<최종정리본>

-recursion theorem – 2 versions-

(1) if a computer program T written in Turing complete programming languages then, a computer program R exists.

t: a computable function ($\Sigma^* X \Sigma^* \rightarrow \Sigma^*$)

Σ^*: the set of all strings over Σ

R computes function value of t

<R>: all string code of R

r: $\Sigma^* \rightarrow \Sigma^*$

R: $\forall_{w \in \Sigma^*} r(w) = t(<R>, w)$

(2) abstract recursive theorem

if:

1. Σ is closed under composition
2. F(x, x) $\in$ Σ ($Z^+ \rightarrow Z^+$)
3. $\forall_{f \in \Sigma} \exists_{a \in Z^+} \forall_{x \in Z^+} E_{F(a,x)} = E_{f(x)}$

then:

$\forall_{f \in \Sigma} \exists_{i \in Z^+} E_i = E_{f(x)}$

proof)

$\forall_{f \in \Sigma} \exists_{a \in Z^+} \forall_{x \in Z^+} E_{F(a,x)} = E_{f(x)}$

the function f is arbitrary

-> $E_{F(a,x)} = E_{f(F(x,x))}$

-> $E_{F(a,a)} = E_{f(F(a,a))}$

-> $E_i = E_{f(i)}$

-a self-copying program-

The result of this code is exactly same as this code

```
program self copy
    L=ip-1
    loop until line[L]="end"
    {
        print(line[L])
        L=L+1
    }
    print("end")
end
```

본 노이만 컴퓨터는 5번째 라인 print(line[L])을 두 가지로 해석이 가능하다.
line[L]은 Data로, print는 Command로 해석가능하다.

-2 ways to represent a graphs-
(1) adjacency matrix
(2) adjacency list

-how to compute the transitive closure of a binary relation-
The transitive closure -> tc()
(1) a binary relation R:
 tc(R) = a binary relation R`
(2) a directed graph G:
 tc(G) = transitive closure of G
-> tc(R) and tc(G) are equivalent

R is a subset of AxA
with
1. R ⊆ R`,
2. R` is a transitive relation such that ∀(transitive relations on A that contains R)
R` ⊆ R``

smallest R`이 무엇인지 아는 것이 가장 중요하다.
1. R^0
2. $R^k[i,j] = \vee(R^{k-1}[i,j]),(\wedge R^{k-1}[i,k], R^{k-1}[k,j])$으로 구한다.

-a closed semiring-
(S, +, ・, 0, 1)
1. monoids-> (S, +, 0), (S, ・, 1)
 a. closed: a + b ∈ S (a ∈ S, b ∈ S)
 b. associative: (a + b) + c = a + (b + c)
 c. identity: a + 0 = 0 + a = a

 a. closed: a ・ b ∈ S (a ∈ S, b ∈ S)
 b. associative: (a ・ b) ・ c = a ・ (b ・ c)
 c. identity: a ・ 1 = 1 ・ a = a

2. + is commutative & idempotent
commutative: x + y = y + x
idempotent: a + a = a

3. distribution: a ・ (b + c) = a ・ b + a ・ c

4. countably infinite: $a_1 + a_2 + \cdots + a_i + \cdots$ exists and unique
-> associative, commutative, idempotent도 잘 적용된다.
5. •는 infinite한 sum에도 잘 distribute된다.
-> $\sum_{i}^{\infty} a_i \cdot \sum_{i}^{\infty} a_j = \sum_{i,j}^{\infty} a_i \cdot a_j = \sum_{i}^{\infty} (\sum_{j}^{\infty} a_i \cdot a_j)$

-Euler cycle problem-
input: an undirected graph
output:
yes, if the graph has Euler cycle such that every edges of G are used only once
no, otherwise

-Hamiltonian cycle problem-
input: an undirected problem
output:
yes, if the graph has Euler cycle such that every nodes of G are used only once
except only strating node
no, otherwise

-Traveling salesman problem-
input: a graph
output: a cycle such that
1. every node must be visited once
2. the sum of traveling cost should be minimized

new version
input: a graph, k
output: a cycle such that
1. every node must be visited once
2. the sum of traveling cost should be smaller than “k”

-Warshall's algorithm-
1. R^0
2. $R^k[i,j] = \vee (R^{k-1}[i,j]), (\wedge R^{k-1}[i,k], R^{k-1}[k,j])$으로 구한다.

