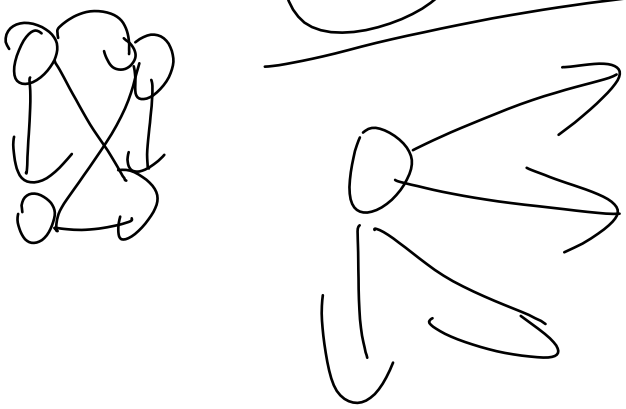
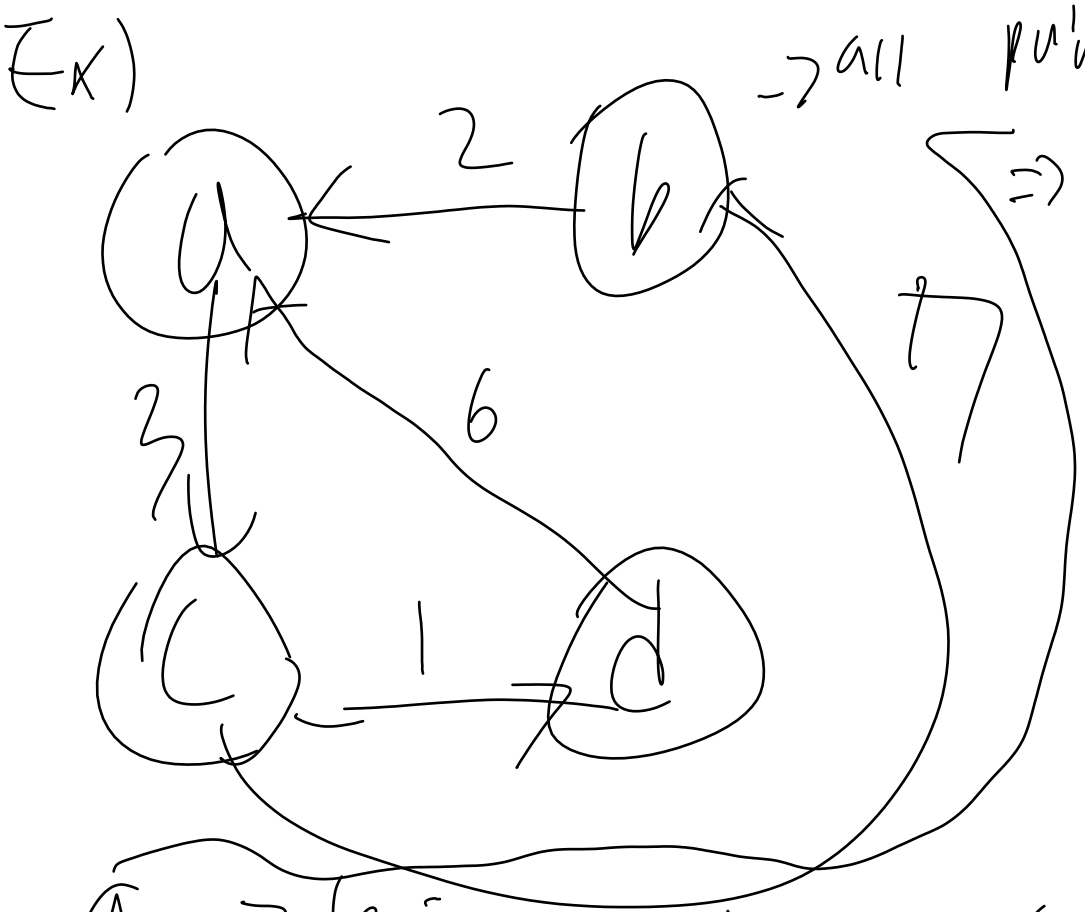
Floyd → all pairs shortest paths

- single source shortest path



Ex)



→ all pairs shortest paths?

