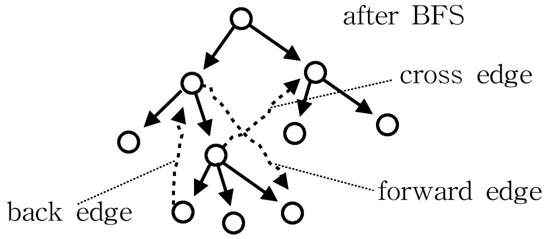
Graph Algorithms

Classification of edges

- *tree edge*
- *back edge*
- *forward edge*
- *cross edge*



Undirected graph

	forward edge = back edge	cross edge
BFS tree	×	○
DFS tree	○	×

Directed graph

	forward edge	back edge	cross edge
BFS tree	×	○	○
DFS tree	○	○	○

Topological sort

A directed acyclic graph(dag) is a digraph without any cycle.

Let $G=(V,E)$ be a dag.

A topological sort of G is an arrangement of the vertex set s.t if $(i,j)\in E$ then i appears before j in the arrangement.

```
Topological-sort(x)
mark[v] ← visited;
∀w∈L(v) // L(v) = adjacency list of v
    if(mark[v]=unvisited)
        Topological-sort[w];
output v;
```

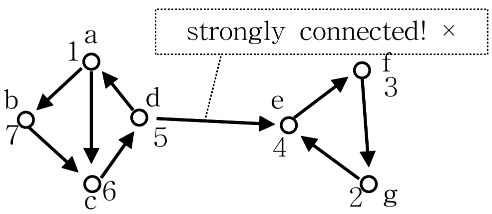
The alg. gives a reverse topological order

Strongly connected component

Let $G=(V,E)$ be a digraph.

A strongly connected component of G is a maximul set of vertices in which there is a directed path between any two vertices in the net.

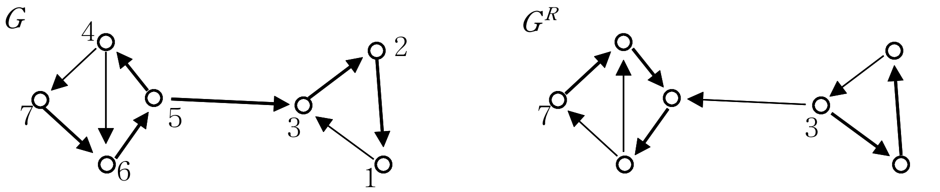
* A digraph is said to be strongly connected if it has only one strongly connected component.



strongly connected component : 2개

Strongly-Connected-Component(G)

1. Call $DFS(G)$ to compute finishing time $f[u]$, $\forall u \in G$
2. Construct G^R by reversing all the edges of G
3. Call $DFS(G^R)$ at the vertex with the highest finishing time;
   if it doesn't cover all the vertice, then call DFS again at the vertex with the highest finishing time among the remaining vertices; ⋯
4. Each tree resulted by step 3 is a strongly connected component



Claim

$v \& w$ are in the same strongly connected component

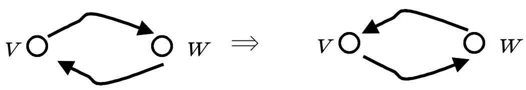
iff $v \& w$ are in the same tree in the DFS forest of step 3

proof.

⇒ easy (trivial)

Assume $v \& w$ are in the same strongly connected component

Then $\exists(v \xrightarrow{*} w)$ and $\exists(w \xrightarrow{*} v)$ in G



Hence $\exists(w \xrightarrow{*} v)$ and $\exists(v \xrightarrow{*} w)$ in G

Suppose DFS in G^R starts at some vertex x and reaches v (or w),

then it also reaches w (or v);

i.e they are in the same tree in the DFS forest.

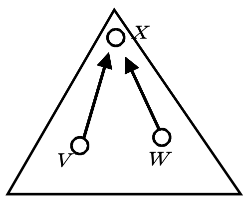
⇐

Assume $v \& w$ are in the same tree in the DFS forest of G^R

$\exists(x \xrightarrow{*} v)$ in G^R ⇒ $\exists(v \xrightarrow{*} x)$ in G

Suppose, for the contradiction that $\exists$ no path $x \xrightarrow{*} v$ in G.

Then $f[x] < f[v]$, contradiction! (Since $f[x] > f[v]$ by step 3)



Similarly, we can show that $\exists x \xrightarrow{*} w$ in G.

Therefore $\exists(v \xrightarrow{*} w)$ and $\exists(w \xrightarrow{*} v)$ in G

$v \& w$ are in the same strongly connected component.♥

Minimum Spanning Tree

Given a connected (undirected) graph $G=(V,E)$ and a weight function $W: E \to R$

A spanning tree T of G is a subgraph of G which includes all vertices of G and is a tree.

The weight of a spanning tree T is $W(T)=\sum_{e \in T} w(e)$

Objective

Given $G=(V,E)$ a connected graph, find a spanning tree with the minimum weight.

Fundamental principle

Thm.

Let $(S, V-S)$ be a bipartition of vertices.

If $\{u,v\}$ is an edge with the minimum weight among those edges crossing S and $V-S$, then there exists a minimum spanning tree containing $\{u,v\}$

Proof.

In any minimum spanning tree T (1) not containing $\{u,v\}$, there is only one path between u & v. The path contains at least one edge crossing S on $V-S$. Take such a crossing edge $\{x,y\}$, Note that $w(\{x,y\}) \ge w(\{u,v\})$
(given condition)

Case 1.

If $w(\{x,y\}) > w(\{u,v\})$, then T is not a minimum spanning tree
($\because$ The tree $T \cup \{u,v\} - \{x,y\}$ is less weighted than T) contradiction to (1)!

Case 2.

If $w(\{x,y\}) = w(\{u,v\})$, then $T \cup \{u,v\} - \{x,y\}$ has the same weight as T.
Hence $T \cup \{u,v\} - \{x,y\}$ is also a minimum spanning tree.