-Dijkstra's algorithm-

input:

1. a directed weighted graph
2. a starting node

output:

all shortest paths from the starting node to all other nodes

```
S = {1}
for i = 2 to n do
    D[i] = C[1,i]       // initialization
for i= 1 to n-1
    choose a vertex w in V - S such that
        D[w] is a minimum
    add w to S
    for each vertex v in V - S
        D[v] = min(D[v], D[w]+C[w,v])
```

-Floyd's algorithm-

input: a directed weighted graph

output: all pairs shortest paths

for k = 1 to n

  for i = 1 to n

    for j = 1 to n

$$D[k][i] = \min(D[k][i], +(D[k][j], C[j,i]))$$

$C_{ij}^{k} \leftarrow C_{ij}^{k-1} + C_{ik}^{k-1} \cdot (\ )^{*} C \cdot C_{kj}^{k-1}$의 상위 형식이 있다. (괄호 "( )"은 자세히 알필요 없다.)

Floyd, Dijkstra, Warshall이 전부 위의 것으로 reduce 가능하다.

(S, +, •, 0, 1)

S = $R^{+} \cup \infty$

+ = min

• = +

0 = ∞

1 = 0

Efficient / inefficient algorithms

(1) the set of efficient algorithm

a. algorithm에 대하여

b. 그 알고리즘에 길이가 n인 input을 넣었을 때

c. 그 알고리즘은 finite한 step을 진행한다.
d. take the maximum value (number of steps)-> n^k
that algorithm is polynomial

(2) the set of computational problems solvable efficiently
if there is one efficient algorithm for the problem,
then that problem is “Problem solvable efficiently”

-K-clique problem (search / decision)-
(k≥2, positive integer)
k-clique is a complete subgraph G`=(V`, E`) of G such that satisfies two conditions
1. |V`|=k
2. every pair vertices are connected

<k-clique decision problem> (A)
input: an undirected graph, k
output:
YES, if there exist k-clique(s)
NO, otherwise

<k-clique search problem> (B)
input: an undirected graph, k
output:
any k-clique, if there exist k-clique(s)
empty set, otherwise

(A)와 (B)의 difficulty는 equivalent

3 cnf sat problem
input: a formula in 3 CNF
output:
YES, if f is satisfiable
NO, otherwise

3 CNF SAT reduces to HP

-P / NP / NPC-
NPC의 정의
1.
(a) a set of decision problems x is in NP
(b) All problems in NP are efficiently reducible to x

2.
If any problem in NPC is solvable efficiently
Then P=NP

P = {x|__________}
1. x is a decision problem
2. x is efficiently solvable

NP = {x|_________}
1. x is a decision problem
2. $\forall_{yes-\in stancea} \exists_{certificateb}$ Verification (a, b) is done efficiently

a ∈ NP
b ∈ P
c ∈ NPC

relative hardness
a ≥ b
a ≤ c
b ≤ c

a proof가 a validation of a proof보다 더 어려운 것이 증명되면, P≠NP이다.
a proof와 a validation of a proof이 동일한 난이도인 것이 증명되면, P=NP이다.

solving이 verifying보다 더 어려운 것이 증명되면, P≠NP이다.
solving과 verifying이 동일한 난이도인 것이 증명되면, P=NP이다.

solving이 효율적이다. ≡ 문제해결 algorithm의 step이 n^k이다. -> P
verifying이 효율적이다. ≡ verify algorithm의 step이 n^k이다. -> NP

a property P(x) = solving이 verifying보다 어렵다.
{x|x is a world in which P(x) is true}
{x|x is a world in which P(x) is not true}-> 이걸 찾으면 P≠NP이다.

