ 2T(n/2) + n, & otherwise \end{cases}$$

assuming n is a power of 2

- Base case: when $n = 1, T(1) = 0 = nlogn$.
- Inductive hypothesis: assume $T(n) = nlogn$.
- Goal: show that $T(2n) = 2nlog(2n)$

$$\begin{aligned} T(2n) &= 2T(n) + 2n \\ &= 2nlogn + 2n \\ &= 2n(\log(2n) - 1) + 2n \\ &= 2nlog(2n) \end{aligned}$$

Analysis of Merg-Sort Recurrence

If $T(n)$ satisfies the following recurrence, then $T(n) \le n\lceil logn \rceil$.

$$T(n) \le \begin{cases} 0, & if\ n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n, & otherwise \end{cases}$$

- Base case: n=1, $T(1) = 0$.
- Define: $n_1 = \lfloor n/2 \rfloor$ and $n_2 = \lceil n/2 \rceil$.
- Induction step: assume true for 1, 2, …, n-1.

$$\begin{aligned} T(n) &\le T(n_1) + T(n_2) + n \\ &\le n_1 \lceil \log_2 n_1 \rceil + n_2 \lceil \log_2 n_2 \rceil + n \\ &\le n_1 \lceil \log_2 n_2 \rceil + n_2 \lceil \log_2 n_2 \rceil + n \\ &= n \lceil \log_2 n_2 \rceil + n \\ &\le n (\lceil \log_2 n \rceil - 1) + n \\ &= n \lceil \log_2 n \rceil \end{aligned}$$

$$\boxed{\begin{aligned} n_2 &= \lceil n/2 \rceil \\ &\le \left\lceil 2^{\lceil \log_2 n \rceil} / 2 \right\rceil \\ &= 2^{\lceil \log_2 n \rceil} / 2 \end{aligned}}$$

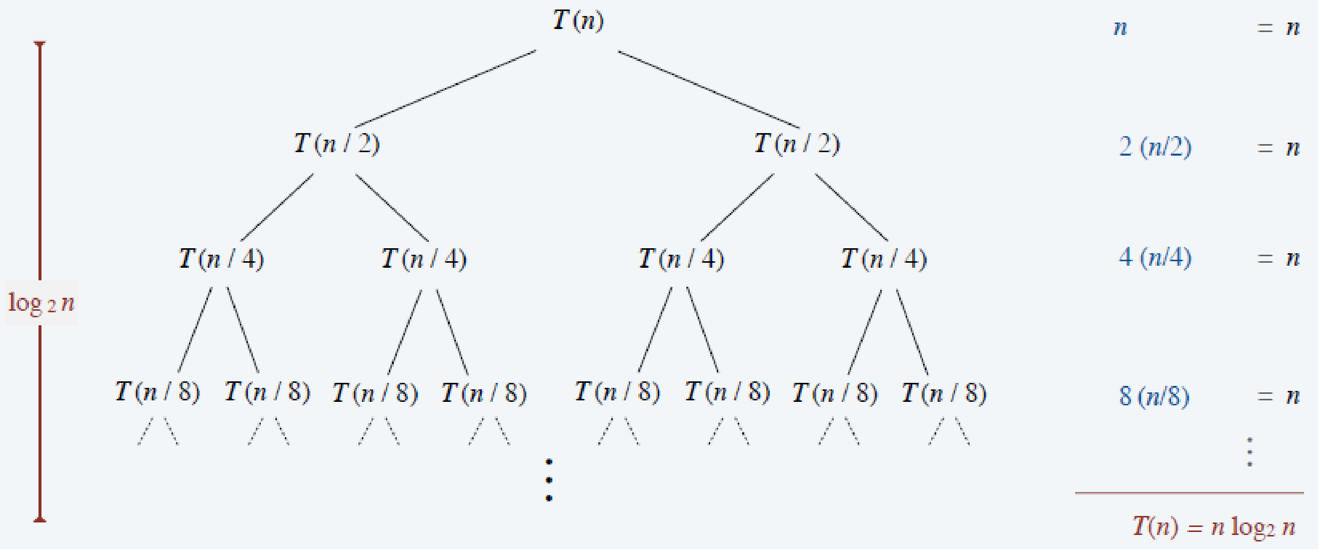
$$\log_2 n_2 \le \lceil \log_2 n \rceil - 1$$

Recursion Tree

If $T(n)$ satisfies the following recurrence, then $T(n)\ is\ O(nlogn)$.

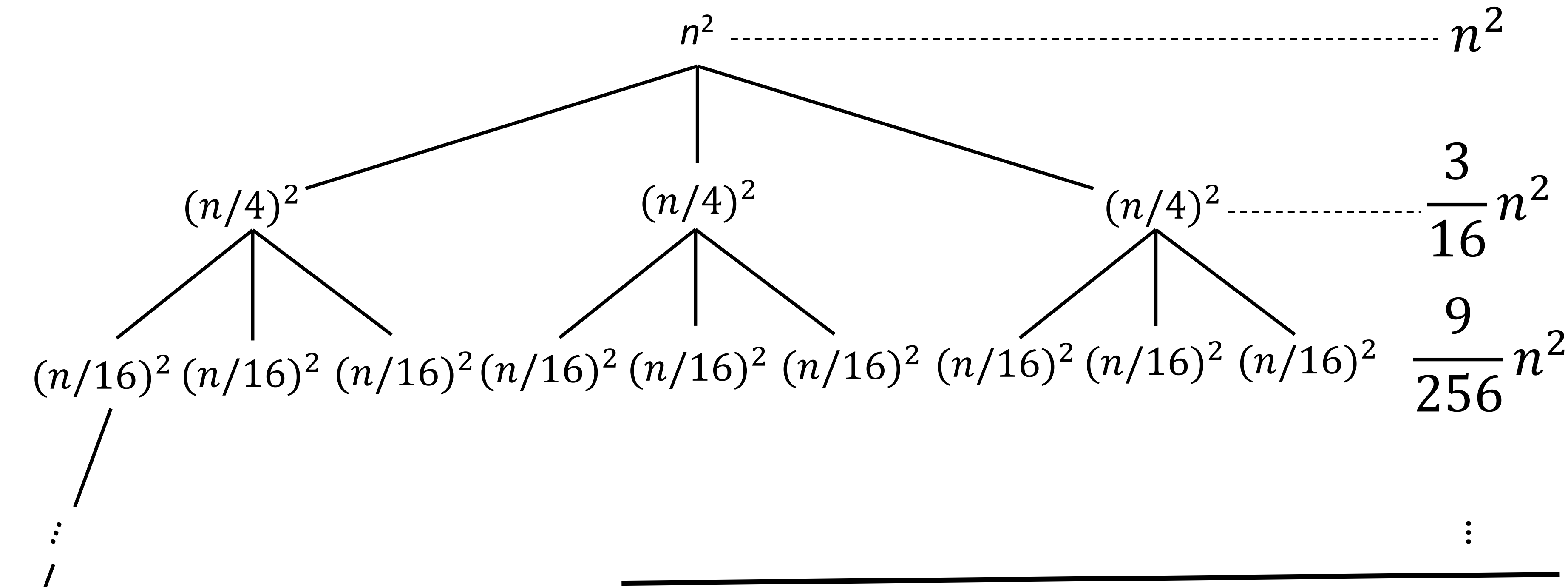
$$T(n) = \begin{cases} 0, & if\ n = 1 \\ 2T(n/2) + n, & otherwise \end{cases}$$

assuming n is a power of 2



Example of Recursion Tree

Solve $T(n) = 3T(n/4) + n^2$:



$$\text{Total} = n^2\left(1 + \frac{3}{16} + \left(\frac{3}{16}\right)^2 + \cdots\right) + n^{\log_4 3}$$

$$= \Theta(n^2)$$ *geometric series*

Review:
Divide-and-Conquer

Divide-and-Conquer Paradigm

Divide-and-Conquer.

- Divide up problem into several subproblems.
- Solve each subproblem recursively.
- Combine solution to subproblems into overall solution.

Most common usage.

- Divide problem of size n into two subproblems of size n/2 in linear time.
- Solve two subproblems recursively.
- Combine two solutions into overall solution in linear time.

Merge-Sort Algorithm

Using divide-and-conquer, we can obtain a merge-sort algorithm

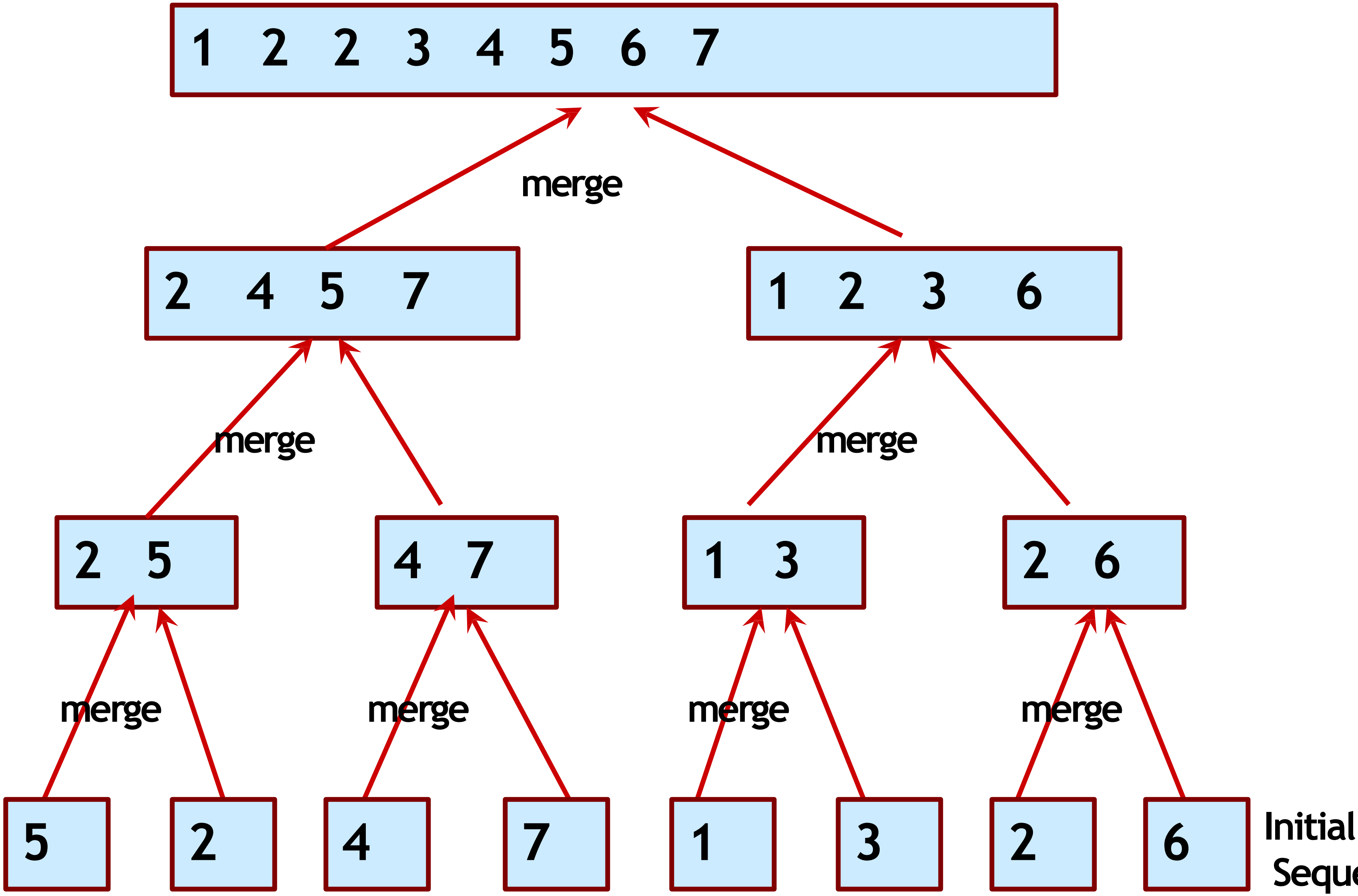
- Divide: Divide the n elements into two subsequences of n/2 elements each.
- Conquer: Sort the two subsequences recursively.
- Combine: Merge the two sorted subsequences to produce the sorted answer.

Merge-Sort (A, p, r)

- **INPUT: a sequence of n numbers stored in array A**
- **OUTPUT: an ordered sequence of n numbers**

```
MERGE-SORT (A, p, r)
1   if p < r
2       then q ← ⌊(p + r)/2⌋
3           MERGE-SORT(A, p, q)
4           MERGE-SORT(A, q + 1, r)
5           MERGE(A, p, q, r)
```

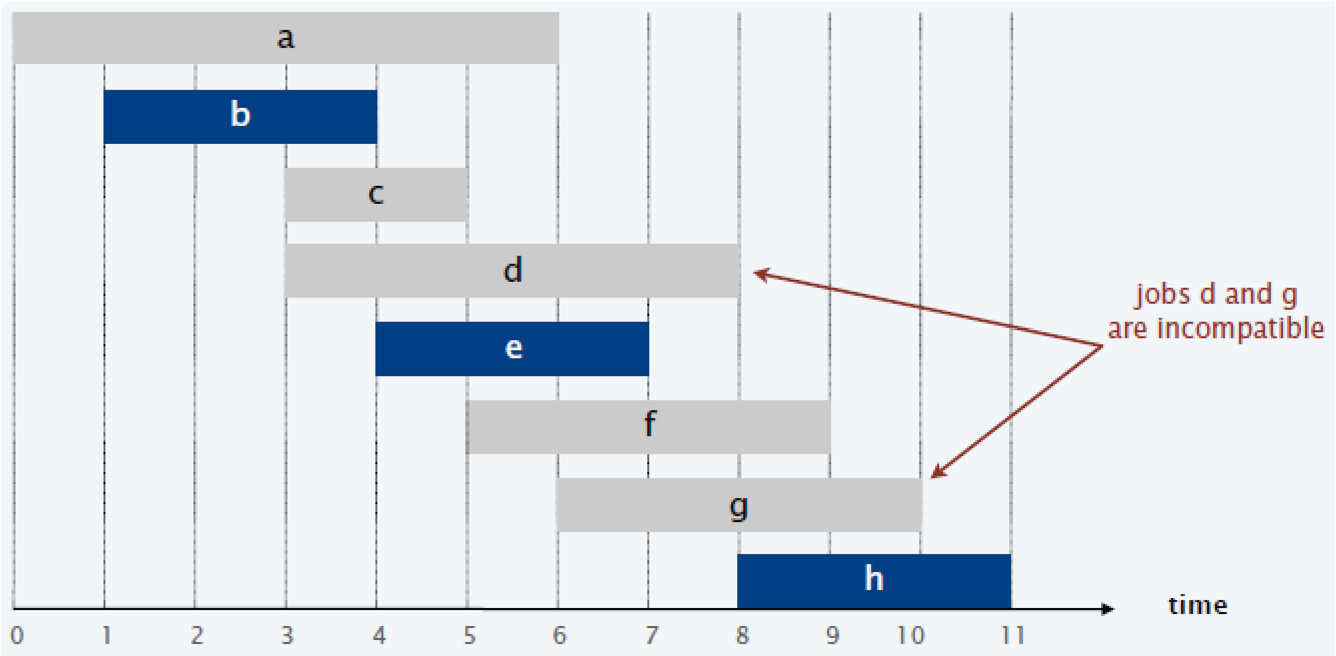
Action of Merge Sort



Review:
Greedy Algorithm

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.



