Algorithms & Complexity

Università della Svizzera italiana

Fabian Bosshard

February 3, 2026

This page is intentionally left blank.

Contents

Preface

This document is an unofficial student-made summary of the course Algorithms & Complexity taught by Evanthia Papadopoulou in Winter 2025/2026 at the Università della Svizzera italiana. It is primarily based on the lecture slides, supplemented by [1, 2, 3]. Selected material has been adapted from [4, 5, 6, 7, 8, 9]. If you find any errors, please report them to fabianlucasbosshard@gmail.com. The LaTeX source code is available on `https://github.com/fabianbosshard/usi-informatics-course-summaries`.

References

[1] David M. Mount. CMSC 451: Design and Analysis of Computer Algorithms – (Fall 2015). 2015. URL: `https://www.cs.umd.edu/class/fall2015/cmsc451-0101/Lects/cmsc451-fall15-lects.pdf`.

[2] David M. Mount. CMSC 451: Design and Analysis of Computer Algorithms – (Spring 2025). 2025. URL: `https://www.cs.umd.edu/class/spring2025/cmsc451-0101/Lects/cmsc451-spring25-lects.pdf`.

[3] Jon Kleinberg and Éva Tardos. Algorithm Design. Pearson, 2006. URL: `https://dl.acm.org/doi/10.5555/2031509`.

[4] Bhawani S. Panda. Cut Vertices, Cut Edges and Biconnected Components (MTL776). URL: `https://web.iitd.ac.in/~bspanda/biconnectedMTL776.pdf`.

[5] Francis Chin and David Houck. Algorithms for Updating Spanning Trees. 1978. URL: `https://www.sciencedirect.com/science/article/pii/0022000078900223`.

[6] Uri Zwick. The Smallest Networks on Which the Ford–Fulkerson Maximum Flow Procedure May Fail to Terminate. 1995. URL: `https://www.sciencedirect.com/science/article/pii/030439759500022O`.

[7] Xiao Liang. Set Cover and Hitting Set. 2025. URL: `https://xiao-liang.github.io/Resources/Courses/CSCI3160-25Fall/slides/lectures/0s_sc.pdf`.

[8] Deeparnab Chakrabarty. Greedy Algorithm for Set Cover. URL: `https://www.cs.dartmouth.edu/~deepc/LecNotes/Appx/1.%20Greedy%20Algorithm%20for%20Set%20Cover.pdf`.

[9] Jeff Erickson. Max-Flow Algorithms and the Golden Ratio. URL: `https://courses.grainger.illinois.edu/cs473/fa2013/notes/22-maxflowalgs.pdf`.

1 Introduction and Basics

1.1 Introduction

typically,

- numerical data → numerical analysis
- discrete data → algorithm design and analysis

two fundamental issues to consider: *correctness*, *efficiency*

1.2 Summations

Arithmetic series:

$$\sum_{i=0}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} = \Theta(n^2)$$

Geometric series:

$$\sum_{i=0}^{n} c^i = 1 + c + c^2 + \cdots + c^n = \frac{c^{(n+1)} - 1}{c - 1}$$

- if $0 < c < 1 : \Theta(1)$
- if $c > 1 : \Theta(c^n)$ (entire sum proportional to the last element)

Quadratic series:

$$\sum_{i=0}^{n} i^2 = 1 + 4 + \cdots + n^2 = \frac{2n^3 + 3n^2 + n}{6} = \Theta(n^3)$$

Linear-Geometric series:

$$\sum_{i=0}^{n-1} ic^i = c + 2c^2 + \cdots + nc^n = \frac{(n-1)c^{(n+1)} - nc^n + c}{(c-1)^2} = \Theta(nc^n)$$

Harmonic series:

$$H_n = \sum_{i=0}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \cdots + \frac{1}{n} = \ln n + O(1) = \Theta(\log n)$$

Summation with general bounds:

$$\sum_{i=a}^{b} f(i) = \sum_{i=0}^{b} f(i) - \sum_{i=0}^{(a-1)} f(i)$$

Approximation using integrals:

$$\int_0^n f(x)\mathrm{d}x \leq \sum_{i=1}^{n} f(i) \leq \int_1^{n+1} f(x)\mathrm{d}x$$

(if $f(x)$ is monotonically increasing)

1.3 Recurrence Relations

Theorem 1.1 (Simplified Master Theorem)**.** Let $a \geq 1, b > 1$ be constants and let $T(n)$ be the recurrence

$$T(n) = aT(n/b) + cn^k \tag{1.1}$$

defined for $n \geq 0$.

- Case 1: $a > b^k$ then $T(n)$ is $\Theta(n^{\log_b a})$
- Case 2: $a = b^k$ then $T(n)$ is $\Theta(n^k \log n)$
- Case 3: $a < b^k$ then $T(n)$ is $\Theta(n^k)$ ◁

1.4 Other

Stirling's approximation:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

2 Stable Matching

motivation: set up pairings between different entities, where each side has a notion of preference

Definition 2.1 (Matching)**.** Given a pair of sets M and W, a matching S is a collection of pairs (m, w), where $m \in M$ and $w \in W$, and each element from either set appears in at most one pair. A matching is perfect if no element remains unmatched. ✍

Definition 2.2 (Stability)**.** Given sets M and W of equal size and a preference ordering for each element of each set, a stable matching is a perfect matching where no unstable pair exists. An unstable pair is a pair (m, w) that is *not* in the matching such that m and w prefer each other to their current partners in the matching. ✍

given two sets M and W of equal size n, where every $m \in M$ and every $w \in W$ has a strict and complete preference list over the elements of the other set

Algorithm 2.1 Propose-and-Reject (Gale-Shapley, 1962)

Require: $2n$ strict and complete preference lists, each consisting of n elements
Ensure: a matching S^* that pairs each $m \in M$ with each $w \in W$

```
initialize all m ∈ M and w ∈ W as free
while there exists a free m ∈ M who has not proposed to every w ∈ W do
    choose such an m
    w ← highest ranked element on m's list to whom m has not yet proposed
    m proposes to w
    if w is free then
        match m and w
    else if w prefers m to her current partner m′ then
        match m and w
        mark m′ as free
    else
        w rejects m's proposal
return S*
```

Observation 2.1. M propose to W in decreasing order of preference. ◀

Observation 2.2. Once a $w \in W$ is matched, it never becomes unmatched, it only trades up. ◀

2.1 Correctness

Claim 2.1 (Correctness: Termination)**.** Algorithm 2.1 terminates after at most n^2 iterations of the while-loop. ◁

Proof. There is one proposal per iteration. By Observation 2.1, once a $m \in M$ has proposed to a $w \in W$, it never proposes to w again. So each $m \in M$ does $\leq n$ proposals. There are n elements in M, and each does $\leq n$ proposals. After $\leq n^2$ iterations, no one is left to propose to. Thus, Algorithm 2.1 must terminate after $\leq n^2$ iterations. □

Claim 2.2 (Correctness: Perfect Matching)**.** Every $m \in M$ and every $w \in W$ gets matched by Algorithm 2.1. ◁

Proof (Contradiction)**.** Assume there is a $m \in M$ who is unmatched. Since $|M| = |W|$, there must also be a $w \in W$ who is unmatched. By Observation 2.2, w was never proposed to. But for m to remain unmatched after termination, m must have proposed to every $w \in W$, including this w. □

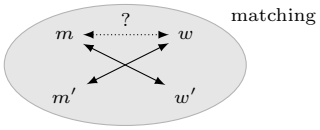
Claim 2.3 (Correctness: Stability)**.** The matching created by Algorithm 2.1 is stable. ◁

Proof (Contradiction)**.** Assume there is an unstable pair (m, w), where $m \in M$ and $w \in W$ after termination of Algorithm 2.1.

We can distinguish two cases:

Case 1: m never proposed to w

Case 2: m proposed to w



In Case 1, by Observation 2.1, m prefers his final partner w' to w, contradicting the assumption that (m, w) is an unstable pair.

In Case 2, w either rejected m or traded up from m to her final partner m'. So, by Observation 2.2, w prefers her final partner m' to m, contradicting the assumption that (m, w) is an unstable pair. □

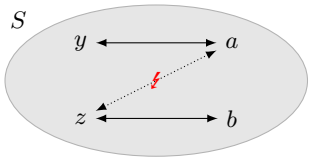
2.2 Male Optimality

For a given problem instance, there may be multiple stable matchings.

Definition 2.3 (Valid Partner)**.** An element $m \in M$ is a valid partner of an element $w \in W$ if there exists a stable matching in which they are paired. ✍

Theorem 2.4 (M-optimality)**.** The matching created by Algorithm 2.1 is M-optimal, i.e., each $m \in M$ is matched with their best valid partner. The order that M propose does not matter. ◁

Proof (Contradiction)**.** Assume there are men that are matched with someone other than their best valid partner. By Observation 2.1, those men must have been rejected or dumped by valid partners during Algorithm 2.1. Let y be the first such man and let a be the first valid woman who rejects/dumps him. When y is dumped/rejected by a in Algorithm 2.1, a forms/reaffirms engagement with a man, say z. So a prefers z to y.



Let S be a stable matching in which y is matched with a (it exists, since a is a valid partner of y). Let b be the partner of z in S. z has not been rejected/dumped by any valid partner (including b) at the point when y is rejected/dumped by a, because the latter is the first such event during Algorithm 2.1. That means z has not yet proposed to b when he proposes to a, so z prefers a to b. Thus (z, a) is an unstable pair in S, contradicting the stability of S. □

It can be proven that the M-optimal matching produced by Algorithm 2.1 is unique and W-pessimal, i.e., each $w \in W$ is matched with their worst valid partner.

2.3 Implementation

We want to implement Algorithm 2.1 in $O(n^2)$ time. This means one iteration of the while-loop must take $O(1)$ time.

To that end, we need to be able to answer the question "Does w prefer m to her current partner m'?" in $O(1)$ time! Let's say that each $w \in W$ has a preference array $\texttt{pref}_w = [m_1, \ldots, m_n]$, where m_i is the i-th most preferred man for w. To answer the above question in constant time, we build an inverse preference array $\texttt{inverse}_w$ in $O(n)$ time once for each $w \in W$ as a preprocessing step.

Algorithm 2.2 Build Inverse Preference Arrays

Require: for each $w \in W$, a preference array $\texttt{pref}_w$ such that $\texttt{pref}_w[i]$ gives the i-th most preferred man for w

Ensure: for each $w \in W$, an inverse preference array $\texttt{inverse}_w$ such that $\texttt{inverse}_w[m]$ gives the rank of m in w's preference list

```
for each w ∈ W do
    initialize the array inverse_w of size n
    for i ← 1, ..., n do
        inverse_w[pref_w[i]] ← i
```

2.4 Other Properties

Claim 2.5. Suppose all men use the same preference list $l : w_1 \succ \cdots \succ w_n$. If the women know about l, they can force Algorithm 2.1 to return any desired pairing P. ◁

Proof (Induction)**.** Consider the desired pairing $P = \{(w_i, m_i)\}_{i=1}^{n}$. Let every woman w_i submit a fake preference list that places m_i at the very top (the order of the remaining men is arbitrary).

(i) Base case: When it is m_1's time to propose, he will first propose to w_1, since all men share the list l. Because m_1 is ranked first by w_1, she accepts and will never dump him (Observation 2.2). So (w_1, m_1) is permanently formed. ✔

(ii) Induction hypothesis: Assume for some $k \geq 2$ that (w_i, m_i) are permanently matched for all $i < k$.

(iii) Induction step: Consider m_k. Because of (ii), each w_i with $i < k$ is already permanently matched with her top choice m_i and thus rejected/dumped m_k at some point. So m_k eventually proposes to w_k, who ranks m_k first and accepts. By Observation 2.2, w_k never drops m_k, so (w_k, m_k) is permanent. ✔

By induction, all pairs (w_i, m_i) form and remain, so the algorithm returns P. □

Claim 2.6. There is at most one man $m^* \in M$ such that the matching created by Algorithm 2.1 matches m^* with his last-choice woman w^*. Moreover, at the moment m^* is matched with w^*, Algorithm 2.1 terminates. ◁

Proof. Let m^* be the first man to propose to his last-choice woman w^* in Algorithm 2.1. By Observation 2.1, he has proposed to every $w \in W \setminus \{w^*\}$ and been rejected/dumped and by Observation 2.2, each such w is (and stays) engaged. Hence exactly $n - 1$ women (and thus $n - 1$ men) are engaged, so m^* is the unique free man. If w^* were engaged, there would be n engaged men, contradicting that m^* is free. Therefore w^* is free and accepts m^*, yielding n engaged pairs and immediate termination. □

3 Graphs

3.1 Representations of Graphs

Operation	**Representation**	
	Adjacency List	Adjacency Matrix
space complexity	$\Theta(n+m)$ optimal	$\Theta(n^2)$ possibly very bad
checking $(u,v) \in E$	$O(\deg(u))$ bad	$\Theta(1)$ optimal
iteration through E	$\Theta(n+m)$ okay (not optimal)	$\Theta(n^2)$ possibly very bad

in adjacency list, we often keep crosslinks between the same edges to avoid traversing lists unnecessarily

Digression 3.1 (Sparse Matrix Representation)**.** We can use a multilist structure to store sparse matrices. For that, we create $2n$ linked lists, one for each row and one for each column. Each non-zero entry a_{ij} is stored in a node that contains the value a_{ij}, the row index i, the column index j, a pointer to the next non-zero entry in row i, and a pointer to the next non-zero entry in column j. With this representation, all standard matrix operations (such as multiplication, transposition) can be performed efficiently. ◀

3.2 Basic Definitions

a graph $G = (V, E)$ consists of a set V of $n = |V|$ vertices and a set E of $m = |E|$ edges. If pairs in E are ordered, the graph is *directed*, otherwise it is *undirected*. In a digraph, self-loops (v, v) are allowed.

in a graph:
number of edges: $0 \leq m \leq \binom{n}{2} = \frac{n(n-1)}{2} \in O(n^2)$
sum of degrees: $\sum_{v \in V} \deg(v) = 2m$

in a digraph:
number of edges: $0 \leq m \leq n^2$
sum of degrees: $\sum_{v \in V} \deg^+(v) = \sum_{v \in V} \deg^-(v) = m$

sparse if m is $O(n)$, else *dense*

Definition 3.1 (Path)**.** A path P in an undirected graph is a sequence of nodes $v_1, \ldots, v_k$ with the property that each consecutive pair v_i, v_{i+1} is joined by an edge in E. ✍

Definition 3.2 (Simple Path)**.** A path is simple if all nodes are distinct. ✍

Definition 3.3 (Connected Graph)**.** An undirected graph is connected if for every pair of nodes u and v, there is a path between u and v. ✍

Definition 3.4 (Cycle)**.** A cycle is a path $v_1, \ldots, v_k$ in which $v_1 = v_k$, $k > 2$, and all other nodes are distinct, i.e., all nodes are distinct except the first and last. ✍

Definition 3.5 (Tree)**.** An undirected graph is a *tree* if it is connected and does not contain a cycle. ✍

Theorem 3.1. Let G be an undirected graph of n nodes. If we pick any two of the following statements, they imply the remaining one:

- G is connected.
- G does not contain a cycle.
- G has $n - 1$ edges. ◁

An acyclic undirected graph (which need not be connected) is a collection of free trees; it is called a forest.

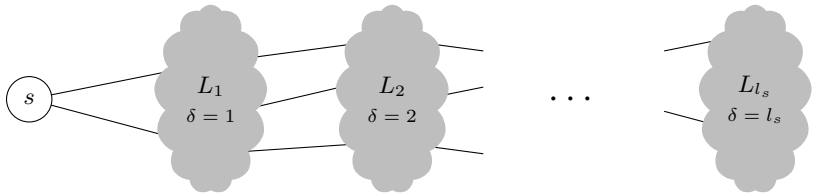
An acyclic digraph is called a directed acyclic graph; a DAG.

Definition 3.6 (Rooted tree)**.** Given a tree T, choose a root node r and orient each edge away from r. Can be used to model hierarchical structure.

- *depth* of a node: # edges from the root to the node
- *height* of a node: # edges from the node to the deepest leaf
- *height* of a tree: height of the root ✍

3.3 Breadth-First Search

intuition: explore outward from s in all possible directions, adding nodes one 'layer' at a time



BFS discovers vertices in increasing order of edge distance from s.

Distance of v: $\delta(s, v) = \min$ # edges in any path from s to v

Layer L_i consists of all nodes v at distance exactly i from s: $L_i = \{v \in V \mid \delta(s, v) = i\}$

the number of layers is $l_s = \max_{v \in V} \delta(s, v)$ and depends on s

to implement BFS, we need to know which vertices have been visited and which haven't

- keep a "frontier" of verices that have been discovered but not yet processed (a FIFO queue)
- initially all vertices (except s) are marked as undiscovered (WHITE)
- when a vertex is discovered, it is marked as discovered (GRAY) and added to the frontier
- after processing a discovered vertex, it is marked as finished (BLACK)

Algorithm 3.1 BFS

```
1:  function BFS(G, s)                                   ▷ s is the source
2:      for all u ∈ V \ {s} do
3:          u.color, u.δ, u.π ← WHITE, ∞, NIL            ▷ initialization
4:      s.color, s.δ, s.π ← GRAY, 0, NIL                 ▷ initialize source s
5:      Q ← ∅                                            ▷ initialize empty queue for vertices to visit
6:      ENQUEUE(Q, s)
7:      while Q ≠ ∅ do
8:          u ← DEQUEUE(Q)                               ▷ get next vertex from the frontier
9:          for all v ∈ Γ(u) do
10:             if v.color = WHITE then                  ▷ first time we have seen v?
11:                 v.color ← GRAY                       ▷ mark it discovered
12:                 v.δ ← u.δ + 1                        ▷ set its distance from s
13:                 v.π ← u                              ▷ set its parent
14:                 ENQUEUE(Q, v)
15:         u.color ← BLACK                              ▷ we are done with u
```

Predecessor pointers define an inverted tree (root = s)
if we reverse these edges, we get a rooted tree called BFS tree
$\exists$ many BFS trees (depends on the order of vertices placed on the queue)

tree edges: edges of BFS tree
cross edges: remaining edges

We define the BFS tree as $T_{\text{BFS}} = (V_\pi, E_\pi)$ where

$$V_\pi = \{v \in V \mid v.\pi \neq \text{NIL}\} \cup \{s\} \quad \text{and} \quad E_\pi = \{(v.\pi, v) \mid v \in V_\pi \setminus \{s\}\}$$

Property 3.2. Let T_{BFS} be a BFS tree of $G = (V, E)$. Let (x, y) be an edge of G. Then the level of x and y differ by at most 1 (same layer or one apart):

$$(x, y) \in E \implies |\delta(s, x) - \delta(s, y)| \leq 1 \qquad \triangleleft$$

Running time of BFS:

We enqueue a vertex only if is WHITE, and we immediately color it GRAY; thus, we enqueue every vertex at most once.

So the *outer loop* at Line 7 executes at most n times (once for each vertex).

For each vertex u, the *inner loop* at Line 9 executes $1 + \deg(u)$ times.

Total time:

$$\begin{aligned} T(n) &= n + \sum_{u \in V} (\deg(u) + 1) = n + \underbrace{\sum_{u \in V} \deg(u)}_{=2m} + \underbrace{\sum_{u \in V} 1}_{=n} \\ &= 2n + 2m \in O(n + m) \end{aligned}$$

Cross edges are not arbitrary. They connect nodes on the same layer or in neighboring layers.

3.4 Bipartite Graphs

Definition 3.7 (Bipartite)**.** An undirected graph $G = (V, E)$ is bipartite if the nodes can be colored red or blue such that every edge has one red and one blue end.

- a bipartite graph is 2-colorable
- a 2-colorable graph is bipartite
- $\Rightarrow$ a bipartite graph is a 2-colorable graph ✍

many graph problems become:

- easier if the underlying graph is bipartite (matching)
- tractable if the underlying graph is bipartite (independent set)

Theorem 3.3. a graph G is bipartite iff it contains no odd length cycle. $\triangleleft$

Algorithm 3.2 Find cross edges and odd cycles in an undirected graph using BFS

```
1:  function BFS(G, s)                                   ▷ s is the source
2:      for all u ∈ V \ {s} do
3:          u.color, u.δ, u.π ← WHITE, ∞, NIL            ▷ initialization
4:      s.color, s.δ, s.π ← GRAY, 0, NIL                 ▷ initialize source s
5:      Q ← ∅                                            ▷ initialize empty queue for vertices to visit
6:      ENQUEUE(Q, s)
7:      while Q ≠ ∅ do
8:          u ← DEQUEUE(Q)                               ▷ get next vertex from the frontier
9:          for all v ∈ Γ(u) do
10:             if v.color = WHITE then                  ▷ first time we have seen v?
11:                 v.color ← GRAY                       ▷ mark it discovered
12:                 v.δ ← u.δ + 1                        ▷ set its distance from s
13:                 v.π ← u                              ▷ set its parent
14:                 ENQUEUE(Q, v)
15:             else
16:                 if v.color = BLACK ∧ v ≠ u.π then    ▷ 2nd time we see (u, v)
17:                     add (u, v) to the list of cross edges
18:                 if v.δ = u.δ then                    ▷ u and v are in the same layer
19:                     odd cycle detected!
20:         u.color ← BLACK                              ▷ we are done with u
```

Algorithm 3.2 shows how to detect cross edges and odd cycles using BFS, with only a few lines added (highlighted in red).

The reason we check only for $v.\texttt{color} = \textsc{black}$ at Line 16 is that if (u, v) is a cross edge, we will encounter it twice: once from u to v and once from v to u. When we first encounter it (say, wlog, from u), both vertices will be GRAY, so we do not count it yet. When we encounter it the second time (from v), u will already be BLACK, so we count it then. This way, we ensure that each cross edge is counted exactly once.

Proof ('⇒' of Theorem 3.3)**.** not possible to 2-color an odd cycle, let alone G. □

Corollary 3.4. any cycle in a bipartite graph has an even # edges. ◁

Lemma 3.5. For '⇐' of Theorem 3.3, we use the layers of BFS: Let G be a connected graph and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node s.

1. if no edge of G joins 2 nodes of the same layer, then G is bipartite
2. if an edge of G joins 2 nodes of the same layer, then G contains an odd length cycle, and hence is not bipartite ◁

Corollary 3.6 (of Lemma 3.5)**.**

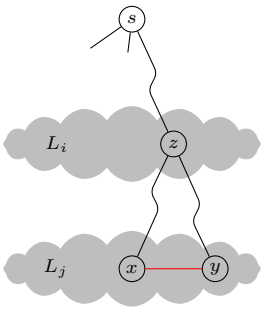
- G has no odd cycle iff no edge joins two nodes of the same BFS layer
- G is bipartite iff no edge joins two nodes of the same BFS layer ◁

Proof (of Lemma 3.5)**.** Suppose no edge joins two nodes in the same layer. Then we can 2-color the graph, e.g. red for odd layers and blue for even layers. So 3.5.1 holds.

Suppose (x, y) is an edge joining two nodes in the same layer L_j. Let

$$z = \text{lca}(x, y)$$

be the lowest level common ancestor in the BFS tree and let L_i be the layer of z.



Consider a cycle that takes edge (x, y), then the path in the BFS tree from y to z, and then the path in the BFS tree from z to x. The length of this cycle is

$$\underbrace{1}_{(x,y)} + \underbrace{(j-i)}_{y \rightsquigarrow z} + \underbrace{(j-i)}_{z \rightsquigarrow x} = 1 + 2(j-i)$$

which is odd. So 3.5.2 holds. □

Proof ('⇐' of Theorem 3.3)**.** Lemma 3.5. □

3.5 Depth-First Search

DFS is recursive process.

we maintain four auxiliary arrays:

- $u.\texttt{color}$: undiscovered (WHITE), discovered (GRAY), finished (BLACK).
- $u.d$ (discovery time): time when DFS started at vertex u.
- $u.f$ (finish time): time when u is finished processing (all neighbors have been visited).
- $u.\pi$ (predecessor pointer): vertex which discovered u. the edge $(u.\pi, u)$ is a tree edge in the DFS recursion tree.

Running time of DFS:

the call DFSVISIT is made exactly once for each vertex

ignoring time for recursive calls, each vertex u is processed in $O(1 + \deg(u))$ time

Total time:

$$T(n) = n + \sum_{u \in V} (\deg(u) + 1) = n + \underbrace{\sum_{u \in V} \deg(u)}_{=2m} + \underbrace{\sum_{u \in V} 1}_{=n}$$
$$= 2n + 2m \in O(n + m)$$

Since DFS always explores all vertices, we define the predecessor subgraph as $G_\pi = (V, E_\pi)$, where

$$E_\pi = \{(v.\pi, v) \mid v \in V \wedge v.\pi \neq \text{NIL}\}$$

It is a DFS *forest*, comprising several DFS trees.

Algorithm 3.3 DFS

```
1:  function DFS(G)
2:      for all u ∈ V do
3:          u.color, u.π ← WHITE, NIL                ▷ initialization
4:      T ← 0
5:      for all u ∈ V do
6:          if u.color = WHITE then                  ▷ undiscovered vertex?
7:              DFSVISIT(u)                          ▷ ...start a new search here
8:  function DFSVISIT(u)                             ▷ new DFS search at u
9:      u.color ← GRAY                               ▷ u has been discovered
10:     T ← T + 1
11:     u.d ← T
12:     for all v ∈ Γ(u) do
13:         if v.color = WHITE then                  ▷ first time we have seen v?
14:             v.π ← u
15:             DFSVISIT(v)                          ▷ ...visit it
16:     u.color ← BLACK                              ▷ we are done with u
17:     T ← T + 1
18:     u.f ← T
```

3.5.1 Edge Classification

for undirected graphs, there are two types of edges:

- *tree edges*
- *back edges*: $v.d < u.d$ AND $u.\pi \neq v$

Observation: for each edge in an undirected graph, either u is a proper ancestor ($u.d < v.d$) or a proper descendant ($u.d > v.d$) of v (by Lemma 3.7).

for directed graphs, there are three types of non-tree edges:

- *back edges* (u, v) where v is an ancestor (not necessarily proper) of u
- *forward edges* (u, v) where v is a proper descendant of u
- *cross edges* (u, v) where u and v are neither ancestor nor descendant of one another

3.5.2 Properties of the DFS Forest G_π

DFS imposes a nesting structure on the discovery-finish time intervals (no overlapping intervals!):

Lemma 3.7 (parenthesis)**.** for any two vertices u_1 and u_2:

- $[u_1.d, u_1.f] \subset [u_2.d, u_2.f] \iff u_1$ is a descendant of u_2
- $[u_1.d, u_1.f] \supset [u_2.d, u_2.f] \iff u_2$ is a descendant of u_1
- $[u_1.d, u_1.f] \cap [u_2.d, u_2.f] = \emptyset \iff$ neither is a descendant of the other ◁

Lemma 3.8. Let (u, v) be any graph edge. If it is a tree, forward, or cross edge, then $u.f > v.f$. If it is a back edge, then $u.f \leq v.f$. ◁

Proof. Lemma 3.7. □

Lemma 3.9. A directed graph has a cycle iff the DFS forest has a back edge. ◁

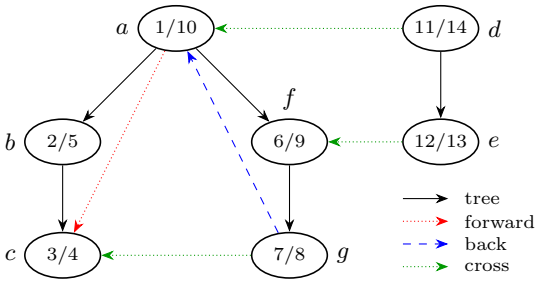
Proof. '⇐' easy: If there is a back edge then there is a cycle. '⇒': Consider a cycle. In a cycle there must be an edge (u, v) with $u.f < v.f$. By Lemma 3.8, (u, v) must be a back edge. □

When during an DFS an edge (u, v) is first explored, the color of v yields information about the edge:

- if v is WHITE, then (u, v) is a tree edge
- if v is GRAY, then (u, v) is a back edge
- if v is BLACK, then (u, v) is a
  - forward edge if $u.d < v.d$
  - cross edge if $u.d > v.d$

Observe that the gray vertices always form a linear chain of descendants corresponding to the stack of active DFSVISIT invocations.

Example 3.1 (Edge Classification)**.**



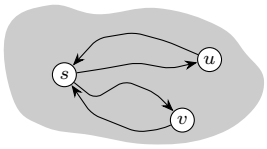
◀

3.6 Connectivity in Directed Graphs

Definition 3.8 (strongly connected)**.** A directed graph is *strongly connected* if for each u and v, there is a path from u to v and a path from v to u. ✍

Lemma 3.10. Let s be any node. G is strongly connected iff every node is reachable from s AND s is reachable from every node. ◁

Proof. '⇒' follows from definition. '⇐': Consider two nodes u and v.



