Undecidable Problems

Algorithmically Solvable Problems

Let us assume we have a problem P.

If there is an algorithm solving the problem P then we say that the problem P is **algorithmically solvable**.

If P is a decision problem and there is an algorithm solving the problem P then we say that the problem P is **decidable (by an algorithm)**.

If we want to show that a problem P is algorithmically solvable, it is sufficient to show some algorithm solving it (and possibly show that the algorithm really solves the problem P).

Algorithmically Unsolvable Problems

A problem that is not algorithmically solvable is **algorithmically unsolvable**.

A decision problem that is not decidable is **undecidable**.

Surprisingly, there are many (exactly defined) problems, for which it was proved that they are not algorithmically solvable.

Halting Problem

Let us consider some general programming language $\mathcal{L}$.

Futhermore, let us assume that programs in language $\mathcal{L}$ run on some idealized machine where a (potentially) unbounded amount of memory is available — i.e., the allocation of memory never fails.

Example: The following problem called the **Halting problem** is undecidable:

Halting problem

Input: A source code of a $\mathcal{L}$ program P, input data x.

Question: Does the computation of P on the input x halt after some finite number of steps?

Halting Problem

Let us assume that there is a program that can decide the Halting problem.

So we could construct a subroutine H, declared as

Bool H(String code, String input)

where $H(P, x)$ returns:

- true if the program P halts on the input x,
- false if the program P does not halt on the input x.

Remark: Let us say that subroutine $H(P, x)$ returns false if P is not a syntactically correct program.

Halting Problem

Using the subroutine H we can construct a program D that performs the following steps:

- It reads its input into a variable x of type String.
- It calls the subroutine $H(x, x)$.
- If subroutine H returns true, program D jumps into an infinite loop

  loop: goto loop

  In case that H returns false, program D halts.

What does the program D do if it gets its own code as an input?

Halting Problem

If D gets its own code as an input, it either halts or not.

- If D halts then $H(D, D)$ returns true and D jumps into the infinite loop. A contradiction!
- If D does not halt then $H(D, D)$ returns false and D halts. A contradiction!

In both case we obtain a contradiction and there is no other possibility. So the assumption that H solves the Halting problem must be wrong.

Reduction between Problems

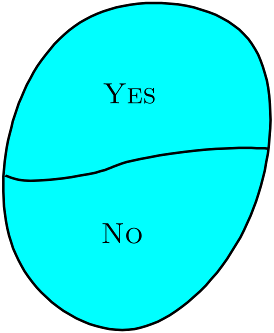
If we have already proved a (decision) problem to be undecidable, we can prove undecidability of other problems by reductions.

Problem P_1 can be **reduced** to problem P_2 if there is an algorithm Alg such that:

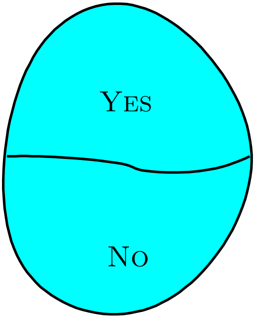
- It can get an arbitrary instance of problem P_1 as an input.
- For an instance of a problem P_1 obtained as an input (let us denote it as w) it produces an instance of a problem P_2 as an output.
- It holds i.e., the answer for the input w of problem P_1 is $\textsc{Yes}$ iff the answer for the input $Alg(w)$ of problem P_2 is $\textsc{Yes}$.

Reductions between Problems

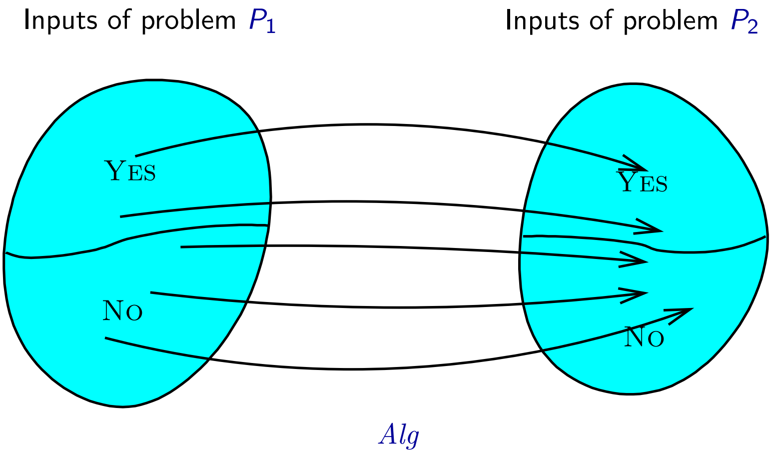
Inputs of problem P_1



Inputs of problem P_2



Reductions between Problems



Reductions between Problems

Let us say there is some reduction Alg from problem P_1 to problem P_2.

If problem P_2 is decidable then problem P_1 is also decidable.

Solution of problem P_1 for an input x:

- Call Alg with x as an input, it returns a value $Alg(x)$.
- Call the algorithm solving problem P_2 with input $Alg(x)$.
- Write the returned value to the output as the result.

It is obvious that if P_1 is undecidable then P_2 cannot be decidable.

Other Undecidable Problems

By reductions from the Halting problem we can show undecidability of many other problems dealing with a behaviour of programs:

- Is for some input the output of a given program YES?
- Does a given program halt for an arbitrary input?
- Do two given programs produce the same outputs for the same inputs?
- ...

Models of Computation

For the use in proofs and in reductions between problems, it is convenient to have the language $\mathcal{L}$ and the machine running programs in this language as simple as possible:

- the number of kinds of instructions as small as possible
- instructions as primitive as possible
- the datatypes, with which the algorithm works, as simple as possible
- it is irrelevant how difficult is to write programs in the given language (it can be extremly user-unfriently)

On the other hand, such language (resp. machine) must be general enough so that any program written in an arbitrary programming language can be compiled to it.

Models of Computation

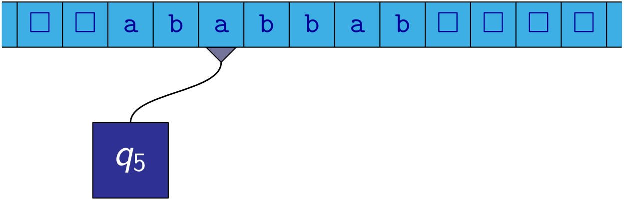
Such languages (resp. machines), which are general enough, so that programs written in any other programming language can be translated to them, are called **Turing complete**.