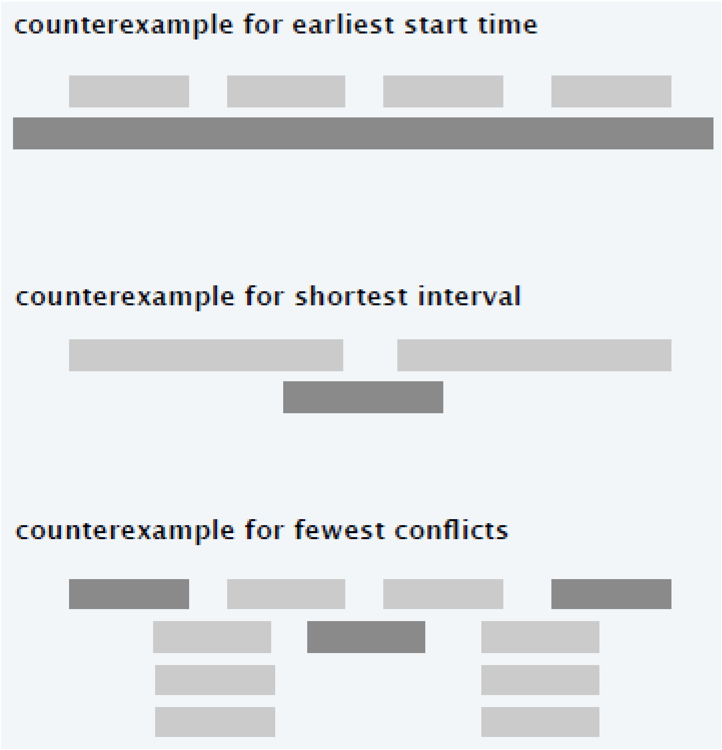
Interval Scheduling: Greedy Algorithms

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

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

Interval Scheduling: Greedy Algorithms

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



Interval Scheduling: Earliest-Finish-Time-First Algorithm

Earliest-Finish-Time-First $(n, s_1, s_2, \dots, s_n, f_1, f_2, \dots, f_n)$

Sort jobs by finish time so that $f_1 \leq f_2 \leq \cdots \leq f_n$

$A \leftarrow \emptyset$ (set of jobs selected)

for $j = 1$ to n

If job j is compatible with A

$A \leftarrow A \cup \{j\}$

Return A

The Earliest-Finish-Time-First algorithm is optimal.

Proposition. Can implement Earliest-Finish-Time-First in $O(nlogn)$ time.

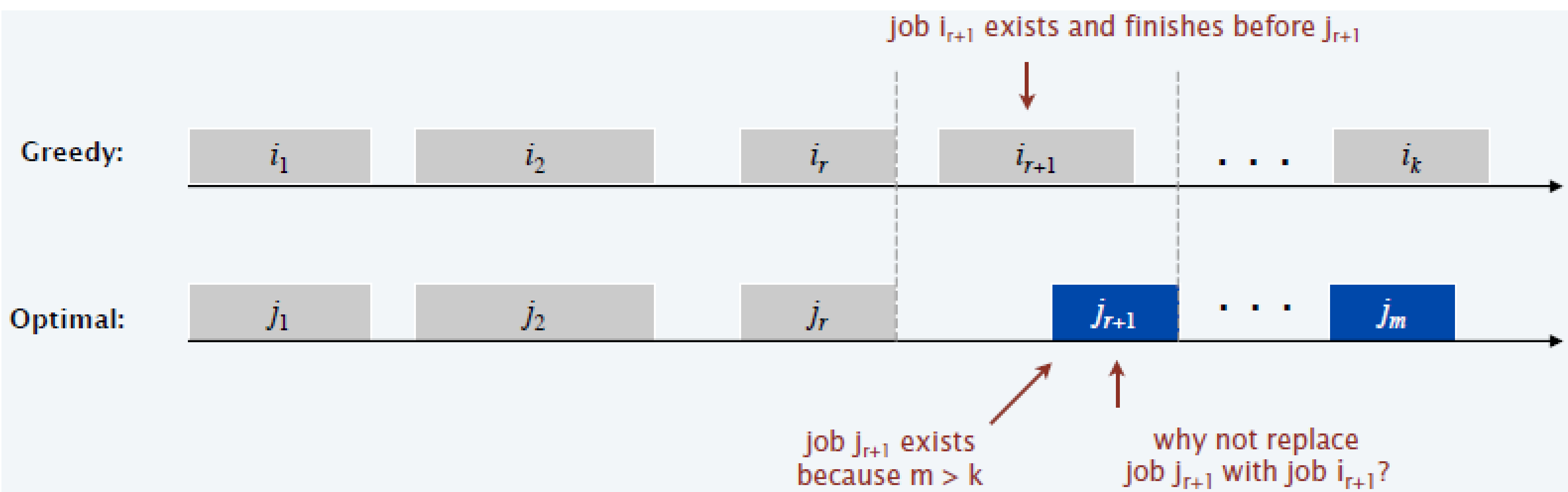
- Keep track of job j^* that was added last to A
- Job j is compatible with A iff $s_n \geq f_{j^*}$.
- Sorting by finish time takes $O(nlogn)$ time.

Interval Scheduling: Analysis of Earliest-Finish-Time-First Algorithm

Theorem. The Earliest-Finish-Time-First algorithm is optimal.

Pf.

- 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 an optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.

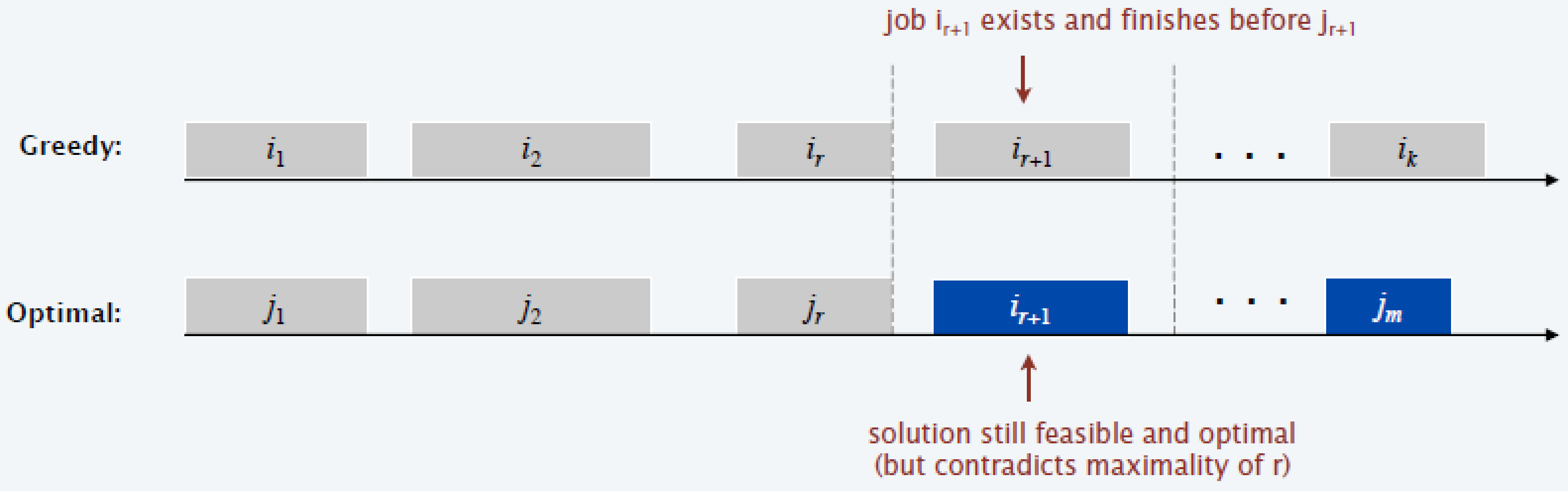


Interval Scheduling: Analysis of Earliest-Finish-Time-First Algorithm

Theorem. The Earliest-Finish-Time-First algorithm is optimal.

Pf.

- 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 an optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.

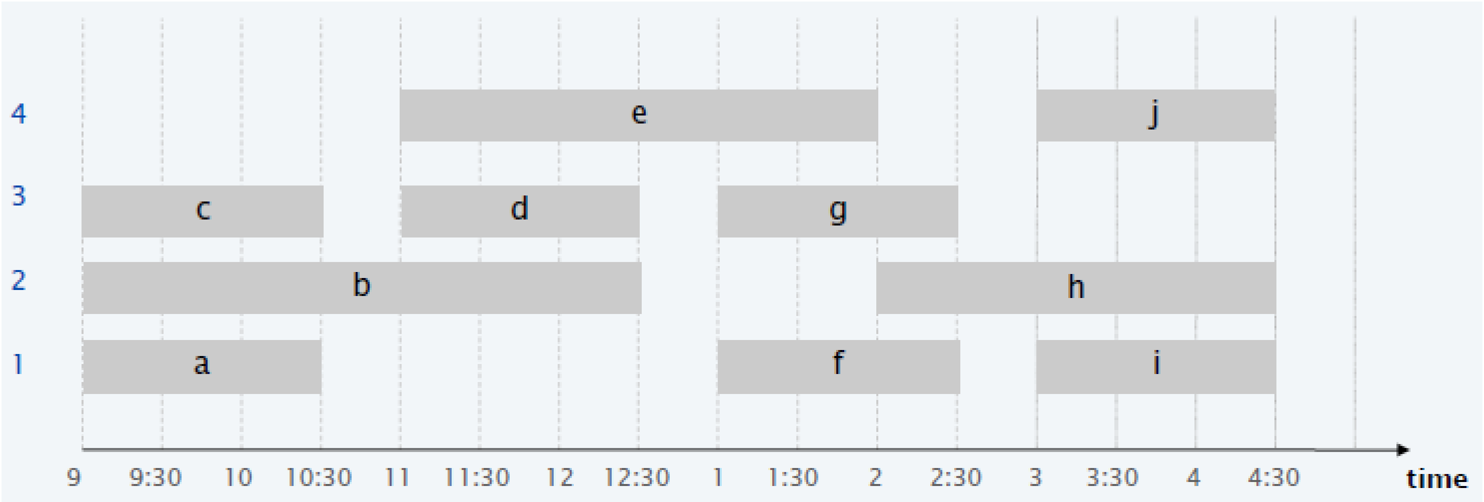


Interval Partitioning

Interval Partitioning.

- 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 lectures occur at the same time in the same room.

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



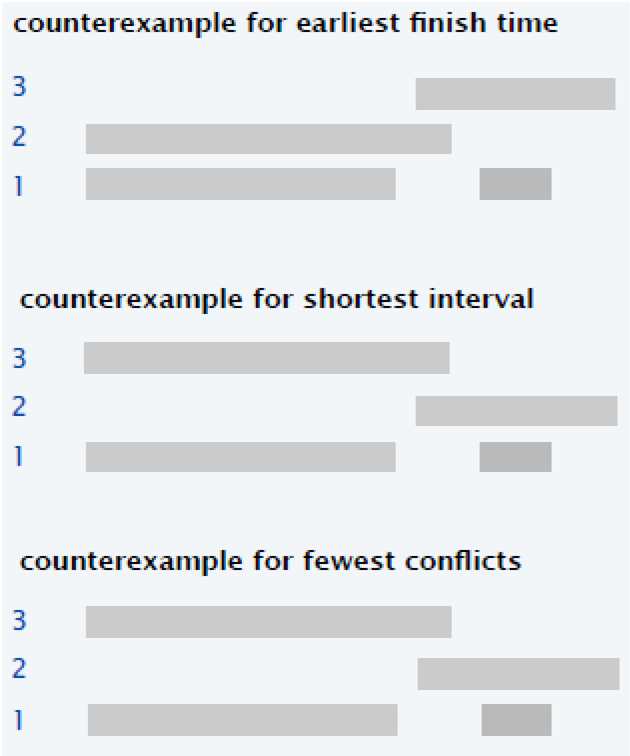
Interval Partitioning: Greedy Algorithm

Greedy template. Consider lectures in some natural order. Assign each lecture to an available classroom; allocate a new classroom if none are available.

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

Interval Partitioning: Greedy Algorithm

Greedy template. Consider lectures in some natural order. Assign each lecture to an available classroom; allocate a new classroom if none are available.



Interval Partitioning: Earliest-Start-Time-First Algorithm

Earliest-Start-Time-First $(n, s_1, s_2, \dots, s_n, f_1, f_2, \dots, f_n)$
Sort lectures by start time so that $s_1 \le s_2 \le \cdots \le s_n$
$d \leftarrow 0$ (the number of allocated classrooms)
for $j = 1$ to n
if lecture j is compatible with some classroom
  Schedule lecture j in any such classroom k.
else
  Allocate a new classroom $d + 1$.
  Schedule lecture j in classroom $d + 1$.
  $d \leftarrow d + 1$
Return schedule.