* If dege weights are all distinct, $\exists$ only one minimum spanning tree.
If not, $\exists$ can be more than on minimum spanning tree.

Kruskal's algorithm

Idea.

Build a minimum spanning tree T by adding minimum edges not generating cycles.

1. $T \leftarrow \emptyset$;
2. Initialize n singleton sets containing a vertex
3. sort the edges in nondecreasing weight order and put then in Q
4. While T has fewer than $|V|-1$ edges

    choose minimum cost edge $\{u, v\}$ in Q;

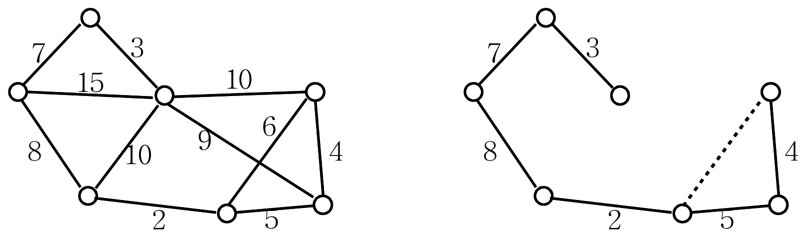
    delete $\{u, v\}$ from Q;

    if u and v belong to different sets

    then $T \leftarrow T \bigcup \{u, v\}$

    merge the two sets containing u and v;

e.g.



2 3 4 5 6 7 8 | 9 10 10 15

Running time

step 3 – $O(|E|\log|E|) = O(|E|\log|V|)$

step 4 – $O(|E|\log^*|E|)$

$\Rightarrow O(|E|\log|V|)$

Prim's algorithm

Idea.

Build a minimum spanning tree T by adding one vertex at a time to T

$Q \leftarrow V$; // Q contains vertices not in T

for each $u \in Q$

  $d[u] \leftarrow \infty$;

$d[r] \leftarrow 0$; // r : root node, chosen at random

while $Q \neq \emptyset$

  $u \leftarrow$ deleteMin(Q);

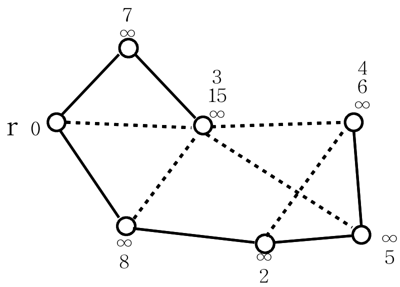
  for each $v \in L(u) \bigcap Q$;

    if($w(u, v) < d[v]$)

      $d[v] \leftarrow w(u, v)$;

      update the priority queue Q with respect to v; $\leftarrow O(\log|V|)$

$O(|E|)$



Shortest paths

Given a digraph $G=(V,E)$ with weight function $W:E\to R$

If P is a path consisting of edges $e_1, e_2, \cdots, e_k$, then $w(P)=\sum_{i=1}^{k} w(e_i)$

1. Single pair shortest path problem
   - Find the shortest path between a pair of vertices u and v
2. Single-source shortest path problem
   - Given source vertex $s\in V$, find the shortest paths from s to all other vertices in the graph
3. All pairs shortest path problem
   - Find the shortest paths between all pairs of vertices

Assumption

- If $(u,v)\notin E$, then $w(u,v)=\infty$
- If $\exists no$ directed path from u to v, then the algorithm should return ∞ as the weight of the shortest path
- $w(u,u)=0 \quad \forall u\in V$
- Denote by $\delta(u,v)$ the weight of the shortest path from u to v

Single-source shortest path

<u>Case 1. non-negative weight only</u>

* Relaxation

We keep an upper bound $d[v]$ of $\delta(s,v) \quad \forall v\in V$.

i.e. $d[v]\geq\delta(s,v)$

We may reduce $d[v]$ by performing a relaxation operation on an arbitrary edge (u,v)

If $d[v]>d[u]+w(u,v)$ then $d[v]\leftarrow d[u]+w(u,v)$

Dijkstra's algorithm - basically the same as Prim's algorithm for m.s.t.

$Q \leftarrow V$;
for each $u \in Q$
    $d[u] \leftarrow \infty$;
$d[s] \leftarrow 0$;
while $Q \neq \emptyset$
    $u \leftarrow$ deleteMin(Q);
    for each $v \in L(u) \cap Q$;
        if $d[v] > d[u] + w(u, v)$
            $d[v] \leftarrow d[u] + w(u, v)$
            update the priority queue Q with respect to v;

* correctness proof : use mathematical induction
* Running time : $O(|E| \log |V|)$

<u>Case 2. Allows negative-weight edges but no negative-weight cycle</u>

Bellman-Ford algorithm

$d[s] \leftarrow 0$;
for each $v \in V - \{s\}$
    $d[v] \leftarrow \infty$;
for $i \leftarrow 1$ to $|V| - 1$
    for each $(u, v) \in E$
        $d[v] \leftarrow \min\{\ d[v],\ d[u] + w(u, v)\ \}$;
negative-cycle-check();

negative-cycle-check()
    for each $(u, v) \in E$
        if $d[v] > d[u] + w(u, v)$;
        then output "no solution!";

Running Time $O(|V||E|)$

Correctness of checking negative weight cycles

Assume a negative cycle $v_0 v_1 \cdots v_k$ ($v_k = v_0$), i.e $\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$

Suppose, for the contradiction that the algorithm doesn't output "no solution!"