Path from u to v: Concatenate $u \rightsquigarrow s$ and $s \rightsquigarrow v$. Path from v to u: Concatenate $v \rightsquigarrow s$ and $s \rightsquigarrow u$. □

Theorem 3.11. We can test if G is strongly connected in $O(n+m)$ time. ◁

Proof. Pick any node s. Run BFS (or DFS) from s in G. Run BFS (or DFS) from s in $G^\top$. If all nodes are reached in both searches, then G is strongly connected (by Lemma 3.10). □

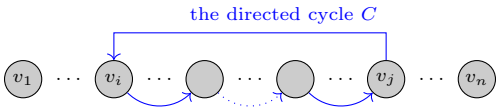
Theorem 3.12 (Tarjan, 1972)**.** The strongly connected components of a directed graph can be computed in $O(n+m)$ ◁

3.7 Topological Sort

Definition 3.9 (topological order)**.** A *topological order* of a directed $G = (V, E)$ is a ordering of its vertices $v_1, \ldots, v_n$ such that for every edge (v_i, v_j), we have $i < j$. ✍

Lemma 3.13. If G has a topological order, then G is a DAG. ◁

Proof (Contradiction)**.** Suppose G has a topological order $v_1, \ldots, v_n$ and also has a directed cycle C. Let v_i be the lowest-indexed node in C and v_j be the node just before v_i in C:



Then (v_j, v_i) is an edge with $j > i$, by our choice of v_i. This contradicts the definition of a topological order, which requires $i < j$ for every edge (v_i, v_j). □

Lemma 3.14. If G is a DAG, then G has a node with no incoming edges (a *source*). ◁

Proof (Contradiction)**.** Suppose G is a DAG and every node has at least one incoming edge. Pick any node v, and begin following edges backwards from v. Since every node has an incoming edge, we can repeat this process and if G is finite, we must eventually revisit a node, say w. Let C denote the sequence of nodes encountered between successive visits to w. Then C is a cycle. □

Caution 3.1. The converse of Lemma 3.14 does not hold! ◀

Lemma 3.15. If G is a DAG, then G has a topological order. ◁

Proof (Induction)**.** The inductive proof spells out a recursive algorithm.

(i) Base case: For $n = 1$, we just have one node, which trivially has a topological order. ✔

(ii) Induction hypothesis: Assume a DAG of $n \geq 2$ nodes has a topological order.

(iii) Induction step: Let G be a DAG with $n+1$ nodes. By Lemma 3.14, G has a source node, say v. $G \setminus \{v\}$ is a DAG (removing v cannot create cycles) with n nodes. By (ii), $G \setminus \{v\}$ has a TO. Append v to the front of this TO. This is a TO for G since v has no incoming edges. ✔ □

Theorem 3.16. Algorithm 3.4 finds a TO in $O(n+m)$ time. ◁

Proof. Maintain the following information:

- `in_degree`$[w]$: number of incoming edges to node w

Algorithm 3.4 Topological Sort (Recursive)

```
1: function TopologicalSort(G)
2:     if G is empty then
3:         return empty list                          ▷ recursion bottoms out here
4:     Find a node v with no incoming edges
5:     Remove v from G
6:     L ← [v] concatenated with TopologicalSort(G \ {v})
7:     return L
```

- S: set of nodes with no incoming edges

Initialization: $O(n + m)$, one scan through G to find out the in-degrees of all nodes and populate S.

When removing a node v from S, we decrement `in_degree`$[w]$ for each outgoing edge (v, w). If `in_degree`$[w]$ hits zero, we add w to S. This is $O(1)$ work per edge incident to v. □

Algorithm 3.5 is an alternative way to find a TO of a DAG using DFS.

Algorithm 3.5 Topological Sort (DFS)

```
1: function topSort(G)
2:     for all u ∈ V do
3:         u.color ← WHITE                            ▷ initialization
4:     S ← ∅                                          ▷ empty stack
5:     for all u ∈ V do
6:         if u.color = WHITE then
7:             topVisit(u)
8:     while S ≠ ∅ do                                 ▷ while stack not empty
9:         output pop(S)                              ▷ pop stack for final TO
10: function topVisit(u)
11:     u.color ← GRAY                                 ▷ mark u visited
12:     for all v ∈ Γ(u) do
13:         if v.color = WHITE then
14:             topVisit(v)
15:     push u onto S                                  ▷ last to finish is top of stack
```

In Lemma 3.8 we have seen that if (u, v) is a tree/forward/cross edge, then $u.f > v.f$. Since a DAG has no back edges (it is acyclic), we have $u.f > v.f$ for every directed edge (u, v). Thus, ordering vertices in decreasing order of finish times when running Algorithm 3.3 yields a TO. Hence, Algorithm 3.5 is correct.

Remark 3.2. Vertex with highest finish time in DFS is a source of the DAG. ◀

3.8 Applications of DFS

3.8.1 Longest Path in a DAG

Given a DAG $G = (V, E)$, where each vertex u is thought of as a task that takes u.`time` time units to complete, and each edge (u, v) represents a precedence constraint (task u must be completed before task v can begin), we want to know the minimum time required to complete all tasks, assuming maximum parallelism. This is equivalent to computing the maximum cost of any path in the DAG, where cost is defined as the sum of u.`time` values of the vertices along the path.

We can solve this in $O(n + m)$ time through DFS. The idea is to associate each vertex u with the maximum cost of any path that starts at this vertex, denoted by u.`cost`. We let `max_cost` be the maximum cost of all u's neighbors.

Because the graph is acyclic, every edge (u, v) goes from u to a vertex v whose finish time is greater than u's (by Lemma 3.8). Therefore, v.`cost` is fully defined before it is accessed by u. The longest path in the entire DAG is the largest value of u.`cost` among all vertices u.

Algorithm 3.6 Longest Path in a DAG

```
1: function LongPathVisit(u)
2:     u.color ← GRAY                          ▷ mark u visited
3:     max_cost ← 0                            ▷ initialize max outgoing cost
4:     for all v ∈ Γ(u) do
5:         if v.color = WHITE then
6:             LongPathVisit(v)                ▷ process v if undiscovered
7:         max_cost ← max(max_cost, v.cost)    ▷ update maximum cost
8:     u.cost ← max_cost + u.time              ▷ save final cost
```

3.8.2 Biconnected Components

Let $G = (V, E)$ be a connected, undirected graph.

Definition 3.10 (cut vertex)**.** any vertex whose removal (along with incident edges) disconnects the graph ✍

Definition 3.11 (bridge)**.** any edge whose removal disconnects the graph ✍

Definition 3.12 (biconnected)**.** a graph is biconnected if it contains no cut vertices ✍

a graph is k-connected if it remains connected after removing any $k - 1$ vertices.

Definition 3.13 (biconnected component)**.** partition E into maximal subgraphs that are biconnected. ✍

Biconnected components are the equivalence classes of the co-cyclicity relation. Two edges $e_1, e_2 \in E$ are co-cyclic if there is a simple cycle that contains both, or if $e_1 = e_2$.

G is biconnected iff it consists of one biconnected component.

Lemma 3.17. The root r of the DFS tree is a cut vertex iff it has ≥ 2 children. ◁

Lemma 3.18. An internal (i.e. not a leaf and not the root) vertex u of the DFS tree is a cut vertex iff there exists a subtree rooted at a child v of u such that there is no back edge from any vertex in this subtree to a proper ancestor of u. ◁

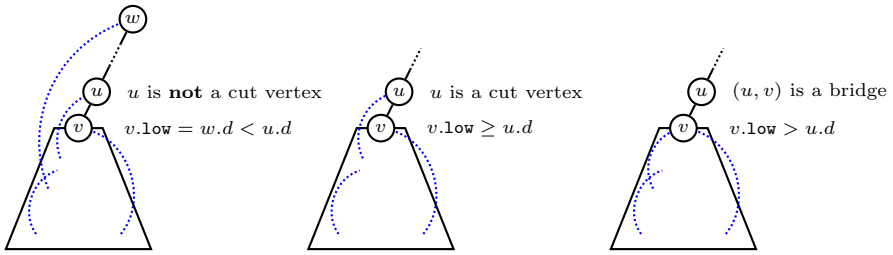
We can exploit the structure of the DFS tree to determine cut vertices.

Keeping track of all back edges from each subtree is expensive.

Instead we can just keep track of one back edge from each subtree, the one that goes highest (closest to the root) in the tree.

Observe: discovery times decrease as we go up the tree.

Idea: keep track of back edge (x, w) with smallest discovery time $w.d$



Definition 3.14 (low value)**.** highest up we can reach from the subtree rooted at v (via back edges):

$$v.\texttt{low} := \min\left(\{v.d\} \cup \left\{w.d \;\middle|\; \begin{array}{l}(x,w) \text{ is a back edge for some} \\ \text{(nonproper) descendant } x \text{ of } v\end{array}\right\}\right)$$

where "nonproper" means that x can be v itself. ✍

Once $v.\texttt{low}$ is computed for all vertices v, we can test wether a given non-root vertex u is a cut vertex by Lemma 3.18 as follows: u is a cut vertex iff it has a child v in the DFS tree such that $v.\texttt{low} \geq u.d$. Moreover, (u, v) is a bridge iff it is a tree edge (i.e. $u = v.\pi$) and $v.\texttt{low} > u.d$.

Algorithm 3.7 Modification of DFSVISIT for `low` computation

```
1:  function LowComputation(u)
2:      u.color ← GRAY                                  ▷ u has been discovered
3:      T ← T + 1
4:      u.low ← u.d ← T                                 ▷ set discovery time and init low
5:      for all v ∈ Γ(u) do
6:          if v.color = WHITE then                     ▷ (u, v) is a tree edge
7:              v.π ← u                                 ▷ v's parent is u
8:              LowComputation(v)
9:              u.low ← min(u.low, v.low)               ▷ update u.low (propagate v.low up)
10:         else if v ≠ u.π then                        ▷ (u, v) is a back edge
11:             u.low ← min(u.low, v.d)                 ▷ update u.low (consider back edge to v)
```

Algorithm 3.8 Modification of DFSVISIT to find cut vertices

```
1:  function findCutVertices(u)
2:      u.color ← GRAY                                  ▷ u has been discovered
3:      T ← T + 1
4:      u.low ← u.d ← T                                 ▷ set discovery time and init low
5:      for all v ∈ Γ(u) do
6:          if v.color = WHITE then                     ▷ (u, v) is a tree edge
7:              v.π ← u                                 ▷ v's parent is u
8:              findCutVertices(v)
9:              u.low ← min(u.low, v.low)               ▷ update u.low (propagate v.low up)
10:             if u.π = NIL then                       ▷ u is root: apply Lemma 3.17
11:                 if this is u's second child then
12:                     label u as a cut vertex
13:             else if v.low ≥ u.d then                ▷ u is internal: apply Lemma 3.18
14:                 label u as a cut vertex
15:         else if v ≠ u.π then                        ▷ (u, v) is a back edge
16:             u.low ← min(u.low, v.d)                 ▷ update u.low (consider back edge to v)
```

Algorithm 3.9 Modification of DFSVISIT to find bridges

```
1:  function findBridges(u)
2:      u.color ← GRAY
3:      T ← T + 1
4:      u.low ← u.d ← T                                 ▷ discover u
5:      for all v ∈ Γ(u) do
6:          if v.color = WHITE then                     ▷ (u, v) is a tree edge
7:              v.π ← u
8:              findBridges(v)
9:              u.low ← min(u.low, v.low)
10:             if v.low > u.d then                     ▷ bridge test
11:                 label edge (u, v) as a bridge
12:         else if v ≠ u.π then                        ▷ (u, v) is a back edge
13:             u.low ← min(u.low, v.d)
```

If we want to to find the biconnected components of G, we can store the edges in a stack while performing Algorithm 3.8. Whenever we reach a cut vertex, edges in a biconnected component will be consecutive on the stack. See Algorithm 3.10.

When do we output a biconnected component? We do *not* only output components "when we reach a cut vertex". Instead, we output a component *after returning from a recursive call* at a child v of u, precisely when $v.\texttt{low} \geq u.d$. This condition means that the subtree rooted at v has no back edge to a *proper* ancestor of u ("proper" means excluding u). Consequently, all edges pushed onto the edge stack $\mathcal{S}$ since traversing the tree edge (u, v) belong to a single biconnected component, and they appear *consecutively* on $\mathcal{S}$. Hence we can pop edges until (u, v) to obtain exactly that component.

Algorithm 3.10 Modification of DFSVISIT to find biconnected components (edge stack)

```
1:  function FINDBICC(u)
2:      u.color ← GRAY                                   ▷ u has been discovered
3:      T ← T + 1
4:      u.low ← u.d ← T                                  ▷ set discovery time and init low
5:      for all v ∈ Γ(u) do
6:          if v.color = WHITE then                      ▷ (u, v) is a tree edge
7:              v.π ← u                                  ▷ v's parent is u
8:              push (u, v) onto S                       ▷ store tree edge on the edge stack
9:              FINDBICC(v)
10:             u.low ← min(u.low, v.low)                ▷ propagate v.low up
11:             if v.low ≥ u.d then                      ▷ biconnected-component boundary at (u, v)
12:                 pop from S until tree edge (u, v) is popped and output them as a BiCC
13:         else if v ≠ u.π and v.d < u.d then           ▷ (u, v) is a back edge to an ancestor
14:             push (u, v) onto S                       ▷ store back edge on the edge stack
15:             u.low ← min(u.low, v.d)                  ▷ consider back edge to v
```

For internal vertices u, the same inequality $v.\texttt{low} \geq u.d$ is also the cut-vertex test. The root requires a separate cut-vertex criterion (Lemma 3.17), while biconnected components must still be output even if the root is *not* a cut vertex (e.g. when the entire graph is biconnected).

3.8.3 Strongly Connected Components

Definition 3.15. A di-graph is strongly connected if every pair of nodes u, v is mutually reachable, i.e., there is a path from u to v and also a path from v to u. ✍

mutual reachability relation between vertices is an equivalence relation.

partitions V into equivalence classes: *strongly connected components* of G

if we collapse the vertices within each strong component into a single vertex, we get the *component digraph*. it is a DAG.

- if there is an edge (or path) from component C to component C', then there cannot be an edge (or path) from C' to C (or else $C \cup C'$ would be one strong component).
- thus, there is no cycle in the component digraph

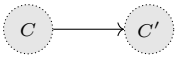
strong components can be computed in $O(n + m)$ time using two DFS:

- recall that we can answer if a digraph is strongly connected by 2 BFS/DFS

If we run our usual DFS and record the finish times we make the following observations:

Observation 3.3. Node of highest finish time must be a source (in-degree 0) of the component digraph. ◀

Observation 3.4. More generally, if C and C' are two strong components such that there is an edge from a vertex in C to a vertex in C', then



$$\max_{u \in C} u.f > \max_{v \in C'} v.f \tag{3.1}$$

i.e. the highest finish time in C is greater than the highest finish time in C'. ◀

Proof. We distinguish two cases:

- Suppose DFS first encounters a vertex u in C. Then it will visit all vertices in C' and C before u can finish. Thus, u will have higher finish time than every vertex in $C \cup C'$.
- Suppose DFS first encounters a vertex v in C'. Then it will "get stuck" in C' (since there are no edges from C' to C). So, all vertices in C will have higher discovery times than those in C'; so they will also have higher finish times. □

Caution 3.5. It may be tempting at this point to try to conclude from Observation 3.3 and Observation 3.4 that the node of lowest finish time must be in a sink (out-degree 0) of the component digraph. However, this is *not* true in general, as one can see from the following counterexample: $V = \{a, b, c\}$, $E = \{(a, b), (b, a), (a, c)\}$. If we run DFS starting from a, exploring neighbors in alphabetical order, we get $b.f < c.f < a.f$, even though b is not in a sink component. ◀

Using DFS we can identify the vertex of highest finish time. It is a vertex in a source node of the component DAG G^{SCC}.

but we want to identify a vertex in a sink of the component DAG:

- if we can do this, then we can start a DFS at a vertex in the sink component, and identify all vertices of this component
- (no other strong component will be visited at this DFS call, as there is no edge from the sink component to another one)
- repeat (after ignoring the vertices of this component)

trick:

- reverse the edges of G to get the transpose graph $G^{\top}$
- strong component of $G^{\top}$ are the same as those of G
- the edges of the component DAG of $G^{\top}$ are the reversed edges of the component DAG of G
- the vertex of highest finish time in $G^{\top}$ must be in the sink node of the original component DAG of G (by Observation 3.3)

Algorithm 3.11 Strongly Connected Components

```
1: function StronglyConnectedComponents(G)
2:     create the transpose graph G^⊤
3:     call DFS(G^⊤) to compute the finish time v.f for each v ∈ V
4:     as the vertices finish (Line 16), push them on a stack
5:     call DFS(G), but in Line 5, consider the vertices in order of decreasing u.f
6:     return the depth-first trees of G_π formed in Line 5
```

By considering vertices in the 2nd DFS in decreasing order of finish times from the 1st DFS, we are visiting the vertices of the component DAG in topologically sorted (reverse) order.

4 Greedy Algorithms

4.1 Interval Scheduling

Given n jobs, we want to schedule a maximum number of non-overlapping jobs, i.e.

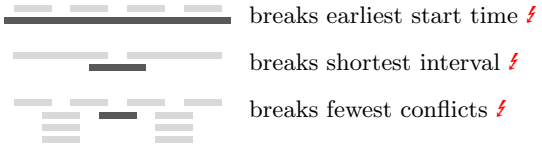
- Job j starts at s_j and finishes at f_j
- Two jobs are compatible if they don't overlap
- Goal: find a *maximum cardinality* subset of mutually compatible jobs

greedy template: Consider jobs in some order. Take a job if it is compatible with all previously selected jobs.

What order?

- (Earliest start time) ascending order of start time s_j.
- (Shortest interval) ascending order of interval length $f_j - s_j$.
- (Fewest conflicts) for each job, count the number of conflicting jobs c_j. ascending order of conflicts c_j.
- (Earliest finish time) ascending order of finish time f_j.

all of the above are valid thoughts, but can easily find counterexample for the first three of them:



(a correct) greedy strategy: increasing order of finish time ✔

or symmetrically: decreasing order of start time ✔

Algorithm 4.1 Greedy Interval Scheduling

```
1: sort jobs by finish time: f_1 ≤ f_2 ≤ ... ≤ f_n
2: A ← ∅                                          ▷ selected jobs
3: for j = 1, ..., n do
4:     if job j is compatible with (last) job in A then
5:         A ← A ∪ {j}
6: return A
```

Implementation. $O(n \log n)$ for sorting

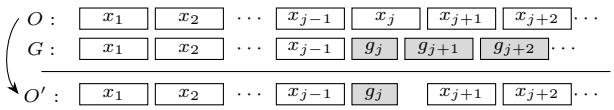
- remember job j^* that was added last to A
- job j is compatible with A if $s_j \geq f_{j^*}$

Theorem 4.1 (Optimality)**.** Algorithm 4.1 is optimal. ◁

Proof. Consider an optimal schedule O, and let G be the greedy schedule. Let $O = [x_1, x_2, \ldots, x_k]$ the activities of O listed in increasing order of finish time, and let $G = [g_1, g_2, \ldots, g_{k'}]$ be the corresponding greedy schedule. If $G = O$, we are done. Otherwise, let j be the first index where $x_j \neq g_j$, i.e.

$$O = [x_1, x_2, \ldots, x_{j-1}, x_j, x_{j+1}, \ldots, x_k]$$
$$G = [x_1, x_2, \ldots, x_{j-1}, g_j, g_{j+1}, \ldots, g_{k'}]$$

where $g_j \neq x_j$. Note that $k \geq j$ since otherwise G would have more activities than O, contradictory. Algorithm 4.1 selects the the activity with the earliest finish time that does not conflicct with any earlier activity. Thus, we know that g_j does not conflict with any earlier activity, and it finishes no later than x_j finishes:



Consider the the modified 'greedier' schedule O' obtained by replacing x_j with g_j in O. That is,

$$O' = [x_1, x_2, \ldots, x_{j-1}, g_j, x_{j+1}, \ldots, x_k]$$

it is a valid schedule, because g_j finishes no later than x_j and therefore cannot create any new conflicts. It has the same # activities as O, so it is at least as good. By repeating this process, we will eventually convert O into G without ever decreasing the number of activities. Thus, G is optimal. □

4.2 Interval Partitioning

Given n jobs, we want to schedule them all using a minimum number of resources (e.g., classrooms), i.e.

- Job j starts at s_j and finishes at f_j
- Goal: find a *minimum number* of resources (e.g., classrooms) to schedule all jobs

Definition 4.1 (depth)**.** The *depth* of a set of open intervals is the maximum number that contain any given time (vertical line), i.e.

$$\text{depth} = \max_t A(t)$$

where $A(t)$ is the number of intervals that contain (i.e. that are *active* at time t). ✍

Key observation: $d = \#\,\text{resources} \geq \text{depth}$

Greedy strategy: Consider jobs in increasing order of start time. Assign job to any compatible resource, or open a new resource.

Algorithm 4.2 Greedy Interval Partitioning

```
sort jobs by starting time: s_1 ≤ s_2 ≤ ... ≤ s_n
d ← 0                                   ▷ number of allocated resources
for j = 1, ..., n do
    if job j is compatible some available resource k then
        schedule job j on resource k
    else
        allocate a new resource d + 1
        schedule job j on resource d + 1
        d ← d + 1
```

Implementation. $O(n \log n)$

- sort takes $O(n \log n)$
- for each ressource k, maintain the finish time of the last job added
- avoid quadratic algorithm (scanning all allocated d resources every single time)
- first idea: priority queue (min-heap) of ressources, keyed by time they become free
- interact with the heap in $O(\log d)$ time per job: get-min, del-min, insert

But actually, we can do better (for the part after sorting):

- maintain a stack/queue Q of available resources
- initially Q is empty
- create a sorted list of $2n$ job events: turn every interval $[s_i, f_i)$ into two events $(s_i, \texttt{start})$ and $(f_i, \texttt{finish})$. finish events come before start events at the same time (lets a job ending at time t free a room that a job starting at time t can immediately reuse).
- at a start event a ressource is allocated from Q (or a new one $(++d)$ is created if Q is empty)
- at a finish event, a ressource is freed and put back onto Q
- principle of 1D plane sweep by vertical line from left to right

Theorem 4.2 (Optimality)**.** Algorithm 4.2 is optimal. ◁

Proof. We can show that $d \leq$ depth (enough to prove optimality):

- Ressource d is allocated because we needed to schedule a job j that is incompatible with all the current $d-1$ ressources, which hold jobs that started before s_j (no later than s_j) and none has finished yet.
- Consider time $s_j + \varepsilon$. At that time $(d-1)+1 = d$ intervals are overlapping and since since depth is the maximum number of overlapping intervals at any time, we have $d \leq$ depth. □

4.3 Minimizing Lateness

Given n jobs, where

- job j requires t_j units processing time and is due at d_j (deadline)
- if job j starts at time s_j, it finishes at $f_j = s_j + t_j$
- lateness: $\ell_j = \max(0, f_j - d_j)$ finish time - deadline
- goal: schedule all jobs to *minimize* maximum lateness $L = \max_j \ell_j$

greedy template: Consider jobs in some order.

- (Shortest processing time) ascending order of processing time t_j.
- (Earliest deadline) ascending order of deadlines d_j.
- (Smallest slack) ascending order of slack time $s_j = d_j - t_j$

but there are problems for first and last:

- short job may have a long deadline, e.g. $[(t_1, d_1) = (1, 100), (t_2, d_2) = (10, 10)]$ ↯
- long job may have 0 slack, e.g. $[(t_1, d_1) = (1, 2), (t_2, d_2) = (10, 10)]$ ↯

the correct greedy strategy is: earliest deadline first ✔

Algorithm 4.3 Greedy Minimizing Lateness

```
sort jobs by deadline: d_1 ≤ d_2 ≤ ... ≤ d_n
t ← 0                                          ▷ current time
for j = 1, ..., n do
    assign job j to interval [t, t + t_j]
    s_j ← t
    f_j ← t + t_j
    t ← t + t_j
return intervals [s_j, f_j]
```

Observation 4.1. There exists an optimal schedule with no idle time (idle time gives no benefit). ◀

Observation 4.2. The greedy schedule has no idle time (by construction). ◀

Definition 4.2 (inversion)**.** An inversion in schedule S is a pair of jobs i, j such that $d_i < d_j$ but j is scheduled before i. ✍

Observation 4.3. The greedy schedule has no inversions (by construction - ascending order of deadlines). ◀

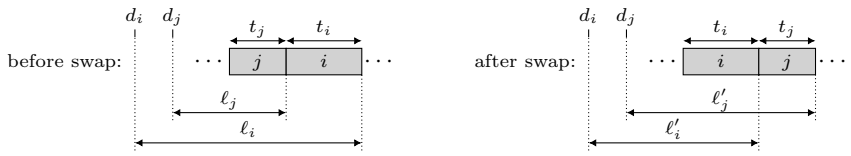
Observation 4.4. If a schedule (with no idle time) has an inversion, there is an inversion with two inverted jobs scheduled consecutively. ◀

Claim 4.3. Swapping two adjacent inverted jobs reduces number of inversions by 1 and does not increase the max lateness $\ell_{\max} := \max\{\ell_1, \ldots, \ell_n\}$. ◁

Proof. Let ℓ be the lateness before the swap and let ℓ' be it after the swap.

- $\ell'_k = \ell_k$ for all $k \neq i, j$ (no change)
- $\ell'_i \leq \ell_i$ because i finishes earlier after the swap
- if job j was late before the swap:

$$\begin{aligned} \ell'_j &= f'_j - d_j && \text{(definition)} \\ &= f_i - d_j && (j \text{ now finishes when } i \text{ used to finish}) \\ &\leq f_i - d_i && (\text{since } d_i < d_j) \\ &= \ell_i && \text{(definition)} \end{aligned}$$



Thus we have $\ell'_{\max} = \max\{\ell'_i, \ell'_j, \text{others}\} \leq \max\{\ell_i, \text{others}\} = \ell_{\max}$. □

Theorem 4.4 (Optimality)**.** Algorithm 4.3 is optimal. ◁

Proof. Let G be the greedy schedule produced by Algorithm 4.3. Let O be an optimal schedule with the fewest number of inversions. We can assume O has no idle time. If O has no inversions, then $O = G$. Otherwise, let i, j be two adjacent inverted jobs in O. By Claim 4.3, swapping i and j does not increase the maximum lateness and strictly decreases the number of inversions. This contradicts the definition of O. ↯ □

Alternatively, we can prove Theorem 4.4 using an exchange argument: We can take O and transform into G by removing inversions 1 by 1 with no change in max lateness.

4.4 Analysis Strategies and Approximation

4.4.1 Correctness Proofs

Greedy algorithm stays ahead. Show that after each step of the greedy algorithm, its solution is at least as good as any other algorithm's.

Exchange argument. Gradually transform any solution to the one found by the greedy algorithm without hurting its quality.

Structural. Discover a simple "structural" bound asserting that every possible solution must have a certain value. Then show that your algorithm always achieves this bound.

Counterexample. To show that a greedy algorithms does *not* work, all we need is a small counterexample.

4.4.2 Greedy Approximation for NP-Hard Problems

The greedy approach is not very powerful as an algorithm design technique.

- One of most common applications of greedy algorithms is to produce approximation solutions to NP-hard problems (see Section 7.4).
- NP-hard optimization problems are challenging computational problems with no known exact solution of worst-case polynomial-time running time.
- Given an NP-hard problem, there are no ideal solutions. Compromise between optimality or running time.
- Instances of such problems where simple greedy algorithms produce solutions that are not far from optimal. E.g., clustering and Facility Location.

4.5 Shortest Paths

Shortest path network.

- directed of undirected graph $G = (V, E)$
- source s, destination/target t
- length/weight function $w : E \to \mathbb{R}$ (assigns cost/length to each edge)

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

Dijkstra's algorithm

- assume $w(e) \geq 0$ for all edges e (non-negative weights)
- maintain a set of explored nodes S for which we have determined the the shortest path distance $u.\delta$ from s to u
- initialize $S = \{s\}$ and $s.\delta = 0$
- repeatedly choose unexplored node v which minimizes

$$d_S(v) := \min_{e=(u,v), u \in S} (u.\delta + w(e)) \tag{4.1}$$

  where $d_S(v)$ is the shortest path to some u in explored part S, followed by a single edge (u, v), i.e. it is the shortest path from s to v using only vertices in S. Add v to S and set $v.\delta = d_S(v)$.

two sets of vertices: S (explored nodes) and $V \setminus S$ (unexplored)

- $V \setminus S$ organized in a priority queue Q (heap of min d value)
- vertices in Q have weight $d_S(v)$: length of shortest path to v using only vertices in S
- vertices in S have weight $u.\delta$: length of shortest s-u path
- every time a vertex u gets explored (joins S), we "relax" all incident edges (u, v): for all vertices v adjacent to u: update $d_S(v)$ to $\min(d_S(v), u.\delta + w(u, v))$
- next node to explore = node with minimum $d_S(v)$

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

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

[†] Individual operations are amortized bounds.

