Design and Analysis of Algorithms
Review

Si Wu

School of CSE, SCUT
cswusi@scut.edu.cn

TA: 1684350406@qq.com

Topics

- Sort
- Algorithm Analysis
- Recurrence
- Divide-and-Conquer
- Greedy Algorithms
- Linear Programming
- Dynamic Programming
- Network Flow
- Approximation Algorithms
- Backtrack
- Branch-and-Bound

Revisit:

- Algorithm Analysis
- Recurrence
- Divide-and-Conquer
- Greedy Algorithms
- Linear Programming
- Dynamic Programming
- Backtrack

Revisit:

Algorithm Analysis

O-notation

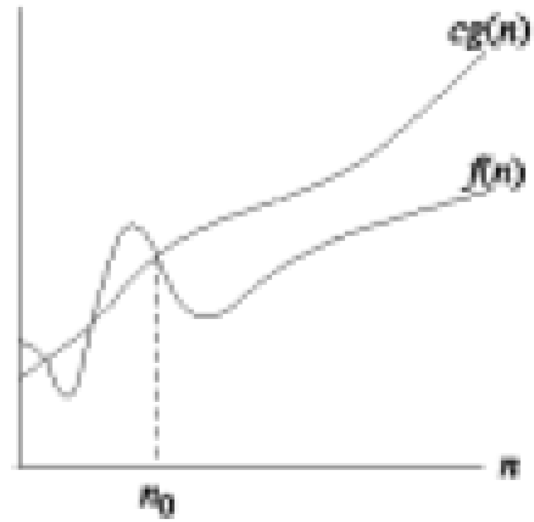
***O(g(n))* = {*f(n)*:** **There exist positive constants *c* and n_0 such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0$}**

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

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

Ex. $\boldsymbol{f(n) = 32n^2 + 17n + 1}$

- $\boldsymbol{f(n)}$ **is** $\boldsymbol{O(n^2)}$
- $\boldsymbol{f(n)}$ **is also** $\boldsymbol{O(n^3)}$
- $\boldsymbol{f(n)}$ **is neither** $\boldsymbol{O(n)}$ **nor** $\boldsymbol{O(nlgn)}$



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

O-notation

Ex.

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

Because $1/3n^2 - 3n \le cn^2$ **if** $c \ge 1/3\text{-}3/n$ **which holds for** $c = 1/3$ **and** $n > 1$**.**

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

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

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

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

Ω-notation

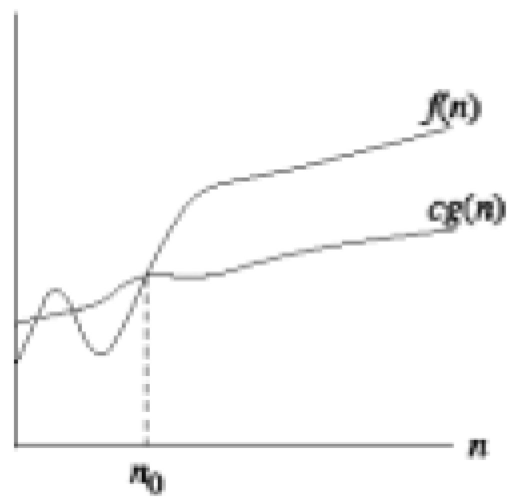
***Ω(g(n))* = {*f(n)*: There exist positive constants *c* and n_0 such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0$}**

- **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

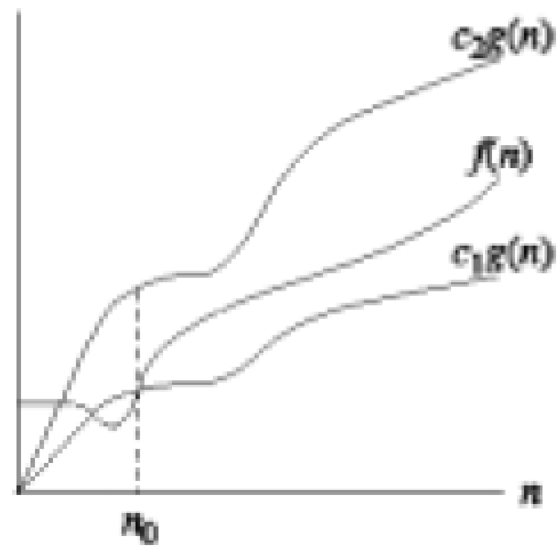
$\Theta(g(n)) = \{f(n)$: There exist positive constants c_1, c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0\}$

- 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 $\Theta(g(n))$.

Revisit:

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.

Induction

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

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

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

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 Merge-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.

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

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

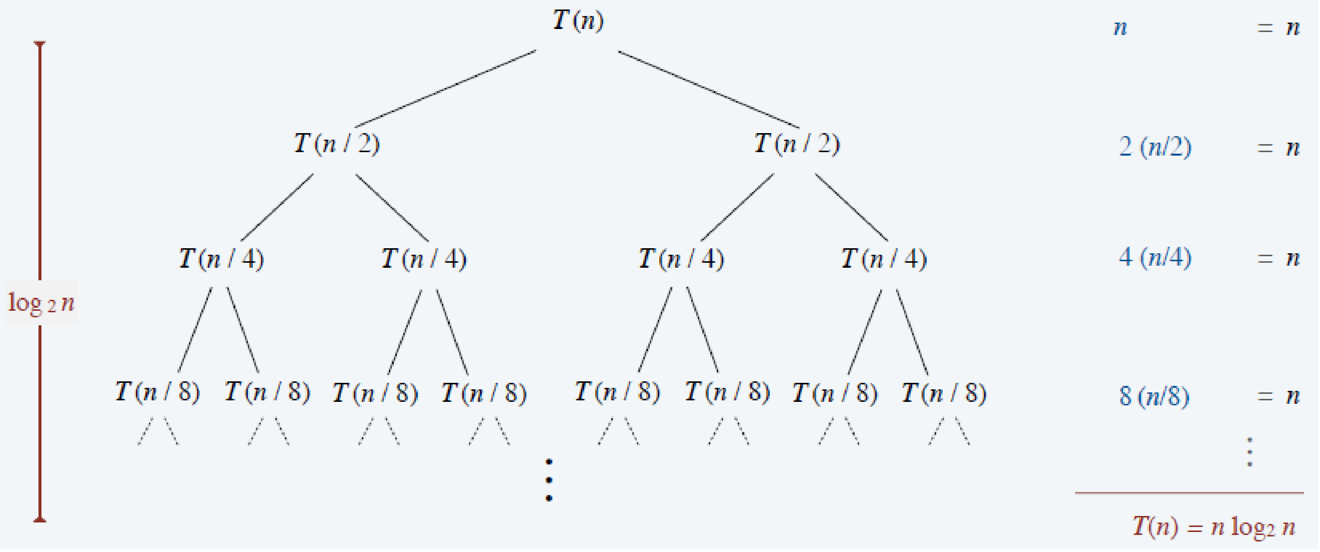
$\log_2 n_2 \le \lceil \log_2 n \rceil - 1$

Recursion Tree

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

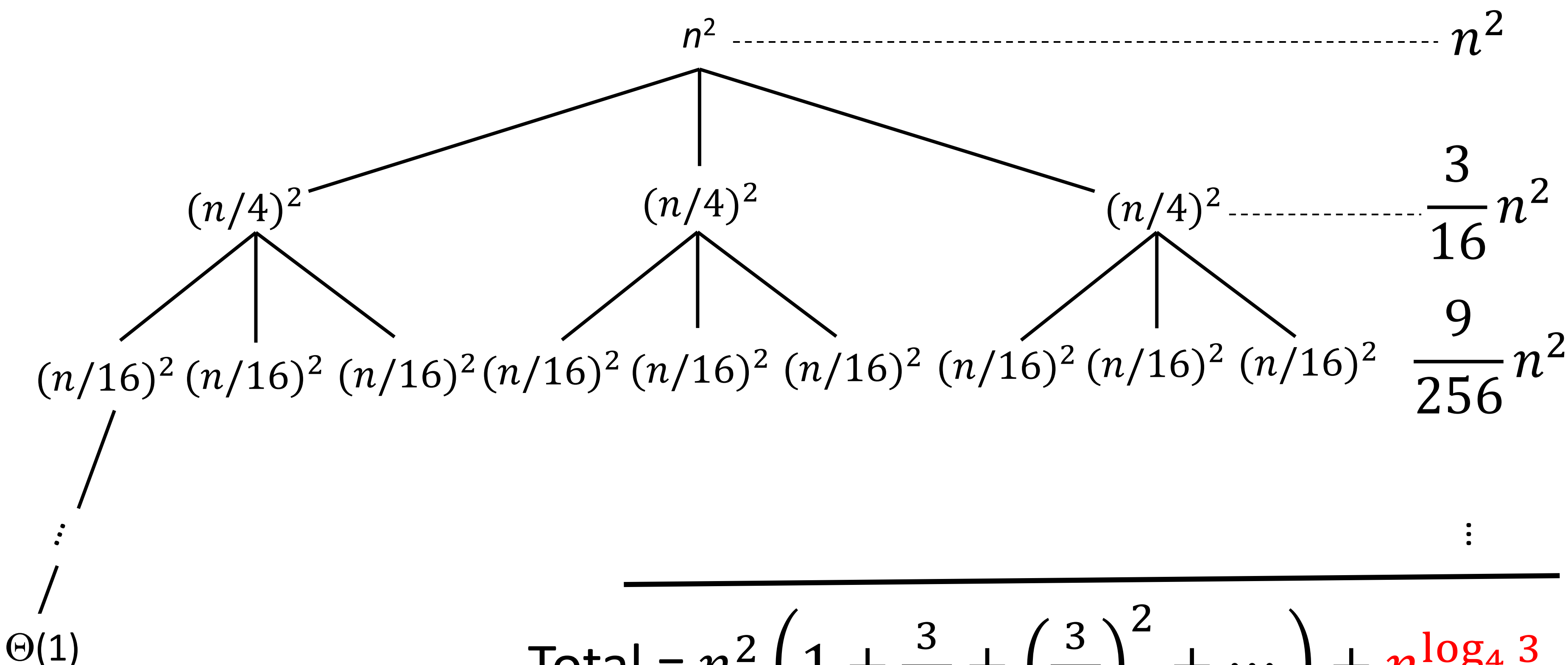
$$T(n) = \begin{cases} 0, & if\ n = 1 \\ 2T(n/2) + n, & otherwise \end{cases}$$

assuming n is a power of 2



Example of Recursion Tree

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



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

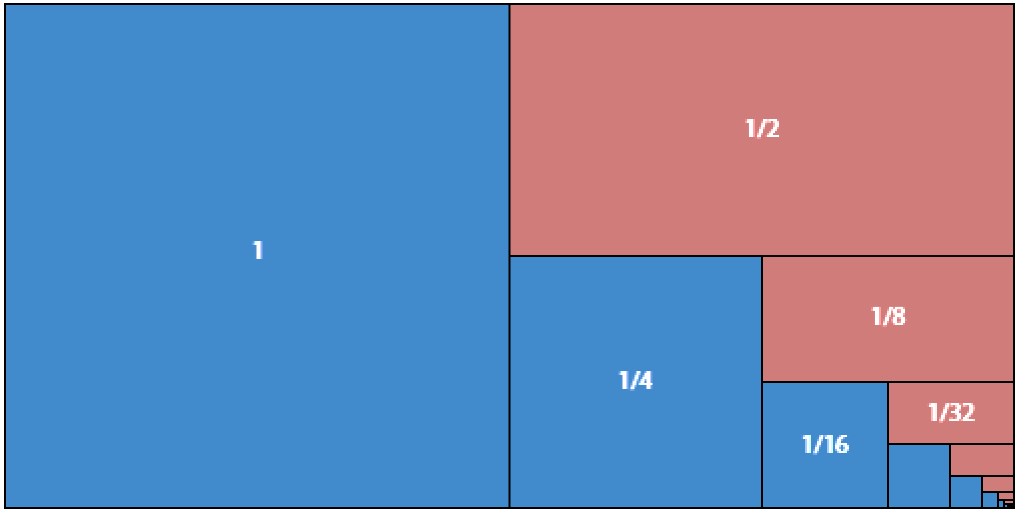
$$= \Theta(n^2)$$ *geometric series*

Geometric Series

Fact 1. For $r \neq 1$, $1 + r + r^2 + r^3 + \ldots + r^{k-1} = \dfrac{1 - r^k}{1 - r}$

Fact 2. For $r = 1$, $1 + r + r^2 + r^3 + \ldots + r^{k-1} = k$

Fact 3. For $r < 1$, $1 + r + r^2 + r^3 + \ldots = \dfrac{1}{1 - r}$



1 + 1/2 +1/4 + 1/8 + ... = 2

Master Method

Goal. Recipe for solving common divide-and-conquer recurrences:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

With $T(0) = 0 \; and \; T(1) = \Theta(1)$.

Terms.

- $a \geq 1$ is the (integer) number of subproblems.
- $b > 1$ is the (integer) factor by which the subproblem size decreases.
- $f(n) =$ work to divide and combine subproblems.

Recursion tree.

- Number of levels: $k = \log_b n$.
- Number of subproblems at level i: a^i.
- Size of subproblem at level i: n/b^i.
- Number of leaves: $n^{\log_b a}$.

Master Theorem

Master Theorem. Suppose that $T(n)$ is a function on the non-negative integers that satisfies the recurrence:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

with $T(0) = 0 \; and \; T(1) = \Theta(1)$, where n/b means either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then,

Case 1. If $f(n) = O(n^k)$ for some constant $k < \log_b a$, then $T(n) = \Theta\left(n^{\log_b a}\right)$.

Ex. $T(n) = 3T(n/2) + 5n$

$a = 3, b = 2, f(n) = 5n, k = 1, \log_b a = 1.58$

$T(n) = \Theta\left(n^{\log_2 3}\right)$

Master Theorem

Master Theorem. Suppose that $T(n)$ is a function on the non-negative integers that satisfies the recurrence:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

with $T(0) = 0\ and\ T(1) = \Theta(1)$, where n/b means either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then,

Case 2. If $f(n) = \Theta(n^k log^p n)$ for $p \geq 0$ and $k = \log_b a$, then $T(n) = \Theta\left(n^k log^{p+1} n\right)$.

Ex. $T(n) = 2T(n/2) + 17n \log n$

$a = 2, b = 2, f(n) = 17n \log n\,, k = 1, p = 1, \log_b a = 1$

$T(n) = \Theta(n\, log^2\, n)$

Master Theorem

Master Theorem. Suppose that $T(n)$ is a function on the non-negative integers that satisfies the recurrence:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

with $T(0) = 0 \; and \; T(1) = \Theta(1)$, where n/b means either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then,

Case 3. If $f(n) = \Omega(n^k)$ for some constant $k > \log_b a$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta\big(f(n)\big)$.

Ex. $T(n) = 3T(n/2) + n^2$
$a = 3, b = 2, f(n) = n^2, k = 2, \log_b a = 1.58$
Regularity condition: $3(n/2)^2 \leq cn^2$ for $c = 3/4$
$T(n) = \Theta(n^2)$

Revisit:

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

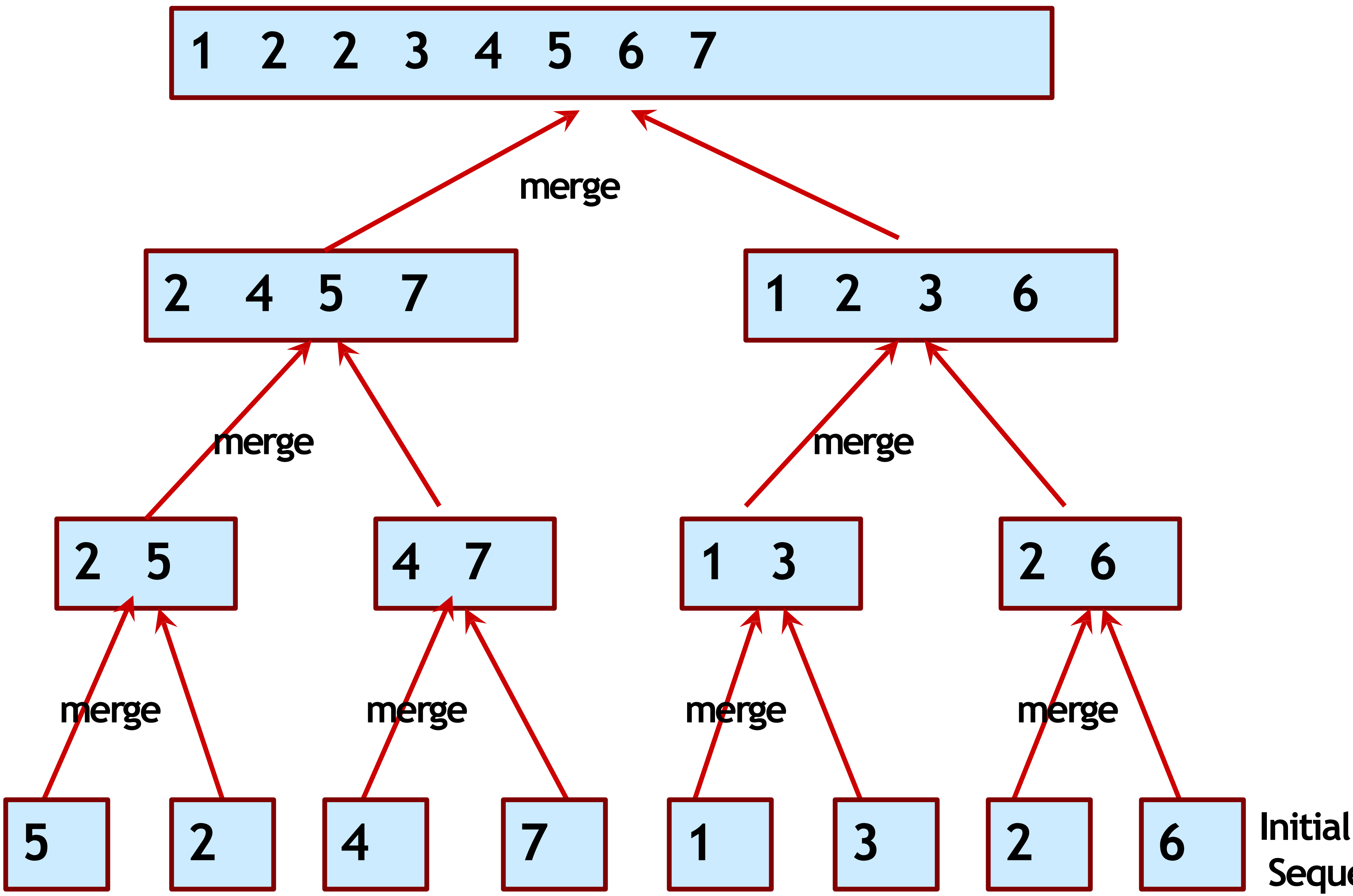
- Divide: Divide the n elements into two subsequences of n/2 elements each.
- Conquer: Sort the two subsequences recursively.
- Combine: Merge the two sorted subsequences to produce the sorted answer.

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

Action of Merge-Sort

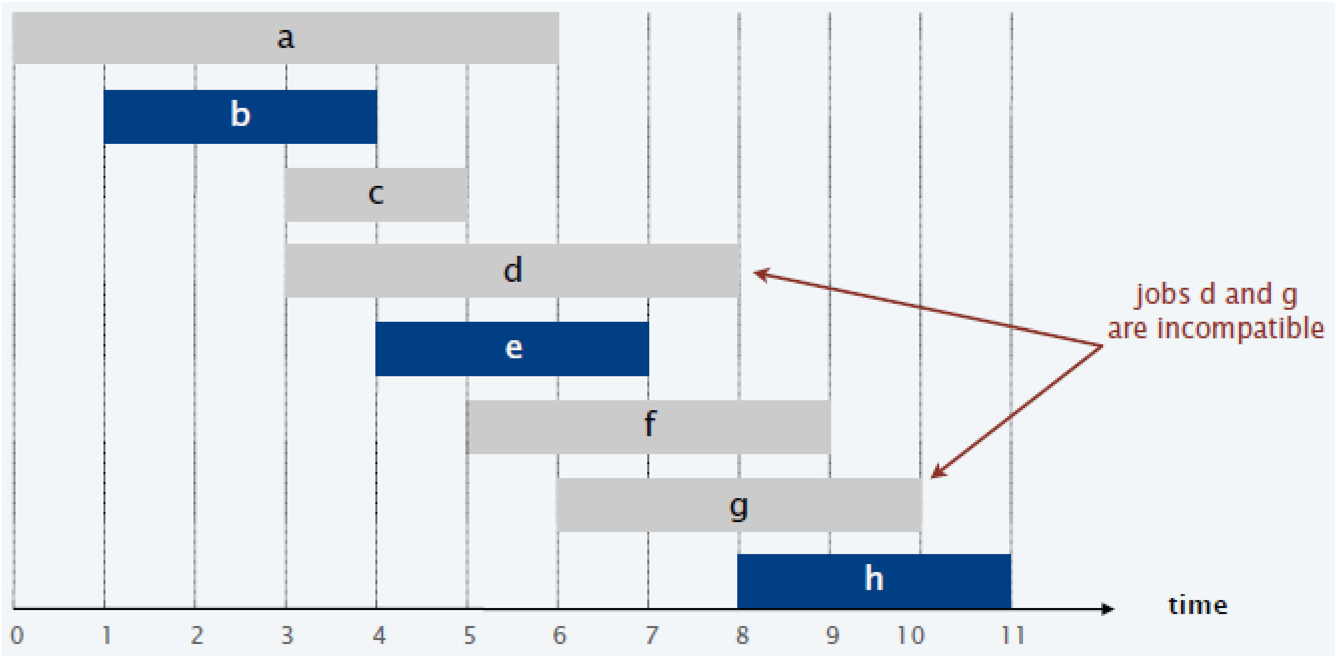


Revisit:

Greedy Algorithms

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.



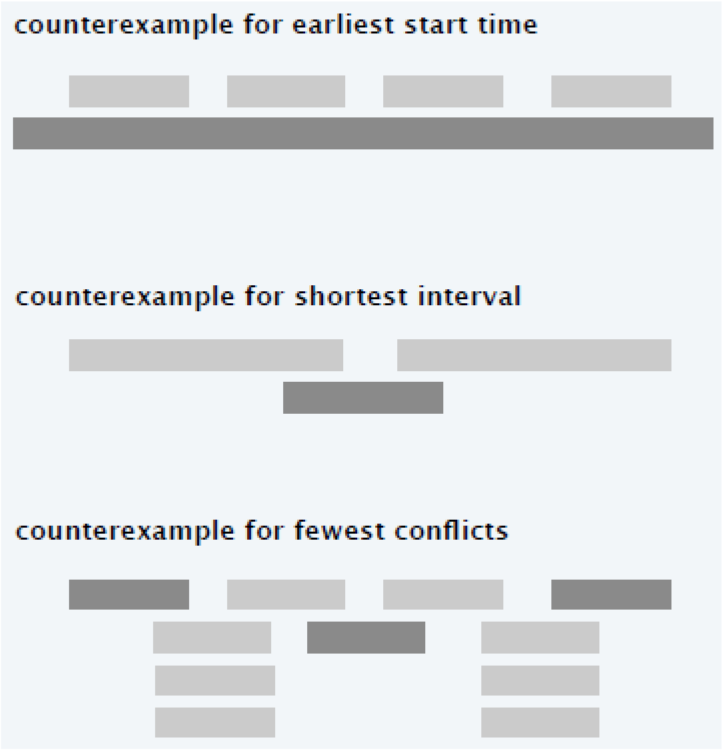
Interval Scheduling: Greedy Algorithms

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 $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)$
Sort jobs by finish time so that $f_1 \le f_2 \le \cdots \le f_n$
$A \leftarrow \emptyset$ (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.

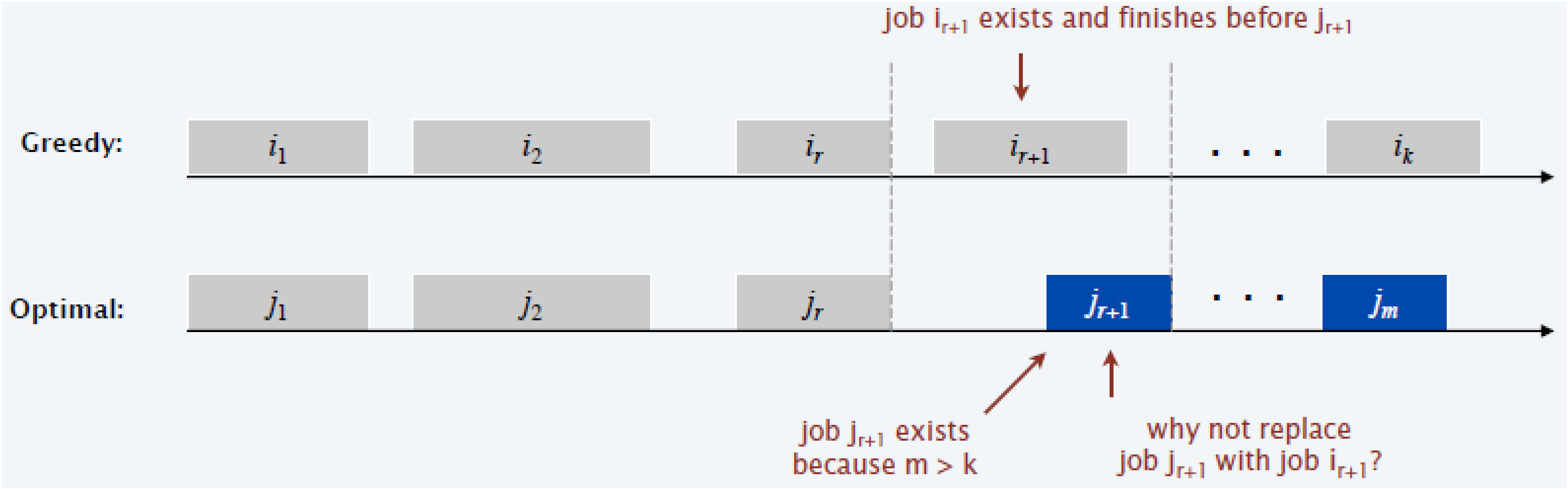
- Keep track of job j^* that was added last to A
- Job j is compatible with A iff $s_n \ge f_{j^*}$.
- Sorting by finish time takes $O(nlogn)$ time.

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.

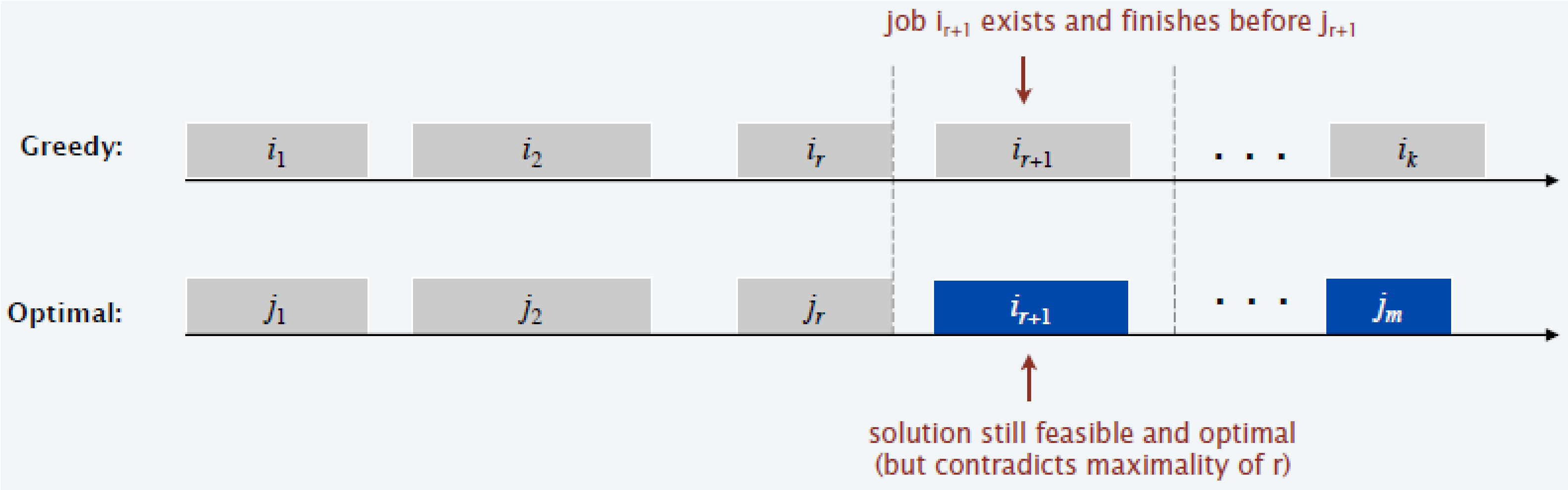


Interval Scheduling: Analysis of Earliest-Finish-Time-First Algorithm

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.