-Kruskal's algorithm-
1. Kruskal`s Algorithm (G, w) (w is weight assinged to edges)
(1) Create n singletons (n = number of V)
-> {a}, {b}, {c}, {d}, {e}, {f}
(2) sort all edges in a non-decreasing order
-> (a, e), (c, f), (b, c), (a, d), (e, f), (b, e), (a, b)
(3)
for each edge (u, v)
if u, v belong to different regions, then (u, v) connected
(이러면 set의 갯수가 줄어든다.)

-Prim's algorithm-

1. Let T={S}
2. for i = 1 to n-1 do
   (1) select an edge with min weight from edges between T & V-T
   (2) update T

-Graph, tree, spanning tree-

an undirected graph G=(V, E)

V: a finite set of nodes(=vertices) $\neq \varnothing$

E: a finite set of unordered pairs of nodes in V

Definition) A forest:

an undirected graph

acyclic (There is no cycle)

Definition) A tree:

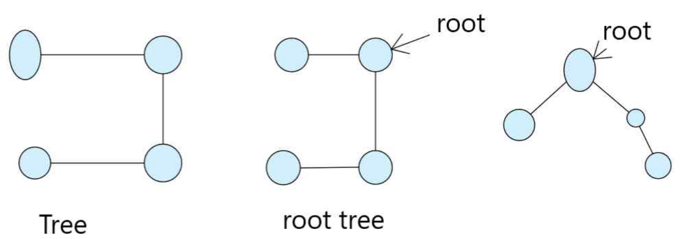
an undirected graph

acyclic

connected (any pair of node has a path)

Definition of a rooted tree)

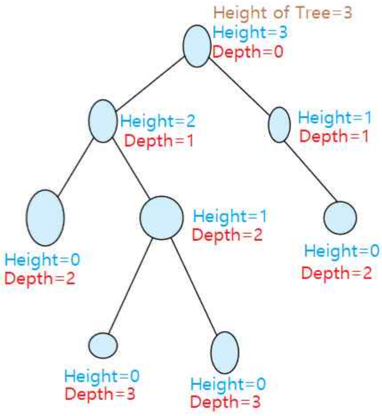
a tree whose root is designated



Definition of a binary tree)

a rooted tree where each node has at most 2 children

Given a binary tree T,
1. The height of a node -> all leaves height = 0
2. The depth fo a node -> The root is at depth 0.
3. the height of T -> the height of the root



Height는 더 큰 것을 선택한다.

-> uniquely defined

Definition) a spanning tree
Given an undirected, connected graph G = (V, E),
a spanning tree T = (V\`, E\`) of G is a subgraph of G such that
1. V\` = V
2. T is a tree

-A matching-
undirecte 그래프 G=(V,E) 가 주어질 때 이 그래프 안의 matching M은 E의 부분집합이면서 다음 조건을 만족하는 것입니다.
- V의 모든 원소 x에 대해
- M 안에 많아야 한 개 혹은 0개의 edge y가 존재하며
- y is incident on x

-A bipartite graph-
bipartite 그래프는 unidrected 그래프이면서 그 그래프의 노드들의 집합인 V를 정확하게 두 개의 집합들 A와 B로 나눌 수 있되 다음 조건을 만족시키는 것을 말합니다.
- A와 B 모두 공집합이 아니고
- A와 B의 교집합이 공집합이고
- A와 B 모두 V의 부분집합들이고
- 모든 edge 들이 A와 B 사이에만 존재

-max bipartite graph matching problem-
input: a bipartite graph G=(V, E)
output: “a” matching M in G such that the number of elements in M is maximum

-Max flow problem-
input: a flow network G
output: max|f|

MFP <- MBGMP (reducable)

-Euclidean algorithm-
if a and b are integers not both zero AND
if q and r are positive integers that satisfy a = b * q + r
then gcd(a,b) = gcd(b,r)
이 property로 유클리드 알고리즘을 만든다.

the significance if this property lies in that after a finite number of steps, it is guaranteed to stop with the gcd of two numbers and b.