Examples of such Turing complete **models of computation** (languages or machines) often used in proofs:

- Turing machine (Alan Turing)
- Lambda calculus (Alonzo Church)
- Minsky machine (Marvin Minsky)
- ...

Turing machine:

- Let us extend a deterministic finite automaton in the following way:
  - the reading head can move in both directions
  - it is possible to write symbols on the tape
  - the tape is extended into infinity



Church-Turing Thesis

Church-Turing thesis

Every algorithm can be implemented as a Turing machine.

It is not a theorem that can be proved in a mathematical sense – it is not formally defined what an algorithm is.

The thesis was formulated in 1930s independently by Alan Turing and Alonzo Church.

Halting Problem

For purposes of proofs, the following version of Halting problem is often used:

Halting problem

Input: A description of a Turing machine M and a word w.

Question: Does the computation of the machine M on the word w halt after some finite number of steps?

Other Undecidable Problems

We have already seen the following example of an undecidable problem:

Problem

Input: Context-free grammars G_1 and G_2.

Question: Is $L(G_1) = L(G_2)$?

respectively

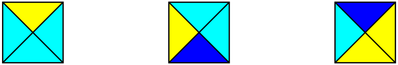
Problem

Input: A context-free grammar generating a language over an alphabet Σ.

Question: Is $L(G) = \Sigma^*$?

Other Undecidable Problems

An input is a set of types of tiles, such as:

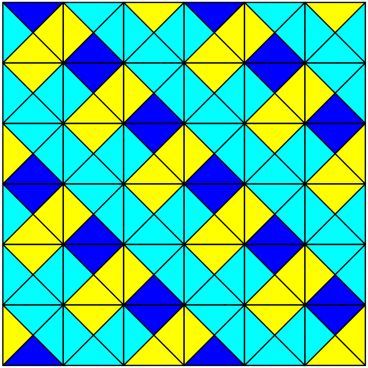


The question is whether it is possible to cover every finite area of an arbitrary size using the given types of tiles in such a way that the colors of neighboring tiles agree.

Remark: We can assume that we have an infinite number of tiles of all types.

The tiles cannot be rotated.

Other Undecidable Problems



Other Undecidable Problems

An input is a set of types of cards, such as:

abb	a	bab	baba	aba
bbab	aa	ab	aa	a

The question is whether it is possible to construct from the given types of cards a non-empty finite sequence such that the concatenations of the words in the upper row and in the lower row are the same. Every type of a card can be used repeatedly.

a	abb	abb	baba	abb	aba
aa	bbab	bbab	aa	bbab	a

In the upper and in the lower row we obtained the word `aabbabbbabaabbaba`.

Other Undecidable Problems

Undecidability of several other problems dealing with context-free grammars can be proved by reductions from the previous problem:

Problem

Input: Context-free grammars G_1 and G_2.

Question: Is $L(G_1) \cap L(G_2) = \emptyset$?

Problem

Input: A context-free grammar G.

Question: Is G ambiguous?

Other Undecidable Problems

Problem

Input: A closed formula of the first order predicate logic where the only predicate symbols are $=$ and $<$, the only function symbols are $+$ and $*$, and the only constant symbols are 0 and 1.

Question: Is the given formula true in the domain of natural numbers (using the natural interpretation of all function and predicate symbols)?

An example of an input:

$$\forall x \exists y \forall z ((x * y = z) \land (y + 1 = x))$$

Remark: There is a close connection with Gödel's incompleteness theorem.

Other Undecidable Problems

It is interesting that an analogous problem, where real numbers are considered instead of natural numbers, is decidable (but the algorithm for it and the proof of its correctness are quite nontrivial).

Also when we consider natural numbers or integers and the same formulas as in the previous case but with the restriction that it is not allowed to use the multiplication function symbol $*$, the problem is algorithmically decidable.

Other Undecidable Problems

If the function symbol $*$ can be used then even the very restricted case is undecidable:

Hilbert's tenth problem

Input: A polynomial $f(x_1, x_2, \ldots, x_n)$ constructed from variables $x_1, x_2, \ldots, x_n$ and integer constants.

Question: Are there some natural numbers $x_1, x_2, \ldots, x_n$ such that $f(x_1, x_2, \ldots, x_n) = 0$?

An example of an input: $5x^2y - 8yz + 3z^2 - 15$

I.e., the question is whether

$$\exists x \exists y \exists z (5 * x * x * y + (-8) * y * z + 3 * z * z + (-15) = 0)$$

holds in the domain of natural numbers.

Other Undecidable Problems

Also the following problem is algorithmically undecidable:

Problem

Input: A closed formula φ of the first-order predicate logic.

Question: Is $\models \varphi$?

Remark: Notation $\models \varphi$ denotes that formula φ is logically valid, i.e., it is true in all interpretations.

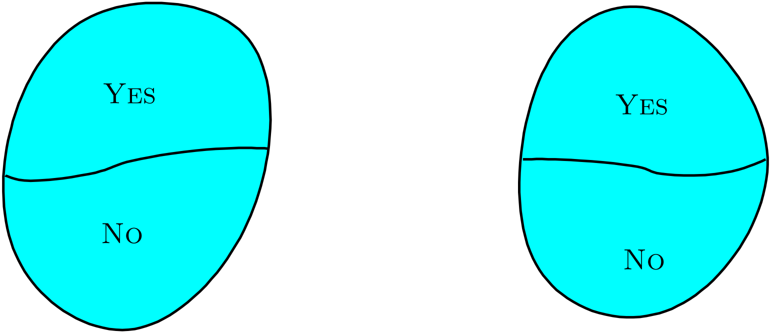
NP-Complete Problems

Polynomial Reductions between Problems

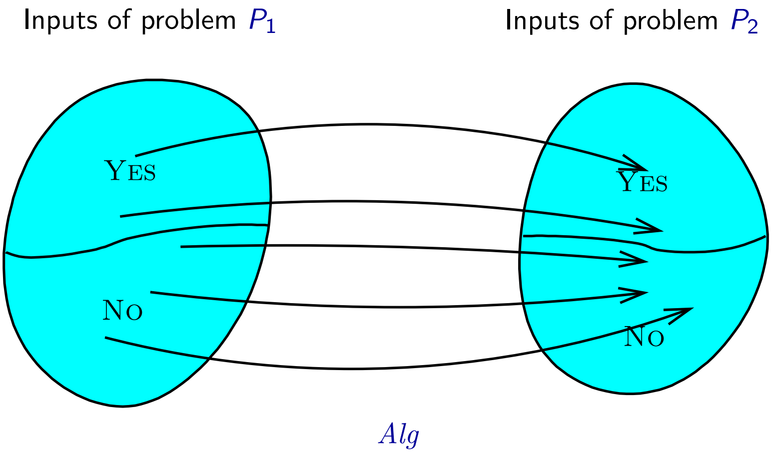
There is a **polynomial reduction** of problem P_1 to problem P_2 if there exists an algorithm Alg with a polynomial time complexity that reduces problem P_1 to problem P_2.

