Algorithms and Complexity (AC)

Third Lecture (25th of September, 2023)

Paul Bouman & Tom van der Zanden (Based on slides by Gerhard Woeginger, Jesper Nederlof and Marie Schmidt)

September – November 2023

LNMB – Landelijk Netwerk Mathematische Besliskunde

Table of contents

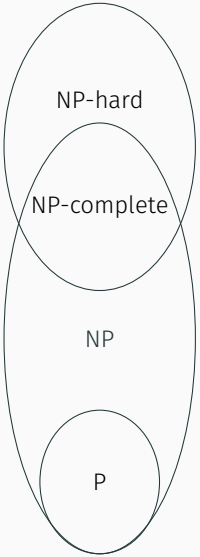
Our program for week 3

- Questions so far or about the assignment?
- pseudo-polynomial time, strong NP-hardness & weak NP-hardness
- co-NP, co-NP versus NP
- Beyond P, NP and coNP
- Exercises 1: questions?, partner-finding?

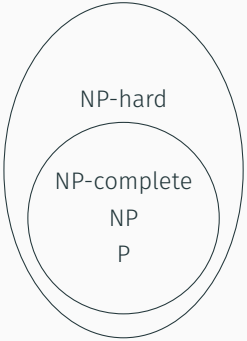
More NP-hardness

Complexity Classes

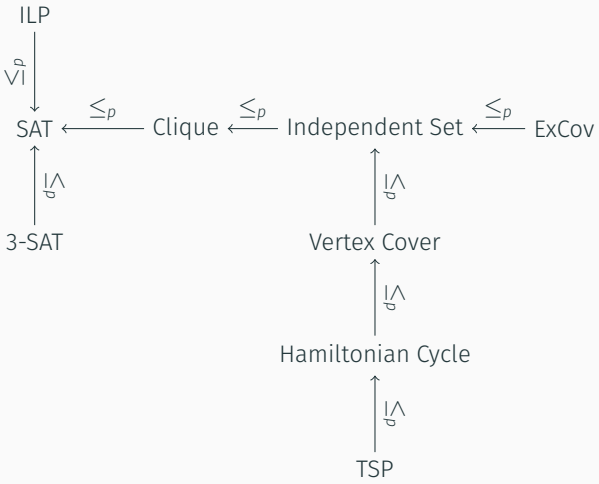
If $P \neq NP$



If $P = NP$



Reductions


ILP
$\leq_p$
SAT
$\leq_p$
Clique
$\leq_p$
Independent Set
$\leq_p$
ExCov
$\geq_p$
3-SAT
$\geq_p$
Vertex Cover
$\geq_p$
Hamiltonian Cycle
$\geq_p$
TSP

Subset Sum (SS)

Instance: positive integers $a_1, \ldots, a_n$; a bound b

Question: does there exist an index set $J \subseteq \{1, \ldots, n\}$ with $\sum_{j \in J} a_j = b$?

Example: $(a_1, \ldots, a_{12}) = (1, \ldots, 12), b = 50$. Yes or no instance?

Yes: $1 + 2 + 3 + 4 + 6 + 7 + 8 + 9 + 10 = 50$.

We argued last week that this problem is in NP.

Subset Sum

Subset Sum (SS)

Instance: positive integers $a_1, \ldots, a_n$; a bound b

Question: does there exist an index set $J \subseteq \{1, \ldots, n\}$ with $\sum_{j \in J} a_j = b$?

Example: $(a_1, \ldots, a_{12}) = (1, \ldots, 12), b = 50$. Yes or no instance?

Yes: $1 + 2 + 3 + 4 + 6 + 7 + 8 + 9 + 10 = 50$.

We argued last week that this problem is in NP. What about NP-completeness/NP-hardness?

Subset Sum (SS)

Instance: positive integers $a_1, \ldots, a_n$; a bound b

Question: does there exist an index set $J \subseteq \{1, \ldots, n\}$ with $\sum_{j \in J} a_j = b$?

Let us try a reduction from Exact Cover.

Exact cover (Ex-Cov)

Instance: a ground set X; subsets $S_1, \ldots, S_m$ of X

Question: do there exist some subsets S_i that form a partition of X?

Subset Sum

First idea: encode the sets at binary strings and interpret these as numbers

Set	Binary Encoding				Decimal Value
$\{1, 3\}$	0	1	0	1	5
$\{2, 4\}$	1	0	1	0	10
$\{2, 3, 4\}$	1	1	1	0	14
$\{4\}$	1	0	0	0	8

Any objections?

Subset Sum

First idea: encode the sets at binary strings and interpret these as numbers

Set	Binary Encoding	Decimal Value
$\{1,3\}$	0 1 0 1	5
$\{2,4\}$	1 0 1 0	10
$\{2,3,4\}$	1 1 1 0	14
$\{4\}$	1 0 0 0	8

Any objections?

Carry-over is a problem:

$$\begin{array}{cccc} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ \hline 1 & 1 & 1 & 0 \end{array} +$$

Subset Sum

First idea: encode the sets at binary strings and interpret these as numbers

Set	Binary Encoding				Decimal Value
$\{1,3\}$	0	1	0	1	5
$\{2,4\}$	1	0	1	0	10
$\{2,3,4\}$	1	1	1	0	14
$\{4\}$	1	0	0	0	8

Any objections?

Carry-over is a problem:

$$\begin{array}{cccc} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ \hline 1 & 1 & 1 & 0 \end{array} +$$

Any suggestion how to resolve this?

Second idea: create enough space in the encoding so carry-over can not reach the next element

Set	Binary Encoding				Decimal Value
$\{1, 3\}$	000	001	000	001	65
$\{2, 4\}$	001	000	001	000	520
$\{2, 3, 4\}$	001	001	001	000	584
$\{4\}$	001	000	000	000	512

Now the elements can not interfere with each other:

$$\begin{array}{cccc} 000 & 001 & 000 & 001 \\ 001 & 000 & 000 & 001 \quad + \\ \hline 001 & 001 & 000 & 010 \end{array}$$

Subset Sum

Transformation

Let $(X = \{x_1, \ldots, x_m\}, \{S_1, \ldots, S_n\})$ be an instance of Ex-Cov.

Subset Sum

Transformation

Let $(X = \{x_1, \ldots, x_m\}, \{S_1, \ldots, S_n\})$ be an instance of Ex-Cov.

We use $(n+1)$-*ary* encoding to avoid carry over.

Transformation

Let $(X = \{x_1, \ldots, x_m\}, \{S_1, \ldots, S_n\})$ be an instance of Ex-Cov.
We use $(n+1)$-*ary* encoding to avoid carry over.

Define numbers a_j as $a_j := \sum_{i=1}^{m} c_{ij} \cdot d_i$ with $c_{ij} = 1$ if $x_i \in S_j$ and $d_i = (n+1)^{i-1}$.
Set $b := \sum_{i=1}^{m} (n+1)^{i-1}$.

Transformation

Let $(X = \{x_1, \ldots, x_m\}, \{S_1, \ldots, S_n\})$ be an instance of Ex-Cov.
We use $(n+1)$-*ary* encoding to avoid carry over.

Define numbers a_j as $a_j := \sum_{i=1}^{m} c_{ij} \cdot d_i$ with $c_{ij} = 1$ if $x_i \in S_j$ and $d_i = (n+1)^{i-1}$.
Set $b := \sum_{i=1}^{m} (n+1)^{i-1}$.

(Ex-Cov) $\Rightarrow$ (SS): since an exact cover contains each element exactly once, no carry over can occur and the sum of a_j values will be exactly b.

Transformation

Let $(X = \{x_1, \ldots, x_m\}, \{S_1, \ldots, S_n\})$ be an instance of Ex-Cov.
We use $(n+1)$-*ary* encoding to avoid carry over.

Define numbers a_j as $a_j := \sum_{i=1}^{m} c_{ij} \cdot d_i$ with $c_{ij} = 1$ if $x_i \in S_j$ and $d_i = (n+1)^{i-1}$.
Set $b := \sum_{i=1}^{m} (n+1)^{i-1}$.

(Ex-Cov) $\Rightarrow$ (SS): since an exact cover contains each element exactly once, no carry over can occur and the sum of a_j values will be exactly b. ✓

Transformation

Let $(X = \{x_1, \ldots, x_m\}, \{S_1, \ldots, S_n\})$ be an instance of Ex-Cov.
We use $(n+1)$-*ary* encoding to avoid carry over.