Time analysis (using a binary heap).

- extract a vertex u from priority queue Q: $O(\log n)$
- for each incidnt edge (u, v), relax it in $O(1)$ and spend $O(\log n)$ if we decrease the key of v in Q

Thus, time to process vertex u is $O(\log n + \deg(u) \log n)$ and total time is

$$\begin{aligned} T(n, m) &= \sum_{u \in V} (\log n + \deg(u) \log n) \\ &= \log n \sum_{u \in V} (1 + \deg(u)) \\ &= (\log n)(n + 2m) \\ &= O((n + m) \log n) \end{aligned}$$

Algorithm 4.4 Dijkstra

```
1:  function DIJKSTRA(G, w : E → ℝ≥0, s)
2:      for all v ∈ V do                                   ▷ initialization
3:          v.δ ← ∞                                        ▷ best known cost from s to v
4:          v.color ← undiscovered
5:          v.π ← NIL                                      ▷ node preceding v on the least-cost path from s
6:      s.δ ← 0                                            ▷ s is the source
7:      Q ← priority queue of all nodes u ∈ V sorted by u.δ
8:      while Q ≠ ∅ do                                     ▷ until all vertices are processed
9:          extract u with minimal u.δ from Q
10:         for all v ∈ Γ(u) do
11:             if u.δ + w(u, v) < v.δ then                ▷ relax edge (u, v), i.e. update d_S(v)
12:                 v.δ ← u.δ + w(u, v)
13:                 decrease key of v in Q to v.δ
14:                 v.π ← u
15:         u.color ← finished
16:     ▷ the π pointers define an 'inverted' shortest path tree ◁
```

Observation 4.5. For all $u \in V \setminus S$ we have

$$u.\delta = d_S(u)$$

i.e. the key in the PQ is exactly $d_S(u)$. ◀

Caution 4.6. We use the same variable/attribute $u.\delta$ to store both $d_S(u)$ for the $u \in V \setminus S$ and 'later' $\delta(s, u)$ for the $u \in S$.

Only once a vertex u is extracted from the queue Q in Line 9 (which means it is added to S), we are guaranteed that $u.\delta = \delta(s, u)$. ◀

Invariant 4.7. For all $u \in S$ we have

$$u.\delta = \delta(s, u)$$

i.e. it is the length of the shortest s-u path. ◀

Proof (Induction on $|S|$)**.**

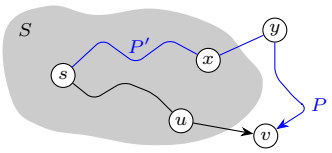
(i) Base case: $|S| = 1$ is trivial: $u.\delta = s.\delta = 0$ ✔

(ii) Induction hypothesis: Assume true for $|S| = k \geq 1$.

(iii) Induction step: Prove true for $|S| = k + 1$.

Let v be the $(k+1)^{\text{th}}$ node added to S, and let (u, v) be the chosen edge, and therefore $v.\delta = u.\delta + w(u, v)$.

Consider any s-v path P:



Let (x, y) be the first edge in P that leaves S, and let P' be the subpath to x. For the length ℓ of P, we have

$$\ell(P) \overset{w(e)\geq 0}{\geq} \ell(P') + w(x, y) \overset{\text{(ii)}}{\geq} x.\delta + w(x, y) \overset{(4.1)}{\geq} d_S(y) \overset{\text{Line 9}}{\geq} v.\delta$$

Since P is arbitrary, this in particular holds for the shortest path, hence $\delta(s, v) \geq v.\delta$.

By (ii), $u.\delta = \delta(s, u)$, so the path consisting of the shortest s-u path followed by (u, v) has length $\delta(s, u) + w(u, v) = v.\delta$. Hence, $\delta(s, v) \leq v.\delta$.

Thus, $v.\delta = \delta(s, v)$ shortest path from s to v. ✔ □

4.6 Minimum Spanning Tree

Prim similar to Dijkstra

Kruskal similar to Strongly Connected Components

Definition 4.3 (MST)**.** Given a connected graph $G = (V, E)$ with real-valued edge weights c_e, an MST is a subset of the edges $T \subseteq E$ such that T is a spanning tree whose sum of edge weights is minimized. ✍

From Cayley's theorem we know that we do not want to find an MST by brute force.

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

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

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

Remark 4.8. All three algorithms produce an MST. ◀

4.6.1 Properties of MSTs

Simplifying assumption. All edge costs c_e are distinct. (unique MST)

Property 4.5 (Cut)**.** Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then the MST contains e. ◁

Proof (Exchange argument)**.** Suppose e does not belong to the MST T^*. Since T^* is a spanning tree there must be an edge f in T^* connecting S and $V \setminus S$. But $c_e < c_f$. Since T^* is a tree, f is the only such edge. Now let $T' = T^* \cup \{e\} \setminus \{f\}$. It is also a spanning tree. But $\text{cost}(T') < \text{cost}(T^*)$, which is a contradiction. ↯ □

Caution 4.9. Property 4.5 is an implication, not a biconditional. The reverse direction is false:

$$(e \text{ is in an MST}) \not\Rightarrow (e \text{ is lightest across some given cut})$$

and also

$$(e \text{ is not lightest across some given cut}) \not\Rightarrow (e \text{ is not in an MST})$$ ◀

Property 4.6 (Cycle)**.** Let C be any cycle, and let f be the max cost edge belonging to C. Then the MST does not contain f. ◁

Proof (Exchange argument)**.** Suppose f does belong to the MST T^*. Deleting f from T^* disconnects S and $V \setminus S$ in T^*. Since C is a cycle there is another edge e incident to S and $V \setminus S$ and $c_e < c_f$. Now let $T' = T^* \cup \{e\} \setminus \{f\}$. It is also a spanning tree. But $\text{cost}(T') < \text{cost}(T^*)$, which is a contradiction. ↯ □

Theorem 4.7. Let T_0 be an MST on $G_0 = (V, E)$. If a new edge $e' = (a, b)$ is added to G_0, we obtain a new graph $G_1 = (V, E \cup \{e'\})$. The MST changes if and only if e' is cheaper than the most expensive edge $f := \arg\max_{e \in P} c_e$ on the unique path P from a to b in T_0. Moreover, in that case the new MST is $T_1 := T_0 \cup \{e'\} \setminus \{f\}$. ◁

Proof. Adding e' to T_0 creates exactly one cycle $C := P \cup \{e'\}$.

'⇒' (change implies $c_{e'} < c_f$):

Assume $c_{e'} > c_f$. Then e' is the maximum cost edge on the cycle C. By Property 4.6, no MST of G_1 contains e'. Let T^* be the MST of G_1. Since $e' \notin T^*$, T^* uses only edges from E, so T^* is a spanning tree of G_0. Thus $\text{cost}(T_0) \leq \text{cost}(T^*)$. On the other hand, T_0 is also a spanning tree of G_1, hence $\text{cost}(T^*) \leq \text{cost}(T_0)$. Therefore $\text{cost}(T^*) = \text{cost}(T_0)$, and by uniqueness $T^* = T_0$. Taking the contrapositive yields the desired result.

'⇐' ($c_{e'} < c_f$ implies change and gives T_1):

Assume $c_{e'} < c_f$. Then T_1 is a spanning tree of G_1 and

$$\text{cost}(T_1) = \text{cost}(T_0) + c_{e'} - c_f < \text{cost}(T_0)$$

so T_0 cannot remain an MST of G_1 (hence the MST changes).

It remains to show T_1 is optimal in G_1. Let S be any spanning tree of G_1. Now distinguish two cases:

- $e' \in S$:

  Removing e' disconnects S into components with vertex sets A, B where $a \in A$ and $b \in B$. Along the (previously defined) path P there exists an edge $h \in P$ crossing the cut (A, B). Then $S \setminus \{e'\} \cup \{h\}$ is a spanning tree of G_0, so by minimality of T_0 on G_0,

  $$\text{cost}(T_0) \leq \text{cost}(S \setminus \{e'\} \cup \{h\}) = \text{cost}(S) - c_{e'} + c_h$$

  hence

  $$\text{cost}(S) \geq \text{cost}(T_0) + c_{e'} - c_h \geq \text{cost}(T_0) + c_{e'} - c_f = \text{cost}(T_1)$$

  where we used $c_h \leq c_f$ since $h \in P$ and f is the maximum edge on P. Thus among all spanning trees containing e', T_1 has minimum cost.

- $e' \notin S$:

  Any spanning tree of G_1 not containing e' is already a spanning tree of G_0 and therefore has cost at least that of T_0, i.e. $\text{cost}(S) \geq \text{cost}(T_0) > \text{cost}(T_1)$.

Consequently T_1 is the (unique) MST of G_1. □

4.6.2 Prim's algorithm

Algorithm 4.5 Prim

```
1:  function PRIM(G, w : E → ℝ, s)
2:      for all v ∈ V do                                   ▷ initialization
3:          v.key ← ∞
4:          v.color ← undiscovered
5:      s.key ← 0                                          ▷ start at root s
6:      s.π ← NIL
7:      Q ← priority queue of all nodes u ∈ V sorted by u.key
8:      while Q ≠ ∅ do                                     ▷ until all vertices are processed
9:          extract u with minimal u.key from Q
10:         for all v ∈ Γ(u) do
11:             if v.color = undiscovered ∧ w(u, v) < v.key then
12:                 v.key ← w(u, v)                        ▷ new lighter edge for v
13:                 decrease key of v in Q to v.key
14:                 v.π ← u
15:         u.color ← finished
16:     ▷ the π pointers define the 'inverted' MST rooted at s   ◁
```

Correctness:

- initialize $S = \{\text{any node}\}$, $T = \emptyset$
- apply cut property to S (and repeat)
  - choose cheapest edge out of S
  - add it to T
  - update S (add one new explored node u to S)
- repeat

use a queue Q to organize the unexplored nodes

weight of unexplored nodes: cost of cheapest edge connecting it to a node in S

The runtime analysis of Algorithm 4.5 is exactly the same as that of Algorithm 4.4

4.6.3 Kruskal's algorithm

Algorithm 4.6 Kruskal

```
1: function KRUSKAL(G, w : E → ℝ)
2:     A ← ∅                                          ▷ initially A is empty
3:     for all v ∈ V do
4:         MAKESET(v)                                 ▷ create set containing singleton v
5:     sort E in non-decreasing order by w
6:     for all (u, v) ∈ E in non-decreasing w-order do
7:         if FIND(u) ≠ FIND(v) then                  ▷ are u and v in different trees?
8:             A ← A ∪ {(u, v)}                       ▷ join subtrees together
9:             UNION(u, v)                            ▷ merge the connected components of u and v
```

Correctness:

- consider edges in ascending order of weight
- initialize $T = \emptyset$
- Case 1: if adding e to T creates a cycle, discard e (according to cycle property)
- Case 2: insert $e = (u, v)$ in T (according to cut property). unite the connected components of u and v

T is a forest (a colleciton of trees) (some trees may be 1 node)

to detect if $e = (u, v)$ creates a cycle, we need to check the component T_u where u belongs; if $v \in T_u$, we have a cycle, else no.

to check wether $v \in T_u$, we could use e.g. DFS, which would be $O(|T_u|)$ each time we check an edge, leading to a quadratic algorithm for Kruskal's method.

when adding e to T, we merge two trees into one

Disjoint-set data structure:

- create(u): Create a set containing a single item u.
- find(u): Find the set that contains a given item u.
- union(u, v): Merge the set containing u and the set containing v into a common set

Theorem 4.8. Given n elements, each initially in its own set, the union-find data structure can perform any sequence of up to n union and find operations in total $O(n\,\alpha(n))$ time, where $\alpha(n)$ is the (extremely slowly growing) inverse Ackermann function. ◁

Union-find: $O(\alpha(n))$ *amortized time* per operation (essentially constant time)

Implementation. Use the union-find data structure.

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

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

Ties in line 5 can be broken by small perturbations to the weights or implicitly the index, i.e.,:

Algorithm 4.7 Boolean Less

```
1: function BOOLEANLESS(e_i, e_j)
2:     if w(e_i) < w(e_j) then return true
3:     else if w(e_i) > w(e_j) then return false
4:     else if i < j then return true                 ▷ break ties by index
5:     else return false
```

Example 4.1 (Clustering). Given a set U of n objects labeled $p_1, \ldots, p_n$ classify into coherent groups.

Distance function $d : U \times U \to \mathbb{R}_{\geq 0}$ satisfying:

- $d(p_i, p_j) = 0 \iff i = j$ (identity or indiscernibility)
- $d(p_i, p_j) \geq 0$ (non-negativity)
- $d(p_i, p_j) = d(p_j, p_i)$ (symmetry)

Spacing. Min distance between any pair of points in different clusters:

$$\text{spacing}(C) = \min\{\text{spacing}(C_s, C_t) \mid \text{for any two clusters in } C\} \tag{4.2}$$

where

$$\text{spacing}(C_s, C_t) = \min\{d(p_i, p_j) \mid p_i \in C_s, p_j \in C_t\}$$

Given an integer k, find a k-clustering maximizing the spacing, i.e., maximize the (min) distance between any two clusters.

can be solved using Kruskal's algorithm:

- consider a graph, nodes are objects, edges weighted by distance between objects
- run Kruskal but stop when we are left with k sets
- equivalently: find MST and remove the $k-1$ most expensive edges
- spacing: length d of the $(k-1)^{\text{th}}$ most expensive MST edge

Theorem 4.9. The MST clustering $C^* = \{C_1^*, \ldots, C_k^*\}$ has max spacing, where the spacing (i.e. the minimum distance between any two clusters) d^* is the length of the $(k-1)^{\text{th}}$ most espensive MST edge. ◁

Proof. Let $C = \{C_1, \ldots, C_k\}$ be any other k-clustering with spacing d. Let p_i, p_j be two points in two different clusters of C, say C_s and C_t, but the same cluster in C^*. Let (p, q) be the MST edge connecting C_s and C_t (on the p_i-p_j MST path). The spacing between C_s and C_t is $d \leq |(p, q)|$. But $|(p, q)| \leq d^*$ (by Kruskal). Thus, $d \leq |(p, q)| \leq d^*$. □

◀

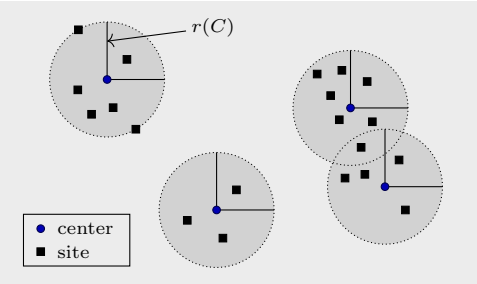
4.7 Center Selection

Problem 4.4 (k-select). select k centers $C = \{c_1, \ldots, c_k\}$ so that the maximum distance from a site its nearest center is minimized. ✍

Problem 4.5 (k-center). select k centers $C \subseteq P$ so that the maximum distance from a site its nearest center is minimized. (centers are located on the input sites) ✍

to call $\delta(\cdot, \cdot)$ a distance function, we require:

- $\delta(u, v) \geq 0$ and $\delta(u, v) = 0 \iff u = v$ (non-negativity and identity)
- $\delta(u, v) = \delta(v, u)$ (symmetry)
- $\delta(u, w) \leq \delta(u, v) + \delta(v, w)$ (triangle inequality)



Definition 4.6 (Covering radius)**.** Given a set of sites/points $P = \{s_1, \ldots, s_n\}$ and a distance function δ, for a set of centers C define:

- $\delta(s_i, C) := \min_{c \in C} \delta(s_i, c)$: distance of site s_i to nearest center in C
- $r(C) := \max_i \delta(s_i, C) = \max_i \min_{c \in C} \delta(s_i, c)$: covering radius ✍

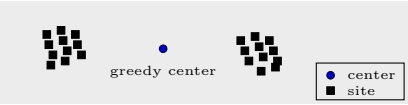
Remark 4.10. For Problem 4.4 centers can be anywhere in the space ⇒ search space infinite. For (discrete) Problem 4.5 centers must be in P ⇒ search space finite. ◀

both k-center and k-select problem is NP-hard, so we look for a greedy approximation algorithm
→ approximation to k-center also useful for the k-select problem

Naive greedy: a false start

- idea: place first center optimally for a single center
- then repeatedly add a new center to reduce $r(C)$ as much as possible

Caution 4.11. This locally optimal improvement strategy can be *arbitrarily bad*, as one can see in the following small example:



So, greedy placement by optimizing current covering radius does *not* yield a good approximation! ◀

so to make the problem more managable, let's restrict ourselves to choosing centers from the given sites (i.e. to the discrete k-center problem) and thus avoid a situation as in Caution 4.11.

Idea: use the given sites as centers, i.e., design a greedy algorithm for the k-center problem

- start with an arbitrary site as first center
- repeatedly add as next center the site that is *farthest* from all existing centers
- interpretation: always satisfy the currently most "unhappy" site

Algorithm 4.8 Greedy Approximation to k-center / k-select

Require: sites $P = \{s_1, \ldots, s_n\}$, integer k
1: $C \leftarrow \emptyset$
2: **for all** $s \in P$ **do**
3: $\quad s.\texttt{dist} \leftarrow \infty$ ▷ distance to nearest center (initially none)
4: **for** $i = 1, \ldots, k$ **do**
5: $\quad$ choose $s^* \in P$ with maximum $s^*.\texttt{dist}$ ▷ farthest site
6: $\quad C \leftarrow C \cup \{s^*\}$
7: $\quad$ **for all** $s \in P$ **do**
8: $\quad\quad s.\texttt{dist} \leftarrow \min\{s.\texttt{dist}, \delta(s, s^*)\}$
9: **return** C

Lemma 4.10. During the execution of Algorithm 4.8, the covering radius $r(C)$ is monotonically decreasing, that is $r(C^{(i)} = \{c_1, \ldots, c_{i-1}, c_i\}) \leq r(C^{(i-1)} = \{c_1, \ldots, c_{i-1}\})$. (radii of disks decreases with each added center.) ◁

Proof. As more centers are added, the distance of any point to its closest center cannot increase (it can only stay the same or decrease). Therefore, since $C^{(i-1)} \subseteq C^{(i)}$, we have $r(C^{(i)}) \leq r(C^{(i-1)})$. □

Lemma 4.11. Upon termination of Algorithm 4.8, all centers $C = \{c_1, \ldots, c_k\}$ are pairwise at least $r(C)$ apart. (no disk contains the center of any other disk.) ◁

Proof. Let $r_i = r(C^{(i)})$ be the covering radius after the i-th center c_i is added in Line 5 of Algorithm 4.8. By Lemma 4.10, the sequence $r_1 \geq \cdots \geq r_k = r(C)$ is non-increasing.

When center c_i is chosen in Line 5, it is the site farthest from C_{i-1}, thus

$$\delta(c_j, c_i) \geq r_i \quad \forall j < i$$

Since $r_i \geq r_k = r(C)$, we get $\delta(c_{j_1}, c_{j_2}) \geq r(C)$ for all $j_1, j_2 \in \{1, \ldots, k\}$ with $j_1 \neq j_2$. □

Running time: $O(k\,n)$ (in general $k \leq n$)

Lemma 4.12. Let $r_{\min} = r(C)/2$, where C is the set of centers returned by Algorithm 4.8. Then for any set C' of k centers, we have

$$r(C') \geq r_{\min} = \frac{r(C)}{2} \tag{4.3}$$

(it is not possible to cover all points of P using k disks whose radii are smaller than $r_{\min}$) ◁

Proof. Let C be the set of centers returned by Algorithm 4.8, and let $r(C)$ be its covering radius. Choose a site $s^\star \in P$ with $\delta(s^\star, C) = r(C)$. By definition of $\delta(s^\star, C)$, we have $\delta(s^\star, c) \geq r(C)$ for all $c \in C$. Moreover, by Lemma 4.11, any two centers $c_{j_1}, c_{j_2} \in C$ satisfy $\delta(c_{j_1}, c_{j_2}) \geq r(C)$. Thus, every pair of distinct points in the set $A := C \cup \{s^\star\}$ is at distance at least $r(C)$. Note that $|A| = k + 1$.

Now let C' be any set of k centers, with covering radius $r(C')$. Assign each point in A to its nearest center in C'. Since $|A| = k + 1$ but $|C'| = k$, by the pigeonhole principle there are two distinct points $x, y \in A$ assigned to the same center $c' \in C'$. Then $\delta(x, c') \leq r(C')$ and $\delta(y, c') \leq r(C')$ and by the triangle inequality,

$$\delta(x, y) \leq \delta(x, c') + \delta(c', y) \leq 2r(C')$$

On the other hand, since $x, y \in A$ and $x \neq y$, we have $\delta(x, y) \geq r(C)$. Therefore

$$r(C) \leq 2r(C')$$

which proves (4.3). □

Theorem 4.13. Let C be the set of centers returned by Algorithm 4.8 and let C^* be an optimal set of k centers. Then

$$\boxed{r(C) \leq 2r(C^*)}$$

i.e., Algorithm 4.8 is a 2-approximation for both the k-center and k-select problems. ◁

Proof. By Lemma 4.12, we have $r(C^*) \geq r_{\min} = r(C)/2$, which can be rearranged to give the desired result. □

Remark 4.12. The greedy algorithm *always* chooses centers at sites, but it is still within a factor 2 of the optimal continuous center-selection solution that may place centers anywhere in the metric space. ◀

Theorem 4.14. Unless P = NP, there is no polynomial-time approximation algorithm for the k-center or k-select problem with approximation factor $\rho < 2$. ◁

5 Dynamic Programming

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

Divide-and-conquer. Break up a problem into two sub-problems of roughly equal size; solve each sub-problem independently; combine the two sub-problem solutions to form solution to original problem.

Dynamic programming. Break up a problem into many overlapping sub-problems; combine solutions to small subproblems to build up solutions to larger and larger sub-problems.

- overlapping sub-problem = a sub-problem whose results can be reused many times
- Hint: express your optimal solution by a recursive formula

Some famous dynamic programming algorithms:

- Viterbi for Hidden Markov Models
- Unix diff for comparing two files
- Smith-Waterman for sequence alignment
- **??** for shortest path routing in networks
- Cocke-Kasami-Younger for parsing context-free grammars

Powerful technique for solving optimization problems, which have certain well-defined clean structural properties.

Definition 5.1 (Principle of optimality)**.** For the global problem to be solved optimally, each subproblem must be solved optimally. ✍

→ this principle must hold to apply DP

The global optimal solution consists of optimal solutions to subproblems.

Polynomial time DP:

- Polynomially-many distinct overlapping subproblems (polynomial in input size).
- DP does not necessarily lead to a polynomial time algorithm

DP selection principle: When given a set of feasible options to choose from, try them all and take the best

5.1 Weighted Interval Scheduling

We denote by $p(j)$ the largest index i with $i < j$ and such that job i is compatible with j, i.e.

$$p(j) := \begin{cases} \max\{i \mid f_i \leq s_j\} & \exists i \text{ with } f_i \leq s_j \\ 0 & \text{otherwise} \end{cases} \tag{5.1}$$

Algorithm 5.1 Compute $p(j)$ values

Require: Jobs $1, \ldots, n$ sorted according to their finish time $f_1 \leq \cdots \leq f_n$

```
Sort all starting and finish times in a single array A
index ← 0
for all e ∈ A do                ▷ go through all times ('events') in ascending order
    if e = f_i then
        index ← i
    else if e = s_j then
        p[j] ← index
```

The algorithm goes through every time slot, and it remembers the index of the last finish time. When we encounter a starting time s_j, its p value is the largest index i such that $f_i \le s_j$, which is exactly the value of `index` in the algorithm. Hence, Algorithm 5.1 is correct.

Sorting takes $O(n \log n)$ time, and it is easy to see that the for loop takes $O(n)$ time.

Assume jobs are labeled so that $f_1 \le \cdots \le f_n$. Let v_j be the value/weight of job j, and let $\mathrm{OPT}(j)$ denote the maximum total weight of a subset of mutually compatible jobs among $1, \ldots, j$. We use the convention $\mathrm{OPT}(0) = 0$.

Recurrence. For $j \ge 1$ we have

$$\mathrm{OPT}(j) = \max\bigl\{v_j + \mathrm{OPT}(p(j)),\ \mathrm{OPT}(j-1)\bigr\}$$

- Either the optimal solution does *not* take job j: then its value is $\mathrm{OPT}(j-1)$.
- Or the optimal solution *does* take job j: then we gain v_j plus the best compatible subset among jobs $\{1, \ldots, p(j)\}$, which has value $\mathrm{OPT}(p(j))$.

Algorithm 5.2 Weighted Interval Scheduling

Require: jobs $1, \ldots, n$ sorted by finish time, values v_j, and precomputed $p(j)$ as in (5.1)

```
OPT[0] ← 0
for j = 1, ..., n do
    OPT[j] ← max{ v_j + OPT[p[j]], OPT[j − 1] }
return OPT[n]
```

Running time. Computing the $\mathrm{OPT}[j]$ table takes $O(n)$ time once the jobs are sorted and the $p(j)$ values are known. Together with sorting and the $p(j)$-computation, the total running time is $O(n \log n)$ and the space usage is $O(n)$.

5.2 Segmented Least Squares

Given n points $(x_1, y_1), \ldots, (x_n, y_n)$ in the plane (assume $x_1 < \cdots < x_n$), we want to approximate them by a small number of straight-line segments.

- For any $1 \le i \le j \le n$, consider fitting a single line to the points $(x_i, y_i), \ldots, (x_j, y_j)$ by least squares.
- Let $e(i, j)$ be the sum of squared vertical errors of this best-fit line:

$$e(i,j) = \sum_{k=i}^{j} \bigl(y_k - (\hat{a}_{ij} x_k + \hat{b}_{ij})\bigr)^2,$$

  where $\hat{a}_{ij}, \hat{b}_{ij}$ are the least-squares coefficients.
- Each segment we use incurs a fixed penalty $C > 0$ (to discourage using too many segments).

Goal. Partition the sequence of points into contiguous blocks $[1, \ell_1], [\ell_1 + 1, \ell_2], \ldots, [\ell_{m-1} + 1, n]$ and fit each block with its own least-squares line so as to minimize

$$\sum_{r=1}^{m} e(\ell_{r-1} + 1, \ell_r) + mC \quad (\ell_0 := 0)$$

Precomputation. Using prefix sums of $x_k, y_k, x_k^2, x_k y_k$, all $e(i, j)$ values can be computed in total $O(n^2)$ time.

Define

$$\mathrm{OPT}(j) := \text{minimum total cost for approximating points } (x_1, y_1), \ldots, (x_j, y_j)$$

using any number of segments, and set $\mathrm{OPT}(0) := 0$.

Recurrence. For $j \ge 1$,

$$\mathrm{OPT}(j) = \min_{1 \le i \le j} \bigl\{\mathrm{OPT}(i-1) + C + e(i, j)\bigr\}.$$

Algorithm 5.3 Segmented Least Squares via DP

Require: points $(x_1, y_1), \ldots, (x_n, y_n)$ with $x_1 < \cdots < x_n$, segment penalty C

```
precompute all e(i, j) for 1 ≤ i ≤ j ≤ n
OPT[0] ← 0
for j = 1, ..., n do
    OPT[j] ← ∞
    for i = 1, ..., j do
        OPT[j] ← min{OPT[j], OPT[i − 1] + C + e(i, j)}
return OPT[n]
```

- The last segment in an optimal solution for the first j points must cover some suffix $i, \ldots, j$.
- Its cost is $e(i, j)$ plus the penalty C, plus the optimal cost for the prefix $1, \ldots, i-1$.

Running time. Precomputing all errors $e(i, j)$ takes $O(n^2)$ time. The DP itself also takes $O(n^2)$ time and $O(n)$ space for the OPT array. Backtracking through the choices of i at each j yields the optimal segmentation.