Polynomial Reductions between Problems

Inputs of problem P_1 Inputs of problem P_2



Polynomial Reductions between Problems



Polynomial Reductions between Problems

Let us say that problem A can be reduced in polynomial time to problem B, i.e., there is a (polynomial) algorithm P realizing this reduction.

If problem B is in the class PTIME then problem A is also in the class PTIME.

A solution of problem A for an input x:

- Call P with input x and obtain a returned value $P(x)$.
- Call a polynomial time algorithm solving problem B with the input $P(x)$.
  Write the returned value as the answer for A.

That means:

If A is not in PTIME then also B can not be in PTIME.

Polynomial Reductions between Problems

There is a big class of algorithmic problems called **NP-complete** problems such that:

- these problems can be solved by exponential time algorithms
- no polynomial time algorithm is known for any of these problems
- on the other hand, for any of these problems it is not proved that there cannot exist a polynomial time algorithm for the given problem
- every NP-complete problem can be polynomially reduced to any other NP-complete problem

Remark: This is not a definition of NP-complete problems. The precise definition will be described later.

Problem SAT

A typical example of an NP-complete problem is the SAT problem:

SAT (boolean satisfiability problem)

Input: Boolean formula φ.

Question: Is φ satisfiable?

Example:

Formula $\varphi_1 = x_1 \wedge (\neg x_2 \vee x_3)$ is satisfiable:
e.g., for valuation v where $v(x_1) = 1$, $v(x_2) = 0$, $v(x_3) = 1$, the formula φ_1 is true.

Formula $\varphi_2 = (x_1 \wedge \neg x_1) \vee (\neg x_2 \wedge x_3 \wedge x_2)$ is not satisfiable:
it is false for every valuation v.

Problem 3-SAT

3-SAT is a variant of the SAT problem where the possible inputs are restricted to formulas of a certain special form:

3-SAT

Input: Formula φ is a conjunctive normal form where every clause contains exactly 3 literals.

Question: Is φ satisfiable?

Problem 3-SAT

Recalling some notions:

- A **literal** is a formula of the form x or $\neg x$ where x is an atomic proposition.
- A **clause** is a disjuction of literals.

  Examples: $x_1 \vee \neg x_2 \qquad \neg x_5 \vee x_8 \vee \neg x_{15} \vee \neg x_{23} \qquad x_6$
- A formula is in a **conjuctive normal form (CNF)** if it is a conjuction of clauses.

  Example: $(x_1 \vee \neg x_2) \ \wedge \ (\neg x_5 \vee x_8 \vee \neg x_{15} \vee \neg x_{23}) \ \wedge \ x_6$

So in the 3-SAT problem we require that a formula φ is in a CNF and moreover that every clause of φ contains exactly three literals.

Example:

$(x_1 \vee \neg x_2 \vee x_4) \wedge (\neg x_1 \vee x_3 \vee x_3) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4) \wedge (x_2 \vee \neg x_3 \vee x_4)$

Problem 3-SAT

The following formula is satisfiable:

$(x_1 \vee \neg x_2 \vee x_4) \wedge (\neg x_1 \vee x_3 \vee x_3) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4) \wedge (x_2 \vee \neg x_3 \vee x_4)$

It is true for example for valuation v where

$$\begin{aligned} v(x_1) &= 0 \\ v(x_2) &= 1 \\ v(x_3) &= 0 \\ v(x_4) &= 1 \end{aligned}$$

On the other hand, the following formula is not satisfiable:

$$(x_1 \vee x_1 \vee x_1) \wedge (\neg x_1 \vee \neg x_1 \vee \neg x_1)$$

Polynomial Reductions between Problems

As an example, a polynomial time reduction from the 3-SAT problem to the independent set problem (IS) will be described.

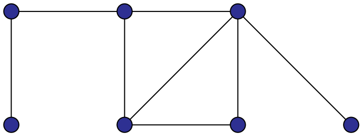
Remark: Both 3-SAT and IS are examples of NP-complete problems.

Independent Set (IS) Problem

Independent set (IS) problem

Input: An undirected graph G, a number k.

Question: Is there an independent set of size k in the graph G?



$k = 4$

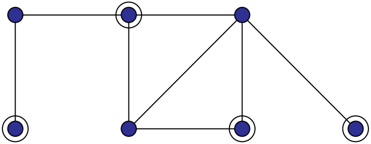
Remark: An **independent set** in a graph is a subset of nodes of the graph such that no pair of nodes from this set is connected by an edge.

Independent Set (IS) Problem

Independent set (IS) problem

Input: An undirected graph G, a number k.

Question: Is there an independent set of size k in the graph G?

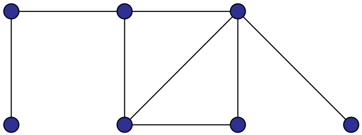


$k = 4$

Remark: An **independent set** in a graph is a subset of nodes of the graph such that no pair of nodes from this set is connected by an edge.

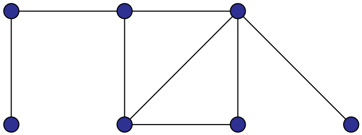
Independent Set (IS) Problem

An example of an instance where the answer is YES:



$k = 4$

An example of an instance where the answer is NO:



$k = 5$

A Reduction from 3-SAT to IS

We describe a (polynomial-time) algorithm with the following properties:

- **Input:** An arbitrary instance of 3-SAT, i.e., a formula φ in a conjunctive normal form where every clause contains exactly three literals.
- **Output:** An instance of IS, i.e., an undirected graph G and a number k.
- Moreover, the following will be ensured for an arbitrary input (i.e., for an arbitrary formula φ in the above mentioned form):

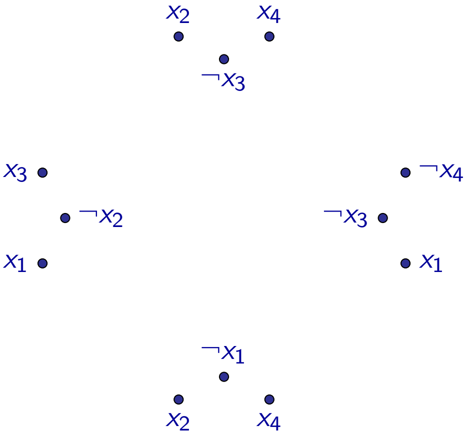
  There will be an independent set of size k in graph G iff formula φ will be satisfiable.