⇒ distance matrix 2개 만들어야함

a ~> b : 10

→ c : 3

d : 4

b → a

b

d

d → a

b

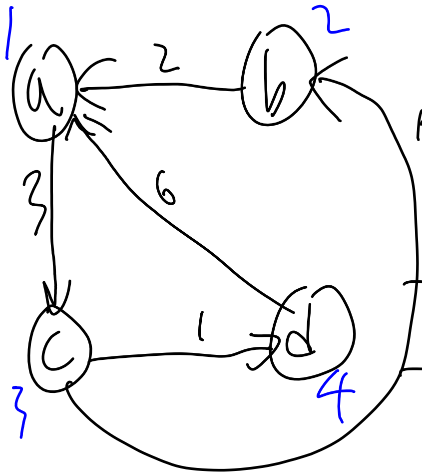
c

a closed semiring

$\Rightarrow (S, +, \cdot, 0, 1)$

Flayd: → all pairs shortest paths

→ single source shortest paths



4x4 matrix

	a	b	c	d → to
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0

↓ from

D

Distance matrix

warshall algorithm과 차이를 비교하라.

$D^0 \rightarrow D^1 \rightarrow D^2 \rightarrow \cdots \rightarrow D^n$

$R^k[i,j] = or(R^{k-1}[i,j], and(R^{k-1}[i,k], R^{k-1}[k,j]))$

Ex 5.5) $(\{0,1\}, \vee, \wedge, 0, 1) \leftarrow (S, +, \cdot, 0, 1)$

Boolean semiring

$D^0 \rightarrow D^1 \rightarrow \cdots D^k \rightarrow \cdots \rightarrow D^n$

$D^k[i,j] = min(D^{k-1}[i,j], +(D^{k-1}[i,k], D^{k-1}[k,j]))$

for k=1 to n do
  for i=1 to n do
    for j=1 to n do

Dijkstra(G)
1. S={1}
2. for i=2 to n do
3.   D[i] = C[1,i]
4. for i=1 to n do
5.   select w in V-S such that D(w) is min
6.   add w to S

$(S, +, \cdot, 0, 1)$

$(R^+ \cup \{0\}, min, +, +\infty, 0)$

positive real number

Dijkstra 다익스트라의 Algorithm

for each node v in V-S (V: all nodes, S: start nodes)

$D(v) = min(D(v), +(w(u,v), D(w)))$ (w: v 바로 이전)

- select
- update

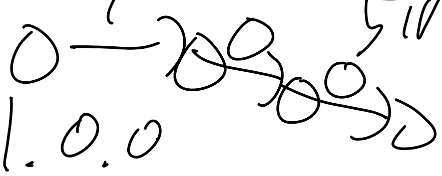
Ex 5.11)

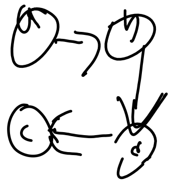
Definitions (s, t, ·, 0, 1)

모든 edge는 'S'로 부터 label이 붙는다.

① the label of an edge ← Define first

② the label of a path





③ the label of a path of length zero

→ 1로 정의됨 (s, t, ·, 0, 1)

~~Ex 5.10~~ ~~d~~ → Don't deal with this

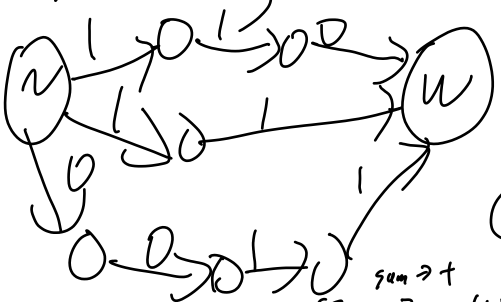
∴ all of labels come from (S, t, ·, 0, 1) S

path: a sequence of edges

④ $c(v, w)$ for each pair of nodes v, w

$c(v, w) = ?$ → the sum of all labels of paths.

ex)



3 path

$c[v, w]$?

sum → +

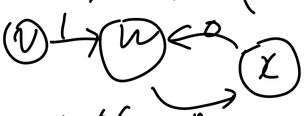
$c[v, w] = +(\underbrace{1 \cdot 1 \cdot 0}_{0}, \underbrace{1 \cdot 1}_{1}, \underbrace{0 \cdot 0 \cdot 1 \cdot 1}_{0}) = 1$

cost from v to w.

ex)

→ no path? C[a,b]=0 (by Definition)

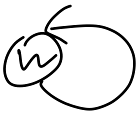
5.11)



the label of a path

$\overrightarrow{v\ w\ x} = 1 \cdot T$

a cycle of a positive length



?