Interval Partitioning: Earliest-Start-Time-First Algorithm

Proposition. The Earliest-Start-Time-First algorithm can be implemented in $O(nlogn)$ time.

Pf. Store classrooms in a priority queue (key=finish time of its last lecture).

- To determine whether lecture j is compatible with some classroom, compare s_j to key of min classroom k in priority queue.
- To add lecture j to classroom k, increase key of classroom k to f_j.
- Total number of priority queue operation is $O(n)$.
- Sorting by start time takes $O(nlogn)$ time.

Remark. This implementation chooses a classroom k whose finish time of its last lecture is the earliest.

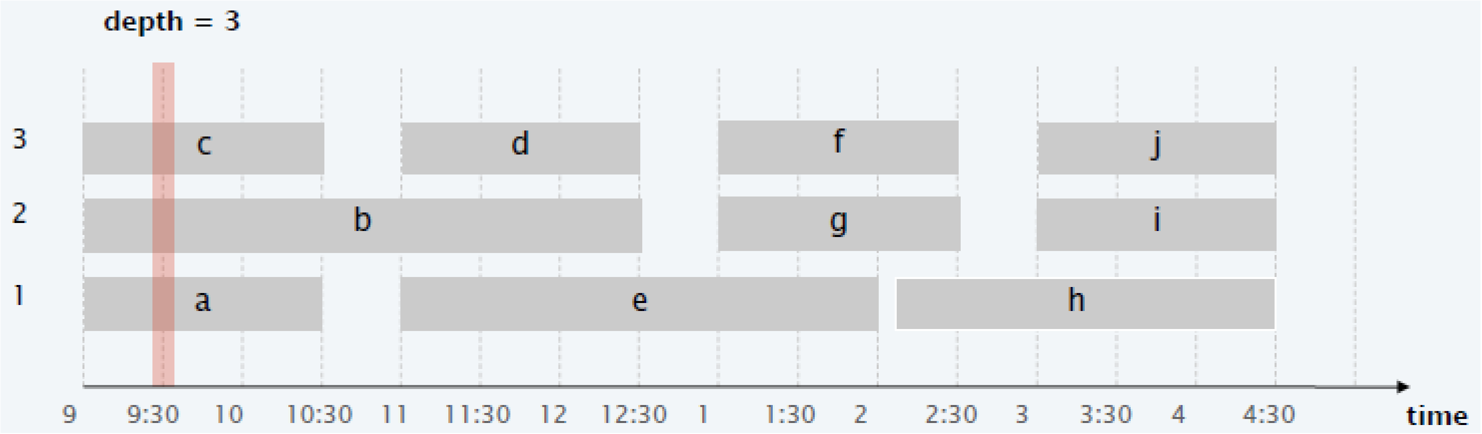
Interval Partitioning: Lower Bound on Optimal Solution

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

Key observation. Number of classrooms needed ≥ depth.

Does minimum number of classrooms needed always equal depth?

Earliest-Start-Time-First algorithm finds a schedule whose number of classrooms equals the depth.



Interval Partitioning: Analysis of Earliest-Start-Time-First Algorithm

Observation. The Earliest-Start-Time-First algorithm never schedules two incompatible lectures in the same classroom.

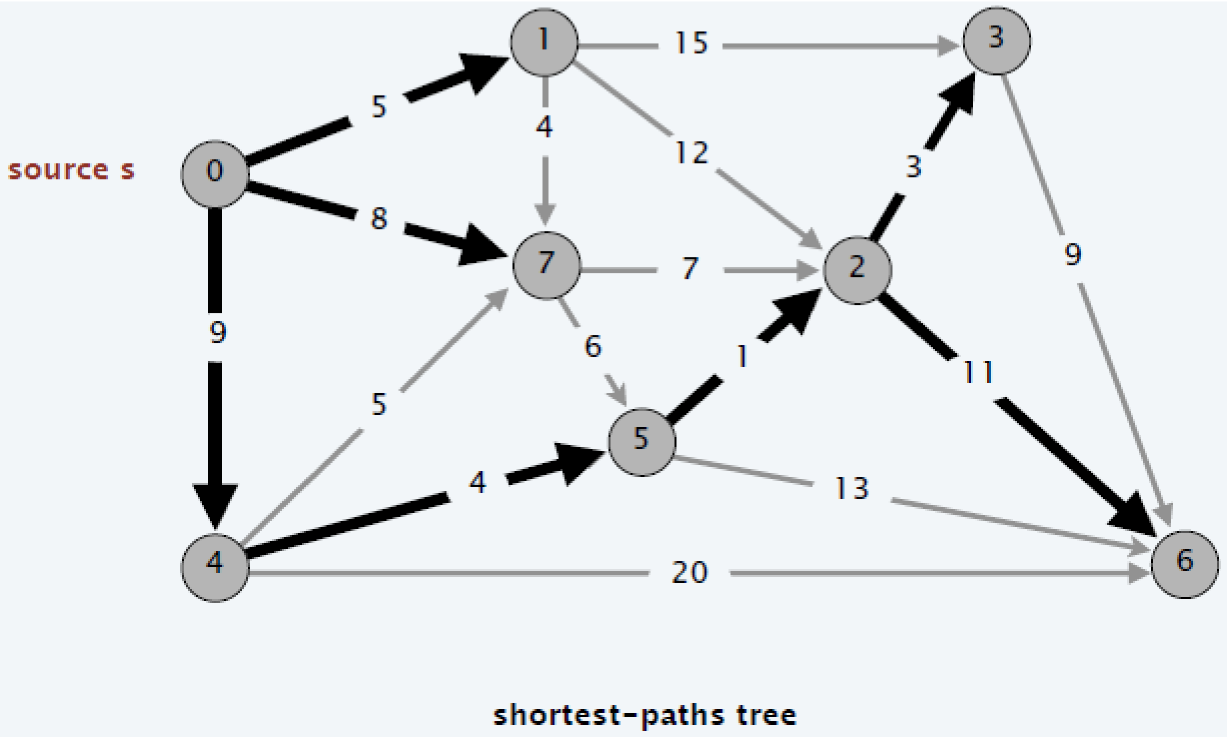
Theorem. Earliest-Start-Time-First algorithm is optimal.

Pf.

- Let $d =$ number of classrooms that the algorithm allocates.
- Classroom d is opened because we needed to schedule a lecture, say j, that is incompatible with all $d - 1$ other classrooms.
- These d lectures each end after s_j.
- 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.

Single-Source Shortest Path Problem

Problem. Given a digraph $G = (V, E)$, edge lengths $l_e \geq 0$, source $s \in V$, find a shortest directed path from s to every node.



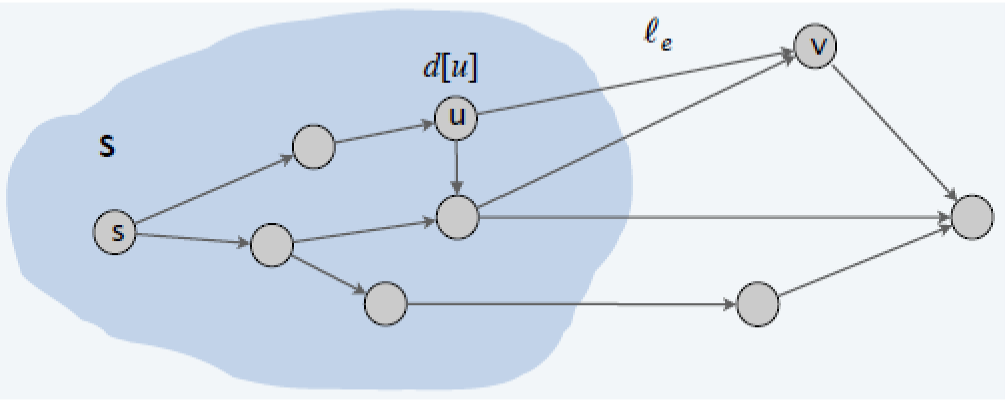
Dijkstra's Algorithm for Single-Source Shortest Paths Problem

Greedy approach. Maintain a set of explored nodes S for which algorithm has determined $d[u]$ = length of a shortest $s \to u$ path.

- Initialize $S \leftarrow \{s\}, d[s] = 0$.
- Repeatedly choose unexplored node $v \notin S$ which minimizes

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

The length of a shortest path from s to some node u in explored part S, followed by a single edge $e = (u, v)$.



Dijkstra's Algorithm for Single-Source Shortest Paths Problem

Greedy approach. Maintain a set of explored nodes S for which algorithm has determined $d[u]$ = length of a shortest $s \to u$ path.

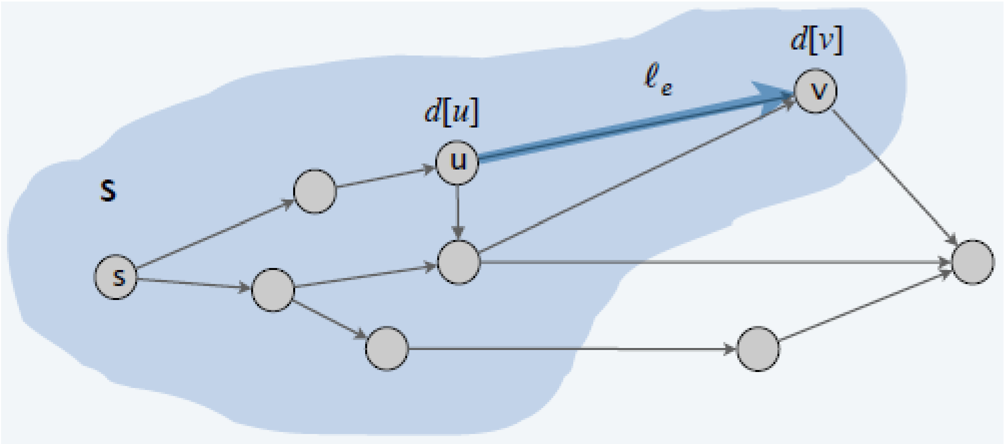
- Initialize $S \leftarrow \{s\}, d[s] = 0$.
- Repeatedly choose unexplored node $v \notin S$ which minimizes

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

add v to S, set $d[v] = \pi(v)$.

The length of a shortest path from s to some node u in explored part S, followed by a single edge $e = (u, v)$.

- To recover path, set $pred[v] \leftarrow e$ that achieves min.



Dijkstra's Algorithm: Proof of Correctness

Invariant. For each node $u \in S$: $d[u]$ = length of a shortest $s \to u$ path.

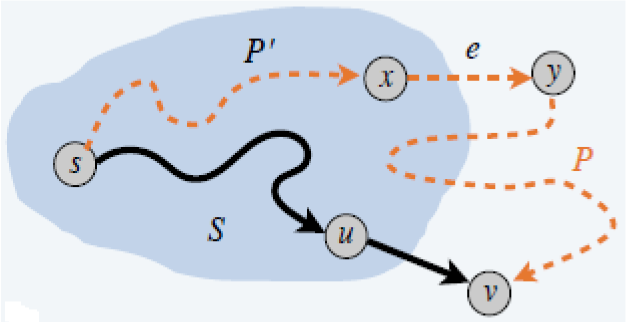
Pf. By induction on $|S|$

Base case: $|S| = 1$ is easy since $S = \{s\}$ and $d[s] = 0$.

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

- Let v be next node added to S, and let (u, v) be the final edge.
- A shortest $s \to u$ path plus (u, v) is an $s \to v$ path of length $\pi(v)$.
- Consider any other $s \to v$ path P. We show that

it is no shorter than $\pi(v)$.



- Let $e = (x, y)$ be the first edge in P that

leaves S, and let P' be the subpath to x.

- The length of P is already $\geq \pi(v)$

as soon as it reaches y:

$$l(P) \geq l(P') + l_e \geq d[x] + l_e \geq \pi(y) \geq \pi(v)$$

Non-negative lengths | Inductive hypothesis | Definition of $\pi(y)$ | Dijkstra chose v instead of y

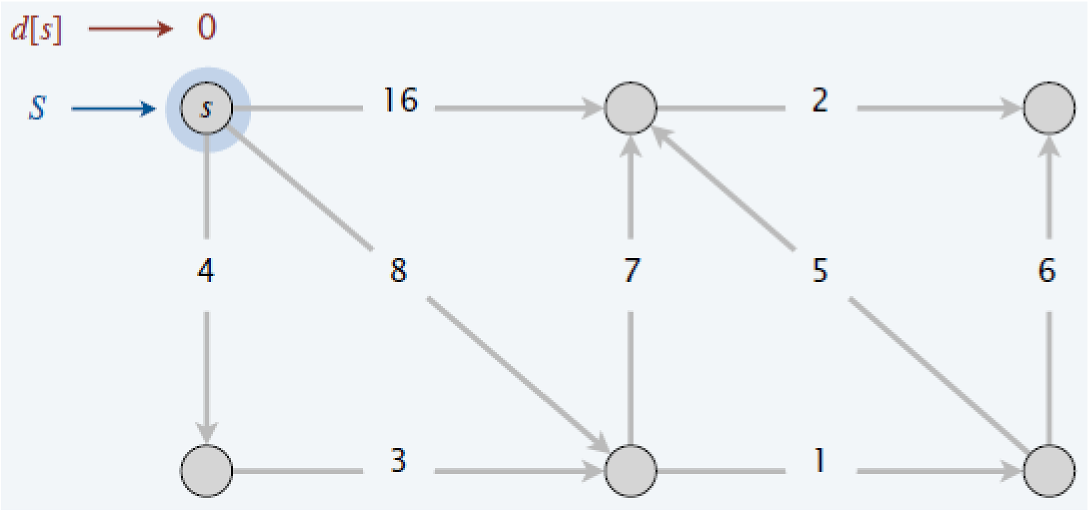
Dijkstra's Algorithm Demo

- Initialize $S \leftarrow \{s\}$ and $d[s] \leftarrow 0$.
- Repeatedly choose unexplored node $v \notin S$ which minimizes

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

Add v to S; set $d[v] \leftarrow \pi(v)$ and $pred(v) \leftarrow argmin$.

The length of a shortest path from s to some node u in explored part S, followed by a single edge $e = (u, v)$.