A Reduction from 3-SAT to IS

$$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee \neg x_3 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee x_4)$$

A Reduction from 3-SAT to IS

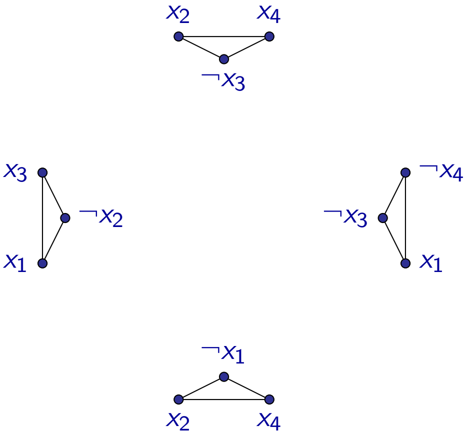
$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee \neg x_3 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee x_4)$



For each occurrence of a literal we add a node to the graph.

A Reduction from 3-SAT to IS

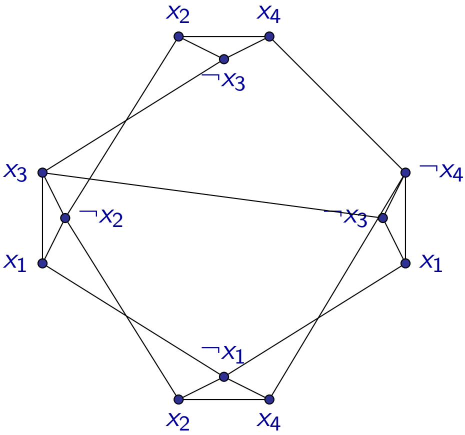
$(x_1 \vee \neg x_2 \vee x_3) \;\wedge\; (x_2 \vee \neg x_3 \vee x_4) \wedge\; (x_1 \vee \neg x_3 \vee \neg x_4) \;\wedge\; (\neg x_1 \vee x_2 \vee x_4)$



We connect with edges the nodes corresponding to occurrences of literals belonging to the same clause.

A Reduction from 3-SAT to IS

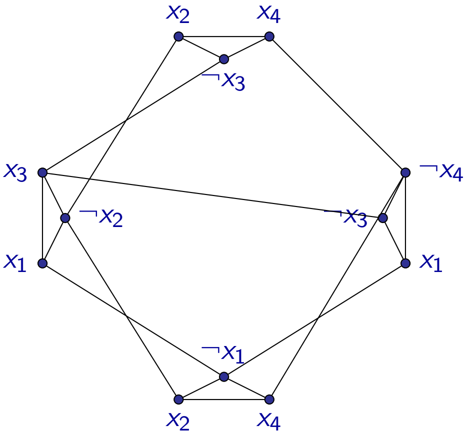
$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee \neg x_3 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee x_4)$



For each pair of nodes corresponding to literals x_i and $\neg x_i$ we add an edge between them.

A Reduction from 3-SAT to IS

$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee \neg x_3 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee x_4)$

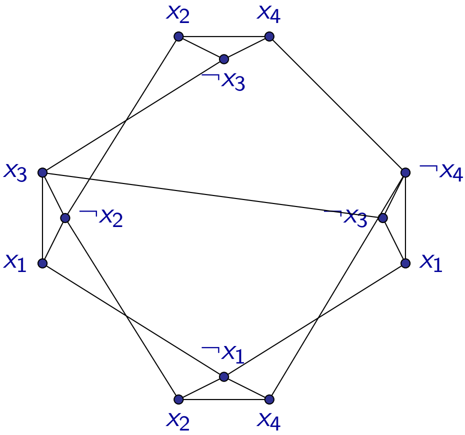


$k = 4$

We put k to be equal to the number of clauses.

A Reduction from 3-SAT to IS

$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee \neg x_3 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee x_4)$



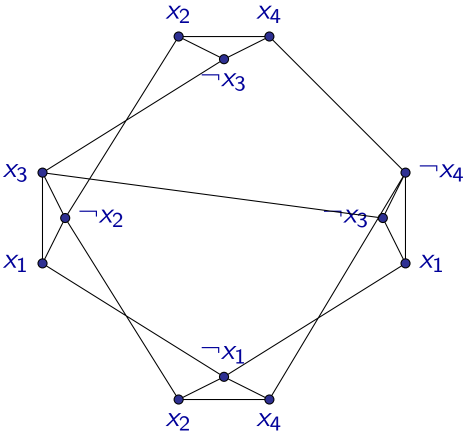
$k = 4$

The constructed graph and number k are the output of the algorithm.

A Reduction from 3-SAT to IS

$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee \neg x_3 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee x_4)$

$v(x_1) = 1$
$v(x_2) = 1$
$v(x_3) = 0$
$v(x_4) = 1$



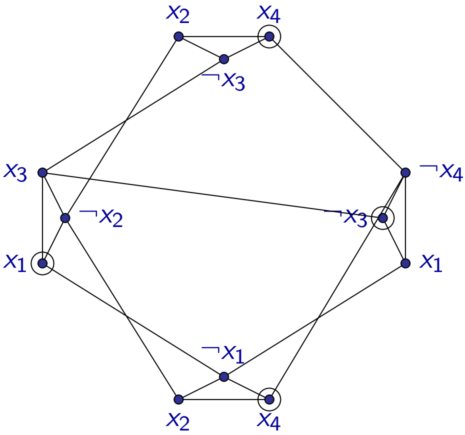
$k = 4$

If the formula φ is satisfiable then there is a valuation v where every clause contains at least one literal with value 1.

