# ADA Final Exam Preparation

Kon Yi

December 17, 2025

# Contents

# Chapter 1

# All-pairs distances problem

## Lecture 1

13 Nov.

**Problem 1.0.1.**

- Input: an edge-weighted directed graph $G$ with $V(G) = \{1, 2, \ldots, n\}$ that has no cycle of negative weight.
- Output: $d_G(i, j)$ for all vertices $i$ and $j$ of $G$.

**Remark 1.0.1.** Here we suppose $G$ has no negative cycle to simplify the problem.

The Naive algorithm is to solve $n$ single-source distances problems directly. Hence, the time complexity for using different algorithm is:

- General edge weights: Bellman-Ford's algorithm takes $O\left(mn^2\right)$ time, which can be $\Theta\left(n^4\right)$ when $m = \Theta\left(n^2\right)$.
- Acyclic: Lawler's algorithm takes $O(mn)$ time.
- Non-negative edge weights: Dijkstra's takes $O\left(mn + n^2 \log n\right)$ time.

However, naive method takes a lot of time to compute unnecessary things, so it takes a lot of times.

## 1.1 A DP algorithm

**Definition 1.1.1.** Let $w_k(i, j)$ be the length of a shortest $(i, j)$-path having at most $k$ edges. Let it be $\infty$ if such a path does not exist.

We have

$w_1(i, j) = w(ij)$ if $(i, j)$ if a edge, otherwise $w_1(i, j) = \infty$.
$w_{n-1}(i, j) = d_G(i, j)$ since a shortest path has at most $n - 1$ edges (note that there is not negative cycle).

Hence, we have

$$w_{2k}(i, j) = \min_{1 \le t \le n} w_k(i, t) + w_k(t, j) \text{ for all } k \ge 1.$$

Also, note that even if $k \ge \frac{n}{2}$ this recurrence relation is still correct, so we can just compute $w_1(i, j)$ then $w_2(i, j)$ then $w_4(i, j)$ and so on, and after $O(\log n)$ rounds we can get the answer.

Now we analyze the time complexity. For each $(i, j)$-pair, we need $O(\log n)$ rounds, where each round takes $O(n)$ times, so each $(i, j)$-pair takes $O(n \log n)$ times. Note that we have $O\left(n^2\right)$ $(i, j)$-pairs, so it takes totally $O\left(n^3 \log n\right)$ times. This is pretty slow, but in general a little bit faster than doing Bellman-Ford's algorithm for $n$ times.

## 1.2 Floyd and Warshall's DP algorithm

**Definition 1.2.1.** Let $d_k(i,j)$ be the length of any shortest $(i,j)$-path whose internal indices are at most $k$. If there is no such a path, then let $d_k(i,j) = \infty$.

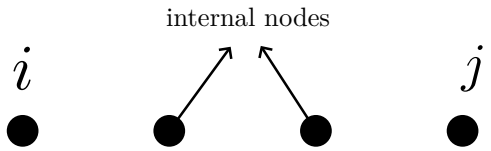



Figure 1.1: Internal Nodes

Thus, we have

$$d_0(i,j) = w(ij), \quad d_n(i,j) = d_G(i,j).$$

Hence, we can define a recurrence relation:

$$\begin{cases} d_0(i,j) = w(i,j) \\ d_{k+1}(i,j) = \min\{d_k(i,j), d_k(i,k+1) + d_k(k+1,j)\}. \end{cases}$$

Note that it corresponds to two cases: walk through $k+1$ or not. If not, then it corresponds to $d_k(i,j)$. If so, then it corresponds to $d_k(i,k+1) + d_k(k+1,j)$ since excluding $k+1$ and seperate this path into two parts, then internal nodes in each part can have indices of at most $k$.

Now we analyze the time complexity of Floyd and Warshall's DP algorithm: Fix $(i,j)$, then it takes $O(n)$ time to compute from $d_0(i,j)$ to $d_n(i,j)$, and since we have $O\left(n^2\right)$ $(i,j)$-pairs, so it take totally $O\left(n^3\right)$ time.

## 1.3 Johnson's reweighting technique

As previously seen, if $G$ is a non-negative weighted graph, then running Dijkstra's algorithm for $n$ times needs $O(mn + n\log n)$ time. Now Johnson gives a method to reweight $w$ into $\hat{w}$ s.t.

- $\hat{w}$ is non-negative
- any shortest $(i,j)$-path of $G$ w.r.t. $\hat{w}$ is a shortest $(i,j)$-path of $G$ w.r.t. $w$.

The idea of reweighting is to

- Assign a weight $h(i)$ to each vertex $i$ of $G$.
- Let $\hat{w}(i,j) = w(i,j) + h(i) - h(j)$.
- Then, for any $(i,j)$-path $P$, we have

$$\hat{w}(P) = w(P) + h(i) - h(j).$$

- Hence, $P$ is a shortest $(i,j)$-path of $G$ w.r.t. $\hat{w}$ if and only if $P$ is a shortest $(i,j)$-path of $G$ w.r.t. $w$.

**Remark 1.3.1.** The challenge is to find a vertex weight $h$ s.t. the resulting adjusted edge weight $\hat{w}$ is non-negative. If $\hat{w}$ is non-negative, then we can apply Dijkstra's algorithm to obtain all-pairs shortest path trees in $O\left(mn + n^2\log n\right)$ time.

**Algorithm 1.1:** Johnson's Technique

1 Let $H$ be obtained from $G$ by adding a new vertex 0 and adding a weight-0 edge from vertex 0 to each vertex $i$ of $G$.
2 $H$ has no negative cycle if and only if $G$ has no negative cycle.
3 Let $h(i)$ be the distance from vertex 0 to vertex $i$ in $H$. That is, $h(i) = d_H(0,i)$.
4 The vertex weight function $h$ can be obtained by Bellman-Ford in $O(mn)$ time.

**Remark 1.3.2.** $H$ has no negative cycle if and only if $G$ has no negative cycle since $G$ is directed and vertex 0 has only out degree, so any cycle in $H$ and $G$ is induced by $\{1, 2, \ldots, n\}$, which does not include 0.

**Remark 1.3.3.** $d_H(0, i) \leq 0$ and $d_H(0, i) < 0$ if there is a path of negative weight from $j$ to $i$ for some $j > 0$ since we can go from 0 to $j$ first, then go from $j$ to $i$.

**Theorem 1.3.1.** $\hat{w}(i, j) \geq 0$ for all $i, j \in [n]$.

**Proof.** Since

$$\hat{w}(i, j) = w(i, j) + h(i) - h(j) = w(i, j) + d_H(0, i) - d_H(0, j),$$

and note that $d_H(0, i) + w(i, j)$ is the shortest distance of a path from 0 and go through $i$ to $j$, which is $\geq$ the distance from 0 to $j$, which is $d_H(0, j)$. Hence, we have

$$d_H(0, i) + w(i, j) - d_H(0, j) \geq 0.$$

■

Now we analyze the time complexity of Johnson's algorithm for general edge weights: We first obtain $h$ by doing one time Bellman-Ford's algorithm, which takes $O(mn)$ time. Then, we run Dijkstra's algorithm for all vertex $i$ of $G$, which takes totally $O\left(mn + n^2 \log n\right)$ time. Note that to here we just store $n$ shortest path tree, and we have to obtain the real distance by running through all $n$ tree, which takes $O(n) \cdot n = O\left(n^2\right)$ time since a tree has $n - 1$ edges. Hence, it totally take $O\left(mn + n^2 \log n\right)$ time for Johnson's technique.

# Chapter 2

# Maximum flow

**Problem 2.0.1** (The maximum flow problem)**.**

- Input: A directed graph $G$ with edge capacity $c : E(G) \to \mathbb{R}^+$ and two vertices, the source $s$ and the sink $t$.
- Output: An $(s,t)$-flow with maximum (flow) value.

**Remark 2.0.1.** For convenience, we allow $G$ has multiple/parallel edges, though merging multiple edges and increase the capacity to get an equivalent graph is not forbbiden.

**Remark 2.0.2.** An $(s,t)$-flow is a function $f : E(G) \to \mathbb{R}^+ \cup \{0\}$ satisfying

- capacity constraint: $f(uv) \leq c(uv)$ for each edge $uv$ of $G$.
- conservation law:
$$\sum_{uv \in E(G)} f(uv) = \sum_{vw \in E(G)} f(vw)$$
for each vertex $v$ of $G$ other than $s$ and $t$.

The value of an $(s,t)$-flow $f$ is
$$\sum_{sv \in E(G)} f(sv) - \sum_{us \in E(G)} f(us).$$

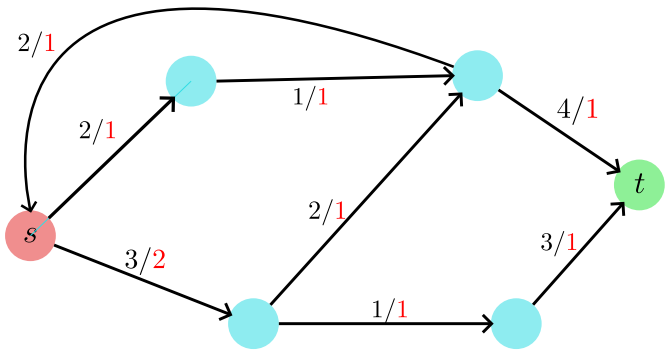



Figure 2.1: The maximum flow problem

## 2.1 Ford-Fulkerson's algorithm

**Intuition.** Reduce the maximum $(s,t)$-flow problem to the reachability problems for a sequence of graph $R$.

**Definition 2.1.1** (Residual graph)**.** The residual graph $R(f)$ with respect to a flow $f$ of $G$ with $V(R(f)) = V(G)$ is defined as follows for each edge $uv$ of $G$:

- If $f(uv) < c_f(g)$, then let $R(f)$ have an edge $uv$ with capacity $c(uv) - f(uv)$.
- If $f(uv) > 0$, then let $R(f)$ have an edge $vu$ with capacity $f(uv)$.

**Corollary 2.1.1.** $R(f)$ is a graph with capacity of every edge positive, just like $G$.

**Remark 2.1.1.** Even if $G$ has only one $uv$ edge, $R(f)$ may have 2 $uv$ edges if $f(uv) < c(uv)$ and $f(vu) > 0$.

**Theorem 2.1.1.** For any $(s,t)$-flow $f$ of $G$, we have the following statements:

(1) If $d_{R(f)}(s,t) = \infty$, then $f$ is a maximum $(s,t)$-flow of $G$.

(2) If $d_{R(f)}(s,t) < \infty$ and $g$ is an $(s,t)$-flow of the residual graph $R(f)$, then $f+g$ remains an $(s,t)$-flow of $G$, where
$$(f+g)(uv) = f(uv) + g(uv) - g(vu)$$
for each edge $uv$ of $G$.

**Remark 2.1.2.** In (2), the $g(uv)$ and $g(vu)$ corresponds to the edges on $R(f)$ and formed by $uv$ in $G$ and we give the flow value to this $uv, vu$ pairs. For $vu$ in $G$, it may also form a $uv, vu$ pairs in $R(f)$, and these two cases should be handled seperately.

