Contents

1. Lecture 2

Some problems are intrinsically harder than others. Today we'll learn to classify them, and how to prove in which category they belong.

The **effort** of an algorithm is measured by the number of elementary operations it requires when applied to a given instance.
An elementary operation is a piece of computation expressible in a programming language which takes some **constant amount of time.**

Definition
If there is a constant C and a number k such that

$$\forall n \geq k, \quad f(n) \leq C \cdot g(n)$$

We say that a function f is of order of a function g and write:

$$f(n) = O(g(n))$$

The statement $f(n) = O(g(n))$ means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$.

Conversely, we introduce the notation $g(n) = \Omega(f(n))$ when $f(n) = O(g(n))$.

Moreover if $f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$ we write $f(n) = \Theta(g(n))$.

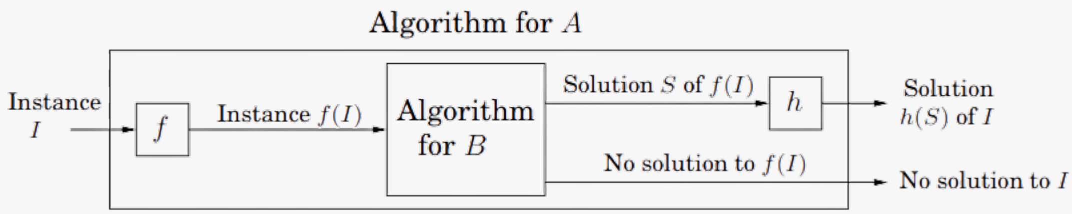
The most important question in combinatorial optimization is: Given a problem, can we find a solution with the desired property in polynomial time?

Definition
A combinatorial problem is a **search problem** if a yes-answer can verified in polynomial time.

More precisely, there is an efficient checking algorithm C that takes as input the given instance I and as well as the proposed solution S, and outputs true if and only if S really is a solution to instance I.Moreover the running time of $C(I,S)$ is bounded by a polynomial in the length of the instance.

NP is the class of all search problems where yes- answer can be **verified** in polynomial time.
The class of all search problems that can be **solved** in polynomial time is denoted by **P**.

Definition A **reduction** from search problem A to search problem B is a polynomial-time algorithm f that transforms any instance I of A into an instance $f(I)$ of B, together with another polynomial-time algorithm h that maps any solution S of $f(I)$ back into a solution$h(S)$ of I.



A reduction from A to B is denoted by A $\rightarrow$ B.

A search problem is **NP-complete** if all other search problems reduce to it.

If a problem B can be used to efficiently solve problem A then we say the Problem B is **at least** as hard as A.

NP-complete are those problems in NP that are at least as hard as any other problem in NP.

Example:

Problem A: Find the smallest element of $[2,8,-1,33,0,3]$
Problem B: Sort $[2,8,-1,33,0,3]$

We can use B to solve A by returning the first element of the sorted array therefore:

A $\rightarrow$ B and Difficulty(B) >= Difficulty(A)

When we call only one time the algorithm, the reduction is also called a transformation.

We can use A to solve B by runing the 'smallest' algorithm n times. A factor n is polynomial is polynomial therefore it's still a reduction:

B $\rightarrow$ A

An another example is the exercise 2 from the last week:

Max maching in a bipartite graph $\rightarrow$ Max flow

But the opposite is not possible. Assume we have an efficient algorithm to find the maximal match of a bipartite graph.That doesn't mean that we can solve **all** flow problems now because some networks are different.

Remark: It is not because we know how to solve a problem that it is simple, we talk about complexity. Maybe by transforming the problem we don't know how to solve into a problem we do know how to solve, we're doing too much work. So if we know how to solve A and we want to solve B using A then:

- B->A
- Difficulty(B) $\leq$Difficulty(A): the difficulty increases in the direction of the arrow
- We say A is at least as hard as B.