A Reduction from 3-SAT to IS

$(x_1 \vee \neg x_2 \vee x_3) \ \wedge \ (x_2 \vee \neg x_3 \vee x_4) \ \wedge \ (x_1 \vee \neg x_3 \vee \neg x_4) \ \wedge \ (\neg x_1 \vee x_2 \vee x_4)$

$v(x_1) = 1$
$v(x_2) = 1$
$v(x_3) = 0$
$v(x_4) = 1$



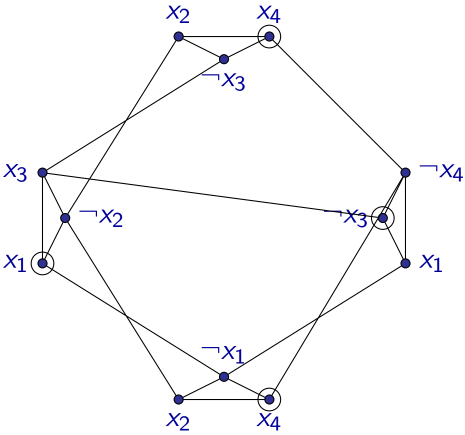
$k = 4$

We select one literal that has a value 1 in the valuation v, and we put the corresponding node into the independent set.

A Reduction from 3-SAT to IS

$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee \neg x_3 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee x_4)$

$v(x_1) = 1$
$v(x_2) = 1$
$v(x_3) = 0$
$v(x_4) = 1$



$k = 4$

We can easily verify that the selected nodes form an independent set.

A Reduction from 3-SAT to IS

The selected nodes form an independent set because:

- One node has been selected from each triple of nodes corresponding to one clause.
- Nodes denoted x_i and $\neg x_i$ could not be selected together.
  (Exactly of them has the value 1 in the given valuation v.)

A Reduction from 3-SAT to IS

On the other hand, if there is an independent set of size k in graph G, then it surely has the following properties:

- At most one node is selected from each triple of nodes corresponding to one clause.

  But because there are k clauses and k nodes are selected, exactly one node must be selected from each triple.

- Nodes denoted x_i and $\neg x_i$ cannot be selected together.

We can choose a valuation according to the selected nodes, since it follows from the previous discussion that it must exist.
(Arbitrary values can be assigned to the remaining variables.)

For the given valuation, the formula φ has surely the value 1, since in each clause there is at least one literal with value 1.

A Reduction from 3-SAT to IS

It is obvious that the running time of the described algorithm polynomial:

Graph G and number k can be constructed for a formula φ in time $O(n^2)$, where n is the size of formula φ.

We have also seen that there is an independent set of size k in the constructed graph G iff the formula φ is satisfiable.

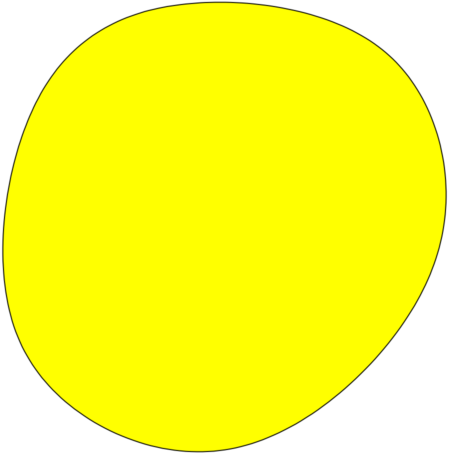
The described algorithm shows that 3-SAT can be reduced in polynomial time to IS.

Complexity Classes

- PTIME — the class of all algorithmic problems that can solve by a (deterministic) algorithm in polynomial time
- NPTIME — the class of algorithmic problems that can be solved by a **nondetermistic** algorithm in polynomial time

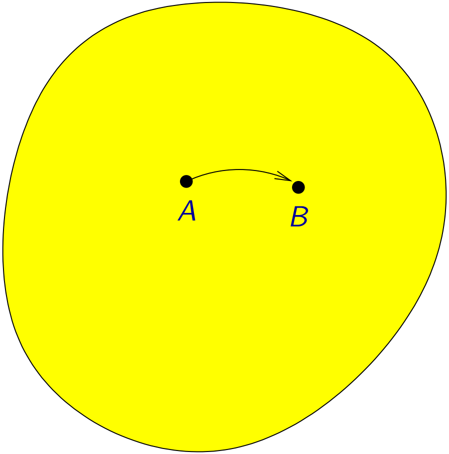
NP-Complete Problems

Let us consider a set of all decision problems.



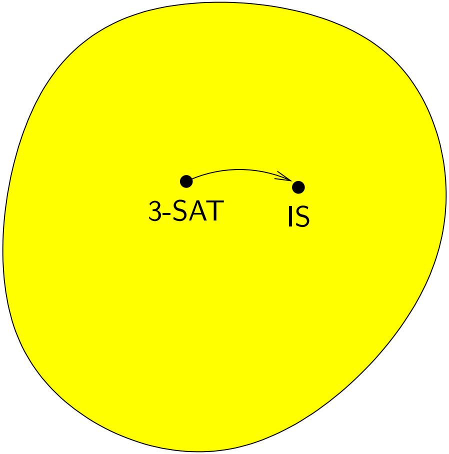
NP-Complete Problems

By an arrow we denote that a problem A can be reduced in polynomial time to a problem B.



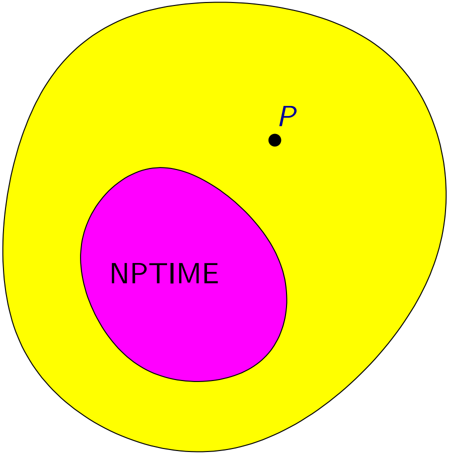
NP-Complete Problems

For example 3-SAT can be reduced in polynomial time to IS.



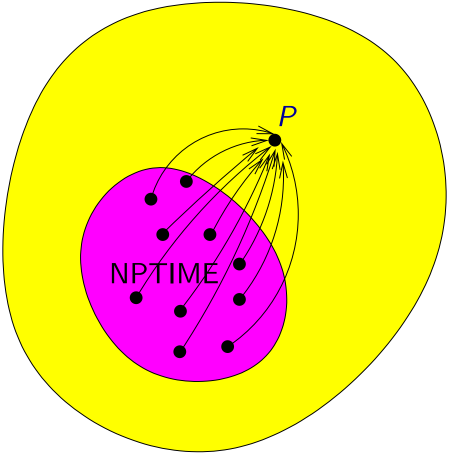
NP-Complete Problems

Let us consider now the class NPTIME and a problem P.