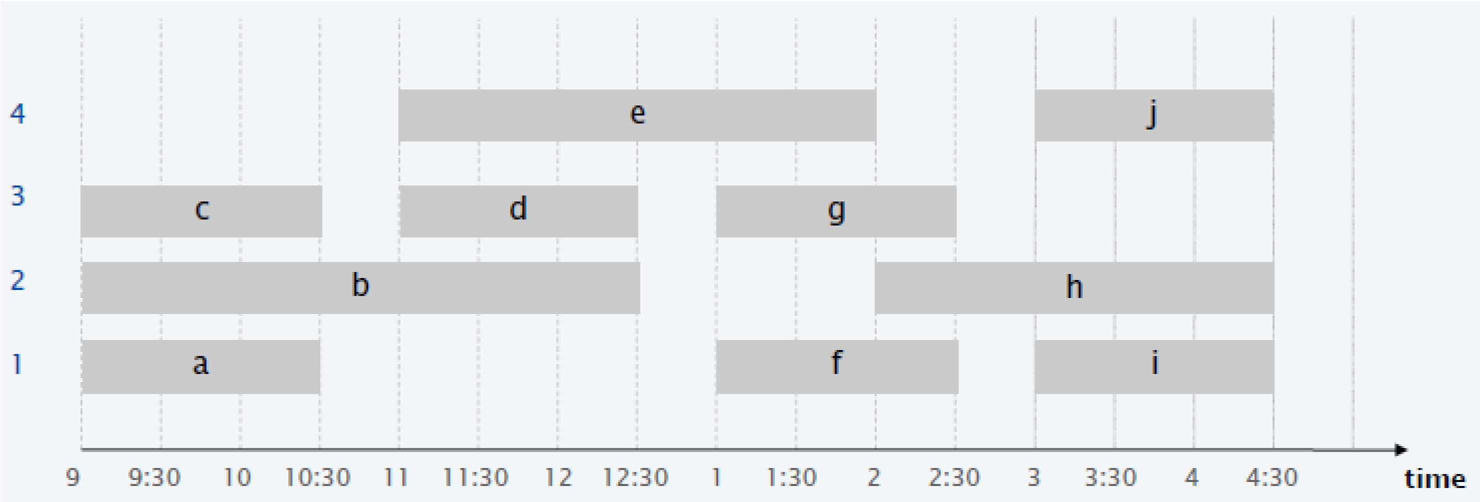


Interval Partitioning

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

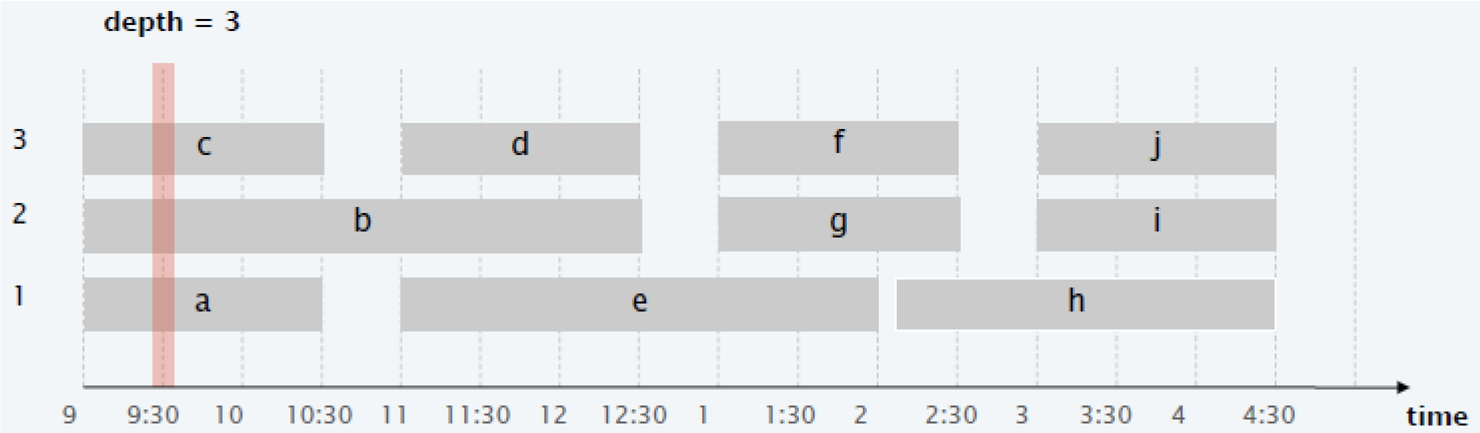
Earliest-Start-Time-First $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)$
Sort lectures by start time so that $s_1 \leq s_2 \leq \cdots \leq s_n$
$d \leftarrow 0$ (the number of allocated classrooms)
for $j = 1$ to n
if lecture j is compatible with some classroom
    Schedule lecture j in any such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

Interval Partitioning: 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.

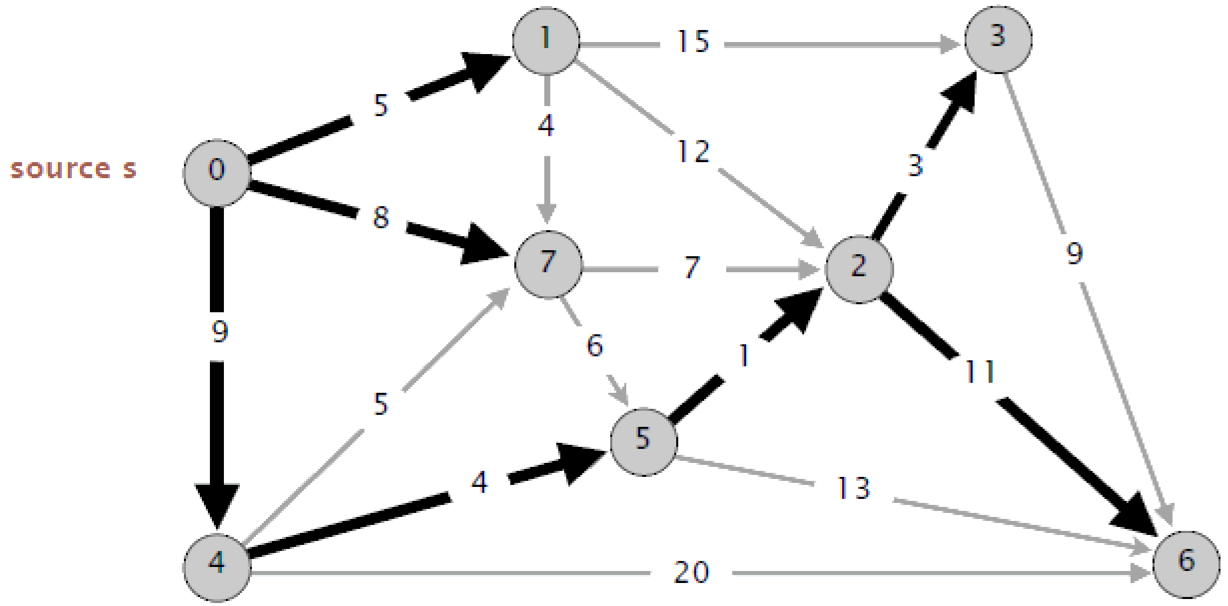
Theorem. Earliest-Start-Time-First algorithm is optimal.

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.



shortest-paths tree

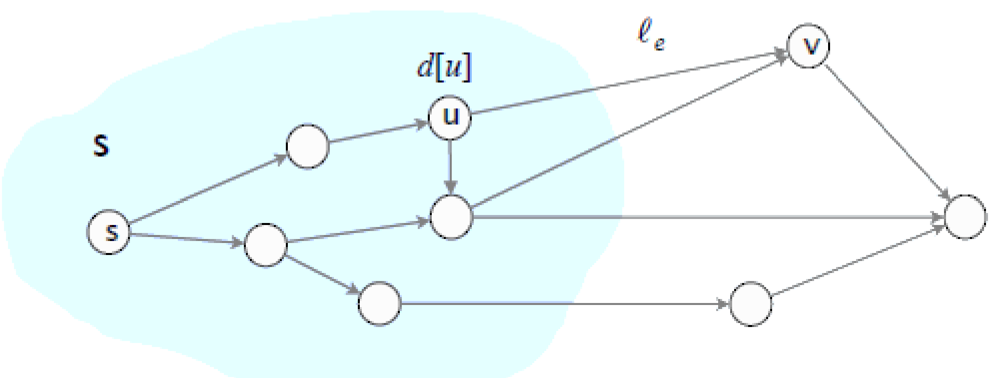
Dijkstra's Algorithm for Single-Source Shortest Paths Problem

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.

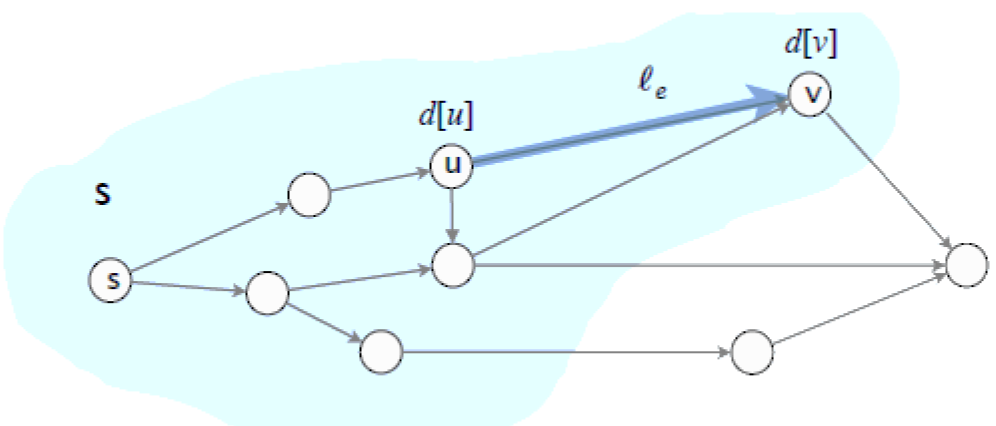
- Initialize $S \leftarrow \{s\}, d[s] = 0$.
- Repeatedly choose unexplored node $v \notin S$ which minimizes

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

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

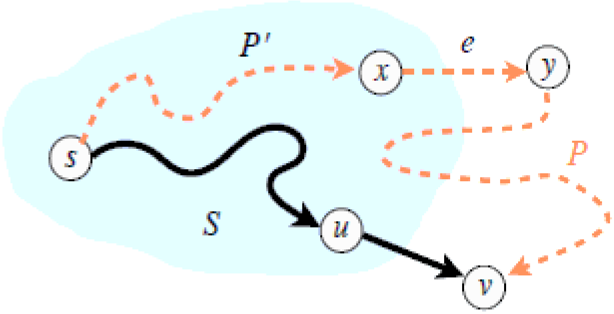
it is no shorter than $\pi(v)$.

- 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

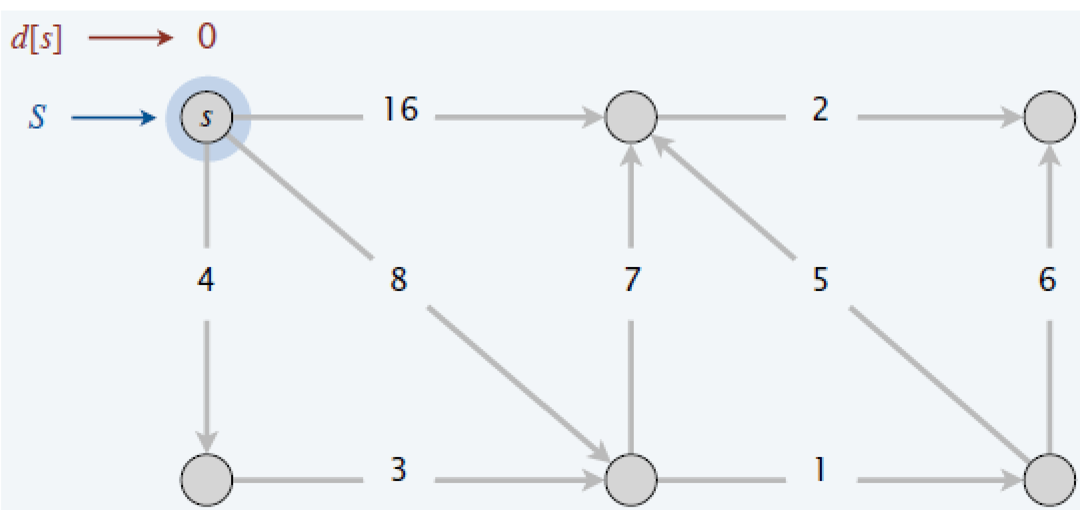
Dijkstra's Algorithm Demo

- 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 $e = (u, v)$.



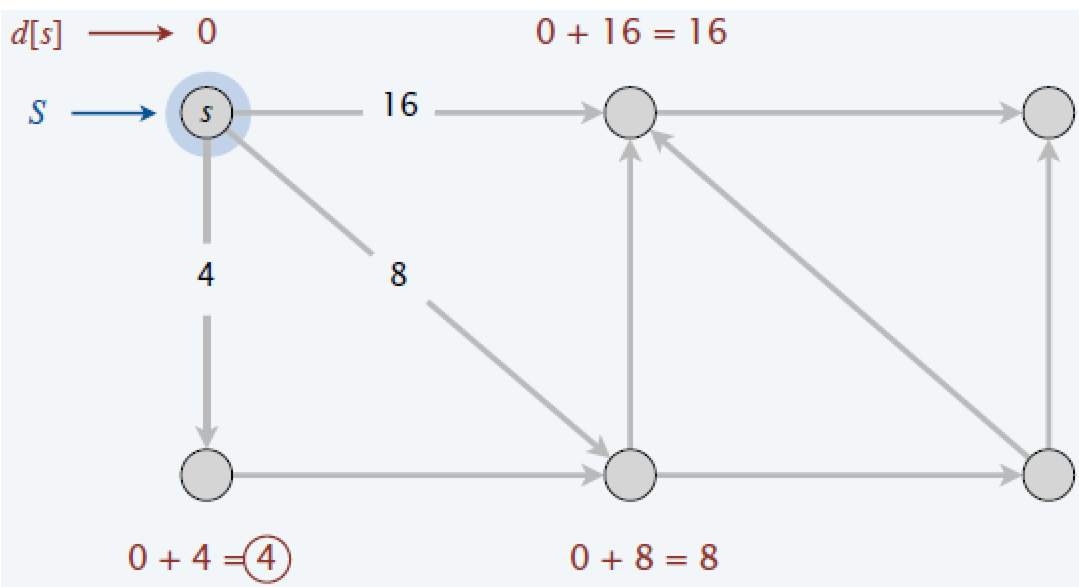
Dijkstra's Algorithm Demo

- 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 $e = (u, v)$.



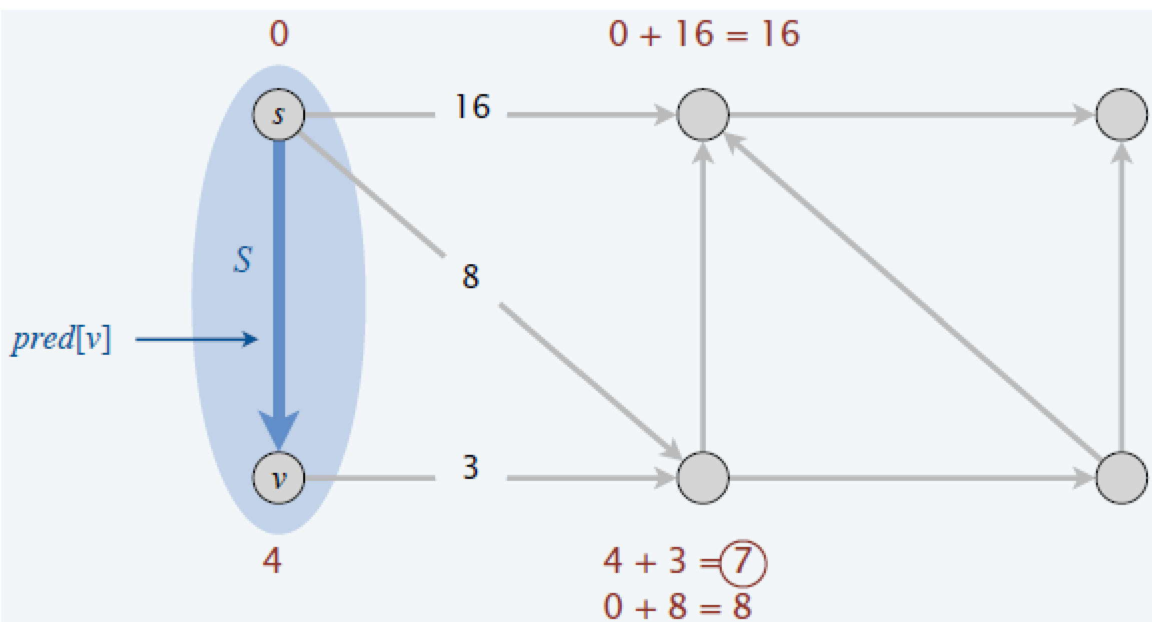
Dijkstra's Algorithm Demo

- 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 $e = (u, v)$.

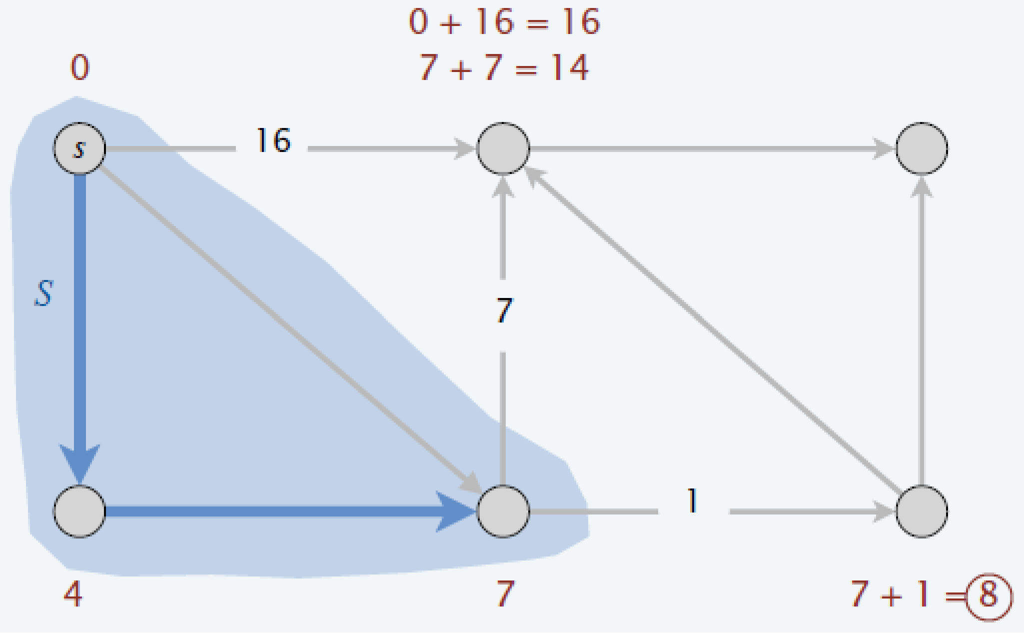


Dijkstra's Algorithm Demo

- 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 $pred(v) \leftarrow argmin$.



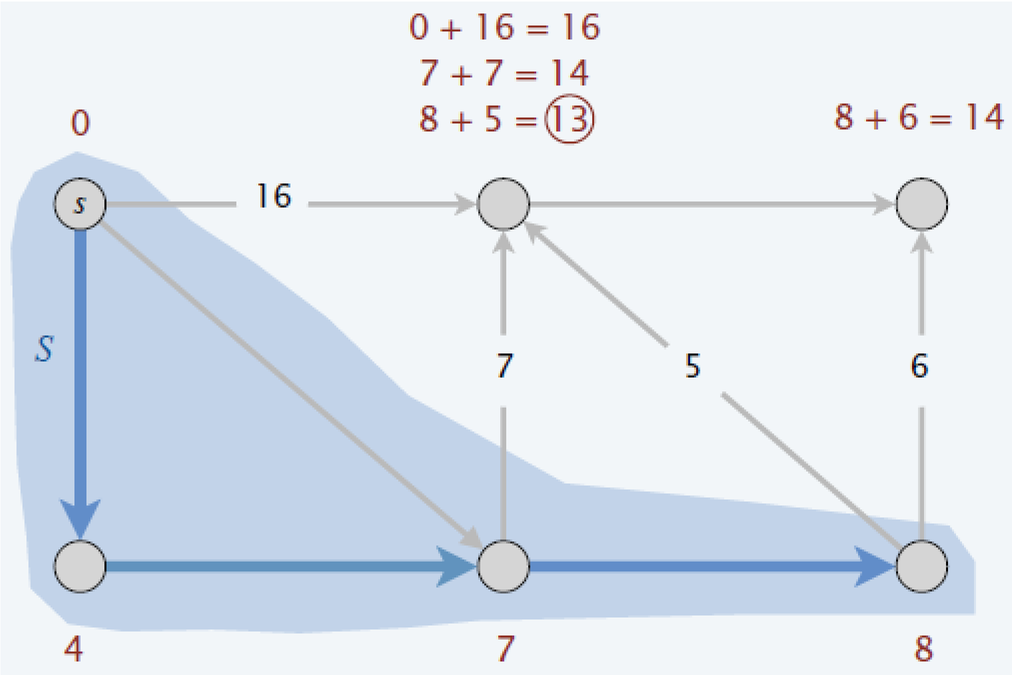
The length of a shortest path from s to some node u in explored part S, followed by a single edge $e = (u, v)$.

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 $pred(v) \leftarrow argmin$.



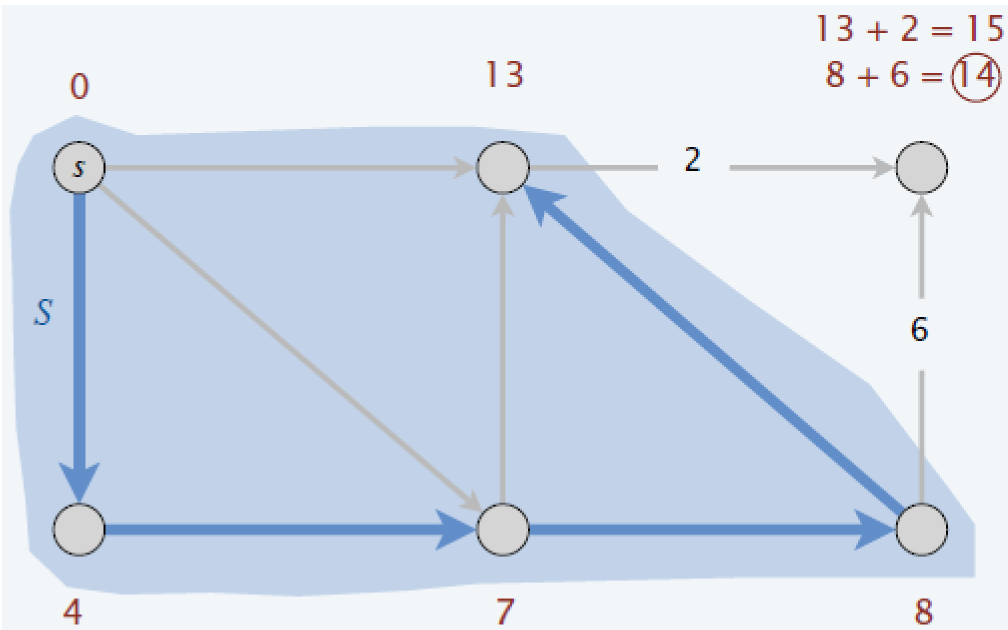
The length of a shortest path from s to some node u in explored part S, followed by a single edge $e = (u, v)$.

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 $e = (u, v)$.

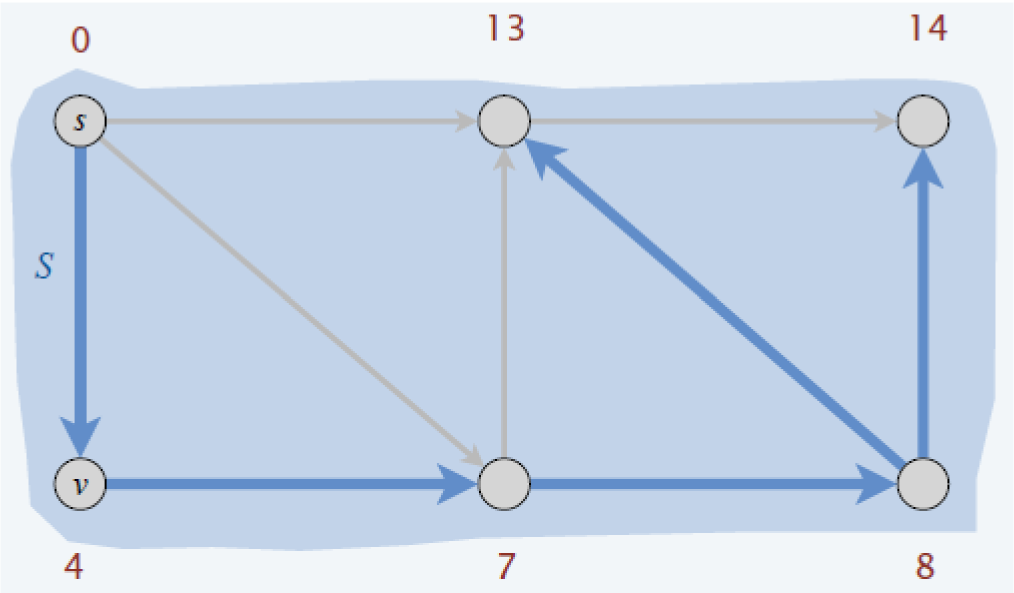
Dijkstra's Algorithm Demo

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

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

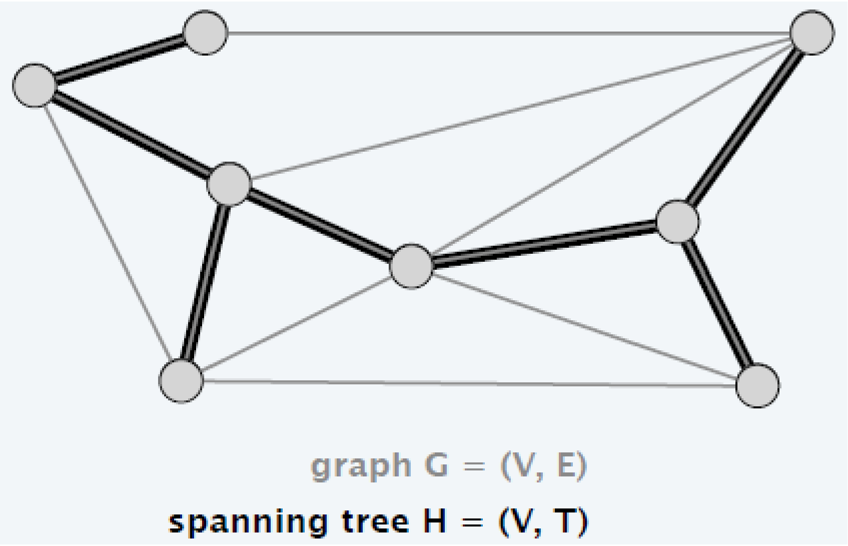
Add v to S; set $d[v] \leftarrow \pi(v)$ and $pred(v) \leftarrow argmin$.