Since difficulty flows in the direction of the arrow, if we know A is hard, we can use the reduction to prove that B is hard as well.
If $A \rightarrow B$ and A is NP-complete then B is NP-complete as well

To put it in a nutshell:

The class NP	A problem is in NP whenever a yes answer can be **verified** in polynomial time.This is the class of search problems.
The class P	A problem is in the class P when it can be **decided and verified** in polynomial time.
NP-Complete	A problem is NP-Complete when in polynomial time it can be **verified** and any other NP problem can be **reduced**.

Verified ? Yes by a deterministic algorithm on a regular computer.

Turn out that being an NP-complete problem is a very strong requirement indeed. Is it possible to be NP-complete ? Maybe the set of NP-complete is empty ?

Nevertheless, they do exists and the first problem that was proven to be NP-complete (by Cook and Levin) is the Satisfiability problem (SAT).

A problem is called **NP-hard** if all search problems can be reduced to it. The subtle difference with the definition of NP-complete is that we do not require that the NP-hard problem itself is a search problem.

1.1 Tutorial

Exercise 1

$f(n)$	$g(n)$
$n-100$	$n-200$
$n^{\frac{1}{2}}$	$n^{\frac{3}{2}}$
$100n+\log(n)$	$n+\log(n)^2$
$n\log(n)$	$10\log(10n)$
$\log(2n)$	$\log(3n)$
$10\log(n)$	$\log(n^2)$
$n^{1.01}$	$n\log(n)^2$
$\frac{n^2}{\log n}$	$n\log(n)^2$
$n^{0.1}$	$\log(n)^{10}$
$\log(n)^{\log(n)}$	$\frac{n}{\log n}$
$\sqrt{n}$	$\log(n)^3$
$n^{\frac{1}{2}}$	$5^{\log_2(n)}$
$n2^n$	3^n
2^n	2^{n+1}
$n!$	2^n

Exercise 2 The Linear Assignment Problem (LAP) is defined as follows: given a square matrix c of numbers c_{ij} assign each row i once to a unique column j such that the sum of the corresponding c_{ij} is minimal. For example, in the matrix below, selecting the red numbers would be a feasible solution with objective value 4.

$$\begin{bmatrix} 3 & 4 & 1 \\ 4 & 0 & 0 \\ 5 & -1 & 9 \end{bmatrix}$$

a) Transform the LAP instance above to an instance of the min cost flow problem that can be used to find the answer to the original problem.

It's given that a polynomial time algorithm exists for the min cost flow problem.
b) For each of the following statements, write true or false:

- LAP $\rightarrow$ Min cost
- min cost $\rightarrow$ LAP
- A polynomial-time algorithm exists for LAP
- LAP is at least as hard as min cost flow

Exercise 3 Search versus decision.
Suppose you have a procedure which runs in polynomial time and tells you whether or not a graph has a Rudrata path (a path that visits each vertex exactly once). Show that you can use it to develop a polynomial- time algorithm for RUDRATA PATH (which returns the actual path, if it exists).

Exercise 4 Optimization versus search.

TSP

Input: A matrix of distances; a budget b
Output: A tour which passes through all the cities and has length $\leq b$, if such a tour exists.

The optimization version of this problem asks directly for the shortest tour.

TSP-OPT

Input: A matrix of distances
Output: The shortest tour which passes through all the cities.

Show that if TSP can be solved in polynomial time, then so can TSP-OPT.

Exercise 5 For each of the problems below, prove that it is NP-complete by showing that it is a generalization of some NP-complete problems we have seen in this chapter.

- **SUBGRAPH ISOMORPHISM**: Given as input two undirected graphs G and H, determine whether G is a subgraph of H (that is, whether by deleting certain vertices and edges of H we obtain a graph that is, up to renaming of vertices, identical to G), and if so, return the corresponding mapping of $V(G)$ into $V(H)$.
- **LONGEST PATH**: Given a graph G and an integer g, find in G a simple path of length g.
- **MAX SAT**: Given a CNF formula and an integer g, find a truth assignment that satisfies at least g clauses.
- **DENSE SUBGRAPH**: Given a graph and two integers a and b, find a set of a vertices of G such that there are at least b edges between them.
- **SPARSE SUBGRAPH**: Given a graph and two integers a and b, find a set of a vertices of G such that there are at most b edges between them.