Define numbers a_j as $a_j := \sum_{i=1}^{m} c_{ij} \cdot d_i$ with $c_{ij} = 1$ if $x_i \in S_j$ and $d_i = (n+1)^{i-1}$.
Set $b := \sum_{i=1}^{m} (n+1)^{i-1}$.

(Ex-Cov) $\Rightarrow$ (SS): since an exact cover contains each element exactly once, no carry over can occur and the sum of a_j values will be exactly b. ✓

(SS) $\Rightarrow$ (Ex-Cov): given a solution to (SS), we know the a_j values will sum to b. The corresponding sets S_j have to be an exact cover, as duplicate elements would result in carry overs that make it impossible to sum to b, and missing elements would result in a value that has to be lower than b.

Transformation

Let $(X = \{x_1, \ldots, x_m\}, \{S_1, \ldots, S_n\})$ be an instance of Ex-Cov.
We use $(n+1)$-*ary* encoding to avoid carry over.

Define numbers a_j as $a_j := \sum_{i=1}^{m} c_{ij} \cdot d_i$ with $c_{ij} = 1$ if $x_i \in S_j$ and $d_i = (n+1)^{i-1}$.
Set $b := \sum_{i=1}^{m} (n+1)^{i-1}$.

(Ex-Cov) $\Rightarrow$ (SS): since an exact cover contains each element exactly once, no carry over can occur and the sum of a_j values will be exactly b. ✓

(SS) $\Rightarrow$ (Ex-Cov): given a solution to (SS), we know the a_j values will sum to b. The corresponding sets S_j have to be an exact cover, as duplicate elements would result in carry overs that make it impossible to sum to b, and missing elements would result in a value that has to be lower than b. ✓

Transformation

Let $(X = \{x_1, \ldots, x_m\}, \{S_1, \ldots, S_n\})$ be an instance of Ex-Cov.
We use $(n+1)$-*ary* encoding to avoid carry over.

Define numbers a_j as $a_j := \sum_{i=1}^{m} c_{ij} \cdot d_i$ with $c_{ij} = 1$ if $x_i \in S_j$ and $d_i = (n+1)^{i-1}$.
Set $b := \sum_{i=1}^{m} (n+1)^{i-1}$.

(Ex-Cov) $\Rightarrow$ (SS): since an exact cover contains each element exactly once, no carry over can occur and the sum of a_j values will be exactly b. ✓

(SS) $\Rightarrow$ (Ex-Cov): given a solution to (SS), we know the a_j values will sum to b. The corresponding sets S_j have to be an exact cover, as duplicate elements would result in carry overs that make it impossible to sum to b, and missing elements would result in a value that has to be lower than b. ✓

Polynomial time:

Transformation

Let $(X = \{x_1, \ldots, x_m\}, \{S_1, \ldots, S_n\})$ be an instance of Ex-Cov.
We use $(n+1)$-*ary* encoding to avoid carry over.

Define numbers a_j as $a_j := \sum_{i=1}^{m} c_{ij} \cdot d_i$ with $c_{ij} = 1$ if $x_i \in S_j$ and $d_i = (n+1)^{i-1}$.
Set $b := \sum_{i=1}^{m} (n+1)^{i-1}$.

(Ex-Cov) $\Rightarrow$ (SS): since an exact cover contains each element exactly once, no carry over can occur and the sum of a_j values will be exactly b. ✓

(SS) $\Rightarrow$ (Ex-Cov): given a solution to (SS), we know the a_j values will sum to b. The corresponding sets S_j have to be an exact cover, as duplicate elements would result in carry overs that make it impossible to sum to b, and missing elements would result in a value that has to be lower than b. ✓

Polynomial time: (Ex-Cov) transforms to $n+1$ numbers with upper bound $(n+1)^{m-1} = O(n^m)$.

Transformation

Let $(X = \{x_1, \ldots, x_m\}, \{S_1, \ldots, S_n\})$ be an instance of Ex-Cov.
We use $(n+1)$-*ary* encoding to avoid carry over.

Define numbers a_j as $a_j := \sum_{i=1}^{m} c_{ij} \cdot d_i$ with $c_{ij} = 1$ if $x_i \in S_j$ and $d_i = (n+1)^{i-1}$.
Set $b := \sum_{i=1}^{m} (n+1)^{i-1}$.

(Ex-Cov) $\Rightarrow$ (SS): since an exact cover contains each element exactly once, no carry over can occur and the sum of a_j values will be exactly b. ✓

(SS) $\Rightarrow$ (Ex-Cov): given a solution to (SS), we know the a_j values will sum to b. The corresponding sets S_j have to be an exact cover, as duplicate elements would result in carry overs that make it impossible to sum to b, and missing elements would result in a value that has to be lower than b. ✓

Polynomial time: (Ex-Cov) transforms to $n+1$ numbers with upper bound $(n+1)^{m-1} = O(n^m)$.
With *binary encoding*, each number needs at most $O(m \log_2 n)$ bits.

Transformation

Let $(X = \{x_1, \ldots, x_m\}, \{S_1, \ldots, S_n\})$ be an instance of Ex-Cov.
We use $(n+1)$-*ary* encoding to avoid carry over.

Define numbers a_j as $a_j := \sum_{i=1}^{m} c_{ij} \cdot d_i$ with $c_{ij} = 1$ if $x_i \in S_j$ and $d_i = (n+1)^{i-1}$.
Set $b := \sum_{i=1}^{m} (n+1)^{i-1}$.

(Ex-Cov) $\Rightarrow$ (SS): since an exact cover contains each element exactly once, no carry over can occur and the sum of a_j values will be exactly b. ✓

(SS) $\Rightarrow$ (Ex-Cov): given a solution to (SS), we know the a_j values will sum to b. The corresponding sets S_j have to be an exact cover, as duplicate elements would result in carry overs that make it impossible to sum to b, and missing elements would result in a value that has to be lower than b. ✓

Polynomial time: (Ex-Cov) transforms to $n+1$ numbers with upper bound $(n+1)^{m-1} = O(n^m)$.
With *binary encoding*, each number needs at most $O(m \log_2 n)$ bits. ✓

2-Partition

2-PARTITION

Instance: positive integers $a_1, \ldots, a_n$ with $\sum_{i=1}^{n} a_i = 2A$.

Question: does there exist an index set $I \subseteq \{1, \ldots, n\}$ with $\sum_{i \in I} a_i = A$?

2-Partition

2-PARTITION

Instance: positive integers $a_1, \ldots, a_n$ with $\sum_{i=1}^{n} a_i = 2A$.

Question: does there exist an index set $I \subseteq \{1, \ldots, n\}$ with $\sum_{i \in I} a_i = A$?

Consider the following reduction from Subset Sum with numbers $a_1, \ldots a_n$ and goal b.

2-Partition

2-PARTITION

Instance: positive integers $a_1, \ldots, a_n$ with $\sum_{i=1}^{n} a_i = 2A$.

Question: does there exist an index set $I \subseteq \{1, \ldots, n\}$ with $\sum_{i \in I} a_i = A$?

Consider the following reduction from Subset Sum with numbers $a_1, \ldots a_n$ and goal b.

Define $T = \sum_{i=1}^{n} a_i$.

2-Partition

2-PARTITION

Instance: positive integers $a_1, \ldots, a_n$ with $\sum_{i=1}^{n} a_i = 2A$.

Question: does there exist an index set $I \subseteq \{1, \ldots, n\}$ with $\sum_{i \in I} a_i = A$?

Consider the following reduction from Subset Sum with numbers $a_1, \ldots a_n$ and goal b.

Define $T = \sum_{i=1}^{n} a_i$. Now construct an instance of 2-Partition with $n + 2$ numbers a'_i such that $a'_i = a_i$ for $1, \ldots, n$. Then define $a'_{n+1} = T + b$ and $a'_{n+2} = 2T - b$.

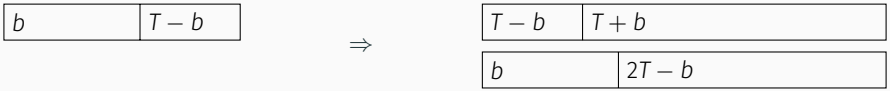
2-PARTITION

Instance: positive integers $a_1, \ldots, a_n$ with $\sum_{i=1}^{n} a_i = 2A$.

Question: does there exist an index set $I \subseteq \{1, \ldots, n\}$ with $\sum_{i \in I} a_i = A$?

Consider the following reduction from Subset Sum with numbers $a_1, \ldots a_n$ and goal b.