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

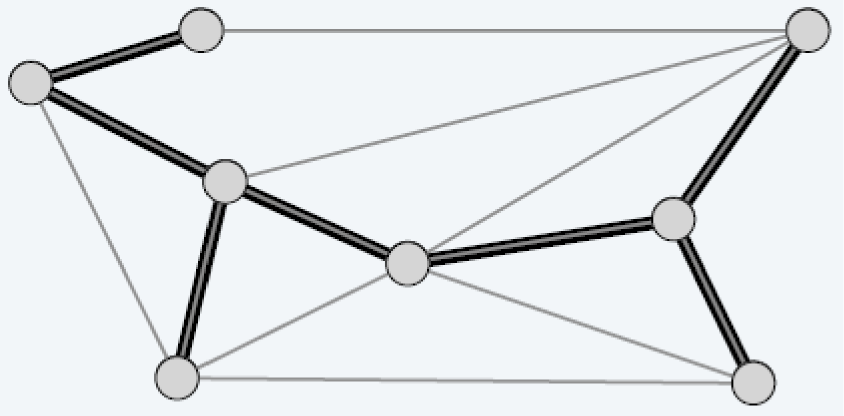
Def. Let $H = (V, T)$ be a subgraph of an undirected graph $G = (V, E)$. H is a spanning tree of G if H is both acyclic and connected.



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.

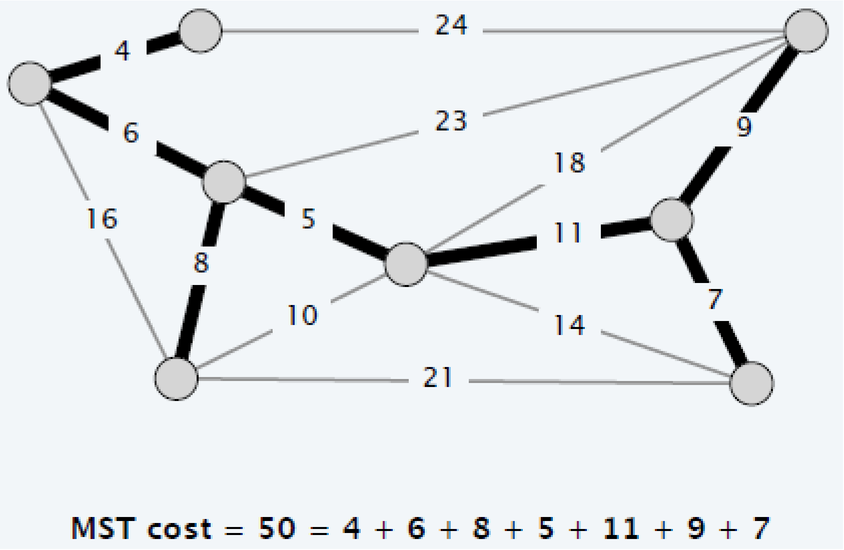


graph G = (V, E)

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.



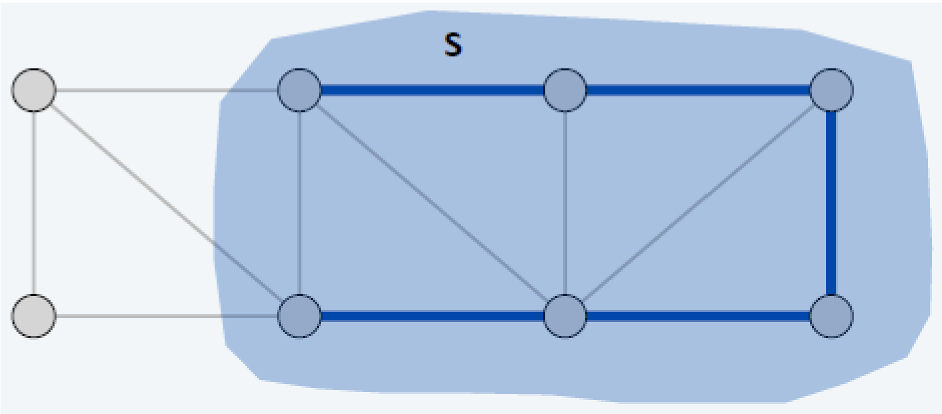
Prim's Algorithm

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.

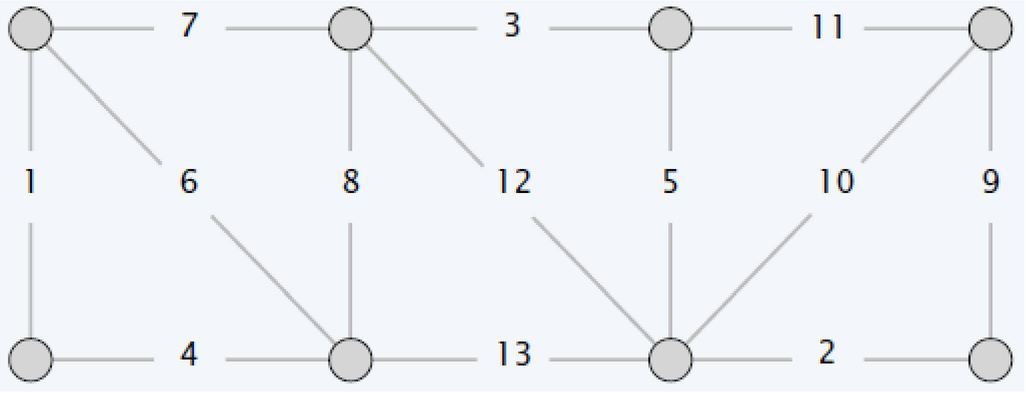
$\pi[v]$ = **weight of cheapest known edge between v and S.**

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

Prim's Algorithm Demo

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

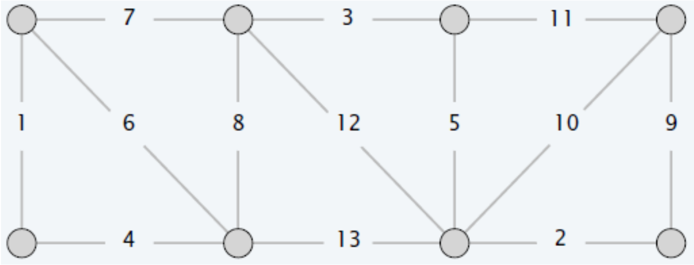
- Add to T a min-weight edge with one endpoint in S.
- Add new node to S.

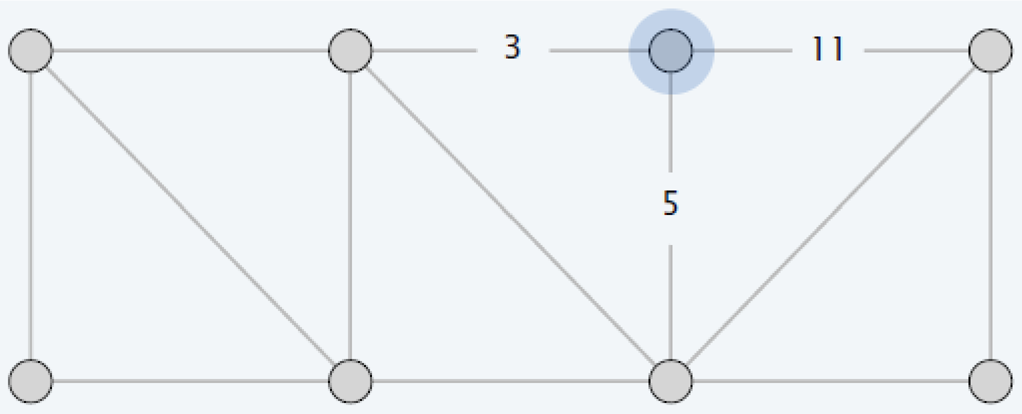


Prim's Algorithm Demo

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.



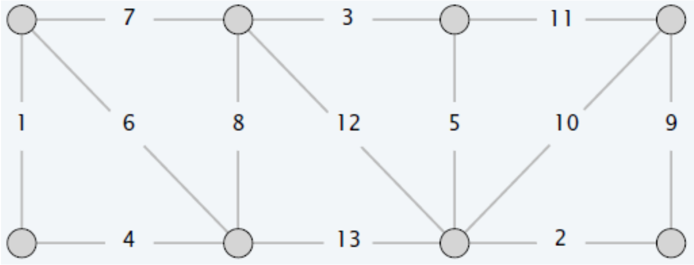


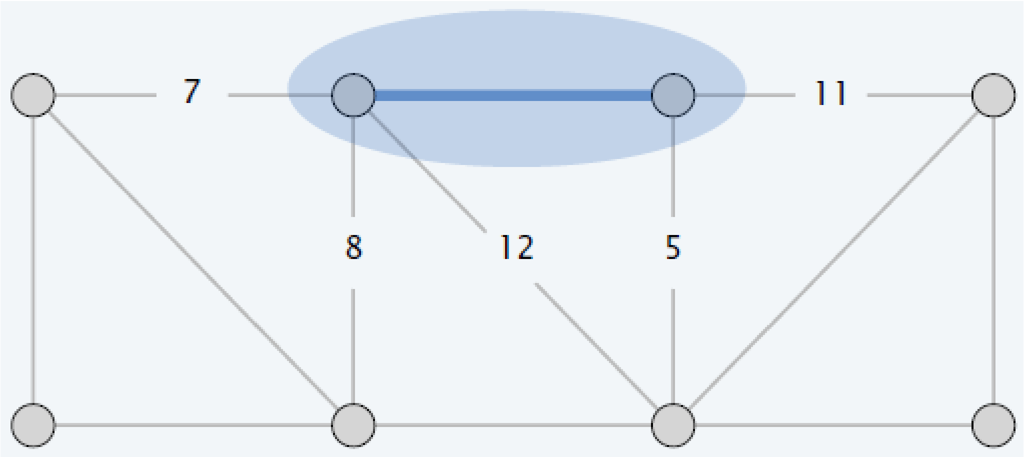
Prim's Algorithm Demo

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.



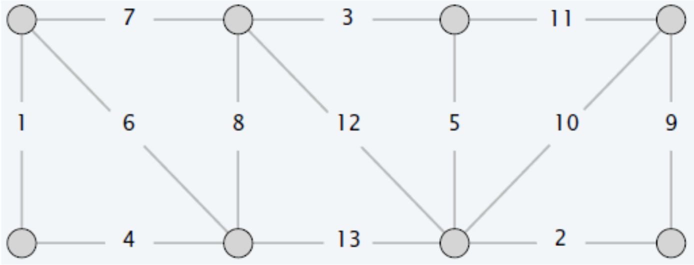


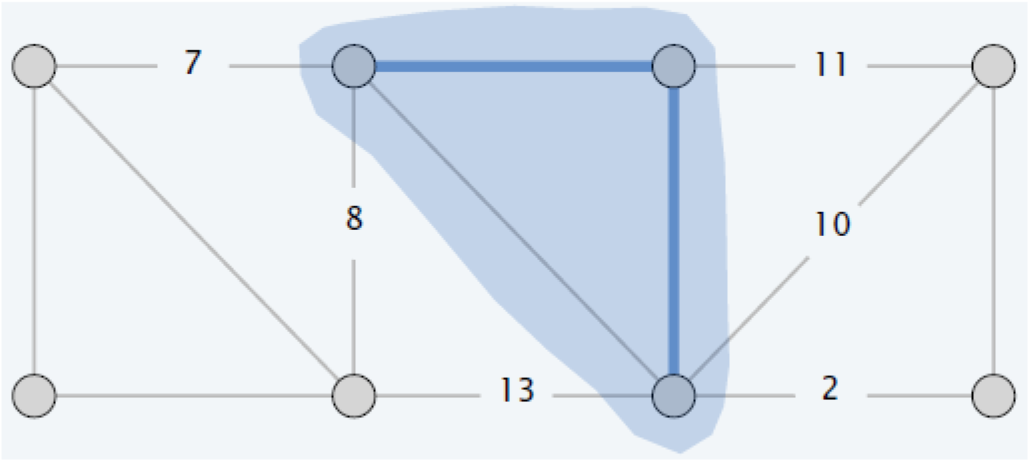
Prim's Algorithm Demo

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.



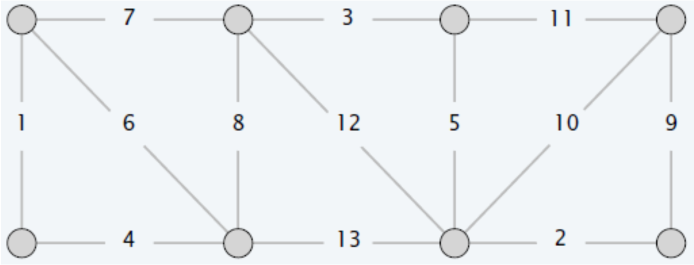


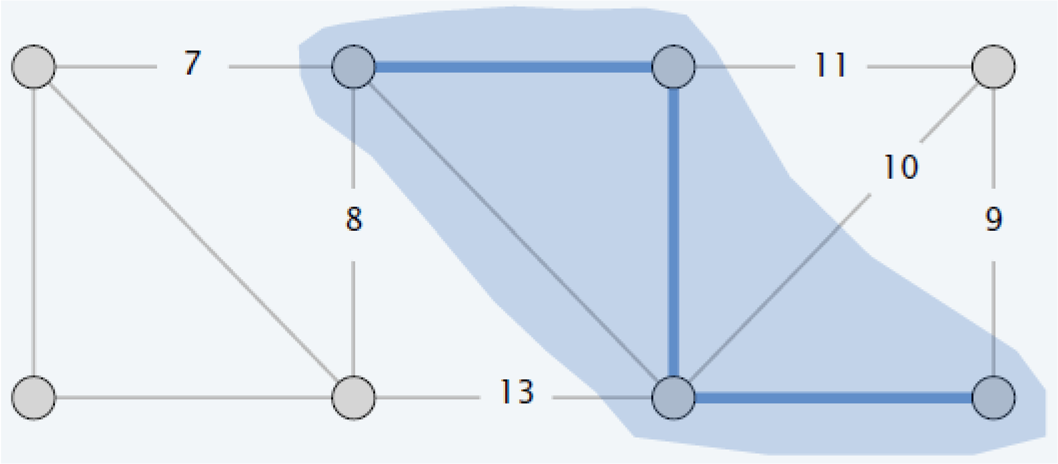
Prim’s Algorithm Demo

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.



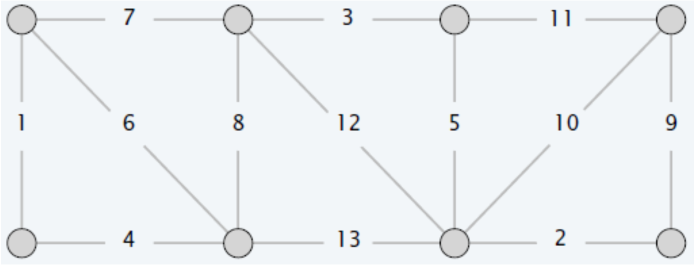


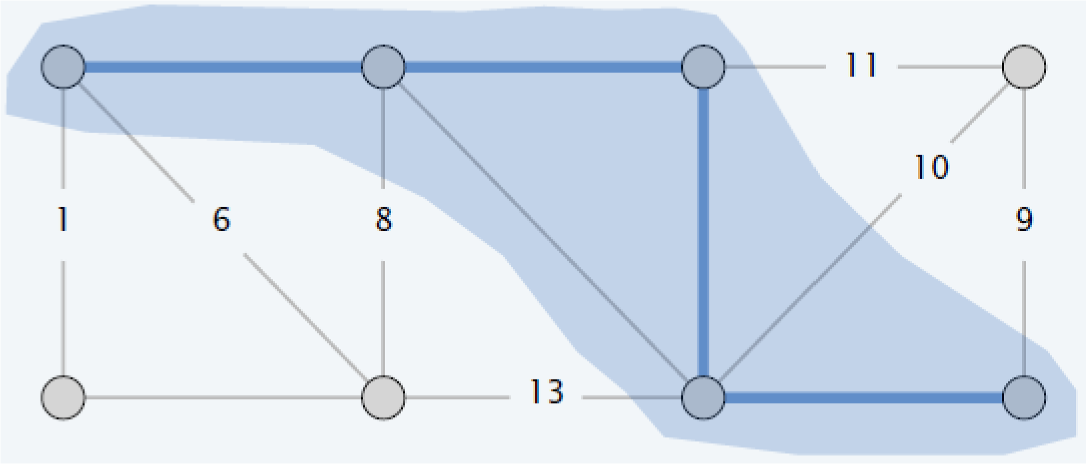
Prim's Algorithm Demo

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.



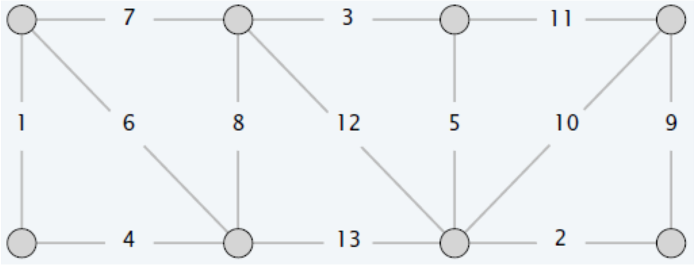


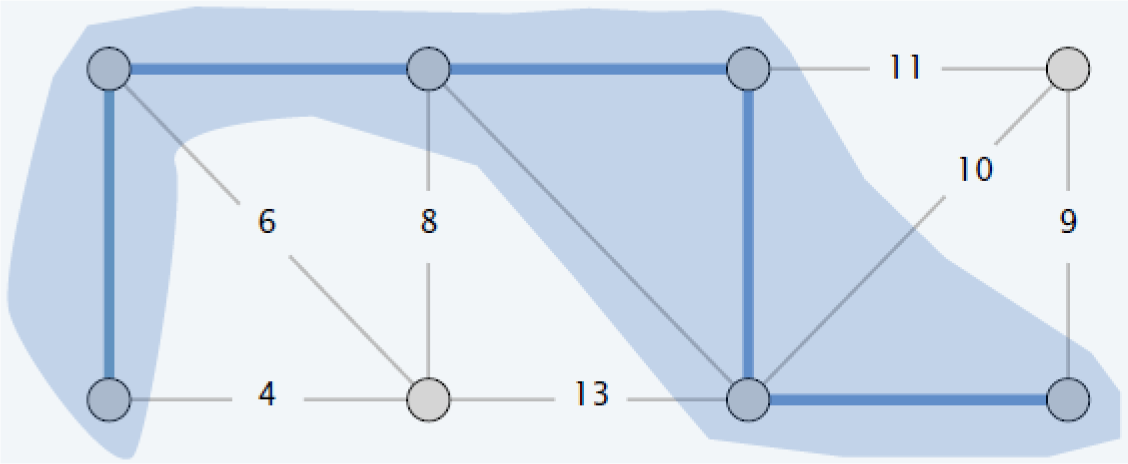
Prim's Algorithm Demo

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.



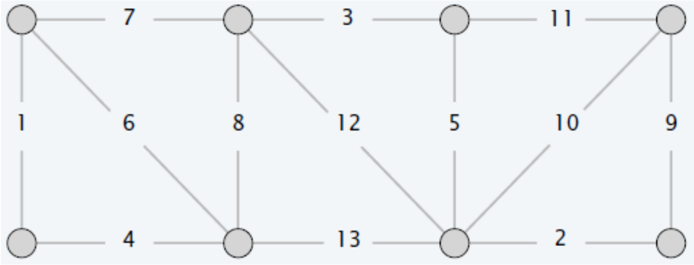


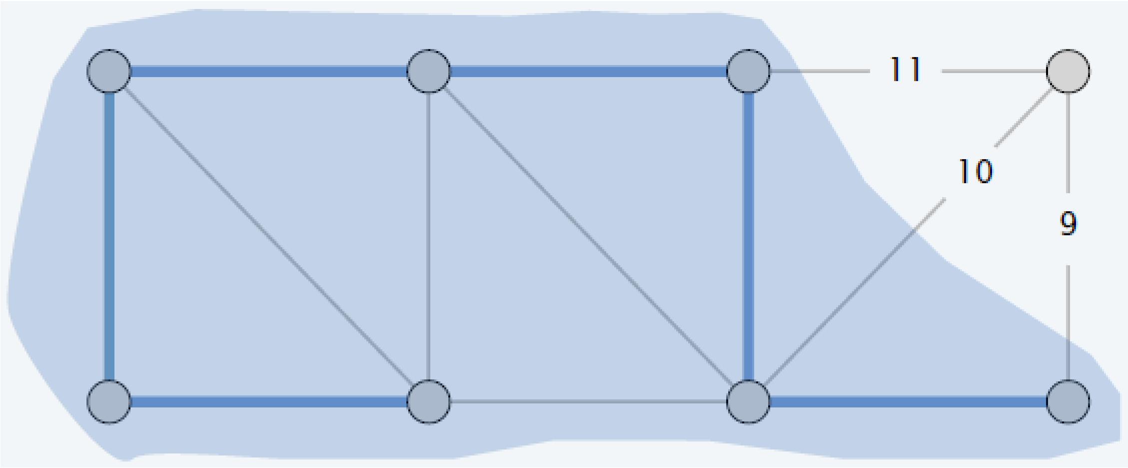
Prim's Algorithm Demo

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.

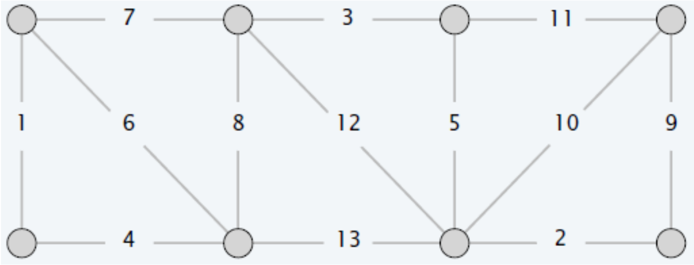


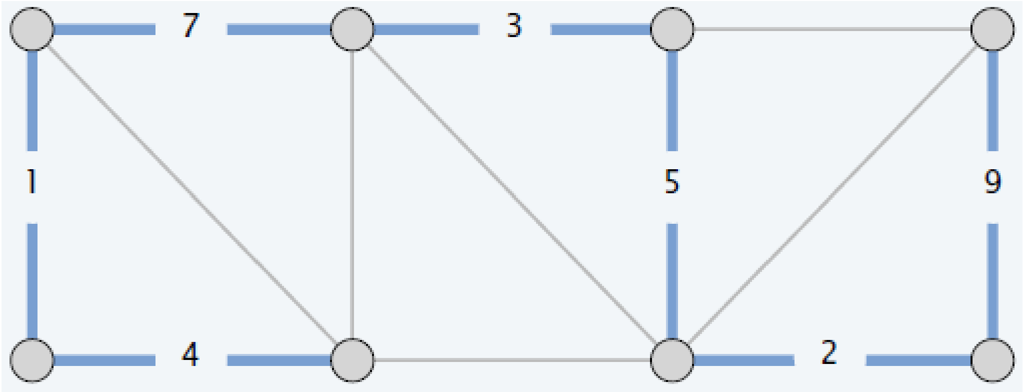


Prim's Algorithm Demo

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.



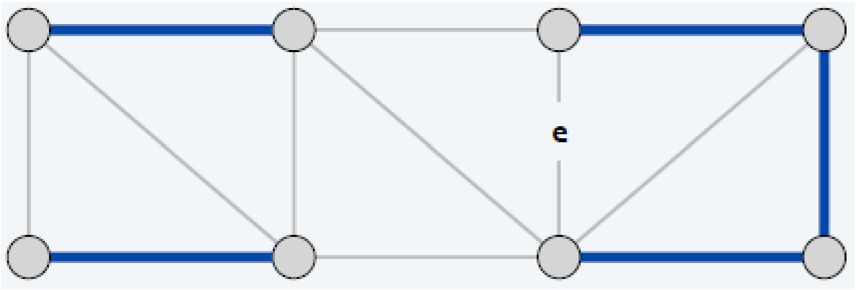


Kruskal's Algorithm

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

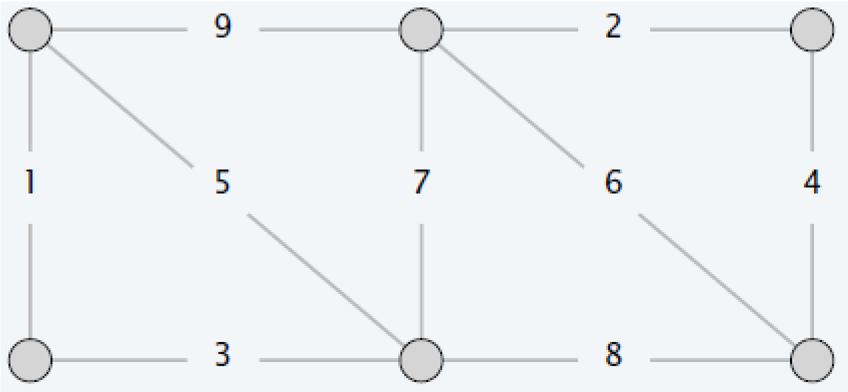
- Sort edges by weights.
- Use union-find data structure to dynamically maintain connected components.

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:

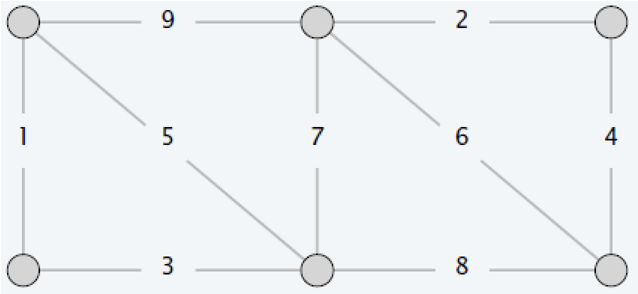
- Add to T unless it would create a cycle.

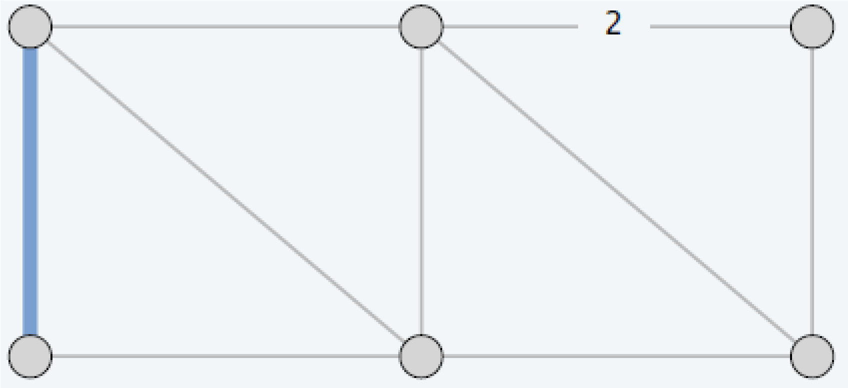


Kruskal's Algorithm Demo

Consider edges in ascending order of weight:

- Add to T unless it would create a cycle.