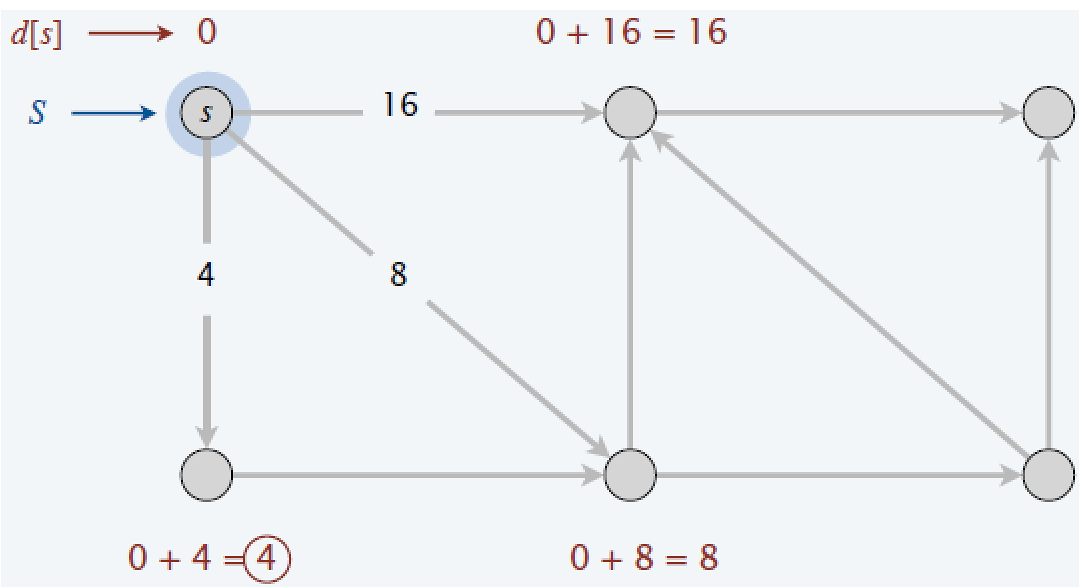
Dijkstra's Algorithm Demo

- Initialize $S \leftarrow \{s\}$ and $d[s] \leftarrow 0$.
- Repeatedly choose unexplored node $v \notin S$ which minimizes

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

Add v to S; set $d[v] \leftarrow \pi(v)$ and $pred(v) \leftarrow argmin$.

The length of a shortest path from s to some node u in explored part S, followed by a single edge $e = (u, v)$.



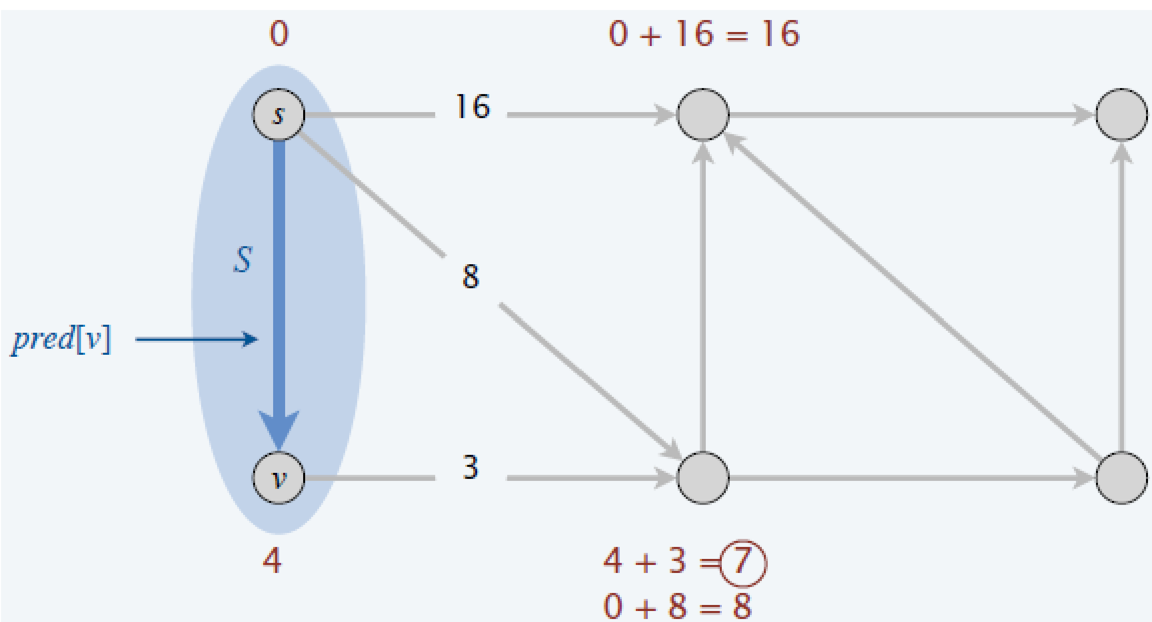
Dijkstra's Algorithm Demo

- Initialize $S \leftarrow \{s\}$ and $d[s] \leftarrow 0$.
- Repeatedly choose unexplored node $v \notin S$ which minimizes

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

Add v to S; set $d[v] \leftarrow \pi(v)$ and $pred(v) \leftarrow$ *argmin.*

The length of a shortest path from s to some node u in explored part S, followed by a single edge $e = (u, v)$.

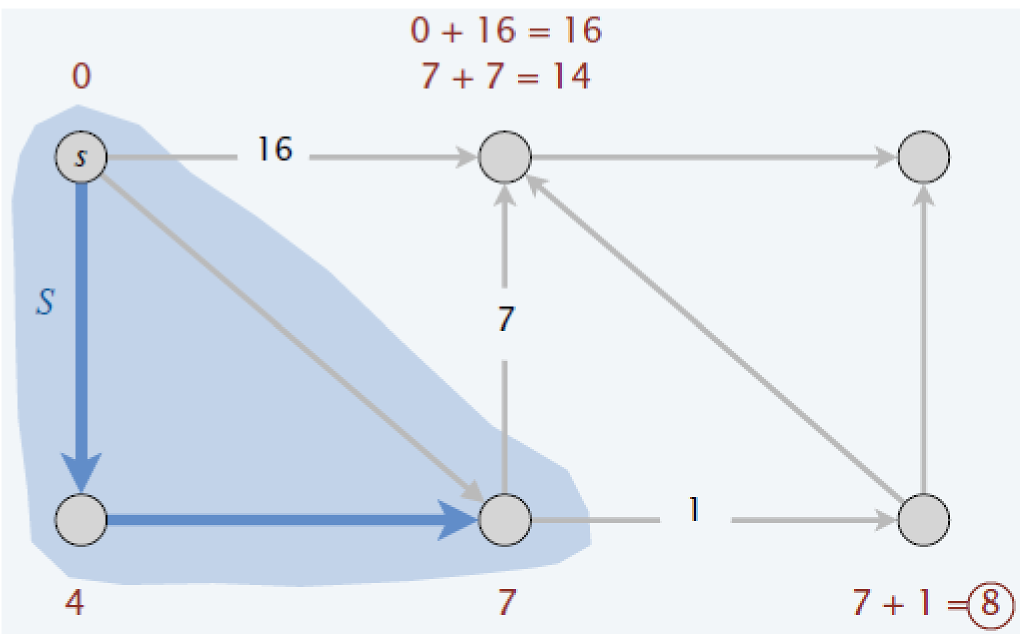


Dijkstra's Algorithm Demo

- Initialize $S \leftarrow \{s\}$ and $d[s] \leftarrow 0$.
- Repeatedly choose unexplored node $v \notin S$ which minimizes

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

Add v to S; set $d[v] \leftarrow \pi(v)$ and $pred(v) \leftarrow argmin$.



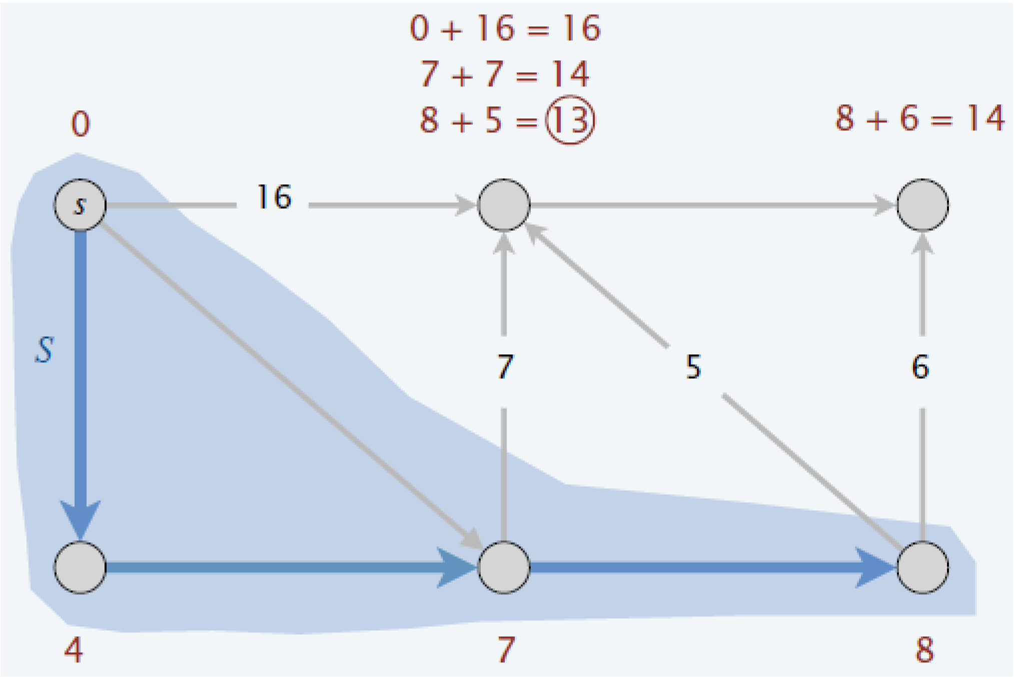
The length of a shortest path from s to some node u in explored part S, followed by a single edge $e = (u, v)$.

Dijkstra's Algorithm Demo

- Initialize $S \leftarrow \{s\}$ and $d[s] \leftarrow 0$.
- Repeatedly choose unexplored node $v \notin S$ which minimizes

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

Add v to S; set $d[v] \leftarrow \pi(v)$ and $pred(v) \leftarrow$ *argmin.*



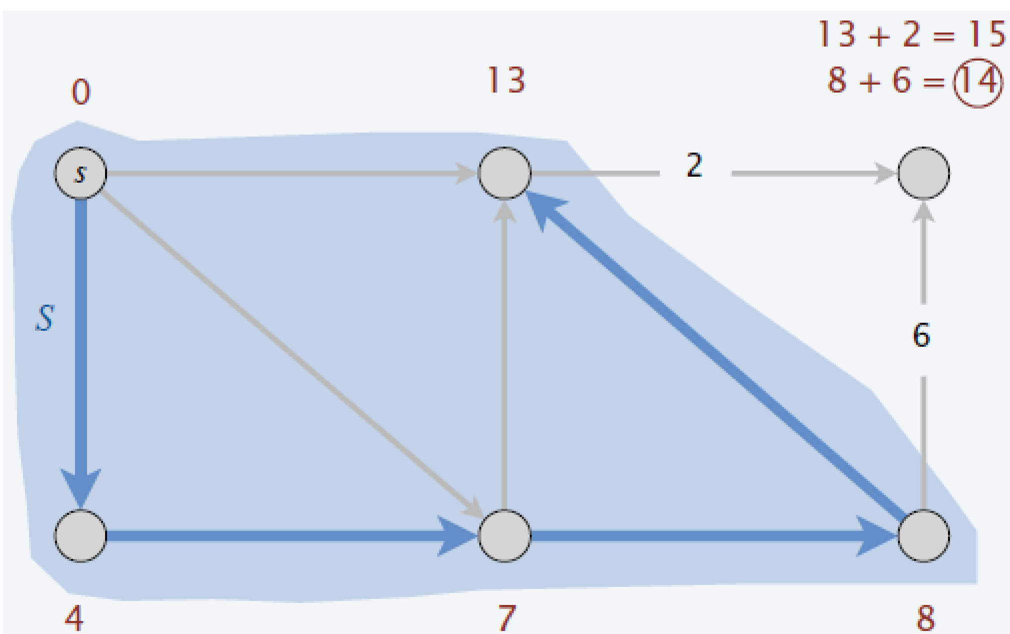
The length of a shortest path from s to some node u in explored part S, followed by a single edge $e = (u, v)$.

Dijkstra's Algorithm Demo

- Initialize $S \leftarrow \{s\}$ and $d[s] \leftarrow 0$.
- Repeatedly choose unexplored node $v \notin S$ which minimizes

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

Add v to S; set $d[v] \leftarrow \pi(v)$ and $pred(v) \leftarrow$ *argmin.*



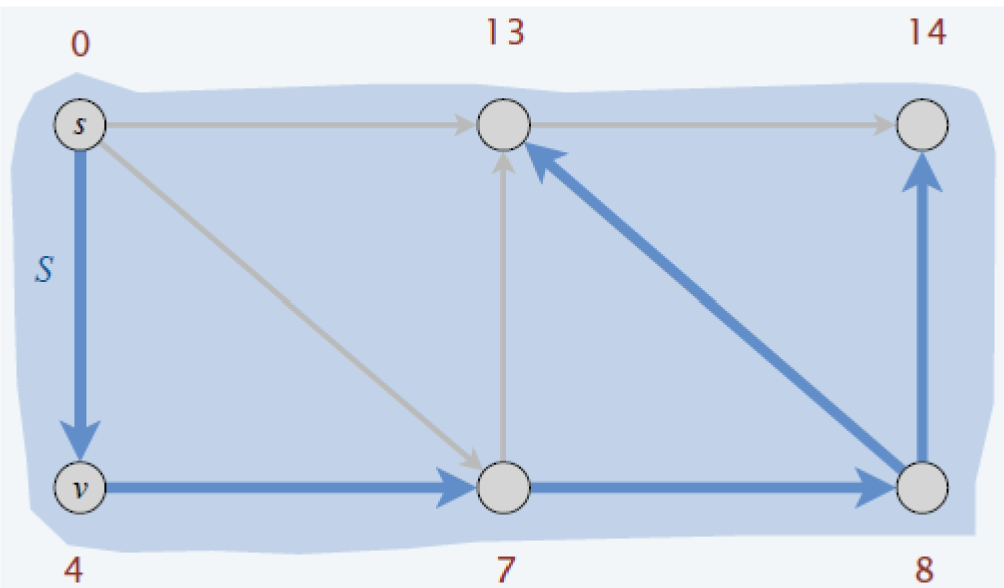
The length of a shortest path from s to some node u in explored part S, followed by a single edge $e = (u, v)$.