the label of the path w to w ? 1

$C^{k}_{ij} \leftarrow C^{k-1}_{ij} + C^{k-1}_{ik} \cdot [\times]^{*} \cdot C^{k-1}_{kj}$

can be cast 형식

$R^{k}[i,j]$

$D[i,j]$

$D(i)$

algorithm has structural view

W □ → reachability

Warshall (∪ ∧)

Floyd (min +) view — an operation performs

Dijkstra (min +)

$+ \ \cdot$

chapter 11) P, NP

P ; polinomial time

NP ; Nondeterministic polinomial time

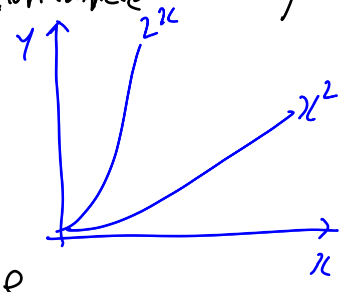
NPC ; NP + complete

) ⇒ sets → classes of problems

two functions

polinomial ; $y = x^2$ → slower increase

exponential ; $y = 2^x$ → faster increase



rule

1. worst-case performance
2. a best Alg

→ check

↓

Criteria of algorithm

a problem → soluble

soluble problem 에만 집중 (일단 그풀 가지 가정)

하나의 problem → alg 1 ; 10초

→ Alg 2 ; 1시간

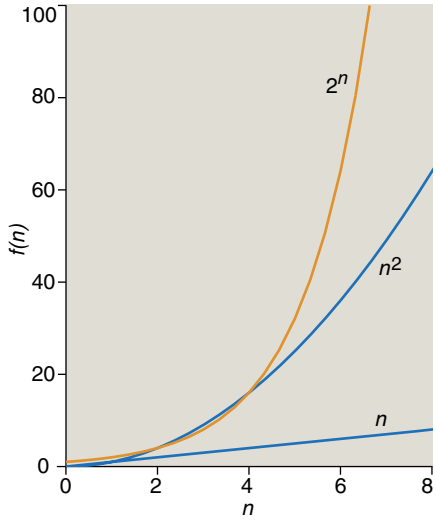
→ Alg 3

효율이 별개?

class P ; → P = {x | x has at least one polinomial algorithm}

→ efficient

⇒



Polynomial and exponential functions define the poles of computational efficiency. The running time of an algorithm is measured as a function of the size of the input, *n*. If the function is a polynomial one, such as *n* or n^2, the algorithm is considered efficient; an exponential growth rate, such as 2^n, makes the algorithm impractically slow.

polynomial-time program is faster than any exponential-time program.

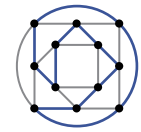
So much for the classification of algorithms. What about classifying the problems that the algorithms are supposed to solve? For any given problem, there might be many different algorithms, some faster than others. The custom is to rate a problem according to the worst-case performance of the best algorithm. The class known as P includes all problems that have at least one polynomial-time algorithm. The algorithm has to give the right answer and has to run in polynomial time on every instance of the problem.

Classifying problems for which we *don't* know a polynomial-time algorithm is where it gets tricky. In the first place, there are some problems that require exponential running time for reasons that aren't very interesting. Think about a program to generate all subsets of a set of *n* items; the computation is easy, but because there are 2^n subsets, just writing down the answer will take an exponential amount of time. To avoid such issues, complexity theory focuses on problems with short answers. Decision problems ask a yes-or-no question ("Does the graph have a Hamiltonian circuit?"). There are also counting problems ("How many Hamiltonian circuits does the graph have?"). Problems of these kinds might conceivably have a polynomial-time solution, and we know that some of them do. The big question is whether *all* of them do. If not, what distinguishes the easy problems from the hard ones?

Conscientious Cheating

The letters NP might well be translated "notorious problems," but the abbreviation actually stands for "nondeterministic polynomial." The term refers to a hypothetical computing machine that can solve problems through systematic guesswork. For the problems in NP, you may or may not be able to compute an answer in polynomial time, but if you happen to guess the answer, or if someone whispers it in your ear, then you can quickly verify its correctness. NP is the complexity class for conscientious cheaters—students who don't do their own homework but who at least check their cribbed answers before they turn them in.