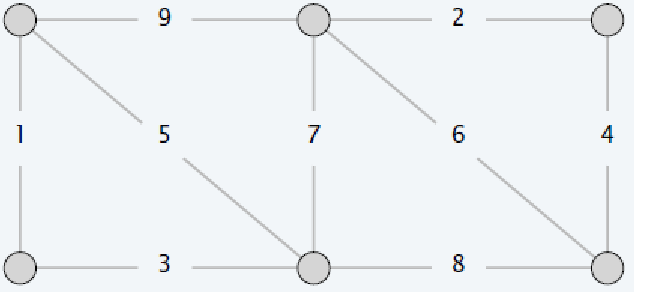


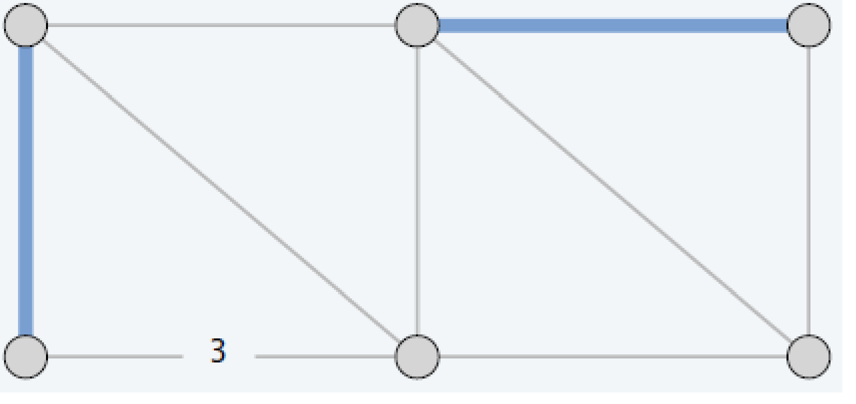


Kruskal's Algorithm Demo

Consider edges in ascending order of weight:

- Add to T unless it would create a cycle.

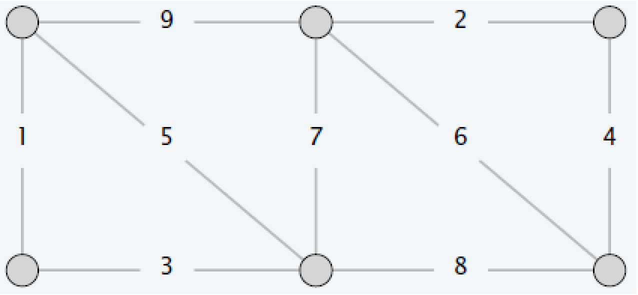


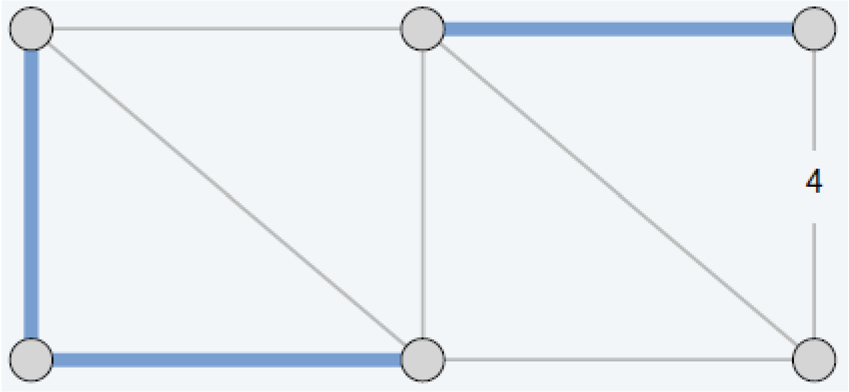


Kruskal's Algorithm Demo

Consider edges in ascending order of weight:

- Add to T unless it would create a cycle.

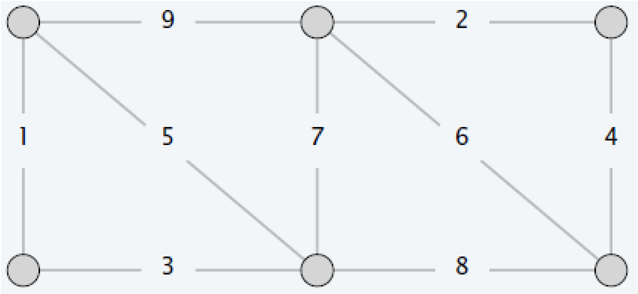


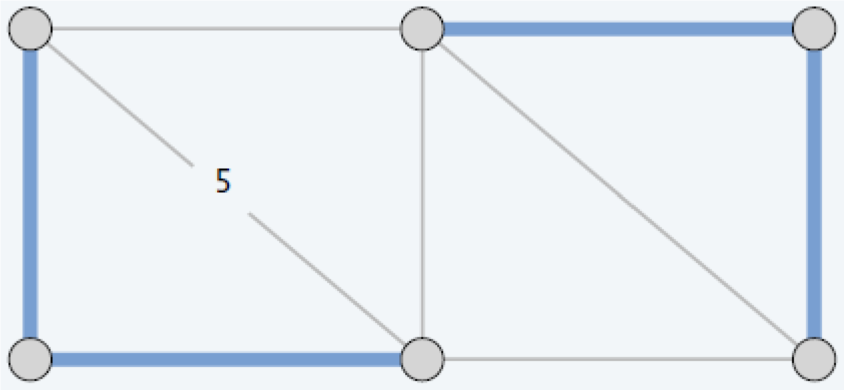


Kruskal's Algorithm Demo

Consider edges in ascending order of weight:

- Add to T unless it would create a cycle.

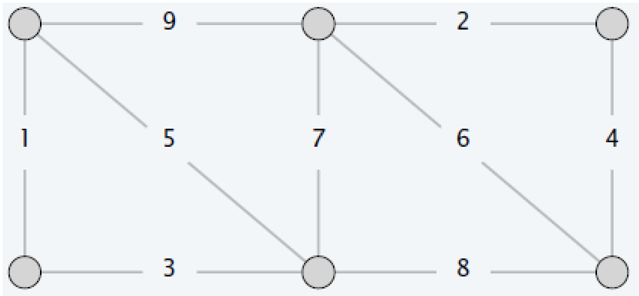


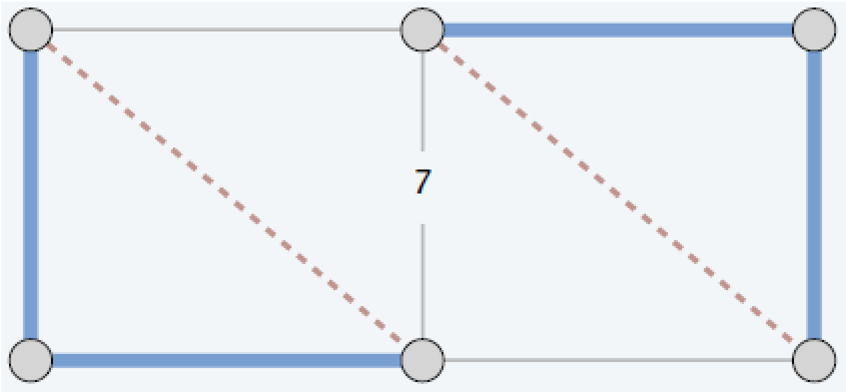


Kruskal's Algorithm Demo

Consider edges in ascending order of weight:

- Add to T unless it would create a cycle.

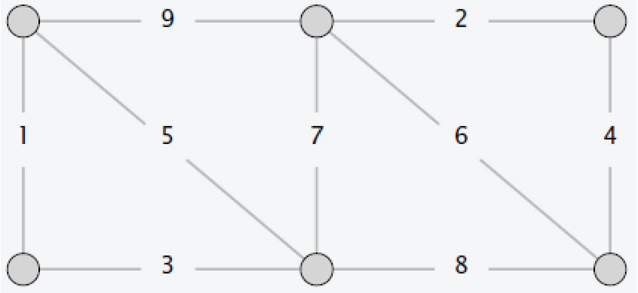


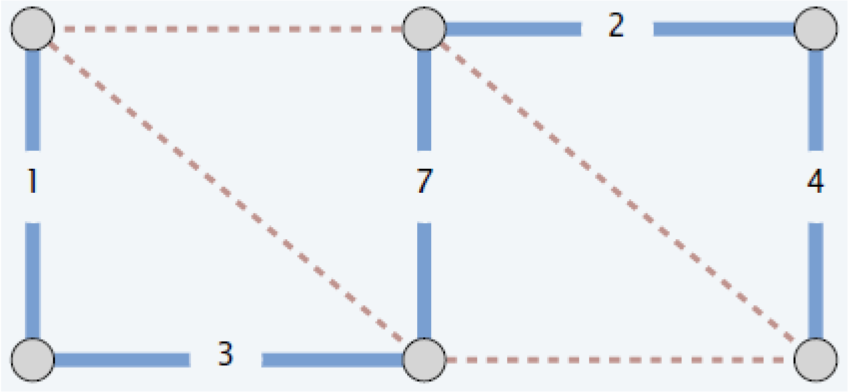


Kruskal's Algorithm Demo

Consider edges in ascending order of weight:

- Add to T unless it would create a cycle.





Revisit:

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.

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

Equivalent Forms

Easy to convert variants to standard form.

$$\begin{aligned} &\max c^T x \\ &s.t. \;\; Ax = b \\ &\qquad\;\; x \geq 0 \end{aligned}$$

- 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.

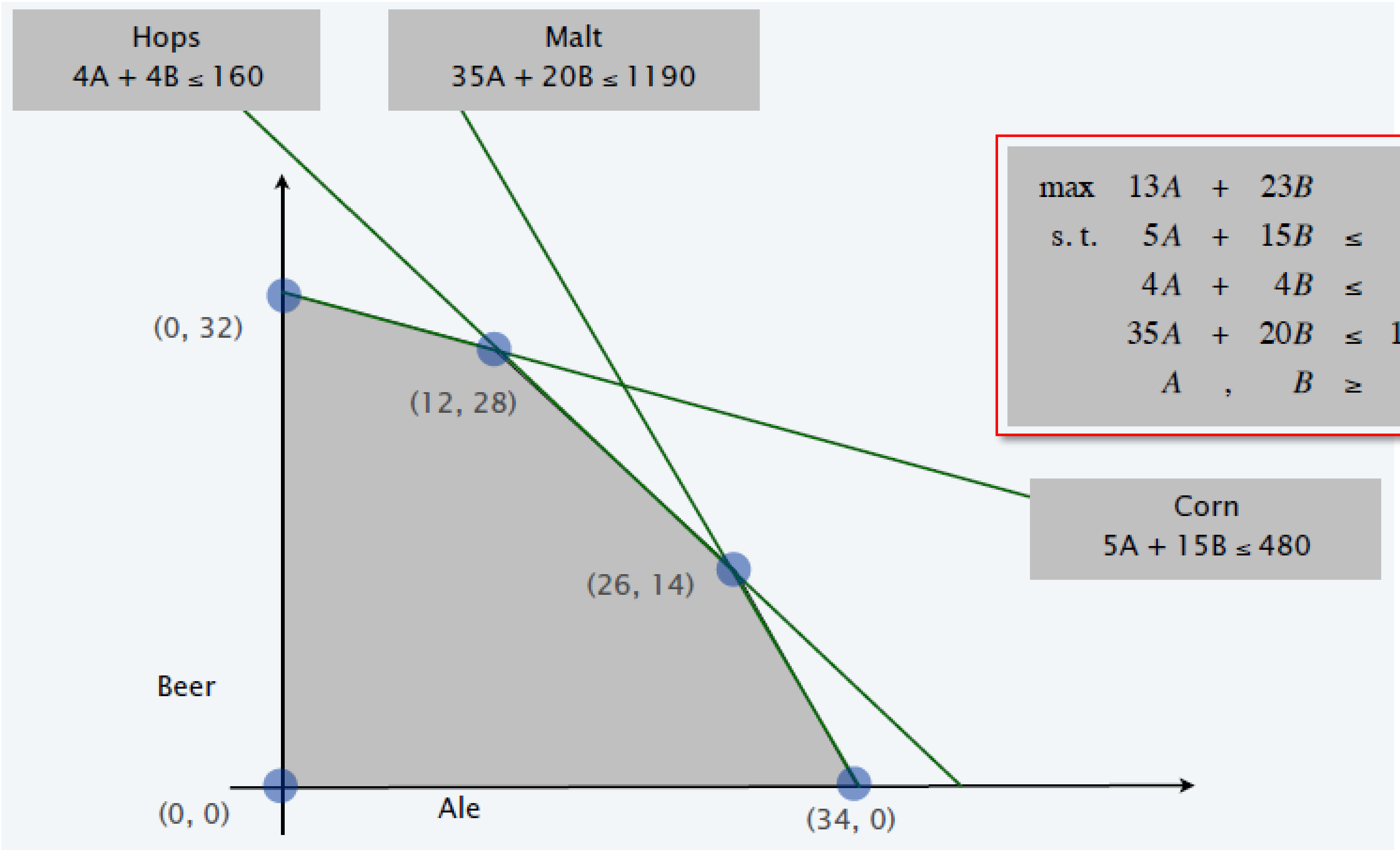
$$
\begin{array}{lrcrcr}
\max & 13A & + & 23B & & \\
\text{s. t.} & 5A & + & 15B & \le & 480 \\
 & 4A & + & 4B & \le & 160 \\
 & 35A & + & 20B & \le & 1190 \\
 & A & , & B & \ge & 0
\end{array}
$$

Standard form.

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

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

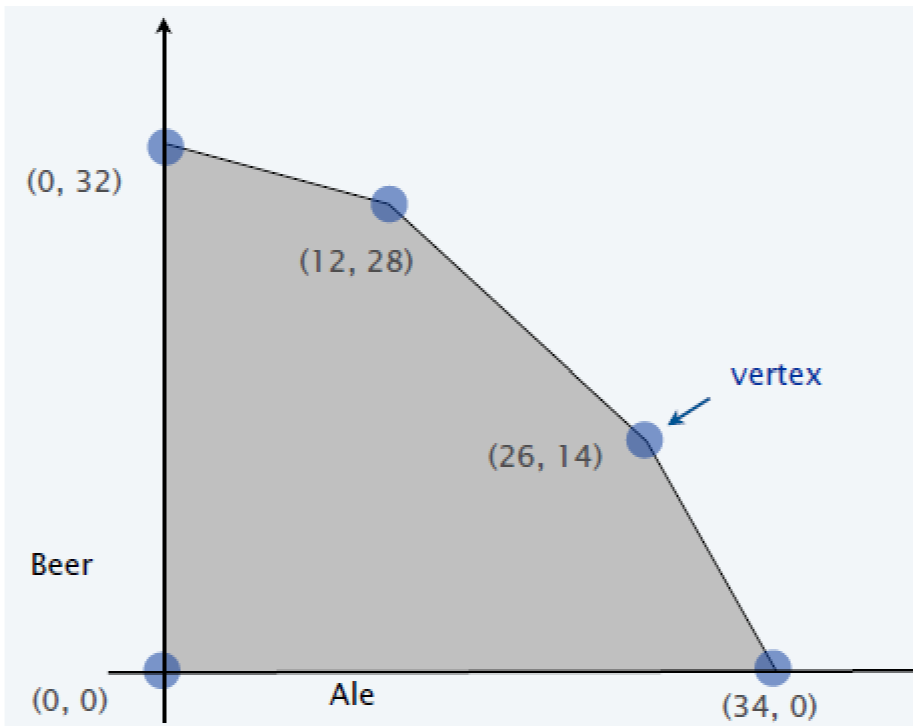
Brewery Problem: Feasible Region



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

Brewery Problem: Geometry

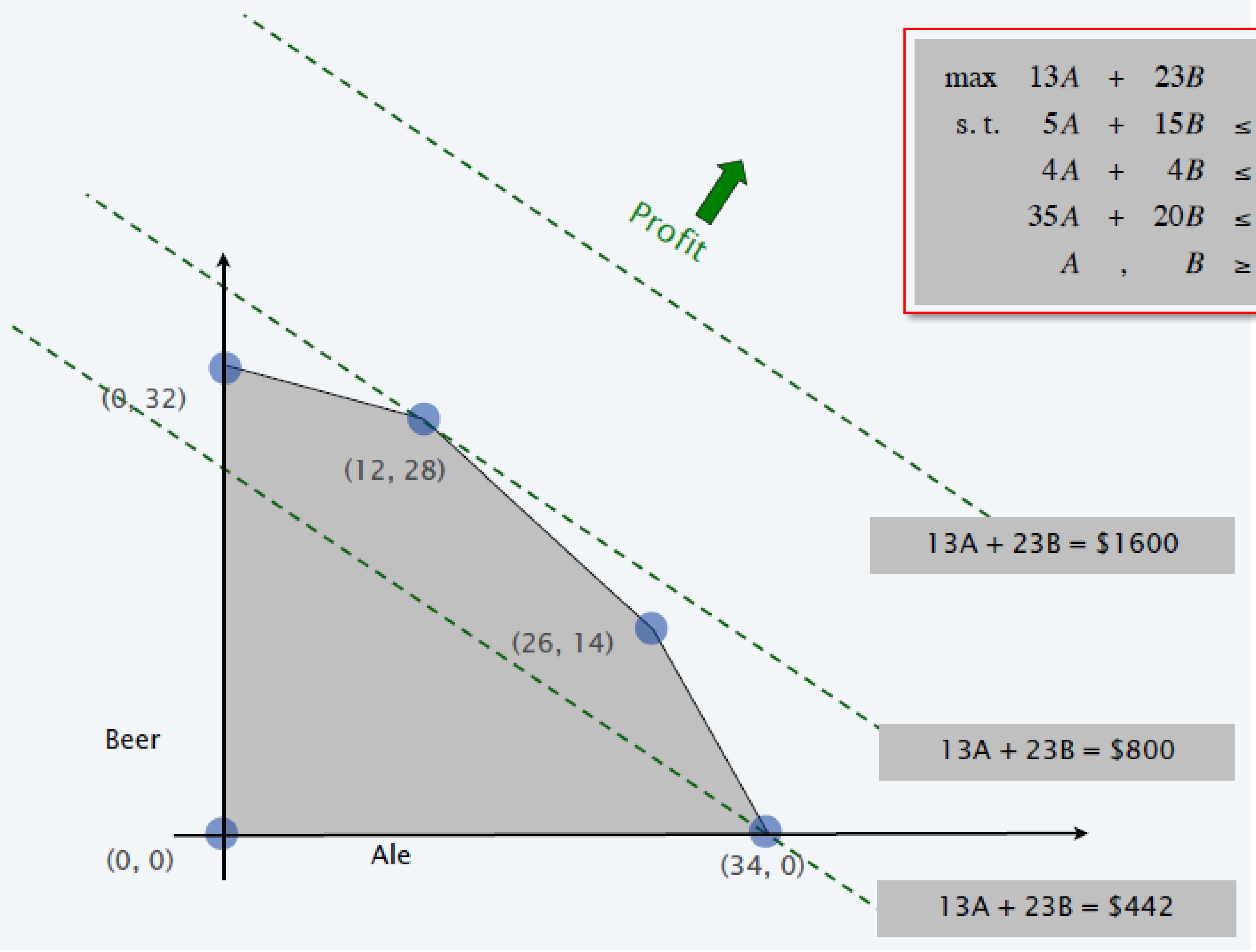
Brewery problem observation. Regardless of objective function coefficients, an optimal solution occurs at a vertex.



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

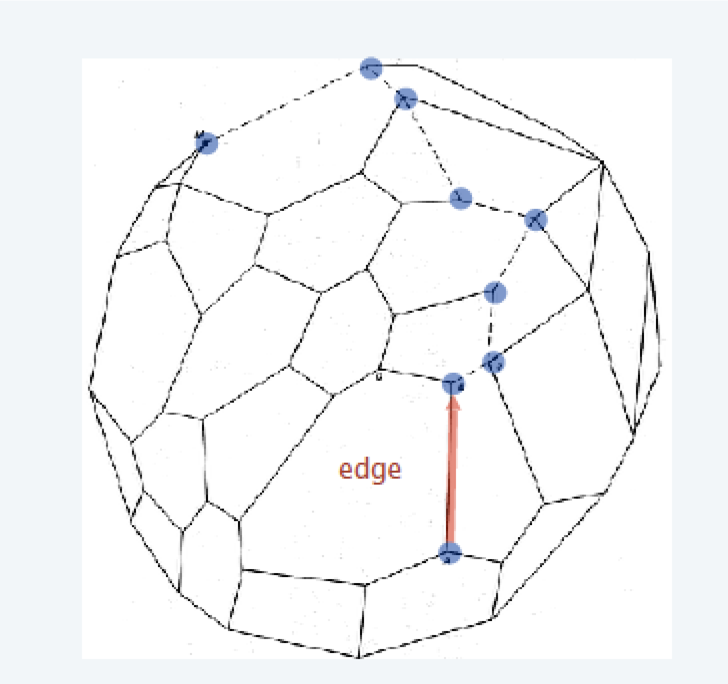
Brewery Problem: Objective Function

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



Simplex Algorithm: Intuition

Simplex algorithm. Move from BFS (Basic Feasible Solution) to adjacent BFS, without decreasing objective function (replace one basic variable with another).



Greedy property. BFS optimal iff no adjacent BFS is better.

Simplex Algorithm: Initialization

max Z subject to

$$
\begin{array}{rcrcrcrcrcrcl}
13A & + & 23B & & & & & & & - & Z & = & 0 \\
\hline
5A & + & 15B & + & S_C & & & & & & & = & 480 \\
4A & + & 4B & & & + & S_H & & & & & = & 160 \\
35A & + & 20B & & & & & + & S_M & & & = & 1190 \\
A & , & B & , & S_C & , & S_H & , & S_M & & & \geq & 0
\end{array}
$$

Basis = $\{S_C, S_H, S_M\}$

$A = B = 0$

$Z = 0$

$S_C = 480$

$S_H = 160$

$S_M = 1190$

Simplex Algorithm: Pivot 1

$$
\begin{array}{l}
\max Z \text{ subject to} \\
\begin{array}{rcrcccccccccr}
13A & + & 23B & & & & & & & - & Z & = & 0 \\
\hline
5A & + & 15B & + & S_C & & & & & & & = & 480 \\
4A & + & 4B & & & + & S_H & & & & & = & 160 \\
35A & + & 20B & & & & & + & S_M & & & = & 1190 \\
A & , & B & , & S_C & , & S_H & , & S_M & & & \geq & 0
\end{array}
\end{array}
$$

Basis = $\{S_C, S_H, S_M\}$
$A = B = 0$
$Z = 0$
$S_C = 480$
$S_H = 160$
$S_M = 1190$

Substitute: $B = 1/15\,(480 - 5A - S_C)$

$$
\begin{array}{l}
\max Z \text{ subject to} \\
\begin{array}{rcccrcccccccr}
\frac{16}{3}A & & & - & \frac{23}{15}S_C & & & & & - & Z & = & -736 \\
\hline
\frac{1}{3}A & + & B & + & \frac{1}{15}S_C & & & & & & & = & 32 \\
\frac{8}{3}A & & & - & \frac{4}{15}S_C & + & S_H & & & & & = & 32 \\
\frac{85}{3}A & & & - & \frac{4}{3}S_C & & & + & S_M & & & = & 550 \\
A & , & B & , & S_C & , & S_H & , & S_M & & & \geq & 0
\end{array}
\end{array}
$$

Basis = $\{B, S_H, S_M\}$
$A = S_C = 0$
$Z = 736$
$B = 32$
$S_H = 32$
$S_M = 550$

Simplex Algorithm: Pivot 1

max Z subject to

$$
\begin{array}{rcrcrcrcrcrcl}
13A & + & 23B & & & & & & & - & Z & = & 0 \\
\hline
5A & + & 15B & + & S_C & & & & & & & = & 480 \\
4A & + & 4B & & & + & S_H & & & & & = & 160 \\
35A & + & 20B & & & & & + & S_M & & & = & 1190 \\
A & , & B & , & S_C & , & S_H & , & S_M & & & \geq & 0
\end{array}
$$

Basis = $\{S_C, S_H, S_M\}$
$A = B = 0$
$Z = 0$
$S_C = 480$
$S_H = 160$
$S_M = 1190$

Q. Why pivot on column 2 (or 1)?
A. Each unit increase in B increases objective value by \$23.

Q. Why pivot on row 2.
A. Preserves feasibility by ensuring *RHS (Right Hand Side)* ≥ 0.
(min ratio rule: min{480/15, 160/4, 1190/20})

Simplex Algorithm: Pivot 2

max Z subject to

$$\begin{array}{rcrcrcrcrcrcl}
\frac{16}{3}A & & & - & \frac{23}{15}S_C & & & & & - & Z & = & -736 \\
\hline
\frac{1}{3}A & + & B & + & \frac{1}{15}S_C & & & & & & & = & 32 \\
\frac{8}{3}A & & & - & \frac{4}{15}S_C & + & S_H & & & & & = & 32 \\
\frac{85}{3}A & & & - & \frac{4}{3}S_C & & & + & S_M & & & = & 550 \\
A & , & B & , & S_C & , & S_H & , & S_M & & & \geq & 0
\end{array}$$

Basis = $\{B, S_H, S_M\}$
$A = S_C = 0$
$Z = 736$
$B = 32$
$S_H = 32$
$S_M = 550$

Substitute: $A = 3/8\,(32 + 4/15\ S_C - S_H)$

max Z subject to

$$\begin{array}{rcrcrcrcrcrcl}
 & & & - & S_C & - & 2\,S_H & & & - & Z & = & -800 \\
\hline
 & & B & + & \frac{1}{10}S_C & + & \frac{1}{8}S_H & & & & & = & 28 \\
A & & & - & \frac{1}{10}S_C & + & \frac{3}{8}S_H & & & & & = & 12 \\
 & & & - & \frac{25}{6}S_C & - & \frac{85}{8}S_H & + & S_M & & & = & 110 \\
A & , & B & , & S_C & , & S_H & , & S_M & & & \geq & 0
\end{array}$$

Basis = $\{A, B, S_M\}$
$S_C = S_H = 0$
$Z = 800$
$B = 28$
$A = 12$
$S_M = 110$

Simplex Algorithm: Optimality

Q. When to stop pivoting?

A. When all coefficients in top row are non-positive.

Q. Why is the resulting solution optimal?

A. Any feasible solution satisfies systems of equations in tableau.

- In particular: $Z = 800 - S_C - 2S_H$, $S_C \geq 0$, $S_H \geq 0$.
- Thus, optimal objective value $Z^* \leq 800$.
- Current BFS has value 800 -> optimal.

max Z subject to

$$
\begin{array}{ccccccccccccc}
 & & & - & S_C & - & 2\,S_H & & & - & Z & = & -800 \\
\hline
 & & B & + & \frac{1}{10} S_C & + & \frac{1}{8} S_H & & & & & = & 28 \\
A & & & - & \frac{1}{10} S_C & + & \frac{3}{8} S_H & & & & & = & 12 \\
 & & & - & \frac{25}{6} S_C & - & \frac{85}{8} S_H & + & S_M & & & = & 110 \\
A & , & B & , & S_C & , & S_H & , & S_M & & & \geq & 0
\end{array}
$$

Basis = $\{A, B, S_M\}$
$S_C = S_H = 0$
$Z = 800$
$B = 28$
$A = 12$
$S_M = 110$

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

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

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	
$\ldots$	$\ldots$				$\ldots\ldots$				$\ldots$	
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.
7. If all entries in the bottom row are non-positive, this the final tableau. If not, go back to Step 3.

Variant Tableau: An Example

$$\max \quad z = 2x_1 + 3x_2$$

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

$$\Rightarrow$$

$$\max \quad z = 2x_1 + 3x_2$$

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

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.