Define $T = \sum_{i=1}^{n} a_i$. Now construct an instance of 2-Partition with $n + 2$ numbers a'_i such that $a'_i = a_i$ for $1, \ldots, n$. Then define $a'_{n+1} = T + b$ and $a'_{n+2} = 2T - b$.



2-Partition

Polynomial time:

2-Partition

Polynomial time: Only two numbers added, of which the magnitude depends directly on the input size of the Subset Sum numbers.

2-Partition

Polynomial time: Only two numbers added, of which the magnitude depends directly on the input size of the Subset Sum numbers. ✓

2-Partition

Polynomial time: Only two numbers added, of which the magnitude depends directly on the input size of the Subset Sum numbers. ✓

(SS) $\Rightarrow$ (2-Partition):

2-Partition

Polynomial time: Only two numbers added, of which the magnitude depends directly on the input size of the Subset Sum numbers. ✓

(SS) $\Rightarrow$ **(2-Partition)**: Given a subset $\mathcal{S}$ that sums to b, determine is $T - b \geq b$, and allocate numbers a'_n and a_{n+1} according either to S or it's complement depending on this.

Polynomial time: Only two numbers added, of which the magnitude depends directly on the input size of the Subset Sum numbers. ✓

(SS) $\Rightarrow$ (2-Partition): Given a subset $\mathcal{S}$ that sums to b, determine is $T - b \geq b$, and allocate numbers a'_n and a_{n+1} according either to S or it's complement depending on this. This will give one partition that adds up to $b + (2T - b) = 2T$ and another partition that adds up to $T - b + (T + b) = 2T$.

Polynomial time: Only two numbers added, of which the magnitude depends directly on the input size of the Subset Sum numbers. ✓

(SS) $\Rightarrow$ (2-Partition): Given a subset $\mathcal{S}$ that sums to b, determine is $T - b \geq b$, and allocate numbers a'_n and a_{n+1} according either to S or it's complement depending on this. This will give one partition that adds up to $b + (2T - b) = 2T$ and another partition that adds up to $T - b + (T + b) = 2T$. ✓

2-Partition

Polynomial time: Only two numbers added, of which the magnitude depends directly on the input size of the Subset Sum numbers. ✓

(SS) $\Rightarrow$ (2-Partition): Given a subset $\mathcal{S}$ that sums to b, determine is $T - b \geq b$, and allocate numbers a'_n and a_{n+1} according either to S or it's complement depending on this. This will give one partition that adds up to $b + (2T - b) = 2T$ and another partition that adds up to $T - b + (T + b) = 2T$. ✓

(2-Partition) $\Rightarrow$ (SS):

Polynomial time: Only two numbers added, of which the magnitude depends directly on the input size of the Subset Sum numbers. ✓

(SS) ⇒ (2-Partition): Given a subset $\mathcal{S}$ that sums to b, determine is $T - b \geq b$, and allocate numbers a'_n and a_{n+1} according either to S or it's complement depending on this. This will give one partition that adds up to $b + (2T - b) = 2T$ and another partition that adds up to $T - b + (T + b) = 2T$. ✓

(2-Partition) ⇒ (SS): Given a valid partition into two sets, consider which set contains the $a'_{n+2} = 2T - b$ element. As the total sum is $4T$, both partitions must sum up to $2T$ and neither can contain both a'_{n+1} and a'_{n+2} as that would sum up to $3T$.

2-Partition

Polynomial time: Only two numbers added, of which the magnitude depends directly on the input size of the Subset Sum numbers. ✓

(SS) ⇒ (2-Partition): Given a subset $\mathcal{S}$ that sums to b, determine is $T - b \geq b$, and allocate numbers a'_n and a_{n+1} according either to S or it's complement depending on this. This will give one partition that adds up to $b + (2T - b) = 2T$ and another partition that adds up to $T - b + (T + b) = 2T$. ✓

(2-Partition) ⇒ (SS): Given a valid partition into two sets, consider which set contains the $a'_{n+2} = 2T - b$ element. As the total sum is $4T$, both partitions must sum up to $2T$ and neither can contain both a'_{n+1} and a'_{n+2} as that would sum up to $3T$. For the set with a'_{n+2}, the other elements in this partition must sum up to b and thus are a valid solution to the (SS) instance.

2-Partition

Polynomial time: Only two numbers added, of which the magnitude depends directly on the input size of the Subset Sum numbers. ✓

(SS) ⇒ (2-Partition): Given a subset $\mathcal{S}$ that sums to b, determine is $T - b \geq b$, and allocate numbers a'_n and a_{n+1} according either to S or it's complement depending on this. This will give one partition that adds up to $b + (2T - b) = 2T$ and another partition that adds up to $T - b + (T + b) = 2T$. ✓

(2-Partition) ⇒ (SS): Given a valid partition into two sets, consider which set contains the $a'_{n+2} = 2T - b$ element. As the total sum is $4T$, both partitions must sum up to $2T$ and neither can contain both a'_{n+1} and a'_{n+2} as that would sum up to $3T$. For the set with a'_{n+2}, the other elements in this partition must sum up to b and thus are a valid solution to the (SS) instance. ✓

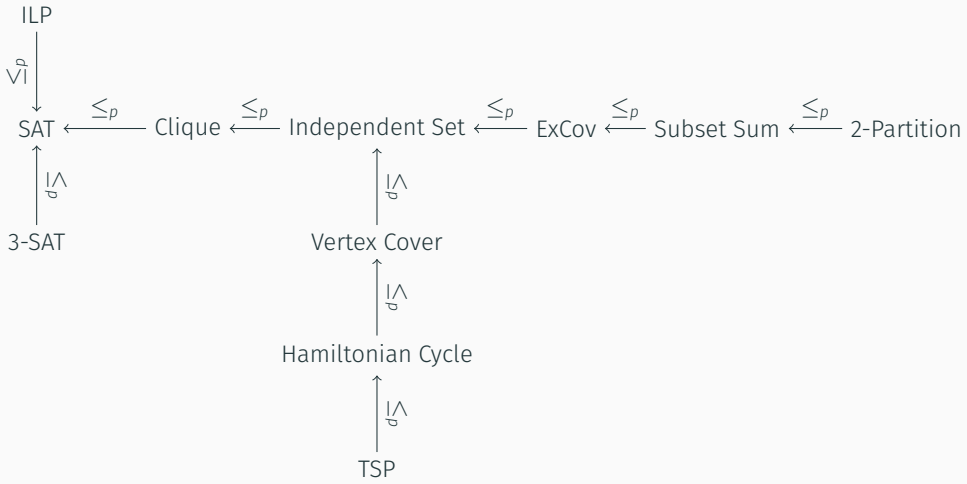
Subset Sum and 2-Partition

Theorems

Subset Sum is NP-hard and NP-complete.

2-Partition is NP-hard and NP-complete.

Reductions


ILP
$\leq_p$ VI
SAT
$\leq_p$
Clique
$\leq_p$
Independent Set
$\leq_p$
ExCov
$\leq_p$
Subset Sum
$\leq_p$
2-Partition
3-SAT
$\leq_p$ IA
Vertex Cover
$\leq_p$ IA
Hamiltonian Cycle
$\leq_p$ IA
TSP

Questions?

Pseudo-polynomial time

An algorithm for SUBSET SUM

Define function $F(k, c)$:
$F(k, c)$=TRUE if and only if $\exists S \subseteq \{1, \ldots, k\} : \sum_{i \in S} a_i = c$

This can be

$$F(k, c) = \begin{cases} \text{FALSE} & \text{if } k < 0 \vee c < 0 \\ \text{TRUE} & \text{if } c = 0 \\ F(k-1, c) \vee F(k-1, c-a_i) & \text{otherwise} \end{cases}$$

We can convert this recursion to a Dynamic Programming algorithm

An algorithm for SUBSET SUM

Dynamic programming algorithm to compute $F(n, b)$

```
def dp_subset_sum(numbers, target):
    k = len(numbers)
    # Initialize a Dynamic Programming Table and values F(0,j)
    table = [[False for j in range(target+1)] for i in range(k)]
    table[0][0] = True
    table[0][numbers[0]] = True
    # Construct the table
    for i in range(1,k):
        number = numbers[i]
        for j in range(target+1):
            table[i][j] = table[i-1][j]
            if j-number >= 0:
                table[i][j] = table[i-1][j] or table[i-1][j-number]
    # Return the final answer
    return table[k-1][target]
```

An algorithm for SUBSET SUM

Dynamic programming algorithm to compute $F(n, b)$

```
def dp_subset_sum(numbers, target):
    k = len(numbers)
    # Initialize a Dynamic Programming Table and values F(0,j)
    table = [[False for j in range(target+1)] for i in range(k)]
    table[0][0] = True
    table[0][numbers[0]] = True
    # Construct the table
    for i in range(1,k):
        number = numbers[i]
        for j in range(target+1):
            table[i][j] = table[i-1][j]
            if j-number >= 0:
                table[i][j] = table[i-1][j] or table[i-1][j-number]
    # Return the final answer
    return table[k-1][target]
```

Running time of this algorithm?