Detecting a Hamiltonian circuit is one example of a problem in NP. Even though I don't know how to solve the problem efficiently for all graphs, if you show me a purported Hamiltonian circuit, I can readily check whether it passes through every vertex once:

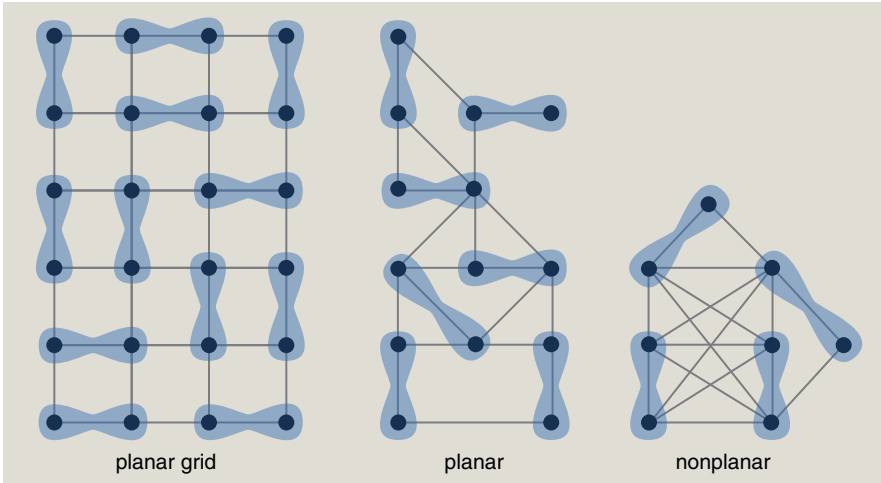


(Note that this verification scheme works only when the answer to the decision problem is "yes." If you claim that a graph *doesn't* have a Hamiltonian circuit, the only way to prove it is to enumerate all possible paths.)

Within the class NP dwells the elite group of problems labeled NP-complete. They have an extraordinary property: If any one of these problems has a polynomial-time solution, then that method can be adapted to quickly solve *all* problems in NP (both the complete ones and the rest). In other words, such an algorithm would establish that P = NP. The two categories would merge.

The very concept of NP-completeness has a whiff of the miraculous about it. How can you possibly be sure that a solution to one problem will work for every other problem in NP as well? After all, you can't even know in advance what all those problems are. The answer is so curious and improbable that it's worth a brief digression.

The first proof of NP-completeness, published in 1971 by Stephen A. Cook of the University of Toronto, concerns a problem called satisfiability. You are given a formula in Boolean logic, constructed from a set of variables, each of which can take on the values *true* or *false*, and the logical connectives AND, OR and NOT. The decision problem asks: Is there a way of assigning *true* and *false* values to the variables that makes the entire formula *true*? With *n* variables there are 2^n possible assignments, so the brute-force approach is exponential and unappealing. But a



The perfect-matching problem pairs up the vertices of a mathematical graph. The number of possible matchings grows exponentially with the size of the graph; nevertheless, the matchings can be counted in polynomial time on a planar graph (one without crossed edges). The problem was first studied on graphs with a periodic structure, such as the rectilinear grid at left, but the algorithm also works on less-regular planar graphs, such as the one in the middle. The graph at right is nonplanar, and its perfect matchings cannot be counted quickly.

lucky guess is easily verified, so the problem qualifies as a member of NP.

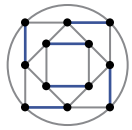
Cook's proof of NP-completeness is beautiful in its conception and a shambling Rube Goldberg contraption in its details. The key insight is that Boolean formulas can describe the circuits and operations of a computer. Cook showed how to write an intricate formula that encodes the entire operation of a computer executing a program to guess a solution and check its correctness. If and only if the Boolean formula has a satisfying assignment, the simulated computer program succeeds. Thus if you could determine in polynomial time whether or not any Boolean formula is satisfiable, you could also solve the encoded decision problem. The proof doesn't depend on the details of that problem, only on the fact that it has a polynomial-time checking procedure.

Thousands of problems are now known to be NP-complete. They form a vast fabric of interdependent computations. Either all of them are hard, or everything in NP is easy.

The Match Game

To understand the new holographic algorithms, we need one more ingredient from graph theory: the idea of a perfect matching.

Consider the double-feature festival. You want to show movies in pairs, with the proviso that any two films scheduled together should have a performer in common; also, no film can be screened more than once. These constraints lead to a graph where the vertices are film titles, and two titles are connected by an edge if the films share an actor. The task is to identify a set of edges linking each vertex to exactly one other vertex. The brute-force method of trying all possible matchings is exponential, but if you are given a candidate solution, you can efficiently verify its correctness:



Thus the perfect-matching problem lies in NP.