**proof of (1).** If $d_{R(f)}(s,t) = \infty$, and there exists $f'$ s.t. $|f'| > |f|$, i.e. $f$ is not a maximum $(s,t)$-flow. Then, we define

$$h(uv) \coloneqq f'(uv) - f(uv) \text{ for all } uv \in E(G).$$

Note that for all $v \neq s,t$ we have

$$\begin{aligned}\sum_x h(xv) &= \sum_x f'(xv) - f(xv) = \sum_x f'(xv) - \sum_x f(xv) = \sum_x f'(vx) - \sum_x f(vx) \\ &= \sum_x f'(vx) - f(vx) = \sum_x h(vx).\end{aligned}$$

Also, we have

$$\begin{aligned}|h| &= \sum_x h(sx) - h(xs) = \sum_x f'(sx) - f(sx) - (f'(xs) - f(xs)) \\ &= \sum_x f'(sx) - f'(xs) - \sum_x f(sx) - f(xs) = |f'| - |f| > 0.\end{aligned}$$

Now we convert $h$ to a $(s,t)$-flow on $R(f)$. Note that

- If $h(uv) \geq 0$, then
$$0 \leq h(uv) = f'(uv) - f(uv) \leq c_f(uv) - f(uv),$$
so $h(uv)$ fits within the foward residual capacity of $uv$ if $h(uv) \geq 0$.

- If $h(uv) < 0$, then
$$0 < -h(uv) = f(uv) - f'(uv) \leq f(uv),$$
so $-h(uv)$ fits within the reverse residual capacity on arc $vu$ if $h(uv) < 0$.

Now we construct a new graph $R(f)'$ by

(1) If $f(uv) < c_f(g)$, then let $R'(f)$ have an edge $uv$ with capacity $c_f(uv) - f(uv)$. If $f(uv) = c_f(g)$, then have an edge $uv$ with capacity 0.

(2) If $f(uv) > 0$, then let $R'(f)$ have an edge $vu$ with capacity $f(uv)$. If $f(uv) = 0$, then have an edge $vu$ with capacity 0.

Hence, for every vertices $u, v \in V(R'(f))$, there is a forward residual arc $uv$, which is formed by $uv$ in (1), and a reverse residual arc $uv$ formed by $vu$ in (2). We can define $g$ on $R'(f)$ by

$$\begin{cases} g(uv) = g_f(uv) = \max\{h(uv), 0\} & \text{on the forward residual arcs } uv. \\ g(vu) = g_r(vu) = \max\{-h(uv), 0\} & \text{on the reverse residual arcs } vu. \end{cases}$$

Then, we claim that $g$ is a flow of $R'(f)$ with flow value $> 0$. Note that $R'(f)$ is in fact an expansion of $R(f)$, but add some edges with capacity 0, and anything else is not changed.

- Capacity constraint: If the forward residual arc $uv$ exists in $R(f)$, then we know
$$c_{R'(f)}(uv) = c_f(uv) - f(uv) > 0.$$
If $g_f(uv) = 0$, then $g_f(uv) < c_{R'(f)}(uv)$, and if $g_f(uv) = h(uv)$, then $h(uv) > 0$ and thus
$$g_f(uv) = h(uv) \leq c_f(uv) - f(uv) = c_{R'(f)}(uv)$$
by the arguments of $h$ above. Hence, if the forward residual arc $uv$ exists in $R(f)$, then $g_f(uv) \leq c_{R'(f)}(uv)$. Now if the forward residual arc $uv$ does not exist in $R(f)$, then $c_{R'(f)}(uv) = 0$ and $c_f(uv) = f(uv)$, so we have
$$h(uv) = f'(uv) - f(uv) \leq c(uv) - f(uv) = f(uv) - f(uv) = 0,$$
so we must have
$$g_f(uv) = \max\{h(uv), 0\} = 0,$$
and thus in this case $g_f(uv) \leq c_{R'(f)}(uv)$.
Now we discuss the reverse residual arc. If the reverse residual arc $vu$ exists in $R(f)$, then
$$c_{R'(f)}(vu) = f(uv) > 0.$$
Now if $g_r(vu) = 0$, then $g_r(vu) \leq c_{R'(f)}(vu)$. If $g_r(vu) = -h(uv)$, then $h(uv) < 0$, and thus
$$g_r(vu) = -h(uv) \leq f(uv) = c_{R'(f)}(vu)$$
by the above arguments about $h$. Now if the reverse residual arc $vu$ does not exist in $R(f)$, then $f(uv) = 0$ and $c_{R'(f)}(vu) = 0$. Hence,
$$h(uv) = f'(uv) - f(uv) = f'(uv) \geq 0,$$
so
$$g_r(vu) = \max\{-h(uv), 0\} = 0,$$
and thus $g_r(vu) \leq c_{R'(f)}(vu)$. Hence, we have shown that the capacity constraint is always correct.

- Conservation law: For $u \neq s, t$, we know

$$\begin{aligned}&\sum_x g_f(ux) + g_r(ux) - \sum_x g_f(xu) + g_r(xu)\\ &= \sum_x \max\{h(ux), 0\} + \max\{-h(xu), 0\} - \max\{h(xu), 0\} - \max\{-h(ux), 0\}\end{aligned}$$

and we can observe that

$$\max\{h(ux), 0\} + \max\{-h(xu), 0\} - \max\{h(xu), 0\} - \max\{-h(ux), 0\} = h(ux) - h(xu)$$

no matter what the sign of $h(ux)$ and $h(xu)$ is. Hence,

$$\sum_x g_f(ux) + g_r(ux) - \sum_x g_f(xu) + g_r(xu) = \sum_x h(ux) - h(xu) = 0.$$

Also,

$$\begin{aligned}|g| &= \sum_x g_f(sx) + g_r(sx) - g_f(xs) - g_r(xs)\\ &= \sum_x \max\{h(sx), 0\} + \max\{-h(xs), 0\} - \max\{h(xs), 0\} - \max\{-h(sx), 0\}\\ &= \sum_x h(sx) - h(xs) = |h| > 0\end{aligned}$$

by similar arguments.

Hence, we know $g$ is a flow with flow value $> 0$ on $R'(f)$. Also, notice that if the forward residual edge $uv$ does not exist in $R(f)$, then $g_r(uv) = 0$, while the reverse residual edge $vu$ does not exist in $R(f)$ implies $g_r(vu) = 0$. Hence, if we define

$$g'(uv) \coloneqq \begin{cases} g_f(uv), & \text{if } uv \text{ exists in } R(f) \text{ as a forward residual arc;}\\ g_r(uv), & \text{if } uv \text{ exists in } R(f) \text{ as a reverse residual arc.}\end{cases}$$

Then, $g'$ is a flow in $R(f)$ and $|g'| = |g| > 0$.

Now we claim that if a positive flow exists in $R(f)$, then $s$ and $t$ are reachable. Actually, for any multi-directed graph $G'$, if $p$ is a $(s,t)$-flow of $G'$ and $|p| > 0$, then $s$ and $t$ are reachable. We prove the general case. For all $e_1, e_2, \ldots, e_k$ are parallel edges between $u$ and $v$, then we define

$$p'(uv) = p(e_1) + p(e_2) + \cdots + p(e_k).$$

Hence, we have

$$\sum_x p'(ux) = \sum_x p'(xu) \text{ for all } u \neq s, t \text{ and } |p| = \sum_x p'(sx) - \sum_x p'(xs).$$

Now if $s$ and $t$ are not reachable, then consider

$$S = \{v \in V(G') \mid \text{there is a directed } sv \text{ path made only of arcs } e \text{ with } p(e) > 0\}.$$

Hence, $t \neq S$ and $s \in S$. First, note that no positive edge leave $S$, otherwise the endpoint of this point should be in $S$, which is a contradiction. Now since

$$\sum_{x \in S} \left( \sum_y p'(xy) - \sum_y p'(yx) \right) = |p| + \sum_{x \in S \setminus \{s\}} 0 = |p| > 0$$

and we know

$$\sum_{x \in S} \left( \sum_y p'(xy) - \sum_y p'(yx) \right) = \sum_{\substack{u \in S \\ v \notin S}} p'(uv) - \sum_{\substack{u \notin S \\ v \in S}} p'(uv)$$

since

$$\sum_{\substack{x\in S\\ y\in S}} p'(xy) - \sum_{\substack{x\in S\\ y\in S}} p'(yx) = 0 \text{ (these are two same things)}$$

However, we know there is no positive edge leaves $S$, so

$$\sum_{u\in S\mathbb{V}\notin S} p'(uv) = 0,$$

and thus

$$0 < |p| = \sum_{x\in S}\left(\sum_{y} p'(xy) - \sum_{y} p'(yx)\right) = \sum_{\substack{u\in S\\ v\notin S}} p'(uv) - \sum_{\substack{u\notin S\\ v\in S}} p'(uv) = -\sum_{\substack{u\notin S\\ v\in S}} p'(uv) \le 0,$$

which is a contradiction. Hence, $s$ and $t$ are reachable.

Now from this claim, we know $d_{R(f)}(s,t) < \infty$, which is a contradiction, so $f$ is a maximum flow. ■

**proof of (2).**

- Capacity constraint: We want to show $(f+g)(uv) \le c_f(uv)$. Note that

$$\begin{aligned}(f+g)(uv) &= f(uv) + g(uv) - g(vu) \le f(uv) + (c_f(uv) - f(uv)) - g(vu)\\ &= c_f(uv) - g(vu) \le c_f(uv).\end{aligned}$$

  Hence, this is true.

- Conservation law: For $v \neq s,t$, we have

$$\begin{aligned}\sum_x (f+g)(vx) - \sum_x (f+g)(xv) &= \sum_x f(vx) + g_f(vx) - g_r(xv) - (f(xv) + g_f(xv) - g_r(vx))\\ &= \sum_x g_f(vx) + g_r(vx) - g_r(xv) - g_f(xv) = 0\end{aligned}$$

  since

$$\sum_x f(vx) - f(xv) = 0$$

  and $g$ is a flow on $R(f)$.

■

**Algorithm 2.1:** Ford-Fulkerson's algorithm

**Setup:** Let $f(uv) = 0$ for each edge $uv$ of $G$. Repeat the following steps until $R(f)$ does not contain an $st$-path.

1 Obtain an $st$-path $P$ of $R(f)$. Let $q$ be the minimum capacity of the edges of $P$ in $R(f)$.
2 Obtain an $st$-flow $g$ of $R(f)$ by letting $g(uv) = q$ for each edge $uv$ of $P$ and $g(uv) = 0$ for all other edges $uv$ of $G$.
3 Let $f = f + g$.

Initially, $f$ is a legal $st$-flow. In each round, $g$ is a legal $st$-flow of $R(f)$ (Easy to check). Then, by previous lemma, we know $f+g$ is also a legal flow of $G$, so the algorithm seems correct. However, will the algorithm terminate?