An algorithm for SUBSET SUM

Size of instance

size(I) = instance size
= length (number of symbols) of reasonable encoding of instance *I*

Size of numbers

number(I)
= value of the largest number occuring in instance *I*

An algorithm for SUBSET SUM

Size of instance

size(I) = instance size
= length (number of symbols) of reasonable encoding of instance *I*

Size of numbers

number(I)
= value of the largest number occuring in instance *I*

As computing a single entry in the table can be done in constant time, and the table contains nc entries, the algorithm for Subset Sum runs in $O(nc)$ time.

An algorithm for SUBSET SUM

Size of instance

size(I) = instance size
= length (number of symbols) of reasonable encoding of instance I

Size of numbers

number(I)
= value of the largest number occuring in instance I

As computing a single entry in the table can be done in constant time, and the table contains nc entries, the algorithm for Subset Sum runs in $O(nc)$ time.

As c is a number in the instance, this means the algorithm runs in $O(\text{size}(I)\text{number}(I))$ time.

An algorithm for SUBSET SUM

Size of instance

size(I) = instance size
= length (number of symbols) of reasonable encoding of instance *I*

Size of numbers

number(I)
= value of the largest number occuring in instance *I*

As computing a single entry in the table can be done in constant time, and the table contains nc entries, the algorithm for Subset Sum runs in $O(nc)$ time.

As c is a number in the instance, this means the algorithm runs in $O(\text{size}(I)\text{number}(I))$ time.

Implication: the algorithm run in polynomial time when the numbers are encoded in *unary*-encoding, which requires the same number of symbols as the size of the number (e.g. tally-marks/"turven").

Questions?

Pseudo-polynomial time

Definition

A decision problem X is solvable in pseudo-polynomial time,
if there exists an algorithm that solves instances I of X
in time polynomially bounded in size(I) and number(I).

Pseudo-polynomial time

Definition

A decision problem X is solvable in pseudo-polynomial time,
if there exists an algorithm that solves instances I of X
in time polynomially bounded in size(I) and number(I).

Which of the decision problems we studied so far is solvable in pseudo-polynomial time?

- SAT?
- 3-SAT?
- CLIQUE?
- IS?
- VC?
- Ex-Cov?
- SUBSET SUM
- PARTITION?
- HC?
- TSP?

Pseudo-polynomial time

Definition

A decision problem X is solvable in pseudo-polynomial time,
if there exists an algorithm that solves instances I of X
in time polynomially bounded in size(I) and number(I).

Observation: number(I) is only relevant for problems that involve numbers (distances, costs, weights, lengths, penalties, profits, time intervals, etc)

Which of the decision problems we studied so far is solvable in pseudo-polynomial time?

- SAT?
- 3-SAT?
- CLIQUE?
- IS?
- VC?
- Ex-Cov?
- SUBSET SUM
- PARTITION?
- HC?
- TSP?

Pseudo-polynomial time

Definition

A decision problem X is solvable in pseudo-polynomial time,
if there exists an algorithm that solves instances I of X
in time polynomially bounded in size(I) and number(I).

Observation: number(I) is only relevant for problems that involve numbers (distances, costs, weights, lengths, penalties, profits, time intervals, etc)

Which of the decision problems we studied so far is solvable in pseudo-polynomial time?

- SAT
- 3-SAT
- CLIQUE
- IS
- VC
- Ex-Cov
- SUBSET SUM
- PARTITION?
- HC
- TSP?

Pseudo-polynomial time

Definition

A decision problem X is solvable in pseudo-polynomial time,
if there exists an algorithm that solves instances I of X
in time polynomially bounded in size(I) and number(I).

Observation: number(I) is only relevant for problems that involve numbers (distances, costs, weights, lengths, penalties, profits, time intervals, etc)

Which of the decision problems we studied so far is solvable in pseudo-polynomial time?

- SAT
- 3-SAT
- CLIQUE
- IS
- VC
- Ex-Cov
- SUBSET SUM
- PARTITION?
- HC
- TSP

Pseudo-polynomial time

Definition

A decision problem X is solvable in pseudo-polynomial time,
if there exists an algorithm that solves instances I of X
in time polynomially bounded in size(I) and number(I).

Observation: number(I) is only relevant for problems that involve numbers (distances, costs, weights, lengths, penalties, profits, time intervals, etc)

Which of the decision problems we studied so far is solvable in pseudo-polynomial time?

- SAT
- 3-SAT
- CLIQUE
- IS
- VC
- Ex-Cov
- SUBSET SUM
- PARTITION (exercise)
- HC
- TSP

Strong NP-hardness

Definition

A decision problem X is strongly NP-hard,
if there exists a polynomial $p : \mathbb{N} \to \mathbb{N}$
such that
restriction of X to instances I with number(I) $\leq p$(size(I)) is NP-hard.

Strong NP-hardness

Definition

A decision problem X is strongly NP-hard,
if there exists a polynomial $p : \mathbb{N} \to \mathbb{N}$
such that
restriction of X to instances I with number(I) $\leq p$(size(I)) is NP-hard.

- SAT, CLIQUE, IS, VC, HC, TSP are strongly NP-hard
- unary NP-hard = strongly NP-hard
- weak NP-hard = NP-hard, but may be solvable in pseudo-polynomial time

Strong NP-hardness

Definition

A decision problem X is strongly NP-hard,
if there exists a polynomial $p : \mathbb{N} \to \mathbb{N}$
such that
restriction of X to instances I with number(I) $\leq p$(size(I)) is NP-hard.

- SAT, CLIQUE, IS, VC, HC, TSP are strongly NP-hard
- unary NP-hard = strongly NP-hard
- weak NP-hard = NP-hard, but may be solvable in pseudo-polynomial time

Theorem

If decision problem X
is strongly NP-hard and solvable in pseudo-polynomial time
then P=NP.

Are all number problems only weakly NP-hard?

THREE PARTITION

Instance: positive integers $a_1, \ldots, a_{3n}$ with $\sum_{i=1}^{3n} a_i = nA$
Question: does there exists a partition of the index set $\{1, \ldots, 3n\}$
into n three-element subsets $T_1, \ldots, T_n$ such that
every three-element set T satisfies $\sum_{i \in T} a_i = A$

Are all number problems only weakly NP-hard?

THREE PARTITION

Instance: positive integers $a_1, \ldots, a_{3n}$ with $\sum_{i=1}^{3n} a_i = nA$
Question: does there exists a partition of the index set $\{1, \ldots, 3n\}$
into n three-element subsets $T_1, \ldots, T_n$ such that
every three-element set T satisfies $\sum_{i \in T} a_i = A$

Theorem

THREE PARTITION is strongly NP-complete.

THREE PARTITION

Instance: positive integers $a_1, \ldots, a_{3n}$ with $\sum_{i=1}^{3n} a_i = nA$
Question: does there exists a partition of the index set $\{1, \ldots, 3n\}$
into n three-element subsets $T_1, \ldots, T_n$ such that
every three-element set T satisfies $\sum_{i \in T} a_i = A$

Theorem

THREE PARTITION is strongly NP-complete.

Proof: proof in Garey-Johnson shows that $SAT \leq_p 3DM \leq_p 4 - PARTITION \leq_p 3 - PARTITION$
Where the instance I constructed in the proof of $3DM \leq_p 4 - PARTITION$ has $number(I) \leq 2^{16}|A|^4$.

Questions?

NP versus coNP

Co-NP

Recall:

Definition

A decision problem X lies in the complexity class NP,
if the YES-instances of X possess certificates of polynomial length
that can be verified in polynomial time

A decision problem X is NP-complete,
if $X \in NP$ and all problems $Y \in NP$ can be reduced to it.

Co-NP

Recall:

Definition

A decision problem X lies in the complexity class NP,
if the YES-instances of X possess certificates of polynomial length
that can be verified in polynomial time

A decision problem X is NP-complete,
if $X \in NP$ and all problems $Y \in NP$ can be reduced to it.