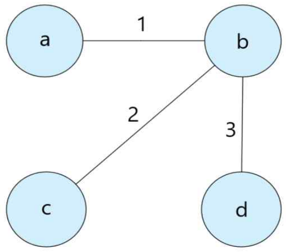
```
Euclid_Algorithm(input: A, B) //A > B >= 0
{
	a = A;
	b = B;
	r = B;

	while ( b is not equal to 0 )
	{
		r = (a mod b);
		a = b;
		b = r;
	}
	return a;
}
```

-pumping lemma-

$\forall_{L \in A} \exists_{m \in Z^+} \forall_{w \in L(|w| \geq m)} \exists_{w = xyz, |xy| \leq m, |y| \geq 1} \forall_{i \in \{0,1,2 \cdots\}} xy^i z \in L$

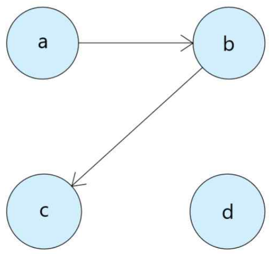
-adjacent & incident-



edge 1 is incident on(upon) a and b

edge 3 is incident on(upon) b and d

a and c are adjacent



b is adjacent to a

c is adjacent to b

Def) a cycle: a finite sequence of nodes such that

1. $x_1 = x_n$
2. $x_2, x_3, \cdots x_{n-1}$ are distinct

-> a finite sequence of sequence로도 정의 가능

-reducibility-

a->Hamiltonian cycle problem

b->Traveling salesman problem(new version)

==> If TSP is solvable, then HAM is solvable.

==> HAM reduces to TSP

TSP에서 모든 weight가 1이고, k가 모든 노드의 개수이면 된다.

A reduces to B

== if B is solvable, then A is solvable

== if A is not solvable, then B is not solvable

problem x, y are solvable,

problem z, w are not solvable

a. x reduces to y (O)

b. x reduces to z (O)

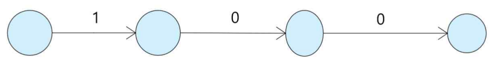
c. z reduces to w (O)

d. w reduces to y (X)

-Path-

Definition

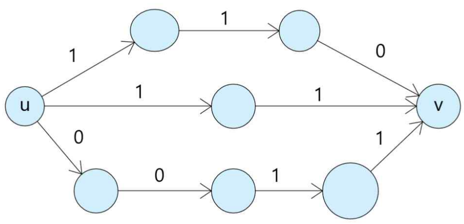
1. the label of an edge (label comes from S where (S, +, •, 0, 1))
2. the label of a path



->path is 1•0•0

3. the label of a path of length zero => 1 where (S, +, •, 0, 1)
4. c(u, v) for each pair of node (u, v)

=> the sum of all labels



c(u, v) = +(1•1•0, 1•1, 0•0•1•1)=1

no path->0

-Definition of CNF-

Conjunction Nomal Form (CNF)

1. a boolean variable
2. a literal
   (1) every boolean variable
   (2) negation of every boolean variable
3. a clause

: one or more literals combination with ∨

4. a formula

: one or more clauses combination with ∧

a formula f is called “satisfiable”, if an assignment exists which makes f be true
formula f is “unsatisfiable”, if none of assignment makes f true

partial order?

1. reflexive
2. anti-symmetric
3. transitive

a binary relation R

A = {1, 2, 3}

R = {(1, 2), (2, 1), (1, 1), (2, 2), (3, 3)} -> not partial order

R = {(1, 2), (1, 1), (2, 2), (3, 3)} -> partial order

anti-symmetric?

~(if aRb then bRa) ≡ if aRb then no bRa

-3-coloring problems-

input: an undirected graph

output:

YES, if G is 3-colorable

NO, otherwise

-Sudoku problem-

input: 완성되지 않은 $n^2 \times n^2$짜리 스토쿠 판

output:

YES, if

every row must have all elements,

every col must have all elements,

every box must have all elements

NO, otherwise

-Asymptohic notations-
O, Ω, Θ, o, w

Definition) Given a function f(n) (=runtime of an algorithm)

Definition)

O(f(n)) is the set of functions that grow slower than f(n),
OR grow at the same rate as f(n) as $n \to \infty$