Thus, $d[v_i] \le d[v_{i-1}] + w[v_{i-1}, v_i]$, $i = 1, 2, \cdots, k$ by $d[v] \le d[u] + w(u, v)$

$\Rightarrow \sum_{i=1}^{k} d[v_i] \le \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w[v_{i-1}, v_i]$, $\sum_{i=1}^{k} d[v_{i-1}] = \sum_{i=1}^{k} d[v_i]$

$\Rightarrow 0 \le \sum_{i=1}^{k} w(v_{i-1}, v_i)$ contradiction

* Bellman-Ford algorithm is basically a dynamic programming although they doesn't explicitly notice it

  $d_{i,v}$: the shortest path length from s to v with at most i edges

  $$d_{0,s} = 0$$

  $$d_{0,v} = \infty \quad \forall u \in V - \{s\}$$

  $$d_{i,v} = \min_{(u,v)\in E} \{d_{i-1,u} + w(u,v)\} \quad i \geq 0$$

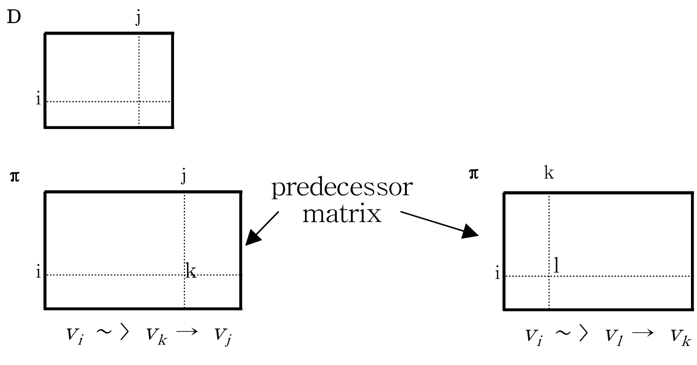
<u>Case 3. Directed acyclic graph, DAG</u>

Topologically sort the vertices;
$d[s] \leftarrow 0$;
for each $v \in V - \{s\}$
  $d[v] \leftarrow \infty$;
for each $u \in V$ in topological order
  for each $v \in L(u)$
    if $d[v] > d[u] + w[u,v]$
    then $d[v] \leftarrow d[u] + w[u,v]$;

Running time : $O(|E|)$ assume $|E| = \Omega(|V|)$

All-pairs shortest path

Compute shortest paths between all pairs of vertices



* Negative-weight edges are allowed. But no negative cycle is allowed.
* Useful for, e.g, road atlas
* A naive solution
  - Apply Bellman-Ford $|V|$ times $\Rightarrow$ $O(|V|^2|E|)$

Floyd-Warshall algorithm : $O(|V|^3)$ by dynamic programming

An optimal substructure (but inefficient)

$d_{ij}^{(m)}$: the minimum weight of paths from v_i to v_j that contain at most m edges.

$$d_{ij}^{(0)} = \begin{cases} 0 & if\ i=j \\ \infty & otherwise \end{cases}$$

$$d_{ij}^{(m)} = \min_{1 \le k \le n} \{ d_{ik}^{(m-1)} + w_{kj} \} \rightarrow O(|V|^4)$$

$$\text{cf. } d_{ij}^{(m)} = \min_{(k,j) \in E} \{ d_{ik}^{(m-1)} + w_{kj} \} \rightarrow O(|V|^2 |E|)$$

improvement

⇓

Floyd-Warshall algorithm

Let $V = \{ v_1, v_2, \cdots, v_n \}$

$d_{ij}^{(k)}$: the minimum height of paths from v_i to v_j using only vertices $\{ v_1, v_2, \cdots, v_k \}$ as intermediate vertices

$$d_{ij}^{(k)} = \begin{cases} w_{ij} \\ \min\{ d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \} & if\ k \ge 1 \end{cases}$$

Floyd-Warshall(W)

for $i \leftarrow 1$ to n

  for $j \leftarrow 1$ to n

    $d_{ij}^{(0)} \leftarrow w_{ij};$

for $k \leftarrow 1$ to n

  for $i \leftarrow 1$ to n

    for $j \leftarrow 1$ to n

      $d_{ij}^{(k)} \leftarrow \min\{ d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \}$

Running time : $O(|V|^3)$

※ Floyd : 1962.

Warshall before Floyd, 1962

Transitive Closure of a graph

$$d_{ij}^{(k)} = \begin{cases} 1 & if\ \exists\ a\ path\ v_i \sim\rangle v_j\ using\ \{ v_1, \cdots, v_k \} \\ 0 & otherwise \end{cases}$$

$$d_{ij}^{(k)} = d_{ij}^{(k-1)} \vee (d_{ik}^{(k-1)} \wedge d_{kj}^{(k-1)}) \rightarrow O(|V|^3)$$

Matrix multiplication

- Want to multiply two $n \times n$ matrices $A \times B$
- $C = AB, \; c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \rightarrow \Theta(n^3)$

○ Divide the matrices into four $\frac{n}{2} \times \frac{n}{2}$ matrices

Then $C = AB$ can be rewritten as

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & f \\ g & h \end{pmatrix}$$

where
$$\begin{aligned} r &= ae + bf \\ s &= ag + bh \\ t &= ce + df \\ u &= cg + dh \end{aligned}$$

$T(n) = 8T(\frac{n}{2}) + \Theta(n^2)$ by master's theorem $T(n) = \Theta(n^3)$. $n^{\log_2 8} = n^3$

○ Strassen's algorithm

$P_i = A_i B_i = (\alpha_{i1} a + \alpha_{i2} b + \alpha_{i3} c + \alpha_{i4} d)(\beta_{i1} e + \beta_{i2} f + \beta_{i3} g + \beta_{i4} h)$

$$\begin{aligned} P_1 &= a(g-h) \\ P_2 &= (a+b)h \\ P_3 &= (c+d)e \\ P_4 &= d(f-e) \\ P_5 &= (a+d)(e+h) \\ P_6 &= (b-d)(f+h) \\ P_7 &= (a-c)(e+g) \end{aligned} \qquad \begin{aligned} r &= P_5 + P_4 - P_2 + P_6 \\ s &= P_1 + P_2 \\ t &= P_3 + P_4 \\ u &= P_5 + P_1 - P_3 - P_7 \end{aligned}$$

The time $T(n) = 7T(\frac{n}{2}) + \Theta(n)$

by master's theorem $T(n) = \Theta(n^{\log_2 7})$

String matching

Given an alphabet Σ, a text string $x \in \Sigma^*$, and a pattern string $p \in \Sigma^*$.