Dijkstra's Algorithm Demo

- Initialize $S \leftarrow \{s\}$ and $d[s] \leftarrow 0$.
- Repeatedly choose unexplored node $v \notin S$ which minimizes

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

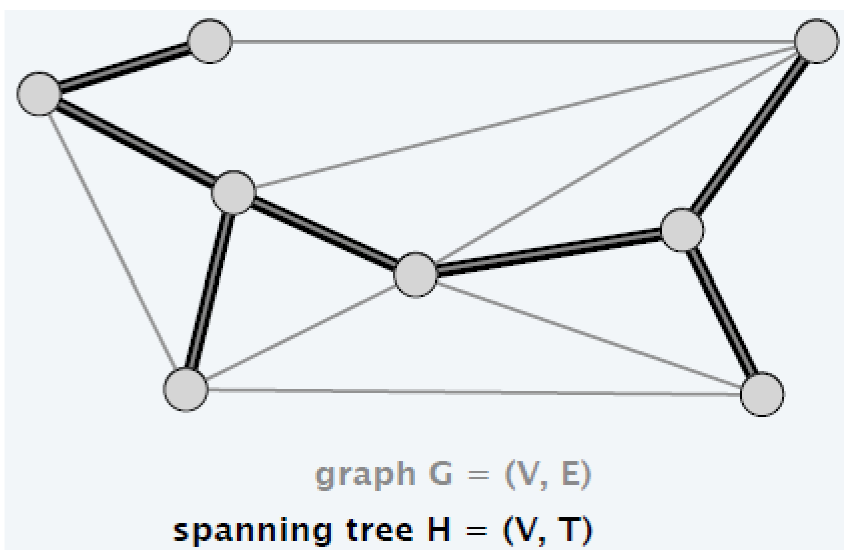
Add v to S; set $d[v] \leftarrow \pi(v)$ and $pred(v) \leftarrow$ *argmin.*



The length of a shortest path from s to some node u in explored part S, followed by a single edge $e = (u, v)$.

Spanning Tree Definition

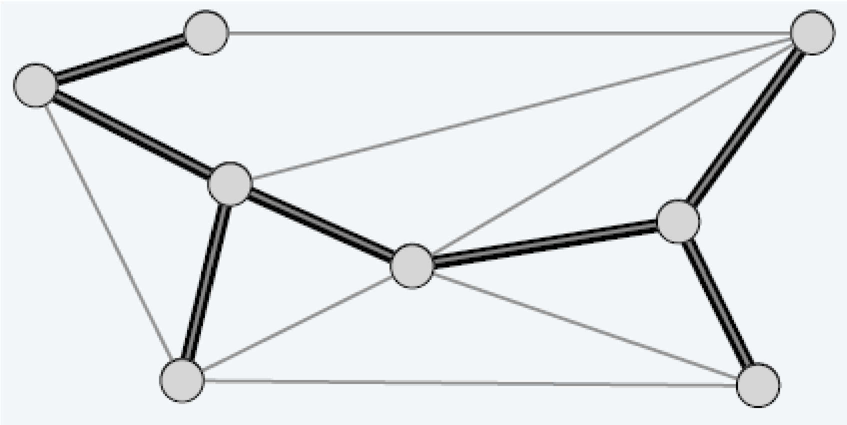
Def. Let $H = (V, T)$ be a subgraph of an undirected graph $G = (V, E)$. H is a spanning tree of G if H is both acyclic and connected.



Spanning Tree Properties

Proposition. Let $H = (V, T)$ be a subgraph of an undirected graph $G = (V, E)$. Then, the following are equivalent:

- H is a spanning tree of G.
- H is acyclic and connected.
- H is connected and has $n - 1$ edges.
- H is acyclic and has $n - 1$ edges.
- H is minimally connected: removal of any edge disconnects it.
- H is maximally acyclic: addition of any edge creates a cycle.
- H has a unique simple path between every pair of nodes.

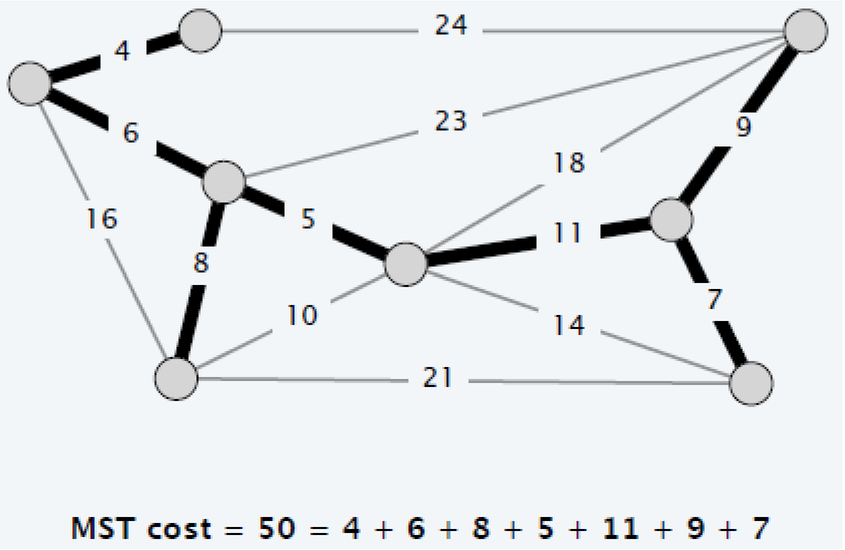


graph G = (V, E)

spanning tree H = (V, T)

Minimum Spanning Tree (MST)

Def. Given a connected, undirected graph $G = (V, E)$ with edge costs c_e, a minimum spanning tree (V, T) is a spanning tree of G such that the sum of the edge costs in T is minimized.



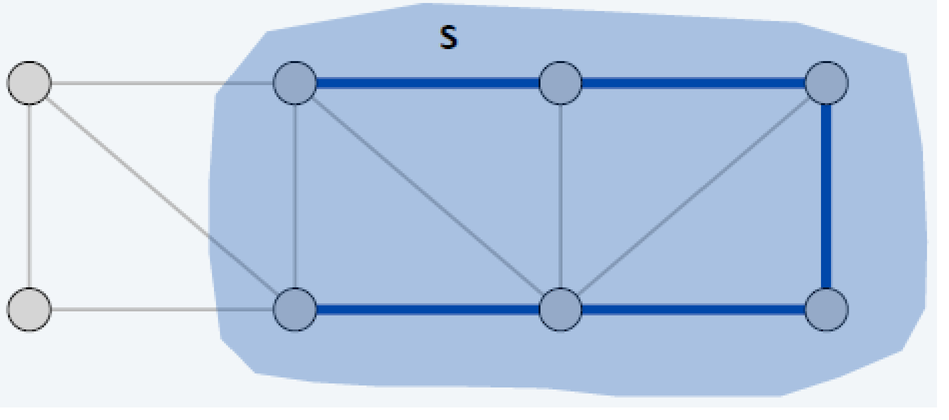
Prim's Algorithm

Initialize $S =$ any node, $T = \emptyset$.

Repeat $n - 1$ times:

- Add to T a min-weight edge with one endpoint in S.
- Add new node to S.

Theorem. Prim's algorithm computes an MST.



Prim's Algorithm: Implementation

Implementation almost identical to Dijkstra's algorithm.

Prim (V, E, c)

Create an empty priority queue PQ.

$S \leftarrow \emptyset, T \leftarrow \emptyset.$

$s \leftarrow$ any node in V.

for each $v \neq s$: $\pi[v] \leftarrow \infty, pred[v] \leftarrow null$; $\pi[s] \leftarrow 0$.

for each $v \in V$: Insert $(PQ, v, \pi[v])$,

while Is-Not-Empty (PQ)

  $u \leftarrow$ Del-Min (PQ).

  $S \leftarrow S \cup \{u\}, T \leftarrow T \cup \{pred[u]\}$.

  for each edge $e = (u, v) \in E$ with $v \notin S$:

    if $c_e < \pi[v]$

      Decrease-Key (PQ, v, c_e).

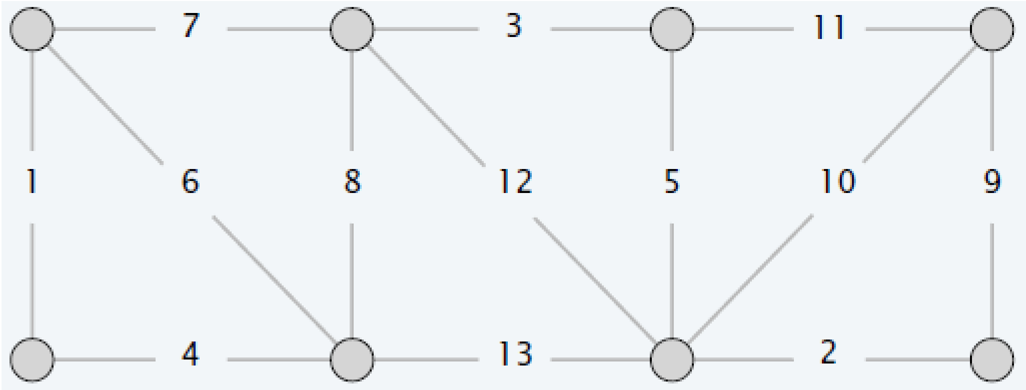
     $\pi[v] \leftarrow c_e; pred[v] \leftarrow e$.

$\pi[v]$ = weight of cheapest known edge between v and S.

Prim's Algorithm Demo

Initialize S=any node, $T = \emptyset$
Repeat n-1 times:

- Add to T a min-weight edge with one endpoint in S.
- Add new node to S.

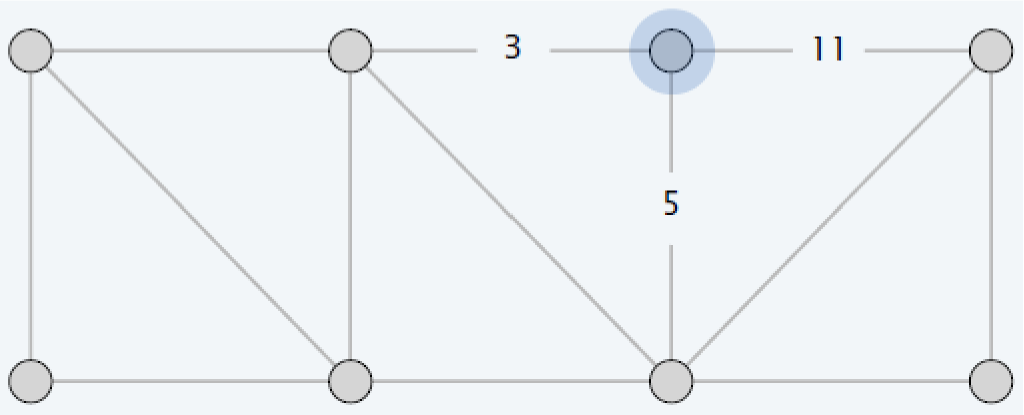


Prim's Algorithm Demo

Initialize S=any node, $T = \emptyset$

Repeat n-1 times:

- Add to T a min-weight edge with one endpoint in S.
- Add new node to S.

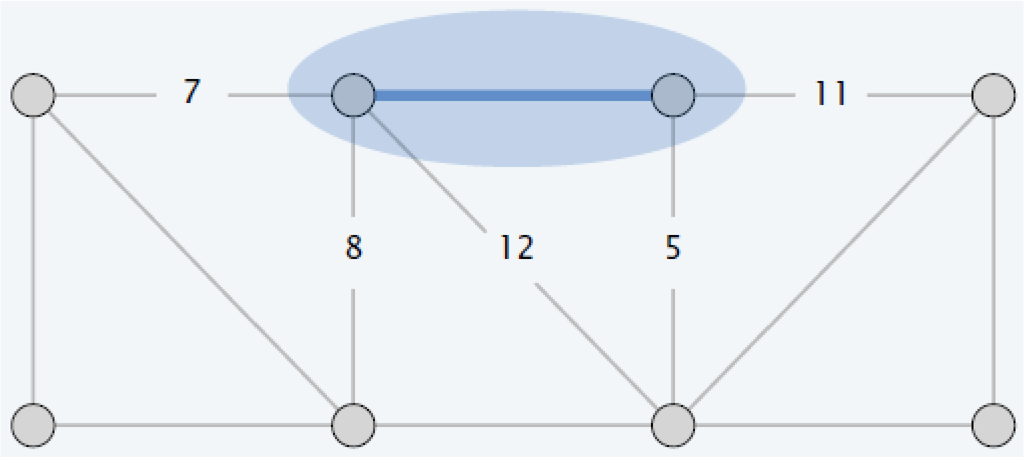


Prim's Algorithm Demo

Initialize S=any node, $T = \emptyset$

Repeat n-1 times:

- Add to T a min-weight edge with one endpoint in S.
- Add new node to S.