We claim that when all edge capacities are integers, then Ford-Fulkerson's algorithm terminates.

Note that $q \geq 1$ since $R(f)$ has all edges' capacities $> 0$ and all edges' capacities are integers. Also,

$$\begin{aligned}
|f+g| &= \sum_x (f+g)(sx) - (f+g)(xs) \\
&= \sum_x f(sx) + g_f(sx) - g_r(xs) - (f(xs) + g_f(xs) - g_r(sx)) \\
&= \left(\sum_x f(sx) - f(xs)\right) + \sum_x g_f(sx) + g_r(sx) - g_f(xs) - g_r(xs) \\
&= |f| + |g| = |f| + q \geq |f| + 1,
\end{aligned}$$

so $|f+g|$ is strictly increasing in each round. Hence, if $G$ is a finite graph, then Ford-Fulkerson's algorithm must terminate.

Now we analyze the time complexity. In each round, searching an $st$-path takes $O(|E|)$ times, where $E$ is the set of edges of $G$. Then, suppose the sum of capacity of all edges are $C$, then we need $O(C)$ round since after each round $|f|$ increases at least 1 and $|f| \leq C$ for all flow $f$ of $G$. Hence, it takes $O(|E|C)$ times.

**Question.** Is this algorithm polynomial time or exponential time?

**Definition 2.1.2.** An algorithm is polynomial time only if its running time is bounded by a polynomial in the size of the input encoding, i.e.

$$T(\text{algorithm}) = (\text{size of the input encoding})^{O(1)}.$$

Also, we can similarly define exponential time, and linear time, e.t.c.

If $C = O(1)$, then the size of input encoding is $O(|E| \cdot 1) = O(1)$, and thus the time complexity $O(|E|C) = O(|E|)$ is linear time.

If $C$ has no restriction, then the space complexity for storing all edges' capacities is $O(|E| \log C)$ (we need $O(\log C)$ bits to store the capacity of an edge). Hence,

$$O(|E|C) \neq O(|E| \log C)^{O(1)},$$

so Ford-Fulkerson's algorithm is not polynomial time.

**Question.** What about when the capacity is not integers?

**Answer.** Then Ford-Fulkerson's algorithm may not stop, and although the flow value converges, but the limit value is not the maximum flow value. ⊛

## 2.2 Edmonds and Karp's algorithm

Now we introduce the first polynomial time algorithm for maximal flow problem.

**Theorem 2.2.1** (Edmonds and Karp, JACM 1972)**.** If one makes sure that the augmenting $st$-path $P$ in $R(f)$ is an $st$-path in $R(f)$ having a minimum number of edges, then the time complexity of Ford-Fulkerson's algorithm is $O\left(m^2 n\right)$.

**Remark 2.2.1.** If we ensure we use the $st$-path of least number of edges in each round, then we can make sure the algorithm terminates within $mn$ rounds.

**Remark 2.2.2.** We do not assume $G$ has integer capacities under this circumstance.

From now on, we assume we pick the shortest $st$-path in each round. We need two observations to prove the theorem: 現邊 and 遞增觀察.

**Lemma 2.2.1 (現邊觀察).** If in some round the residual graph $R(f+g)$ has some edge $uv$ where $uv$ does not exist in $R(f)$, then

$$d^\star_{R(f)}(s,u) = d^\star_{R(f)}(s,v) + 1,$$

where $d^\star_{R(f)}(s,w)$ for a vertex $w$ is the distance of $s$ to $w$ in the unweighted version of $R(f)$.

**Proof.** If $R(f)$ has no $uv$ this edge, but $R(f+g)$ has $uv$ this edge, then the only possibility is the unweighted shortest $st$-path $P$ of $R(f)$ go through $vu$ this edge. The reason is as follows:

Since $uv$ is not in $R(f)$, so $P$ does not go through $uv$. If $P$ does not go through $vu$, then $g(uv) = g(vu) = 0$ no matter it is an forward residual arc or an reverse residual arc, and thus

$$\begin{aligned}(f+g)(uv) &= f(uv) + g_f(uv) - g_r(vu) = f(uv)\\ (f+g)(vu) &= f(vu) + g_f(vu) - g_r(uv) = f(vu).\end{aligned}$$

Since $R(f)$ does not have $uv$, and $f+g$ and $f$ share same flow value between $u$ and $v$, so it is impossible that $R(f+g)$ contains $uv$, which is a contradiction. Now that $vu$ is in $P$, which is an unweighted shortest $st$-path of $R(f)$, so the path is like

$$s \to \cdots \to v \to u \to \cdots \to t,$$

and thus

$$d^\star_{R(f)}(s,u) = d^\star_{R(f)}(s,v) + 1.$$

■

**Lemma 2.2.2 (遞增觀察).** Let the augmenting path $P$ be an $st$-path whose number of edges is minimized in the residual graph $R(f)$. Let $g$ be the saturating flow for $R(f)$ corresponding to $P$. For each vertex $v$ of $G$, we have

$$d^\star_{R(f)}(s,v) \le d^\star_{R(f+g)}(s,v).$$

**Proof.** Assume for contradiction that there is a vertex $v$ of $G$ with

$$d^\star_{R(f)}(s,v) > d^\star_{R(f+g)}(s,v). \tag{2.1}$$

Thus, $d^\star_{R(f+g)}(s,v) \neq \infty$. Let $v$ be such a vertex closest to $s$ in the unweighted version of $R(f+g)$. We know $v \neq s$ since

$$d^\star_{R(f)}(s,s) = 0 = d^\star_{R(f,g)}(s,s).$$

Let $Q$ be an unweighted shortest $sv$-path of $R(f+g)$. Let $uv$ be the last edge of $Q$. (Note that $u$ could be $s$.) We have

$$d^\star_{R(f)}(s,u) \le d^\star_{R(f+g)}(s,u) \tag{2.2}$$

since we suppose $v$ is the vertex closest to $s$ in $R(f+g)$ which violates the assumption in the lemma and $d^\star_{R(f+g)}(u) + 1 = d^\star_{R(f+g)}(v)$.

- Case 1: $uv \subseteq R(f)$, then we know

  $$d^\star_{R(f)}(s,v) \le d^\star_{R(f)}(s,u) + 1 \le d^\star_{R(f+g)}(s,u) + 1 = d^\star_{R(f+g)}(s,v),$$

  which contradicts to Equation 2.1.

- Case 2: $uv \not\subseteq R(f)$, then since $uv \subseteq R(f+g)$, so by **現邊觀察** we have

  $$d^\star_{R(f)}(s,v) = d^\star_{R(f)}(s,u) - 1 \le d^\star_{R(f+g)}(s,u) - 1 = d^\star_{R(f+g)}(s,v) - 2,$$

  which contradicts to Equation 2.2.

Hence, it is impossible that such $v$ exists. ■

Now we prove Theorem 2.2.1.

**proof of Theorem 2.2.1.** Since each round takes $O(m)$ time (BFS), we prove the theorem by showing that Edmond-Karp's algorithm halts in $O(mn)$ rounds.

**Claim 2.2.1.** Each round saturates at least one edge of the $O(m)$ edges of $G \cup G^r$, causing them to disappear in the residual graph of the next round.

**Proof.** Suppose in $R(f)$, the saturating flow is $g$ and the corresponding path of minimal number of edges is $P$, then if $uv \in P$ and $c_{R(f)}(uv) = \min_{e \in P} c_{R(f)}(e)$, we claim that $uv \notin R(f+g)$. Suppose $c_{R(f)}(uv) = q$, then we have two cases:

- Case 1: $uv$ is a forward residual arc. Then $q = c_G(uv) - f(uv)$, and thus

  $$(f+g)(uv) = f(uv) + g_f(uv) - g_r(vu) = f(uv) + q - 0 = f(uv) + c_G(uv) - f(uv) = c_G(uv).$$

  Hence, the forward residual arc $uv$ will not appear in $R(f+g)$.

- Case 2: $uv$ is a reverse residual arc. Then, $q = f(vu)$. Hence,

  $$(f+g)(vu) = f(vu) + g_f(vu) - g_r(uv) = f(vu) + 0 - q = f(vu) - f(vu) = 0,$$

  so the reverse residual arc $uv$ will not appear in $R(f+g)$.

Thus, in each round at least one edge of $P$ will disappear in next round. ⊛

Hence, it sufficies to show that each edge $uv$ of $G \cup G^r$ disappears $O(n)$ times in residual graphs through out the algorithm. Suppose that an edge $uv$ of $G \cup G^r$ is not in $R(f)$ and

- appears in $R(f+g)$ then
- disappears in $R(f+g+\cdots+g'+h)$ for the first time after $R(f+g)$,

where $g'$ could be $g$. The saturating flow $h$ of $R(f+g+\cdots+g')$ corresponding to the augmenting unweighted shortest $st$-path $P$ saturates $uv$. Thus, $uv \in E(P)$. We have

$$\begin{aligned} d^\star_{R(f)}(s,u) &= d^\star_{R(f)}(s,v) + 1 && \text{by 現邊觀察} \\ &\le d^\star_{R(f+g+\cdots+g')}(s,v) + 1 && \text{by 遞增觀察} \\ &= d_{R(f+g+\cdots+g')}(s,u) + 2 && \text{by } uv \in E(P). \end{aligned}$$

By $d^\star_H(s,u) \in \{0,1,\ldots,n-1,\infty\}$ for any residual graph $H$, $uv$ can appear and disappear $O(n)$ times in residual graphs throughout the algorithm. Thus, the algorithm halts in $O(mn)$ rounds, and thus runs in $O\left(m^2 n\right)$ time. ■

## 2.3 An Application: Bipartite Matching

**Definition 2.3.1.** A graph $G$ is bipartite if there are disjoint vertex subsets $U$ and $V$ of $G$ with $U \cup V = V(G)$ s.t. each edge of $G$ is between a vertex in $U$ and a vertex in $V$.

**Definition 2.3.2.** An edge subset $M$ of $G$ is a matching of $G$ if $M = \varnothing$ or the minimal subgraph $H$ of $G$ with edge $E(H) = M$ has the maximum degree 1.

**Problem 2.3.1** (Maximum matching of bipartite graph).
Input: An undirected bipartite graph $G$.
Output: A matching $M \subseteq E(G)$ with maximum $|M|$.

Actually, we can convert this problem to the maximum flow problem. Let $G(s,t)$ be the unit-capacity graph obtained from $G$ by adding

- new vertices $s$ and $t$.

- new edges $su$ for all $u \in U$ and
- new edges $vt$ for all $v \in V$.