Revisit:

Dynamic Programming

Algorithmic Paradigms

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

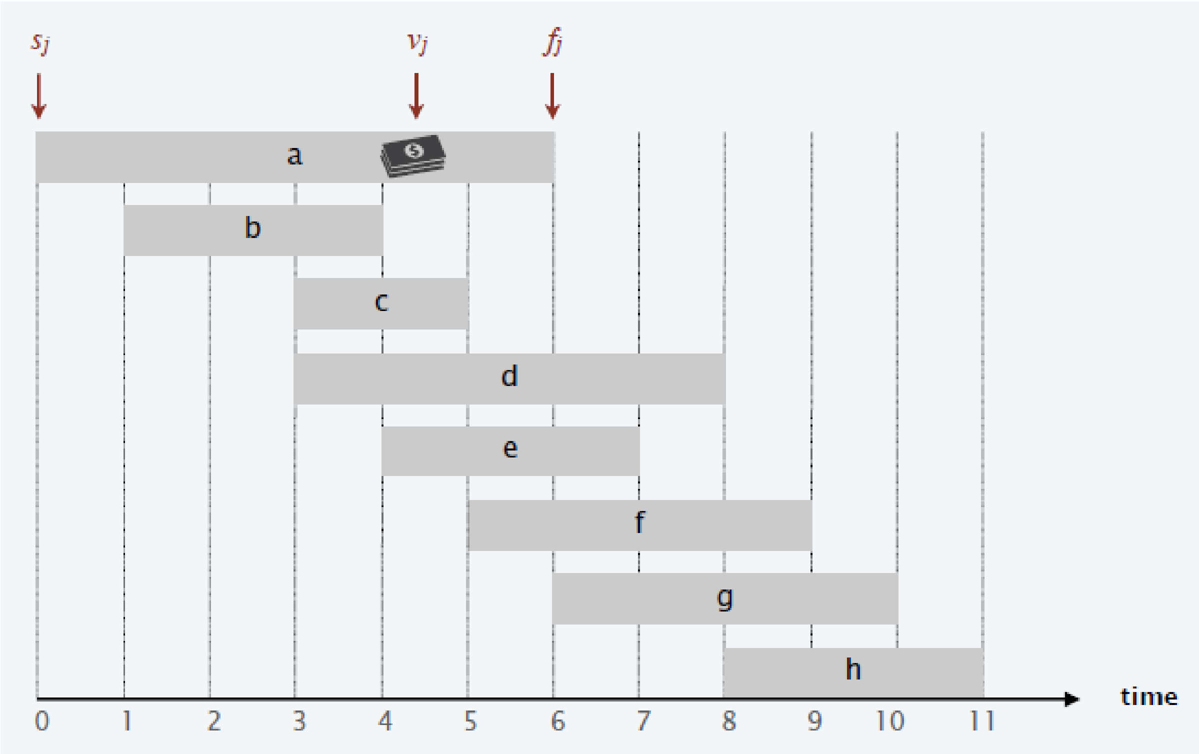
Divide-and-conquer. Break up a problem into independent sub-problems, solve each sub-problem, and combine solutions to sub-problems to form solution to original problem.

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).

Weighted Interval Scheduling

Weighted interval scheduling problem.

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



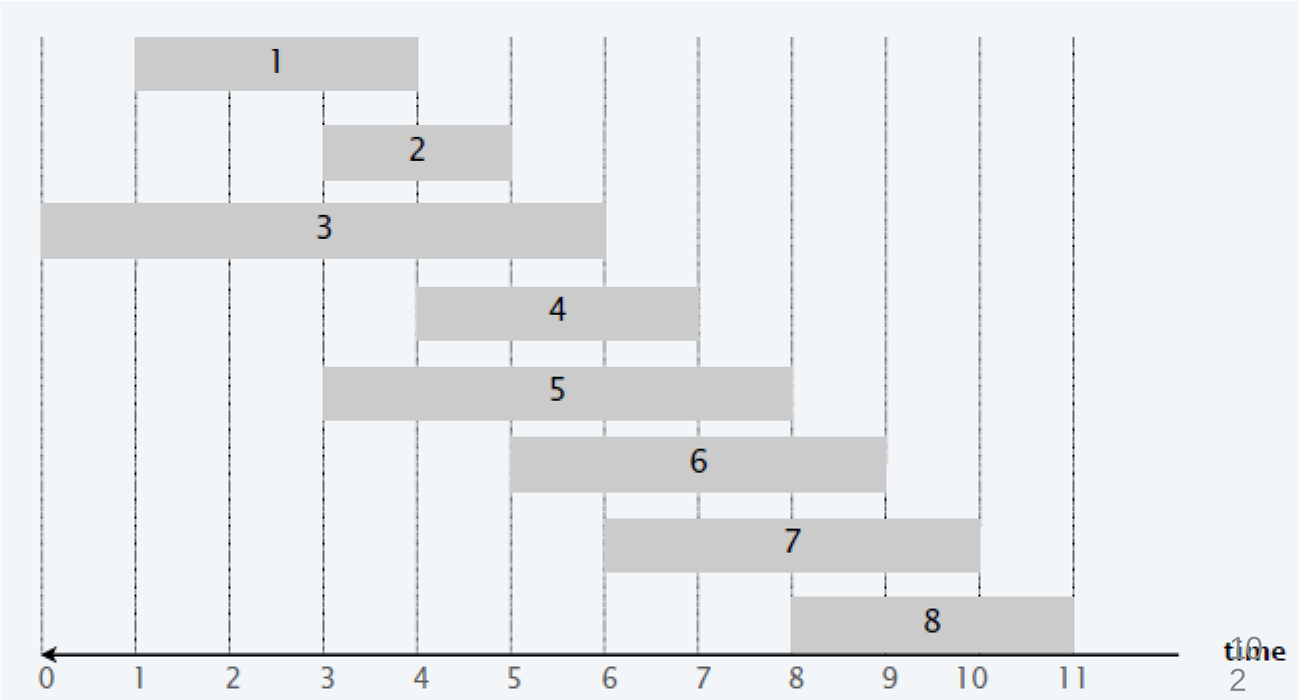
Weighted Interval Scheduling

Notation. Label jobs by finishing time: $f_1 \leq f_2 \leq \cdots \leq f_n$.

Def. $p(j)$ = largest index $i < j$ such that job i is compatible with job j.

Ex.

$p(1) = 0,$
$p(2) = 0,$
$p(3) = 0,$
$p(4) = 1,$
$p(5) = 0,$
$p(6) = 2,$
$p(7) = 3,$
$p(8) = 5.$



Dynamic Programming: Binary Choice

Notation. $OPT(j)$ = value of optimal solution to the problem consisting of job requests 1, 2, ..., j.

Goal. $OPT(n)$ = value of optimal solution to the original problem.

Case 1. $OPT(j)$ selects job j.

- Collect profit v_j.
- Can't use incompatible jobs $\{p(j)+1, p(j)+2, \ldots, j-1\}$.
- Must include optimal solution to the problem consisting of remaining compatible jobs 1, 2, ..., $p(j)$.

Dynamic Programming: Binary Choice

Notation. $OPT(j)$ = value of optimal solution to the problem consisting of job requests 1, 2, ..., j.

Goal. $OPT(n)$ = value of optimal solution to the original problem.

Case 2. $OPT(j)$ does not selects job j.

- Must include optimal solution to the problem consisting of remaining jobs 1, 2, ..., $j-1$.

$$OPT(j) = \begin{cases} 0 & if\ j = 0 \\ \max\{v_j + OPT(p(j)), OPT(j-1)\} & otherwise \end{cases}$$

Weighted Interval Scheduling: Memorization

Top-down dynamic programming (memorization). Cache result of each sub-problem; lookup as needed.

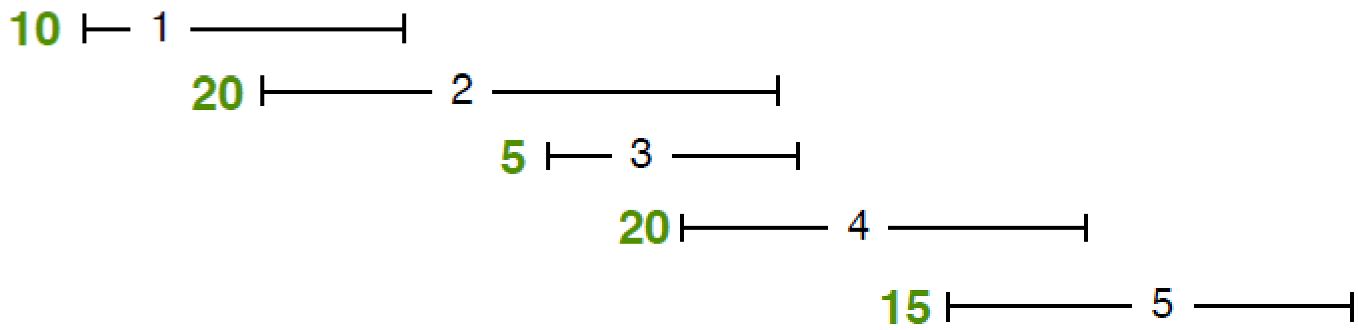
Top-Down $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n, v_1, v_2, \ldots, v_n)$

Sort jobs by finish time so that $f_1 \leq f_2 \leq \cdots \leq f_n$
Compute $p[1], p[2], \ldots, p[n]$.
$M[0] \leftarrow 0$.
Return M-Compute-Opt(n).

M-Compute-Opt(j)

If $M[j] = uninitialized$
   $M[j] \leftarrow \max\{\, v_j +$ M-Compute-Opt$(p[j])$, M-Compute-Opt$(j-1)\}$.
Return $M[j]$

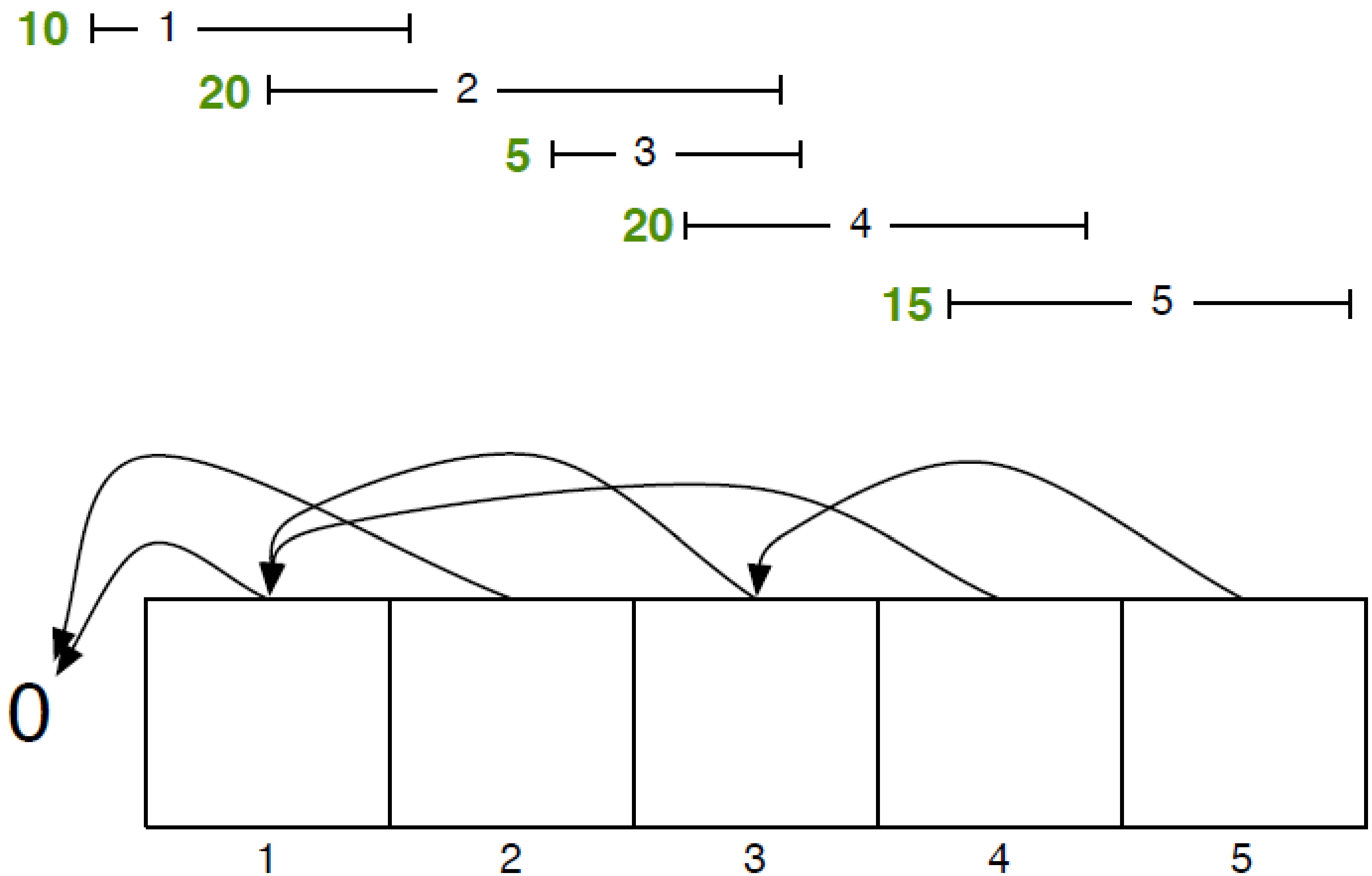
Weighted Interval Scheduling: Demo



Bottom-Up $(n, s_1, s_2, \dots, s_n, f_1, f_2, \dots, f_n, v_1, v_2, \dots, v_n)$

Sort jobs by finish time so that $f_1 \leq f_2 \leq \cdots \leq f_n$
Compute $p[1], p[2], \dots, p[n]$.
$M[0] \leftarrow 0.$
For $j = 1$ To n
  $M[j] \leftarrow \max\{ v_j + M\big[p[j]\big],\ M[j-1]\}.$