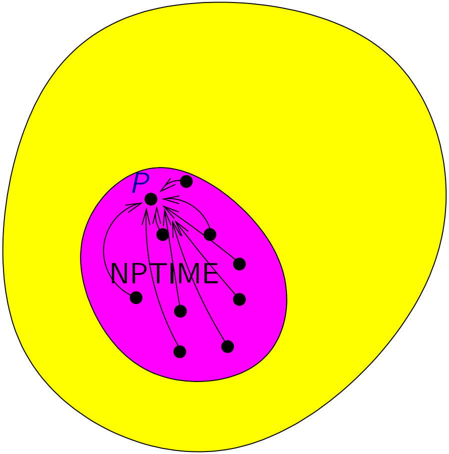
NP-Complete Problems

A problem P is NP-**hard** if every problem from NPTIME can be reduced in polynomial time to P.



NP-Complete Problems

A problem P is **NP-complete** if it is NP-hard and it belongs to the class NPTIME.



NP-Complete Problems

If we have found a polynomial time algorithm for some NP-hard problem P, then we would have polynomial time algorithms for all problems P' from NPTIME:

- At first we would apply an algorithm for the reduction from P' to P on an input of a problem P'.
- Then we would use a polynomial algorithm for P on the constructed instance of P and returned its result as the answer for the original instance of P'.

Is such case, PTIME = NPTIME would hold, since for every problem from NPTIME there would be a polynomial-time (deterministic) algorithm.

NP-Complete Problems

On the other hand, if there is at least one problem from NPTIME for which a polynomial-time algorithm does not exist, then it means that for none of NP-hard problems there is a polynomial-time algorithm.

It is an open question whether the first or the second possibility holds.

NP-Complete Problems

It is not difficult to see that:

If a problem A can be reduced in a polynomial time to a problem B and problem B can be reduced in a polynomial time to a problem C, then problem A can be reduced in a polynomial time to problem C.

So if we know about some problem P that it is NP-hard and that P can be reduced in a polynomial time to a problem P', then we know that the problem P' is also NP-hard.

NP-Complete Problems

Theorem

Problem SAT is NP-complete.

It can be shown that SAT can be reduced in a polynomial time to 3-SAT and we have seen that 3-SAT can be reduced in a polynomial time to IS.

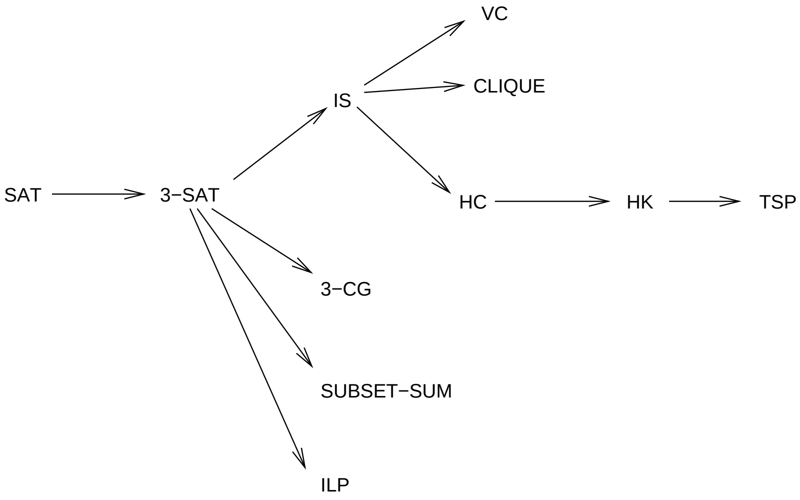
This means that problems 3-SAT and IS are NP-hard.

It is not difficult to show that 3-SAT and IS belong to the class NPTIME.

Problems 3-SAT and IS are NP-complete.

NP-Complete Problems

By a polynomial reductions from problems that are already known to be NP-complete, NP-completeness of many other problems can be shown:



Examples of Some NP-Complete Problems

The following previously mentioned problems are NP-complete:

- SAT (boolean satisfiability problem)
- 3-SAT
- IS — independent set problem

On the following slides, examples of some other NP-complete problems are described:

- CG — graph coloring (remark: it is NP-complete even in the special case where we have 3 colors)
- VC — vertex cover
- CLIQUE — clique problem
- HC — Hamiltonian cycle
- HK — Hamiltonian circuit
- TSP — traveling salesman problem
- SUBSET-SUM
- ILP — integer linear programming

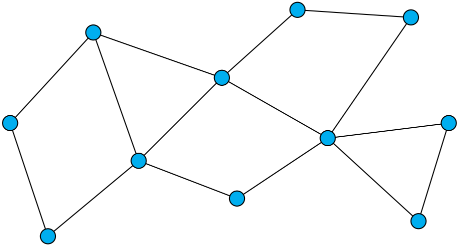
Graph Coloring

Graph coloring

Input: An undirected graph G, a natural number k.

Question: Is it possible to color nodes of the graph G using k colors in such a way that there is no pair of nodes where both nodes are colored with the same color and connected with an edge?

Example: $k = 3$



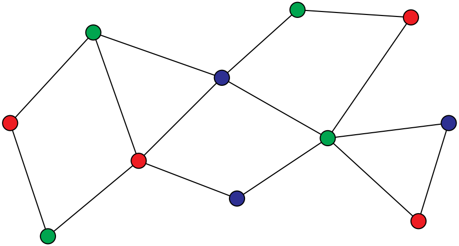
Graph Coloring

Graph coloring

Input: An undirected graph G, a natural number k.

Question: Is it possible to color nodes of the graph G using k colors in such a way that there is no pair of nodes where both nodes are colored with the same color and connected with an edge?

Example: $k = 3$



Answer: YES

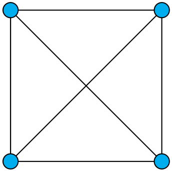
Graph Coloring

Graph coloring

Input: An undirected graph G, a natural number k.

Question: Is it possible to color nodes of the graph G using k colors in such a way that there is no pair of nodes where both nodes are colored with the same color and connected with an edge?

Example: $k = 3$



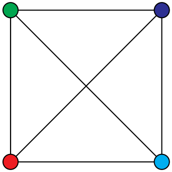
Graph Coloring

Graph coloring

Input: An undirected graph G, a natural number k.

Question: Is it possible to color nodes of the graph G using k colors in such a way that there is no pair of nodes where both nodes are colored with the same color and connected with an edge?