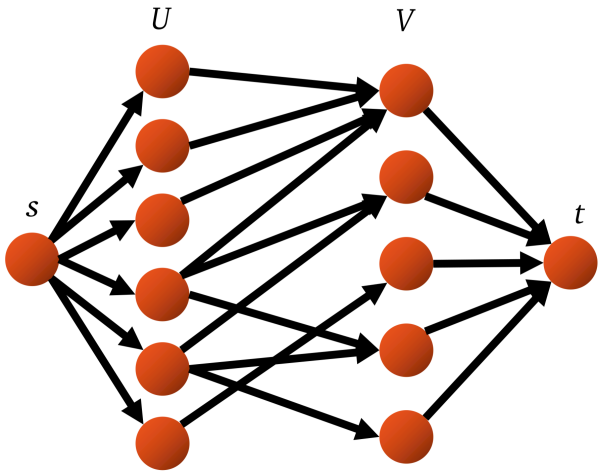



Figure 2.2: Converted graph

Then, we have:

**Theorem 2.3.1.** $G$ has a maximum matching with $k$ edges if and only if $G(s,t)$ has a maximum flow with value $k$.

**Proof.** We first prove a lemma:

> **Lemma 2.3.1.** If $G$ is a graph with capacity all integer, then $G$ has an integral maximum flow.
>
> **Proof.** Note that in this case the maixmum flow made by Ford-Fulkerson's (or Edmonds-Karp's) algorithm is an integral flow. ■

If $G$ has a maximum matching with $k$ edges, then we can gives these $k$ edges flow value 1 and other edges from $U$ to $V$ 0, then in this case the flow value is $k$. Now if this flow is not a maximal flow, then there exists a flow of flow value at least $k+1$, and by lemma, there exists an integral flow with flow value at least $k+1$, but this means there exists at least $k+1$ edges from $U$ to $V$ is of flow value 1, which means there exists $u \in U$ or $v \in V$ with 2 edges incident to them of flow value 1, but this is impossible since each vertex in $U$ has 1 in-degree and each vertex in $V$ has 1 out-degree. Hence, maximum flow value is $k$.

Now if $G(s,t)$ has a maximum flow with flow value $k$, then by lemma $G(s,t)$ exists an integral flow with value $k$. Hence, in this integral flow, $s$ has $k$ out-degree with flow value 1, i.e. there exists $k$ vertices of $U$ has an out-degree of value 1, and note that any vertex of $V$ won't have 2 in-degree edges with flow value 1, so by collecting the edges from $U$ to $V$ whose flow value is 1, this collection is a matching, and if this matching is not maximum, then there exists a maximum mathcing with $k$ edges, and thus $G(s,t)$ has a maximum flow with value $k+1$, but we know the maximum flow value is $k$, so it is maximum matching. ■

# Chapter 3

# Nearest Points

**Problem 3.0.1.**
Input: $n$ points in the plane.
Output: Two points whose distance is minimized.

**Remark 3.0.1.** Without lose of generality, we suppose $n = 2^k$.

The Naive algorithm takes $O\left(n^2\right)$ times, which solve 老大問題 on the $O\left(n^2\right)$ point pairs' distances.

**Algorithm 3.1:** A smart $O(n \log n)$ algorithm

**Data:** $n$ points $p[1], p[2], \ldots, p[n]$

1 Sort $\{p[i]\}_{i=1}^{n}$ by their $y$-coordinates in descending order.

2 Use minimum selection to find the middle $x$-coordinate of $\{p[i]\}_{i=1}^{n}$, say $m$.

3 Collect the $p[i]$'s with $p[i].x \leq m$ to be $L$ and the part $p[i].x > m$ to be $R$.

4 Divide and conquer on $L, R$ to find the minimum distance among $L$ and $R$, say they are $d_l$ and $d_r$, respectively.

5 Suppose $d = \min\{d_l, d_r\}$, then collect $\{m[i]\}_{i=1}^{n_m}$ where $m - d \leq p[m[i]].x \leq m + d$ by searching through $\{p[i]\}_{i=1}^{n}$, so we have $p[m[i]].y \geq p[m[i+1]].y$ for all $i$.

6 For each $m[i]$, find

$$d[i] = \min_{\substack{\max\{i-8,1\} \leq j \leq \min\{i+8, n_m\} \\ j \neq i}} d(p[m[j]], p[m[i]]).$$

7 Return

$$\min\left\{\min_{1 \leq i \leq n_m} d[i], d_l, d_r\right\}.$$

We first explain the correctness. In step 4, if we find out $d_l$ and $d_r$, then the minimum distance may be $d_l, d_r$, or $\min_{l \in L, r \in R} d(l, r)$. Hence, we need to compute $\min_{l \in L, r \in R} d(l, r)$. Note that for each $p[i] \in L$, suppose $p[i] = (x[i], y[i])$, then we just need to check the distance between $p[i]$ and all points in $[m, m + d] \times [y[i] - d, y[i] + d]$. This area is a $d \times 2d$ box in $R$. Note that if we partition this box into 2 $d \times d$ subboxes, then in each subbox, there must be at most 4 points, otherwise if there are 5 points in a subbox, then by pigeonhole principle there must be two points have distance less than $d$, which is impossible. In the implementation, we just need to find the 8 points above and below $p[m[i]]$ for each $i$, then we can make sure we have checked all the candidates for being the minimum distance pairs, where this pair is crossing between $L$ and $R$. (See Figure 3.1)

Now since minimum selction takes $O(n)$ time, and in each round the worst case is that all points are between $x = m - d$ and $x = m + d$, in this case, since each point takes $O(1)$ time to check, so we need $O(n)$ time in each round, and since the divide and conquer's time is

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n),$$

so $T(n) = O(n \log n)$ by master's theorem. Also, the sorting in the beginning takes $O(n \log n)$ time, so the final time complexity is $O(n \log n)$.

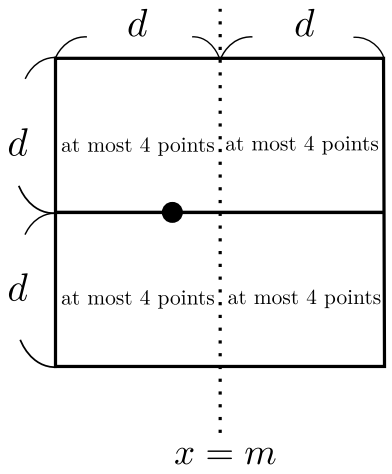



Figure 3.1: The nearest point box

# Chapter 4

# Convex Hull

**Problem 4.0.1.**
Input: $n$ points in the plane.
Output: A minimum convex polygon that contains all the points.

**Algorithm 4.1:** Naive Algorithm

1 Let $l$ be the point with minimal $x$-coordinate, which must be on the convex hull.
2 Link every points with $l$, then the point $x$ with the maximal slope $m_{xl}$ must be on the convex hull.
3 Rotate the plane so that $x$ becomes new $l$, and repeat this step until we rotate back to the first $l$.

Note that this algorithm takes $O\left(n^2\right)$ time since each round takes $O(n)$ time, and we need $O(n)$ rounds.

## 4.1 An $O(n \log n)$ algorithm for convex hull

**Algorithm 4.2:** The $O(n \log n)$ algorithm for convex hull

1 First find $l$, which is the point with minimal $x$-coordinate.
2 Sort other $n-1$ points by the slope of the line incident by that point and $l$ in ascending order. We call the sorted points $p_1, p_2, \ldots, p_n$, where $p_1 = l$.
3 Let $C_3 = \{p_1, p_2, p_3\}$.
4 **for** $i = 4$ *to* $n$ **do**
5 Use $C_{i-1}$ to calculate $C_i$.
6 The answer is $C_n$.

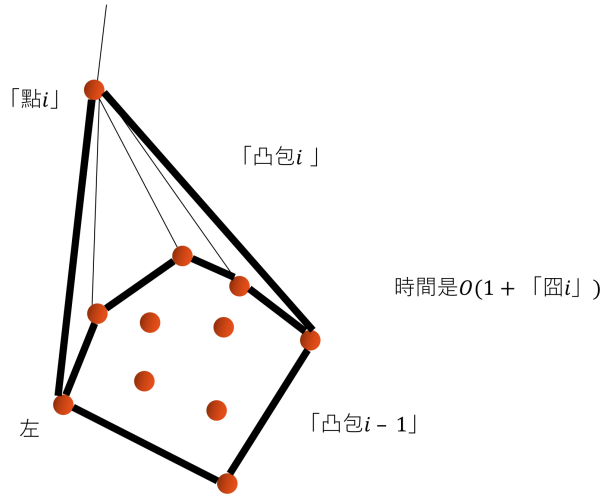



Figure 4.1: From $C_{i-1}$ to $C_i$

Here 囧 $i$ is the set of points which is on $C_{i-1}$ but not on $C_i$, and note that summing up all 囧 $i$ will not exceed $n$ since each point will be "wrapped" at most once.

**Remark 4.1.1.** Note that we record the points on the convex hull starting from $i$ and clockwise in each round, so that we in each round, if we have $C_{i-1}$, we follow this order to calculate the slope of the line incident to $p_i$ and $x$ where $x$ is the point on $C_{i-1}$, and note that this slope initially is positive and will increase first, and then at some point become negative and keep increasing (but remain negative), but at some point if the slope decrease or become positive, then we can end calculating. Note that we only calculate $O(1 + 囧 i)$ times.

## 4.2 An application: Farthest pair of points in $O(n \log n)$ time

**Problem 4.2.1.**
Input: $n$ points in the plane.
Output: Two points whose distance is maximized.

The naive algorithm takes $O\left(n^2\right)$ time, but in fact we can use convext hull to design an $O(n \log n)$ time algorithm.

**Theorem 4.2.1.** The farthest two points must be both on convex hull.

Now if for any point $p$ on the convex hull $C$ , we can find the point $q$ on $C$ s.t.

$$d(p, q) = \max_{\substack{s \in C \\ s \neq p}} d(p, s)$$

in $O(\log n)$ time, then we can find the point pair with maximum distance in $O(n \log n)$ time (find convex hull also takes $O(n \log n)$ time so the total time complexity is $O(n \log n)$).

Note that fix $p \in C$, then if $C$ is

$$p = m_1 \to m_2 \to \cdots \to m_k \to m_1 = p,$$

then the sequence $\{d(p, m_i)\}_{i=2}^{k}$ is bitonic.

**Definition 4.2.1.** A bitonic sequence $A[1], A[2], \ldots, A[n]$ is an integer sequence s.t.

$$A[1] \leq A[2] \leq \cdots \leq A[k] \geq A[k+1] \geq \cdots \geq A[n]$$

for some $k$.

**Theorem 4.2.2.** If $\{A[i]\}_{i=1}^{n}$ is a bitonic sequence, then we can find

$$\arg\max_{1 \leq j \leq n} A[j]$$

in $O(\log n)$ time.

**Proof.** We can use binary search, and observe if at some term it is increasing or decreasing to decide we should go left or right in the next step. ■

Hence, by this theorem, we can find

$$\arg\max_{2 \leq i \leq k} d(p, m_i)$$

in $O(\log k)$ time for every $p \in C$, and note that $k \leq n$ since $k$ is the size of $C$, and hence it takes $O(\log n)$ time to find the maximum distance between a fixed point and any other point on $C$, and we're done.