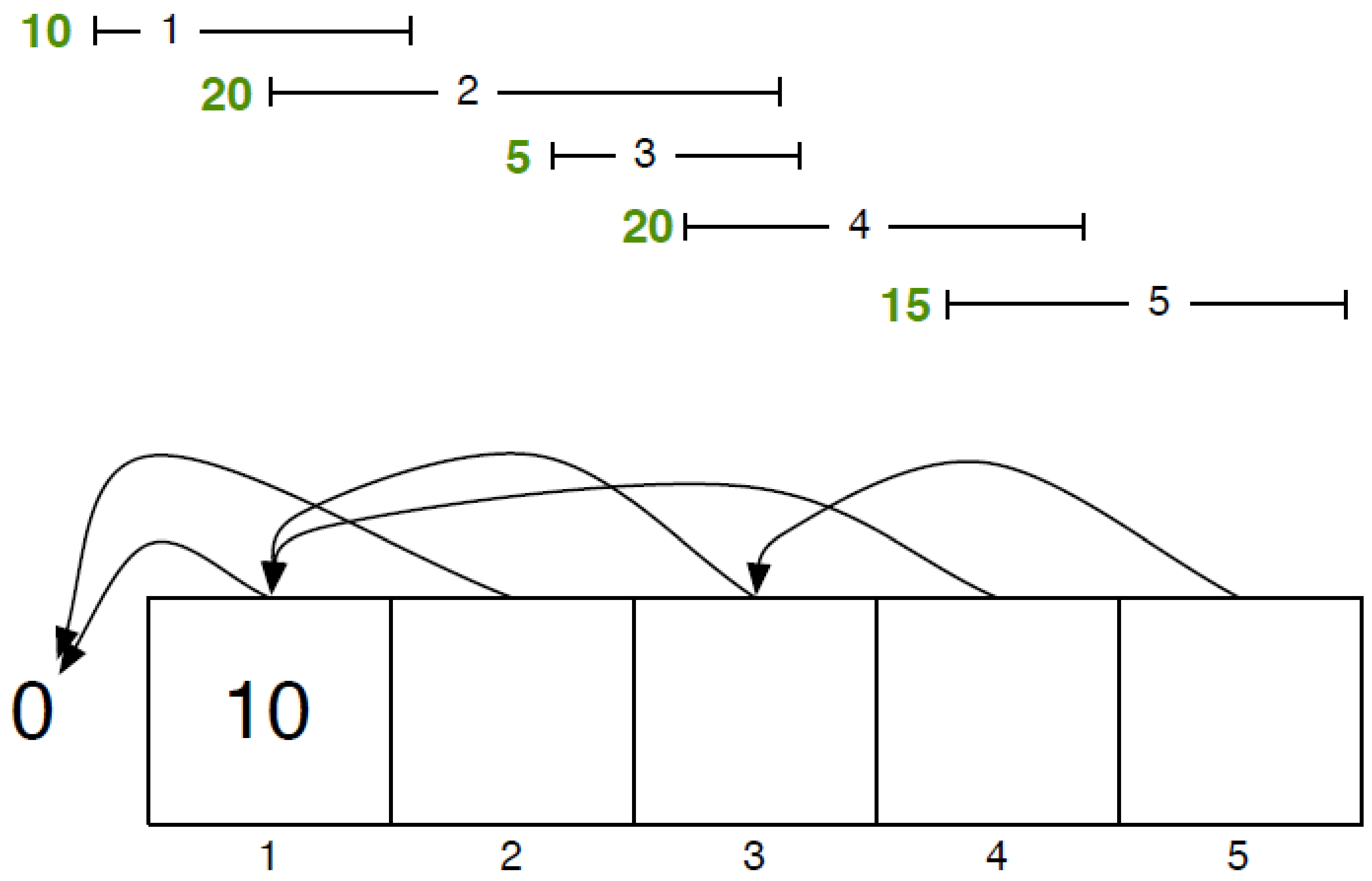
Weighted Interval Scheduling: Demo



$v_j + M[p(j)]$

$M[j-1]$

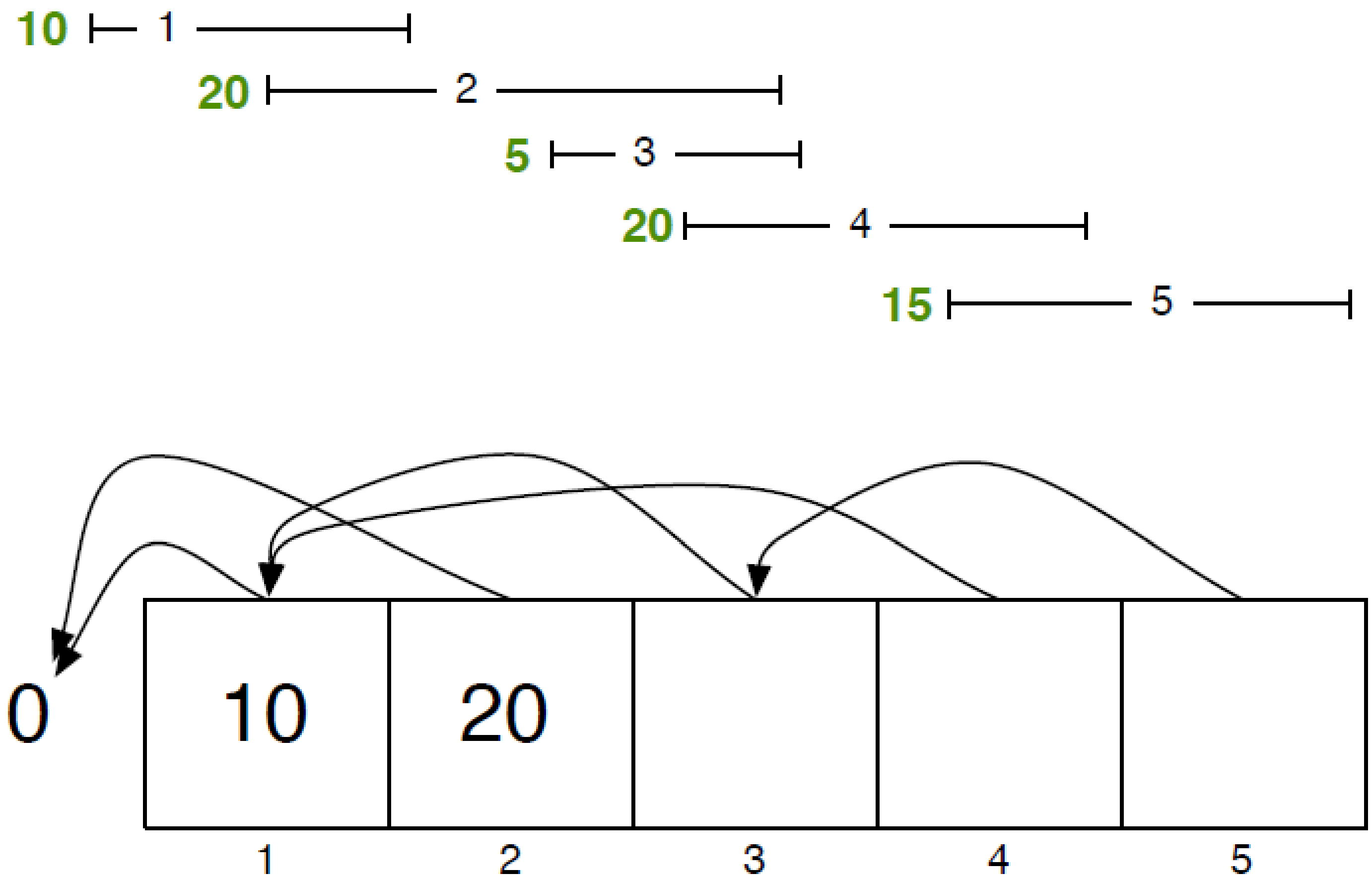
Weighted Interval Scheduling: Demo



$v_j + M[p(j)]$ 10

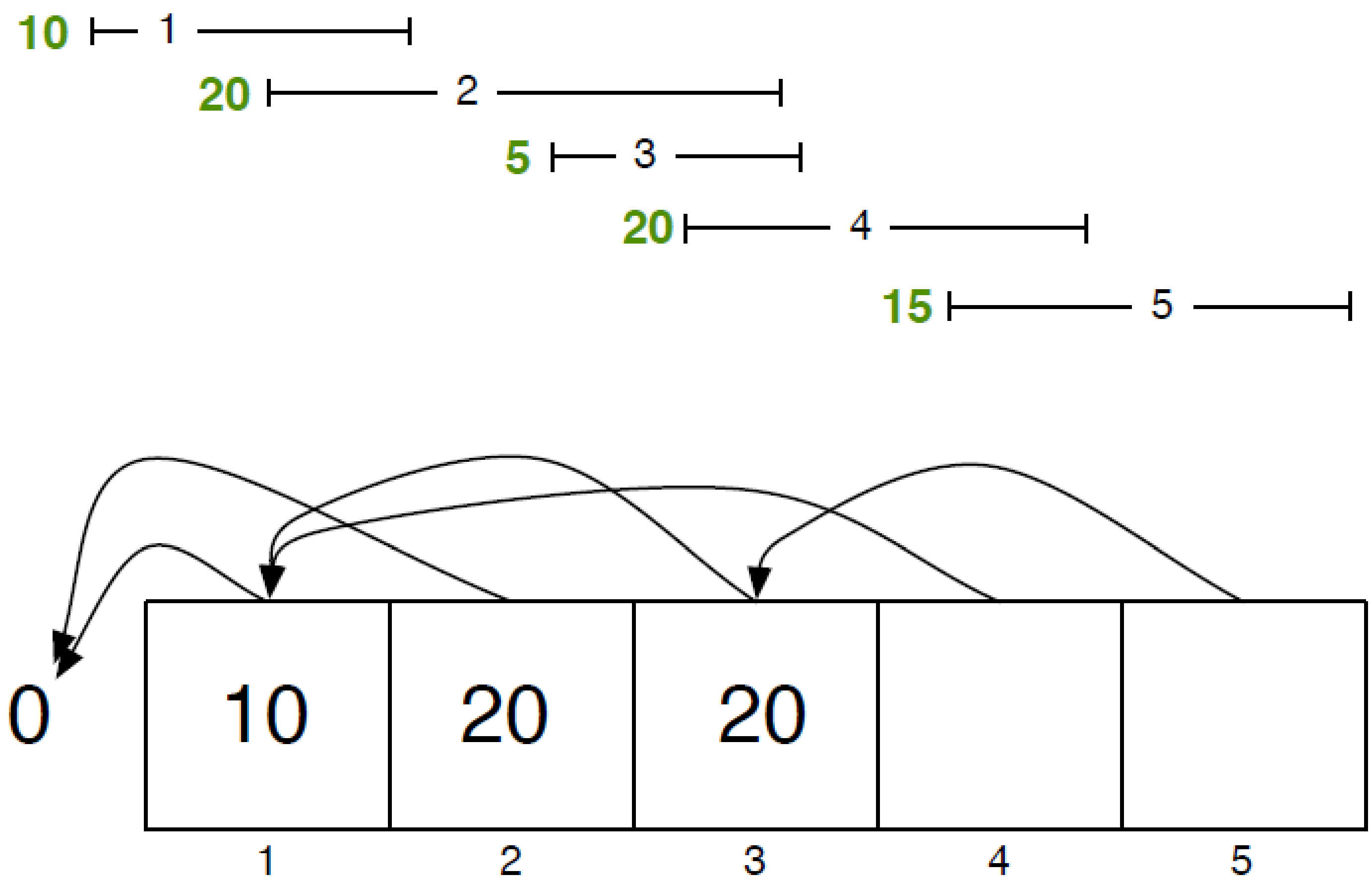
$M[j-1]$ 0

Weighted Interval Scheduling: Demo



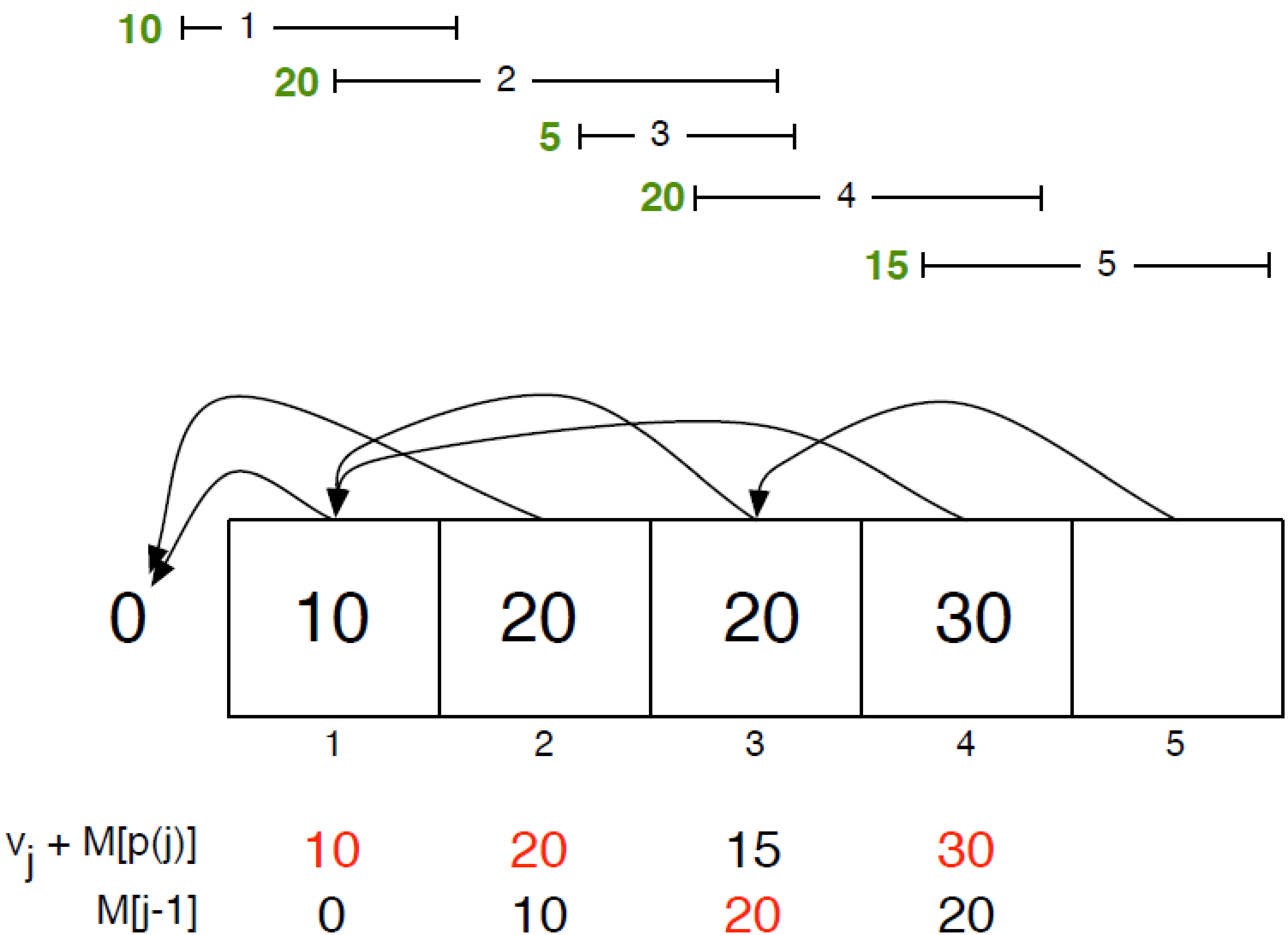
$v_j + M[p(j)]$	10	20
$M[j-1]$	0	10

Weighted Interval Scheduling: Demo

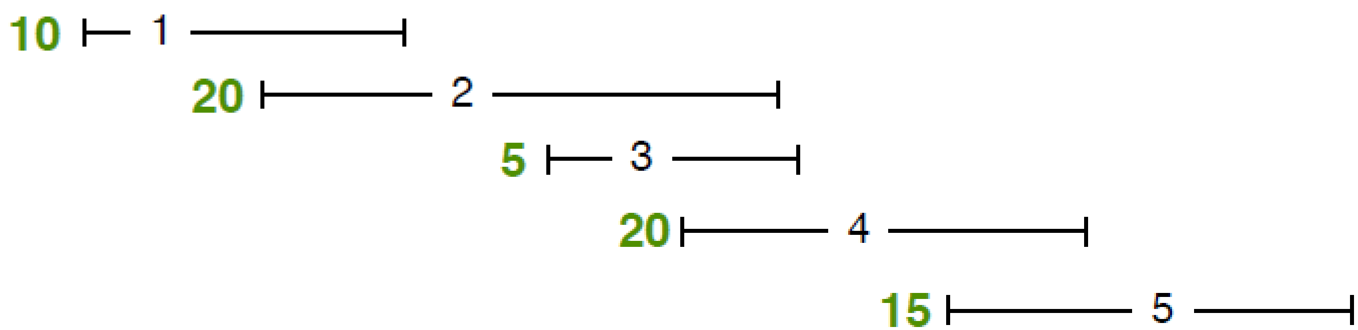


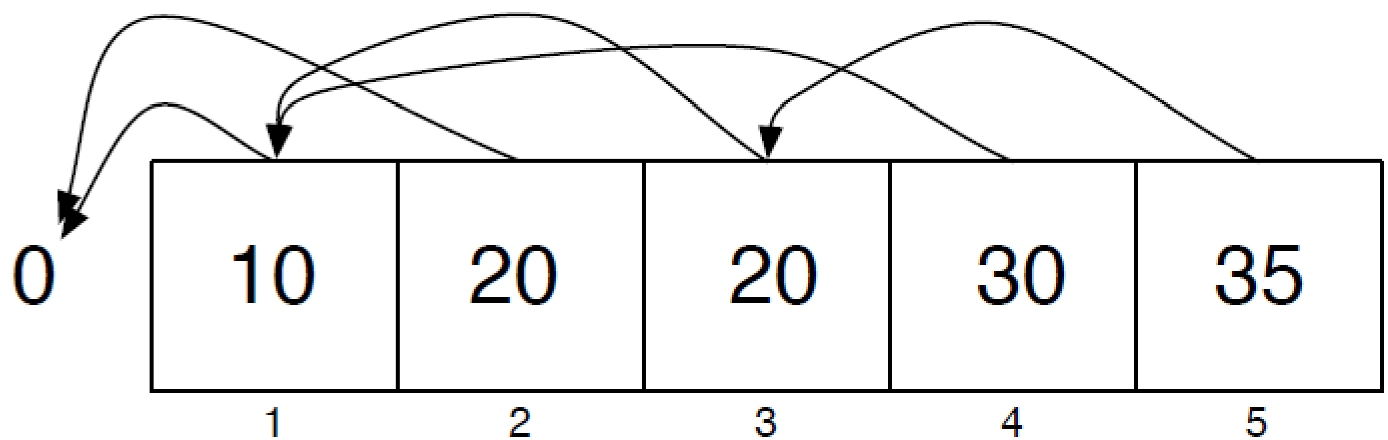
$v_j + M[p(j)]$	10	20	15
$M[j-1]$	0	10	20

Weighted Interval Scheduling: Demo



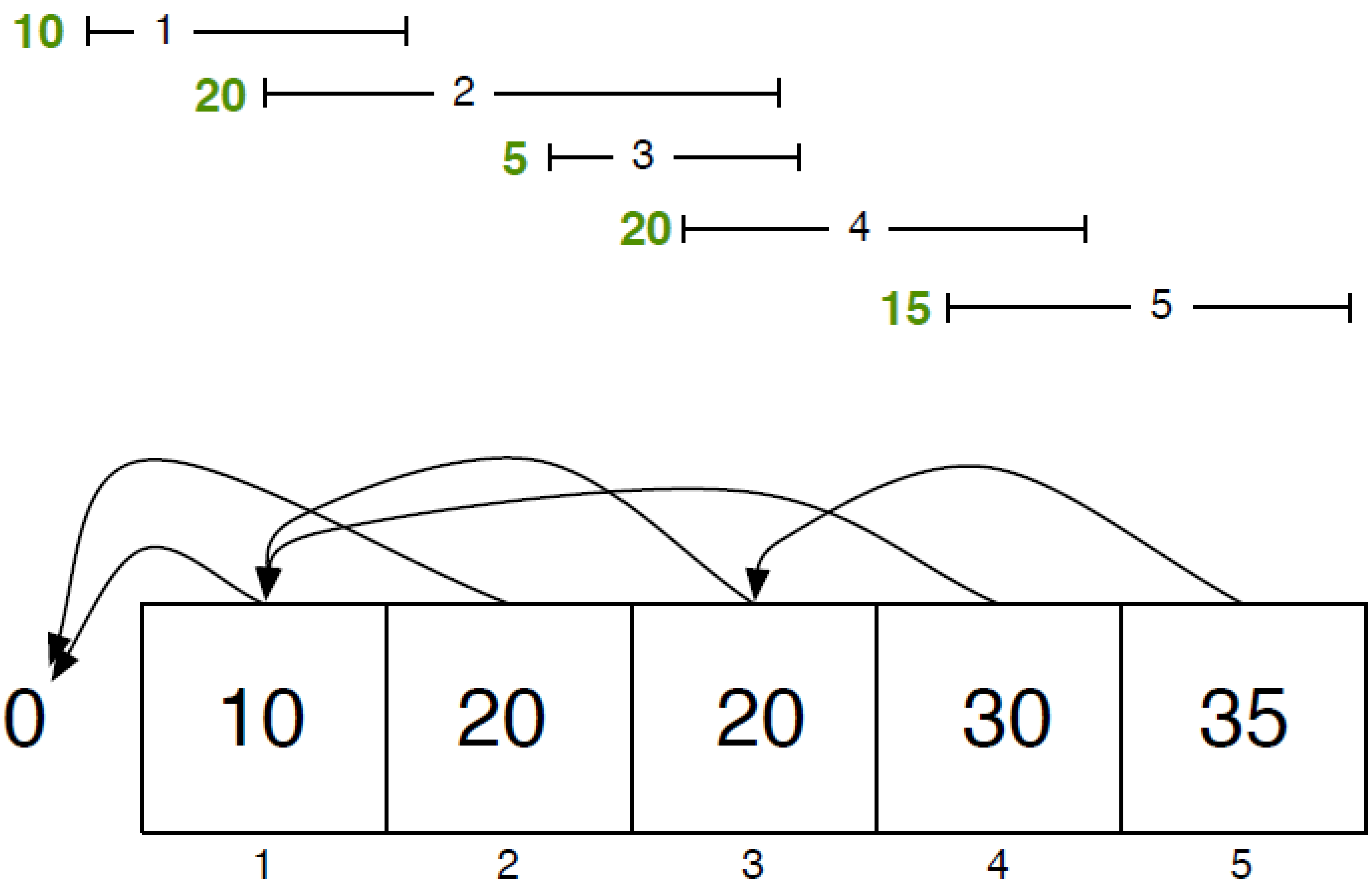
Weighted Interval Scheduling: Demo





$v_j + M[p(j)]$	10	20	15	30	35
$M[j-1]$	0	10	20	20	30

Weighted Interval Scheduling: Demo



	1	2	3	4	5
$v_j + M[p(j)]$	10	20	15	30	35
$M[j-1]$	0	10	20	20	30

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.

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.

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)

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: 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 $(n, W, w_1, w_2, \dots, w_n, v_1, v_2, \dots, v_n)$

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)

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

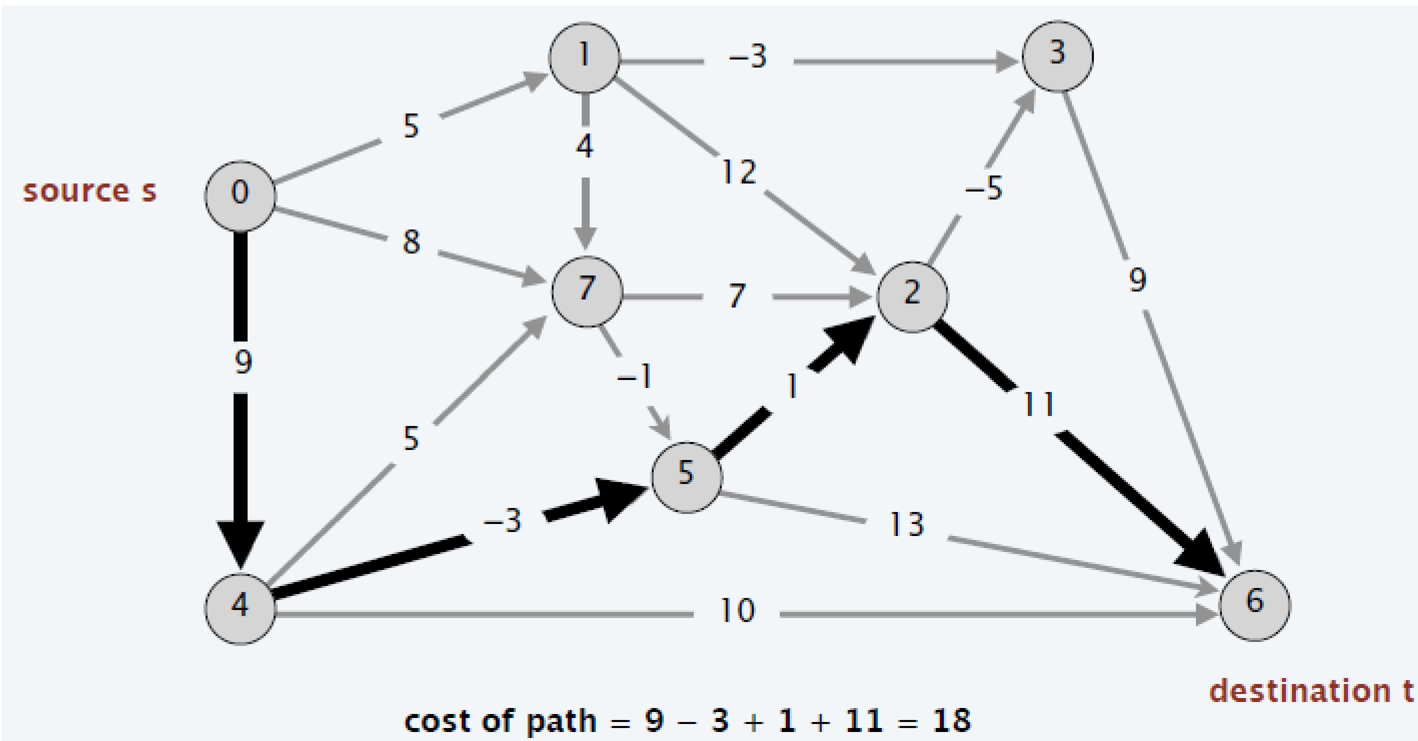
weight limit w

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.

Shortest Paths

Shortest-path problem. Given a digraph $G = (V, E)$, with arbitrary edge weights or cost c_{vw}, find cheapest path from node s to node t.



Shortest Paths: Dynamic Programming

Def. $OPT(i, v)$ = cost of shortest $v \to t$ path that uses $\leq i$ edges.

- Case 1: Cheapest $v \to t$ path uses $\leq i - 1$ edges.
  - $OPT(i, v) = OPT(i - 1, v)$.
- Case 2: Cheapest $v \to t$ path uses exactly i edges.
  - If (v, w) is the first edge, then OPT uses (v, w), and then selects best $w \to t$ path using $\leq i - 1$ edges.

$$OPT(i, v) = \begin{cases} \infty & if\ i = 0 \\ \min\left\{OPT(i-1, v), \min\limits_{(v,w)\in E}\{OPT(i-1, w) + c_{vw}\}\right\} & otherwise \end{cases}$$

Observation. If no negative cycles, $OPT(n - 1, v)$ = cost of cheapest $v \to t$ path.

Shortest Paths: Implementation

Shortest-Paths (V, E, c, t)

For each node $v \in V$

$\quad M[0, v] \leftarrow \infty.$

$M[0, t] \leftarrow 0.$

For $i = 0$ To $n - 1$

$\quad$For each node $v \in V$

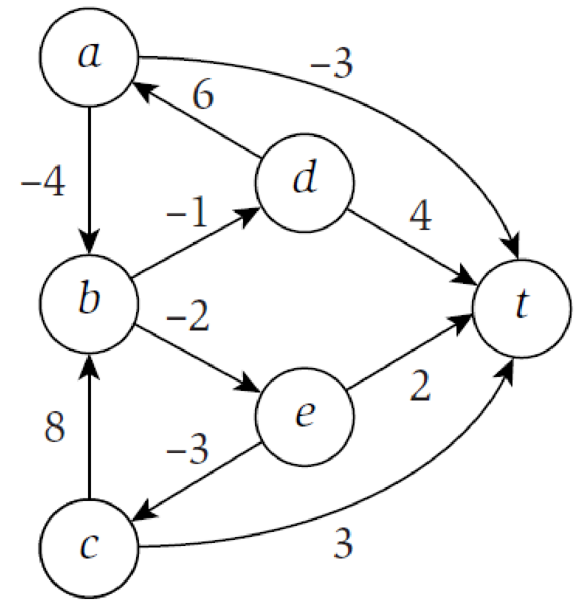
$\quad\quad M[i, v] \leftarrow M[i - 1, v].$

$\quad\quad$For each edge $(v, w) \in E$

$\quad\quad\quad M[i, v] \leftarrow \min\{M[i, v], M[i - 1, w] + c_{vw}\}.$

Shortest Paths: An Example

Ex. Considering the following directed graph, find a shortest path from each node to t.



Shortest-Paths (V, E, c, t)

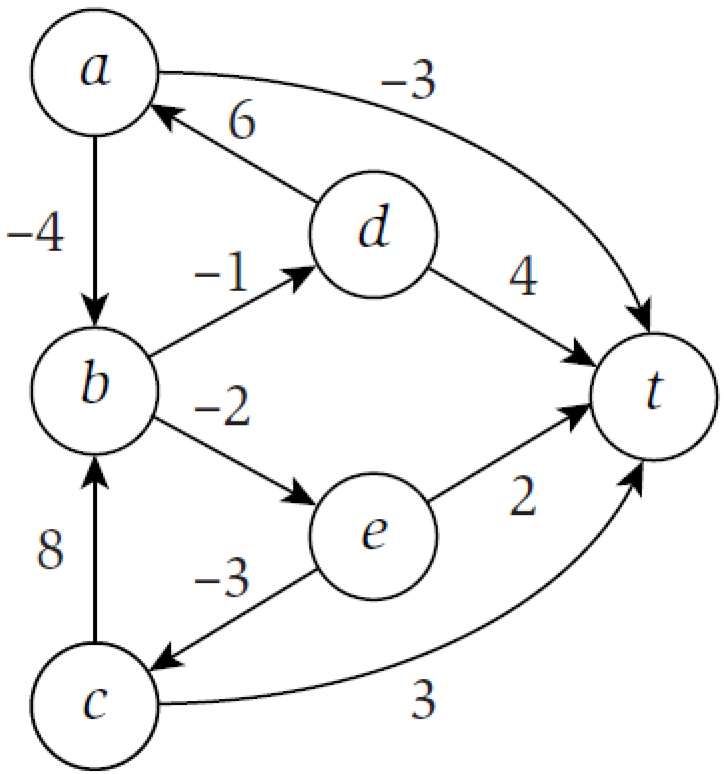
For each node $v \in V$
  $M[0, v] \leftarrow \infty$.
$M[0, t] \leftarrow 0$.
For $i = 0$ To $n - 1$
  For each node $v \in V$
    $M[i, v] \leftarrow M[i-1, v]$.
  For each edge $(v, w) \in E$
    $M[i, v] \leftarrow \min\{M[i, v], M[i-1, w] + c_{vw}\}$.

	0	1	2	3	4	5
t						
a						
b						
c						
d						
e						

Shortest Paths: An Example

Ex. Considering the following directed graph, find a shortest path from each node to t.

	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	−3	−3	−4	−6	−6
b	∞	∞	0	−2	−2	−2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0



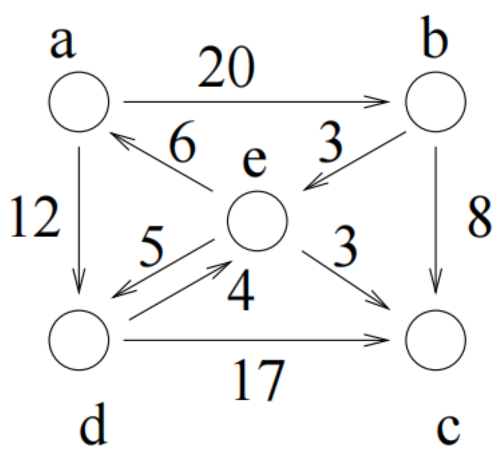
Each row corresponds to the shortest path from a node to t, as we allow the path to use an increasing number of edges

All-Pairs Shortest Paths

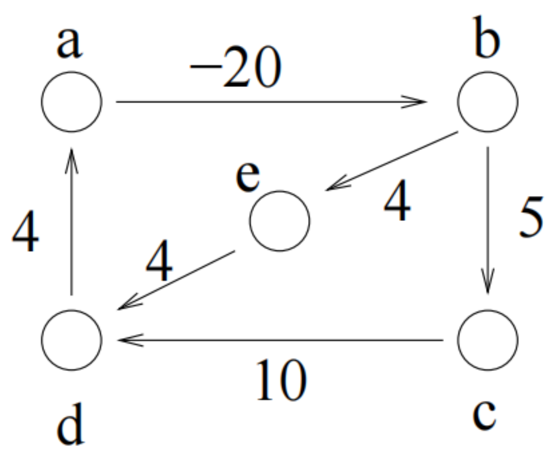
Input: weighted digraph $G = (V, E)$ with weight function $w : E \rightarrow \mathbb{R}$

Find: lengths of the shortest paths (i.e., distance) between all pairs of vertices in G.

- we assume that there are no cycles with zero or negative cost.



without negative cost cycle



with negative cost cycle

Input and Output Formats

Input Format:

- To simplify the notation, we assume that
  $V = \{1, 2, \ldots, n\}$.
- Adjacency matrix: graph is represented by an n x n matrix containing edge weights

$$w_{ij} = \begin{cases} 0 & if \;\; i = j, \\ w(i,j) & if \;\; i \neq j \; and \; (i,j) \in E, \\ \infty & if \;\; i \neq j \; and \; (i,j) \notin E. \end{cases}$$

Output Format: an $n \times n$ matrix $D = [d_{ij}]$ in which d_{ij} is the length of the shortest path from vertex i to j.

Step 1: Space of Subproblems

For m = 1, 2, 3, ...

Define $d_{ij}^{(m)}$ to be the length of the shortest path from i to j that contains at most m edges.

Let $D^{(m)}$ be the $n \times n$ matrix $[d_{ij}^{(m)}]$

We will see (next page) that solution D satisfies $D=D^{n-1}$.

Subproblems: (Iteratively)compute $D^{(m)}$for m=1,...,n-1.

Step 1: Space of Subproblems

Lemma

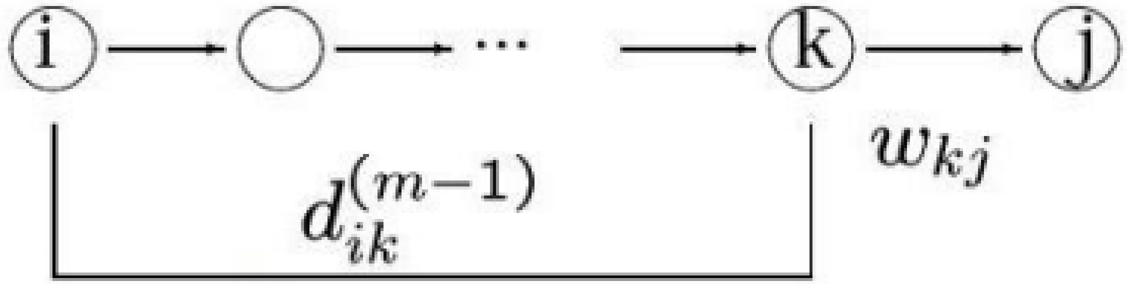
- $D^{(n-1)} = D$
- $d_{ij}^{(n-1)}$ = true distance from i to j

Proof

- We prove that any shortest path P from i to j contains at most n – 1 edges.
- First note that since all cycles have positive weight, a shortest path can have no cycles (if there were a cycle, we could remove it and lower the length of the path).
- A path without cycles can have length at most n – 1 (since a longer path must contain some vertex twice, that is, contain a cycle).

Step 2: Building $D^{(m)}$ from $D^{(m-1)}$

Consider a shortest path from i to j that contains at most m edges.



Let k be the vertex imriately before j on the shortest path.

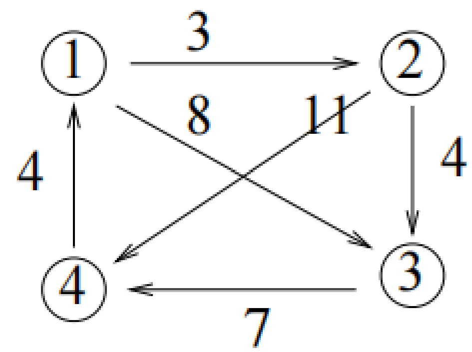
The sub-path from i to k must be the shortest 1-k path with at most m-1 edges: $d_{ij}^{(m)} = d_{ik}^{(m-1)} + w_{kj}$

Since we don't know k,we try all possible choices:

$$d_{ij}^{(m)} = \min_{1 \le k \le n}\{d_{ik}^{(m-1)} + w_{kj}\}$$

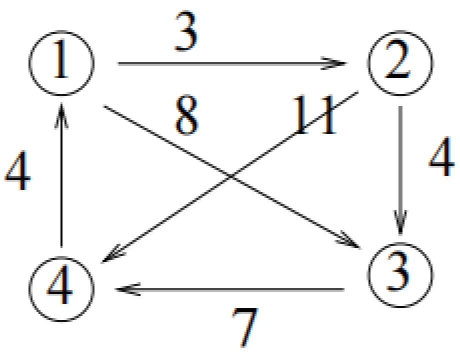
Example: Bottom-up Computation of $D^{(m-1)}$

$D^{(1)} = [w_{ij}]$ is just the weight matrix:



Example: Bottom-up Computation of $D^{(m-1)}$

$D^{(1)} = [w_{ij}]$ is just the weight matrix:

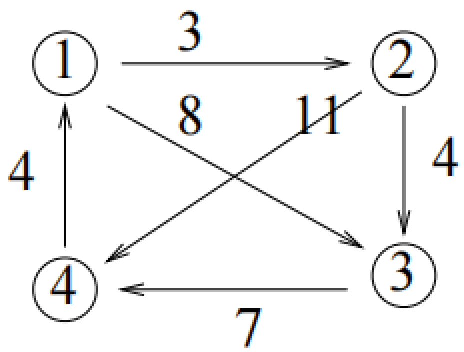


$$D^{(1)} = \begin{bmatrix} 0 & 3 & 8 & \infty \\ \infty & 0 & 4 & 11 \\ \infty & \infty & 0 & 7 \\ 4 & \infty & \infty & 0 \end{bmatrix}$$

$$d_{ij}^{(2)} = \min_{1 \le k \le 4} \{ d_{ik}^{(1)} + w_{kj} \}$$

Example: Bottom-up Computation of $D^{(m-1)}$

$D^{(1)} = [w_{ij}]$ is just the weight matrix:



$$D^{(1)} = \begin{bmatrix} 0 & 3 & 8 & \infty \\ \infty & 0 & 4 & 11 \\ \infty & \infty & 0 & 7 \\ 4 & \infty & \infty & 0 \end{bmatrix}$$

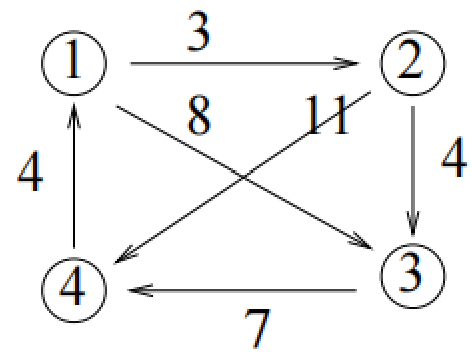
$$d_{ij}^{(2)} = \min_{1 \le k \le 4} \{ d_{ik}^{(1)} + w_{kj} \}$$

$$D^{(2)} = \begin{bmatrix} 0 & 3 & 7 & 14 \\ 15 & 0 & 4 & 11 \\ 11 & \infty & 0 & 7 \\ 4 & 7 & 12 & 0 \end{bmatrix}$$

Example: Bottom-up Computation of $D^{(m-1)}$

$D^{(1)} = [w_{ij}]$ is just the weight matrix:

$$D^{(1)} = \begin{bmatrix} 0 & 3 & 8 & \infty \\ \infty & 0 & 4 & 11 \\ \infty & \infty & 0 & 7 \\ 4 & \infty & \infty & 0 \end{bmatrix}$$



$$\color{red}{d_{ij}^{(2)}} = \min_{1 \le k \le 4} \left\{ d_{ik}^{(1)} + w_{kj} \right\}$$

$$\color{red}{d_{ij}^{(3)}} = \min_{1 \le k \le 4} \left\{ d_{ik}^{(2)} + w_{kj} \right\}$$

$$D^{(2)} = \begin{bmatrix} 0 & 3 & 7 & 14 \\ 15 & 0 & 4 & 11 \\ 11 & \infty & 0 & 7 \\ 4 & 7 & 12 & 0 \end{bmatrix}$$

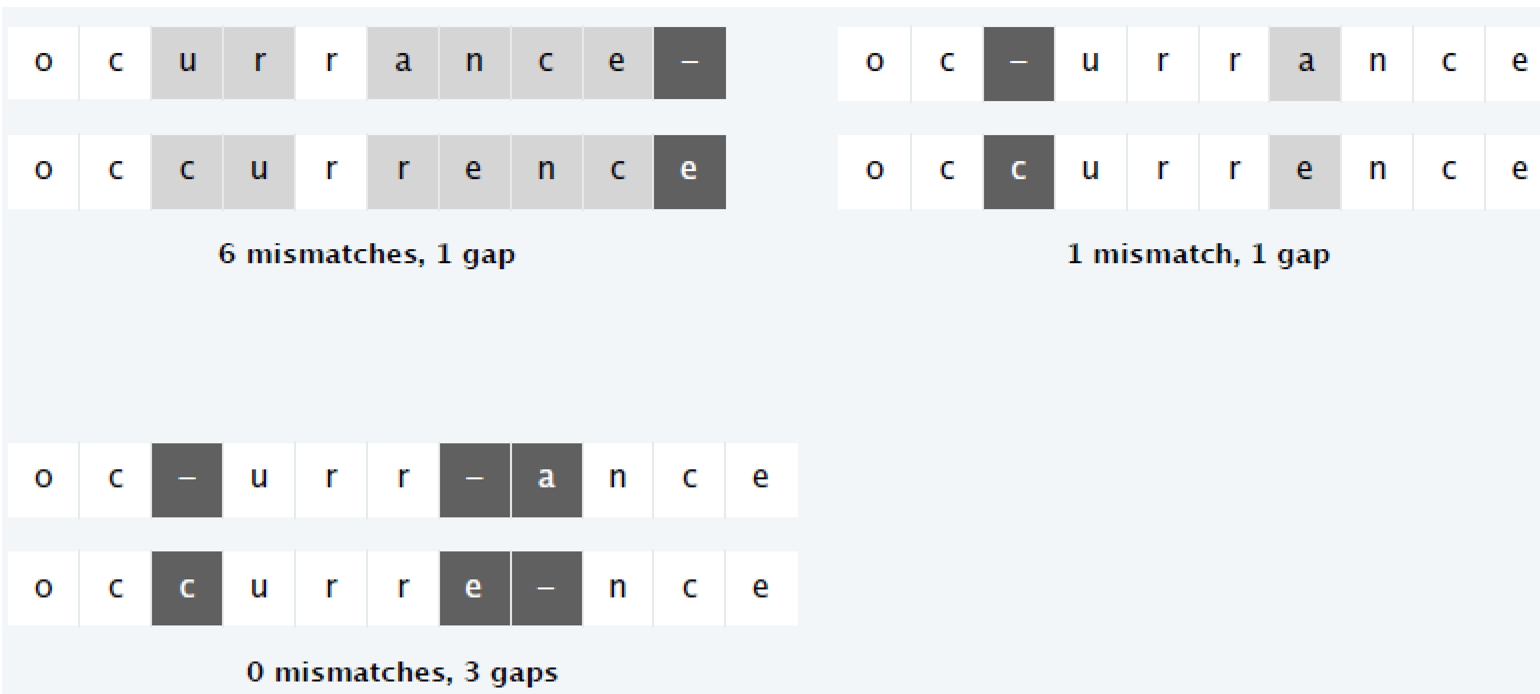
$$D^{(3)} = \begin{bmatrix} 0 & 3 & 7 & 14 \\ 15 & 0 & 4 & 11 \\ 11 & 14 & 0 & 7 \\ 4 & 7 & 11 & 0 \end{bmatrix}$$

$D^{(3)}$ gives the distances between any pair of vertices.