5.3 Knapsack Problem

Problem 5.2 (0/1 Knapsack Problem)**.** Given n items, each with a weight $w_i \in \mathbb{N}$ and a value $v_i \in \mathbb{N}$, and a "knapsack" with capacity $W \in \mathbb{N}$, fill the knapsack (respecting its capacity) to maximize the total value. ✍

0/1 refers to the fact that the items are not divisible, i.e. each item can either be included (1) or excluded (0) from the knapsack, but not fractionally included (otherwise it would be the fractional knapsack problem, which can be solved greedily by taking items in descending order of value-to-weight ratio).

recursive formulation:

$$\text{opt}(i, w) = \begin{cases} 0 & i = 0 \quad \triangleright \text{ no items left} \\ \text{opt}(i-1, w) & w_i > w \quad \triangleright \text{ item too heavy} \\ \max\{v_i + \text{opt}(i-1, w-w_i),\ \text{opt}(i-1, w)\} & \text{otherwise} \quad \triangleright \text{ include or exclude item } i\text{?} \end{cases}$$

top down memoized:

Algorithm 5.4 Knapsack (top-down, memoized)

```
function OPT(i, w)
    if M[i, w] = NIL then
        if i = 0 then
            M[i, w] ← 0                                   ▷ no items left
        else if w_i > w then
            M[i, w] ← OPT(i − 1, w)                       ▷ item too heavy
        else
            M[i, w] ← max{v_i + OPT(i − 1, w − w_i), OPT(i − 1, w)}   ▷ include or exclude?
    return M[i, w]
```

Bottom-up (tabulation) version. Instead of recursion + memoization, we can fill a DP table iteratively. Let W be the knapsack capacity. We build a table OPT$[i, w]$ for $i = 0, \ldots, n$ and

$w = 0, \dots, W$, where $\mathrm{OPT}[i, w]$ denotes the maximum value achievable using items $1, \dots, i$ with capacity w.

Algorithm 5.5 Knapsack (bottom-up DP)

Require: items $1, \dots, n$ with weights w_i, values v_i; capacity W

for $w = 0, \dots, W$ **do**
  $\mathrm{OPT}[0, w] \leftarrow 0$ ▷ no items
for $i = 1, \dots, n$ **do**
  for $w = 0, \dots, W$ **do**
    if $w_i > w$ **then**
      $\mathrm{OPT}[i, w] \leftarrow \mathrm{OPT}[i-1, w]$ ▷ item i too heavy
    else
      $\mathrm{OPT}[i, w] \leftarrow \max\{v_i + \mathrm{OPT}[i-1, w-w_i],\ \mathrm{OPT}[i-1, w]\}$
return $\mathrm{OPT}[n, W]$

Running time. The table has $(n+1)(W+1) = O(nW)$ entries, and each is filled in $O(1)$ time, so the running time is $O(nW)$ and the space is $O(nW)$. By keeping only two rows at a time, the space can be reduced to $O(W)$.

Remark 5.1. 0/1 Knapsack is NP-hard in general, so we do not expect a polynomial-time algorithm in the input size. The DP runs in time polynomial in n and W, which is called *pseudo-polynomial* time, because W is exponential in the number of bits needed to represent it (i.e. the input size). ◀

Approximation algorithm. There exists a polynomial algorithm that produces a feasible solution that has value within 0.01% of the optimum.

5.4 Sequence Alignment

Goal. Given two strings $X = x_1 x_2 \cdots x_m$ and $Y = y_1 y_2 \cdots y_n$, find an alignment of minimum total cost.

Definition 5.3 (Alignment)**.** An alignment M between two strings X and Y is a set of ordered pairs (x_i, y_j) such that each item occurs in at most one pair and no two pairs cross. ✍

Definition 5.4. Two pairs (x_i, y_j) and $(x_{i'}, y_{j'})$ cross if $i < i'$ but $j > j'$. ✍

Definition 5.5 (Cost Model)**.** Aligning x_i with y_j incurs a *mismatch* penalty $\alpha_{x_i y_j}$. Leaving a character unmatched (aligning it with a gap "–") incurs a *gap* penalty $\delta > 0$. ✍

Definition 5.6 (Cost)**.** The cost of an alignment M is

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

i.e. it is the sum of all gap and mismatch penalties. ✍

Definition 5.7 (Edit distance)**.** The edit distance between strings X and Y is the minimum Cost over all Alignments ✍

DP formulation. Let $\mathrm{OPT}(i, j)$ denote the minimum cost of aligning the prefixes $x_1 \cdots x_i$ and $y_1 \cdots y_j$. If either of the prefixes is empty and the other has length k, the only option is to align all characters of the other prefix with gaps, incurring a cost of δ per gap, i.e. $\mathrm{OPT}(k, 0) = \mathrm{OPT}(0, k) = k \cdot \delta$. Otherwise we have three choices:

- Case 1: match x_i with y_j: $\alpha_{x_i y_j} + \mathrm{OPT}(i-1, j-1)$
  - pay mismatch cost $\alpha_{x_i y_j}$ plus whatever the minimum cost is for aligning the prefixes $x_1 \cdots x_{i-1}$ and $y_1 \cdots y_{j-1}$
- Case 2a: leave x_i unmatched (align with gap): $\delta + \mathrm{OPT}(i-1, j)$

- pay gap cost δ for x_i plus min cost for aligning $x_1 \cdots x_{i-1}$ and $y_1 \cdots y_j$

- Case 2b: leave y_j unmatched (align with gap): $\delta + \mathrm{OPT}(i, j-1)$
  - pay gap cost δ for y_j plus min cost for aligning $x_1 \cdots x_i$ and $y_1 \cdots y_{j-1}$

We get the recurrence:

$$\mathrm{OPT}(i,j) = \begin{cases} j \cdot \delta & \text{if } i = 0 \\ i \cdot \delta & \text{if } j = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + \mathrm{OPT}(i-1, j-1) \\ \delta + \mathrm{OPT}(i-1, j) \\ \delta + \mathrm{OPT}(i, j-1) \end{cases} & \text{otherwise} \end{cases} \tag{5.2}$$

To compute $\mathrm{OPT}(m, n)$:

- build up the values of $\mathrm{OPT}(i, j)$ using the recurrence – fill an $m \times n$ table
- $O(mn)$ subproblems
- to compute $\mathrm{OPT}(i, j)$ we need: $\mathrm{OPT}(i-1, j)$, $\mathrm{OPT}(i-1, j-1)$, $\mathrm{OPT}(i, j-1)$
- can fill table *row by row* or *column by column*
- cost we are looking for will be in bottom-right cell $\mathrm{OPT}(m, n)$

Algorithm 5.6 Sequence Alignment

Require: strings $X = x_1 \cdots x_m$, $Y = y_1 \cdots y_n$; gap penalty δ; mismatch costs α_{pq}

```
1: for i = 0, ..., m do
2:     M[i, 0] ← iδ
3: for j = 0, ..., n do
4:     M[0, j] ← jδ
5: for i = 1, ..., m do
6:     for j = 1, ..., n do
7:         M[i, j] ← min{α_{x_i y_j} + M[i−1, j−1], δ + M[i−1, j], δ + M[i, j−1]}
8: return M[m, n]
```

Running time. The table has $(m+1)(n+1) = O(mn)$ entries and each entry is computed in $O(1)$, so the running time is $O(mn)$ and the space usage is $O(mn)$.

Remark 5.2. We can reduce the space to $O(\min\{m, n\})$ if only the optimal cost is needed by keeping only two rows/columns in memory at a time while "filling the table". Because to compute $M[i, j]$ we only need $M[i-1, j-1]$, $M[i-1, j]$, and $M[i, j-1]$. ◀

Recovering an optimal alignment. Two options:

- Do a postprocessing step after filling the table: Starting at (m, n), backtrack to $(0, 0)$ by comparing $M[i, j]$ with the three candidate predecessors: $(i-1, j-1)$, $(i-1, j)$, $(i, j-1)$. Record the corresponding aligned characters (x_i vs. y_j, x_i vs. "–", or "–" vs. y_j). Ties correspond to multiple optimal alignments.
- Store another array (hint table): Every time we make a decision in Line 7 of Algorithm 5.6, keep a hint of where we came from.

5.5 Longest Common Subsequence

Let us think of character strings as sequences of characters. Given $X = \langle x_1, \ldots, x_l \rangle$, we say that $Z = \langle z_1, \ldots, z_k \rangle$ is a subsequence of X if there is a strictly increasing sequence of k indices $\langle i_1, \ldots, i_k \rangle$ such that $Z = \langle x_{i_1}, \ldots, x_{i_k} \rangle$.

Given two strings $X = \langle x_1, \ldots, x_m \rangle$ and $Y = \langle y_1, \ldots, y_n \rangle$, the longest common subsequence of X and Y is a longest sequence Z that is a subsequence of both X and Y.

5.5.1 Brute force

A brute force approach would be to enumerate all subsequences of X and check for each whether it is also a subsequence of Y. There are 2^m subsequences of X and checking wether it is a subsequence of Y takes $O(n)$ time: Scan Y for first occurence, from there scan for second, and so on. In total this takes $\Theta(n2^m)$ time.

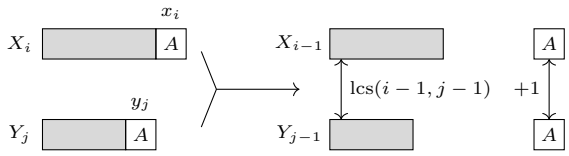
5.5.2 Dynamic programming formulation

A prefix X_i of a sequence X is just an initial string of values, $X_i = \langle x_1, \ldots, x_i \rangle$. X_0 is the empty sequence. The idea is to compute the longest common subsequence for every possible pair of prefixes (there are $O(mn)$ such pairs).

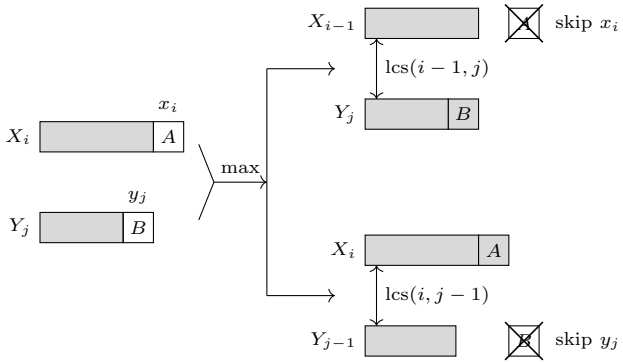
Let $\text{lcs}(i, j)$ denote the length of the longest common subsequence of X_i and Y_j. We have the following recursive relation:

$$\text{lcs}(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ \text{lcs}(i-1, j-1) + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max(\text{lcs}(i-1, j), \text{lcs}(i, j-1)) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

Last characters match:



Last characters do not match:



DP table. We store values in a table $L[0 \ldots m, 0 \ldots n]$ where $L[i, j] = \text{lcs}(i, j)$. We can go recursive/top-down with memoization (Algorithm 5.7) or iterative/bottom-up (Algorithm 5.8).

Algorithm 5.7 Longest Common Subsequence (top-down DP)

```
1: function LCS(i, j)
2:     if L[i, j] = NIL then
3:         if i = 0 ∨ j = 0 then                          ▷ base case
4:             L[i, j] ← 0
5:         else if x_i = y_j then                         ▷ last characters match
6:             L[i, j] ← LCS(i − 1, j − 1) + 1
7:         else                                           ▷ last characters do not match
8:             L[i, j] ← max{LCS(i − 1, j), LCS(i, j − 1)}
9:     return L[i, j]                                     ▷ return stored value
```

Running time of Algorithm 5.8. The table has $(m + 1)(n + 1) = O(mn)$ entries, each filled in $O(1)$ time, so the running time is $O(mn)$ and the space is $O(mn)$.

Algorithm 5.8 Longest Common Subsequence (bottom-up DP)

Require: strings $X = x_1 \cdots x_m$, $Y = y_1 \cdots y_n$

```
L ← new array [0...m, 0...n]
for i = 0, ..., m do
    L[i, 0] ← 0
for j = 0, ..., n do
    L[0, j] ← 0
for i = 1, ..., m do                                  ▷ iterate over rows
    for j = 1, ..., n do                              ▷ iterate over columns
        if x_i = y_j then
            L[i, j] ← L[i − 1, j − 1] + 1             ▷ take x_i = y_j for LCS
        else
            L[i, j] ← max{L[i − 1, j], L[i, j − 1]}
return L[m, n]
```

5.5.3 Reconstructing the actual subsequence

To retrieve the actual subsequence, we can use another table, a "hint" table $h[0 \ldots m, 0 \ldots n]$ that records which of the three cases was used to compute each $L[i, j]$ (Algorithm 5.9).

Algorithm 5.9 LCS (bottom-up) with hints

Require: strings $X = x_1 \cdots x_m$, $Y = y_1 \cdots y_n$

```
L ← new array [0...m, 0...n]                          ▷ stores lcs lengths
h ← new array [0...m, 0...n]                          ▷ hint table
for i = 0, ..., m do                                  ▷ initialize column 0
    L[i, 0] ← 0
    h[i, 0] ← skipX
for j = 0, ..., n do                                  ▷ initialize row 0
    L[0, j] ← 0
    h[0, j] ← skipY
for i = 1, ..., m do
    for j = 1, ..., n do
        if x_i = y_j then
            L[i, j] ← L[i − 1, j − 1] + 1
            h[i, j] ← addXY
        else
            if L[i − 1, j] ≥ L[i, j − 1] then
                L[i, j] ← L[i − 1, j]
                h[i, j] ← skipX
            else
                L[i, j] ← L[i, j − 1]
                h[i, j] ← skipY
return L[m, n], h
```

Now we can use the hints to reconstruct the LCS:

- start at the last entry of the table, $[m, n]$
- if $h[i, j] = \text{addXY}$ ($\nwarrow$), then $x_i = y_j$ is appended to LCS, continue with $[i - 1, j - 1]$
- if $h[i, j] = \text{skipX}$ ($\uparrow$), then x_i is not in LCS, continue with $[i - 1, j]$
- if $h[i, j] = \text{skipY}$ ($\leftarrow$), then y_j is not in LCS, continue with $[i, j - 1]$
- stop when reaching $[0, 0]$

Because the characters are generated in revers order, prepend them to a sequence, so that when we are done, the sequence is in correct order. Or use recursion to print them in correct order directly (Algorithm 5.10).

Alternatively, the hints can be discovered based on the filled L table (Algorithm 5.11, Algorithm 5.12). If $x_i = y_j$ and $L[i, j] = L[i - 1, j - 1] + 1$, then x_i is part of some LCS and we move to $[i - 1, j - 1]$; otherwise, if $L[i - 1, j] \geq L[i, j - 1]$ move to $[i - 1, j]$, else move to $[i, j - 1]$.

Algorithm 5.10 recursive LCS printing using hints

```
1: function PRINT-LCS(h, X, i, j)
2:     if i = 0 ∨ j = 0 then
3:         return
4:     if h[i, j] = addXY then
5:         PRINT-LCS(h, X, i − 1, j − 1)
6:         print x_i
7:     else if h[i, j] = skipX then
8:         PRINT-LCS(h, X, i − 1, j)
9:     else
10:        PRINT-LCS(h, X, i, j − 1)
```

Algorithm 5.11 recursive LCS printing discovering hints

```
1: function PRINT-LCS(L, X, Y, i, j)
2:     if i = 0 ∨ j = 0 then
3:         return
4:     if x_i = y_j ∧ L[i, j] = L[i − 1, j − 1] + 1 then
5:         PRINT-LCS(L, X, Y, i − 1, j − 1)          ▷ take x_i = y_j
6:         print x_i
7:     else if L[i − 1, j] ≥ L[i, j − 1] then
8:         PRINT-LCS(L, X, Y, i − 1, j)              ▷ skip x_i
9:     else
10:        PRINT-LCS(L, X, Y, i, j − 1)              ▷ skip y_j
```

Algorithm 5.12 recursive LCS printing discovering hints (alternative condition)

```
1: function PRINT-LCS(L, X, i, j)
2:     if i = 0 ∨ j = 0 then
3:         return
4:     if L[i, j] = L[i − 1, j] then
5:         PRINT-LCS(L, X, i − 1, j)                 ▷ move up i.e. skip x_i
6:     else if L[i, j] = L[i, j − 1] then
7:         PRINT-LCS(L, X, i, j − 1)                 ▷ move left i.e. skip y_j
8:     else                  ▷ L[i, j] must be different from all 3 neighbors, so x_i = y_j
9:         PRINT-LCS(L, X, i − 1, j − 1)             ▷ move diagonally, take x_i = y_j
10:        print x_i
```

Algorithm 5.10, Algorithm 5.11 and Algorithm 5.12 are called with initial parameters $i = m$ and $j = n$.

Remark 5.3. Again, we can reduce the space to $O(\min\{m, n\})$ if only the length of the LCS is needed by keeping only two rows/columns in memory at a time while "filling the table". Because to compute $L[i, j]$ we only need $L[i-1, j-1]$, $L[i-1, j]$, and $L[i, j-1]$. ◀

5.6 Sequence Alignment in linear space

There is a cool trick that allows linear space without compromising the time complexity asymptotically. It combines Dynamic Programming with Divide and Conquer. The same trick can be used for Longest Common Subsequence.

Can we avoid using quadratic space?

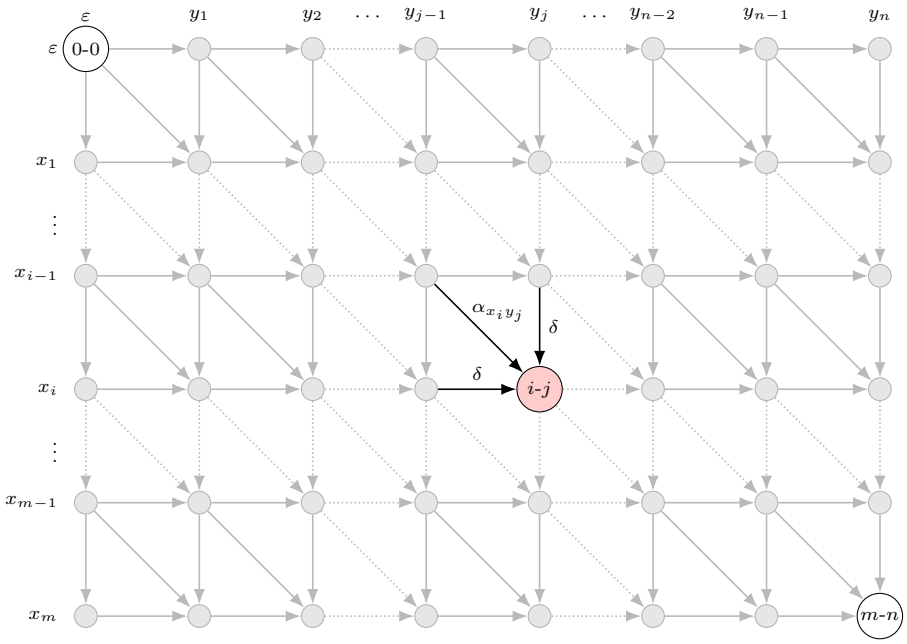
easy: optimal value: $O(m + n)$ space and $O(mn)$ time.

- use a $2 \times n$ matrix. compute $\mathrm{OPT}(i, \cdot)$ from $\mathrm{OPT}(i - 1, \cdot)$
- but no longer a simple way to recover the alignment itself

Theorem 5.1 (Hirschberg 1975)**.** Optimal alignment in $O(m+n)$ space and $O(mn)$ time. Clever combination of divide-and-conquer and dynamic programming. ◁

A DP matrix can be seen as a directed acyclic graph (DAG). The optimal alignment corresponds to the shortest path in the DAG from node $(0,0)$ to (m,n).

Edit distance graph: Add a node for every entry $M[i,j]$ and an edge from $M[i-1,j-1]$, $M[i-1,j]$, $M[i,j-1]$ to $M[i,j]$ with weights according the Cost Model:



Let $f(i,j)$ be the (length of) shortest path from $(0,0)$ to (i,j) and $g(i,j)$ be the (length of) shortest path from (i,j) to (m,n).

Observation 5.4. $f(i,j) = M[i,j] = \mathrm{OPT}(i,j)$. ◀

Observation 5.5. Let P be the shortest path from $(0,0)$ to (m,n). Then

$$\mathrm{OPT}(m,n) = f(i,j) + g(i,j) \tag{5.3}$$

for any node (i,j) that lies on the path P. ◀

Observation 5.6 (Computing $f(\cdot,j)$)**.** Given a column j, we can compute the value of $f(\cdot,j) = \mathrm{OPT}[\cdot,j]$ in $O(mn)$ time and $O(m+n)$ space.

- run space efficient sequence alignment to compute $\mathrm{OPT}(m,n)$
- keep the entries of column j in a separate array ◀

We can compute $g(i,j)$ by reversing the edge orientations (formula indices) and inverting the roles of $(0,0)$ and (m,n).

- we start with $g(m,n) = 0$ and reverse indices in the recursive formula (5.2) for $\mathrm{OPT}(i,j)$

The backward function $g(i,j)$:

- $g(m,n) = 0$
- $g(m,j) = (n-j)\cdot\delta$; $g(i,n) = (m-i)\cdot\delta$
- substitute $i-1$ by $i+1$ and $j-1$ by $j+1$ in (5.2) for $\mathrm{OPT}(i,j)$

Observation 5.7 (Computing $g(\cdot,j)$)**.** Given a column j, we can compute the values of $g(\cdot,j)$ in $O(mn)$ time and $O(m+n)$ space.

- run space efficient sequence alignment to compute $\mathrm{OPT}(0,0)$, (reversing the role of nodes (m,n) and $(0,0)$ and reversing indices)
- keep the entries of column j in a separate array ◀

Given a column j, compute $f(\cdot, j)+g(\cdot, j)$ in a separate array for all i (i.e. the entire column). Let q be the index of the minimum

$$q \coloneqq \arg\min_{i\in[1\dots m]}\big(f(i,j)+g(i,j)\big) \tag{5.4}$$

then

$$f(q,j)+g(q,j)=\mathrm{OPT}(m,n) \tag{5.5}$$

i.e. we have found one pair x_q-y_j (alignment x_q-y_j / x_q-gap / y_j-gap) in $O(mn)$ time and $O(m+n)$ space!

Now we just need to put everything together using divide and conquer principle:

- Divide & Conquer: choose column $j = n/2$
- Compute $f(\cdot, n/2)$. Compute $g(\cdot, n/2)$. Compute $f(\cdot, n/2)+g(\cdot, n/2)$
- Let q be an index that minimizes $f(i, n/2)+g(i, n/2)$
- Store the pair x_q-$y_{n/2}$ (possible alignment x_q-$y_{n/2}$ / x_q-gap / $y_{n/2}$-gap)
- Recurse for $X[1\dots q]$, $Y[1\dots n/2]$ (top left subproblem) and $X[q\dots m]$, $Y[n/2\dots n]$ (bottom right subproblem)

Divide: Compute $f(i, n/2)$ and $g(i, n/2)$, for all i, using two space-efficient DP algorithms (Observation 5.6 and Observation 5.7) in $O(mn)$ time and $O(m+n)$ space.

- find the index q that minimizes $f(i, n/2)+g(i, n/2)$
- store pair x_q-$y_{n/2}$ (align x_q-$y_{n/2}$ or x_q-gap or $y_{n/2}$-gap)

Conquer: recursively compute optimal alignment in top left and bottom right pieces.

Algorithm 5.13 Divide-and-Conquer-Alignment (linear space)

```
1:  function DIVIDE-AND-CONQUER-ALIGNMENT(X, Y)
2:      m ← |X|                                          ▷ number of symbols in X
3:      n ← |Y|                                          ▷ number of symbols in Y
4:      if m ≤ 2 ∨ n ≤ 2 then
5:          compute optimal alignment using ALIGNMENT(X, Y)
6:      FORWARD-SPACE-EFFICIENT-ALIGNMENT(X, Y[1 : n/2])        ▷ array f(·, n/2)
7:      BACKWARD-SPACE-EFFICIENT-ALIGNMENT(X, Y[n/2 : n])       ▷ array g(·, n/2)
8:      q ← arg min_{i∈[1...m]} (f(i, n/2) + g(i, n/2))         ▷ applying (5.4)
9:      add pair (q, n/2) to global list P
10:     DIVIDE-AND-CONQUER-ALIGNMENT(X[1 : q], Y[1 : n/2])
11:     DIVIDE-AND-CONQUER-ALIGNMENT(X[q : m], Y[n/2 : n])
12:     return P
```

What is the running time of Algorithm 5.13? For sure it takes at least as long as Algorithm 5.6 because we need to compute $f(\cdot, n/2)$ and $g(\cdot, n/2)$ at each level of recursion.

Let $T(m, n)$ be the maximum running time of Algorithm 5.13 on strings X and Y of length m and n. Then

$$T(m,n)=\underbrace{T(q,n/2)}_{\text{Line 10}}+\underbrace{T(m-q,n/2)}_{\text{Line 11}}+\underbrace{O(mn)}_{\text{Lines 6,7}}$$

for some $q \in [1\dots m]$ (see Equation 5.4). We can also upper bound $T(m, n)$ as

$$T(m,n)\le 2T(m,n/2)+O(mn) \overset{(1.1)}{\Longrightarrow} T(m,n)=O(mn\log n)$$

but this analysis is not tight, actually $T(m, n) = O(mn)$. We increase time by only a constant factor. But we reuse space during recursive calls.

Theorem 5.2. $T(m, n) = O(mn)$. ◁

Proof (Induction on n). Choose a constant $c > 0$ such that

$$T(m, 2) \leq cm \tag{5.6}$$
$$T(2, n) \leq cn \tag{5.7}$$
$$T(m, n) \leq cmn + T(q, n/2) + T(m - q, n/2) \tag{5.8}$$

for all $m, n \geq 1$ and all $q \in [1 \dots m]$.

We prove by induction on n the statement

$$P(n): \quad T(m, n) \leq 2cmn \quad \forall m \geq 1$$

(i) **Base cases:** If $m = 2$ or $n = 2$, then by (5.6) and (5.7) we have

$$T(m, n) \leq \max\{cm, cn\} \leq 2cmn \quad ✔$$

(ii) **Induction hypothesis:** Assume $P(\tilde{n})$ holds for all $\tilde{n} < n$, i.e.

$$T(\tilde{m}, \tilde{n}) \leq 2c\tilde{m}\tilde{n}$$

for all $\tilde{m} \geq 1$ and all $\tilde{n} < n$.

(iii) **Induction step:** Fix any $m \geq 1$. Then

$$\begin{aligned} T(m, n) &\overset{(5.8)}{\leq} cmn + T(q, n/2) + T(m - q, n/2) \\ &\overset{\text{(ii)}}{\leq} cmn + 2c \cdot q \cdot \frac{n}{2} + 2c \cdot (m - q) \cdot \frac{n}{2} \\ &= cmn + cqn + c(m - q)n \\ &= cmn + cmn \\ &= 2cmn \quad ✔ \end{aligned}$$

Since m was arbitrary, this proves $P(n)$.

Hence $T(m, n) = O(mn)$. □

5.7 Shortest Paths with Negative Weights

For graphs with possibly negative edge weights (but no negative-weight cycles reachable from the source), Dijkstra's greedy algorithm no longer works.

Caution 5.8. It may be tempting to use Dijkstra with negative weights by adding a constant to all edge weights to make them non-negative. However, this approach does not preserve shortest paths (the new shortest paths may differ from the original ones). ◀

Observation 5.9. If some path from s to t contains a negative cost cycle, then there does not exist a well-defined shortest s-t path (cost can be made arbitrarily small by traversing the negative cycle multiple times). If there is no negative cost cycle, the shortest path is simple (no cycle) and has at most $n - 1$ edges (otherwise some vertex would be repeated, forming a cycle that could be removed to yield a shorter path). ◀

Recursively formulate OPT$(v, ?)$: length of shortest v-t path.

- Think of a parameter to facilitate the recursive formulation of a shortest path.
- Idea: use number of edges i.

Notation 5.8. OPT$(i, v) :=$ length of shortest v-t path P using *at most* i edges. ✍

compute OPT$(n - 1, v)$ for every node v.

Goal: OPT$(n - 1, s)$

We need to distinguish the following cases:

- Case 1: P uses $< i$ $(\leq i - 1)$ edges: OPT$(i, v) =$ OPT$(i - 1, v)$

- Case 2: P uses exactly i edges.
  - Let (v, w) be the first edge out of v. OPT uses (v, w) and selects best w-t path using at most $i - 1$ edges: $\text{OPT}(i - 1, w)$.
  - Choose minimum among all possibilities of edges incident to v.
- Base case ($i = 0$): $\text{OPT}(0, v) = \{\infty \text{ if } v \neq t; 0 \text{ if } v = t\}$

We thus have the following recursive formula:

$$\text{OPT}(i, v) = \begin{cases} 0 \text{ if } v = t;\ \infty \text{ if } v \neq t & \text{if } i = 0 \\ \min\left\{\text{OPT}(i-1, v),\ \min\limits_{(v,w)\in E}\{\text{OPT}(i-1, w) + c_{vw}\}\right\} & \text{otherwise} \end{cases} \tag{5.9}$$

Assuming there is no negative cycle from v to t, then $\text{OPT}(n - 1, v)$ is the length of the shortest v-t path (by Observation 5.9)

Thus, $\text{OPT}(n - 1, s)$ is the length of the shortest path from s to t (assuming there is no negative cycle from s to t).