Now we define:

Definition

A decision problem X lies in the complexity class coNP,
if the NO-instances of X possess certificates of polynomial length
that can be verified in polynomial time

Co-NP

Recall:

Definition

A decision problem X lies in the complexity class NP,
if the YES-instances of X possess certificates of polynomial length
that can be verified in polynomial time

A decision problem X is NP-complete,
if $X \in NP$ and all problems $Y \in NP$ can be reduced to it.

Now we define:

Definition

A decision problem X lies in the complexity class coNP,
if the NO-instances of X possess certificates of polynomial length
that can be verified in polynomial time

A decision problem X is $coNP$-complete,
if $X \in coNP$ and all problems $Y \in coNP$ can be reduced to it.

Problems in coNP

Non-HAMILTONICITY

Instance: an undirected graph $G = (V, E)$

Question: is G not Hamiltonian?

Problems in coNP

Non-HAMILTONICITY

Instance: an undirected graph $G = (V, E)$

Question: is G not Hamiltonian?

Un-Satisfiability (UNSAT)

Instance:

a set of logical variables $X := \{x_1, \ldots, x_n\}$ and a set of clauses C over X

Question: Is there no truth assignment for X that simultaneously satisfies all clauses in C?

Problems in coNP

Non-HAMILTONICITY

Instance: an undirected graph $G = (V, E)$
Question: is G not Hamiltonian?

Un-Satisfiability (UNSAT)

Instance:
a set of logical variables $X := \{x_1, \ldots, x_n\}$ and a set of clauses C over X

Question: Is there no truth assignment for X that simultaneously satisfies all clauses in C?

TAUTOLOGY

Instance: a set of logical variables $X := \{x_1, \ldots, x_n\}$ and a logical formula Φ in disjunctive normal form (DNF) over X

Question: are all truth settings for X satisfying for Φ?

Problems in coNP

Theorem

Non-HAMILTONICITY, UNSAT and TAUTOLOGY are coNP-complete.

Problems in coNP

Theorem

Non-HAMILTONICITY, UNSAT and TAUTOLOGY are coNP-complete.

Lemma

If X is NP-complete, $\bar{X}$ is coNP-complete.

$\Rightarrow$ NP-completeness of Non-HAMILTONICITY & UNSAT

Questions?

Excursion: Logical formulas

Let X be a set of logical *variables*.

- *Truth assignment*: $t : X \to \{true, false\}$
- *Literals:* We call x and $\neg x$ literals corresponding to variable $x \in X$. $t(\neg x) = true \Leftrightarrow t(x) = false$
- *(disjunctive) clause* over X: disjunction of literals, e.g., $(x_1 \vee \neg x_2 \vee \ldots \vee x_k)$.

Excursion: Logical formulas

Let X be a set of logical *variables*.

- *Truth assignment*: $t : X \to \{true, false\}$
- *Literals:* We call x and $\neg x$ literals corresponding to variable $x \in X$. $t(\neg x) = true \Leftrightarrow t(x) = false$
- *(disjunctive) clause* over X: disjunction of literals, e.g., $(x_1 \vee \neg x_2 \vee \ldots \vee x_k)$.
- *conjunctive clause* over X: conjunction of literals, e.g., $(\neg x_1 \wedge x_2 \wedge \ldots \wedge x_j)$.

Excursion: Logical formulas

Let X be a set of logical *variables*.

- *Truth assignment*: $t : X \to \{true, false\}$
- *Literals:* We call x and $\neg x$ literals corresponding to variable $x \in X$. $t(\neg x) = true \Leftrightarrow t(x) = false$
- *(disjunctive) clause* over X: disjunction of literals, e.g., $(x_1 \vee \neg x_2 \vee \ldots \vee x_k)$.
- *conjunctive clause* over X: conjunction of literals, e.g., $(\neg x_1 \wedge x_2 \wedge \ldots \wedge x_j)$.
- logical formula in X:
  (general) logical expression in variables from X, e.g., $[(x1 \vee \neg x_2) \wedge (x_1 \vee x_3)] \vee \neg(x_1 \vee x_2)$

Excursion: Logical formulas

Let X be a set of logical *variables*.

- *Truth assignment*: $t : X \to \{true, false\}$
- *Literals:* We call x and $\neg x$ literals corresponding to variable $x \in X$. $t(\neg x) = true \Leftrightarrow t(x) = false$
- *(disjunctive) clause* over X: disjunction of literals, e.g., $(x_1 \vee \neg x_2 \vee \ldots \vee x_k)$.
- *conjunctive clause* over X: conjunction of literals, e.g., $(\neg x_1 \wedge x_2 \wedge \ldots \wedge x_j)$.
- logical formula in X:
  (general) logical expression in variables from X, e.g., $[(x1 \vee \neg x_2) \wedge (x_1 \vee x_3)] \vee \neg(x_1 \vee x_2)$
- logical formula in conjunctive normal form (CNF):
  conjunction of disjunctive clauses, e.g. $(x1 \vee \neg x_2) \wedge (x_1 \vee x_3)$
- logical formula in disjunctive normal form (DNF):
  disjunction of conjunctive clauses, e.g. $(x1 \wedge \neg x_2) \vee (x_1 \wedge x_3)$

Satisfiability (SAT) - as we use it / CNF-SAT

Instance: set of logical variables $X := \{x_1, \ldots, x_n\}$, logical formula Φ in CNF
Question: does there exist a truth assignment for X that satisfies Φ?

Satisfiability (SAT) - as we use it / CNF-SAT

Instance: set of logical variables $X := \{x_1, \ldots, x_n\}$, logical formula Φ in CNF
Question: does there exist a truth assignment for X that satisfies Φ?

Satisfiability (SAT) - more general

Instance: set of logical variables $X := \{x_1, \ldots, x_n\}$, (any) logical formula Φ
Question: does there exist a truth assignment for X that satisfies Φ?
NP-complete (in NP & generalization of CNF-SAT)

Excursion: Logical formulas

Satisfiability (SAT) - as we use it / CNF-SAT

Instance: set of logical variables $X := \{x_1, \ldots, x_n\}$, logical formula Φ in CNF
Question: does there exist a truth assignment for X that satisfies Φ?

Satisfiability (SAT) - more general

Instance: set of logical variables $X := \{x_1, \ldots, x_n\}$, (any) logical formula Φ
Question: does there exist a truth assignment for X that satisfies Φ?
NP-complete (in NP & generalization of CNF-SAT)

DNF-SAT

Instance: set of logical variables $X := \{x_1, \ldots, x_n\}$, logical formula Φ in DNF
Question: does there exist a truth assignment for X that satisfies Φ?
In P (as the formula can be interpreted as an enumeration of solutions)

De Morgan's law

- $\neg(l_1 \wedge l_1) = \neg l_1 \vee \neg l_2$
- $\neg(x \vee y) = \neg x \wedge \neg y$

This implies that the negation of a CNF formula is a DNF formula and vice versa.

TAUTOLOGY is coNP-complete

TAUTOLOGY

Instance: a set of logical variables $X := \{x_1, \ldots, x_n\}$ and DNF-formula Φ over X
Question: are all truth settings for X satisfying assignments for Φ?

Theorem

TAUTOLOGY is coNP-complete.

Proof:

TAUTOLOGY is coNP-complete

TAUTOLOGY

Instance: a set of logical variables $X := \{x_1, \ldots, x_n\}$ and DNF-formula Φ over X
Question: are all truth settings for X satisfying assignments for Φ?

Theorem

TAUTOLOGY is coNP-complete.

Proof: We show: CNF-SAT $\leq_p \overline{TAUTOLOGY}$.

TAUTOLOGY is coNP-complete

TAUTOLOGY

Instance: a set of logical variables $X := \{x_1, \ldots, x_n\}$ and DNF-formula Φ over X
Question: are all truth settings for X satisfying assignments for Φ?

Theorem

TAUTOLOGY is coNP-complete.

Proof: We show: CNF-SAT $\leq_p \overline{TAUTOLOGY}$.

$\overline{TAUTOLOGY}$

Instance: a set of logical variables $X := \{x_1, \ldots, x_n\}$ and a DNF-formula Φ over X
Question: is there a truth setting for X that does not satisfy Φ?

TAUTOLOGY is coNP-complete

TAUTOLOGY

Instance: a set of logical variables $X := \{x_1, \ldots, x_n\}$ and DNF-formula Φ over X
Question: are all truth settings for X satisfying assignments for Φ?

Theorem

TAUTOLOGY is coNP-complete.