Hint: The following problems provide the answer in some order

- The decision problem **Rudrata path** asks if a graph contains a path that visits each node exactly once. The path does not have to return to its starting point.
- The decision problem **Rudrata cycle** asks if a graph contains a cycle that visits each node exactly once.
- The decision problem **independent set** asks if a graph contains a subset S V of certain size, such that none of the edges between the nodes in S exist.
- The decision problem **clique** asks if a graph contains a subset $S \subset V$ of certain size, such that all possible edges between the nodes in S exist.

Exercise 6 The **k-SPANNING TREE** problem is the following:
Input: An undirected graph $G = (V, E)$
Output: A spanning tree of G in which each node has degree $\leq k$, if such a tree exists.
Show that for any $k \geq 2$:

(a) k-SPANNING TREE is a search problem.
(b) k-SPANNING TREE is NP-complete. (Hint: Start with k = 2 and consider the relation between this problem and RUDRATA PATH.)

Exercise 7 Determine which of the following problems are NP-complete and which are solvable in polynomial time. In each problem you are given an undirected graph G = (V, E), along with:

(a) A set of nodes $L \subseteq V$, and you must find a spanning tree such that its set of leaves includes the set L.
(b) A set of nodes $L \subseteq V$, and you must find a spanning tree such that its set of leaves is precisely the set L.
(c) A set of nodes LV , and you must find a spanning tree such that its set of leaves is included in the set L.
(d) An integer k, and you must find a spanning tree with k or fewer leaves.