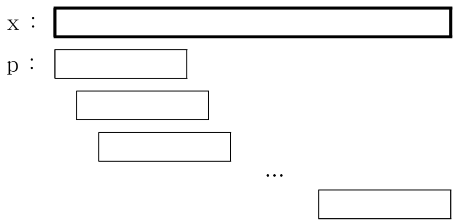
We want to find the first occurrence or all occurrence of p in x.

Application : Search in editors, DNA sequence, approaximate matching

$x[1], \quad \cdots \quad , x[n] \qquad |x| = n$

$p[1], \cdots, p[n] \qquad |p| = m \quad n \geq m$

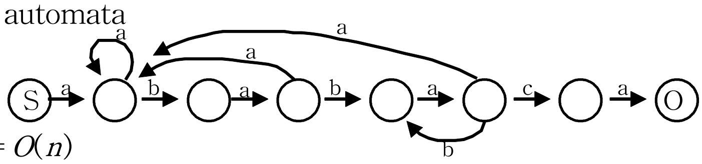
An elementary-school algorithm



running time : $\Theta(nm)$

An intermediate thinking with automata

pattern → a b a b a c a



running time – $O(n + m|\Sigma|) = O(n)$

Knuth-Morris-Pratt algorithm (KMP) : $O(n+m) = O(n)$

- Effectively manipulate the relationship between x's suffix and p's prefix
- Problem with the elementary-school algorithm
  - every time a miss match occurs, restarts without using the info, gained so far
- Idea : use gained info , preprocessing the pattern string

KMP algorithm

$i \leftarrow 1$; // ptr to the text

$j \leftarrow 1$; // ptr to the pattern

while $j \leq m$ & $i \leq n$

  if $j = 0$ or $x[i] = p[j]$

  then { i++; j++; }

  else $j \leftarrow \pi[j]$;

if $j > m$ then "match at $i - j + 1$"; else "no match";

Running time

Everytime we go through the loop the algorithm, advances in the text(by i++) or shift the pattern by $j \leftarrow \pi[j]$

Note that $\pi[j] < j \quad \forall j$, so $j \leftarrow \pi[j]$ decreases j

thus, each time we go through the loop, $i + (i - j)$ will be increased by at least 1

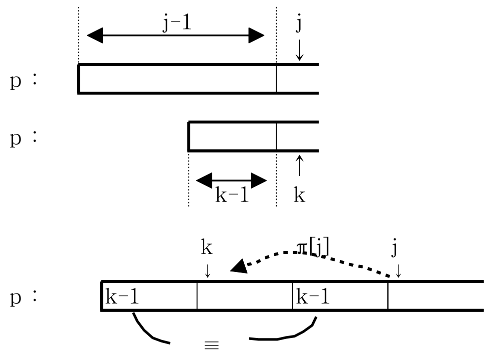
$i + (i - j) \leq 2i \leq 2(n+1)$

i.e. we go through the loop at most $2n$ time. Running time is $O(n)$

Preprocessing

```
j ← 1;
k ← 0;
π[1] ← 0;
while j≤m
    if( k=0 or p[j]=p[k])
    then { j++; k++; π[j]← k; }
    else k ← π[k];
```

Idea. match p against itself. Situation after j & k are incremented

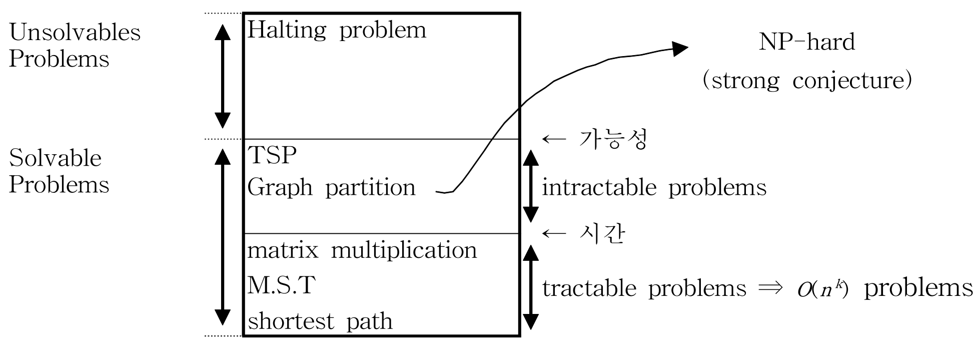


Time $O(m)$ ←Using the same technique as in the KMP alg.

$j+(j-k)\le 2j\le 2(m+1)$

Thus the running time of KMP is $O(n)$

NP-completeness



NP : non-deterministic polynomial

▽ NP-complete problems are a class of problems which most people believe will be a subset of more-than- $O(n^k)$ problems

▽ Decision problem : Optimization problem

Decision problem → Is there a path from u to v with $length \le k$?

Optimization problem → What is the shortest path length from u to v?

▼ The theory of NP-completeness

restricts attention to decision problem(Yes/No problem)

An intuitive definition

P : the class of problems that can be "decided" (by answering yes or no) in polynomial time
NP : the class of problems that can be "verified" (by answering only yes) in polynomial time

- decision : Say "yes" or "no" in any case
- verification : just say "yes", doesn't have to answer when the answer is "no"

A formal-language definition

$P=\{L\subseteq\Sigma^*|\exists\, an\ alg.\ A\ that\ decides\ L\ in\ polynomial\ time\}$

$NP=\{L\subseteq\Sigma^*|\exists\, an\ alg.\ A\ that\ verifies\ L\ in\ polynomial\ time\}$

An alternative definition

$P=\bigcup Time(N^k)$

$NP=\bigcup_{k\geq 0} NTime(N^k)$

where

$NTime(N^k)=\{L\subseteq\Sigma^*|L\ is\ accepted\ by\ some\ non\ deterministic\ Turing\ machine\ in\ time\ O(T)\}$

$Time(N^k)=\{L\subseteq\Sigma^*|L\ is\ decided\ by\ some\ deterministic\ Turing\ machine\ in\ time\ O(T)\}$

Poly-time verification example

Hamiltonian cycle of an undirected graph $G=(V,E)$ is a simple cycle that contains every vertex in V.