Example: $k = 3$



Answer: No

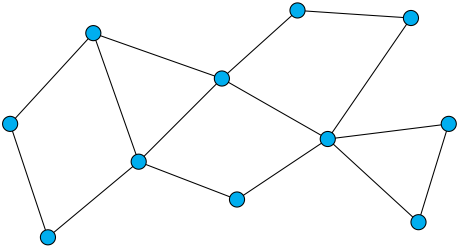
VC – Vertex Cover

VC – vertex cover

Input: An undirected graph G and a natural number k.

Question: Is there some subset of nodes of G of size k such that every edge has at least one of its nodes in this subset?

Example: $k = 6$



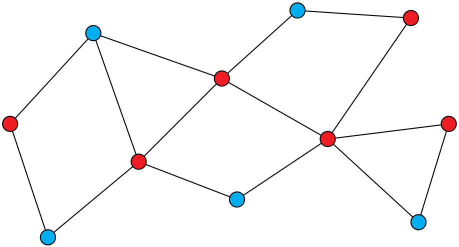
VC – Vertex Cover

VC – vertex cover

Input: An undirected graph G and a natural number k.

Question: Is there some subset of nodes of G of size k such that every edge has at least one of its nodes in this subset?

Example: $k = 6$



Answer: YES

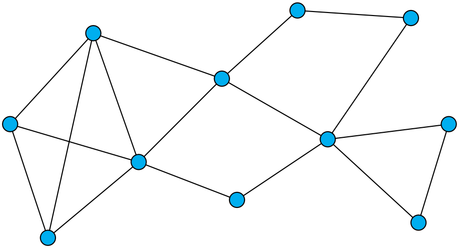
CLIQUE

CLIQUE

Input: An undirected graph G and a natural number k.

Question: Is there some subset of nodes of G of size k such that every two nodes from this subset are connected by an edge?

Example: $k = 4$



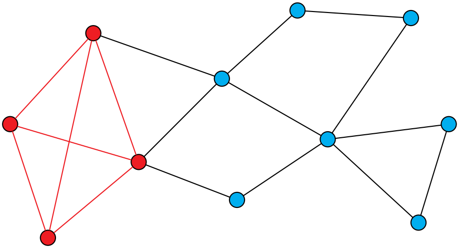
CLIQUE

CLIQUE

Input: An undirected graph G and a natural number k.

Question: Is there some subset of nodes of G of size k such that every two nodes from this subset are connected by an edge?

Example: $k = 4$



Answer: YES

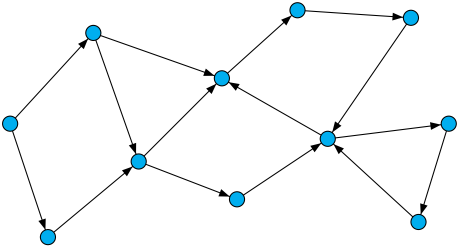
Hamiltonian Cycle

HC – Hamiltonian cycle

Input: A directed graph G.

Question: Is there a Hamiltonian cycle in G (i.e., a directed cycle going through each node exactly once)?

Example:



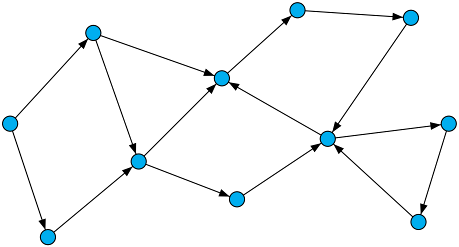
Hamiltonian Cycle

HC – Hamiltonian cycle

Input: A directed graph G.

Question: Is there a Hamiltonian cycle in G (i.e., a directed cycle going through each node exactly once)?

Example:



Answer: NO

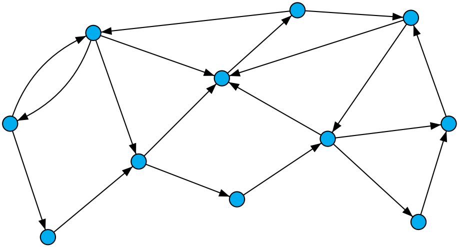
Hamiltonian Cycle

HC – Hamiltonian cycle

Input: A directed graph G.

Question: Is there a Hamiltonian cycle in G (i.e., a directed cycle going through each node exactly once)?

Example:



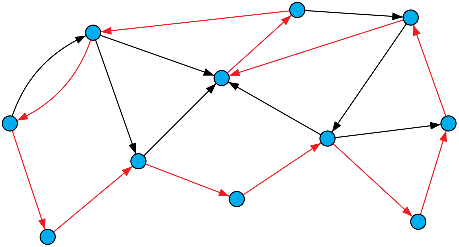
Hamiltonian Cycle

HC – Hamiltonian cycle

Input: A directed graph G.

Question: Is there a Hamiltonian cycle in G (i.e., a directed cycle going through each node exactly once)?

Example:



Answer: YES

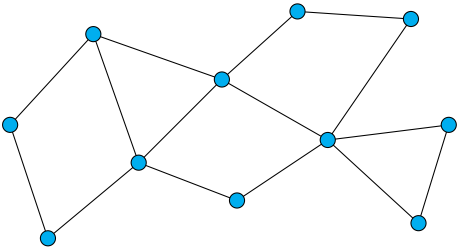
Hamiltonian Circuit

HK – Hamiltonian circuit

Input: An undirected graph G.

Question: Is there a Hamiltonian circuit in G (i.e., an undirected cycle going through each node exactly once)?

Example:



Answer: NO

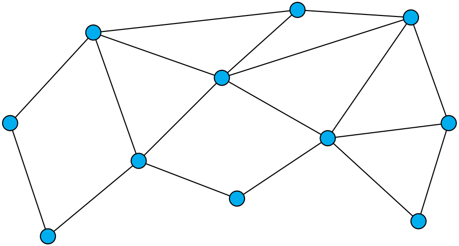
Hamiltonian Circuit

HK – Hamiltonian circuit

Input: An undirected graph G.

Question: Is there a Hamiltonian circuit in G (i.e., an undirected cycle going through each node exactly once)?

Example:



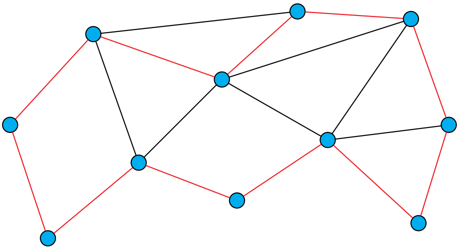
Hamiltonian Circuit

HK – Hamiltonian circuit

Input: An undirected graph G.

Question: Is there a Hamiltonian circuit in G (i.e., an undirected cycle going through each node exactly once)?