String Similarity

Q. How similar are two strings?

Ex. ocurrance & occurrence.



Edit Distance

Edit distance.

- Gap penalty δ; mismatch penalty α_{pg}.
- Cost = sum of gap and mismatch penalties.

C	T	–	G	A	C	C	T	A	C	G
C	T	G	G	A	C	G	A	A	C	G

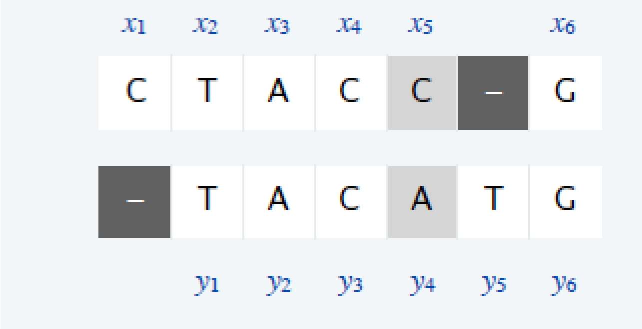
$$\text{cost} = \delta + \alpha_{CG} + \alpha_{TA}$$

Applications. Speech recognition, computational biology,...

Sequence Alignment

Goal. Given two strings $x_1 x_2 \ldots x_m$ and $y_1 y_2 \ldots y_n$ find a min-cost alignment.

Def. An alignment M is a set of ordered pairs $x_i - y_j$ such that each item occurs in at most one pair and no crossings ($x_i - y_j$ and $x_h - y_k$ cross if $i < h$, but $j > k$).



an alignment of CTACCG and TACATG:
$M = \{ x_2–y_1, x_3–y_2, x_4–y_3, x_5–y_4, x_6–y_6 \}$

Sequence Alignment

Goal. Given two strings $x_1 x_2 \dots x_m$ and $y_1 y_2 \dots y_n$ find a min-cost alignment.

Def. An alignment M is a set of ordered pairs $x_i - y_j$ such that each item occurs in at most one pair and no crossings ($x_i - y_j$ and $x_h - y_k$ cross if $i < h$, but $j > k$).

Def. The cost of an alignment M is:

$$cost(M) = \underbrace{\sum_{(x_i, y_j) \in M} \alpha_{x_i y_j}}_{\text{mismatch}} + \underbrace{\sum_{i: x_i \text{ unmatched}} \delta + \sum_{j: y_j \text{ unmatched}} \delta}_{\text{gap}}$$

Sequence Alignment: Problem Structure

Def. $OPT(i,j)$ = min cost of aligning prefix strings $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_j$.
Goal. $OPT(m,n)$.

Case 1. $OPT(i,j)$ includes $x_i - y_j$.
Pay mismatch for $x_i - y_j$ + min cost of aligning $x_1 x_2 \ldots x_{i-1}$ and $y_1 y_2 \ldots y_{j-1}$.

Case 2a. $OPT(i,j)$ leaves x_i unmatched.
Pay gap for x_i + min cost of aligning $x_1 x_2 \ldots x_{i-1}$ and $y_1 y_2 \ldots y_j$.

Sequence Alignment: Problem Structure

Def. $OPT(i,j)$ = min cost of aligning prefix strings $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_j$.
Goal. $OPT(m,n)$.

Case 2b. $OPT(i,j)$ leaves y_j unmatched.
Pay gap for y_j + min cost of aligning $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_{j-1}$.

$$OPT(i,j) = \begin{cases} j\delta & if\ i = 0 \\ min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases} & otherwise \\ i\delta & if\ j = 0 \end{cases}$$

Sequence Alignment: Bottom-Up Algorithm

Sequence-Alignment $(m, n, x_1, \ldots, x_m, y_1, \ldots, y_n, \delta, \alpha)$

For $i = 0$ To m

$M[i, 0] \leftarrow i\delta.$

For $j = 0$ To n

$M[0, j] \leftarrow j\delta.$

For $i = 1$ To m

For $j = 1$ To n

$$M[i, j] \leftarrow \min\{\alpha[x_i, y_j] + M[i-1, j-1], \delta + M[i-1, j], \delta + M[i, j-1]\}.$$

Return $M[m, n]$.

Sequence Alignment: An Example

Ex. Align the words *mean* and *name*. Assume that $\delta = 2$; matching a vowel with a different vowel, or a consonant with a different consonant, costs 1; while matching a vowel, or a consonant with each other costs 3.

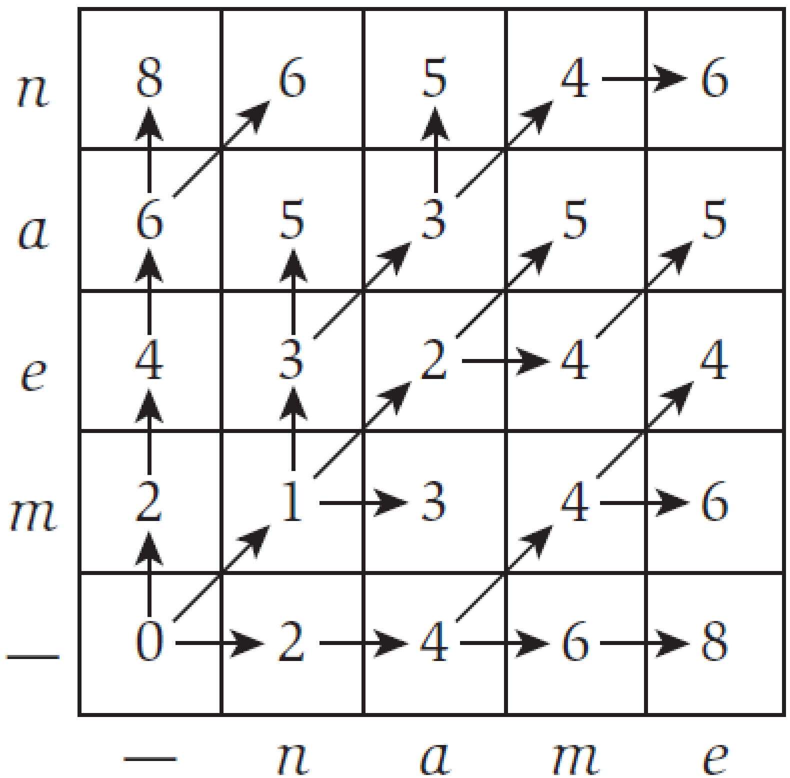
Sequence-Alignment $(m, n, x_1, \dots, x_m, y_1, \dots, y_n, \delta, \alpha)$

For $i = 0$ To m
  $M[i, 0] \leftarrow i\delta.$
For $j = 0$ To n
  $M[0, j] \leftarrow j\delta.$
For $i = 1$ To m
  For $j = 1$ To n
    $M[i, j] \leftarrow \min\{\alpha[x_i, y_j] + M[i-1, j-1],$
      $\delta + M[i-1, j], \delta + M[i, j-1]\}.$
Return $M[m, n]$.

n					
a					
e					
m					
-					
	-	n	a	m	e

Sequence Alignment: An Example

Ex. Align the words *mean* and *name*. Assume that $\delta = 2$; matching a vowel with a different vowel, or a consonant with a different consonant, costs 1; while matching a vowel, or a consonant with each other costs 3.



$$\boxed{\begin{aligned} M[i,j] \leftarrow \min\{\alpha[x_i, y_j] + M[i-1, j-1],\\ \delta + M[i-1, j], \delta + M[i, j-1]\}\end{aligned}}$$

By following arrows backward from node (4,4), we can trace back to construct the alignment.

Longest Common Substring

A slightly different problem (longest common subsequence) with a similar solution

Given two strings $X = x_1x_2...x_m$ and $Y = y_1y_2...y_n$, find their longest common substring Z , i.e., a largest k for which there are indices i and j with $x_ix_{i+1}...x_{i+k-1} = y_jy_{j+1}...y_{j+k-1}$.

For example:
X : DEADBEEF
Y : EATBEEF
Z : BEEF //pick the longest contiguous substring

Show how to do this by dynamic programming.

LCS Solution

Step 1: Space of Subproblems

For $1 \le i \le m$, and $1 \le j \le n$,

- Define $d_{i,j}$ to be the length of the longest common substring ending at x_i and y_j . (Does this work?)
- Let D be the $m \times n$ matrix $[d_{i,j}]$.
  - How does D provide answer?

LCS Solution

Step 2: Recursive Formulation

Case 1: If $x_i = y_j$, then $z_k = x_i = y_j$ and

z_{k-1} is a LCS of X and Y ending at x_{i-1} and y_{j-1}

Case 2: If $x_i \neq y_j$, then there cannot be a common substring ending at x_i and y_j!

$$d_{i,j} = \begin{cases} d_{i-1,j-1} + 1 & if\ x_i = y_j \\ 0 & if\ x_i \neq y_j \end{cases}$$

Finally, we can find length of longest common substring by finding maximum $d_{i,j}$ among all possible ending position i and j.

$$LCSSubString(X, Y) = \max\{d_{i,j}\}$$

LCS Solution

Step 3: Bottom-up Computation

Similar to *Longest Common Subsequence* we set the first row and column of the matrix $d[0, j]$ and $d[i, 0]$ to be 0.

Calculate $d[1, j]$ for $j = 1, 2, \ldots, \mathrm{n}$
Then, the $d[2, j]$ for $j = 1, 2, \ldots, \mathrm{n}$
Then, the $d[3, j]$ for $j = 1, 2, \ldots, \mathrm{n}$

etc., filling the matrix row by row and left to right.

For this problem we do not need to create another $m \times n$ matrix for storing arrows. Instead, we use l_{max} and p_{max} to store the largest length of common substring and its i position respectively. This suffices to reconstruct the solution.

LCS Solution

LONGEST-COMMON-SUBSTRING(*X*,*Y*)

$m \leftarrow length(X);\ n \leftarrow length(Y);$

$l_{max} \leftarrow 0;\ p_{max} \leftarrow 0;$

$\boldsymbol{for}$ i $\leftarrow 0$ $\boldsymbol{to}$ m // initialization

$\quad d[i,0] \leftarrow 0;$

$\boldsymbol{for}$ $j \leftarrow 0$ $\boldsymbol{to}$ n

$\quad d[0,j] \leftarrow 0;$

$\boldsymbol{for}$ i $\leftarrow 1$ $\boldsymbol{to}$ m // dynamic programming

$\quad \boldsymbol{for}$ $j \leftarrow 1$ $\boldsymbol{to}$ n

$\quad\quad \boldsymbol{if}(x_i \neq y_j)$

$\quad\quad\quad d[i,j] \leftarrow 0;$

$\quad\quad$ ***else***

$\quad\quad\quad d[i,j] \leftarrow d[i-1,j-1]+1\,;$

$\quad\quad\quad \boldsymbol{if}(d[i,j] > l_{max})$

$\quad\quad\quad\quad l_{max} \leftarrow d[i,j];\, p_{max} \leftarrow i;$

$\quad\quad\quad$ ……

$\boldsymbol{return}\ l_{max},\ p_{max};$

LCS Example

- Take the two strings: X = “EL GATO” and Y =“GATER”.
- We’ll fill in the following table D:

$$d_{i,j} = \begin{cases} d_{i-1,j-1} + 1 & if\ x_i = y_j \\ 0 & if\ x_i \neq y_j \end{cases}$$

LCS Example

- Take the two strings: *X* = "EL GATO" and *Y* ="GATER".
- We'll fill in the following table *D*:

$$d_{i,j} = \begin{cases} d_{i-1,j-1} + 1 & if\ x_i = y_j \\ 0 & if\ x_i \neq y_j \end{cases}$$

When filling *D*, we only look if the two letters in the strings are equal and if they are we add one to the element to the left and up.

	_	E	L	G	A	T	O
_	0	0	0	0	0	0	0
G	0	0	0	1	0	0	0
A	0	0	0	0	2	0	0
T	0	0	0	0	0	3	0
E	0	1	0	0	0	0	0
R	0	0	0	0	0	0	0

Review of Matrix Multiplication

- The product $C = AB$ of a $p \times q$ matrix A and a $q \times r$ matrix B is a $p \times r$ matrix C given by.

$$c[i,j] = \sum_{k=1}^{q} a[i,k]b[k,j], \quad \text{for } 1 \leq i \leq p \text{ and } 1 \leq j \leq r$$

- Complexity of Matrix multiplication: Note that C has pr entries and each entry takes $\Theta(q)$ time to compute so the total procedure takes $\Theta(pqr)$ time.

Matrix Multiplication of ABC

- Given $p \times q$ matrix A, $q \times r$ matrix B and $r \times s$ matrix C, ABC can be computed in two ways: $(AB)C$ and $A(BC)$.
- The number of multiplications needed are:

$$\begin{aligned} mult[(AB)C] &= pqr + prs, \\ mult[A(BC)] &= qrs + pqs. \end{aligned}$$

Implication: Multiplication "sequence" (parenthesization) is important!!

Developing a Dynamic Programming Algorithm

Step 1: Define Space of Subproblems

- Original Problem:

  Determine minimal cost multiplication sequence for $A_{1..n}$.

- Subproblems: For every pair *1 ≤ i ≤ j ≤ n*:

  Determine minimal cost multiplication sequence for $A_{i..j} = A_i A_{i+1} \ldots A_j$.

  Note that $A_{i..j}$ is a $p_{i-1} \times p_j$ matrix.

- There are $\binom{n}{2} = \theta(n^2)$ such subproblems. (Why?)
- How can we solve larger problems using subproblem solutions?

Relationships among Subproblems

- At the last step of any optimal multiplication sequence (for a subbroblem), there is some k such that the two matrices $A_{i..k}$and $A_{k+1..j}$ are multipled together. That is,

$$A_{i..j} = (A_i \cdots A_k)(A_{k+1} \cdots A_j) = A_{i..k}A_{k+1..j}$$

- Question. How do we decide where to split the chain (what is *k*)?

  ANS: Can be any *k*. Need to check all possible values.

- Question. How do we parenthesize the two subchains $A_{i..k}$ and $A_{k+1..j}$?
- For some problems, **the subtrees will not overlap**.

  ANS: $A_{i..k}$ and $A_{k+1..j}$ must be computed optimally, so we can apply the same procedure recursively.

Relationships among Subproblems

Step 2: Constructing optimal solutions from optimal subproblem solution

- For $1 \le i \le j \le n$, let $m[i,j]$ denote the minimum number of multiplications needed to compute $A_{i..j}$. This optimum cost must satisify the following recursive definition.

$$m[i,j] = \begin{cases} 0, & i = j, \\ min_{i \le k < j}(m[i,k] + m[k+1,j] + p_{i-1}p_k p_j) & i < j \end{cases}$$

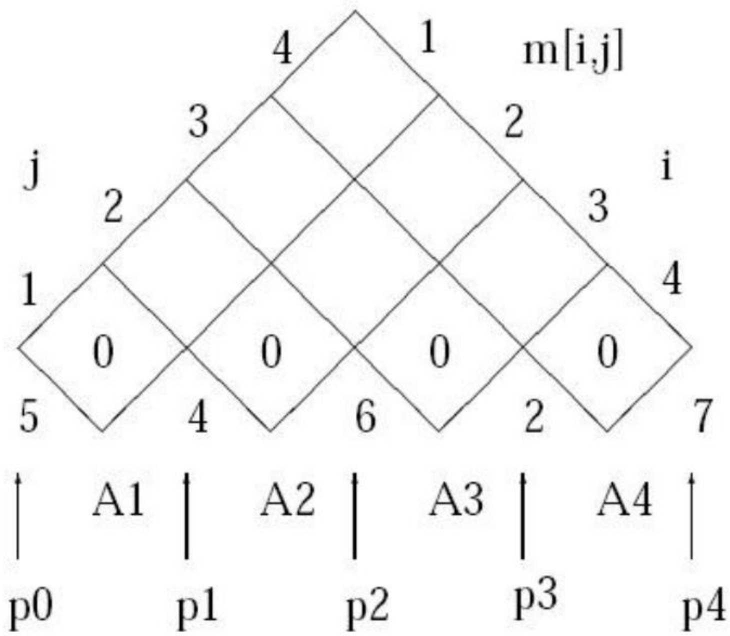
$$A_{i..j} = A_{i..k}A_{k+1..j}$$

Example for the Bottom-Up Computation

- Example.

A chain of four matrices A_1, A_2, A_3 and A_4, with $p_0 = 5$, $p_1 = 4$, $p_2 = 6$, $p_3 = 2$ and $p_4 = 7$. Find $m[1,4]$.

S0: Initialization

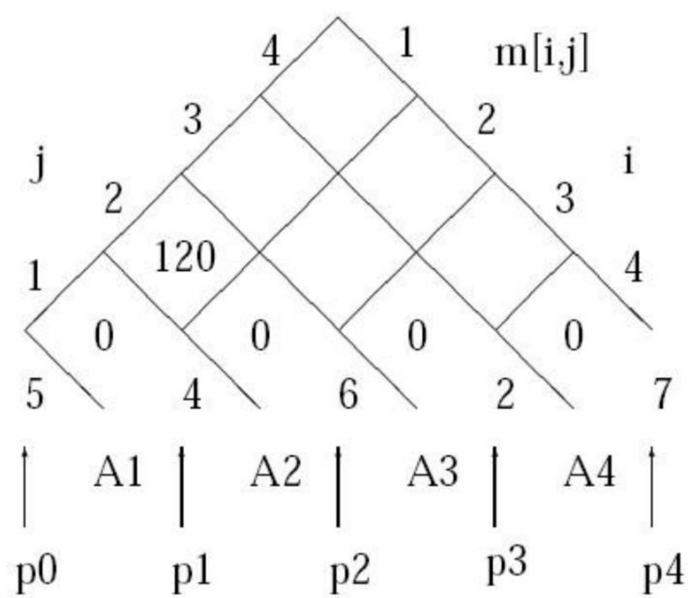


Example – Continued

- Step 1: Computing $m[1,2]$

By definition

$$
\begin{aligned}
m[1,2] &= \min_{1\le k<2}(m[1,k]+m[k+1,2]+p_0p_kp_2 \\
&= m[1,1]+m[2,2]+p_0p_1p_2 = 120
\end{aligned}
$$

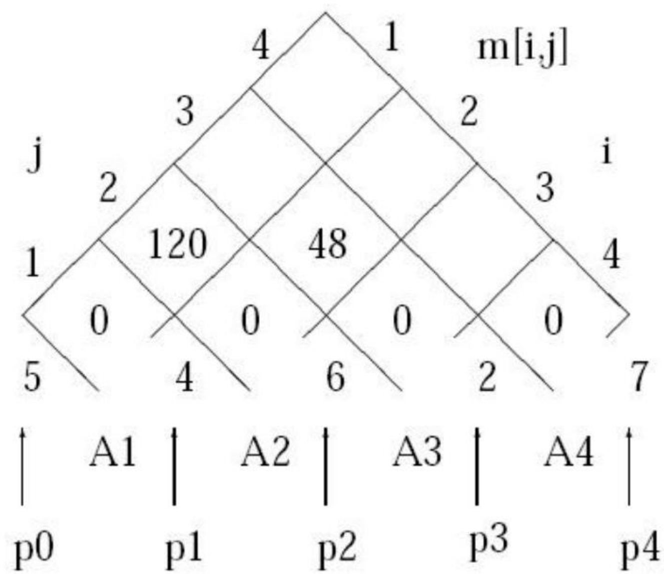


Example – Continued

- Step 2: Computing $m[2,3]$

By definition

$$
\begin{aligned}
m[2,3] &= \min_{2\le k<3}(m[2,k] + m[k+1,3] + p_1 p_k p_3 \\
&= m[2,2] + m[3,3] + p_1 p_2 p_3 = 48
\end{aligned}
$$

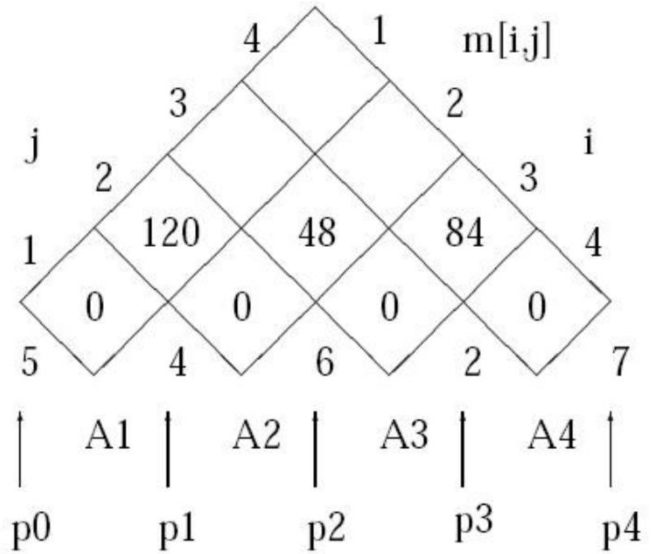


Example – Continued

- Step 3: Computing $m[3, 4]$

By definition

$$m[3,4] = \min_{3 \le k < 4} (m[3,k] + m[k+1,4] + p_2 p_k p_4$$
$$= m[3,3] + m[4,4] + p_2 p_3 p_4 = 84$$

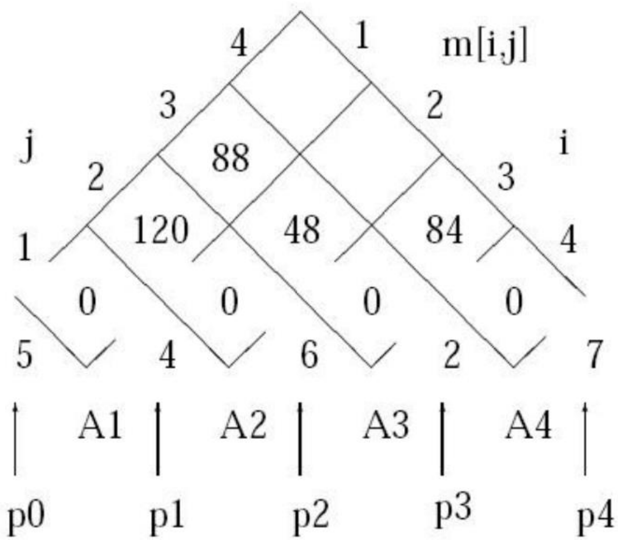


Example – Continued

- Step 4: Computing $m[1,3]$

By definition

$$
\begin{aligned}
m[1,3] &= \min_{1\le k<3}(m[1,k]+m[k+1,3]+p_0p_kp_3 \\
&= \min\begin{Bmatrix} m[1,1]+m[2,3]+p_0p_1p_3 \\ m[1,2]+m[3,3]+p_0p_2p_3 \end{Bmatrix} \\
&= 88
\end{aligned}
$$

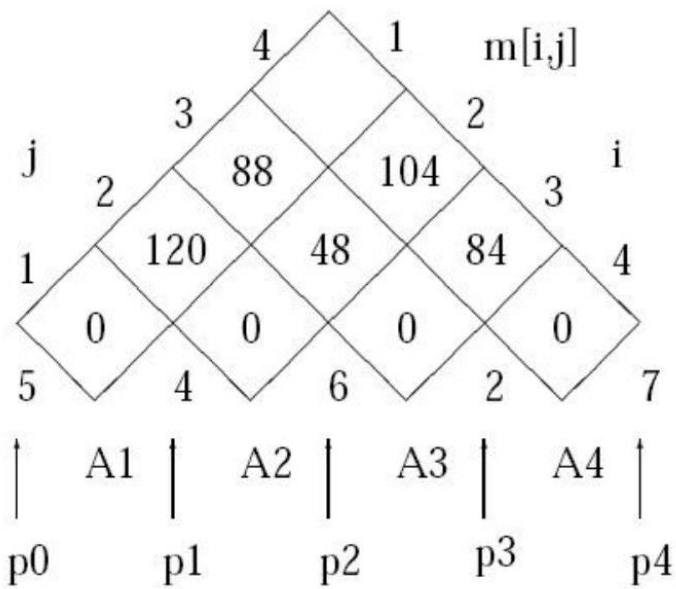


Example – Continued

- Step 5: Computing $m[2,4]$

By definition

$$
\begin{aligned}
m[2,4] &= \min_{2\le k<4} (m[2,k] + m[k+1,4] + p_1 p_k p_4 \\
&= \min \begin{Bmatrix} m[2,2] + m[3,4] + p_1 p_2 p_4 \\ m[2,3] + m[4,4] + p_1 p_3 p_4 \end{Bmatrix} \\
&= 104
\end{aligned}
$$

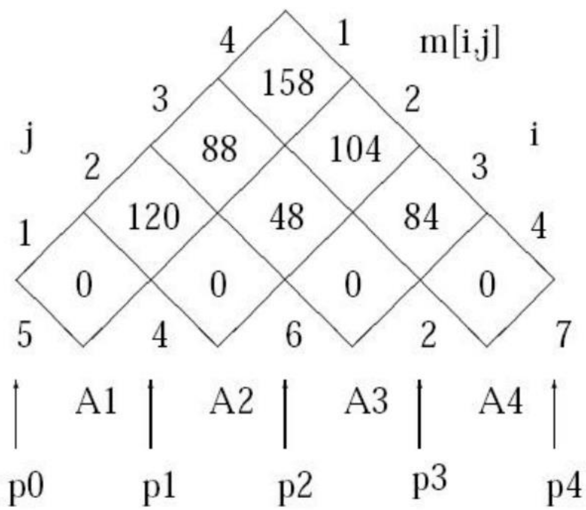


Example – Continued

- Step 6: Computing $m[1,4]$

By definition

$$
\begin{aligned}
m[1,4] &= \min_{1\le k<4}\left(m[1,k]+m[k+1,4]+p_0p_kp_4\right)\\
&= \min\begin{Bmatrix} m[1,1]+m[2,4]+p_0p_1p_4 \\ m[1,2]+m[3,4]+p_0p_2p_4 \\ m[1,3]+m[4,4]+p_0p_3p_4 \end{Bmatrix}\\
&= 158
\end{aligned}
$$



The Dynamic Programming Algorithm

Matrix-Chain(p, n): // *l* is length of sub-chain

for $i = 1$ **to** n **do** $m[i, i] = 0$;

;

for $l = 2$ **to** n **do**

  for $i = 1$ **to** $n - l + 1$ **do**

    $j = i + l - 1$;

    $m[i, j] = \infty$;

    for $k = i$ **to** $j - 1$ **do**

      $q = m[i, k] + m[k + 1, j] + p[i - 1] * p[k] * p[j]$;

      if $q < m[i, j]$ **then**

        $m[i, j] = q$;

        $s[i, j] = k$;

      end

    end

  end

end

return m and s; (Optimum in $m[1, n]$)

Revisit:

Backtrack

Backtrack Paradigm

- Recurve approach is essentially travelling the whole tree defined by the recursive relation.