Hamiltonian-cycle problem

- Does G have a Hamiltonian cycle? or equivalently.
- $G\in HAM$ when $HAM=\{\langle G\rangle|G\ has\ a\ hamiltonian\ cycle\}$ ($\langle G\rangle$: G의 encoding)

* A simple deciding algrithm

- enumerate all $\approx(|V|-1)!$ permutations of the vertices and check to see if they contain a Hamiltonian cycle

a Hamiltonian cycle → $\Omega((|V|-1)!)$ at least $\Omega(2^{|V|-1})$: non-polynomial

* What if a certificate is given? (certificate : a seq. of vertices)

- proving that the certificate makes a Hamiltonian cycle is obviously easy.

→ $O(|V|^2)$ adjacency list 표현 가정

* $P\subseteq NP$ (trivial from definition),

$P=NP$? ← open question

conjecture : $P\neq NP$

Reducibility

A language L_1 is said to be "poly-time reducible" to a language L_2, denoted by $L_1 \le_p L_2$,

if $\exists$ a poly-time computable function $f: \Sigma^* \rightarrow \Sigma^*$ such that $\forall x \in \Sigma^*$, $x \in L_1 \Leftrightarrow f(x) \in L_2$

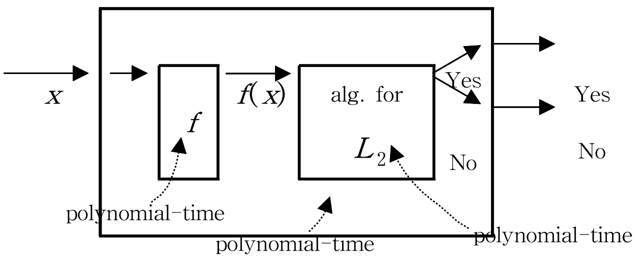
Intuitively

$L_1 \le_p L_2$ means that L_1 is no harder than L_2

Thm.

If $L_1 \le_p L_2$ and $L_2 \in P$, then $L_1 \in P$

<proof>



Thm.

If $L_1 \le_p L_2$ and $L_2 \in NP$, then $L_1 \in NP$

NP-completeness

A language $L (\in \Sigma^*)$ is NP-complete, if

1. $L \in NP$
2. $\forall A \in NP$, $A \le_p L$

* If a language satisfies at least the above 2, it is called NP-hard

Thm.

Let $L \in NP-complete$ and $L \in P$, then $P = NP$

<proof>

Want to show $P \subseteq NP$ & $NP \subseteq P$

$P \subseteq NP$: trivial

$NP \subseteq P$

Let $A \in NP$, then $A \le_p L$ by the def. of NP-completeness

Since $L \in P$, $A \in P$ by the previous theorem.

$\therefore$ $NP \subseteq P$

* meaning

If $\exists$ only one NP-complete problem that can be solved in poly. time, then all NP-complete problems (absolutely including NP-complete problems) can be solved in poly. time.

◎ Think about the difficulty to prove that a language is NP-complete by the definition of NP-completeness
→ devise a simple method

Thm. If $L \in NP$ and $\exists A \in NP-complete$ s.t $A \le_p L$, then $L \in NP-complete$

proof.

① $L \in NP$: given

② $\forall B \in NP,\ B \le_p L$?

Since A is NP-complete, $\forall B \in NP, B \le_p A$

Since $A \le_p L$, $B \le_p A \le_p L$ i.e $B \le_p L$

◎ To prove a language L to be NP-complete, we only have to show that

(1) $L \in NP$

(2) For a known NP-complete problem A, $A \le_p L$

◎ Starting point : we should have the 1st NP-complete problem

Def.

A **Boolean formula** is an expression containing Boolean variables and operations :

$\wedge$, $\vee$, $\rightarrow$, $\neg$, $\Leftrightarrow$

A boolean formula $\varnothing$ is **satisfiable** if $\exists$an assignment of Boolean values to its variables s.t the formula evaluates to true

e.g $((a \vee b \vee c) \wedge \overline{d}) \rightarrow (c \vee \overline{f})$: $d = 1$

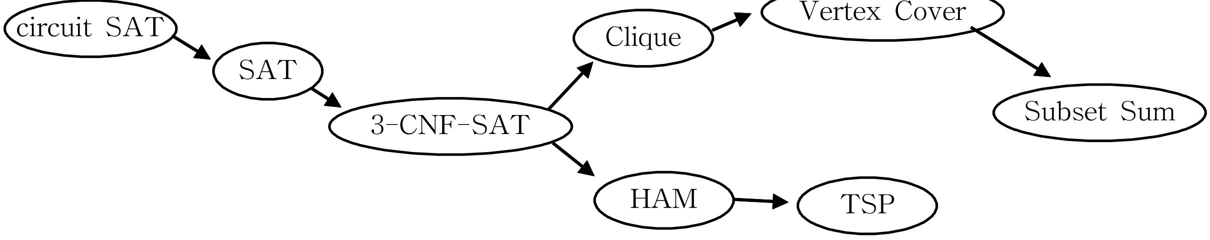
Def.

GSAT = $\{\langle \varnothing \rangle | \varnothing \textit{ is a satisfiable Boolean formula}\}$: General satisfiability

Thm. (Cook's Thm)

GSAT is NP-complete

※ The textbook uses circuit satisfiability problem as the 1st NP-complete problem. Either is Okay for the starting.



◎ To prove that a language L is NP-complete

(1) Show $L \in NP$

(2) Choose a known NP-complete problem L', describe an alg. for a function f that maps every instance of L' to an instance of L

(3) Show that the process of the above (2) can be done in poly. time

(4) Prove that $x \in L'$ iff $f(x) \in L$

※ Step (2), (3) → polytime transformation

Def.

A **literal** is either a Boolean variable or its negation.