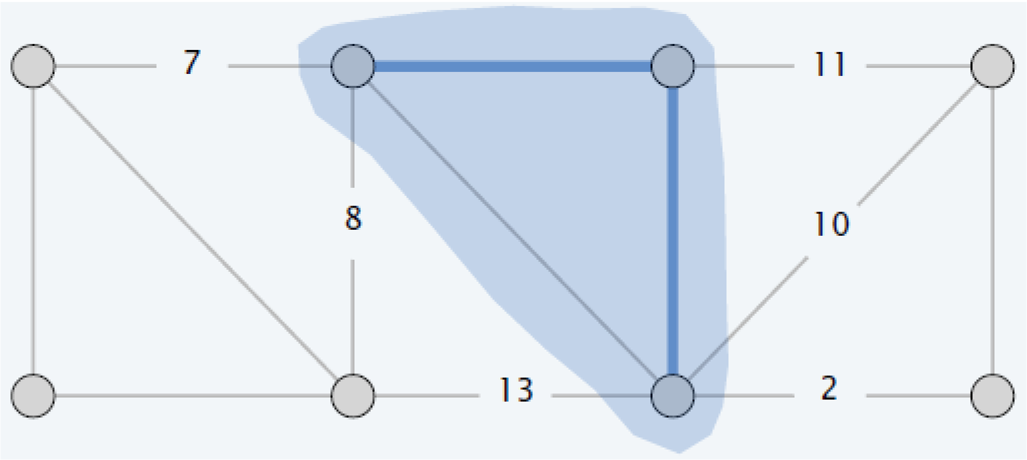


Prim's Algorithm Demo

Initialize S=any node, $T = \emptyset$

Repeat n-1 times:

- Add to T a min-weight edge with one endpoint in S.
- Add new node to S.

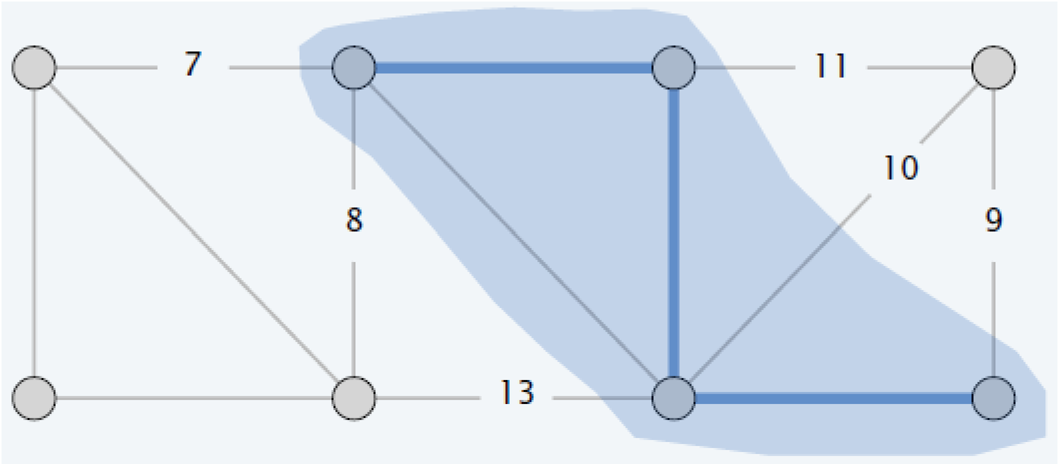


Prim's Algorithm Demo

Initialize S=any node, $T = \emptyset$

Repeat n-1 times:

- Add to T a min-weight edge with one endpoint in S.
- Add new node to S.

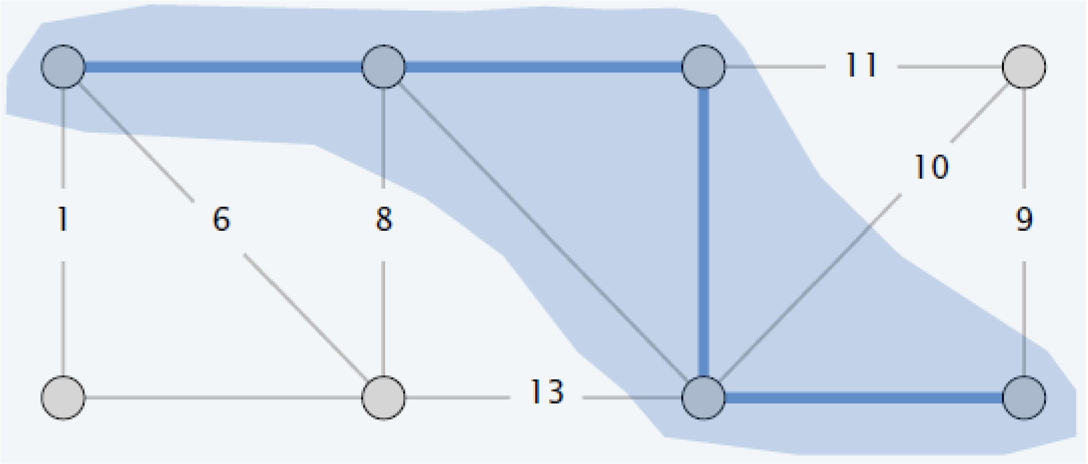


Prim's Algorithm Demo

Initialize S=any node, $T = \emptyset$

Repeat n-1 times:

- Add to T a min-weight edge with one endpoint in S.
- Add new node to S.

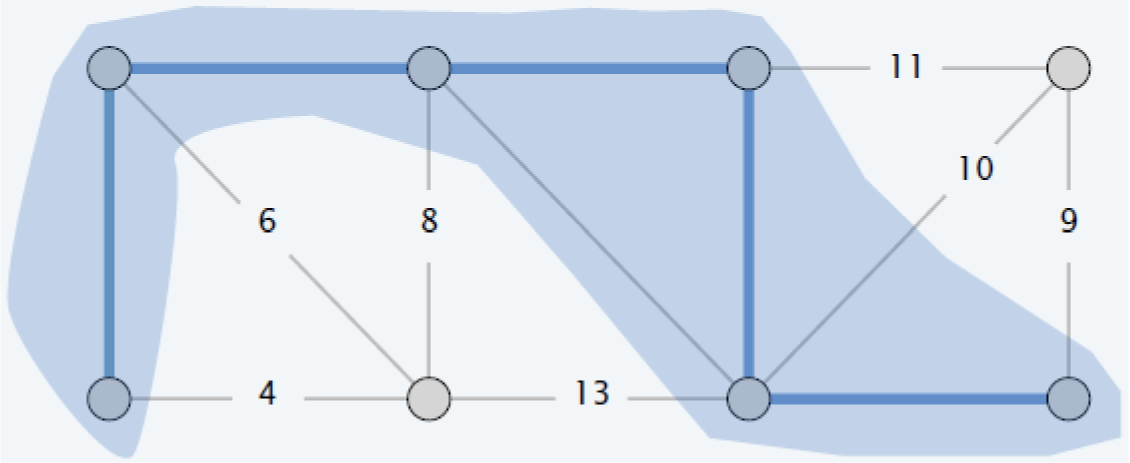


Prim's Algorithm Demo

Initialize S=any node, $T = \emptyset$

Repeat n-1 times:

- Add to T a min-weight edge with one endpoint in S.
- Add new node to S.

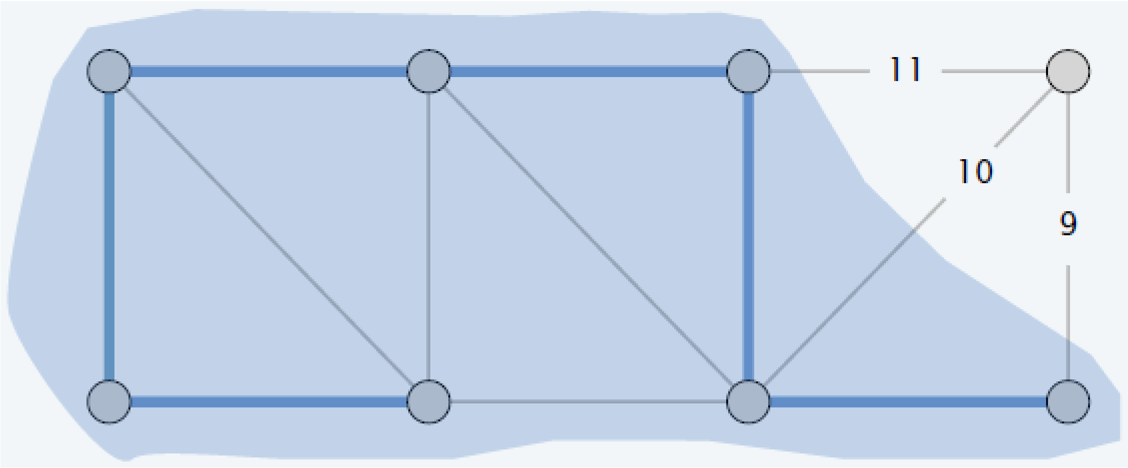


Prim's Algorithm Demo

Initialize S=any node, $T = \emptyset$

Repeat n-1 times:

- Add to T a min-weight edge with one endpoint in S.
- Add new node to S.

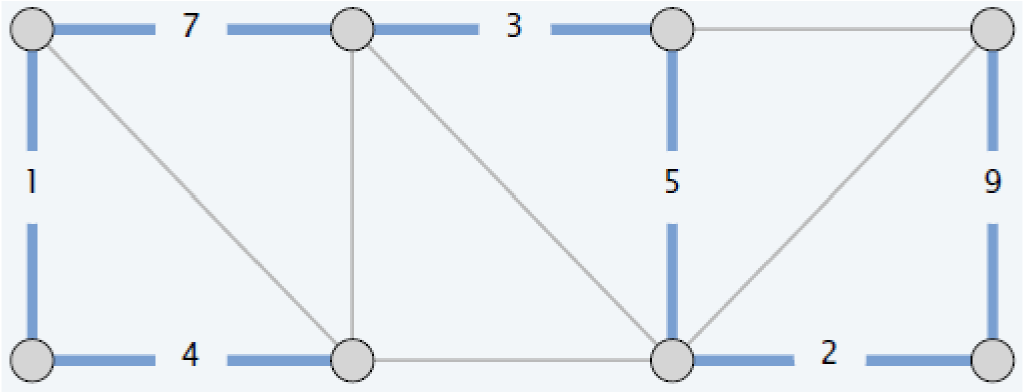


Prim's Algorithm Demo

Initialize S=any node, $T = \emptyset$

Repeat n-1 times:

- Add to T a min-weight edge with one endpoint in S.
- Add new node to S.

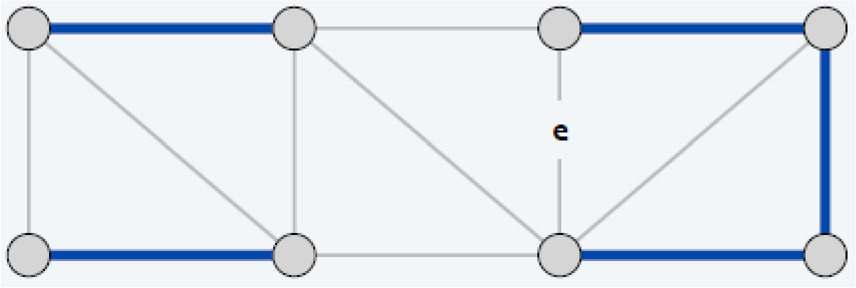


Kruskal’s Algorithm

Consider edges in ascending order of weight:

- Add to tree unless it would create a cycle.

Theorem. Kruskal’s algorithm computes an MST.



Kruskal's Algorithm: Implementation

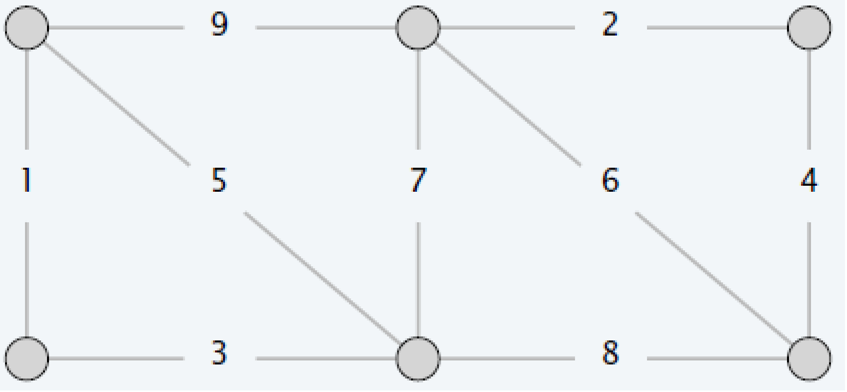
- Sort edges by weights.
- Use union-find data structure to dynamically maintain connected components.

Kruskal (V, E, c)
Sort m edges by weight so that $c(e_1) \leq c(e_1) \leq \cdots \leq c(e_m)$.
$T \leftarrow \emptyset$.
for each $v \in V$: Make-Set (v).
for $i = 1$ to m
    $(u, v) \leftarrow e_i$.
    if Find-Set $(u) \neq$ Find-Set (v) ← **are u and v in same component?**
        $T \leftarrow T \cup \{e_i\}$.
        Union (u, v). ← **make u and v in same component**
Return T.

Kruskal's Algorithm Demo

Consider edges in ascending order of weight:

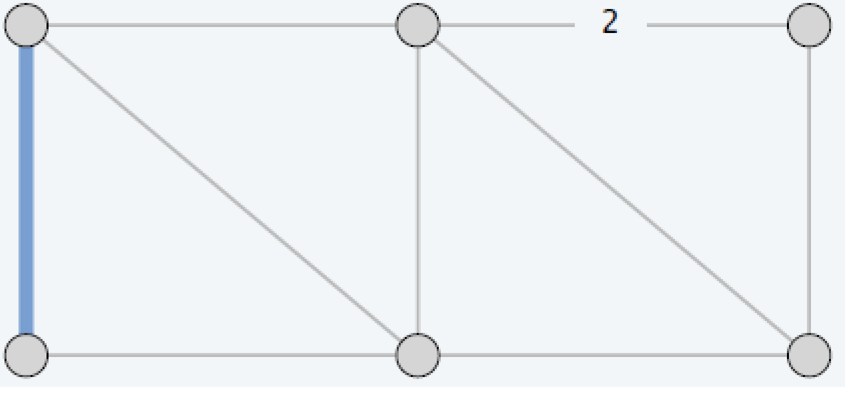
- Add to T unless it would create a cycle.



