Formale Methoden der Informatik

Block 1: Foundations of Complexity Theory

1. Computation and Computability

Reinhard Pichler

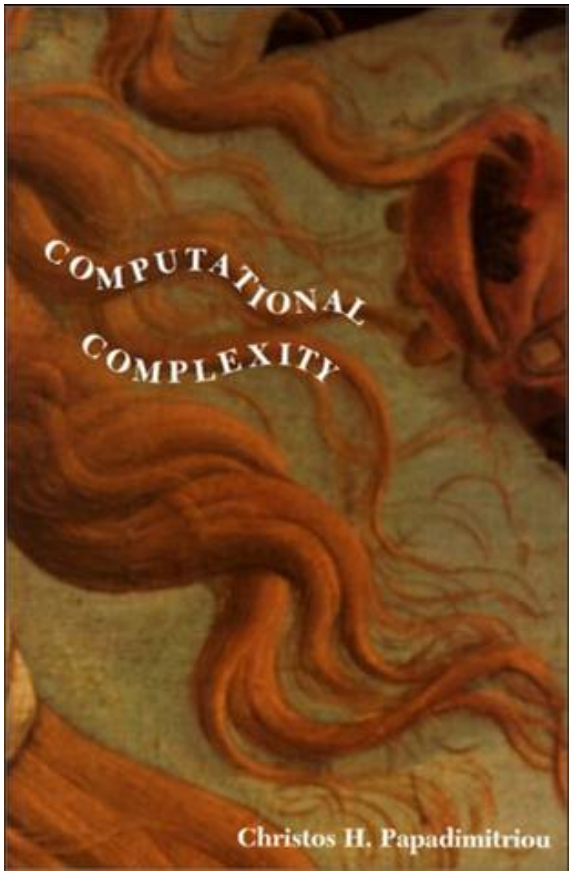
Institut für Informationssysteme
Arbeitsbereich DBAI
Technische Universität Wien