Proof: We show: CNF-SAT $\leq_p \overline{TAUTOLOGY}$.

$\overline{TAUTOLOGY}$

Instance: a set of logical variables $X := \{x_1, \ldots, x_n\}$ and a DNF-formula Φ over X
Question: is there a truth setting for X that does not satisfy Φ?
Question reformulated: is there a truth setting for X that satisfies $\neg\Phi$?

TAUTOLOGY is coNP-complete

TAUTOLOGY

Instance: a set of logical variables $X := \{x_1, \ldots, x_n\}$ and DNF-formula Φ over X
Question: are all truth settings for X satisfying assignments for Φ?

Theorem

TAUTOLOGY is coNP-complete.

Proof: We show: CNF-SAT $\leq_p \overline{TAUTOLOGY}$.

$\overline{TAUTOLOGY}$

Instance: a set of logical variables $X := \{x_1, \ldots, x_n\}$ and a DNF-formula Φ over X
Question: is there a truth setting for X that does not satisfy Φ?
Question reformulated: is there a truth setting for X that satisfies $\neg\Phi$?

Let X' be a set of logical variables and Φ' a CNF-formula on X.

TAUTOLOGY is coNP-complete

TAUTOLOGY

Instance: a set of logical variables $X := \{x_1, \ldots, x_n\}$ and DNF-formula Φ over X
Question: are all truth settings for X satisfying assignments for Φ?

Theorem

TAUTOLOGY is coNP-complete.

Proof: We show: CNF-SAT $\leq_p \overline{TAUTOLOGY}$.

$\overline{TAUTOLOGY}$

Instance: a set of logical variables $X := \{x_1, \ldots, x_n\}$ and a DNF-formula Φ over X
Question: is there a truth setting for X that does not satisfy Φ?
Question reformulated: is there a truth setting for X that satisfies $\neg\Phi$?

Let X' be a set of logical variables and Φ' a CNF-formula on X.
Then $\Phi := \neg\Phi'$ is a DNF-formula on X (De-Morgan's law).

TAUTOLOGY is coNP-complete

TAUTOLOGY

Instance: a set of logical variables $X := \{x_1, \ldots, x_n\}$ and DNF-formula Φ over X
Question: are all truth settings for X satisfying assignments for Φ?

Theorem

TAUTOLOGY is coNP-complete.

Proof: We show: CNF-SAT $\leq_p \overline{TAUTOLOGY}$.

$\overline{TAUTOLOGY}$

Instance: a set of logical variables $X := \{x_1, \ldots, x_n\}$ and a DNF-formula Φ over X
Question: is there a truth setting for X that does not satisfy Φ?
Question reformulated: is there a truth setting for X that satisfies $\neg\Phi$?

Let X' be a set of logical variables and Φ' a CNF-formula on X.
Then $\Phi := \neg\Phi'$ is a DNF-formula on X (De-Morgan's law).
Thus (X, Φ) is an instance of $\overline{TAUTOLOGY}$

TAUTOLOGY is coNP-complete

TAUTOLOGY

Instance: a set of logical variables $X := \{x_1, \ldots, x_n\}$ and DNF-formula Φ over X
Question: are all truth settings for X satisfying assignments for Φ?

Theorem

TAUTOLOGY is coNP-complete.

Proof: We show: CNF-SAT $\leq_p \overline{TAUTOLOGY}$.

$\overline{TAUTOLOGY}$

Instance: a set of logical variables $X := \{x_1, \ldots, x_n\}$ and a DNF-formula Φ over X
Question: is there a truth setting for X that does not satisfy Φ?
Question reformulated: is there a truth setting for X that satisfies $\neg\Phi$?

Let X' be a set of logical variables and Φ' a CNF-formula on X.
Then $\Phi := \neg\Phi'$ is a DNF-formula on X (De-Morgan's law).
Thus (X, Φ) is an instance of $\overline{TAUTOLOGY}$ which is satisfiable if and only in (X', Φ') is satisfiable.

Questions?

NP versus coNP

Problems in $NP \cap coNP$ have

- good certificates for YES-instances
- good certificates for NO-instances

Example

MaxFlow (LP):

Instance: a directed graph $G = (V, E)$ with a dedicated source node s and sink node t and edge capacities c_e for all $e \in E$, a flow value F

Question: Is there an s-t-*flow* of size at least F?

That is: is there an assignment $f : E \to \mathcal{R}$ such that
$\sum_{e \text{ outgoing edge of } s} f_e = F$,
$\sum_{e \text{ ingoing edge of } t} f_e = F$,
$\sum_{e \text{ ingoing edge of } t} f_e - \sum_{e \text{ outgoing edge of } v} f_e = 0$?

The Soviet railway system problem

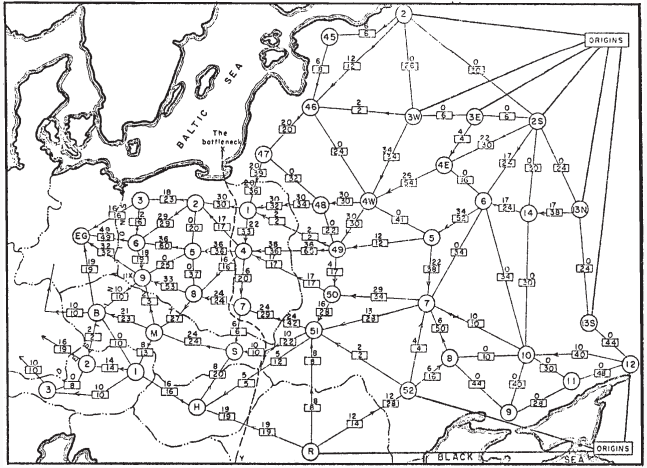


Fig. 2. From Harris and Ross [11]: Schematic diagram of the railway network of the Western Soviet Union and Eastern European countries, with a maximum flow of value 163,000 tons from Russia to Eastern Europe, and a cut of capacity 163,000 tons indicated as "The bottleneck"

Example

MaxFlow (LP):
Instance: a directed graph $G = (V, E)$ with a dedicated source node s and sink node t and edge capacities c_e for all $e \in E$, a flow value F

Question: Is there an s-t-*flow* of size at least F?

That is: is there an assignment $f : E \to \mathcal{R}$ such that
$\sum_{e \text{ outgoing edge of } s} f_e = F$,
$\sum_{e \text{ ingoing edge of } t} f_e = F$,
$\sum_{e \text{ ingoing edge of } t} f_e - \sum_{e \text{ outgoing edge of } v} f_e = 0$?

Example

MaxFlow (LP):
Instance: a directed graph $G = (V, E)$ with a dedicated source node s and sink node t and edge capacities c_e for all $e \in E$, a flow value F

Question: Is there an s-t-*flow* of size at least F?

That is: is there an assignment $f : E \to \mathcal{R}$ such that
$\sum_{e \text{ outgoing edge of } s} f_e = F$,
$\sum_{e \text{ ingoing edge of } t} f_e = F$,
$\sum_{e \text{ ingoing edge of } t} f_e - \sum_{e \text{ outgoing edge of } v} f_e = 0$?

MaxFlow in NP

NP versus coNP (3)

Example

MaxFlow (LP):
Instance: a directed graph $G = (V, E)$ with a dedicated source node s and sink node t and edge capacities c_e for all $e \in E$, a flow value F

Question: Is there an s-t-*flow* of size at least F?

That is: is there an assignment $f : E \to \mathcal{R}$ such that
$\sum_{e \text{ outgoing edge of } s} f_e = F$,
$\sum_{e \text{ ingoing edge of } t} f_e = F$,
$\sum_{e \text{ ingoing edge of } t} f_e - \sum_{e \text{ outgoing edge of } v} f_e = 0$?

MaxFlow in NP
MaxFlow in coNP (min-cut/max-flow theorem)

Example

MaxFlow (LP):
Instance: a directed graph $G = (V, E)$ with a dedicated source node s and sink node t and edge capacities c_e for all $e \in E$, a flow value F

Question: Is there an s-t-*flow* of size at least F?

That is: is there an assignment $f : E \to \mathcal{R}$ such that
$\sum_{e \text{ outgoing edge of } s} f_e = F$,
$\sum_{e \text{ ingoing edge of } t} f_e = F$,
$\sum_{e \text{ ingoing edge of } t} f_e - \sum_{e \text{ outgoing edge of } v} f_e = 0$?