Kruskal’s Algorithm Demo

Consider edges in ascending order of weight:

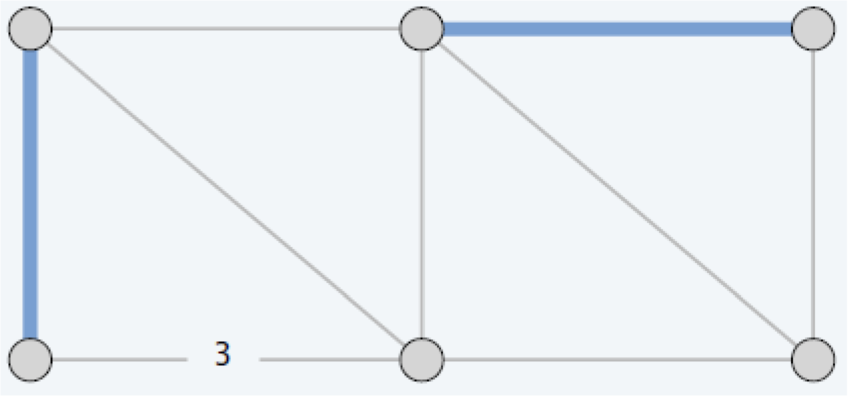
- Add to T unless it would create a cycle.



Kruskal's Algorithm Demo

Consider edges in ascending order of weight:

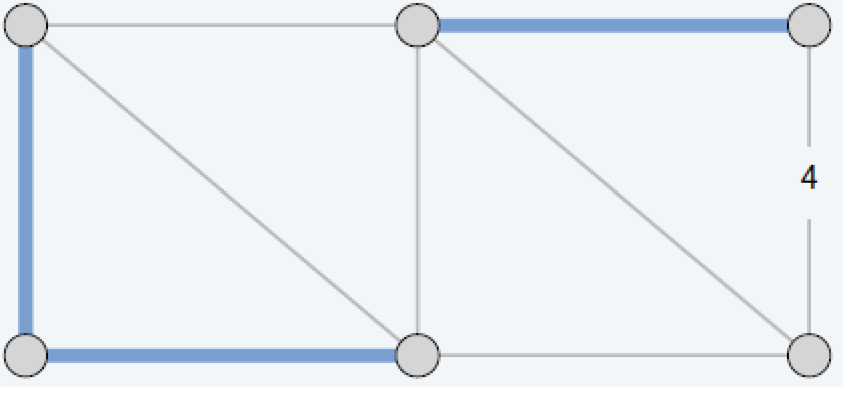
- Add to T unless it would create a cycle.



Kruskal's Algorithm Demo

Consider edges in ascending order of weight:

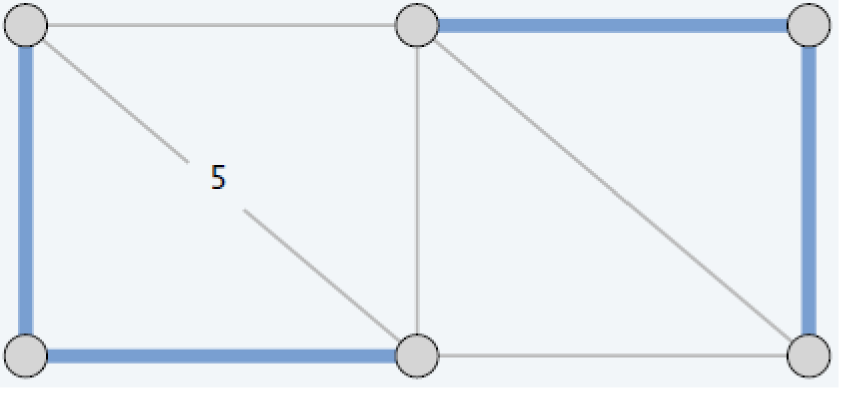
- Add to T unless it would create a cycle.



Kruskal’s Algorithm Demo

Consider edges in ascending order of weight:

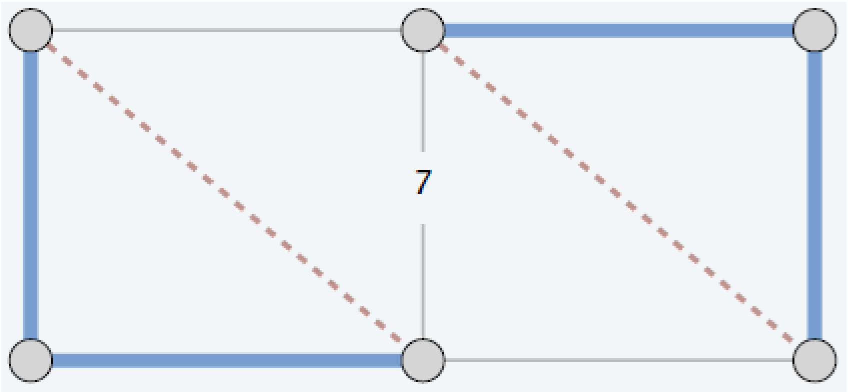
- Add to T unless it would create a cycle.



Kruskal’s Algorithm Demo

Consider edges in ascending order of weight:

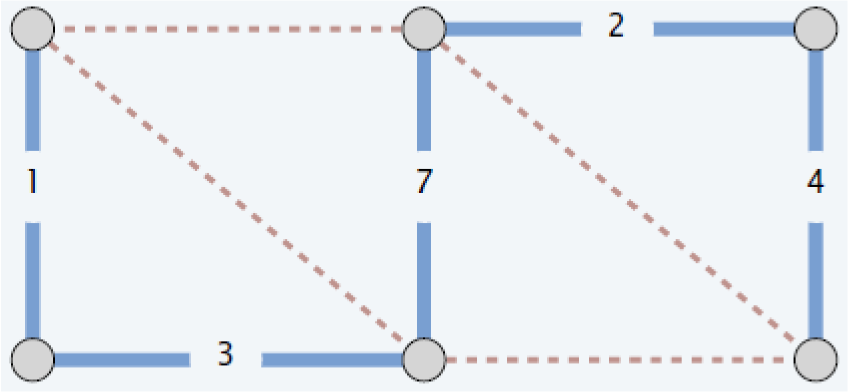
- Add to T unless it would create a cycle.



Kruskal's Algorithm Demo

Consider edges in ascending order of weight:

- Add to T unless it would create a cycle.



Review:
Linear Programming

Standard Form

"Standard form" of a linear program.

- Input: real numbers a_{ij}, c_j, b_i.
- Output: real numbers x_j.
- $n = \#$ decision variables, $m = \#$ constraints.
- Maximize linear objective function subject to linear equalities.

$$
\begin{aligned}
\max \sum_{j=1}^{n} c_j x_j \\
s.t. \sum_{j=1}^{n} a_{ij} x_j = b_i \quad 1 \le i \le m \\
x_j \ge 0 \quad 1 \le j \le n
\end{aligned}
$$

Equivalent Forms

Easy to convert variants to standard form.

$$\max c^T x$$
$$s.t.\ \ Ax = b$$
$$x \geq 0$$

- Less than to equality. $x + 2y - 3z \leq 17 \ \rightarrow x + 2y - 3z + s = 17, s \geq 0$
- Greater than to equality. $x + 2y - 3z \geq 17 \ \rightarrow x + 2y - 3z - s = 17, s \geq 0$
- Min to max. $\min x + 2y - 3z \ \rightarrow \max -x - 2y + 3z$
- Unrestricted to nonnegative. x unrestricted $\rightarrow x = x^+ - x^-, x^+ \geq 0, x^- \geq 0$

Brewery Problem: Converting to Standard Form

Original input.

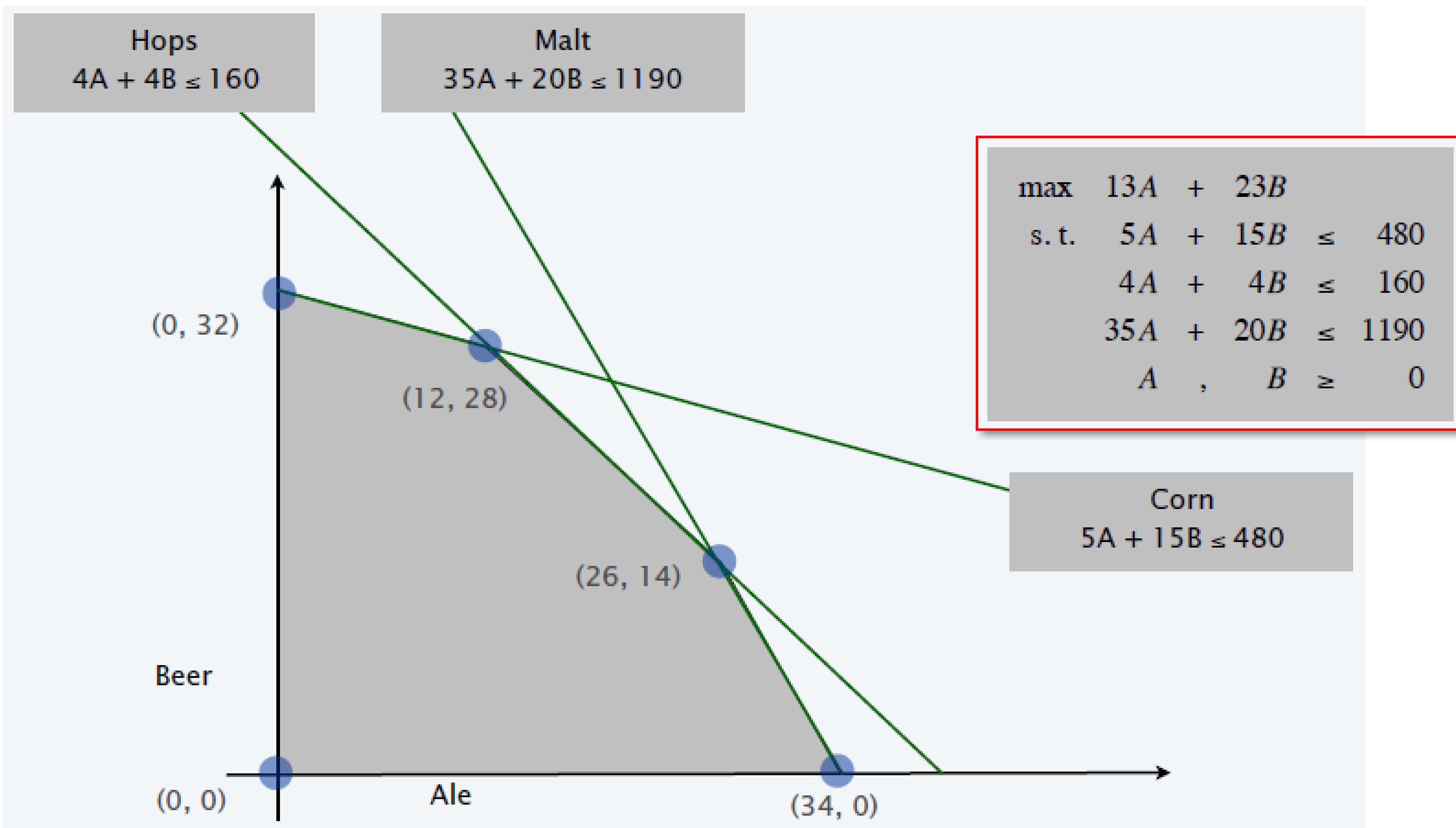
$$
\begin{array}{lrcrcr}
\max & 13A & + & 23B & & \\
\text{s.t.} & 5A & + & 15B & \le & 480 \\
 & 4A & + & 4B & \le & 160 \\
 & 35A & + & 20B & \le & 1190 \\
 & A & , & B & \ge & 0
\end{array}
$$

Standard form.

- Add slack variable for each inequality.
- Now a 5-dimensional problem.

$$
\begin{array}{lrcrcccccccr}
\max & 13A & + & 23B & & & & & & & & \\
\text{s.t.} & 5A & + & 15B & + & S_C & & & & & = & 480 \\
 & 4A & + & 4B & & & + & S_H & & & = & 160 \\
 & 35A & + & 20B & & & & & + & S_M & = & 1190 \\
 & A & , & B & , & S_C & , & S_H & , & S_M & \ge & 0
\end{array}
$$