A **clause** is a Boolean formula of the form $l_1 \vee l_2 \vee \cdots \vee l_k$ where each l_i is a literal.

A Boolean formula is said to be in **conjuntive normal form** (CNF) if it is of the form $c_1 \wedge c_2 \wedge \cdots \wedge c_m$ where each c_i is a clause.

e.g $(x_1 \vee \overline{x_2} \vee x_3) \wedge x_4 \wedge (\overline{x_4} \vee x_5)$

A Boolean formula is said to be in k-**CNF** if it is in CNF and each clause has at most k distinct literals (slightly different from the def. in the text).

Thm.

- SAT = $\{\langle \varnothing \rangle | \varnothing \text{ is a satisfiable Boolean formula in CNF}\}$ is NP-complete
- 3SAT = $\{\langle \varnothing \rangle | \varnothing \text{ is a satisfiable Boolean formula in } 3-CNF\}$ is NP-complete

proof.

① SAT & 3SAT $\in NP$

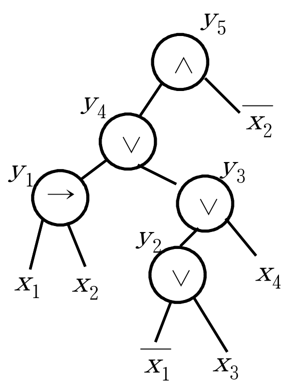
A satisfying assingment of variables can be verified in poly. time.

② GSAT $\leq_p$ SAT, GSAT $\leq_p$ 3SAT

Rewrite the original formula of GSAT
as a conjunction of formulas describing the operation.

For example, $\varnothing = ((x_1 \rightarrow x_2) \vee (\overline{x_1} \vee x_3) \vee x_4) \wedge \overline{x_2}$

Step 1. Introduce new variables(y_i's)



Step 2. Construct a conjunction $\varnothing_1$ of formulas each with at most 3 literals, defining how the subformulas and the new variables are related.

$\varnothing_1$ is

$$(y_1 \leftrightarrow (x_1 \rightarrow x_2)) \wedge (y_2 \leftrightarrow (\overline{x_1} \vee x_3)) \wedge (y_3 \leftrightarrow (y_2 \vee x_4)) \wedge (y_4 \leftrightarrow (y_1 \vee t_3)) \wedge (y_5 \leftrightarrow (y_4 \wedge \overline{x_2}))$$

Fact. $\varnothing$ is satisfiable iff $\varnothing_2 = \varnothing_1 \wedge y_5$ is satisfiable. ($\varnothing \equiv \varnothing_1 \wedge y_5$)

Step 3. Convert each subformula of $\varnothing_2$ into an equivalent 3-CNF formula.

Use the following rules :

$$A \rightarrow B \equiv \overline{A} \vee B$$
$$A \leftrightarrow B \equiv (\overline{A} \vee B) \wedge (A \vee \overline{B})$$
$$(A \wedge B) \vee (C \wedge D) \equiv (A \vee C) \wedge (A \vee D) \wedge (B \vee C) \wedge (B \vee D)$$

e.g

$$\begin{aligned}(y_1 \leftrightarrow (x_1 \rightarrow x_2)) &= (\overline{y_1} \vee (x_1 \rightarrow x_2)) \wedge (y_1 \vee (\overline{x_1 \rightarrow x_2})) \\ &= (\overline{y_1} \vee (\overline{x_1} \vee x_2)) \wedge (y_1 \vee (\overline{\overline{x_1} \vee x_2})) \\ &= (\overline{y_1} \vee \overline{x_1} \vee x_2) \wedge (y_1 \vee (x_1 \wedge \overline{x_2})) \\ &= (\overline{y_1} \vee \overline{x_1} \vee x_2) \wedge (y_1 \vee x_2) \wedge (y_1 \vee \overline{x_2})\end{aligned}$$

Let the result of Step 3 be a CNF $\varnothing_3$.

$\varnothing$ is satisfiable iff $\varnothing_3$ is satisfiable.

Finally, the transformation of Step 1, 2 & 3 can be done in poly. time w.r.t the length of the original formula.

Thm. EXACT-3SAT
$= \{\langle \varnothing \rangle \mid \varnothing$ *is a satisfiable Boolean formula in CNF with exactly* 3 *distinct literals per clause*$\}$
is NP-complete

proof.

① When a satisfying assingment of variables is provided, its satisfiability can be checked in poly. time. ∴ EXACT-3SAT ∈ NP

② Show $3SAT \le_p EXACT-3SAT$

Convert each clause of 3SAT(with "at most" 3 literals) into a conjunction of clauses with "exactly" 3 literals by the following rules :

$(l_1 \vee l_2 \vee l_3)$ form : Okay

$(l_1 \vee l_2) \equiv (l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \overline{p})$

$(l) \equiv (l \vee p_1 \vee p_2) \wedge (l \vee \overline{p_1} \vee p_2) \wedge (l \vee p_1 \vee \overline{p_2}) \wedge (l \vee \overline{p_1} \vee \overline{p_2})$

# of literals in the new formula ≤ 12 × # of literals in the original formula

It is obviously a poly-time transformation.

The original formula is satisfiable iff the new formula is satisfiable.

Thm. HAM = $\{\langle G \rangle \mid G$ *has a Hamiltonian cycle*$\}$ is NP-complete

Thm. Longest-Cycle = $\{\langle G, k \rangle \mid$ *Graph G has a simple cycle of length*$\ge k\}$ is NP-complete

proof.

① When a simple cycle is provided, it's easy(poly-time checkable) to verify that the $length \ge k$
∴ It is in NP

② $HAM \le_p Longest-Cycle$

Given an instance of HAM(a graph G), transform it to an instance G' of Longest-Cycle by assigning $w_e = 1,\ \forall e \in E$ using the same graph.

This transformation takes $\Theta(|E|)$ (poly-time)

$\exists$ a Hamiltonian cycle in G iff $\exists$ a simple cycle of $length \ge |V|$ in G'

Note. The transformation only has to provide 1-to-1 correspondence between their instances that preserves "yes" or "no" answers.