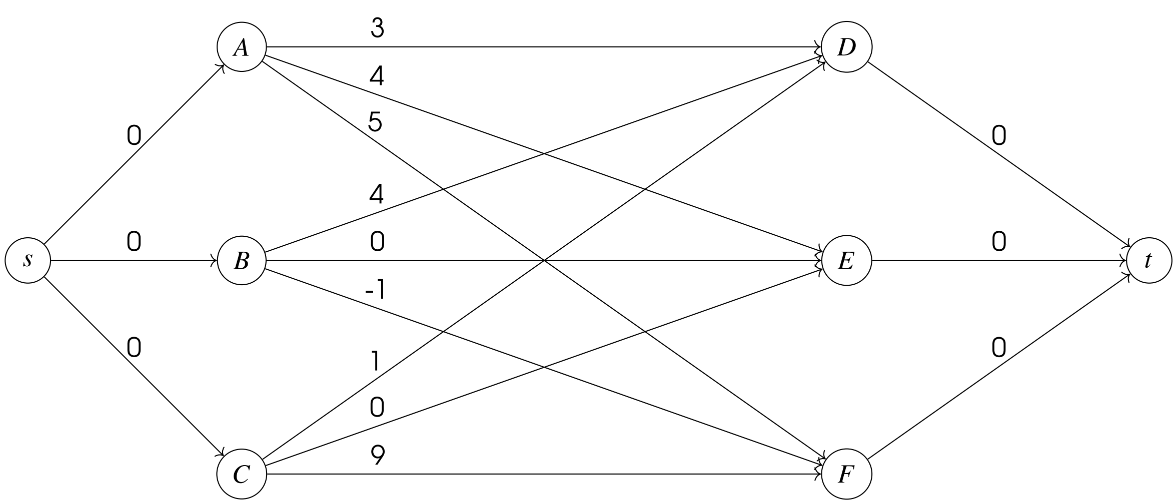
Exercise 8

Solution 1

$f(n)$	$g(n)$	$f(n) = O(g(n))$	$f(n) = \Omega(g(n))$	f(n)=$\Theta(g(n))$
$n-100$	$n-200$	✓	✓	✓
$n^{\frac{1}{2}}$	$n^{\frac{3}{2}}$	✓	✗	✗
$100n+\log(n)$	$n+\log(n)^2$	✓	✓	✓
$n\log(n)$	$10\log(10n)$	✓	✓	✓
$\log(2n)$	$\log(3n)$	✓	✓	✓
$10\log(n)$	$\log(n^2)$	✓	✓	✓
$n^{1.01}$	$n\log(n)^2$	✗	✓	✗
$\frac{n^2}{\log n}$	$n\log(n)^2$	✗	✓	✗
$n^{0.1}$	$\log(n)^{10}$	✗	✓	✗
$\log(n)^{\log(n)}$	$\frac{n}{\log n}$	✗	✓	✗
$\sqrt{n}$	$\log(n)^3$	✗	✓	✗
$n^{\frac{1}{2}}$	$5^{\log_2(n)}$	✓	✗	✗
$n2^n$	3^n	✓	✗	✗
2^n	2^{n+1}	✓	✓	✓
$n!$	2^n	✗	✓	✗

Solution 2

a) The problem requires a flow of 3 to be sent from s to t.



b)

- LAP → Min cost: true
- min cost → LAP: false not every min cost problem can be represent by a square matrix
- A polynomial-time algorithm exists for LAP: true
- LAP is at least as hard as min cost flow: false, it's the opposite

Solution 3

1) Remove an arc + run the algorithm: if there is no more Rudrata path, put back the arc an never remove it again

2) Repeat step 1 until we have tried all arcs

Solution 4 Run the algorithm that solves TSP for a budget b several times, for a budget $n = 0, 1, 2, 3, \ldots, b$

However, you need to consider the following:
If the magnitude of a number matters, we use binary representation. That is, for a number x we define the input size as $log_2(x)$. Polynomial may depend on input size and on $log_2(x)$, but not on x itself! Therefore, an algorithm that takes I steps where I is the optimal length of the TSP, is not considered polynomial.

The efficient (polynomial time) approach is to apply binary search. Let L be a lower bound on the optimal value and U an upper bound.
Initially, take $L = 0$ and let U be an upper bound on the optimal value, in this case we could take $U = \sum_{i,j} d(i,j)$ i.e., the sum of all distances. In each iteration we check if there is a valid solution, i.e, a tour of length at most b and then adjust L or U. We maintain the invariant that there is valid solution of value at most U and there is no solution of value $L-1$.

The number of times that we check for a valid solution of value at most b is only $O(\log(= \sum_{i,j} d(i,j)))$ which is polynomial in the input size

Solution 5
- clique $\rightarrow$ SUBGRAPH ISOMORPHISM
- Rudrata path $\rightarrow$ longest path
- SAT $\rightarrow$ MAX SAT
- clique $\rightarrow$ dense subgraph
- independant set $\rightarrow$ sparse subgraph

Solution 6
(a) Given a graph we should verify that it is a spanning tree and all degrees are at most k. To verify that's a spanning tree it is enough to verify that it has $n-1$ edges and that it connects all vertices. The latter can be done by breath- or depth-first search. To verify all degrees are at most k we only need to check for each vertex that it has at most k neighbors. Clearly, this can be done in polynomial time.
(b) The k-SPANNING TREE is a generalization of RUDRATA PATH. Assume we want to find a RUDRATA PATH. Let $k = 2$. There is spanning tree in which each node has a degree of at most 2 if and only of there is a Rudrata path.

Solution 7 Assume that there exists a spanning tree T such that its set of leaves includes L. If we delete the vertices L from the tree then it remains connected. Hence, if we delete L and all its incident edges from G then the graph remains connected. Now the following algorithm finds a spanning tree T such that its set of leaves includes L if such a tree exists: - Delete L and all its incident edges from G. Let this be G'. - Find a spanning tree T' in G'. - For each vertex v in L, add an edge that connects v to T'.

Solution 8