Ω(f(n)): is the set of functions that grow faster than f(n),
OR grow at the same rate as f(n) as $n \to \infty$

Θ(f(n)): is the set of functions that grow at the same rate as f(n) as $n \to \infty$

o(f(n)): is the set of functions that grow slower than f(n) as $n \to \infty$

w(f(n)): is the set of functions that grow faster than f(n) as $n \to \infty$

ex) f(n)= n^2+n+1
(1) h(n) = 100n+10 $\in$ O(f(n))
(2) g(n) = $2n^2+5$ $\in$ O(f(n)) (coefficient does not matter)
$\notin$ o(f(n)) little Oh는 같은 것을 포함하지 않는다.
(3) I(n) = n^3+1 $\notin$ O(f(n))

-insertion sort (input array: A)-
-> n-1 comparision needed (Best case analysis)
-> $\frac{n(n-1)}{2}$ compare is needed
even with a same algorithm and same input size,
algorithm can work more or less.

-an equivalence relation-
Definition) an equivalence relation:
a binary relation R is called an equivalence relation
if
(1) R is reflexive
(2) R is symmetric
(3) R is transitive

- 어떻게 k clique 문제의 Search version이 Decision version으로 reduce 되는가-
to show this reduction, we start with a decider for the decision problem - let's call the decider, D

consider an example graph that consists of 4 nodes, a, b, c, d and 4 edges exist - (a,c), (c,d), (a,d), (b,d)

let's call this graph G and assume that we want to solve the Search version with input (G, 3)

all that we can use here is the decider D that can correctly answers "yes".

now, since there is a clique of size 3, we need to identify all 3 nodes which form a clique - how can we solve this? well, using the following steps, it is guaranteed that the nodes can be found.

- run D with input {a,b,c}, 3
- run D with input {a,c,d}, 3
- run D with input {a,b,d}. 3
- run D with input {b,c,d}, 3

$_4C_3 = 4$

because "there are 3 nodes that form a clique" and "D is assume to work correctly", it is clear that one of these must return yes - in this example, the second says "yes" and we have found an answer! of course, if we are given a different example, the same idea will be applied again and again.

D 가 이미 yes 라고 답을 했으므로 노드 4 개 중 어떤 3 개의 경우
D 는 분명 yes 를 출력합니다. 따라서 모든 가능성들을 체크해서
그 중 답이 yes 가 되게 하는 노드들이 clique 을 만들게 되고
이 아이디어를 이용하여 다른 예제들에 대해서도 설명할 수 있습니다.

-어떻게 3 cnf sat 문제가 Halting problem 으로 reduce 되는가-
정지문제를 해결하는 알고리즘이 HALT(P,x) 라고 한다면 우리는
다음과 같은 프로그램 A 를 만들어 A 와 A 의 입력을 HALT 에 넣어서
해결할 수 있습니다.

program A (f)

만일 f가 satisfiable 하면 정지, 그렇지 않으면 무한 루프

여기서 f가 satifiable 한지 아닌지는 어차피 유한개의 boolean 변수들에
모든 가능한 값들 대입해 보면 알 수 있으므로 유한 단계 안에 체크가
가능하고

A와 f를 HALT의 입력으로 넣어 나온 결과가 yes 즉 halt 한다면 f는
satisfiable, 그렇지 않다면 unsatifiable 하다고 결론 내릴 수 있습니다.

Definition of a flow network
: a flow network is a directed graph G=(V, E) with
1. There exist 2 special nodes in V “s”, “t”.
2. For each edge(<u, v>), a positive real number is assigned.
3. every node must be on a path from s to t

Definition of a flow
: a flow f: VxV -> R in a flow network G=(V, E) is a real-valued function subject to
1. flow conservation constraint
2. capacity constraint
3. skew symmetric constraint

-partition-
What is a partition?
Given a non empty set A, a partition of A is a union of finite number of non empty subsets of A, A_1, A_2, … , A_n such that A_1, A_2, … , A_n are pairwise disjoint

or

a partition of A is the union of infinite number of non empty subsets of A, A_1, A_2, … , A_n such that A_1, A_2, … , A_n are pairwise disjoint set A.