※ MAX-SNP-Complete

Def. Clique = $\{\langle G, k \rangle \mid G = (V, E)$ *is a graph* and G *has a clique of size*$\ge k\}$

* Remind : a clique is a complete subgraph

input : a graph $G = (V, E)$ and an integer k

question : Does G have a clique of $size \ge k$?

Max Clique : Given a graph $G = (V, E)$, find a clique of maximum size.

Thm. $Clique \in P$ iff Max Clique has a poly-time algorithm. proof. trivial

Thm. Clique is NP-complete

Heuristics !! problem space (=problem search space) / local optima (=attractor)

Thm. Clique is NP-complete

proof.

① $Clique \in NP$

Given a vertex set of $size \geq k$, check to see if it is a clique trivially in poly. time.

$\therefore$ $Clique \in NP$

② $EXACT-3SAT \leq_p Clique$

Given an instance of EXACT-3SAT $\emptyset = c_1 \wedge c_2 \wedge \cdots \wedge c_m$ where each $c_i = (l_{i1} \vee l_{i2} \vee l_{i3})$.

We want to construct an instance $G_\emptyset$ of Clique such that $\emptyset \in EXACT-3SAT$ iff $\langle G_\emptyset, m \rangle \in Clique$

$G_\emptyset = (V, E)$ is constructed as follows

V : we have one vertex for each literal in $\emptyset$, So $|V| = 3m$.

E : $\exists$an edge between $l_{ir} \& l_{js}$

if i) $i \neq j$ (i.e. l_{ir} and l_{js} are not in the same clause)

and ii) they are consistant. i.e. $l_{ir} \neq \overline{l_{js}}$

The construction can be done in poly. time. w.r.t the # of literals.

Claim. $\emptyset \in EXACT-3SAT \Leftrightarrow \langle G_\emptyset, m \rangle \in clique$

($\Rightarrow$) Assume $\emptyset \in EXACT-3SAT$

$\exists$an assignment that makes $\emptyset$ true. i.e each clause of $\emptyset$ has at least one literal with value 1 (true). Pick m vertices corresponding to these literals, one from each clause. The set of vertices forms a clique of size m in $G_\emptyset$ by the construction above.

($\Leftarrow$) Assume $\langle G_\emptyset, m \rangle \in Clique$ ($\exists$a clique of $size \geq m$ $(= m)$)

By the construction above, note that a clique in $G_\emptyset$ cannot contain two vertices derived from the same clause. Therefore the clique of size m has exactly one vertex derived from each clause.

Assign 1(true) to the literals corresponding to the vertices of the clique (of size m) in $G_\emptyset$.

Then by the construction above.

1. Each clause has a literal with value 1
2. The assignment is consistant (i.e. $\exists$no such edge $x - \overline{x}$)

so. $\emptyset$ is satisfiable i.e $\emptyset \in EXACT-3SAT$

recursive set - decidable
recursively enumerable set - verifiable
} without time factor

P - "quickly" decidable
NP - "quickly" verifiable
} with time factor

$CO-NP = \{L \subseteq \Sigma^* | \exists \text{ an alg. } A \text{ that verifies } \overline{L} \text{ in poly. time}\}$ cf. $\overline{L} \in NP$

$NP = CO-NP$ (?) open question

$P \subseteq NP \cap CO-NP$ (true)

$(NP \cap CO-NP) - P = \emptyset$ (?) open question

Thm. $NP \neq CO-NP \Rightarrow P \neq NP$

proof. Easy (use the fact that $P = CO-P$)

$P = NP \rightarrow P = CO-P \rightarrow NP = CO-NP$

Approximation

If a problem is known to be NP-complete, it is strongly believed that there is no poly time algorithm for it.
The best we can do is finding a poly-time approximation algorithm for it.

Approximation algorithm is for the optimization version of the problem.

For a minimization problem, the ratio bound $\rho(n)$ is a measure for an algorithm's performance such that $\frac{c}{c^*} \leq \rho(n)$

where

n : the size of the problem,

c^* : the optimal solution cost,

c : the cost of a solution produced by the algorithm

TSP (Traveling Salesman Problem)

$TSP = \{\langle G, k \rangle \mid G = (V, E)$ *is a complete graph with weighted edges* $, k \in Z$ and *G has a Ham Cycle of cost* $\leq k\}$

Thm. TSP is NP-complete

proof. $HAM \leq_p TSP$

$G \in HAM \Leftrightarrow \langle G', 0 \rangle \in TSP$

TSP with triangle inequality(metric TSP) ($w_{ij} \le w_{ik} + w_{kj}$, $\forall$ *cities* i, j, k)

1. NN (Nearest Neighbor alg.)

- start at a random city(vertex), keep visiting the nearest unvisited neighbor

Thm. The ratio bound for NN

$$\rho(n) = \frac{1}{2}(\lceil \log_2 n \rceil + 1) \quad \forall I$$

$$\frac{c}{c^*} > \frac{1}{3}\left(\log_2(n+1) + \frac{4}{3}\right) \quad \textit{for some large instance}$$

Guarantees almost nothing (이론적임, 실용성은 없는 알고리즘~ ^^;)

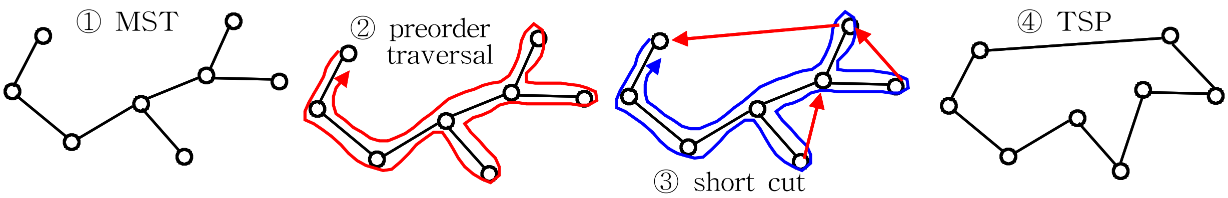
2. MST (Minimum Spanning Tree alg.)