Algorithm 5.14 Shortest Path (1st attempt)

```
1: function SHORTEST-PATH(G, t)
2:     for all v ∈ V do
3:         M[0, v] ← ∞
4:     M[0, t] ← 0
5:     for i = 1, ..., n − 1 do
6:         for all v ∈ V do
7:             M[i, v] ← M[i − 1, v]
8:             for all (v, w) ∈ E do                ▷ use adjacency list of node v
9:                 M[i, v] ← min{M[i, v], M[i − 1, w] + c_vw}
```

Analysis of Algorithm 5.14:

- Time: $\Theta(mn)$, since $\sum_{v\in V} \deg^+(v) = m$
- Space: $\Theta(n^2)$ for the $n \times n$ table M

We can improve the space usage by keeping only two rows of of the table M and different from Sequence Alignment, we do not need to come up with a different algorithm to reconstruct the shortest paths.

We only need to maintain the latest "successor" for each node. No need for a full matrix of hints. Just additional $\Theta(n)$ space for a successor array.

So we use a $2 \times n$ table plus a n-array for the successors. After i iterations, $M[2, v]$ is length of shortest v-t path using $\leq i$ edges and we can use the successor array to retrieve the path.

5.7.1 Practical improvement: one array only

We can do even better. Maintain only one *single* n-array $M[1 \ldots n]$ and update it in place.

In practice, this not only reduces space but potentially also time, because we do not insist to at most use i edges, but allow the best we can so far using any number of edges.

Observation 5.10. Throughout the algorithm, each value $M[v]$ is the length of *some* v-t path (or ∞). After i rounds of updates, we have $M[v] \leq \text{OPT}(i, v)$ for all v. But it could be much smaller because a sequence of updates may discover a very good path that uses *more than* i edges (this is why the one-array version is a "practical improvement" rather than a faithful DP table). ◀

So with this approach we only need two n-arrays, one for the values and one for the successors. In the worst case we still have the same time complexity, but usually it converges faster doing this trick.

5.7.2 Practical improvement: early termination

Observation 5.11. No need to check edges of the form (v, w) unless $M[w]$ changed in the previous iteration. ◀

Observation 5.12. Once no change occurs at an iteration i, the algorithm can stop:

- no change at iteration i means: $\mathrm{OPT}(i, v) = \mathrm{OPT}(i-1, v)$ for all v.
- the values of $\mathrm{OPT}(i+1, v)$ are computed using $\mathrm{OPT}(i, v)$.
- then $\mathrm{OPT}(i+1, v) = \mathrm{OPT}(i, v) = \mathrm{OPT}(i-1, v)$, for all v.
- thus, no change at iteration $i+1$.
- by induction, no change for any $j > i$. ◀

Algorithm 5.15 Bellman-Ford

```
1:  function PUSH-BASED-SHORTEST-PATH(G, t)
2:      for all v ∈ V do
3:          M[v] ← ∞                                          ▷ distance array
4:          succ[v] ← NIL                                     ▷ successor array
5:      M[t] ← 0
6:      for i = 1, ..., n − 1 do
7:          for all w ∈ V do
8:              if M[w] has been updated in previous iteration then   ▷ Observation 5.11
9:                  for all (v, w) ∈ E do                     ▷ incoming edges into w
10:                     if M[w] + c_vw < M[v] then
11:                         M[v] ← M[w] + c_vw
12:                         succ[v] ← w
13:         if no M[v] changed in this iteration then         ▷ Observation 5.12
14:             return
```

Algorithm 5.15 implements the Bellman-Ford algorithm with the practical improvements discussed above.

- Memory: $O(m+n)$.
- Running time: $O(mn)$ worst case, but substantially faster in practice: once no change on M, the iteration stops.
- The path whose length is $M[v]$ after i iterations, can have $\gg i$ edges.

5.7.3 Standard implementation

Algorithm 5.16 shows the standard variant of the Bellman-Ford algorithm. It starts from the source s, which is why it uses $pred[u]$ (predecessor) pointers instead of successor pointers.

Algorithm 5.16 Bellman-Ford: Alternative Implementation

```
1:  function BELLMAN-FORD(G, s)
2:      for all u ∈ V do
3:          d[u] ← ∞                                  ▷ shortest path (so far) from s to u
4:          pred[u] ← NIL                             ▷ predecessor of u on this path
5:      d[s] ← 0
6:      repeat
7:          converged ← true
8:          for all (u, v) ∈ E do                     ▷ relax along each edge
9:              if d[u] + w(u, v) < d[v] then
10:                 d[v] ← d[u] + w(u, v)
11:                 pred[v] ← u
12:                 converged ← false
13:     until converged
14:     ▷ The pred pointers define an inverted shortest path tree   ◁
```

After $\leq n-1$ iterations of Algorithm 5.16, the d-values of all vertices will have their final values. Time: $O(mn)$.

5.7.4 Comparison Bellman-Ford and Dijkstra

Even though Dijkstra's algorithm theoretically has better time complexity, it is centralized and you need to make choices based on the entire graph. So if a graph is huge, it is difficult to implement Dijkstra.

Bellman-Ford has some advantages: Dijkstra requires global information of network, whereas Bellman-Ford uses only *local* knowledge of neighboring nodes. More flexible and *decentralized* than Dijkstra's.

So Bellman-Ford is used much more often in practice, e.g. for the design of distributed routing algorithms to determine the most efficient path in a communication network. If the graph is reasonable and there are no negative weights, Dijkstra should be used.

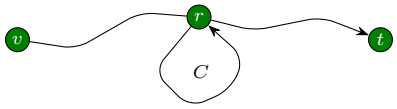
5.7.5 Negative cycles

Problem 5.9 (Negative Cycle)**.** Given a directed graph $G = (V, E)$ with edge weights $c : E \to \mathbb{R}$, determine if G contains a negative cycle. ✍

Lemma 5.3. If $\mathrm{OPT}(n, v) = \mathrm{OPT}(n-1, v)$ for all v, then there is no negative cycle on any path to t. ◁

Proof (Contrapositive)**.** If there were a negative cycle from v to t, then $\mathrm{OPT}(i, v)$ would keep on decreasing in every iteration of $i > n - 1$. □

Lemma 5.4. If $\mathrm{OPT}(n, v) < \mathrm{OPT}(n-1, v)$ for some node v, then some path from v to t contains a cycle C with negative cost. ◁



Proof. Any path with n edges contains a cycle (it must have a vertex r repeating by the pigeonhole principle). Since $\mathrm{OPT}(n, v) < \mathrm{OPT}(n-1, v)$, this cycle must have negative cost. □

Lemma 5.4 gives a condition to check if there is a negative cycle on a path to a given t.

Claim 5.5. Can detect the existence of a negative cost cycle in G in $O(mn)$ time. ◁

Add a "super-sink" t and connect all nodes to t with a 0-cost edge. Check if $\mathrm{OPT}(n, v) = \mathrm{OPT}(n-1, v)$ for all nodes v on new graph G':

- if yes, then no negative cycle in G (Lemma 5.3)
- if no, then extract cycle from shortest path from v to t (from the entry whose value reduced)

Theorem 5.6. G has a negative cycle iff there is a non-simple path from some node v to t in G' that contains a negative cycle. ◁

Proof. Let $G' = (V', E')$ with $V' = V \cup \{t\}$ and $E' = E \cup \{(v, t) : v \in V\}$, where each new edge has cost 0.

$\Rightarrow$: Suppose G contains a negative cycle C, and let v be a vertex on C. In G' there is a path from v to t that traverses C once and then takes the edge (v, t). This path contains the negative cycle C, hence there exists a path to t containing a negative cycle.

$\Leftarrow$: Suppose there exists a path from some v to t in G' that contains a negative cycle C. No directed cycle in G' can use t, since all added edges enter t and t has no outgoing edges. Therefore C uses only edges in E, so C is a negative cycle in G. □

5.8 Matrix Chain Multiplication

Problem 5.10 (Matrix Chain Multiplication)**.** Given matrices $\underline{\boldsymbol{A}}_1, \ldots, \underline{\boldsymbol{A}}_n$ with dimensions $\underline{\boldsymbol{A}}_i \in \mathbb{R}^{p_{i-1} \times p_i}$ (so the dimension vector is $\vec{\boldsymbol{p}} = (p_0, \ldots, p_n)^\top$), find a parenthesization of $\underline{\boldsymbol{A}}_1 \cdots \underline{\boldsymbol{A}}_n$ that minimizes the number of scalar multiplications. ✍

Cost of one multiplication. Multiplying an $a \times b$ matrix by a $b \times c$ matrix yields an $a \times c$ matrix. Each of the ac entries takes b scalar multiplications and $b-1$ additions, so the total cost is abc scalar multiplications and $ac(b-1)$ additions, i.e. $\Theta(abc)$ arithmetic operations.

DP formulation. For $1 \le i \le j \le n$, let $m(i,j)$ be the minimum cost to compute the product $\underline{\boldsymbol{A}}_{i..j} := \prod_{l=i}^{j} \underline{\boldsymbol{A}}_l = \underline{\boldsymbol{A}}_i \cdots \underline{\boldsymbol{A}}_j$ (where $\underline{\boldsymbol{A}}_{i..j}$ has dimension $p_{i-1} \times p_j$). We seek $m(1,n)$.

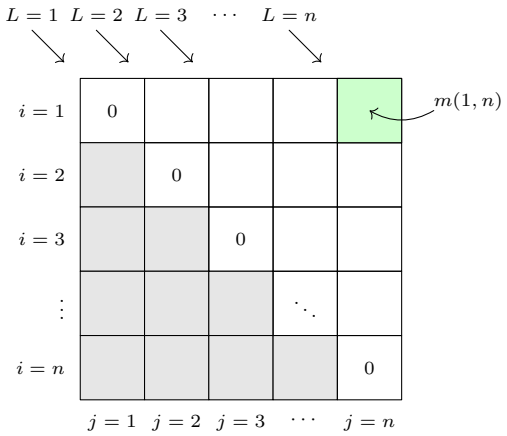
Observation 5.13. $\underline{\boldsymbol{A}}_{i..j} = \underline{\boldsymbol{A}}_{i..k} \cdot \underline{\boldsymbol{A}}_{k+1..j}$ for any $i \le k < j$. ◀

Recurrence. If $i = j$, no multiplication is needed, so $m(i,i) = 0$. Now consider $i < j$. We must split the product somewhere $i \le k < j$ (top-level parenthesization). The miniumum cost of those subproblems is $m(i,k)$ and $m(k+1,j)$ by definition. Finally, the cost to multiply the two resulting matrices is $p_{i-1}p_kp_j$. Thus,

$$m(i,j) = \begin{cases} 0 & i = j \\ \min\limits_{i \le k < j} \left\{ m(i,k) + m(k+1,j) + p_{i-1}\, p_k\, p_j \right\} & i < j \end{cases}$$

Interpretation: choose the last/top-level split k optimally, compute both subchains optimally, then multiply the results ($p_{i-1} \times p_k$ times $p_k \times p_j$).

Bottom-up computation (Algorithm 5.17). Compute by increasing chain length $L = j - i + 1$ (diagonal-by-diagonal), since $m(i,j)$ depends on shorter subchains.



Algorithm 5.17 Matrix Chain Multiplication

Require: dimensions $p_0, \ldots, p_n$ where $\underline{\boldsymbol{A}}_i$ is $p_{i-1} \times p_i$

allocate $m[1 \ldots n, 1 \ldots n]$ ▷ costs
allocate $s[1 \ldots n-1, 2 \ldots n]$ ▷ split points
for $i = 1, \ldots, n$ **do**
  $m[i,i] \leftarrow 0$
for $L = 2, \ldots, n$ **do** ▷ length of subchain
  for $i = 1, \ldots, n - L + 1$ **do**
    $j \leftarrow i + L - 1$
    $m[i,j] \leftarrow \infty$
    for $k = i, \ldots, j-1$ **do** ▷ try all splits
      `cost` $\leftarrow m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$
      if `cost` $< m[i,j]$ **then** ▷ found new minimum?
        $m[i,j] \leftarrow$ `cost` ▷ save its cost
        $s[i,j] \leftarrow k$ ▷ save split point
return $m[1,n]$ and split table s

Time and space complexity of Algorithm 5.17: There are three nested loops, and each may iterate at most n times. Thus the running time is $O(n^3)$. The tables use $\Theta(n^2)$ space.

Reconstructing the optimal parenthesization. Given the split table s from Algorithm 5.17, we can reconstruct the optimal parenthesization recursively (Algorithm 5.18).

Algorithm 5.18 Matrix Chain Multiplication

Require: marices $\underline{\boldsymbol{A}}_1, \ldots, \underline{\boldsymbol{A}}_n$, split table s

```
1: function MULTIPLY(i, j)
2:     if i = j then                                   ▷ base case
3:         return A_i
4:     else
5:         k ← s[i, j]                                 ▷ optimal split point
6:         X ← MULTIPLY(i, k)                          ▷ A_{i..k}
7:         Y ← MULTIPLY(k + 1, j)                      ▷ A_{k+1..j}
8:         return X · Y                                ▷ A_{i..j} = A_{i..k} · A_{k+1..j}
```

restricted version of a more general optimization problem, called linear programming.

Definition 6.1 (Flow Network)**.** Abstraction for stuff *flowing* through the edges of a network. A flow network is a directed graph $G = (V, E)$ with a *source* and a *sink* node $s, t \in V$ (where flow originates and is consumed, respectively). Each edge $(u, v) \in E$ has a capacity $c(u, v) \in \mathbb{N}_0$, where $c(u, v) = 0$ if $(u, v) \notin E$. ✍

Remark 6.1. Multiple sources/sinks can be modeled by adding a *super-source* and *super-sink* with infinite capacity. ◀

Caution 6.2. If we have further restrictions, such as flow from certain sources can only go to certain sinks, we have the *multi-commodity flow problem*, which is NP-hard. ◀

6.1 Flows and Cuts

Two rich algorithmic problems with a beautiful duality. Cornerstones in combinatorial optimization.

Definition 6.2 (Flow)**.** A *s-t flow* is a function $f : E \to \mathbb{R}_{\geq 0}$ that satisfies

- *Capacity:* For each $e \in E$
$$0 \leq f(e) \leq c(e) \tag{6.1}$$
- *Conservation:* For each $v \in V \setminus \{s, t\}$
$$\sum_{u \in V} f(u, v) = \sum_{w \in V} f(v, w) \tag{6.2}$$

where $s, t \in V$ are the source and sink nodes. ✍

Convention. We extend f and c to all ordered pairs $(u, v) \in V \times V$ by setting $f(u, v) = 0$ and $c(u, v) = 0$ whenever $(u, v) \notin E$. ◀

Definition 6.3. The *value* $v(f)$ (also denoted by $|f|$) of a flow f is

$$\boxed{v(f) \coloneqq \sum_{w \in V} f(s, w) = \sum_{u \in V} f(u, t)} \tag{6.3}$$

where equality follows from flow conservation. ✍

Problem 6.4 (Max Flow)**.** Find an s-t flow f of maximum value (6.3). ✍

Definition 6.5. An *s-t cut* is a partition (A, B) of V with $s \in A$ and $t \in B = V \setminus A$. ✍

Definition 6.6 (Cut Capacity)**.** The *capacity* of the cut is

$$\boxed{c(A, B) \coloneqq \sum_{a \in A} \sum_{b \in B} c(a, b)} \tag{6.4}$$

i.e. the summation of all edges going from A to B. ✍

Problem 6.7 (Min Cut)**.** Find an s-t cut (A, B) of minimum capacity (6.4). ✍

Definition 6.8. The *net flow* across the cut is

$$\boxed{f(A, B) \coloneqq \sum_{u \in A} \sum_{v \in B} f(u, v) - \sum_{u \in B} \sum_{v \in A} f(u, v)} \tag{6.5}$$

✍

Lemma 6.1 (Flow Value)**.** Let f be any flow and (A, B) be any cut. Then

$$f(A, B) = v(f) \tag{6.6}$$

i.e. the net flow across any cut equals the amount leaving s. ◁

Proof.

$$
\begin{aligned}
v(f) &\overset{(6.3)}{=} \sum_{v \in V} f(s,v) \\
&= \sum_{v \in V} f(s,v) + 0 \\
&\overset{(6.2)}{=} \sum_{v \in V} f(s,v) + \sum_{u \in A \setminus \{s\}} \left(\sum_{v \in V} f(u,v) - \sum_{v \in V} f(v,u) \right) \\
&= \sum_{u \in A} \left(\sum_{v \in V} f(u,v) - \sum_{v \in V} f(v,u) \right) \\
&= \sum_{u \in A} \sum_{v \in V} f(u,v) - \sum_{u \in A} \sum_{v \in V} f(v,u) \\
&= \sum_{u \in A} \left(\sum_{v \in A} f(u,v) + \sum_{v \in B} f(u,v) \right) - \sum_{u \in A} \left(\sum_{v \in A} f(v,u) + \sum_{v \in B} f(v,u) \right) \\
&= \cancel{\sum_{u \in A} \sum_{v \in A} f(u,v)} + \sum_{u \in A} \sum_{v \in B} f(u,v) - \cancel{\sum_{u \in A} \sum_{v \in A} f(v,u)} - \sum_{u \in A} \sum_{v \in B} f(v,u) \\
&= \sum_{u \in A} \sum_{v \in B} f(u,v) - \sum_{u \in B} \sum_{v \in A} f(u,v) \\
&= f(A,B)
\end{aligned}
$$

□

Theorem 6.2 (Weak Duality)**.** Let f be any flow and (A, B) be any cut. Then

$$\boxed{v(f) \leq c(A,B)} \tag{6.7}$$

i.e. the value of the flow is at most the capacity of the cut. ◁

Proof.

$$
\begin{aligned}
v(f) &\overset{(6.6)}{=} f(A,B) \overset{(6.5)}{=} \sum_{u \in A} \sum_{v \in B} f(u,v) - \sum_{u \in B} \sum_{v \in A} f(u,v) \\
&\leq \sum_{u \in A} \sum_{v \in B} f(u,v) \\
&\overset{(6.1)}{\leq} \sum_{u \in A} \sum_{v \in B} c(u,v) \\
&\overset{(6.4)}{=} c(A,B)
\end{aligned}
$$

□

Corollary 6.3 (Certificate of Optimality)**.** Let f be any flow and (A, B) be any cut. If $v(f) = c(A, B)$, then f is a Max Flow and (A, B) is a Min Cut. ◁

Observation 6.3. In addition, if $v(f) = c(A, B)$, then

$$f_{\text{in}}(A) \coloneqq \sum_{u \in B} \sum_{v \in A} f(u,v) = 0 \tag{6.8}$$

and

$$f_{\text{out}}(A) \coloneqq \sum_{u \in A} \sum_{v \in B} f(u,v) = v(f) = c(A,B) \tag{6.9}$$

i.e. all edges from A to B are saturated and all edges from B to A carry zero flow. ◀

Proof. From the Proof of Theorem 6.2, we have

$$v(f) = f_{\text{out}}(A) - f_{\text{in}}(A) \leq f_{\text{out}}(A) \leq c(A,B)$$

but if $v(f) = c(A, B)$, all inequalities must be equalities, yielding (6.8) and (6.9). □

Caution 6.4. Dynamic Programming (Section 5) cannot be applied because the problem does not exhibit optimal substructure (Definition 5.1). ◀

A naïve greedy strategy (Section 4) will not work for Max Flow either, because local optimality does not imply global optimality. This happens, because once we send flow along an edge, we might later want to "take it back" to reroute it more efficiently, so in addition to increase flow along edges, we need the ability to *reduce* it as well.

We need the notion of a

6.2 Residual Network

Given a flow f, each original edge $e = (u, v) \in E$ gives rise to up to two *residual arcs*:

- a *forward residual arc* e^+ from u to v with residual capacity
$$c_f(e^+) \coloneqq c(e) - f(e) \tag{6.10}$$
included if $f(e) < c(e)$
- a *backward residual arc* e^- from v to u with residual capacity
$$c_f(e^-) \coloneqq f(e) \tag{6.11}$$
included if $f(e) > 0$

Each residual arc $a \in E_f$ arises from a unique original edge $e \in E$ (its *parent edge*); denote this edge by $\pi(a)$. We indicate whether a is `forward` or `backward` by $\text{sign}(a)$. We view a residual arc as a labeled object $a = (e, \sigma)$ with $\sigma \in \{\texttt{forward}, \texttt{backward}\}$, so its parent edge is $\pi(a) = e$ and $\text{sign}(a) = \sigma$.

Definition 6.9 (Residual Network)**.** The *residual network* $G_f = (V, E_f)$ with respect to flow f is the directed multigraph on vertex set V whose arc multiset E_f contains exactly all residual arcs e^+ and e^- with positive residual capacity, as defined in (6.10)–(6.11). ✍

If both (u, v) and (v, u) are edges in G, then G_f may contain two distinct residual arcs from u to v; hence we view G_f as a directed multigraph.

Since each edge of the original network G may result in the generation of at most two arcs in the residual network G_f, the latter is of the same asymptotic size as G. Note that if $f \equiv 0$, then G_f contains exactly the forward residual arc $e^+ = (e, \texttt{forward})$ for every $e = (u, v) \in E$, with residual capacity $c(e)$; in particular, G_f has the same reachability structure as G.

Definition 6.10 (Augmenting Path)**.** An *augmenting path* is a simple s-t path P in the residual network G_f. ✍

The *bottleneck capacity* (also called residual capacity) of an Augmenting Path P is

$$\boxed{c_f(P) \coloneqq \min_{a \in P} c_f(a)} \tag{6.12}$$

where the minimum is taken over all arcs on the path P. Since all arcs in P have positive residual capacity, $c_f(P) > 0$.

Algorithm 6.1 Augment

```
function AUGMENT(f, c, P)
    b ← c_f(P)                                   ▷ bottleneck capacity (6.12)
    for all residual arc a ∈ P do
        e ← π(a)                                 ▷ retrieve original edge
        if sign(a) = forward then                ▷ a is forward
            f(e) ← f(e) + b
        else                                     ▷ a is backward
            f(e) ← f(e) − b
    return f
```

Algorithm 6.2 Ford-Fulkerson Method

Require: Flow network $G = (V, E)$, source s, sink t, capacities c

```
for all e ∈ E do
    f(e) ← 0                                     ▷ initial flow
G_f ← Residual Network   ▷ initially only forward arcs with c_f(e⁺) = c(e), since f ≡ 0
while there exists an Augmenting Path P in Residual Network G_f do
    f ← AUGMENT(f, c, P)                         ▷ augment flow along P
    Update G_f based on new flow f
return f
```

At the end of the Algorithm 6.2, the Residual Network G_f contains no Augmenting Paths.

Min cut (at the end of Ford-Fulkerson Method): Let A be the set of vertices reachable from the source s in residual network G_f. By definition of A, $s \in A$. Since G_f contains no augmenting path, $t \notin A$. (A, B) with $B = V \setminus A$ is a cut. (A, B) is a Min Cut by Theorem 6.6.

6.3 Max-Flow Min-Cut Theorem

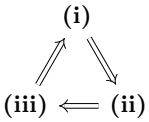
Theorem 6.4 (Augmenting Path)**.** Flow f is a Max Flow iff there are no Augmenting Paths in the Residual Network G_f. ◁

Theorem 6.5 (Max Flow / Min Cut, Ford-Fulkerson 1956)**.** The value of the Max Flow is equal to the value of the Min Cut. ◁

Theorem 6.6 (Max Flow / Min Cut)**.** Let f be a flow in G. The following 3 are equivalent:

- **(i)** There is cut (A, B) such that $v(f) = c(A, B)$.
- **(ii)** f is a Max Flow and (A, B) is a Min Cut.
- **(iii)** The Residual Network G_f contains no Augmenting Paths. ◁

Proof. We prove the implications in the following triangle:



(i) ⇒ (ii):

Corollary 6.3.

(ii) ⇒ (iii) (contrapositive):

If there were an augmenting path in G_f, we could improve f by sending flow along this path.

(iii) ⇒ (i):

Let A be the set of vertices reachable from the source s in G_f. By construction, $s \in A$. Since G_f contains no augmenting path, $t \notin A$. If we set $B = V \setminus A$, then (A, B) is a cut. We now want to show again that the inqualities in the Proof of Theorem 6.2 are actually equalities. Let $u \in A$ and $v \in B$. If $(u, v) \in E$ and $f(u, v) < c(u, v)$, then the forward residual arc (u, v) has positive residual capacity and is therefore present in G_f, implying $v \in A$, a contradiction. Hence $f(u, v) = c(u, v)$. Similarly, if $(v, u) \in E$ and $f(v, u) > 0$, then the backward residual arc (u, v) is present in G_f, again implying $v \in A$, a contradiction. Hence $f(v, u) = 0$.

This closes the triangle of implications and proves Theorem 6.6. □

Assumption 6.5. All capacities are integers between 1 and C. ◀

Invariant 6.6. Every flow value $f(e)$ and every residual capacity $c_f(e)$ remains an integer throughout the execution of Algorithm 6.2. ◀

Theorem 6.7. Algorithm 6.2 terminates in at most $v(f^*)$ iterations, where $v(f^*) \leq nC$ is the value of a maximum flow. ◁

Proof. Each Augment increases $v(f)$ by at least 1 and $v(f^*) \leq \deg^+(s) \cdot C \leq nC$. □

Corollary 6.8. Algorithm 6.2 runs in time $O(v(f^*) \cdot m)$ (not polynomial in n!), since an Augmenting Path can be found in $O(m)$ time. ◁

Since $v(f^*) \leq nC$, we also have that Algorithm 6.2 runs in time $O(nCm)$.

Corollary 6.9. If $C = 1$, Algorithm 6.2 runs in $O(nm)$ time. ◁

Theorem 6.10 (Integrality)**.** If all capacities are integers (Assumption 6.5), there exists a Max Flow, such that $f(e)$ is an integer for all $e \in E$. ◁

Proof. Algorithm 6.2 terminates (Theorem 6.7) and maintains Invariant 6.6. □

Corollary 6.11. If all capacities are integers (Assumption 6.5) and there exists a flow of value k, there exists an *integral* flow of value k. ◁

Proof. Deleting $v(f^*) - k$ unit augmenting paths from an integral Max Flow f^*. □

6.4 Choosing Augmenting Paths

Running time of generic Ford-Fulkerson Method is *pseudo-polynomial*, not polynomial in size of the input n, m.

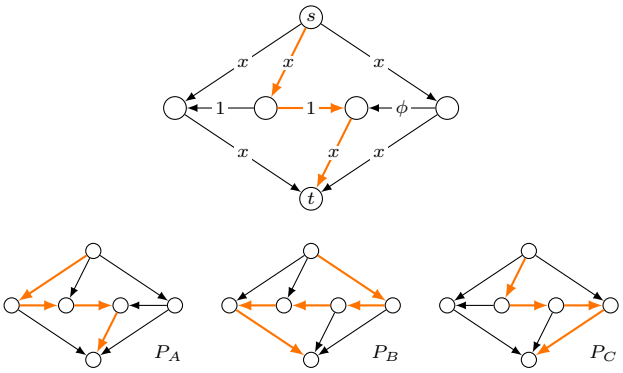
Use care when selecting Augmenting Paths:

- Some choices lead to exponential algorithms.
- Clever choices lead to polynomial algorithms (Edmonds-Karp).
- If capacities are irrational, may not terminate!

Recall that for $c(e) \in \mathbb{N}$, Algorithm 6.2 halts (Theorem 6.7). If we multiply all capacities by the same (positive) factor, the Max Flow increases proportionally. Thus, if $c(e) \in \mathbb{Q}_{>0}$ for all $e \in E$, Algorithm 6.2 eventually halts.

However, if we allow $c(e) \in \mathbb{R}_{>0}$, Algorithm 6.2 can loop forever, finding smaller and smaller Augmenting Paths! Worse yet, it may not even converge to the Max Flow, or even to a significant fraction of it! One of the simplest examples of this effect is shown in Example 6.1.

Example 6.1 (Uri Zwick, 1993). Consider the following network



where $\phi = \frac{\sqrt{5}-1}{2} \approx 0.618$ denotes the golden ratio, satisfying $\phi^2 + \phi = 1$. x is some large integer.

To see how Algorithm 6.2 can loop forever, we track the residual capacities of the three horizontal edges.