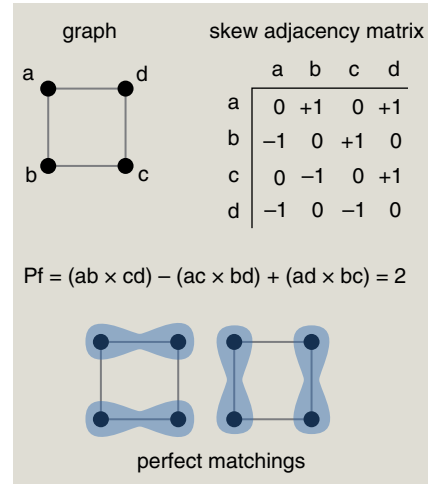
In the 1960s Jack Edmonds, now of the University of Waterloo, devised an efficient algorithm that finds a perfect matching if there is one. The Edmonds algorithm works in polynomial time, which means the decision problem for perfect matching is in P. (Indeed, Edmonds's 1965 paper includes the first published discussion of the distinction between polynomial and exponential algorithms.)

Another success story among matching methods applies only to planar graphs—those that can be drawn without crossed edges. On a planar graph you can efficiently solve not only the decision problem for perfect matching but also the counting problem—that is, you can learn how many different subsets of edges yield a perfect matching. In general, counting problems seem more difficult than decision problems, since the solution conveys more information. The main complexity class for counting problems is called #P (pronounced "sharp P"); it includes NP as a subset, so #P problems must be at least as hard as NP.

The problem of counting planar perfect matchings has its roots in physics and chemistry, where the original question was: If diatomic molecules are adsorbed on a surface, forming a single layer, how many ways can they be arranged? Another version asks how many ways dominos (2-by-1 rectangles) can be placed on a chessboard without gaps or overlaps. The answers exhibit clear signs of exponential growth; when you arrange dominos on square boards of size 2, 4, 6 and 8, the number of distinct tilings is 2, 36, 6,728 and 12,988,816. Given this rapid proliferation, it seems quite remarkable that a polynomial-time algorithm can count the configurations. The ingenious method was developed in the early 1960s by Pieter W. Kasteleyn and, independently, Michael E. Fisher and H. N. V. Temperley. It has come to be known as the FKT algorithm.

The mathematics behind the FKT algorithm takes some explaining. In outline, the idea is to encode the structure of an n-vertex graph in an n-by-n matrix; then the number of perfect matchings is given by an easily computed property of the matrix. The illustration on this page shows how the graph is represented in matrix form.

The computation performed on the matrix is essentially the evaluation of a determinant. By definition, a determinant is a sum of $n!$ terms, where each term is a product of n elements chosen from the matrix. The symbol $n!$ denotes the factorial of n, or in other words $n \times (n-1) \times \ldots \times 3 \times 2 \times 1$. The trouble is,



The fast algorithm for counting planar perfect matchings works by translating the problem into the language of matrices and linear algebra. The pattern of connections within the graph is encoded in an adjacency matrix—an array of numbers with rows and columns labeled by the vertices of the graph. If two vertices are joined by an edge, the corresponding element of the matrix is either +1 or –1; otherwise the element is 0. Elements of the matrix are combined in a sum of products called the Pfaffian, which yields the number of perfect matchings. For the simple graph shown here there are two perfect matchings.

$n!$ is not a polynomial function of n; it qualifies as an exponential. Thus, under the rules of complexity theory, the whole scheme is really no better than the brute-force enumeration of all perfect matchings. But this is where the rabbit comes out of the hat. There are alternative algorithms for computing determinants that *do* achieve polynomial performance; the best-known example is the technique called Gaussian elimination. With these methods, all but a polynomial number of terms in that giant summation magically cancel out. We never have to compute them, or even look at them.

(The answer sought in the perfect-matching problem is actually not the determinant but a related quantity called the Pfaffian. However, the Pfaffian is equal to the square root of the determinant, and so the computational procedure is essentially the same.)

The existence of a shortcut for evaluating determinants and Pfaffians is like a loophole in the tax code—a windfall for those who can take advantage of it, but you can only get away with such special privileges if you meet very stringent conditions.

Closely related to the determinant is another quantity associated with