- Construct a min. spanning tree T starting at a random vertex.
- Return the Ham. cycle H that visits the vertices in the order of a preorder traversal of T

Thm. The ratio bound $\rho(n)$ for MST

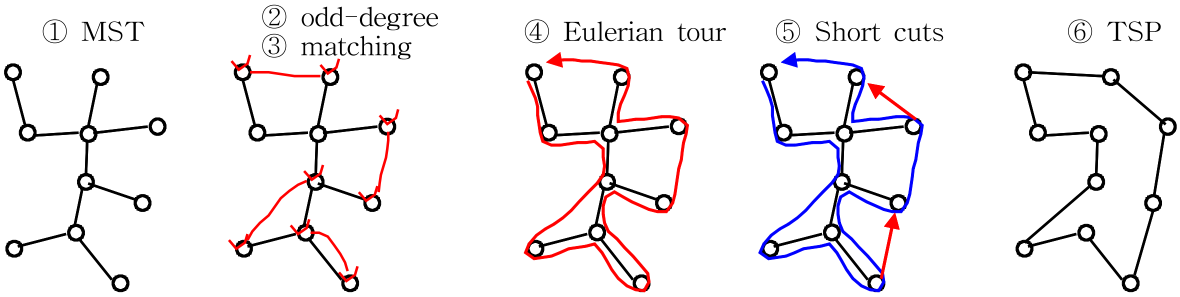
$$\rho(n) \le 2$$

proof. $c = cost(H) \le 2cost(T) \le 2c^*$



3. MM (Minimum-weight Matching alg.)

- Find a min. spanning tree T starting at a random vertex r
- Let V' be the set of odd-degree vertices in T.
  (The # of odd-degree vertices in a graph is always even)
- Find a matching of V' which has maximum cardinality and minimum weight, say M
- Add M to T (to get an Eulerian graph T')
- Find an Eulerian tour C_1 of T'
- Convert C_1 into a TSP tour C_2 (a Ham. cycle) by using short cuts.



* A matching of a graph G is a set of edges no two of which share an end point

Remind.

- An Eulerian tour of a graph is a cycle which contains every edge exactly once.
- Eulerian graph - every vertex has an even degree
- Every Eulerian graph has at least an Eulerian tour

Thm. The ratio bound $\rho(n)$ for MM

$$\rho(n) \leq \frac{3}{2}$$

proof.

Note. $cost(T) \leq c^*$
$cost(C_{1)} = cost(T) + cost(M)$

Claim. $cost(M) \leq \frac{1}{2} c^*$

proof.

An optimal tour $C_{opt}(cost\ c^*)$ can be changed to a tour C'_{opt} with only vertices in V' by using short cuts.

The $cost(C'_{opt}) \leq c^*$

Take alternate edges in C'_{opt} and then we have two matchings of V'.

Then the smaller of the two matchings must have $cost \leq \frac{1}{2} cost(C'_{opt})$

Since M is a min. weight matching of V'

$$\begin{aligned} cost(M) &\leq \text{the smaller matching above} \\ &\leq \frac{1}{2} cost(C'_{opt}) \leq \frac{1}{2} c^* \end{aligned}$$

therefore

$$cost(C_2) \leq cost(C_1) = cost(T) + cost(M) \leq \frac{3}{2} c^*$$

TSP without triangle inequality

Thm. If $P \neq NP$, $\exists$no poly-time approximation alg. with a ratio bound ρ for the TSP without triangle inequality.

proof.

Show that if $\exists$a poly-time approximatino alg. A with a ratio bound ρ, then the Ham cycle problem (a known NP-complete problem) is in P.

Given a Ham cycle problem instance $G = (V, E)$, we construct an instance of TSP $G' = (V, E')$ as follow :

$$w_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ \rho|V| + 1 & o.w \end{cases} \quad \leftarrow \text{amplification!!}$$

This is a poly time transformation.

Run alg. A on G', if A return a tour length $\leq \rho|V|$, then $\exists$a Ham. cycle in G, otherwise $\exists$no Ham. cycle

Thus HAM can be solved in poly time.

Contradiction as far as $P \neq NP$

※ TSP : combinatorial explosion

Some stochostic(추이적) Approximation Methods

▷ Simulated Annealing (SA)

```
s ← initial solution;
t ← initial temperature;
repeat
    repeat
        s' ← perturb( s);
        Δs ← cost( s') - cost( s);
        if( Δs<0 or random() < f(Δs,t))
            s ← s'; //accept
    until(time to change temperature)
    change t;
until(stopping condition)
```

▷ Genetic Algorithm (GA) : 유전 알고리즘

```
create a fixed # of initial solution;
repeat
    for i ← 1 to k
        choose two parent solution P1, P2 from the population;
        offspring_i ← cross over( P1, P2);
        offspring_i ← mutation( offspring_i);
        local-optimization( offspring_i); //optimal
    replace the whole or part of the population with offspring_1,…, offspring_k
until(stopping criterion)
return the best solution in the population;
```

▷ Large-step Markov Chain

```
s ← initial solution;
repeat
    s' ← perturb( s);
    s'' ← local-optimization( s');
    Δs ← cost( s'') - cost( s);
    if( Δs<0)        ← SA적인 acceptance 가미 가능
        s ← s'';
until(stopping condition)
```

▷ Tabu Search (TS)

$N(x)$: neighborhood of a solution
T : Tabu list
A : Aspiration function

```
x0 ← initial solution;
Initialize Tabu list T and aspiration function A;
i ← 1;
repeat
    pick the best x_i ∈ N(x_{i-1});
    if( x_i ∉ T)
    then accept x_i;
         update T and A;
    else if(cost( x_i) < A( x_{i-1}))
        then accept x_i;
             update T and A;
        else reject x_i;
    i++;
until(stopping condition)
```