# Chapter 5

# Some Types of Balanced Search Trees

## Lecture 3

27 Nov. 14:20

## 5.1 Red-Black Tree

**Definition 5.1.1** (Binary search tree)**.** A binarty tree is a binary search tree iff its in-order traversal is an increasing sequence.

**Definition 5.1.2** (Red-Black Tree)**.** A binary search tree is a red-black tree if its root is black, and any path from leaves to the root has same number of black nodes, and any two red nodes are not adjacent on the tree.

## 5.2 B-Tree

**Definition 5.2.1** (B-tree of order $t \geq 2$)**.**

- Each non-root internal node has at least $t$ children.
- Each internal children has at most $2t$ children.
- Each internal node has one more children than key.
- All leaf nodes are at the same level.
- The in-order traversal yields an increasing sequence of the keys.

**Definition 5.2.2** (2-3-4 tree)**.** A 2-3-4 tree is exactly a B-tree of order 2: The first condition can be simplified as Each ~~non-root~~ internal node has at least 2 children.

We mainly focus on 2-3 tree:

**Definition 5.2.3** (2-3 tree)**.** A 2-3 tree is a 2-3-4 tree such that

- Each internal node has at most 3-children. (An internal node having 1 or 4 children may appear temporarily during the insertion precess.)

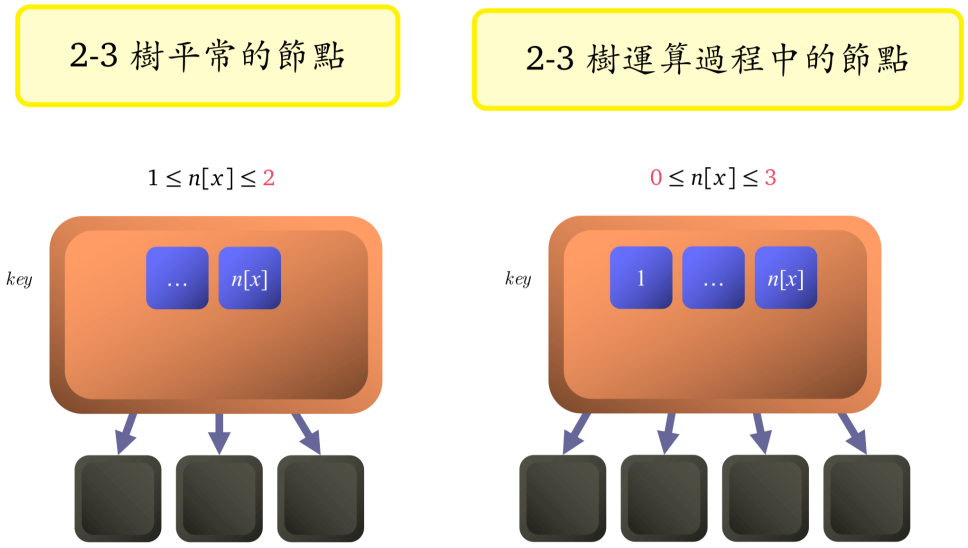



Figure 5.1: The number of key of node $x$ in a 2-3 tree

**Corollary 5.2.1.** If the tree has $n$ nodes, then the membership query time is $O(\log n)$.

**Theorem 5.2.1.** We can insert a key into a 2-3 tree in $O(\log n)$ time.

**Proof.** We can first insert the key into a leaf, then whenever a node is full, i.e. has 3 keys or equivalently 4 children (if it is an internal node), then we split this node into 2 part, and raise the key in the middle to one layer up, and keep doing this until some ancestor node has only two keys or the height of the tree increases by 1.

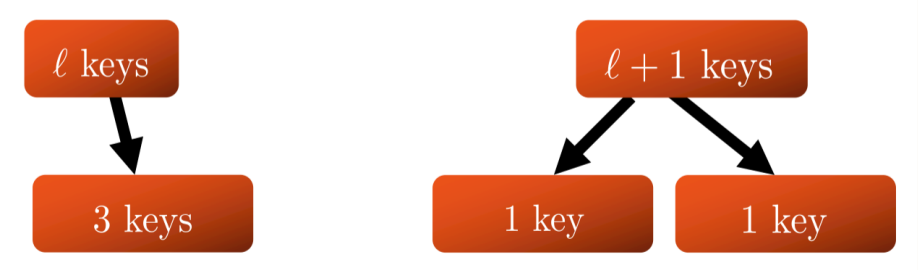



Figure 5.2: Insertion of 2-3 tree

**Remark 5.2.1.** Thus in the process of insertion there may appear some trees with nodes of 3 keys, but after the insertion finishing, such nodes do not exist.

■

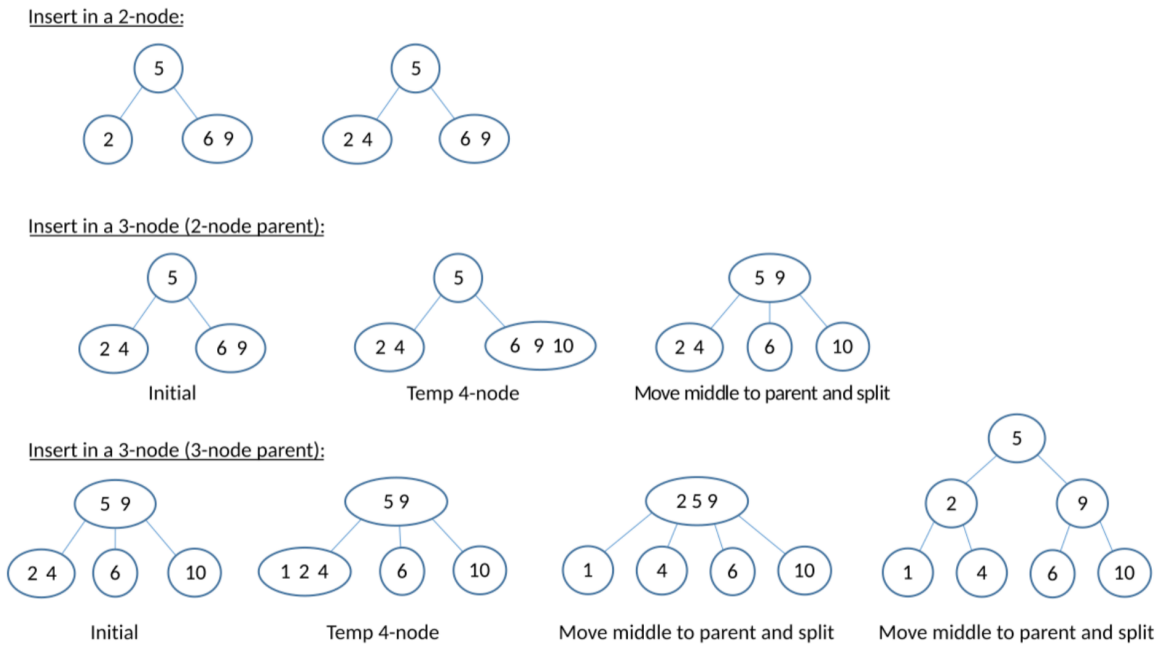



Figure 5.3: Insetion in different cases

**Theorem 5.2.2.** We can delete a node in $O(\log n)$ time.

**Proof.** We first find the key $k$, which is the key we want to delete. If $k$ is on the leaf, then we finish the first step. If not, then we know $k$ is between two leaves $l$ and $r$, and thus we can spend $O(\log n)$ time to find the position of $k$, and we can use the rightmost key on $l$ or leftmost key on $r$ to replace $k$, and then we have to fix the tree if it does not satisfy the definition of 2-3 tree.

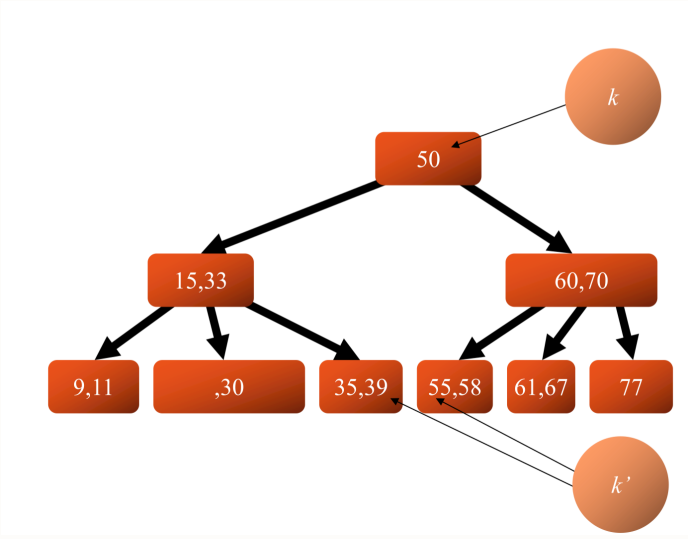
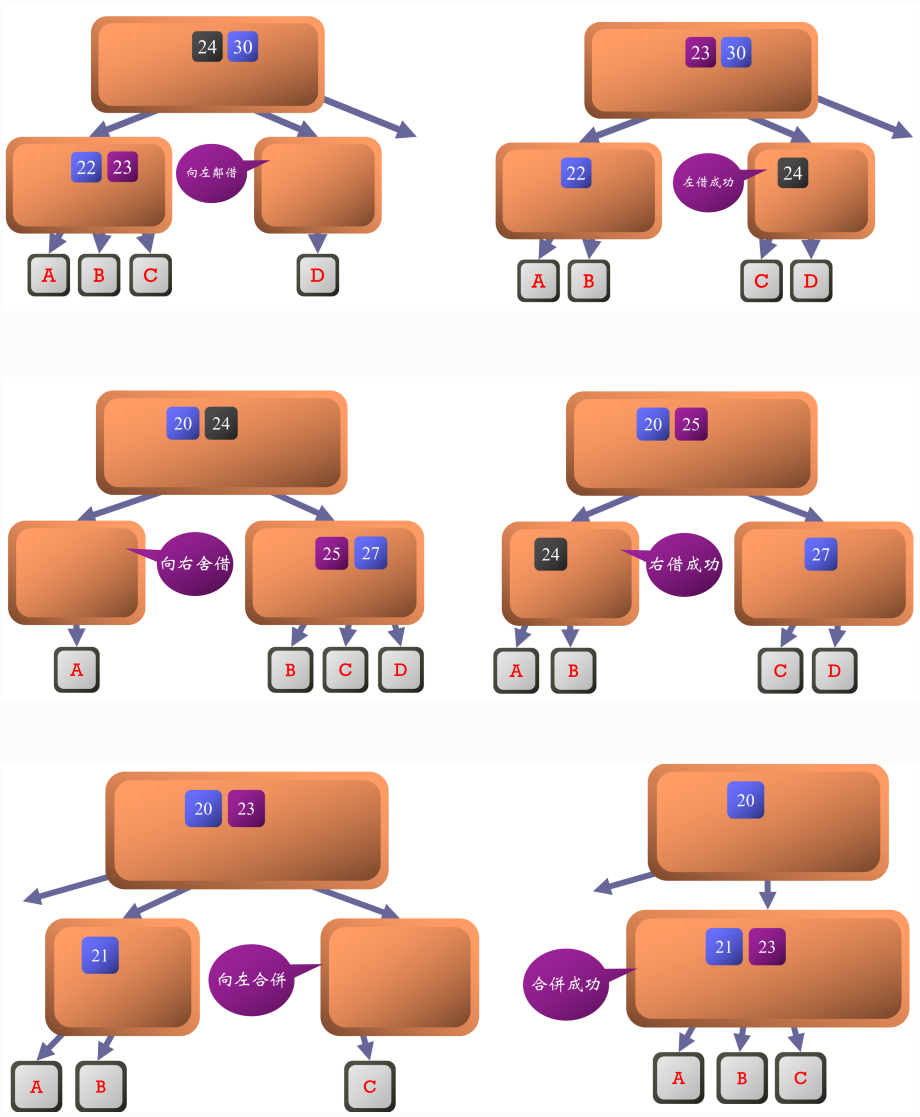



Figure 5.4: Find $k$ in the treee

Now suppose we replace $k$ by the key on the node $x$, i.e. $x$ is one of $l, r$, then if $x$ originally has two keys, then the deletion is finished (because one key is left and it is still a legal 2-3 tree). Now if $x$ has 0 key, then we spread the empty node above by

- borrow keys from neighbors first.
- if it does not work (neighbors not exist or they have only one key), then we look at its parent

and adjust some keys in itself, its neighbor, and its parent.