  The subtrees may repeat, so we need to cache intermediate results to improve efficiency. This is exactly the essence of **dynamic programming**.

- For some problems, **the subtrees will not overlap**.

  In such case, there is no better algorithm other than traveling the entire tree. But, we can travel the entire tree smartly.

  This is what **backtrack** technique concerns: *stop visiting the subtree if the solution won't appear and backtrack to the parent node.*

  - Basic backtrack strategy: **Domino property defined by problem constraint.**

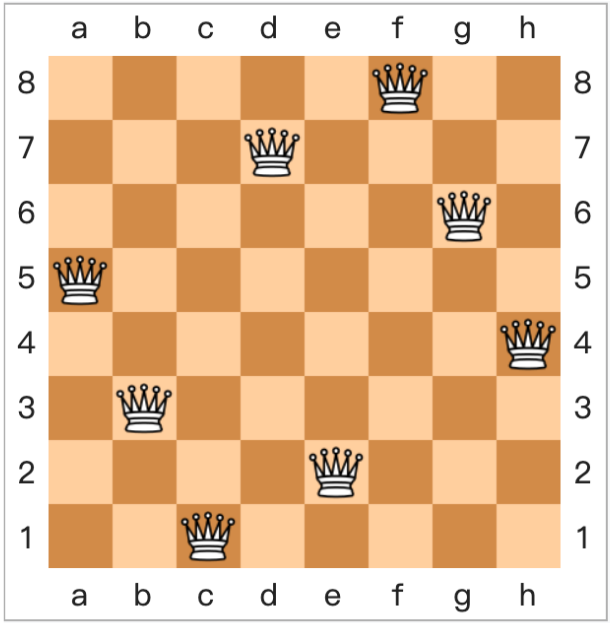
  - Advanced backtrack strategy: ***Branch-and-bound.***

Example: 8-Queen Problems

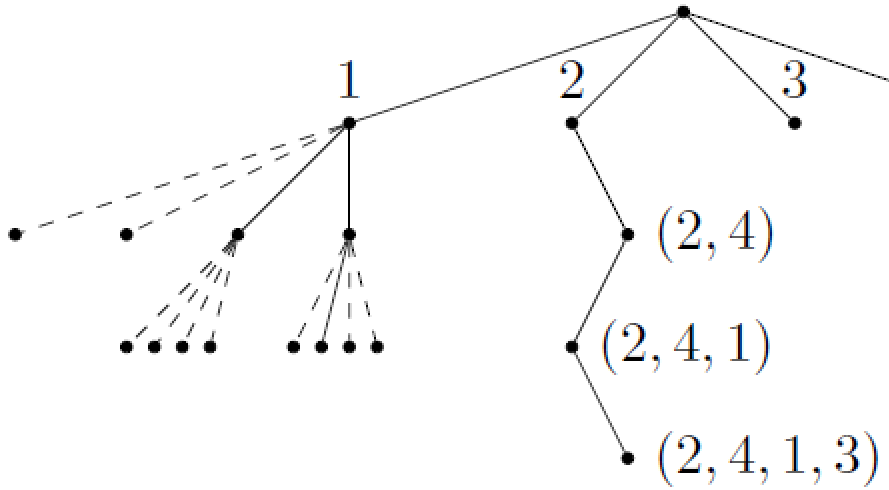
- 8-queen puzzle. Placing eight chess queens on an 8×8 chessboard so that no two queens threaten each other.

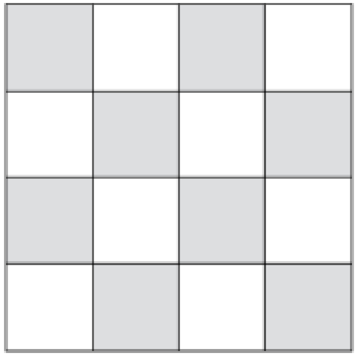
A solution requires that no two queens share the same row, column, or diagonal.

8-queen puzzle is a special case of the more general n-queen problem: placing n non-attacking queens on an $n \times n$ chessboard.



Demo of Quadtree for 4-Queen Puzzle





- Travel the tree via depth-first order to find all solutions.

  i-th level node represent i-th element in solution vector in the i-th level, the branching choice is less than $n - i$ leaves correspond to solutions.

Example: 0-1 Knapsack Problem

- Problem. Given n items with value v_i and weight w_i, as well as a knapsack with weight capacity W. The number of each item is 1. Find a solution that maximizes the overall value.
- Solution. n dimension vector $(x_1, x_2, \ldots, x_n) \in \{0,1\}^n$, $x_i = 1 \Leftrightarrow$ selecting item i.
- Nodes: $(x_1, x_2, \ldots, x_k)$ corresponds to partial solution.
- Search space. In all levels, the branching choice is always 2 (perfect binary tree with 2^n leaves).
- Candidate solution. Satisfy constraint $\sum_{i=1}^{n} w_i x_i \leq W$.
- Optimal solution. The candidate solutions that achieve maximal values.

Demo

- Ex.

Table: $n = 4$, $W = 13$

item	1	2	3	4
value	12	11	9	8
weight	8	6	4	3

Solution. n dimension vector $(x_1, x_2, \dots, x_n) \in \{0,1\}^n$, $x_i = 1 \Leftrightarrow$ selecting item i.
Nodes: $(x_1, x_2, \dots, x_k)$ corresponds to partial solution.
Search space. In all levels, the branching choice is always 2 (perfect binary tree with 2^n leaves).

Demo

- Ex.

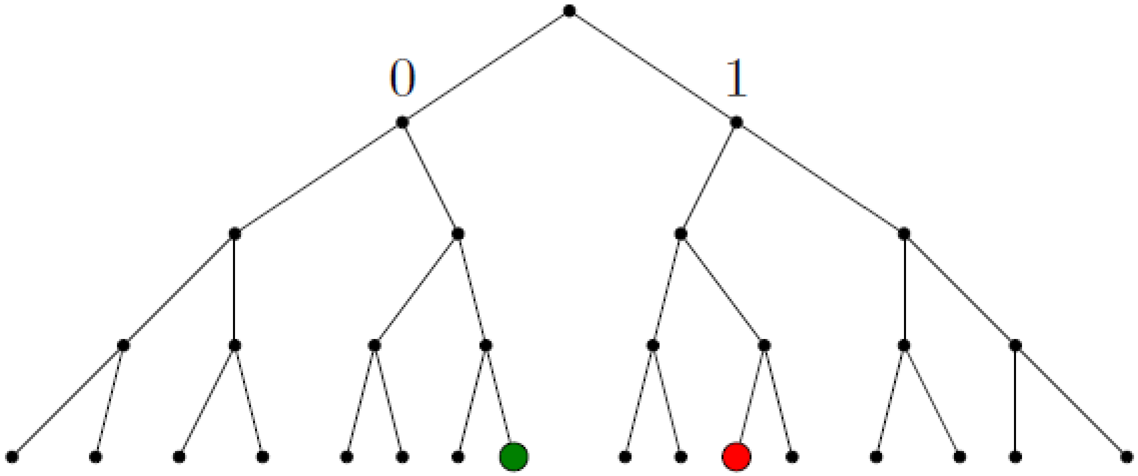
Table: $n = 4$, $W = 13$

item	1	2	3	4
value	12	11	9	8
weight	8	6	4	3

Two candidate solutions

1. $(0, 1, 1, 1)$: $v = 28$, $w = 13$
2. $(1, 0, 1, 0)$: $v = 21$, $w = 12$

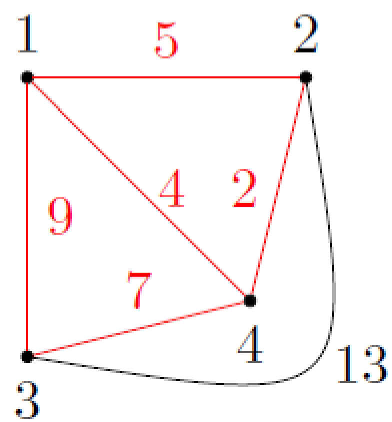
Optimal solution is $(0, 1, 1, 1)$



Example: Traversal Salesman Problem

- Problem. Given n cities $C = \{c_1, c_2, \dots, c_n\}$ and $d(c_i, c_j) \in Z^+$. Find a cycle with minimal length that travels each city once.
- Solution. A permutation of $(1,2,\dots,n) \Rightarrow (k_1, k_2, \dots, k_n)$ such that

$$\min\left\{\sum_{i=1}^{n-1} d\left(c_{k_i}, c_{k_{i+1}}\right) + d\left(c_{k_n}, c_{k_1}\right)\right\}$$



$C = \{1,2,3,4\}$

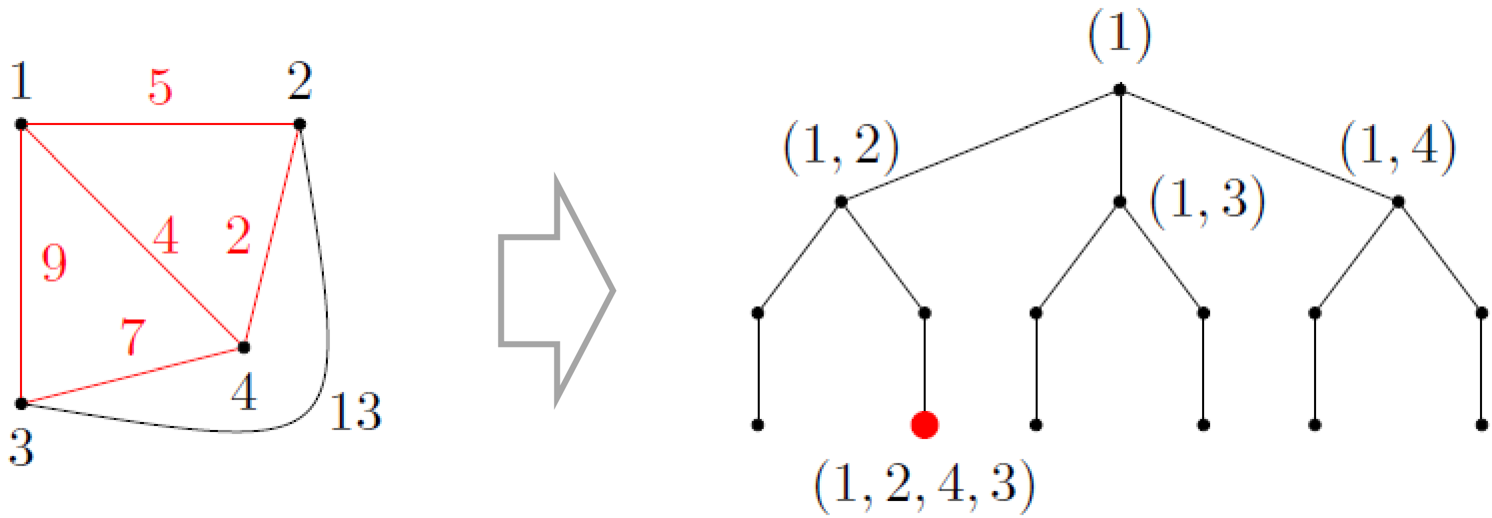
$d(1,2) = 5,\ d(1,3) = 9$

$d(1,4) = 4,\ d(2,3) = 13$

$d(2,4) = 2,\ d(3,4) = 7$

Solution is (1,2,4,3), length of cycle is 5+2+7+9=23.

Search Space of TSP



- Any node can serve as the root, cause TSP is defined over an undirected graph.
- Search space. In the i-th level, the branching choice is always $n - i \Longrightarrow$ obtain a tree with $(n-1)!$ leaves (number of all possible permutations over $\{1,2,\ldots,n\}$.

Classical Examples of Backtrack

- Problem: n-Queen Puzzle, 0-1 Knapsack, TSP
- Solution: Vector
- Search space: Tree

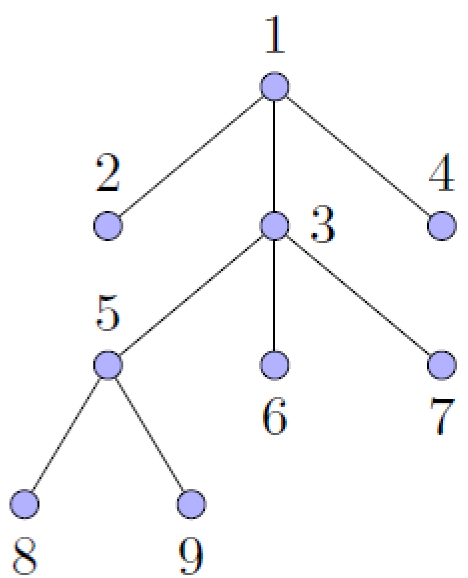
  Nodes correspond to partial solutions, leaves correspond to candidate solutions.
- Search order: Depth-first, Breadth-first,…

Main Idea of Backtrack

- Scope of application. Search or optimization problem
- Search space. Tree

  Leaves: candidate solution

  Nodes: partial solution
- How to search. Systematically traversal the tree: DFS, BFS, …



DFS: $1 \to 2 \to 3 \to 5 \to 8 \to 9 \to 6 \to 7 \to 4$

BFS: $1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9$

States of Nodes

- The tree is explored dynamically. Let v be the candidate node (corresponding to partial solution) and P be the predicate that checks if v satisfies the constraint.

  $P(v) = 1 \Rightarrow$ expand

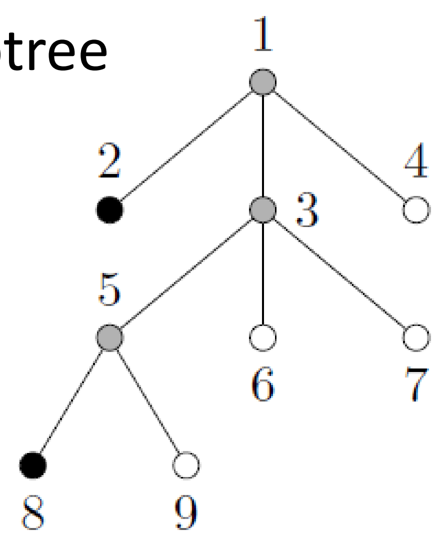
  $P(v) = 0 \Rightarrow$ backtrack to the parent node

- States of node

  Black: finishing the traversal of this subtree

  Gray: visiting its subtree

  White: unexplored



DFS: $1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 8$

finished visiting: $2, 8$

being visited: $1, 3, 5$

unexplored: $9, 6, 7, 4$

Basic Backtrack Technique: Domino Property

- At note $v = (x_1, \dots, x_k)$,

  $P(x_1, \dots, x_k) = 1 \Rightarrow (x_1, \dots, x_k)$ meet some property.

Example. n-queen puzzle, placing k queens in positions without attacking each other.

Domino property (admit safe backtrack)

$$P(x_1, \dots, x_{k+1}) = 1 \Rightarrow P(x_1, \dots, x_k) = 1, 0 < k < n$$

Converse-negative proposition

$$P(x_1, \dots, x_k) = 0 \Rightarrow P(x_1, \dots, x_{k+1}) = 0, 0 < k < n$$

k-dimension vector does not satisfy constrain $\Rightarrow$ its k+1-dimension extension does not satisfy constraint either

This property guarantees that backtracking will not miss any solution. Safely backtrack when $P(x_1, \dots, x_k) = 0$

Domino Property

- The premise condition to use backtrack: Domino Property.

- General steps of backtrack algorithm:

  Define solution vector (include the range of every element),
  $$(x_1, \ldots, x_n) \in X_1 \times \cdots \times X_n$$

  After fixing $(x_1, \ldots, x_{k-1})$, update admissible range of x_k as $A_k \subseteq X_k$ using predicate P.

  Decide if Domino property is satisfied.

  Decide the search strategy: DFS, BFS.

  Decide the data structure to store the search path.

Backtrack Recursive Template

Algorithm 1: BackTrack(n) // output all solutions

for $k = 1\ to\ n$ do $A_k \leftarrow X_k$; // initialize
ReBack(1);

Algorithm 2: ReBack(k) // k is the current depth of recursion

if $k = n$ then return solution $(x_1, \ldots, x_n)$;
else
    while $A_k \neq \emptyset$ do
        $x_k \leftarrow A_k$ // according to some order;
        $A_k \leftarrow A_k - \{x_k\}$;
        update A_{k+1}, ReBack(k+1)
    end
end

The above is the oversimplified pseudocode. One must be careful when dealing with domains A_k and solution vector x when coding.

Loading Problem

- Problem. Given n containers with weight w_i, two boats with weight capacity W_1 and W_2 s.t. $w_1 + \cdots + w_n \leq W_1 + W_2$.
- Goal. If there exists a scheme to load the n containers on two boats. Please give a scheme if it is solvable.

Ex.

$$w_1 = 90, w_2 = 80, w_3 = 40, w_4 = 30, w_5 = 20, w_6 = 12,$$
$$w_7 = 10, W_1 = 152, W_2 = 130$$

Solution: load 1, 3, 6, 7 on boat 1 and the rest on boat 2.

Main idea: Let the total weights be W.

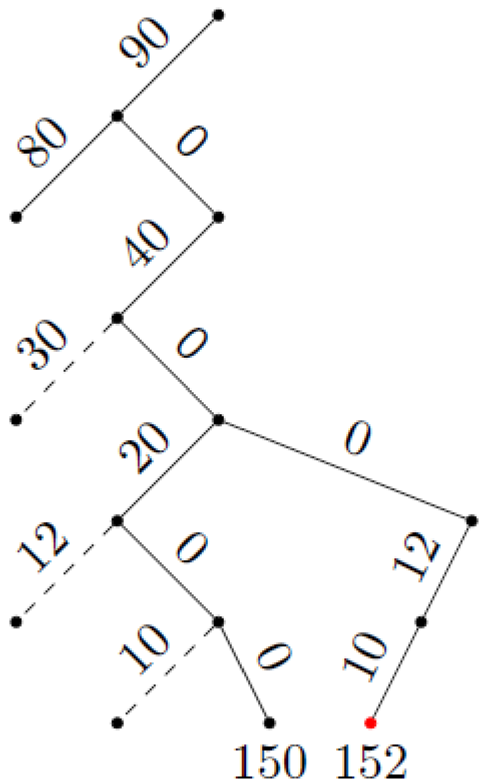
Load on boat 1 first. Using backtrack to find a solution that maximizes W_1^*, where W_1^* is the real capacity.

Then check if $W - W_1^* \leq W_2$. Return "yes" if true and "no" otherwise.

Demo

Ex.

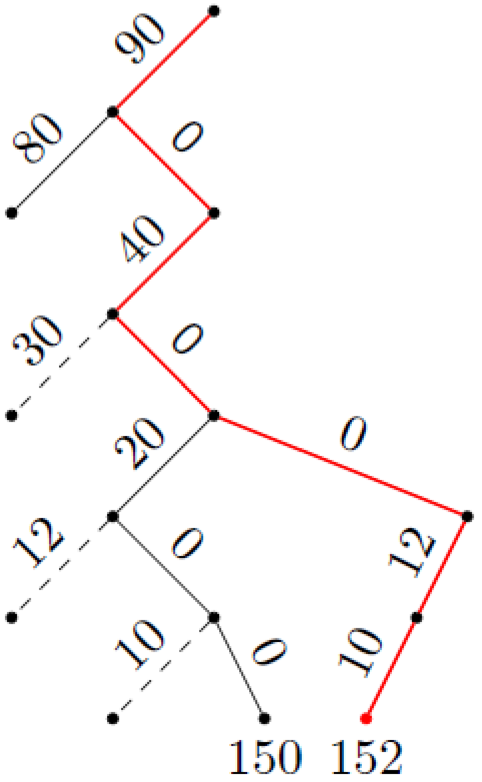
$$w_1 = 90, w_2 = 80, w_3 = 40, w_4 = 30, w_5 = 20, w_6 = 12,$$
$$w_7 = 10, W_1 = 152, W_2 = 130$$



Demo

Ex.

$$w_1 = 90, w_2 = 80, w_3 = 40, w_4 = 30, w_5 = 20, w_6 = 12,$$
$$w_7 = 10, W_1 = 152, W_2 = 130$$



it is loadable

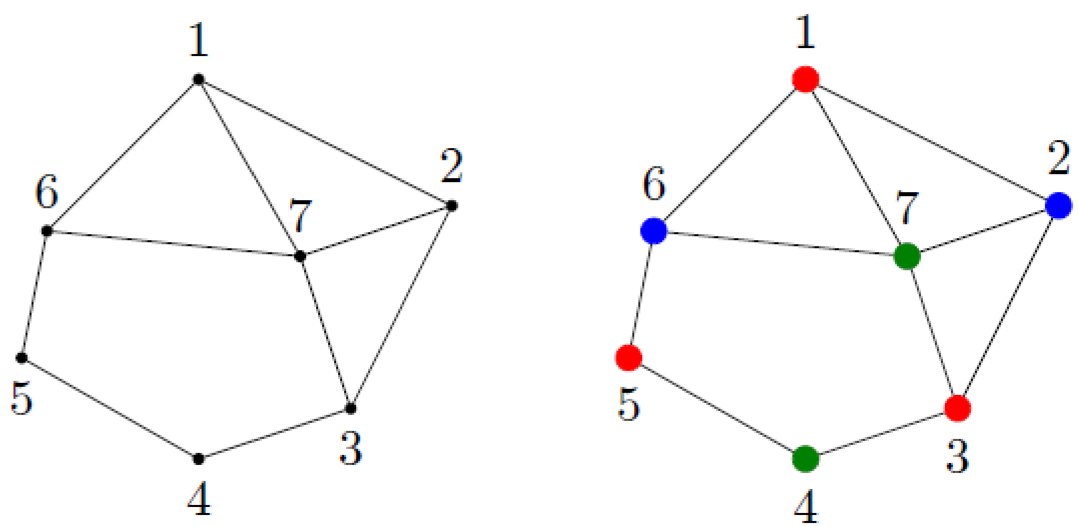
$1, 3, 6, 7$ on boat 1

$2, 4, 5$ on boat 2

time complexity $W(n) = O(2^n)$

Graph Coloring Problem

- Problem. Undirected graph G and m colors. Coloring the vertices to ensure the connected two vertices with different color.
- Goal. Output all possible coloring schemes. Output “no” if there is none.



$n = 7, m = 3$

Algorithm Design

- Input. $G = (V, E), V = \{1,2, \dots, n\}$, color set $= \{1,2, \dots, m\}$
- Solution vector. $(x_1, x_2, \dots, x_n), x_i \in [m]$

  $(x_1, x_2, \dots, x_k)$ gives partial solution for vertex set $\{1,2, \dots, k\}$

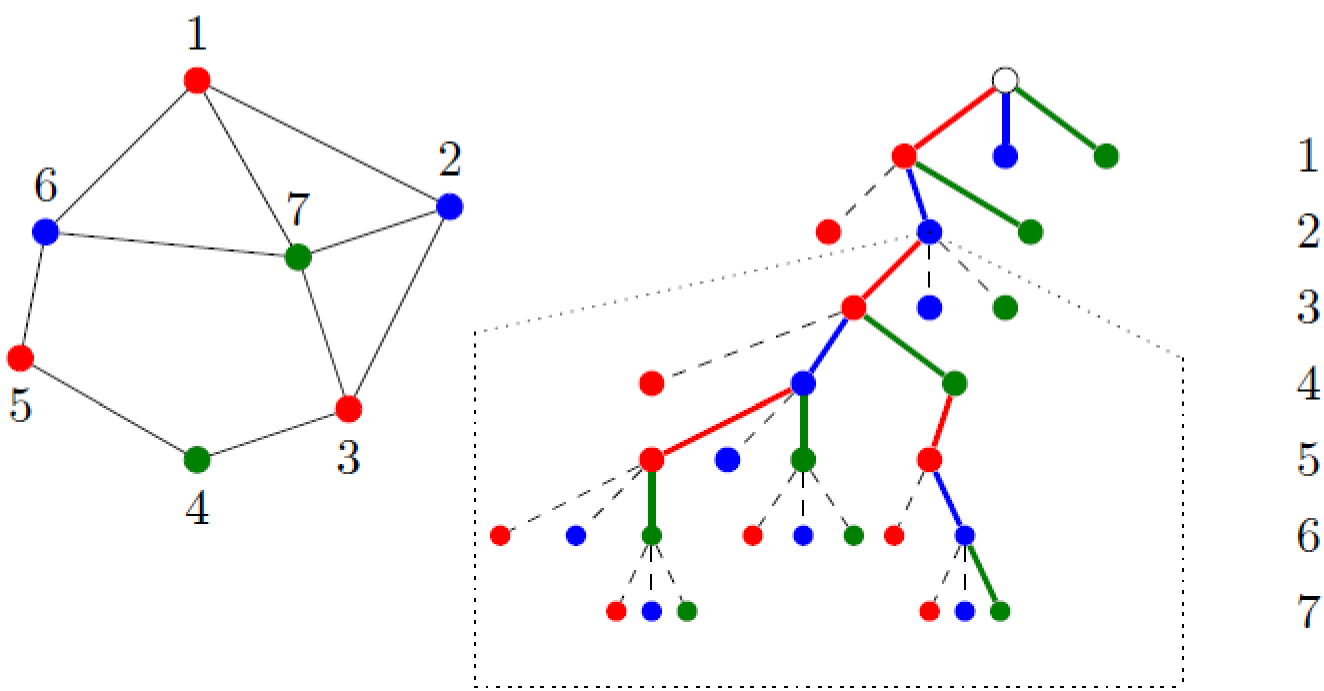
Search tree. m-fork tree

Constraint. At node $(x_1, x_2, \dots, x_k)$, the set of available colors for node $k + 1$ is not empty.

  If the nodes in adjacent list have used up m colors, then node $k + 1$ is not colorable. In this case, back to parent node.

Search strategy: DFS

Demo



The first solution vector: $(1, 2, 1, 3, 1, 2, 3)$

Applications of Graph Coloring

- Arrangement of meeting room

  There are n events to be arranged, if the slots of event i and event j overlap, we say i and j are not compatible. How to arrange these events with smallest number of meeting rooms?

- Modeling

  Treat event as node, if i, j are not compatible, then add an edge between i and j.

  Treat meeting rooms as colors.

The arrangement problem is transformed into finding a coloring scheme with smallest colors.