Suppose the Ford-Fulkerson Method starts by choosing the central augmenting path (shown in the large figure above), adding 1 to the flow. The 3 horizontal edges, now have residual capacities 1, 0, and ϕ (from left to right). Suppose inductively that the horizontal residual capacities are ϕ^{k-1}, 0, ϕ^k for some positive integer $k \in \mathbb{N}$. Then Augment along ...

i. P_B, adding ϕ^k to the flow; the residual capacities are now ϕ^{k+1}, ϕ^k, 0

ii. P_C, adding ϕ^k to the flow; the residual capacities are now ϕ^{k+1}, 0, ϕ^k

iii. P_B, adding ϕ^{k+1} to the flow; the residual capacities are now 0, ϕ^{k+1}, ϕ^{k+2}

iv. P_A, adding ϕ^{k+1} to the flow; the residual capacities are now ϕ^{k+1}, 0, ϕ^{k+2}

implying by induction that after $4n + 1$ Augmentations ($n \in \mathbb{N}_0$), the flow has value

$$v(f) = 1 + 2\sum_{i=1}^{2n} \phi^i$$

and the horizontal edges have residual capacities ϕ^{2n}, 0, ϕ^{2n+1}.

As n (the number of Augmentations) goes to ∞, the value of the flow converges to

$$\lim_{n\to\infty} v(f) = 1 + 2\sum_{i=1}^{\infty} \phi^i = 1 + 2 \cdot \frac{\phi}{1-\phi} = 2 + \sqrt{5}$$

even though the Value of the Max Flow clearly is $v(f^*) = 2x + 1 \gg 2 + \sqrt{5}$. ◀

Although computers cannot represent irrational numbers exactly, this limitation is an artifact of hardware rather than an inherent property of the underlying problem. Restricting capacities to integers is therefore artificial. Moreover, considering irrational inputs is practically illuminating: it reveals pathological worst-case behavior of algorithms under floating-point arithmetic, where round-off errors can cause a careless implementation of the Ford-Fulkerson Method to fail to terminate.

Goal: Choose augmenting paths so that:

- can find augmenting paths efficiently
- result in few iterations

Strategies:

- max bottleneck capacity (dependency on C)
- sufficiently large bottleneck capacity [Dinitz 1970]. reduce dependency on C to $\log C$. Time $O(m^2 \log C)$.
- simple idea:
  don't do anything fancy with capacities, just choose the path with fewest number of edges [Edmonds-Karp 1972]. Polynomial time algorithm. Time $O(nm^2)$.

6.4.1 Edmonds-Karp (BFS)

Idea: Always choose an augmenting path with the minimum number of edges in the residual network.

- Implemented by running BFS in G_f
  - with BFS, we take the least number of edges
  - with DFS, we do not have that property!
  - BFS: $O(m + n) = O(m)$, since $m \geq n - 1$
- Each augmentation increases the shortest path distance
- Total number of iterations: $O(nm)$, so total running time is $O(nm^2)$

So we went from $O(nC)$ iterations to $O(nm)$ iterations!

6.5 Applications

6.5.1 Bipartite Matching

Definition 6.11 (Matching)**.** Given an undirected graph $G = (V, E)$, $M \subseteq E$ is a *matching* if each node appears in at most one edge in M. ✍

Definition 6.12 (Bipartite Matching)**.** A *bipartite matching* is a Matching in an undirected, bipartite graph $G = (L \cup R, E)$. ✍

Problem 6.13 (Max Bipartite Matching)**.** Find a Bipartite Matching of maximum cardinality. ✍

Max Flow formulation of Problem 6.13:

- create digraph $G' = (L \cup R \cup \{s, t\}, E')$
- direct all edges from L to R and assign infinite (or unit) capacitiy

- add unit capacity edges from s to all nodes in L
- add unit capacity edges from all nodes in R to t

Theorem 6.12. a bipartite graph G has a Matching of size k iff the network formulation G' has a flow of value k. ◁

or equivalently:

Theorem 6.13. max cardinality matching in G = value of Max Flow in G'. ◁

Proof ("≤"). Let M be a max matching in G with $|M| = k$. Define a flow f in G' by $f(s,u) = 1$, $f(u,v) = 1$, $f(v,t) = 1$ for each $(u,v) \in M$, and $f(e) = 0$ for all other edges. Capacity constraints (6.1) hold since all used edges have capacity at least 1. For any vertex $x \in L \cup R$, either x is incident to no edge of M (then $f_{\text{in}}(x) = f_{\text{out}}(x) = 0$), or to exactly one edge of M (then $f_{\text{in}}(x) = f_{\text{out}}(x) = 1$); hence flow conservation (6.2) holds. Thus f is a valid flow in G' and $v(f) = \sum_{u \in L} f(s,u) = |M| = k$. In particular, if f^* is a max flow in G', then $v(f^*) \geq v(f) = k$. □

Proof ("≥"). Let f be a max flow in G' with $v(f) = r$. By integrality (Theorem 6.10), we may assume f is integral, hence $f(e) \in \{0,1\}$ by capacity constraints (6.1). Define $M \coloneqq \{(u,v) \in E \mid u \in L, v \in R, f(u,v) = 1\}$. For each $u \in L$ we have $\sum_{v \in R} f(u,v) = f(s,u) \leq 1$ by flow conservation (6.2) at u, so at most one edge of M is incident to u. Similarly, for each $v \in R$ we have $\sum_{u \in L} f(u,v) = f(v,t) \leq 1$, so at most one edge of M is incident to v. Thus M is a matching in G and $|M| = \sum_{u \in L} \sum_{v \in R} f(u,v) = \sum_{u \in L} f(s,u) = v(f) = r$. In particular, if M^* is a max matching in G, then $|M^*| \geq |M| = r$. □

The edges of the matching are the ones that carry flow from L to R.

We reduced the problem 'Max Bipartite Matching' to the known problem 'Max Flow'.

Definition 6.14 (Perfect Matching)**.** A matching $M \subseteq E$ is perfect if each node appears in exactly one edge in M. ✍

A necessary condition for a Perfect Matching in a bipartite graph is $|L| = |R|$.

To compute a perfect matching (if one exists), compute a max cardinality matching:

- if cardinality $= n = |L| = |R|$, then the matching is perfect
- if cardinality $< n$, then no perfect matching exists

A cut with capacity $< n$ provides a *certificate* that no perfect matching exists!

Notation 6.15 (Neighborhood)**.** For a subset $S \subseteq V$ of nodes in a graph $G = (V,E)$, we write

$$N(S) \coloneqq \bigcup_{u \in S} \Gamma(u) \tag{6.13}$$

for the set of nodes adjacent to at least one node in S. In order to avoid ambiguity, sometimes a subscript is added to indicate the graph, e.g., $N_G(S)$. ✍

Definition 6.16 (Deficiency)**.** Let $G = V, E$ be a graph and U be an Independent Set of vertices in G. The *deficiency* of U is defined as

$$\text{def}_G(U) \coloneqq |U| - |N_G(U)| \tag{6.14}$$

Suppose G is bipartite with bipartition $L \cup R$. The *deficiency* of G with respect to one of its parts is the maximum deficiency of any subset of that part, i.e.,

$$\text{def}(G;X) \coloneqq \max_{U \subseteq X} \text{def}_G(U) \tag{6.15}$$

where $X \in \{L,R\}$. Sometimes this quantitiy is called the *critical difference* of G. ✍

Observation 6.7. If a bipartite graph $G = (L \cup R, E)$ has a Perfect Matching, then $|N(S)| \geq |S|$ for all subsets $S \subseteq L$. ◀

Proof. Suppose G has a perfect matching. Then each node in any $S \subseteq L$ is matched to a different node in $N(S)$. Thus, $|S| \leq |N(S)|$. □

Theorem 6.14 (Marriage [Frobenius 1917, Hall 1935])**.** Let $G = (L \cup R, E)$ be a bipartite graph with $|L| = |R|$. Then, G has a perfect matching iff $|N(S)| \geq |S|$ for all subsets $S \subseteq L$. ◁

Proof ('⇒')**.** Observation 6.7. □

Proof ('⇐')**.** We prove the contrapositive, i.e.

$$G \text{ has no perfect matching} \implies \exists S \subseteq L : |N(S)| < |S|$$

Suppose G does not have a perfect matching

- Formulate G' as Max Flow problem and let (A, B) be Min Cut in G'.
- Since no perfect matching, value of Max Flow $v(f^*) < n$ (Theorem 6.13) and therefore (by Theorem 6.6), capacity of the Min Cut $\text{cap}(A, B) < n = |L|$.
- Define $L_A = L \cap A$, $L_B = L \cap B$, $R_A = R \cap A$, $R_B = R \cap B$.
- Since Min Cut can't use ∞ edges (otherwise the cut capacity would be ∞), no edge from L to R can be in cut; only edges from s to B (L_B) and edges from A to t (R_A).
- Since all these edges have unit capacity, $\text{cap}(A, B) = |L_B| + |R_A|$. Furthermore, no edge from L_A to R_B implies $N(L_A) \subseteq R_A$.
- $|N(L_A)| \leq |R_A| = \text{cap}(A, B) - |L_B| < |L| - |L_B| = |L_A|$.
- $|N(L_A)| < |L_A|$. Choose $S = L_A$. □

Theorem 6.15 (Hall)**.** Let $G = (L \cup R, E)$ be bipartite and let $X \in \{L, R\}$. Then, G has a X-saturating matching iff $|N(S)| \geq |S|$ for all subsets $S \subseteq X$, i.e., $\text{def}(G; X) = 0$. ◁

Theorem 6.16 (quantitative Hall)**.** Let $G = (L \cup R, E)$ be bipartite and let $X \in \{L, R\}$. Then the maximum matching size is

$$\nu(G) = |X| - \text{def}(G; X) = \min_{U \subseteq X} \Big(|X| - \big(|U| - |N_G(U)| \big) \Big)$$

Equivalently, there exists a maximum matching that leaves exactly $\text{def}(G; X)$ vertices of X unmatched. ◁

Remark 6.8. Hall's condition appears in increasing generality:

$$\begin{array}{cll} & \text{Theorem 6.14} & \text{(balanced perfect matching)} \\ \Leftarrow & \text{Theorem 6.15} & (X\text{-saturating matching}) \\ \Leftarrow & \text{Theorem 6.16} & \text{(exact maximum size via deficiency)} \end{array}$$

Also note that Theorem 6.15 and Theorem 6.16 do not require $|L| = |R|$. ◀

Which max flow algorithm to use for Bipartite Matching?

- Generic augmenting path: $O(m \cdot \text{val}(f^*)) = O(mn)$, since $\text{val}(f^*) \leq n$
- Capacity scaling: $O(m^2 \log C) = O(m^2)$
- Shortest augmenting path: $O(m\sqrt{n})$

Non-bipartite Matching:

- Structure of non-bipartite graphs is more complicated, but well-understood [Tutte-Berge, Edmonds-Galai]
- Blossom algorithm: $O(n^4)$ [Edmonds 1965]
- Best known: $O(m\sqrt{n})$ [Micali-Vazirani 1980]

Definition 6.17. A graph is said to be k-regular if every node has degree k. ✍

Lemma 6.17. Let $G = (L \cup R, E)$ be a k-regular bipartite graph with $k \geq 1$. Then $|L| = |R|$ (i.e. G is balanced). ◁

Proof. We count $|E|$ in two different ways. Because G is bipartite with partite sets L and R, every edge is incident with exactly one vertex of L and exactly one vertex of R. Hence, summing degrees over either L or R counts each edge exactly once. Since $\deg(v) = k$ for all $v \in L \cup R$, we have $k|L| = |E| = k|R|$, and since $k \geq 1$, we have $|L| = |R|$. □

Theorem 6.18. Every k-regular bipartite graph with $k \geq 1$ has a Perfect Matching. ◁

Proof. Let $G = (L \cup R, E)$ be a k-regular bipartite graph with $k \geq 1$. By Lemma 6.17, $|L| = |R| = n$.

Add the source s and sink t to construct the usual network G' with unit capacities (as described Above).

Define a (fractional) flow f by

$$f(e) = \begin{cases} \frac{1}{k} & \text{if } e \text{ is from } L \text{ to } R \\ 1 & \text{otherwise} \end{cases}$$

It is easy to verify that f satisfies capacity constraints (6.1) and flow conservation (6.2). Hence, f is a valid flow in G' with value $v(f) = n$.

All capacities are integral, and there exists a flow of value n; thus by Corollary 6.11 there exists an integral flow f' with $v(f') = n$. Since all capacities are 1, we have $f'(e) \in \{0, 1\}$ for all arcs.

Let $M \coloneqq \{(u, v) \in E \mid f'(u, v) = 1\}$. Because $v(f') = n$, all arcs (s, u) and (v, t) are saturated. By conservation, each $u \in L$ has exactly one outgoing arc (u, v) with flow 1, and each $v \in R$ has exactly one incoming arc (u, v) with flow 1. Therefore every vertex is incident to exactly one edge of M, so M is a perfect matching. □

Definition 6.18. A bipartite graph $G = (L \cup R, E)$ is (k_L, k_R)-semi-regular if every node in L has degree k_L and every node in R has degree k_R. ✍

Note that in a (k_L, k_R)-semi-regular bipartite graph, we have

$$|E| = k_L|L| = k_R|R| \implies \frac{|L|}{|R|} = \frac{k_R}{k_L}$$

Theorem 6.19. A (k_L, k_R)-semi-regular bipartite graph $G = (L \cup R, E)$ has a X-saturating matching where X denotes the part with larger degree, i.e. $k_X = \max\{k_L, k_R\}$. ◁

Proof. Denote by Y the other part, i.e. $k_Y = \min\{k_L, k_R\}$. Consider any subset $S \subseteq X$ and its neighborhood $N(S) \subseteq Y$. Denote by $E(S, N(S))$ the set of edges with one endpoint in S and the other in $N(S)$. Counting the edges in $E(S, N(S))$ by their endpoints in S gives

$$|E(S, N(S))| = k_X|S|$$

since each node in S has degree k_X. Counting the edges in $E(S, N(S))$ by their endpoints in $N(S)$ gives

$$|E(S, N(S))| \leq k_Y|N(S)|$$

since each node in $N(S)$ has degree k_Y and there may be edges from $N(S)$ to nodes outside of S. Combining these two equations gives

$$k_X|S| \leq k_Y|N(S)|$$

or equivalently

$$|N(S)| \geq \frac{k_X}{k_Y}|S| \geq |S|$$

since $k_X \geq k_Y$. By Theorem 6.15, there exists a X-saturing matching. □

6.5.2 Edge Disjoint Paths

Definition 6.19 (Edge Disjoint Paths)**.** Two paths are edge disjoint if they have no edge in common. ✍

Problem 6.20 (Max Edge Disjoint Paths)**.** Given a directed graph $G = (V, E)$ and two nodes $s, t \in V$, find the maximum number of edge disjoint s-t paths. ✍

Max Flow formulation of Problem 6.20: assign unit capacity to each edge in G.

Theorem 6.20. G has k edge-disjoint s-t paths iff G' has a flow of value k. ◁

Theorem 6.21. Max number of edge-disjoint s-t paths equals value of Max Flow. ◁

Proof ("$\leq$")**.** Let $P_1, \ldots, P_k$ be k edge disjoint s-t paths. Create a flow f as follows: Make each path P_i carry 1 unit of flow from s to t, i.e. set $f(e) = 1$ if e participates in some path P_i, and $f(e) = 0$ otherwise. Since the paths are edge disjoint, no edge carries more than 1 unit of flow, so capacity constraints (6.1) are satisfied. Flow conservation (6.2) is satisfied since at an internal node, each path has exactly one incoming and one outgoing edge. The value of the flow is $v(f) = k$ (k edges out of s carry flow one unit of flow). □

Proof ("$\geq$")**.** Let f be a flow of value k. By Integrality, there exists a 0-1 flow f of value k. Consider an edge (s, u) with $f(s, u) = 1$. By flow conservation (6.2), there exists an edge (u, v) with $f(u, v) = 1$. Continue until reaching t, always choosing a new edge with flow 1. Once reaching t we found one path. Remove all edges of this path from the graph and repeat k times. We obtain k (not necessarily simple) edge disjoint s-t paths (one for each edge our of s with flow 1). □

can eliminate cycles to get simple paths if desired

We reduced the problem 'Max Edge Disjoint Paths' to the known problem 'Max Flow'.

6.5.3 Network Connectivity

Definition 6.21. A set of edges $F \subseteq E$ disconnects t from s if all s-t paths use at least one edge in F. ✍

Problem 6.22 (Network Connectivity)**.** Given a directed graph $G = (V, E)$ and two nodes $s, t \in V$, find the minimum number of edges whose removal disconnects t from s. ✍

We can solve Problem 6.22 by finding a Min Cut in a network with unit capacities on all edges.

Theorem 6.22 ([Menger, 1927])**.** The max number of edge disjoint s-t paths is equal to the min number of edges whose removal disconnects t from s. ◁

Special case of Max Flow / Min Cut (Theorem 6.6):

- derive a Flow Network with unit capacities
- min number of edges whose removal disconnects t from s = Max Flow
- max number of edge disjoint s-t paths = Max Flow

6.6 Extensions

6.6.1 Circulation with Demands

Definition 6.23 (Circulation with Demands)**.** directed graph $G = (V, E)$ with

- edge capacities $c(e)$, $e \in E$
- node supply and demands $d(v)$, $v \in V$

where demand if $d(v) > 0$, supply if $d(v) < 0$, and transshipment node if $d(v) = 0$. ✍

Definition 6.24 (Circulation)**.** A *circulation* is a function f that satisfies

- *Capacity*: For each $e \in E$:
$$0 \leq f(e) \leq c(e) \tag{6.16}$$
- '*Conservation*': For each $v \in V$:
$$\sum_{(u,v)\in E} f(u,v) - \sum_{(v,w)\in E} f(v,w) = d(v) \tag{6.17}$$
✍

Problem 6.25 (Circulation)**.** Given V, E, c, d, does there exist a circulation f? ✍

Problem 6.25 is a feasibility (decision) problem (not an optimization one).

Observation 6.9. A necessary condition for the existence of a Circulation is

$$\sum_{v\in T} d(v) = \sum_{v:\, d(v)>0} d(v) = -\sum_{v:\, d(v)<0} d(v) = -\sum_{v\in S} d(v) =: D \tag{6.18}$$

or

$$\sum_{v\in V} d(v) = 0 \tag{6.19}$$

where S denotes the supply nodes and T the demand nodes. ◀

Max Flow formulation of Problem 6.25:

- can assume (6.18) holds; otherwise answer `no` immediately
- add a new source s and sink t
- for each v with $d(v) < 0$, add edge (s, v) with capacity $-d(v)$ (supply)
- for each v with $d(v) > 0$, add edge (v, t) with capacity $d(v)$ (demand)

Claim 6.23. G has a Circulation iff G' has Max Flow of value D. ◁

Proof ('⇒')**.** Assume G has a circulation f. Define an s-t flow f' in G' by setting $f'(e) := f(e)$ for all original edges $e \in E$, and by saturating the new edges: $f'(s, v) := -d(v)$ for $v \in S$ and $f'(v, t) := d(v)$ for $v \in T$. Capacities are satisfied by construction, and flow conservation in G' holds at every $v \in V$ because the extra inflow $-d(v)$ (resp. extra outflow $d(v)$) exactly cancels the imbalance $f_{\text{in}}(v) - f_{\text{out}}(v) = d(v)$. Finally, $v(f') = \sum_{v\in V} f'(s, v) = \sum_{v\in S}(-d(v)) = D$. □

Proof ('⇐')**.** Assume G' has an s-t flow f' with $v(f') = D$. Since $\sum_{v\in S} c(s, v) = \sum_{v\in S}(-d(v)) = D$, the value bound $v(f') \leq \sum_{v\in S} c(s, v)$ forces every edge (s, v) to be saturated, i.e. $f'(s, v) = -d(v)$ for all $v \in S$; similarly $f'(v, t) = d(v)$ for all $v \in T$. Now restrict f' to the original edges E and call this restriction f. Then f satisfies $0 \leq f(e) \leq c(e)$, and for each $v \in V$ the flow-conservation equation in G', after subtracting the (now fixed) contribution from (s, v) or (v, t), becomes exactly $f_{\text{in}}(v) - f_{\text{out}}(v) = d(v)$, so f is a valid circulation in G. □

We reduced the problem 'Circulation' to the known problem 'Max Flow'.

Theorem 6.24 (Integrality for Circulations)**.** If all capacities $c(e)$ and demands $d(v)$ are integers and there exists a Circulation, then there exists one that is integer-valued. ◁

Proof. Follows from Max Flow formulation and standard Integrality theorem. □

6.6.2 Circulation with Demands and Lower Bounds

Definition 6.26 (Circulation with Demands and Lower Bounds)**.** directed graph $G = (V, E)$ with

- edge capacities $c(e)$ and lower bounds $\ell(e)$, $e \in E$
- node supply and demands $d(v)$, $v \in V$

where demand if $d(v) > 0$, supply if $d(v) < 0$, and transshipment node if $d(v) = 0$. ✍

Definition 6.27 (Circulation)**.** A *circulation* is a function f that satisfies

- *Capacity*: For each $e \in E$:
$$\ell(e) \leq f(e) \leq c(e) \tag{6.20}$$
- '*Conservation*': For each $v \in V$:
$$\sum_{(u,v)\in E} f(u, v) - \sum_{(v,w)\in E} f(v, w) = d(v) \tag{6.21}$$
✍

Problem 6.28 (Circulation with Lower Bounds)**.** Given V, E, ℓ, c, d, does there exist a circulation f? ✍

Idea is to reduce Problem 6.28 to Problem 6.25 (with no lower bounds)

- generate an initial (invalid) circulation f_0 based on $\ell(e)$:
$$f_0(u, v) = \ell(u, v)$$
- compute $L(v)$ for each node v ($L(v)$ is the excess flow coming into v (because of lower bounds), it may be negative (flow deficit)):
$$L(v) = f_{0,\text{in}}(v) - f_{0,\text{out}}(v) = \textstyle\sum_{(u,v)\in E} \ell(u, v) - \sum_{(v,w)\in E} \ell(v, w)$$
- build residual circulation network G':
  - same vertices as G
  - capacity for edges: $c'(u, v) = c(u, v) - \ell(u, v)$
  - demands for nodes: $d'(v) = d(v) - L(v)$ (remainder demand)
- compute a valid circulation f_1 on G'
- final circulation on G is: $f = f_0 + f_1$

Theorem 6.25. There exists a circulation in G iff there exists a circulation in G'. And if all demands, capacities, and lower bounds are integers, then there is an integer-valued circulation. ◁

Proof. f is a circulation in G iff f', where $f'(e) = f(e) - \ell(e)$ for every $e \in E$, is a circulation in G'.

'⇒':

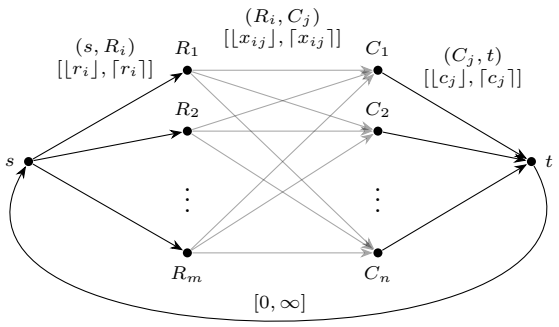
- suppose there is a circulation f in G
- define f' in G' by $f'(e) = f(e) - \ell(e)$
- f' satisfies capacity (6.20) and conservation (6.21) in G'

'⇐':

- suppose there is a circulation f' in G'
- define f in G by $f(e) = f'(e) + \ell(e)$
- f satisfies capacity (6.16) and conservation (6.17) in G □

Example 6.2 (Table Rounding). Given an $m \times n$ table (x_{ij}) with row sums $r_i \coloneqq \sum_{j=1}^{n} x_{ij}$ and column sums $c_j \coloneqq \sum_{i=1}^{m} x_{ij}$, we want to round each entry x_{ij} to an integer $y_{ij} \in \{\lfloor x_{ij} \rfloor, \lceil x_{ij} \rceil\}$ such that $\sum_{j=1}^{n} y_{ij} \in \{\lfloor r_i \rfloor, \lceil r_i \rceil\}$ for all rows i and $\sum_{i=1}^{m} y_{ij} \in \{\lfloor c_j \rfloor, \lceil c_j \rceil\}$ for all columns j.

We can model this as a Circulation with Lower Bounds: Create nodes $R_1, \ldots, R_m$ (rows), $C_1, \ldots, C_n$ (columns), and two nodes s, t. Add edges (s, R_i) with bounds $[\lfloor r_i \rfloor, \lceil r_i \rceil]$, (R_i, C_j) with bounds $[\lfloor x_{ij} \rfloor, \lceil x_{ij} \rceil]$, (C_j, t) with bounds $[\lfloor c_j \rfloor, \lceil c_j \rceil]$, and one extra edge (t, s) with bounds $[0, \infty]$. Set all node demands to 0.



Claim. There is a consistent rounding iff the above network has an *integral* valid circulation. ◁

Proof. Each edge except (t, s) corresponds to one displayed number in the table (entries and row/column aggregates). Given an integral circulation f, round each number to the integer $f(e)$. Flow conservation at each R_i enforces that the rounded row entries sum to the rounded row total, and conservation at each C_j enforces the analogous column condition. Conversely, any consistent rounding defines integer values on all edges except (t, s); setting $f(t, s)$ to the total rounded sum makes conservation hold at s and t. All bounds are respected because every rounded value lies in $\{\lfloor \cdot \rfloor, \lceil \cdot \rceil\}$. □

All lower/upper bounds are integers, so by Theorem 6.25, if a valid circulation exists then an integral one exists. Since the entries from the original table induce a valid circulation (by setting $f(e) = x_{ij}$ or $f(e) = r_i$ or $f(e) = c_j$ on the corresponding edges), a valid circulation always exists, and hence also a consistent rounding always exists. ◀

7 Complexity Theory

'efficiently solvable' = solvable in polynomial time.

many problems arre 'hard', i.e. no known polynomial time solution is known

a large class of such problems has been characterized and shown to be equivalent: NP-complete problems. a polynomial algo for one would mean a polynomial time algo for *all* NP-complete problems.

NP-complete in practice: prolems that are computationally hard for all partical purposes. look for an approximation algorithm.

Algorithm design patterns.	Example
Greed.	$O(n \log n)$ interval scheduling.
Divide-and-conquer.	$O(n \log n)$ FFT, merge sort.
Dynamic programming.	$O(n^2)$ edit distance.
Duality.	$O(n^3)$ bipartite matching.
Reductions.	
Local search.	
Randomization.	
Algorithm design anti-patterns.	
NP-completeness.	$O(n^k)$ algorithm unlikely.
PSPACE-completeness.	$O(n^k)$ certification algorithm unlikely.
Undecidability.	No algorithm possible.

Working Definition [Cobham 1964, Edmonds 1965, Rabin 1966]. In practice, we are able to solve problems with polynomial time algorithms.

Which problems will we be able to solve in practice?

Yes	Probably No
Shortest path	Longest path
Matching	3D-matching
Min Cut	Max Cut
2-SAT	3-SAT
Planar 4-color	Planar 3-color
Bipartite vertex cover	Vertex cover
Primality testing	Factoring

Desired. Classify problems according to those that can be solved in polynomial-time and those cannot.

...those that cannot: provably require exponential-time.

Frustrating news. Huge number of fundamental problems have defied classification for decades.

This chapter. Show that some fundamental problems are "computationally equivalent" and appear to be different manifestations of one really hard problem.

Decision problem. A problem whose output is Yes/No

Example 7.1 (decision version of MST). Given a weighted graph G and an integer k, does G have a spanning tree of weight (at most) k? ◀

A decision problem can be seen as a language recognition problem.

Example 7.1 (continuing). Define the language

$$L_{\text{MST}} = \{(G, k) \mid G \text{ has a spanning tree of weight at most } k\}.$$

where (G, k) is a reasonable encoding of the pair G, k as a string. Given a string $x = (G, k)$, does x belong to the language L_{MST}? ◀

So solving the decision problem = solving a language membership problem.

The language of a decision problem Π:

$$L(\Pi) = \{x \in \{0,1\}* \mid x \text{ is a representation of a Yes-instance of } \Pi\}.$$

Solving Π: determine the anser (Y/N) to an arbitrary instance of Π

Solving $L(\Pi)$: determine if a string x (representing an arbitrary instance of Π) belongs in $L(\Pi)$.