7 October, 2013

Outline

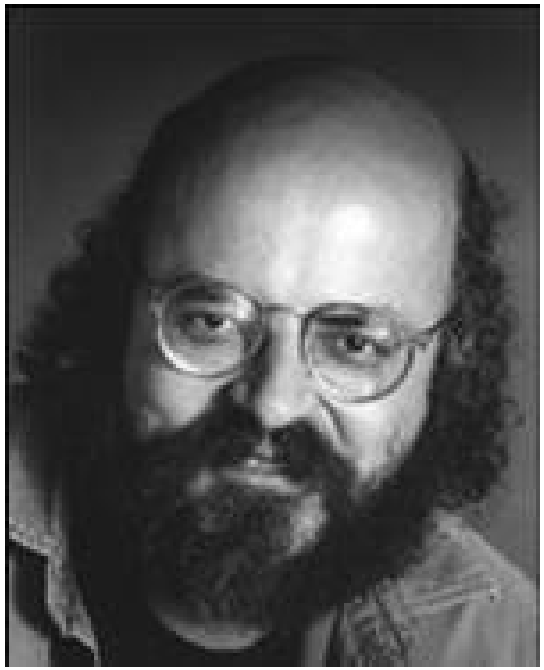
Background Reading

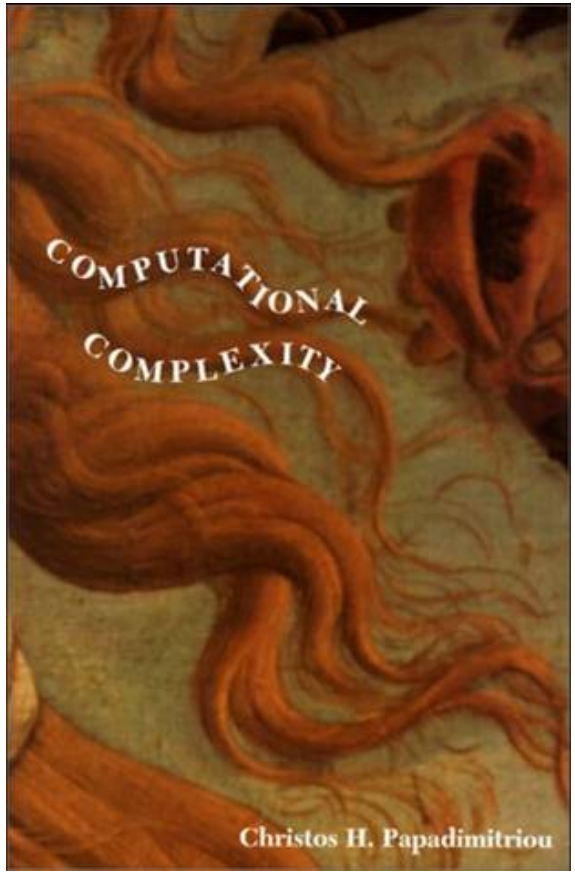
- Christos H. Papadimitriou: Computational Complexity. Addison Wesley, 1994.



Background Reading

- Christos H. Papadimitriou: Computational Complexity. Addison Wesley, 1994.





Structure of Block 1

Topics	Papadimitriou's book
1 Computation and computability	Sect. 2 & 3
2 Complexity of problems and algorithms	Sect. 1 & 4
3 Reductions	Sect. 8
4 NP-completeness	Sect. 9
5 Other important complexity classes	Sect. 16 & 19 (small parts)
6 Turing machines	Sect. 2

Caveat. Usually, textbooks on computability and complexity introduce Turing machines first and define most concepts in terms of Turing machines. Due to rather bad experience with this approach in previous years of this lecture, we provide most definitions in terms of a simple programming language. Turing machines are only introduced at the end.

Related Lectures

- Complexity Theory
  181.142 – 2.0 VU – Komplexitätstheorie
  Reinhard Pichler
  (in the summer term)
- Complexity Analysis
  184.215 – 2.0 VU – Komplexitätsanalyse
  Thomas Eiter
  (in the summer term)
- Database Theory
  181.140 – 2.0 VU – Datenbanktheorie
  Reinhard Pichler
  (in the summer term)

Problems

Computability theory focuses on analyzing the existence of algorithms to solve problems.

Complexity theory focuses on analyzing how hard it is to solve a problem in terms of computation time and required space.

Definition of Problems

A problem is a question together with an infinite set of possible instances (i.e. possible inputs).

A problem is a decision problem if the question has a yes/no answer.

Example Problem

REACHABILITY:
INSTANCE: A graph (V, E) and nodes $u, v \in V$.
QUESTION: Is there a path in the graph from u to v?

REACHABILITY is a decision problem.

Some other Types of Problems

Different questions than those with yes/no answer

- function problems: compute some output function $f(x)$ (depending on the instance x), e.g.: compute a path with some specific property
- optimization problems: compute the optimal value ($=$ min/max) of some function $f(x)$ (depending on the instance x), e.g.: length of the shortest path
- enumeration problems: compute all possible solutions to a problem instance, e.g.: all (cycle-free) paths from u to v
- counting problems: compute the number of possible solutions to a problem instance, e.g.: How many (cycle-free) different paths from u to v exist?

Algorithms

Definition of an Algorithm

An algorithm for a problem $\mathcal{P}$ is a description of computation steps that allow to solve any given instance of the problem $\mathcal{P}$.

- The definition is a bit vague...
  - What is a "description"?
  - What is a "computation step"?
- The answer is "it depends", however we require the following:
  1. the description must be understandable/sharable by humans
  2. an algorithm performs finitely many computation steps
  3. works on all instances of the problem: for each possible instance of the problem, the execution of the computation steps results in the correct answer to the question.
  4. each step is "simple", i.e. can be performed by a machine

Programs

Programming Languages

Programming languages allow us to write programs, which are formal descriptions of computation steps.