MaxFlow in NP
MaxFlow in coNP (min-cut/max-flow theorem)
MaxFlow is in P (Ford-Fulkerson algorithm)

NP versus coNP (3)

Example

Linear Programming (LP): Instance: a matrix A; vectors c and b; a bound t

Question: does there exist a real vector x with $Ax \leq b$ and $cx \leq t$?

NP versus coNP (3)

Example

Linear Programming (LP): Instance: a matrix A; vectors c and b; a bound t

Question: does there exist a real vector x with $Ax \leq b$ and $cx \leq t$?

- LP lies in NP

NP versus coNP (3)

Example

Linear Programming (LP): Instance: a matrix A; vectors c and b; a bound t

Question: does there exist a real vector x with $Ax \leq b$ and $cx \leq t$?

- LP lies in NP
- LP lies in coNP

NP versus coNP (3)

Example

Linear Programming (LP): Instance: a matrix A; vectors c and b; a bound t

Question: does there exist a real vector x with $Ax \leq b$ and $cx \leq t$?

- LP lies in NP
- LP lies in coNP

LP-duality tells us that if and only if there is a y with $A^t y = c$ and $b^t y > t$, then the answer is 'NO'.

NP versus coNP (3)

Example

Linear Programming (LP): Instance: a matrix A; vectors c and b; a bound t

Question: does there exist a real vector x with $Ax \leq b$ and $cx \leq t$?

- LP lies in NP
- LP lies in coNP

LP-duality tells us that if and only if there is a y with $A^t y = c$ and $b^t y > t$, then the answer is 'NO'.

LP lies in P (see, e.g., Karmakar's algorithm).

Is $P = NP \cap coNP$?

- FACT: $P \subseteq NP \cap coNP$

Is $P = NP \cap coNP$?

- FACT: $P \subseteq NP \cap coNP$
- Some people think that $P \neq NP \cap coNP$
- Some people think that $P = NP \cap coNP$

Is $P = NP \cap coNP$?

- FACT: $P \subseteq NP \cap coNP$
- Some people think that $P \neq NP \cap coNP$
- Some people think that $P = NP \cap coNP$

Problems that are in $NP \cap coNP$ but not known to be in P are rare.

Two famous examples are *Graph Isomorphism* and *Integer Factorization*.

Questions?

Example

Integer Factorization (version a):
Instance: integers y, u (given in binary).
Question: Is there an integer x that divides y and satisfies $1 < x \leq u$?

Version a) is in NP

Example

Integer Factorization (version a):
Instance: integers y, u (given in binary).
Question: Is there an integer x that divides y and satisfies $1 < x \leq u$?

Version a) is in NP and coNP.

Example

Integer Factorization (version a):
Instance: integers y, u (given in binary).
Question: Is there an integer x that divides y and satisfies $1 < x \leq u$?

Version a) is in NP and coNP.
It is not known whether it is in P, NP-complete, coNP-complete, or none of this.

Example

Integer Factorization (version a):
Instance: integers y, u (given in binary).
Question: Is there an integer x that divides y and satisfies $1 < x \leq u$?

Version a) is in NP and coNP.
It is not known whether it is in P, NP-complete, coNP-complete, or none of this.

Note: basic arithmetic (division, multiplication) is in polynomial time. Primality testing is in P.

Example

Integer Factorization (version a):
Instance: integers y, u (given in binary).
Question: Is there an integer x that divides y and satisfies $1 < x \leq u$?

Version a) is in NP and coNP.
It is not known whether it is in P, NP-complete, coNP-complete, or none of this.

Note: basic arithmetic (division, multiplication) is in polynomial time. Primality testing is in P.

Many cryptographic protocols are based on the difficulty of factoring large composite integers - an algorithm that efficiently factors an arbitrary integer would render these insecure.

Example

Integer Factorization (version b):
Instance: integers y, l, u (given in binary).
Question: Is there an integer x that divides y and satisfies $l \leq x \leq u$?

Version b) is in NP.
Is it also in coNP?

It is not straightforward to give a NO-certificate.

Example

Integer Factorization (version b):
Instance: integers y, l, u (given in binary).
Question: Is there an integer x that divides y and satisfies $l \leq x \leq u$?

Version b) is in NP.
Is it also in coNP?

It is not straightforward to give a NO-certificate.
The integer factorization $\Pi_{i=1,\ldots m} p_i$ with p_i primes (not necessarily different ones) does not (directly) give us a certificate.

Example

Integer Factorization (version b):
Instance: integers y, l, u (given in binary).
Question: Is there an integer x that divides y and satisfies $l \leq x \leq u$?

Version b) is in NP.
Is it also in coNP?

It is not straightforward to give a NO-certificate.
The integer factorization $\Pi_{i=1,\ldots m} p_i$ with p_i primes (not necessarily different ones) does not (directly) give us a certificate.
Any ideas?

Example

Integer Factorization (version b):
Instance: integers y, l, u (given in binary).
Question: Is there an integer x that divides y and satisfies $l \leq x \leq u$?

Version b) is in NP.
Is it also in coNP?

It is not straightforward to give a NO-certificate.
The integer factorization $\Pi_{i=1,\ldots m} p_i$ with p_i primes (not necessarily different ones) does not (directly) give us a certificate.
Any ideas?
Idea: work with the integer factorization, and then consider for each cardinality $1, \ldots, m$ the possible sets. For a fixed cardinality the sets are nicely ordered.

Example

Integer Factorization (version b):
Instance: integers y, l, u (given in binary).
Question: Is there an integer x that divides y and satisfies $l \leq x \leq u$?

Version b) is in NP.
Is it also in coNP?

It is not straightforward to give a NO-certificate.
The integer factorization $\Pi_{i=1,\ldots m} p_i$ with p_i primes (not necessarily different ones) does not (directly) give us a certificate.
Any ideas?
Idea: work with the integer factorization, and then consider for each cardinality $1, \ldots, m$ the possible sets. For a fixed cardinality the sets are nicely ordered. Then construct a certificate for each cardinality separately.

NP versus coNP

- FACT: $P \subseteq NP \cap coNP$
- Some people think that $P \neq NP \cap coNP$
- Some people think that $P = NP \cap coNP$

NP versus coNP

- FACT: $P \subseteq NP \cap coNP$
- Some people think that $P \neq NP \cap coNP$
- Some people think that $P = NP \cap coNP$

- Most people think that $NP \neq coNP$

NP versus coNP

- FACT: $P \subseteq NP \cap coNP$
- Some people think that $P \neq NP \cap coNP$
- Some people think that $P = NP \cap coNP$

- Most people think that $NP \neq coNP$

Theorem

If coNP contains some NP-complete problem X, then NP=coNP.

NP versus coNP

- FACT: $P \subseteq NP \cap coNP$
- Some people think that $P \neq NP \cap coNP$
- Some people think that $P = NP \cap coNP$

- Most people think that $NP \neq coNP$

Theorem

If coNP contains some NP-complete problem X, then NP=coNP.

Hence:

- X being NP-complete is indication for $X \notin coNP$

NP versus coNP

- FACT: $P \subseteq NP \cap coNP$
- Some people think that $P \neq NP \cap coNP$
- Some people think that $P = NP \cap coNP$

- Most people think that $NP \neq coNP$

Theorem

If coNP contains some NP-complete problem X, then NP=coNP.

Hence:

- X being NP-complete is indication for $X \notin coNP$
- X being coNP-complete is indication for $X \notin NP$

NP versus coNP

- FACT: $P \subseteq NP \cap coNP$
- Some people think that $P \neq NP \cap coNP$
- Some people think that $P = NP \cap coNP$

- Most people think that $NP \neq coNP$

Theorem

If coNP contains some NP-complete problem X, then NP=coNP.

Hence:

- X being NP-complete is indication for $X \notin coNP$
- X being coNP-complete is indication for $X \notin NP$
- $X \in NP \cap coNP$ is indication for X not being (co)NP-complete

NP versus coNP

- FACT: $P \subseteq NP \cap coNP$
- Some people think that $P \neq NP \cap coNP$
- Some people think that $P = NP \cap coNP$

- Most people think that $NP \neq coNP$

Theorem

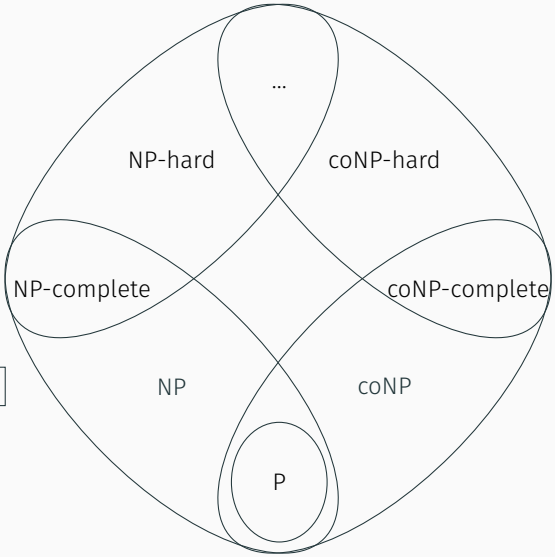
If coNP contains some NP-complete problem X, then NP=coNP.