Given a string x we can determine if $x \in L(\Pi)$:

- decode x as an instance of Π
- feed x to an algorithm that solves Π
- if answer is Y, accept x; otherwise reject x

time required: time to decode x + time to solve Π

Definition 7.1 (P). set of all languages (i.e., decision problems) for which membership can be determined in (worst case) polynomial time. ✍

7.1 Reductions

Definition 7.2 (polynomial time reduction). Given two decision problems X and Y, we write

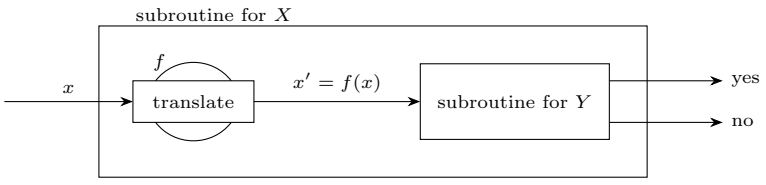
$$\boxed{X \leq_P Y} \tag{7.1}$$

if there is a polynomial-time computable function f such that

$$x \in X \iff f(x) \in Y \tag{7.2}$$

where membership '$\in$' means 'is a Yes-instance of the problem'. ✍

Definition 7.2 can be viewed as follows:



Remark 7.1. We say "X reduces polynomially to Y". (7.2) means:

- $x \in X \implies f(x) \in Y$
- $f(x) \in Y \implies x \in X$

or equivalently, by taking contrapositives,

- $f(x) \notin Y \implies x \notin X$
- $x \notin X \implies f(x) \notin Y$

where '$\notin$' means 'is a No-instance of the problem'. ◀

Purpose. Classify problems according to their relative difficulty.

Property 7.1 (transitivity). $L_1 \leq_P L_2$ and $L_2 \leq_P L_3$ implies $L_1 \leq_P L_3$ ◁

Proof. Idea: compose the two algorithms.

Solve an instance x of X by an algorithm to solve Z:

- transform x to $y = f_{X \to Y}(x)$ (an instance of Y).
- transform y to $z = f_{Y \to Z}(y)$ (an instance of Z)
- feed z to the algorithm for Z. If the answer is yes, then it is yes for y. Output yes for x. □

Example 7.2. 3-SAT $\leq_P$ Independent Set $\leq_P$ Vertex Cover $\leq_P$ Set Cover ◀

Fact 7.2. Assume $X \leq_P Y$.

Design algorithms. If we can solve Y in poly time, we can also solve X in poly time as well (the algorithm that solves Y in poly time can be used to solve X in poly time).

Establish intractability. If we cannot solve X in poly time, we cannot solve Y in poly time either. ◁

Establish equivalence. If $X \leq_P Y$ and $Y \leq_P X$, we write $X \equiv_P Y$.

Next, we introduce *three basic reduction strategies*; Reduction by simple equivalence, Reduction from special case to general case, Reduction by encoding with gadgets.

7.1.1 Reduction by simple equivalence

Problem 7.3 (Independent Set)**.** Given a graph $G = (V, E)$ and an integer k, is there a subset of vertices $S \subseteq V$ such that $|S| \geq k$ and no two vertices in S are joined by an edge? (for each edge at most one of its endpoints is in S) ✍

Problem 7.4 (Vertex Cover)**.** Given a graph $G = (V, E)$ and an integer k, is there a subset of vertices $T \subseteq V$ such that $|T| \leq k$ and, for each edge, at least one of its endpoints is in T? (for each edge at least one of its endpoints is in T) ✍

Claim 7.3. Vertex Cover $\equiv_P$ Independent Set. (IS $\leq_P$ VC and VC $\leq_P$ IS) ◁

Proof. The idea is to show that S is an independent set in G iff $V \setminus S$ is a vertex cover in G. And then

- IS $\leq_P$ VC: given an instance (G, k) of IS, construct the instance $(G, n - k)$ of VC.
- VC $\leq_P$ IS: given an instance (G, k) of VC, construct the instance $(G, n - k)$ of IS.

To prove the idea, we show the two directions of the equivalence:

'$\Rightarrow$': Let S be an independent set in G. Consider an arbitrary edge $(u, v) \in E$. S independent set $\Rightarrow u \notin S$ or $v \notin S \Rightarrow u \in V \setminus S$ or $v \in V \setminus S$. Thus, $V \setminus S$ is a vertex cover in G.

'$\Leftarrow$': Let T be a vertex cover in G. Set $S = V \setminus T$. Consider two nodes $u, v \in S$. An edge (u, v) would not be covered by T, thus no such edge can exit. So, no two nodes in S are joined by an edge $\Rightarrow S$ is an independent set in G. □

7.1.2 Reduction from special case to general case

Problem 7.5 (Set Cover)**.** Given a set U (*universe*) of elements, a collection $F = \{S_1, \ldots, S_m\}$ of subsets of U, and an integer k, does there exist a collection C of at most k of these subsets whose union is U? ✍

Claim 7.4. Vertex Cover $\leq_P$ Set Cover. ◁

Proof. Given an instance $(G = (V, E), k)$ of Vertex Cover, we construct an instance $(U, \{S_i\}, k)$ of Set Cover such that G has a VC of size k iff U has a SC of size k.

Create the Set Cover instance as follows:

- $U = E$ (universe = edges of G)

- k remains the same
- $S_v = \{e \in E \mid e \text{ is incident to } v\}$ for each $v \in V$

G has a VC of size $\leq k$ iff U has a SC of size $\leq k$:

- Vertex cover S has nodes incident to all edges in E. Thus, the sets S_v, where $v \in S$, must "cover" E (their union is U).
- Given a set cover for U, we take the vertex of each set of the set cover. The result will be a vertex cover. □

Caution 7.2. Note that for every $e \in E$, there are exactly two sets S_u and S_v such that $e \in S_u$ and $e \in S_v$ (where $e = (u, v)$). A reduction from Set Cover to Vertex Cover can therefore not work, since in Set Cover an element of the universe can belong to an arbitrary number of sets. So Vertex Cover is a *special case* of Set Cover where each element of the universe belongs to exactly two sets. ◀

7.1.3 Reduction by encoding with gadgets

Literal: boolean variable or its negation

Clause: disjunction of literals

Definition 7.6 (Disjunctive Normal Form)**.** A formula Φ is in *disjunctive normal form* (DNF) if there exist literals $L_{i,j}$ such that

$$\Phi = \bigvee_{i=1}^{n} \left(\bigwedge_{j=1}^{m_i} L_{i,j} \right) = (L_{1,1} \wedge \ldots \wedge L_{1,m_1}) \vee \ldots \vee (L_{n,1} \wedge \ldots \wedge L_{n,m_n})$$

✍

Definition 7.7 (Conjunctive Normal Form)**.** A formula Φ is in *conjunctive normal form* (CNF) if there exist literals $L_{i,j}$ such that

$$\Phi = \bigwedge_{i=1}^{n} \left(\bigvee_{j=1}^{m_i} L_{i,j} \right) = (L_{1,1} \vee \ldots \vee L_{1,m_1}) \wedge \ldots \wedge (L_{n,1} \vee \ldots \vee L_{n,m_n})$$

✍

Problem 7.8 (SAT)**.** Given a boolean formula Φ, does it have a satisfying truth assignment? ✍

Remark 7.3. SAT is trivial if the formulas are restricted to those in DNF, that is, they are a disjunction of conjunctions of literals. Such a formula is indeed satisfiable if and only if at least one of its conjunctions is satisfiable, and a conjunction is satisfiable if and only if it does not contain both x and $\neg x$ for some variable x. This can be checked in linear time.

But it can take exponential time and space to convert a general SAT problem into an equisatisfiable DNF.

Furthermore, checking if a CNF is a tautology is also trivial, since a formula in CNF is a tautology iff all its clauses are tautologies, and a clause of a CNF is a tautology iff it contains both x and $\neg x$ for some x. On the other hand, for a formula in DNF, the tautology problem does not "localize" in the same way; the relation between the clauses is important here. ◀

Caution 7.4. While for a DNF formula both checking satisfiability as well as unsatisfiability are easy (both in P), the problem of checking unsatisfiability for a CNF formula is coNP-complete. Checking tautology for a DNF formula is coNP-complete as well. See Section 7.3.3. ◀

Problem 7.9 (CNF-SAT)**.** Given CNF formula Φ, does it have a satisfying truth assignment? ✍

Remark 7.5. Converting a general SAT problem into an equisatisfiable CNF formula can be done in polynomial time and space (using auxiliary variables if necessary), using the *Tseitin transformation*. ◀

We have SAT $\equiv_P$ CNF-SAT.

Problem 7.10 (3-SAT)**.** SAT where each clause has exactly 3 literals (each corresponding to a different variable). ✍

Claim 7.5. 3-SAT $\leq_P$ Independent Set. ◁

Proof. Given an instance Φ of 3-SAT with k clauses, we construct an instance (G, k) of Independent Set, which has an independent set of size k iff Φ is satisfiable.

Construction. G contains one vertex for each literal (3 vertices for each clause). Connect the 3 literals of one clause in a triangle (one clause - one triangle). Connect each literal to each of its negations (*conflict links*). If Φ has k clauses, G has k triangles.

Remarks.

- The reason for conflict links is that that x and $\neg x$ cannot both be in the same independent set. Enforces the restriction that only one of x and $\neg x$ can be set to 1.
- Reduction does not attempt to solve the 3-SAT problem.

Claim: G has an independent set of size $k = |\Phi|$ iff Φ is satisfiable.

'$\Rightarrow$': Let S be an independent set of size k. S can contain ≤ 1 vertex from each triangle. Since $|S| = k$, and there are k triangles, S must contain exactly one vertex from each triangle. Set these literals to true, and any other variables in a consistent way. (Then at least 1 literal is set to true in each clause.) Truth assignment is consistent (not both x and $\neg x$ can be in S because they are connected by a conflict link in G). All clauses are satisfied. Thus, Φ is satisfiable.

'$\Leftarrow$': Suppose Φ has a satisfying assignment. Each clause of Φ is true in this assigment. Select one true literal from each clause (one vertex from each triangle) and put it in a set S. S is an independent set of size k by construction, because no two nodes in S belong to the same triangle, and no two conflicting literals can be in S (not both can be true). Thus, there can be no edge connecting any two vertices in S, because non-triangle edges connect only conflicting literals $x_i, \neg x_i$. □

Decision problem. Does there exist a vertex cover of size ($\leq$) k?

Search problem. Find a vertex cover of minimum cardinality.

Decision problem: Yes/No answer. It is easier than search problem but not necessarily fundamentally easier.

Why deal with decision problems?

- In answering complexity questions, our goal is to show that a problem cannot be solved efficiently. If we can do this for the simpler decision problem, then the same is true for the more general optimization problem. (Decision problems are simpler to establish proofs)
- Typically, if we can solve the decision problem efficiently, we can construct an efficient solution for the optimization problem too, i.e. optimization problem $\equiv_P$ decision problem for many problems of interest.

7.2 Clique, Vertex Cover, Dominating Set

7.2.1 Clique

Problem 7.11 (Clique)**.** Given a graph $G = (V, E)$ and an integer k, does G have a subset $V' \subseteq V$ of size $|V'| \geq k$ that is complete (for each $u, v \in V'$, $(u, v) \in E$)? ✍

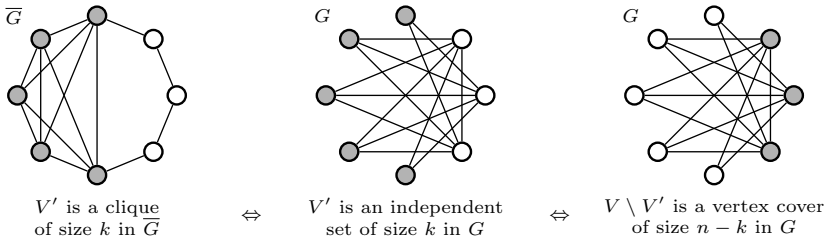
Claim 7.6. Clique $\equiv_P$ Independent-Set $\equiv_P$ Vertex-Cover. ◁

Lemma 7.7. Let $G = (V, E)$ be a graph with $n := |V|$ vertices, and a subset $V' \subseteq V$ of size $|V'| = k$. The following are equivalent:

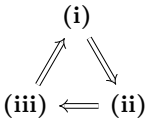
(i) V' is a clique of size k in the complement graph $\overline{G}$.

(ii) V' is an independent set of size k in G.

(iii) $V \setminus V'$ is a vertex cover of size $n - k$ in G. ◁



Proof. We prove the implications in the following triangle:



(i) ⇒ (ii):

If V' is a clique in $\overline{G}$, then for each $u, v \in V'$, (u, v) is an edge of $\overline{G}$ implying that (u, v) is not an edge of G, implying that V' is an independent set in G.

(ii) ⇒ (iii):

Claim 7.3.

(iii) ⇒ (i):

If $V \setminus V'$ is a vertex cover for G, then for any $u, v \in V'$ there is no edge (u, v) in G (because not covered by $V \setminus V'$), implying that there is an edge (u, v) in $\overline{G}$, implying that V' is a clique in $\overline{G}$.

This closes the triangle of implications and proves Lemma 7.7. □

7.2.2 Dominating Set

Definition 7.12 (Dominating Set)**.** A dominating set in a graph $G = (V, E)$ is a subset of vertices $V' \subseteq V$ such that every vertex in the graph is either in V' or is adjacent to some vertex in V'. ✍

Problem 7.13 (Dominating Set)**.** Given a graph $G = (V, E)$ and an integer k, does G have a dominating set of size k? ✍

Dominating Set is NP-complete. Reduction from Vertex Cover. VC $\leq_P$ DS.

Caution 7.6. Note:

- If G has no isolated vertices, a vertex cover for G is a dominating set for G.
- But a dominating set for G need not be a vertex cover ◀

Transformation from VC to DS.

- Given (G, k) an instance of VC, produce (G', k') an instance of DS such that G has a vertex cover of size k iff G' has a dominating set of size k'.
- Let V' be the VC of G. Let V'' be the DS of G'.
- Observation. If G has isolated vertices we must include them in V''.
- Map the notion of "incident edge" to notion of "adjacent vertex". Map edges to vertices.

So, given (G, k) for VC create graph G' as follows:

- Initially, $G' = G$.
- For each edge (u, v) in G we create a new vertex w_{uv} in G'.
- Add edges (u, w_{uv}) and (v, w_{uv}) in G'.
- Let I denote the set of isolated vertices in G.
- Set $k' = k + |I|$.
- Output (G', k').

Lemma 7.8. G has a vertex cover of size k iff G' has a dominating set of size k'. ◁

Proof. '⇒' (if V' is a vertex cover for G, then $V'' = V' \cup I$ is a dominating set for G'): Indeed all vertices of G' are either in V'' or adjacent to V''. $|V''| = |V'| + |I| \leq k + |I| = k'$.

'⇐' (if G' has a dominating set V'' of size k' then G has a vertex cover of size $k = k' - n_I$): All isolated vertices of G' must be in V''. Let $V''' = V'' \setminus V_I$ be the remaining $k = k' - n_I$ vertices. Modify V''' so that it contains no middle vertex (substitute the "middle vertex" by any adjacent regular vertex. Still dominates the same vertices). Let V' be the resulting set. V' must be a VC of G (if an edge not covered then middle vertex not adjacent to DS). V' must be a vertex cover for G. If there is a edge (u, v) in G not covered by V' (neither u nor v is in V'), then the middle vertex w_{uv} would not be adjacent to any vertex of V'' in G'. This contradicts that V'' was a dominating set for G'. □

7.3 NP-Completeness

7.3.1 Complexity Classes

Complexity class: collection of languages (decision problems), which are similar in terms of how hard it is to determine membership (solve).

Recall Definition 7.1: P is the set of all languages (i.e., decision problems) for which membership can be *determined* in (worst case) polynomial time.

P is a complexity class

Problem	Description	Algorithm
Multiple	Is x a multiple of y?	Grade school division
RelPrime	Are x and y relatively prime?	Euclid (300 BCE)
Primes	Is x prime?	AKS (2002)
Edit-Distance	Is the edit distance between x and y less than 5?	Dynamic programming
LSolve	Is there a vector $\vec{x}$ that satisfies $\underline{\boldsymbol{A}}\vec{x} = \vec{b}$?	Gauss-Edmonds elim.

Not all languages are in P.

Definition 7.14 (Simple Cycle)**.** Given a graph $G = (V, E)$, a simple cycle is a closed path that visits each vertex at most once. ✍

Problem 7.15 (Hamiltonian Cycle)**.** Given a graph $G = (V, E)$, does G contain a Simple Cycle that visits each vertex exactly once? ✍

or in terms of languages:

$$\text{HC} = \{G \mid G \text{ has a simple cycle that visits all vertices}\}$$

There is no known polynomial time algorithm for Problem 7.15.

NP: non-deterministic polynomial time.

alternative (equivalent) definition (to avoid non-determinism): set of all languages (i.e., decision problems) for which membership can be *verified* in (worst case) polynomial time.

A decision problem (language recognition problem) may be hard to solve, but, given a string y (a potential solution) it may be easy to verify that y corresponds to a solution, a yes answer of the decision problem.

Example 7.3 (Hamiltonian Cycle Verification)**.** Hard to find a Hamiltonian cycle in a graph. But suppose that someone gives a permutation of vertices. then it is easy to check if this is a legal cycle that visits all vertices in the graph exactly once. So it is easy to *verify* a Hamiltonian Cycle.

A sequence of vertices that forms a HC is called a certificate. ◀

Note that not all decision problems are easy to verify:

- Given a graph G, does G have a *unique* Hamiltonian cycle?

$$\text{UHC} = \{G \mid G \text{ has a unique Hamiltonian cycle}\}$$

- Given a graph G, does G have *no* Hamiltonian cycle?

$$\overline{\text{HC}} = \{G \mid G \text{ has no Hamiltonian cycle}\}$$

No known polynomial time verification algorithm for either.

Definition 7.16 (Certificate)**.** Piece of information (a string t) which helps to verify that an instance x of problem X has a solution, i.e. that $x \in X$. ✍

Definition 7.17 (Certifier)**.** An algorithm $C(s,t)$ is a *certifier* (also called *verification algorithm*), if for every string s with $s \in X$ (s is a yes instance of X), iff there exists a string t such that $C(s,t) = \text{Yes}$.

- s: input string (a possible instance of problem X)
- t: certificate
- for a poly-time certifier, $|t| \leq p(|s|)$ for some polynomial $p(\cdot)$

Note, $C(s,t) = \text{No}$ when t is not a certificate. C is a Yes/No algorithm. ✍

Definition 7.18 (NP)**.** decision problems for which there exists a *poly-time* certifier, i.e. $C(s,t)$ is a poly-time algorithm and $|t| \leq p(|s|)$ for some polynomial $p(\cdot)$. ✍

Problem 7.19 (Composites)**.** Given an integer s, is s composite (i.e., not prime)? ✍

Certificate. A non-trivial factor t of s (i.e., $1 < t < s$ and $t \mid s$). Note that such a certificate exists iff s is composite. Moreover $|t| \leq |s|$.

Algorithm 7.1 Certifier for Composites

```
1: function BooleanC(s, t)
2:     if t ≤ 1 ∨ t ≥ s then return false
3:     else if s is a multiple of t then return true
4:     else return false
```

Conclusion. Composites is in NP.

Fact 7.9. For SAT a Certificate is a satisfying assignment of variables and the Certifier checks that each clause in Φ has at least one true ltieral. ◁

Conclusion. SAT is in NP.

Fact 7.10. For HC a Certificate is a permutation of the n nodes (which is a Ham. cycle if G has one) and the Certifier checks that the certificate contains each node in V exactly once and that there is an edge between each pair of adjacent nodes in the permutation. ◁

Conclusion. Hamiltonian Cycle is in NP.

P: Decision problems for which there is a poly-time algorithm.

NP: Decision problems for which there is a poly-time certifier.

EXP: Decision problems for which there is a exponential-time algorithm.

Claim 7.11. $\text{P} \subseteq \text{NP}$. ◁

Proof. Consider any problem X in P. There exists a poly-time algorithm $A(s)$ that solves X: returns yes, if $s \in X$, and returns no, otherwise. Then there exists a poly-time certifier. Certificate: $t = \epsilon$ (empty string), certifier $C(s,t) = A(s)$. Thus, X in NP. □

Claim 7.12. $\text{NP} \subseteq \text{EXP}$. ◁

Proof. Consider any problem X in NP. Since X is in NP, there exists a polynomial time certifier $C(s,t)$ for X. Let $p(|s|)$ be the polynomial time complexity of $C(s,t)$; $|t| \leq p(|s|)$.

We construct an algorithm that solves X on instance s, based on $C(s,t)$ (use brute force):

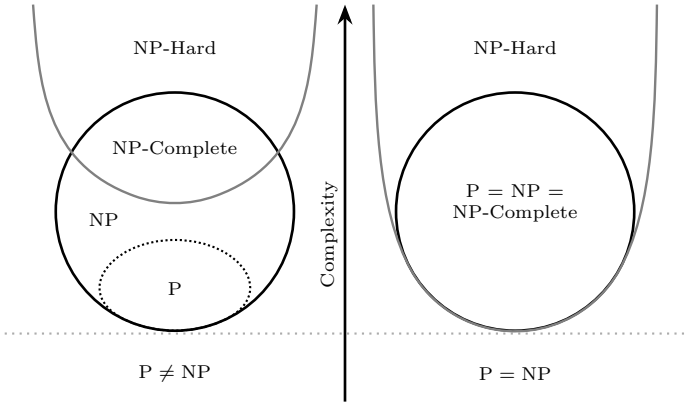
- Generate all strings t with $|t| \leq p(|s|)$. Run $C(s,t)$ on each string t.
- If $C(s,t)$ returns `yes` for any such string t, stop and return `yes`.
- There is an exponential number of possible such strings t, given $p(|s|)$. Thus, this is an exponential-time algorithm.
- After $\leq p(|s|)$ rounds the algorithm stops with a `yes` or `no` answer. □

Open Question 7.13 (P vs NP)**.** Is P = NP, i.e. is the decision problem as easy as the verification problem? ◁

The consesus is probably no. There is $1M prize for proof. would break RSA

If yes: efficient algorithms for 3-COLOR, TSP, FACTORING, SAT, ...

If no: no efficient algorithms for 3-COLOR, TSP, FACTORING, SAT, ...



Definition 7.20 (NP-Hard)**.** A problem Y with the property that for every problem X in NP, $X \leq_P Y$. ✍

Definition 7.21 (NP-Complete)**.** A problem Y that is in NP and is NP-hard. ✍

If one NP-complete problem is found to have a poly time algorithm, then all problems in NP have

Theorem 7.14. Suppose Y is an NP-complete problem. Y is solvable in polynomial time iff P = NP. ◁

Proof. '⇐' (if): If P = NP, then Y can be solved in poly-time since Y is in P.

'⇒' (only if): Suppose Y can be solved in poly-time. Let X be any problem in NP. Since $X \leq_P Y$ and Y can be solved in poly-time, we can solve X in poly-time. Thus, X in P, NP ⊆ P. We already know P ⊆ NP. Thus, P = NP. □

7.3.2 NP-complete Problems

Fundamental Question. Do there exist "natural" NP-complete problems? ◁

Problem 7.22 (CIRCUIT-SAT)**.** Given a combinational logic circuit built out of AND, OR, and NOT gates, is there a way to set the input so that the output is 1? ✍

Theorem 7.15 (Cook 1971, Levin 1973)**.** CIRCUIT-SAT is NP-complete. ◁

Proof (sketch)**.** Membership in NP is immediate, so we focus on NP-hardness:

- Any algorithm that takes a fixed number of bits n as input and produces a yes/no answer can be represented by a circuit. Further, if algorithm takes poly in n time, then circuit has poly-size.
  - sketchy part of proof: fixing the number of bits is important, and reflects basic distinction between algorithms and circuits
- Consider some problem X in NP. It has a poly-time certifier $C(s,t)$. To determine whether s is in X, need to know if there exists a certificate t of length $\leq p(|s|)$ such that $C(s,t) = \texttt{yes}$.
- $C(s,t)$ is an algorithm on $|s|+p(|s|)$ bits (input s, certificate t). For fixed input length $|s|$, $C(\cdot,\cdot)$ is an algorithm of fixed input length, so we convert it into a poly-size circuit family such that K_s is satisfiable iff $C(s,t) = \texttt{yes}$ for some t.
  - First $|s|$ bits are hard-coded with s; remaining $p(|s|)$ bits represent t.
- $X \leq_p$ CIRCUIT-SAT: Given an arbitrary instance s of X, transform s (as sketched above) into a poly-size circuit K_s (in poly time), which is satisfiable iff s is yes instance of X. □

Remark 7.7. Once we establish first "natural" NP-complete problem (SAT), others fall like dominoes. ◀

Recipe to establish NP-completeness of a problem Y:

1. Show that Y is in NP
2. Choose an NP-complete problem X
3. Make a reduction: prove that $X \leq_P Y$

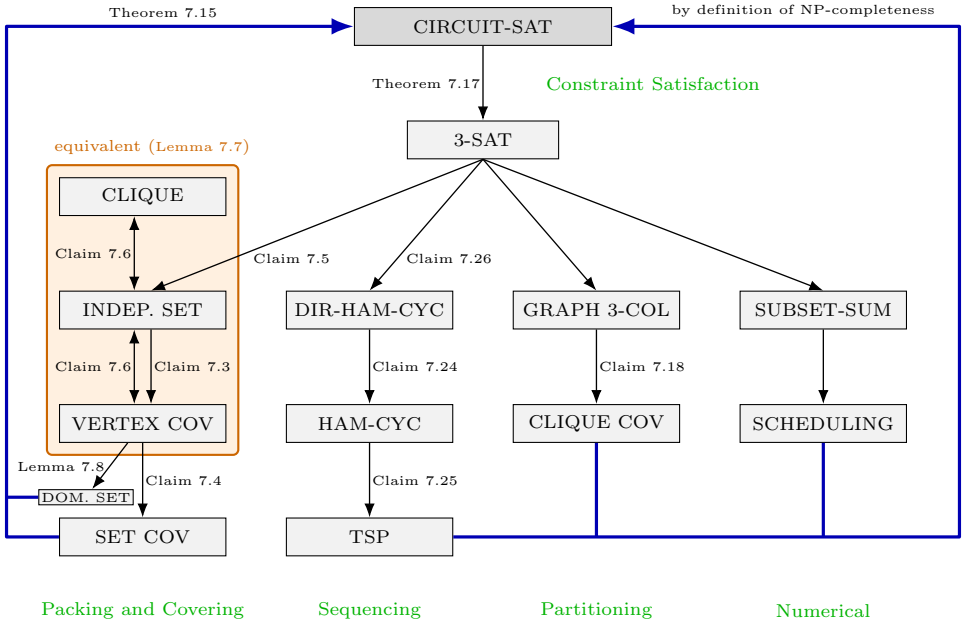
Claim 7.16. If X is an NP-complete problem and $X \leq_p Y$, then Y is NP-hard. Further, if Y is in NP, then Y is NP-complete. ◁

Proof. Let W be any problem in NP. Then $W \leq_p X \leq_p Y$. By transitivity, $W \leq_p Y$. Thus, Y is NP-hard. □

Theorem 7.17. 3-SAT is NP-complete. ◁

Proof. It suffices to show that CIRCUIT-SAT $\leq_p$ 3-SAT, since 3-SAT is in NP. Let K be any circuit. Create a 3-CNF formula K' which is satisfiable iff K is satisfiable. There is a mechanical way to do it (skipped). □

All problems below are NP-complete and polynomially reduce to one another:



Basic categories of NP-complete problems and paradigmatic examples:

- Packing problems: SET PACKING, INDEPENDENT SET
- Covering problems: SET COVER, VERTEX COVER
- Constraint satisfaction problems: SAT, 3-SAT
- Sequencing problems: HAM-CYCLE, TSP
- Partitioning problems: 3D-MATCHING, 3-COLOR
- Numerical problems: SUBSET-SUM, KNAPSACK

In practice, most problems are either known to be in P or known to be NP-complete. Notable exeptions: Factoring, graph isomorphism, Nash equilibrium.

Problem 7.23 (3Col)**.** Given a graph G, can each of its vertices be labeled with one of three different "colors" such that no two adjacent vertices have the same label?

- arises in various partitioning problems (adjacent object not in same group)
- planar graphs can be colored with 4 colors (well known)
- determining if 3 colors are possible is hard (even for planar graphs)
- 3Col is known to be NP-complete ✍

Problem 7.24 (Clique Cover)**.** Given a graph $G = (V, E)$ and an integer k, can we partition the vertex set into k cliques?

- every vertex needs to be in exactly one clique!
- Problem 7.24 arises in clustering.
- 3Col $\leq_P$ CCov ✍

Claim 7.18. A graph $G = (V, E)$ is 3-colorable iff its complement $\overline{G} = (V, \overline{E})$ has a clique cover with $k = 3$ cliques. ◁

Proof. '⇒': Suppose G is 3-colorable. Let V_1, V_2, V_3 be the 3 color classes. Every pair of distinct vertices in V_i are not adjacent in G. Thus, they are adjacent in $\overline{G}$. So, V_i forms a complete subgraph in $\overline{G}$. Thus, this is a clique cover of size 3 for $\overline{G}$.