**Remark 5.2.2.** A case is ommited in the slides, which is that: Suppose $p$ is the parrent of $c_l$ and $c_r$, and $c_l$ and $p$ has one child, while $c_r$ has 0 keys now, then we can move the key in $p$ into $c_r$, then we spread the empty node one layer up, and then we can continue the recursion.

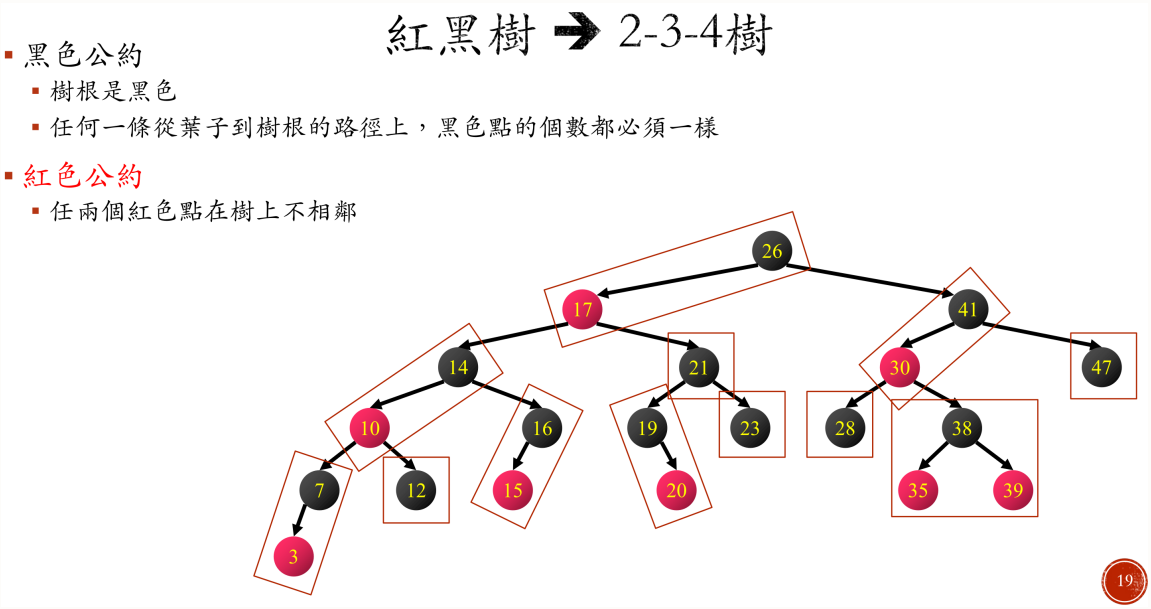
■

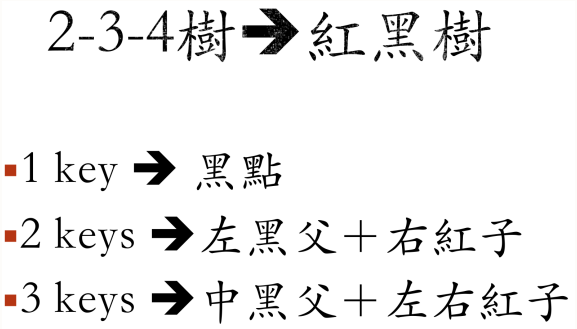
**Corollary 5.2.2.** 2-3 tree is a data structure for representing a set of at most $n$ distinct numbers:

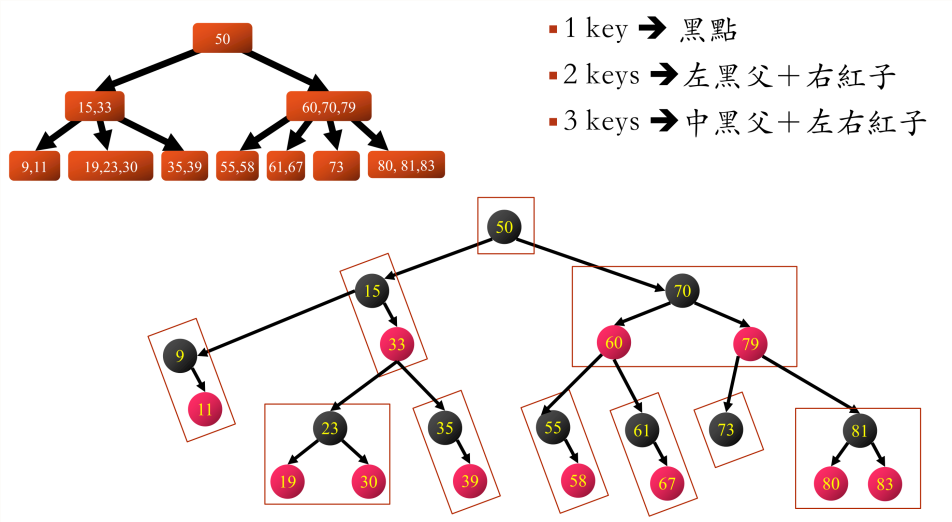
- $O(n)$ space.
- $O(1)$ time for initialization.
- $O(\log n)$ time for membership query.
- $O(\log n)$ time for insertion.
- $O(\log n)$ time for deletion.

**Theorem 5.2.3.** We can reduce a 2-3-4 tree to a red-black tree and a red-black tree to a 2-3 tree in $O(n)$ time.

**Proof.** Consider the following construction:




■

# Chapter 6

# Hash Function

**Example 6.0.1.** Suppose that we want to represent an initially empty set $S$ of at most $n$ numbers such that each element of $S$ is positive integer no more than $n$. We would like to support the following operations on $S$ for any given integer $i$ with $1 \leq i \leq n$:

- membership: determining whether $i$ belongs to $S$.
- insertion: inserting $i$ into $S$.
- deletion: deleting $i$ from $S$.

**Attempt #1.** Maintaining $S$ using a search tree having height $O(\log n)$ like $B$-tree, red-black tree, or AVL-tree:

- Space: $O(n)$.
- Time:
  - $O(1)$ time for initialization.
  - $O(\log n)$ time for membership query.
  - $O(\log n)$ time for insertion.
  - $O(\log n)$ time for deletion.

■

**Attempt #2.** Keep all the elements of $S$ in an unordered array $A[1 \ldots n]$.

- Space: $O(n)$.
- Time:
  - creation and initialization: $O(1)$.
  - membership: $O(n)$ (linear search).
  - insertion: $O(1)$.
  - deletion: $O(n)$.

■

**Attempt #3.** Keep all the elements of $S$ in an sorted array $B[1 \ldots n]$.

- Space: $O(n)$.
- Time:
  - creation and initialization: $O(1)$.
  - membership: $O(\log n)$ (binary search).

  - insertion: $O(n)$.
  - deletion: $O(n)$.

■

**Attempt #4.** Keep all the elements of $S$ in a binary array $C[1 \dots n]$. Specifically, for each $j = 1, \dots, n$, we maintain the condition that $C[j] = 1$ holds if and only if $S$ contains element $j$.

- Space: $O(n)$.
- Time:
  - creation and initialization: $O(n)$.
  - membership: $O(1)$ (direct look-up).
  - insertion: $O(1)$.
  - deletion: $O(1)$.

■

Now if our problem becomes

**Example 6.0.2.** Suppose that we want to represent an initially empty set $S$ of at most $n$ numbers such that each element of $S$ is positive integer no more than $m$ (with $m \geq n$). We would like to support the following operations on $S$ for any given integer $i$ with $1 \leq i \leq m$:

- membership: determining whether $i$ belongs to $S$.
- insertion: inserting $i$ into $S$.
- deletion: deleting $i$ from $S$.

Then, if there is a function $H : [m] \to [n]$ where for all $x \in [n]$, there exists $O(1)$ distinct $k \in [m]$ s.t. $H(k)$ has the same value, then we can still use $O(n)$ space to implement Attempt # 4.

**Remark 6.0.1.** It is easy to achieve this assumption when $m = O(n)$ (use modulo function), but for $m = \omega(n)$, it is impossible.

## 6.1 Randomized Algorithm and Communication Complexity

**Problem 6.1.1.**

- Input: two $n \times n$ matrices $A$ and $B$.
- Output: determining whether $A = B$.

**Remark 6.1.1.** The time complexity is $\Theta\left(n^2\right)$, but here we care the communication complexity: Imagine that $A$ and $B$ are at two different planet far from each other, so transmitting numbers is expensive, and thus we want the complexity of the data transmitted to be as low as possible, and other computations taken in each planet is not that important. Hence, we can finish under $O\left(n^2\right)$ communication complexity, and now we are curious about whether we can reduce the communication complexity to $o\left(n^2\right)$ if we permit small percentage of error. After all, interstellar communication has not low percentage of error.

**Algorithm 6.1:** Fingerprinting Approach

1 Choose an $n$-element column vector $r$, where each element of $r$ is either 0 or 1 independently and equally likely.
2 Compare $Ar$ and $Br$, and output whether they are identical.

**Remark 6.1.2.** As a matter of fact, 0 and 1 can be any two distinct numbers, and the commmucation complexity is $O(n)$ since we just need to transmit $r$ and $Ar$ from planet $A$ to planet $B$.