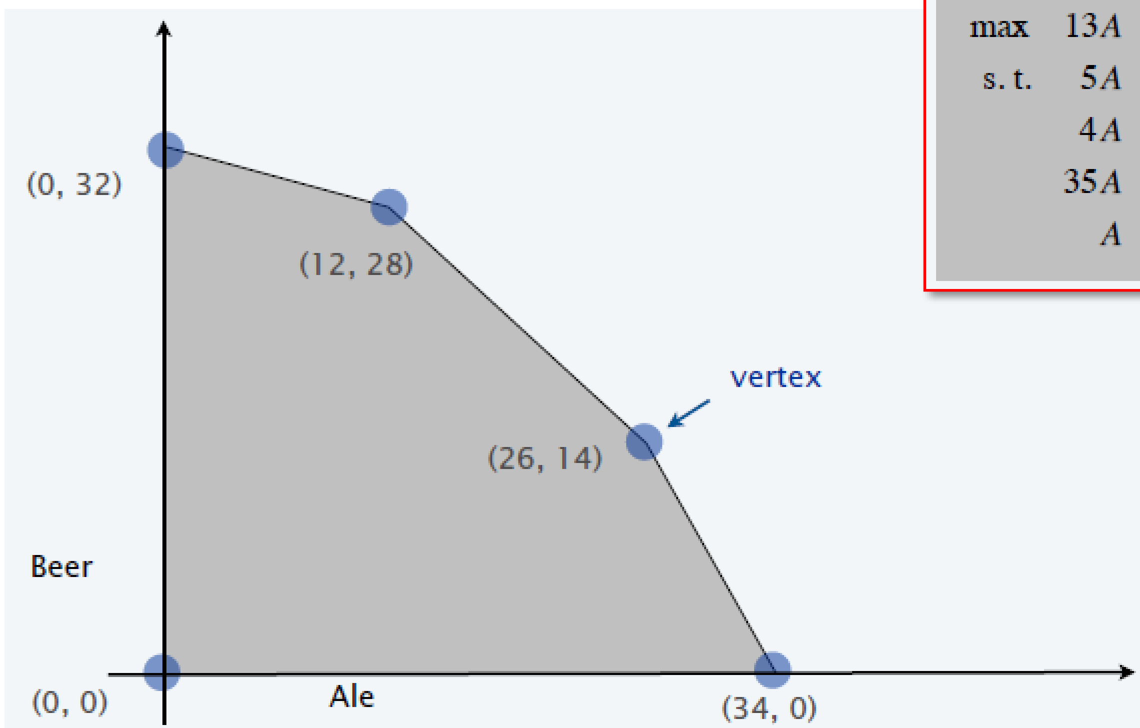
Brewery Problem: Feasible Region



Brewery Problem: Geometry

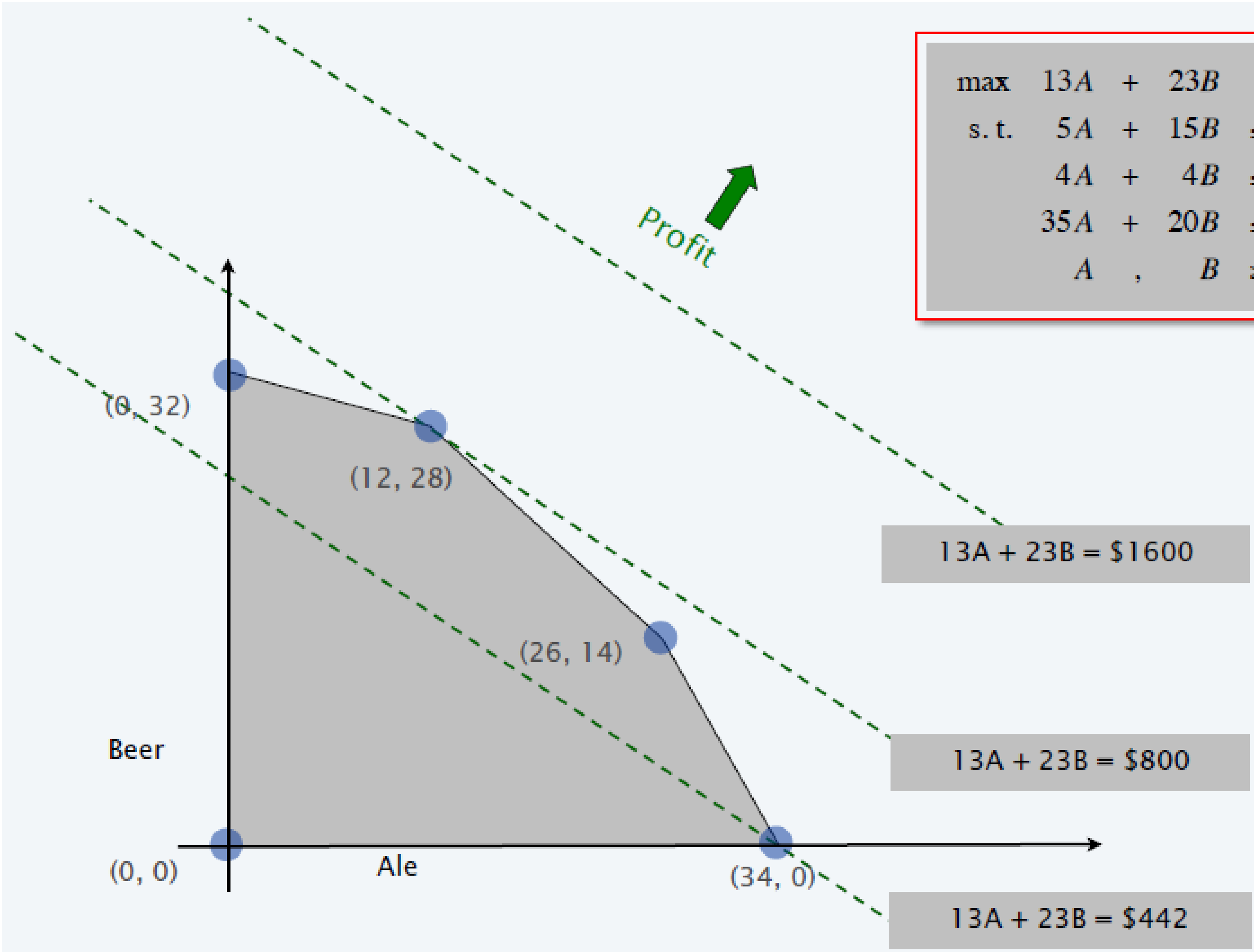
Brewery problem observation. Regardless of objective function coefficients, an optimal solution occurs at a vertex.

$$
\begin{array}{rrcrcr}
\max & 13A & + & 23B & & \\
\text{s. t.} & 5A & + & 15B & \leq & 480 \\
& 4A & + & 4B & \leq & 160 \\
& 35A & + & 20B & \leq & 1190 \\
& A & , & B & \geq & 0
\end{array}
$$



Brewery Problem: Objective Function

$$
\begin{array}{lrcrcr}
\max & 13A & + & 23B & & \\
\text{s.t.} & 5A & + & 15B & \le & 480 \\
 & 4A & + & 4B & \le & 160 \\
 & 35A & + & 20B & \le & 1190 \\
 & A & , & B & \ge & 0
\end{array}
$$



Variant Tableau

The constraints are a linear system including m equations and n variables. m of the variables can be evaluated in terms of the other $n - m$ varaibles

$$x_1 = b_1 - a_{1,m+1}x_{m+1} - \cdots - a_{1,n}x_n$$

$$x_2 = b_2 - a_{2,m+1}x_{m+1} - \cdots - a_{2,n}x_n$$

$$\ldots\ldots$$

$$x_m = b_m - a_{m,m+1}x_{m+1} - \cdots - a_{m,n}x_n$$

Objective function $z = \sum_{j=1}^{n} c_j x_j$

$$= \sum_{i=1}^{m} c_i b_i + \sum_{j=m+1}^{n} (c_j - \sum_{i=1}^{m} c_i a_{ij}) x_j.$$

Let $z^0 = \sum_{i=1}^{m} c_i b_i$, $\sigma_j = c_j - \sum_{i=1}^{m} c_i a_{ij}$, and we have

$$z = z^0 + \sum_{j=m+1}^{n} \boxed{\sigma_j} x_j$$

indicator

Variant Tableau

c_j		c_1	c_2	$\ldots$	c_m	c_{m+1}	$\ldots$	c_n	b	θ
C_B	X_B	x_1	x_2	$\ldots$	x_m	x_{m+1}	$\ldots$	x_n		
c_1	x_1	1	0	$\ldots$	0	$a'_{1,m+1}$	$\ldots$	a'_{1n}	b'_1	
c_2	x_2	0	1	$\ldots$	0	$a'_{2,m+1}$	$\ldots$	a'_{2n}	b'_2	
$\ldots$	$\ldots$				……				$\ldots$	
c_m	x_m	0	0	$\ldots$	1	$a'_{m,m+1}$	$\ldots$	a'_{mn}	b'_m	
σ_j		0	0	$\ldots$	0	$c_{m+1} - \sum_{i=1}^{m} c_i a'_{i,m+1}$				

Variant Tableau

To solve a linear programming problem, use the following steps:

1. Convert each inequality in the set of constraints to an equation by adding slack variables.
2. Create the initial simplex tableau.
3. Select the pivot column. (The column with the “most positive value” element in the last row.)
4. Select the pivot row. (The row with the smallest non-negative result when the last element in the row is divided by the corresponding in the pivot column.)
5. Use elementary row operations calculate new values for the pivot row so that the pivot is 1.
6. Use elementary row operations to make all numbers in the pivot column equal to 0 except for the pivot number. If all entries in the bottom row are non-positive, this the final tableau. If not, go back to step 3.
7. If you obtain a final tableau, then the linear programming problem has a maximum solution.

Variant Tableau: An Example

$$\max \quad z = 2x_1 + 3x_2$$
$$s.t.\begin{cases} 2x_1 + x_2 \le 4 \\ x_1 + 2x_2 \le 5 \\ x_1, x_2 \ge 0 \end{cases}$$

$$\Rightarrow$$

$$\max \quad z = 2x_1 + 3x_2$$
$$s.t.\begin{cases} 2x_1 + x_2 + x_3 = 4 \\ x_1 + 2x_2 + x_4 = 5 \\ x_1, x_2, x_3, x_4 \ge 0 \end{cases}$$

Variant Tableau: An Example

Pivot column. The column of the tableau representing the variable to be entered into the solution mix.

Pivot row. The row of the tableau representing the variable to be replaced in the solution mix.

Basic variable. Variables in the solution mix.

Initial tableau

Pivot column

Min ratio rule

Pivot row

c_j		2	3	0	0	b	θ
C_B	X_B	x_1	x_2	x_3	x_4		
0	x_3	2	1	1	0	4	4/1
0	x_4	1	2	0	1	5	5/2
	σ_j	2	3	0	0		

Variant Tableau: An Example

c_j		2	3	0	0	b	θ
C_B	X_B	x_1	x_2	x_3	x_4		
0	x_3	2	1	1	0	4	4/1
0	x_4	1	2	0	1	5	5/2
σ_j		2	**3**	0	0		

- Since the entry 3 is the most positive entry in the last row of the tableau, the second column in the tableau is the pivot column.
- Divide each positive number of the pivot column into the corresponding entry in the column of constants. The ratio 5/2 is less then the ratio 4/1, so row 2 is the pivot row.

Variant Tableau: An Example

c_j		2	3	0	0	b	θ
C_B	X_B	x_1	x_2	x_3	x_4		
0	x_3	3/2	0	1	-1/2	3/2	1
3	x_2	1/2	1	0	1/2	5/2	5
σ_j		**1/2**	0	0	-3/2		

- Since the entry 1/2 is the most positive entry in the last row of the tableau, the first column in the tableau is the pivot column.
- Divide each positive number of the pivot column into the corresponding entry in the column of constants. The ratio 3/2 is less then the ratio 5/2, so row 1 is the pivot row.

Variant Tableau: An Example

c_j		2	3	0	0	b	θ
C_B	X_B	x_1	x_2	x_3	x_4		
2	x_1	1	0	2/3	-1/3	1	
3	x_2	0	1	-1/3	2/3	2	
σ_j		0	0	-1/3	-4/3		

- The last row of the tableau contains no positive numbers, so an optimal solution has been reached.

Review:
Dynamic Programming

Algorithmic Paradigms

Greedy. Build up a solution incrementally, myopically optimizing some local criterion.

Divide-and-conquer. Break up a problem into independent sub-problems, solve each sub-problem, and combine solutions to sub-problems to form solution to original problem.

Dynamic programming. Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems (caching away intermedia results in a table for later reuse).

Knapsack Problem

- Given n items and a "Knapsack".
- Item i weights $w_i > 0$ and has value $v_i > 0$.
- Knapsack has weight capacity of W.
- Goal: pack knapsack so as to maximize total value.

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

knapsack instance
(weight limit W = 11)

Ex. {1,2,5} has value 35 and weight 10.
Ex. {3,4} has value 40 and weight 11.
Ex. {3,5} has value 46 but exceeds weight limit.

Greedy by value. Repeatedly add item with maximum v_i.
Greedy by weight. Repeatedly add item with maximum w_i.
Greedy by ratio. Repeatedly add item with maximum v_i/w_i.

Observation. None of greedy algorithms is optimal.

Dynamic Programming: False Start

Def. $OPT(i)$ = max-profit subset of items 1,2, … i.
Goal. $OPT(n)$.

Case 1. $OPT(i)$ does not select item i.

- OPT selects best of {1, 2, …, $i - 1$}.

Case 2. $OPT(i)$ selects item i.

- Selecting item i does not immediately imply that we will have to reject other items.
- Without knowing what other items were selected before i, we don't even know if we have enough room for i.

Conclusion. Need more sub-problems.

Dynamic Programming: Adding a New Variable

Def. $OPT(i, w)$ = max-profit subset of items 1,2, … i with weight limit w.

Goal. $OPT(n, W)$.

Case 1. $OPT(i, w)$ does not select item i.

- $OPT(i, w)$ selects best of {1, 2, …, $i - 1$} using weight limit w.

Case 2. $OPT(i, w)$ selects item i.

- Collect value v_i.
- New weight limit = $w - w_i$.
- $OPT(i, w)$ selects best of {1, 2, …, $i - 1$} using this new weight limit.

$$OPT(i, w) = \begin{cases} 0 & if\ i = 0 \\ OPT(i-1, w) & if\ w_i > w \\ \max\{OPT(i-1, w), v_i + OPT(i-1, w - w_i)\} & otherwise \end{cases}$$

Knapsack Problem: Bottom-Up Dynamic Programming

Knapsack $(n, W, w_1, w_2, \ldots, w_n, v_1, v_2, \ldots, v_n)$

For $w = 0$ To W
    $M[0, w] \leftarrow 0.$

For $i = 1$ To n
    For $w = 0$ To W
        If $w_i > w$
            $M[i, w] \leftarrow M[i-1, w].$
        Else
            $M[i, w] \leftarrow \max\{M[i-1, w], v_i + M[i-1, w-w_i]\}.$

Return $M[n, W]$.

Knapsack Problem: Bottom-Up Dynamic Programming Demo

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

knapsack instance
(weight limit W = 11)

$$OPT(i,w) = \begin{cases} 0 & if\ i = 0 \\ OPT(i-1,w) & if\ w_i > w \\ \max\{OPT(i-1,w),\ v_i + OPT(i-1,w-w_i)\} & otherwise \end{cases}$$

weight limit w

subset of items 1, …, i	0	1	2	3	4	5	6	7	8	9	10	11
{ }												
{ 1 }												
{ 1, 2 }												
{ 1, 2, 3 }												
{ 1, 2, 3, 4 }												
{ 1, 2, 3, 4, 5 }												

Knapsack Problem: Bottom-Up Dynamic Programming Demo

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

knapsack instance
(weight limit W = 11)

$$OPT(i,w) = \begin{cases} 0 & if\ i = 0 \\ OPT(i-1,w) & if\ w_i > w \\ \max\{OPT(i-1,w), v_i + OPT(i-1,w-w_i)\} & otherwise \end{cases}$$

weight limit w

subset of items 1, …, i	0	1	2	3	4	5	6	7	8	9	10	11
{ }	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	35	40

OPT(i, w) = max-profit subset of items 1, …, i with weight limit w.