Example:



Answer: YES

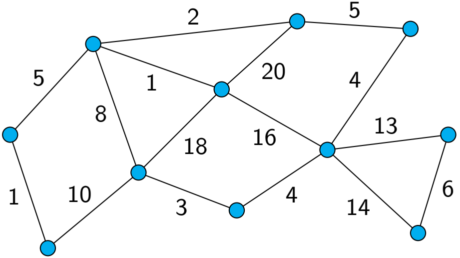
Traveling Salesman Problem

TSP - traveling salesman problem

Input: An undirected graph G with edges labelled with natural numbers and a number k.

Question: Is there a closed tour going through all nodes of the graph G such that the sum of labels of edges on this tour is at most k?

Example: $k = 70$



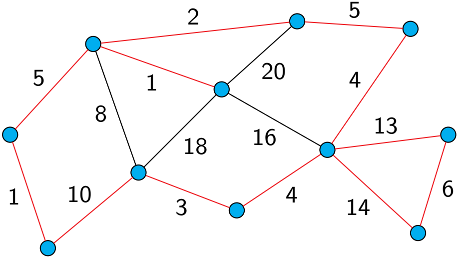
Traveling Salesman Problem

TSP - traveling salesman problem

Input: An undirected graph G with edges labelled with natural numbers and a number k.

Question: Is there a closed tour going through all nodes of the graph G such that the sum of labels of edges on this tour is at most k?

Example: $k = 70$



Answer: YES, since there is a tour with the sum 69.

SUBSET-SUM

Problem SUBSET-SUM

Input: A sequence $a_1, a_2, \ldots, a_n$ of natural numbers and a natural number s.

Question: Is there a set $I \subseteq \{1, 2, \ldots, n\}$ such that $\sum_{i \in I} a_i = s$?

In other words, the question is whether it is possible to select a subset with sum s of a given (multi)set of numbers.

Example: For the input consisting of numbers $3, 5, 2, 3, 7$ and number $s = 15$ the answer is $\textsc{Yes}$, since $3 + 5 + 7 = 15$.

For the input consisting of numbers $3, 5, 2, 3, 7$ and number $s = 16$ the answer is $\textsc{No}$, since no subset of these numbers has sum 16.

SUBSET-SUM

Remark:
The order of numbers $a_1, a_2, \ldots, a_n$ in an input is not important.

Note that this is not exactly the same as if we have formulated the problem so that the input is a set $\{a_1, a_2, \ldots, a_n\}$ and a number s — numbers cannot occur multiple times in a set but they can in a sequence.

SUBSET-SUM

Problem SUBSET-SUM is a special case of a **knapsack problem**:

Knapsack problem

Input: Sequence of pairs of natural numbers $(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)$ and two natural numbers s and t.

Question: Is there a set $I \subseteq \{1, 2, \ldots, n\}$ such that $\sum_{i \in I} a_i \leq s$ and $\sum_{i \in I} b_i \geq t$?

SUBSET-SUM

Informally, the knapsack problem can be formulated as follows:

We have n objects, where the i-th object weights a_i grams and its price is b_i dollars.

The question is whether there is a subset of these objects with total weight at most s grams (s is the capacity of the knapsack) and with total price at least t dollars.

Remark:

Here we have formulated this problem as a decision problem.

This problem is usually formulated as an optimization problem where the aim is to find such a set $I \subseteq \{1, 2, \ldots, n\}$, where the value $\sum_{i \in I} b_i$ is maximal and where the condition $\sum_{i \in I} a_i \leq s$ is satisfied, i.e., where the capacity of the knapsack is not exceeded.

SUBSET-SUM

That SUBSET-SUM is a special case of the Knapsack problem can be seen from the following simple construction:

Let us say that $a_1, a_2, \ldots, a_n$, s_1 is an instance of SUBSET-SUM.
It is obvious that for the instance of the knapsack problem where we have the sequence (a_1, a_1), $(a_2, a_2), \ldots,$ (a_n, a_n), $s = s_1$ and $t = s_1$, the answer is the same as for the original instance of SUBSET-SUM.

SUBSET-SUM

If we want to study the complexity of problems such as SUBSET-SUM or the knapsack problem, we must clarify what we consider as the size of an instance.

Probably the most natural it is to define the size of an instance as the total number of bits needed for its representation.

We must specify how natural numbers in the input are represented – if in binary (resp. in some other numeral system with a base at least 2 (e.g., decimal or hexadecimal) or in unary.

- If we consider the total number of bits when numbers are written in **binary** as the size of an input, no polynomial time algorithm is known for SUBSET-SUM.
- If we consider the total number of bits when numbers are written in **unary** as the size of an input, SUBSET-SUM can be solved by an algorithm whose time complexity is polynomial.

ILP – Integer Linear Programming

Problem ILP (integer linear programming)

Input: An integer matrix A and an integer vector b.

Question: Is there an integer vector x such that $Ax \leq b$?

An example of an instance of the problem:

$$A = \begin{pmatrix} 3 & -2 & 5 \\ 1 & 0 & 1 \\ 2 & 1 & 0 \end{pmatrix} \qquad b = \begin{pmatrix} 8 \\ -3 \\ 5 \end{pmatrix}$$

So the question is if the following system of inequations has some integer solution:

$$\begin{aligned} 3x_1 - 2x_2 + 5x_3 &\leq 8 \\ x_1 + x_3 &\leq -3 \\ 2x_1 + x_2 &\leq 5 \end{aligned}$$

ILP – Integer Linear Programming

One of solutions of the system

$$\begin{array}{rcl} 3x_1 - 2x_2 + 5x_3 & \leq & 8 \\ x_1 + x_3 & \leq & -3 \\ 2x_1 + x_2 & \leq & 5 \end{array}$$

is for example $x_1 = -4$, $x_2 = 1$, $x_3 = 1$, i.e.,

$$x = \begin{pmatrix} -4 \\ 1 \\ 1 \end{pmatrix}$$

because

$$\begin{array}{rcrcr} 3 \cdot (-4) - 2 \cdot 1 + 5 \cdot 1 & = & -9 & \leq & 8 \\ -4 + 1 & = & -3 & \leq & -3 \\ 2 \cdot (-4) + 1 & = & -7 & \leq & 5 \end{array}$$

So the answer for this instance is $\textsc{Yes}$.

ILP – Integer Linear Programming

Remark: A similar problem where the question for a given system of linear inequations is whether it has a solution in the set of **real** numbers, can be solved in a polynomial time.