Hence:

- X being NP-complete is indication for $X \notin coNP$
- X being coNP-complete is indication for $X \notin NP$
- $X \in NP \cap coNP$ is indication for X not being (co)NP-complete

- FACT: If $P = coNP$ then $P = NP$ (P closed under complementation)

Complexity Classes



If $P \neq NP$, $NP \neq coNP$

Questions?

Beyond NP and coNP

A short note on other complexity classes

The class NP relates to questions with polynomial proofs of $\exists x \in I : \text{YES}(x)$ for a solution x of an instance I.

The class coNP relates to questions with polynomial proofs of $\forall x \in I : \neg\text{YES}(x)$

Polynomial Hierarchy and Complexity classes $\Sigma_k P$ and $\Pi_k P$

The class $\Sigma_k P$ relates to question with polynomial proofs of properties A
$\exists x_1, \forall x_2, \exists x_3, \ldots (\forall \text{ or } \exists) x_k : A(x_1, \ldots, x_k)$

The class $\Pi_k P$ relates to question with polynomial proofs of properties A
$\forall x_1, \exists x_2, \forall x_3, \ldots (\forall \text{ or } \exists) x_k : A(x_1, \ldots, x_k)$

The polynomial hierarchy class PH is the union of all these set for all k.

NP = $\Sigma_1 P$ and coNP = $\Pi_1 P$.

If coNP = NP, this would imply that $\Sigma_k P$ = NP and $\Pi_k P$ = coNP for all $k \geq 1$ (this is not expected).

The class $\Sigma_2 P$ relates to robust programming, stochastic programming, Stackelberg games, etcetera.

Gerhard Woeginger gave an excellent keynote on this topic at EURO2018 (video link).

Complexity class PSPACE

PSPACE is the set of decision problems that can be solved in *polynomial space* on a *deterministic Turing machine*.

Some other classes

Complexity class PSPACE

PSPACE is the set of decision problems that can be solved in *polynomial space* on a *deterministic Turing machine*.

Complexity class EXPTIME and EXPSPACE

EXPTIME is the et of decision problems that can be solved in *exponential time* on a *deterministic Turing machine*.

EXPSPACE is the set of decision problems that can be solved in *exponential space* on a *deterministic Turing machine*.

Some other classes

Complexity class PSPACE

PSPACE is the set of decision problems that can be solved in *polynomial space* on a *deterministic Turing machine*.

Complexity class EXPTIME and EXPSPACE

EXPTIME is the et of decision problems that can be solved in *exponential time* on a *deterministic Turing machine*.

EXPSPACE is the set of decision problems that can be solved in *exponential space* on a *deterministic Turing machine*.

It is known that $P \subseteq NP \subseteq PH \subseteq PSPACE$

Some other classes

Complexity class PSPACE

PSPACE is the set of decision problems that can be solved in *polynomial space* on a *deterministic Turing machine*.

Complexity class EXPTIME and EXPSPACE

EXPTIME is the et of decision problems that can be solved in *exponential time* on a *deterministic Turing machine*.

EXPSPACE is the set of decision problems that can be solved in *exponential space* on a *deterministic Turing machine*.

It is known that $P \subseteq NP \subseteq PH \subseteq PSPACE$

It is known that $P \subset EXPTIME$ and $PSPACE \subset EXPSPACE$

An unsolvable decision problem

Problem: CheckTermination (also called ‘Halting Problem’)

Input: two text pieces *`text1`* and *`text2`*

An unsolvable decision problem

Problem: CheckTermination (also called 'Halting Problem')

Input: two text pieces `text1` and `text2` **Question:** does the Python program listed in `text1` terminate on the input in `text2`?

Suppose there exists an algorithm for CheckTermination

An unsolvable decision problem

Problem: CheckTermination (also called 'Halting Problem')

Input: two text pieces *text1* and *text2* **Question:** does the Python program listed in *text1* terminate on the input in *text2*?

Suppose there exists an algorithm for CheckTermination
Then there is a Python program *CT(text1,text2)* implementing this

An unsolvable decision problem

Problem: CheckTermination (also called 'Halting Problem')

Input: two text pieces `text1` and `text2` **Question:** does the Python program listed in `text1` terminate on the input in `text2`?

Suppose there exists an algorithm for CheckTermination
Then there is a Python program `CT(text1,text2)` implementing this

We construct a new Python program `wrong` that takes input `text3`

An unsolvable decision problem

Problem: CheckTermination (also called 'Halting Problem')

Input: two text pieces *text1* and *text2* **Question:** does the Python program listed in *text1* terminate on the input in *text2*?

Suppose there exists an algorithm for CheckTermination
Then there is a Python program *CT(text1,text2)* implementing this

We construct a new Python program *wrong* that takes input *text3*

- First, wrong checks whether the Python program listed in *text3* terminates on the input in *text3* using *CT(text3,text3)*

An unsolvable decision problem

Problem: CheckTermination (also called 'Halting Problem')

Input: two text pieces *text1* and *text2* **Question:** does the Python program listed in *text1* terminate on the input in *text2*?

Suppose there exists an algorithm for CheckTermination
Then there is a Python program *CT(text1,text2)* implementing this

We construct a new Python program *wrong* that takes input *text3*

- First, wrong checks whether the Python program listed in *text3* terminates on the input in *text3* using *CT(text3,text3)*
- If the answer is 'NO', wrong(text3) stops

An unsolvable decision problem

Problem: CheckTermination (also called 'Halting Problem')

Input: two text pieces *text1* and *text2* **Question:** does the Python program listed in *text1* terminate on the input in *text2*?

Suppose there exists an algorithm for CheckTermination
Then there is a Python program *CT(text1,text2)* implementing this

We construct a new Python program *wrong* that takes input *text3*

- First, wrong checks whether the Python program listed in *text3* terminates on the input in *text3* using *CT(text3,text3)*
- If the answer is 'NO', wrong(text3) stops
- If the answer is 'YES', then wrong(text3) goes into an infinite loop

An unsolvable decision problem

Problem: CheckTermination (also called 'Halting Problem')

Input: two text pieces *text1* and *text2* **Question:** does the Python program listed in *text1* terminate on the input in *text2*?

Suppose there exists an algorithm for CheckTermination
Then there is a Python program *CT(text1,text2)* implementing this

We construct a new Python program *wrong* that takes input *text3*

- First, wrong checks whether the Python program listed in *text3* terminates on the input in *text3* using *CT(text3,text3)*
- If the answer is 'NO', wrong(text3) stops
- If the answer is 'YES', then wrong(text3) goes into an infinite loop

An unsolvable decision problem

Problem: CheckTermination (also called 'Halting Problem')

Input: two text pieces *text1* and *text2* **Question:** does the Python program listed in *text1* terminate on the input in *text2*?

Suppose there exists an algorithm for CheckTermination
Then there is a Python program *CT(text1,text2)* implementing this

We construct a new Python program *wrong* that takes input *text3*

- First, wrong checks whether the Python program listed in *text3* terminates on the input in *text3* using *CT(text3,text3)*
- If the answer is 'NO', wrong(text3) stops
- If the answer is 'YES', then wrong(text3) goes into an infinite loop

What does CT(text4,text4) do if *text4* is the Python code of *wrong*???

An unsolvable decision problem

Problem: CheckTermination (also called 'Halting Problem')

Input: two text pieces *text1* and *text2* **Question:** does the Python program listed in *text1* terminate on the input in *text2*?

Suppose there exists an algorithm for CheckTermination
Then there is a Python program *CT(text1,text2)* implementing this

We construct a new Python program *wrong* that takes input *text3*

- First, wrong checks whether the Python program listed in *text3* terminates on the input in *text3* using *CT(text3,text3)*
- If the answer is 'NO', wrong(text3) stops
- If the answer is 'YES', then wrong(text3) goes into an infinite loop

What does CT(text4,text4) do if *text4* is the Python code of *wrong*???
Conclusion: There is no algorithm for CheckTermination

Technique is called *diagonalization.*

Questions?