**Theorem 6.1.1.**

$$\Pr(Ar = Br \mid A = B) = 1$$
$$\Pr(Ar = Br \mid A \neq B) \leq 0.5.$$

**Proof.** The first equation is trivial. Now we show the second one. Let $C = A - B$, then we want to show

$$\Pr\left(Cr = 0^n \mid C \neq 0^{n\times n}\right) \leq 0.5.$$

Assuming $C \neq 0^{n\times n}$, then there is a row $i$ of $C$ whose nonzero elements are $C_{i,j_1}, C_{i,j_2}, \ldots, C_{i,j_\ell}$ for all $\ell \geq 1$. Thus,

$$\Pr\left(Cr = 0\right) \leq \Pr\left(\sum_{k=1}^{\ell} C_{i,j_k} r_{j_k} = 0\right),$$

so we can just show

$$Pr\left(\sum_{k=1}^{\ell} C_{i,j_k} r_{j_k} = 0\right) \leq 0.5.$$

Note that

$$\Pr\left(\sum_{k=1}^{\ell} C_{i,j_k} r_{j_k} = 0\right) = \sum_{\substack{(r_{j_1}, r_{j_2}, \ldots, r_{j_{\ell-1}}) \\ \text{determined}}} \Pr\left(\sum_{k=1}^{\ell} C_{i,j_k} r_{j_k} = 0\right) \cdot \frac{1}{2^{j_\ell - 1}},$$

and since for each determined $(r_{j_1}, \ldots, r_{j_\ell - 1})$ pair, we know

$$\Pr\left(\sum_{k=1}^{\ell} C_{i,j_k} r_{j_k} = 0\right) \leq 0.5,$$

which is because $C_{i,j_\ell} \neq 0$, and this shows

$$\Pr\left(\sum_{k=1}^{\ell} C_{i,j_k} r_{j_k} = 0\right) = \sum_{\substack{(r_{j_1}, r_{j_2}, \ldots, r_{j_{\ell-1}}) \\ \text{determined}}} \Pr\left(\sum_{k=1}^{\ell} C_{i,j_k} r_{j_k} = 0\right) \cdot \frac{1}{2^{j_\ell - 1}} \leq \sum_{\substack{(r_{j_1}, r_{j_2}, \ldots, r_{j_{\ell-1}}) \\ \text{determined}}} \frac{0.5}{2^{j_\ell - 1}} = 0.5,$$

and we're done. ■

**Question.** Is the error rate 0.5 too high?

**Answer.** If we repeat this algorithm $t = O(1)$ times, then we can maintain the communication complexity at $O(tn) = O(n)$ with the error rate not more than

$$\frac{1}{2^t}.$$

⊛

**Problem 6.1.2.**

- Input: three $n \times n$ matrices $A, B, C$.
- Output: determing whether $AB = C$.

**Answer.** Use last algorithm to give $t = O(1)$ distinct $r$ and check

$$ABr = Cr,$$

then we can reduce the time complexity of this problem to $O\left(n^2\right)$ with error rate at most $\frac{1}{2^t}$. ⊛

**Problem 6.1.3.**

- Input: Degree $n$ polynomial $A(x)$ and $B(x)$ and a degree $2n$ polynomial $C(x)$.
- Output: Determine whether the product $P(x)$ of $A(x)$ and $B(x)$ is equal to $C(x)$.

**Remark 6.1.3.** By the definition of multiplication of polynomials, we need $O\left(n^2\right)$ time. If using Fast Fourier Transform, then we can compute within $O(n \log n)$ time. Also, we need $O(n)$ time to check whether $P(x) = C(x)$.

**Algorithm 6.2:** 隨機指紋術 (again)

1 Let $R$ consist of arbitrary $4n$ distinct numbers. Choose a number $r$ from $R$ uniformly at random. Evaluate

$$A(r) \cdot B(r) = C(r)$$

in $O(n)$ time. Report whether the equality holds.

**Theorem 6.1.2.**

$$\Pr\left(A(r)B(r) = C(r) \mid A(x)B(x) = C(x)\right) = 1$$
$$\Pr\left(A(r)B(r) = C(r) \mid A(x)B(x) \neq C(x)\right) \leq 0.5$$

**Proof.** Let $D(x) = A(x)B(x) - C(x)$. If $D(x) \neq 0$, then its degree is at most $2n$, and thus $D(x)$ has at most $2n$ roots. Now that $r$ is chosen out of $4n$ numbers, then the probability of $D(r) = 0$ is not more than 0.5, so we're done. ■

# Chapter 7

# P and NP

## Lecture 4

4 Dec. 14:20

**Problem 7.0.1.** If the solution of a problem can be verified quickly, then can we find the solution quickly, too?

**Definition 7.0.1.**

- P (Polynomial time): The problems that can be solved in polynomial time. e.g. sorting.
- NP (Nondeterministic Polynomial time): The problems that can be verified whether a solution is true or not in polynomial time. e.g. sudoku, traveling salesman problem.

**Remark 7.0.1.** We call P the class of all problems that can be solved in polynomial time, while NP consists of the problems that can be solved in non-deterministic polynomial time.

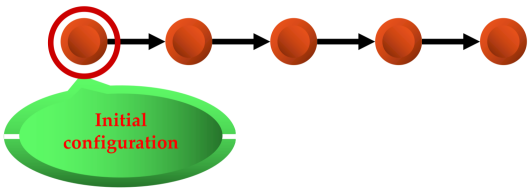



Figure 7.1: Deterministic algorithm

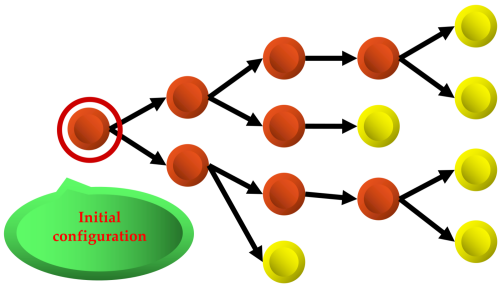



Figure 7.2: Non-deterministic algorithm

**Algorithm 7.1:** Non-deterministic sort

```
1 for i = 1 to n do
2     for j = 1 to n − 1 do
3         Non-deterministically, either exchange A[j] and A[j + 1] or do nothing.
```

**Remark 7.0.2.** This is not a random algorithm.

**Problem 7.0.2** (Vertex Cover Problem)**.**

- Input: A graph $G$ and an integer $k$.
- Output: Determine whether $G$ admits a set of at most $k$ vertices that cover all edges of $G$.

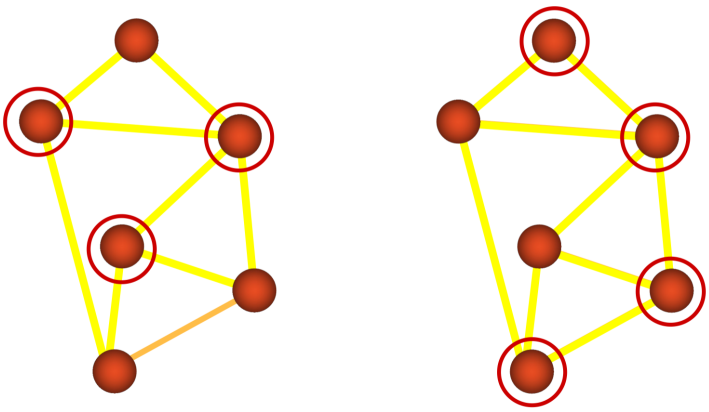


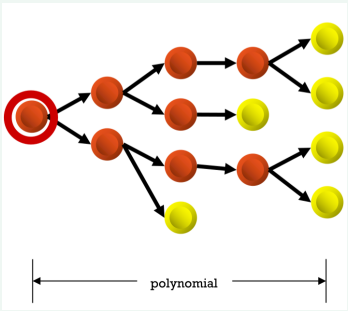
Figure 7.3: Vertex Cover Problem

**Algorithm 7.2:** A non-deterministic algorithm for vertex cover problem

1 Let $S = \varnothing$.
2 For each $x \in V(G)$, insert $x$ non-deterministically into $S$.
3 If $|S| \leq k$ and $S$ is a vertex cover of $G$, then output yes. Otherwise, output no.

**Remark 7.0.3** (The standard of correctness of a non-deterministic algorithm)**.** If the correct answer is yes, then there is a computation path of the algorithm that leads to yes. If the correct answer is no, then all computation paths of the algorithm lead to no.

**Definition 7.0.2.** We say that a non-deterministic algorithm $N$ runs in polynomial time if for any input $x$ of $N$, any computation on $x$ takes time polynomial in the size of $x$.




**Theorem 7.0.1.** P $\subseteq$ NP.

**Theorem 7.0.2.** Each NP problem can be solved in (deterministic) exponential time.

**Remark 7.0.4.** If P=NP, then each non-deterministic polynomial time algorithm has a deterministic version that runs in polynomial time. Thus,

- If P=NP, then all hardest problems in NP can be solved by polynomial time (deterministic) algorithm.
- If P$\neq$NP, then there is a hardest problem in NP that cannot be solved by any polynomial time (deterministic) algorithm.

**Definition 7.0.3.** A decision (i.e. yes-no) problem $\prod$ is polynomial time verifiable if there is a

deterministic polynomial time algorithm $V$ s.t. the following statement holds for each instance $x$ of $\prod$:

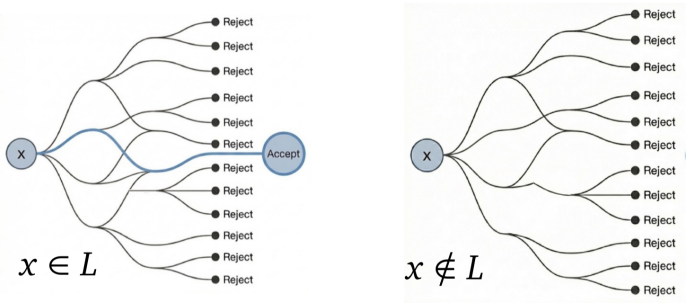
- $x$ is a YES-instance of $\prod$ if and only if there is a certificate $y$ with $|y| \leq |x|^{O(1)}$ s.t. $V(x, y) =$ "Yes".

**Theorem 7.0.3.** A problem $\prod$ is NP if and only if $\prod$ is polynomial-time verifiable.

Thus, we have two equivalent definition for NP:

- One if based on the computational power of Non-deterministic Turing Machine (NTM), which emphasize the ability to generate/guess the solution in polynomial time.
- The other one is based on the correctness of the verifier, which emphasize the ability to verify a purported solution.

**Remark 7.0.5.** NTM is a non-deterministic algorithm with $O(1)$ degree of nondeterminism, where degree of nondeterminism is the maximal branch number of below figure.




**Corollary 7.0.1.** A language $L \in$NP iff there exists a NTM $M$ deciding $L$ in polynomial time, so $L$ is the set of all Yes-instance, and for all $y \notin L$, $y$ is a No-instance.

**Corollary 7.0.2.** A language $L \in NP$ iff there exists a deterministic verifier $V$ and a polynomial $p$ s.t.

$$L = \{x \mid \exists y,\ |y| \leq p(|x|) \text{ and } V(x, y) = 1\},$$

where $y$ is called the certificate of $x$.

Now we prove the equivalence of two definition of NP (Theorem 7.0.3).