'⇐': Suppose $\overline{G}$ has a clique cover of size 3, denoted V_1, V_2, V_3. Give the vertices of V_i color i, $i = 1, 2, 3$. No two vertices in V_i are adjacent in G. Thus, this is a legal coloring for G. □

7.3.3 co-NP and the Asymmetry of NP

Asymmetry of NP: We only have short proofs of `yes` instances. We do not necessarily have proofs for `no` inst ances.

Example 7.4 (SAT vs TAUTOLOGY)**.** Can prove a CNF formula is satisfiable by giving a truth assignment (Fact 7.9). How could we prove that a formula is *not* satisfiable, i.e., a contradiction? How can we prove that a formula is always true, i.e., a tautology? ◀

Example 7.5 (HAM-CYCLE vs NO-HAM-CYCLE)**.** Can prove that a graph is Hamiltonian by giving a Hamiltonian cycle (Fact 7.10). How can we prove that a graph is *not* Hamiltonian? ◀

The complement of SAT, i.e. $\overline{\text{SAT}}$, is the set of all Boolean CNF formulas that are not satisfiable, i.e., the CONTRADICTION (formulas that are always false).

TAUTOLOGY $\equiv_P$ CONTRADICTION: Φ is a tautology iff $\neg\Phi$ is a contradiction.

Thus, $\overline{\text{SAT}} \equiv$ CONTRADICTION $\equiv_P$ TAUTOLOGY.

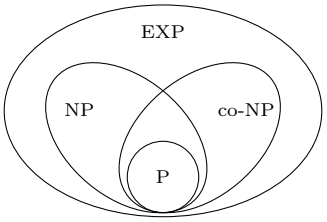
Definition 7.25 (Complement)**.** Given a decision problem X, its complement $\overline{X}$ is the same problem with the `yes` and `no` answers reversed, i.e., s in X iff s not in $\overline{X}$. ✍

Definition 7.26 (co-NP)**.** The Complements of decision problems in NP, i.e., a problem X belongs to co-NP iff $\overline{X}$ belongs to NP. ✍

Examples NP	Examples co-NP
SAT	TAUTOLOGY
HAM-CYCLE	NO-HAM-CYCLE
COMPOSITES	PRIMES

Open Question 7.19. Does NP = co-NP?

- Do `yes` instances have succinct certificates iff `no` instances do?
- Consensus opinion: no.
- Easy to get a simple short certificate for most problems in NP but to get a no certificate is often not easy. ◁



Claim 7.20. P = co-P. ◁

Proof. If we have an algorithm A that decides X in poly-time, we can design an algorithm $\overline{A}$ that decides $\overline{X}$ in poly-time: Run A on input x and flip its answer. □

Theorem 7.21. If NP ≠ co-NP, then P ≠ NP. ◁

Proof (idea)**.** P is closed under complement (Claim 7.20). If P = NP, then NP is also closed under complement, i.e., NP = co-NP. Contrapositive yields the result. □

Good characterization. [Edmonds 1965] NP ∩ co-NP.

- If problem X is in both NP and co-NP, then:
  - for `yes` instance, there is a succinct certificate
  - for `no` instance, there is a succinct disqualifier (certificate)
- Provides conceptual leverage for reasoning about a problem.

Example 7.6. Given a bipartite graph, is there a perfect matching?

- if yes, can exhibit a perfect matching.
- if no, can exhibit a set of nodes S such that $|N(S)| < |S|$ (Theorem 6.14).
- Bipartite perfect matching is in NP ∩ co-NP. ◀

Observation 7.8. P ⊆ NP ∩ co-NP. ◀

Open Question 7.22. Does P = NP ∩ co-NP?

- mixed opinions.
- many examples where problem found to have non-trivial good characterization, but only years later discovered to be in P.
  - linear programming [Khachiyan, 1979]
  - primality testing [Agrawal-Kayal-Saxena, 2002] ◁

Fact 7.23. Factoring is in NP ∩ co-NP, but not known to be in P.[1] ◁

[1] if poly-time algorithm for factoring, can break RSA cryptosystem.

7.3.4 Sequencing Problems

Example 7.7. Vertices and faces of a dodecahedron have a Hamiltonian cycle. A bipartite graph with odd number of vertices cannot have a Hamiltonian cycle. ◀

Problem 7.27 (DIR-HAM-CYCLE)**.** Given a digraph $G = (V, E)$, does G contain a directed Simple Cycle that visits each vertex exactly once? ✍

Claim 7.24. DIR-HAM-CYCLE $\leq_P$ HAM-CYCLE. ◁

Proof (gadget construction)**.** Given a directed graph $G = (V, E)$ with $n = |V|$ nodes, construct an undirected graph G' with $3n$ nodes as follows: Substitute each node v in G by three nodes $v_{\text{in}}, v_{\text{mid}}, v_{\text{out}}$ in G'. Each directed edge (u, v) in E becomes an undirected edge $(u_{\text{out}}, v_{\text{in}})$ in G'.

Now G has a directed Hamiltonian cycle iff G' has a Hamiltonian cycle.

- '⇒': Suppose G has a directed Hamiltonian cycle. Then G' has an undirected Hamiltonian cycle (same order).
- '⇐': Suppose G' has an undirected Hamiltonian cycle. must visit nodes either in order $v_{\text{in}} \to v_{\text{mid}} \to v_{\text{out}}$ or $v_{\text{out}} \to v_{\text{mid}} \to v_{\text{in}}$. The v_{mid} nodes in the cycle make up a directed Hamiltonian cycle in G, or reverse of one. □

Problem 7.28 (Traveling Salesman)**.** Given a set of n cities and a pairwise distance function $d(u, v)$, is there a tour of length $\leq D$ that visits each city exactly once and returns to the starting city? ✍

Claim 7.25. HAM-CYCLE $\leq_P$ TSP. ◁

Proof. Given an instance $G = (V, E)$ of Hamiltonian Cycle, create $n = |V|$ cities with distance function

$$d(u, v) = \begin{cases} 1 & \text{if } (u, v) \in E \\ 2 & \text{otherwise} \end{cases}$$

Remark. TSP instance in reduction satisfies $\triangle$-inequality: $d(u, v) \leq 2 \leq d(u, w) + d(w, v)$

TSP instance has tour of length $\leq n$ iff G is Hamiltonian.

- '⇐': If G has a Hamiltonian cycle, then it defines a tour of length n in the TSP instance.
- '⇒': Suppose there is a tour T of length $\leq n$ in the TSP instance. The length of T is the sum of n terms, each of which has cost at least 1 (1 or 2). Since the length of T is at most n, each term must have cost 1. Thus, each pair of consecutive cities in T must be connected by an edge in G. Thus, the order of T gives a Hamiltonian cycle in G. □

Claim 7.26. 3-SAT $\leq_P$ DIR-HAM-CYCLE. ◁

Proof. Let Φ be a 3-CNF formula with n variables $x_1, \ldots, x_n$ and k clauses $C_1, \ldots, C_k$, i.e.,

$$\Phi = \bigwedge_{j=1}^{k} C_j = \bigwedge_{j=1}^{k} (L_{j,1} \vee L_{j,2} \vee L_{j,3}),$$

where each literal $L_{j,m}$ is either x_i or $\neg x_i$ for some $i \in \{1, \ldots, n\}$.

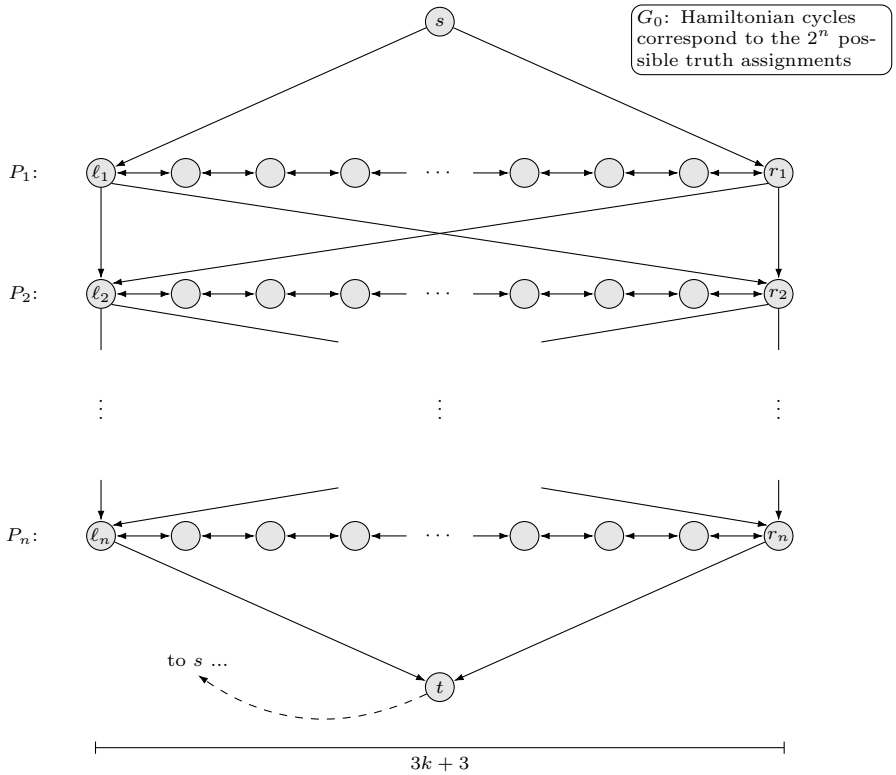
We construct a directed graph G as follows.

Let $b := 3k + 3$. For each variable x_i construct a directed path

$$P_i = (v_{i,1}, v_{i,2}, \ldots, v_{i,b-1}, v_{i,b})$$

where for every $q \in \{1, \ldots, b-1\}$ we add both directed edges $(v_{i,q}, v_{i,q+1})$ and $(v_{i,q+1}, v_{i,q})$. Let $\ell_i := v_{i,1}$ and $r_i := v_{i,b}$.

For $i = 1, \ldots, n-1$, add all four directed edges $\ell_i \to \ell_{i+1}$, $\ell_i \to r_{i+1}$, $r_i \to \ell_{i+1}$, $r_i \to r_{i+1}$. Add vertices s, t and directed edges $s \to \ell_1$, $s \to r_1$, $\ell_n \to t$, $r_n \to t$, and $t \to s$. Let G_0 denote the resulting graph:



G_0 has exactly 2^n different Hamiltonian cycles, corresponding to the n independent choices of direction for traversing each path P_i.

This naturally models the n independent choices of how to set each variable $x_1, \ldots, x_n$ to `true` or `false`, and hence the 2^n truth assignments. Thus, we identify each Hamiltonian cycle H_0 in G_0 uniquely with a truth assignment a as follows: If H_0 traverses path P_i from left to right, set $a(x_i) := 1$; if H_0 traverses path P_i from right to left, set $a(x_i) := 0$.

Lemma. Every Hamiltonian cycle in G_0 traverses each path P_i entirely from ℓ_i to r_i or entirely from r_i to ℓ_i. Conversely, every choice of directions for $P_1, \ldots, P_n$ induces a unique Hamiltonian cycle in G_0. ◁

Proof. Since the only edges incident to internal vertices of P_i are the two path edges to their predecessor/successor, a Hamiltonian cycle cannot enter or leave P_i at an internal vertex. Hence it must enter at one endpoint and leave at the other, traversing P_i completely in one direction. The endpoint connections guarantee that all 2^n direction choices are feasible and independent. □

Next we add nodes to model the constraints imposed by the clauses of Φ.

For each clause C_j, add a vertex c_j. We reserve positions $3j$ and $3j+1$ on each path P_i for clause C_j. For each literal $L \in C_j$:

- if $L = x_i$, add edges $(v_{i,3j}, c_j)$ and $(c_j, v_{i,3j+1})$
- if $L = \neg x_i$, add edges $(v_{i,3j+1}, c_j)$ and $(c_j, v_{i,3j})$

This completes the construction of G.

Lemma (Splicing)**.** Let H be a Hamiltonian cycle containing a directed edge $p \to q$. If G contains edges $p \to c_j$ and $c_j \to q$, then replacing $p \to q$ in H by $p \to c_j \to q$ yields another Hamiltonian cycle. ◁

Proof. The replacement preserves indegree and outdegree 1 at all vertices and introduces no repetitions. □

Now Φ is satisfiable iff G has a Hamiltonian cycle.

- '$\Rightarrow$': Suppose Φ is satisfiable, and let a be a satisfying assignment. By the first Lemma, a induces a Hamiltonian cycle H_0 in G_0.

  Fix a clause C_j. Choose a literal $L \in C_j$ satisfied by a (at least one must exist by assumption that Φ is satisfiable).

  - If $L = x_i$, then $a(x_i) = 1$, so H_0 traverses P_i left-to-right and in particular uses the edge $v_{i,3j} \to v_{i,3j+1}$. By construction, G contains edges $v_{i,3j} \to c_j$ and $c_j \to v_{i,3j+1}$, so by the second Lemma we can splice c_j into the tour between $v_{i,3j}$ and $v_{i,3j+1}$.
  - If $L = \neg x_i$, then $a(x_i) = 0$, so H_0 traverses P_i right-to-left and in particular uses the edge $v_{i,3j+1} \to v_{i,3j}$. By construction, G contains edges $v_{i,3j+1} \to c_j$ and $c_j \to v_{i,3j}$, so by the second Lemma we can splice c_j into the tour between $v_{i,3j+1}$ and $v_{i,3j}$.

  Do this once for each clause C_j. The resulting directed cycle visits every vertex exactly once, hence is a Hamiltonian cycle of G.

- '$\Leftarrow$': Suppose G has a Hamiltonian cycle H. Fix j. Since c_j has indegree and outdegree 1 in H, the cycle contains a subpath $p \to c_j \to q$.

  Moreover, if H enters c_j from $v_{i,3j}$ then it must leave to $v_{i,3j+1}$: otherwise $v_{i,3j+1}$ would remain unvisited and would then have only one unvisited neighbor $v_{i,3j+2}$ available, so the tour will not be able to visit this node while remaining Hamiltonian. Symmetrically, if H enters from $v_{i,3j+1}$ then it must leave to $v_{i,3j}$.

  By construction of G, necessarily $(p, q) = (v_{i,3j}, v_{i,3j+1})$ for some i (if c_j encodes a positive occurrence x_i) or $(p, q) = (v_{i,3j+1}, v_{i,3j})$ (if it encodes a negative occurrence $\neg x_i$). In either case, the shortcut edge $p \to q$ is a path edge of P_i and hence belongs to G_0. Contract $p \to c_j \to q$ to $p \to q$. Repeating this for all $j = 1, \ldots, k$ yields a Hamiltonian cycle H_0 of G_0.

  By the first Lemma, H_0 induces an assignment a by setting $a(x_i) = 1$ iff H_0 traverses P_i left-to-right. Fix a clause C_j. Since H visits c_j, it must splice through c_j using some path P_i in the direction that matches the corresponding literal in C_j; hence that literal is true under a. Therefore every clause is satisfied by a, so Φ is satisfiable.

Complexity: The graph has $O(n \cdot k)$ vertices and edges and is constructible in polynomial time. □

7.4 Approximation Algorithms

We saw all these problems that are NP-complete and reduce one to another. So they are the same hard. They appear extremely often in practice.

So how do we cope with NP-completeness?

- **brute-force search**: viable only for small input sizes (e.g. $n \leq 20$).
- **heuristics**: strategy for producing a valid solution, but no guarantee on how close is to optimal
- **general search algorithms**: powerful techniques for solving general combinatorial optimization problems:
  - branch-and-bound: breaking problem down into smaller subproblem and using a bounding function
  - metropolis-hastings: Markov chain Monte Carlo method for obtaining a sequence of random samples from a probability distribution from which direct sampling is difficult
  - simulated annealing: models the physical process of heating a material and then slowly lowering the temperature to decrease defects, thus minimizing the system energy
  - genetic algorithms: metaheuristic used to generate high-quality solutions to optimization and search problems via biologically inspired operators such as selection, crossover, and mutation

performance varies considerably from problem to problem and instance to instance

- **approximation algorithms**: algorithm that runs in polynomial time and produces a solution that is guaranteed to be within some factor of the optimal solution

Performance Bounds:

- most NP-complete problems states as decision problems (because of theoretical resons involving complexity)
- they are natural optimization problems, e.g.
  - Vertex Cover: find vertex cover of minimum size
  - Clique: find the clique of maximum size
- an approximation algorithm returns a legitimate answer, but not necessarily one of optimal size

Measuring how good an approximation algorithm is:

- define the *performance ratio* of an approximation
- given an instance I of our problem,
  - let $C(I)$ be the cost of solution produced by the approximation algorithm
  - let $C^*(I)$ be the cost of optimal solution
- for a *minimization problem*: $C(I)/C^*(I) \geq 1$
- for a *maximization problem*: $C^*(I)/C(I) \geq 1$

Definition 7.29 (Performance Ratio Bound)**.** We say that an approximation algorithm achieves performance ratio bound $\rho(n)$ if

$$\max_I \left(\frac{C(I)}{C^*(I)}, \frac{C^*(I)}{C(I)} \right) \leq \rho(n) \tag{7.3}$$

for all instances I of size $|I| = n$ of the problem. Note that $\rho(n) \geq 1$ with equality iff the approximate solution is the true optimal solution. ✍

The Performance Ratio Bound is an upper bound for the worst-case performance.

NP-complete problems are equivalent with respect to complexity, but their approximability varies considerably.

- some problems are *inapproximable*: no polynomial-time algorithm achieves a ratio bound $< \infty$ unless P=NP
- some can be approximated, but the ratio bound is a function of n (e.g. Set Cover can be approximated within a factor $O(\log n)$)
- some can be approximated and the ratio bound is constant (e.g. Vertex Cover can be approximated within a factor 2)
- some can be approximated arbitrarily well (e.g. Knapsack)
  - user provides a parameter $\epsilon > 0$ and the algorithm achieves a ratio bound $1 + \epsilon$
  - as ϵ approaches 0, the running time gets worse
  - if it runs in polynomial time for any fixed ϵ, it is called a *polynomial-time approximation scheme* (PTAS)

Algorithm 7.2 2-for-1 heuristic for Vertex Cover

```
1: function VC-2-Approx(G = (V, E))
2:     T ← ∅
3:     while E ≠ ∅ do
4:         Select an arbitrary edge (u, v) from E
5:         T ← T ∪ {u, v}                          ▷ add both endpoints to cover
6:         Remove from E all edges incident to either u or v
7:     return T
```

7.4.1 Vertex Cover

Vertex Cover has an approximation algorithm with ratio bound $\rho(n) = 2$.

Claim 7.27. Algorithm 7.2 achieves a performance ratio bound of $\rho(n) = 2$. ◁

Proof. Consider the set T output by Algorithm 7.2. Let A be the set of edges selected in line 4. $|T| = 2|A|$ because both endpoints of each edge of A are added to T. But also the optimal solution T^* must cover the edges in A, which are non-adjacent. Thus, $|T^*| \geq |A|$. Therefore, $|T| \leq 2|T^*|$. □

Algorithm 7.3 Greedy heuristic for Vertex Cover

```
1: function VC-Greedy-Approx(G = (V, E))
2:     T ← ∅
3:     while E ≠ ∅ do
4:         Select the vertex u of maximum degree in G
5:         T ← T ∪ {u}                             ▷ add vertex to cover
6:         Remove from E all edges incident to u
7:     return T
```

The greedy heuristic (Algorithm 7.3) does not achieve a constant performance bound, but merely one of $\Theta(\log n)$. Nevertheless, experimental studies show that it often works quite well in practice, and for "typical" graphs, it will perform better than the 2-for-1 heuristic (Algorithm 7.2).

7.4.2 Independent Set

Unfortunately, approximation factors are not preserved by our transformations.

Claim 7.28. If we apply the Algorithm 7.2 to find an Independent Set of maximum size, the performance ratio is

$$\rho(n, k) = \frac{n - k}{n - 2k}$$

which can be arbitrarily large (e.g. for $k = (n - 1)/2$, $\rho(n, k) = (n + 1)/2$). ◁

Proof. We know from Lemma 7.7 that V' is a Vertex Cover for G iff $V \setminus V'$ is a Independent Set for G. Let T^* be an minimum (i.e. optimal) Vertex Cover of size $|T^*| = k$. Then $S^* = V \setminus T^*$ is a maximum (i.e. optimal) Independent Set of size $|S^*| = n - k$. Algorithm 7.2 returns a Vertex Cover T with size $|T| \leq 2k$ (Claim 7.27). Thus $S = V \setminus T$ is an Independent Set of size $|S| \geq n - 2k$. The performance ratio is therefore $\frac{|S^*|}{|S|} = \frac{n-k}{|S|} \leq \frac{n-k}{n-2k}$. □

7.4.3 Traveling Salesman Problem

Problem 7.28 was formulated in terms of cities and distances, now we reformulate it in terms of graphs:

Problem 7.30 (TSP)**.** Given a *complete undirected graph* $G = (V, E)$ with non-negative edge weights $w(u, v)$, find a cycle that visits all vertices and has minimum cost. ✍

Often edge weights satisfy the $\triangle$-inequality: $w(u, v) \leq w(u, x) + w(x, v)$

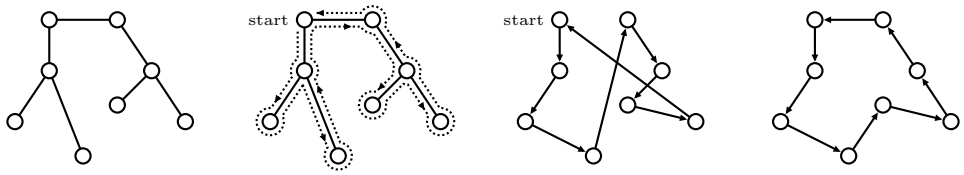
- euclidian distance
- shortest path in a graph

When cost function satisfies $\triangle$-inequality, there is an approximation algorithm for TSP with a ratio bound of $\rho(n) = 2$.

Observation 7.9. A TSP with one edge removed is just a spanning tree (not necessarily minimum). Thus, cost min TSP tour $\geq$ cost MST. ◀

Idea:

- compute a MST T of G efficiently, e.g. using Kruskal
- find some way to convert the MST T into a TSP tour H while increasing its cost by a constant factor



MST | Twice-around tour | Shortcut tour | Optimum tour

Remark 7.10.
- given a free tree, there is a tour of the tree called a *twice around tour* that traverses the edges of the tree twice, once in each direction
- this path is not simple, because it revisits vertices, but we can make it simple by *short-cutting*, i.e. skipping over already visited vertices
- the order in which vertices are visited using short-cuts is a *preorder traversal* (see Algorithms & Data Structures, chapter 'Binary Search Trees') of the MST T
- the triangle inequality assures that the path length will not increase when we take short-cuts
- in fact, any subsequence of the twice-around tour which visits each vertex exactly once will suffice (not necessarily in preorder) ◀

Algorithm 7.4 Approximation for TSP

```
1: function TSP-Approx(G = (V, E))
2:     T ← MST(G)                          ▷ e.g. using Kruskal
3:     r ← arbitrary root of T
4:     H ← list of vertices visited by a preorder walk of T starting at r
5:     return H
```

Claim 7.29. Algorithm 7.4 achieves a performance ratio bound of $\rho(n) = 2$. ◁

Proof. Let H^* be an optimal TSP tour and H be the tour returned by Algorithm 7.4. Let T be the MST computed in line 2. By Observation 7.9, $W(T) \leq W(H^*)$. The twice around tour of T has cost $2 \cdot W(T)$. By the triangle inequality, short-cuts do not increase the cost of a tour, i.e. $W(H) \leq 2 \cdot W(T)$. Thus, $W(H) \leq 2 \cdot W(T) \leq 2 \cdot W(H^*) \Rightarrow \frac{W(H)}{W(H^*)} \leq 2$. □

7.4.4 Set Cover

Set Cover can be generalized where each set S_i has a positive cost w_i and we want to compute the set cover of minimum total weight. Algorithm 7.5 can be generalized to this too.

The 2-for-1 heuristic for Vertex Cover relies on the fact that each element (each edge) appears in exactly two sets (one for each endpoint). This is not true for Set Cover!

widely believed, there is no constant factor approximation for Set Cover.

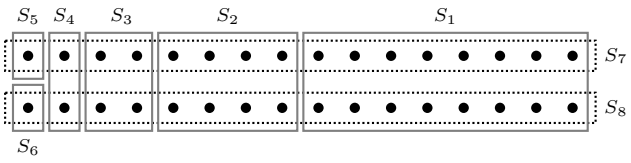
Algorithm 7.5 is a greedy heuristic that at each stage selects the set that covers the greatest number of uncovered elements.

The Greedy heuristic for Set Cover can be "fooled" into picking the wrong set over and over again, as the following example shows.

Algorithm 7.5 Greedy heuristic for Set Cover

```
1: function SC-Greedy-Approx(U, F = {S_1, ..., S_n})
2:     X ← U                                   ▷ uncovered elements
3:     C ← ∅                                   ▷ sets in the cover
4:     while X ≠ ∅ do
5:         Select S_i ∈ F that covers the most elements of X
6:         C ← C ∪ {S_i}
7:         X ← X \ S_i
8:     return C                                ▷ Theorem 7.30: |C| ≤ (1 + ln |U|) · |C*|
```

Example 7.8. Consider the following instance of Set Cover:



The optimal set cover consists of sets S_7 and S_8, each of size 16, but...

- initially the three sets S_1, S_7, S_8 each cover 16 elements
- if ties are broken in the worst possible way, Algorithm 7.5 picks S_1
- now the sets S_2, S_7, S_8 each cover 8 of the remaining elements
- again, if we choose poorly, S_2 is chosen
- and so on... ◀

We generalize Example 7.8 to any number of elements that is a power of 2.

- although there is an optimal solution 2 sets, Algorithm 7.5 will select roughly $\log m$ sets, where $m = |U|$
- thus, the Greedy heuristic for Set Cover achieves an approximation factor of roughly $\frac{\log m}{2}$ on this example
- it is possible to slighlty adjust the example such that there are no ties and yet Algorithm 7.5 has essentially the same ratio bound

Theorem 7.30. Algorithm 7.5 achieves an approximation factor of $1+\ln m$, where $m = |U|$ denotes the number of elements to be covered. ◁

Proof. Let $c = |C^*|$ be the size of an optimal set cover. At any iteration i with m_{i-1} uncovered elements remaining, some optimal set must cover at least m_{i-1}/c of them (Pigeonhole principle). Since Algorithm 7.5 chooses the set covering the maximum number of uncovered elements, it removes at least this many, leaving

$$m_i \leq m_{i-1} - \frac{m_{i-1}}{c} = m_{i-1}\left(1 - \frac{1}{c}\right)$$

elements uncovered after iteration i. Applying this recursively, we obtain

$$m_i \leq m\left(1 - \frac{1}{c}\right)^i$$

where we used $m_0 = m = |U|$.

Now denote by $g := |C|$ the number of sets returned by Algorithm 7.5 (equivalently, the number of iterations). Right before the g^{th} iterations of Algorithm 7.5 we therefore have

$$m_{g-1} \leq m\left(1 - \frac{1}{c}\right)^{g-1} = m\left(\left(1 - \frac{1}{c}\right)^c\right)^{\frac{g-1}{c}} \leq m\left(\frac{1}{\mathrm{e}}\right)^{\frac{g-1}{c}}$$

elements uncovered, where we used $(1 - 1/x)^x \leq 1/\mathrm{e}$.

Since at least one element remains uncovered before the last iteration, i.e. $m_{g-1} \geq 1$, we have

$$1 \leq m \left(\frac{1}{\mathrm{e}}\right)^{\frac{g-1}{c}} \quad \Rightarrow \quad \mathrm{e}^{\frac{g-1}{c}} \leq m \quad \Rightarrow \quad \frac{g-1}{c} \leq \ln m$$

which implies $g \leq c \ln m + 1$.

Therefore, Algorithm 7.5 returns a Set Cover of size at most $1+\ln(m)\cdot c$. Since $1+\ln(m)\cdot c \leq (1 + \ln m) \cdot c$, we have $g \leq (1 + \ln m) \cdot c$, establishing the claimed $(1 + \ln m)$-approximation ratio. □

Even though the Greedy heuristic for Set Cover has this relatively high approximation factor, it tends to perform well in practice.

- the example in which the approximation bound is $\Omega(\log m)$ is not typical of Set Cover instances
- more sophisticated approximation algorithms exist for special cases of Set Cover, but for the general problem no significantly better approximation bound is believed to be computable in polynomial time
- such an algorithm would imply that NP has quasipolynomial time algorithms (Uriel Feige, 1998), and the experts do not believe that this is the case