**Proof.**

($\Rightarrow$) Suppose we have a verifier $V$ and its corresponding polynomial $p$, then we can construct a NTM $M$ decides the language. Suppose the length of the input $x$ is $n$. We first non-deterministically choose a string $y$ of length at most $p(n)$, then we call $V(x, y)$. If $V(x, y) = 1$, then $M$ accept, otherwise $M$ reject. Note that $M$ decides the language since $M$ accept the input $x$ iff $y$ s.t. $V(x, y) = 1$ exists. Also, select $y$ takes $O(p(n))$ time and verify $V(x, y)$ takes polynomial time, so the total time is still polynomial.

- Now if we have a NTM $M$ decides $L$ in polynomial time, then we want to construct a polynomial verifier $V$. In this case, the certificate is the path on $M$ from beginning configuration to accept configuration, and we can verify this path by simulating it, we know this takes polynomial time.

■

## 7.1 NP-completeness and NP-hardness

Note that if a problem is an NP problem, then it is highly likely a very difficult problem. Also, if we say a problem is NP-hard, then this problem is at least as hard as all the problems in NP.

**Definition 7.1.1.** A problem $X$ is NP-hard if the following condition holds: If $X$ can be solved in (deterministic) polynomial time, then all problems in NP can be solved in (deterministic) polynomial time.

**Remark 7.1.1.** This does not rule out that the possibility that $X$ ias an unsolvable problem, i.e. a problem that cannot be solved by any algorithm (like Halting Problem).

**Definition 7.1.2.** A problem is NP-complete if it is an NP-hard problem that is also in NP.

**Remark 7.1.2.** Thus, an NP-complete problem is one of the hardest problems in the class NP. That is, an NP-complete problem is a hardness representative of the class NP.

**Remark 7.1.3.** Suppose $X$ is an NP-complete problem.

- If one proves that $X$ can be solved by a polynomial-time algorithm, then P=NP.
- If one proves that $X$ cannot be solved by any polynomial-time algorithm, then P≠NP.

**Problem 7.1.1** (Satisfiability).

- Input: A boolean formula with variables.
- Output: Whether there is a truth assignment for the variables that satisfies the input boolean formula.

**Remark 7.1.4.** SAT is the first known NP-complete problem.

**Corollary 7.1.1.**

- If one proves that SAT can be solved by a polynomial-time algorithm, then P=NP.
- If one proves that SAT cannot be solved by any polynomial-time algorithm, then P≠NP.

## 7.2 Polynomial-time reduction

**Definition 7.2.1.** Problem $A$ can be reduced (in polynomial time) to problem $B$ if the following condition holds:

- If problem $B$ has a polynomial-time algorithm, then so does problem $A$.

**Remark 7.2.1.** We also call this condition as problem $B$ can be reduced from problem $A$.

## 7.3 Example for reduction

**Problem 7.3.1** (Hamiltonian Cycle).

- Input: An undirected graph $G$.

- Output: Whether or not $G$ contains a simple cycle that contains all vertices of $G$.

**Problem 7.3.2** (Hamiltonian Path).

- Input: An undirected graph $G$ and two vertices $s, t$ of $G$.
- Output: Whether or not $G$ contains a simple $st$-path that contains all vertices of $G$.

**Corollary 7.3.1.** Hamiltonian cycle problem can be reduced to Hamitonian path problem.

**Proof.** Let $G$ be an input instance for the Hamiltonian Cycle problem. Suppose that the Hamiltonian Path Problem has a polynomial-time algorithm $B$, then we can design a polynomial-time algorithm $A$ for the Hamiltonian cycle problem:

**Algorithm 7.3:** Finding Hamiltonian Cycle

1 Run $B(G, s, t)$ for each edge $st$ of $G$. If all iterations return no, then answer no. Otherwise, answer yes.

Note that $G$ has a Hamiltonian cycle iff there exists $s, t$ s.t. an $st$-Hamiltonian path exists and $s, t$ are adjacent. ■

**Remark 7.3.1.** Since Hamiltonian Cycle is one of the Karp's 21 NP-complete problems, so the Hamiltonian Path problem is also NP-complete.

**Problem 7.3.3** (Longest Path Problem).

- A graph $G$ and two vertices $u$ and $v$ of $G$.
- A longest simple $uv$-path of $G$.

**Remark 7.3.2.** The problem is in NP since for given $u, v$, we can non-deterministically select a list of non-repeatitive vertices of $G$ starting from $u$ and ending at $v$, then we can check whether this list form a simple $uv$-path in polynomial time.

**Corollary 7.3.2.** We can reduce the Hamiltonian Path Problem to Longest Path Problem in polynomial time, so Longest Path Problem is also NP-complete.

**Proof.** Suppose $(G, u, v)$ is an input instance for the Hamiltonian Path problem. Suppose that the Longest Path Problem has a polynomial-time algorithm $B$, then we can just run $B(G, u, v)$ to check whether the longest simple $uv$-path passes all vertices of $G$. If yes, then return yes, otherwise return no. ■

**Problem 7.3.4** (Vertex Cover Problem).

- Input: A graph $G$ and an integer $k$.
- Output: Determine whether $G$ admits a set of at most $k$ vertices that cover all edges of $G$.

**Remark 7.3.3.** Vertex Cover Problem is one of Karp's 21 NP-complete problem.

**Problem 7.3.5** (Independent Set/Stable Set).

- A graph $G$ and an integer $k$.
- Determine whether $G$ contains a subset $S$ of $V(G)$ with $|S| \geq k$ any two vertices in $S$ are not adjacent in $G$.

**Lemma 7.3.1.** For each subset $S$ of $V(G)$, $S$ is a vertex cover of $G$ if and only if $V(G) \setminus S$ is an independent set of $G$.

**Proof.** If $S$ is a vertex cover of $G$, then any edge of $G$ has at least one endpoint in $S$. Equivalently, for eny edge of $G$, at most one endpoint is in $V(G) \setminus S$, so $V(G) \setminus S$ is an independent set.

If $V(G) \setminus S$ is an independent set, then we can use similar argument to show $S$ is a vertex cover. ■

**Theorem 7.3.1.** We can reduce Independent Set Problem to Vertex Cover Problem.

**Proof.** Let $(G, k)$ be an input instance for the independent set problem. Suppose that the Vertex Cover Problem has a polynomial-time algorithm $B$, then we can run $B(G, |V(G)| - k)$ to check whether there is a vertex cover of at most $|V(G)| - k$ vertices. If yes, then there is an independent set of at least $k$ vertices, so we output yes, otherwise we output no. ■

## 7.4 A less obvious example

**Definition.** If $x$ is a boolean variable

**Definition 7.4.1.** $x$ and $\overline{x}$ are literals.

**Definition 7.4.2.** If $y_1, y_2, \ldots, y_m$ are literals, then

$$y_1 \vee y_2 \vee \cdots \vee y_m$$

is a clause.

**Definition 7.4.3.** If $C_1, C_2, \ldots, C_n$ are clauses, then

$$C_1 \wedge C_2 \wedge \cdots \wedge C_n$$

is a CNF (conjunctive normal form).

**Definition 7.4.4.** A $k$-CNF is a CNF each of whose clause has at most $k$ literals.

**Remark 7.4.1.** Each boolean formula has an equivalent CNF.

**Problem 7.4.1** ($K$-SAT).

- Input: A $k$-CNF $\phi$.
- Output: Determine whether $\phi$ is satisfiable.

**Example 7.4.1.**

$$\phi = (x \vee y \vee \overline{z}) \wedge (\overline{x} \vee z \vee w) \wedge (x \vee w)$$

is a YES-instance of 3-SAT, while

$$\phi = (x \vee \overline{y}) \wedge (\overline{x} \vee y) \wedge (\overline{x} \vee \overline{y}) \wedge (x \vee y)$$

is a NO-instance of 2-SAT.

**Definition 7.4.5.** We define the graph $G(\phi)$ for a 3-CNF $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_n$ by:

- For each $C_i = \alpha \vee \beta \vee \gamma$, we construct a triangle on three new vertices $\alpha, \beta, \gamma$.
- For any two vertices $x$ and $\overline{x}$ of the same variable $x$ that are complement to each other, we add an edge $x\overline{x}$ between them.

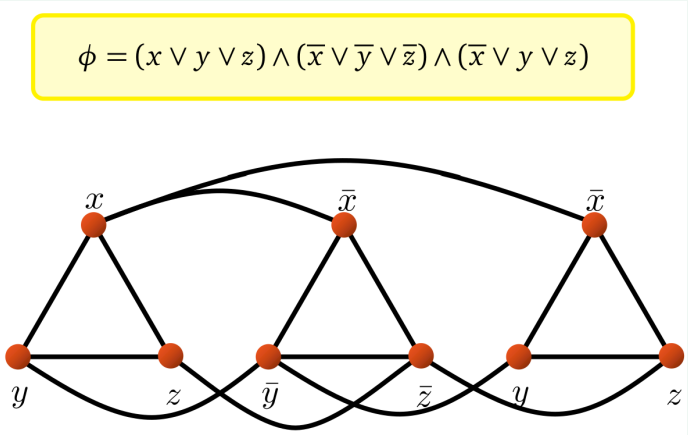



Figure 7.4: $G(\phi)$ for $\phi = (x \vee y \vee z) \wedge (\overline{x} \vee \overline{y} \vee \overline{z}) \wedge (\overline{x} \vee y \vee z)$.

**Lemma 7.4.1.** The 3-CNF $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_n$ is satisfiable if and only if the graph $G(\phi)$ has an independent set of size $n$.

**Proof.**

($\Rightarrow$) If a truth assignment $T$ satisfies $\phi$, then $T$ satisfies at least one literal in each clause. We choose an arbitrary one satisfied literal from each triangle. The chosen vertices have to be pairwise non-adjacent in $G(\phi)$ since all vertices are chosen from different triangle and if $x$ is true, then $\overline{x}$ cannot be true.

($\Leftarrow$) Let $S$ be an independent set of $G(\phi)$ of size $n$. Each triangle has exactly one vertex in $S$. For each literal $\alpha \in S$, we assign $T(\alpha)$ to be true and $T(\overline{\alpha})$ to be false, then $T$ is a truth assignment that satisfies $\phi$.

■

**Theorem 7.4.1.** We can reduce 3SAT to Independent Set problem.

**Proof.** If $\phi$ is an input instance for the 3SAT problem, and independent set problem has a polynomial-time algorithm $B$, then we can run $B(G(\phi), n)$ and report the answer obtained. ■

## 7.5 Unsolved problem

There are two famous problem such that we don't know they are NP or NP-complete.

**Problem 7.5.1** (Factorization). Factorize a non-prime $N$ into product of two factors.

**Problem 7.5.2** (Graph Isomorphism). Determine whether a matrix $A$ is an adjacency matrix of a graph $G$.

**Remark 7.5.1.** Subgraph isomorphism problem

- Determine if a input matrix $A$ is an adjacency matrix of a subgraph of the input graph $G$.

is NP-complete. (Reduce from Hamiltonian cycle)

# Appendix