- In imperative languages (like C or JAVA) the computation steps are given explicitly ("assign", "allocate", "go to", "output")
- In declarative languages (functional like HASKELL, logical like PROLOG) the computation steps are implicit:
  - an interpreter takes a program with its input and performs computation steps to produce an output
- Programs are understandable/sharable by humans and can be executed by a machine.

Algorithms vs. Programs

Not all programs are legitimate algorithms!

- Why? A program may not produce an answer for all allowed instances (e.g., it may enter an infinite loop, raise an exception, etc.)

The Question in Computability Theory

Given a problem $\mathcal{P}$, can we write a program that is an algorithm for $\mathcal{P}$?

It turns out that:

- Some problems have algorithms. (obvious)
- For some problems no algorithm exists! (maybe not so obvious)

Our Programming Language

- To talk about programs we need to fix a programming language
- We use the very core of procedural languages (call it SIMPLE):
  - variables of different types (String, Integer, Boolean, Void)
  - variable assignments (e.g. $x := y + z$)
  - "if/then/else" statements
  - "while" loops
  - "for" loops, "repeat" loops (can be simulated using "while" loops)
  - "return" statement
- A program may take some input:
  - a list $L = (V_1, V_2, \ldots, V_n)$ of values of different types;
  - alternatively, takes a single string I as an input:
    - (any list L can be coded into a string, just think of XML)
- A program returns a value of some type using the "return" statement.

Our Programming Language (Example)

Example

String multiply(String s, Integer n)
    $result := s$
    while $n > 1$ **do** $\{$ $result := result + s$; $n := n - 1$; $\}$
    return $result$;

- concatenates n copies of the string s
- input: one string, one integer
  - again, both arguments can be encoded into a single string!
- output: a string

Suitability of Our Programming Language

- Is our language adequate? That is:
  - Are there problems not solvable in SIMPLE, but solvable in JAVA, C?
  - Is SIMPLE powerful enough to express all algorithms?
- It is adequate! All problems that can be solved in JAVA, C, or any other known language can also be solved in SIMPLE.
- Because in SIMPLE one can implement:
  - e.g. a Java Virtual Machine (JVM) to run JAVA programs,
  - in general, an interpreter for all known programming languages.
- Thus, if we can prove that an algorithm cannot be implemented in SIMPLE, it cannot be implemented in any other programming language either.

Church-Turing Thesis

Any algorithm can be programmed in SIMPLE.

Decidability of a Problem

Decidability

A decision problem $\mathcal{P}$ is called decidable if there exists an algorithm for $\mathcal{P}$. Otherwise, if there doesn't exist an algorithm for $\mathcal{P}$, then $\mathcal{P}$ is called undecidable.

By the Church-Turing Thesis we have:

Theorem

A decision problem is decidable if and only if there exists a SIMPLE program for it.

Many real world problems are decidable:

- Given a database and an SQL query, check if the output is empty.
- Given a regular expression E and a string S, check if S matches E.
- Given a natural number n, check if n is a prime number.

More Complex Programs

- There are many real world problems for which algorithms are not obvious!
- Many of such problems are related to questions about behavior of programs.
  - Given a program Π and its input I, does the program enter an infinite loop?
  - Given a program Π, does it terminate on all inputs?
- Algorithms for such problems would be great for ensuring correctness of programs.
- Moreover, many mathematical problems could be solved using an algorithm for termination.
  - We consider Goldbach's Conjecture, which has been an open problem in mathematics since 1742.

Goldbach's Conjecture

Every even integer greater than 2 is the sum of two primes.

We can write a program that checks the conjecture for $4, 6, 8, 10, 12, \ldots$

```
Boolean test(Integer n)    /* checks if n is the sum of two primes */
    for all i ≤ n, j ≤ n do {
        if i is prime, j is prime, and i + j = n then return true; }
    return false;

Void testConjecture()
    n:=4;
    while test(n)=true do { n := n + 2 }
```

Theorem

Goldbach's Conjecture is true iff testConjecture() does not terminate.

Goldbach's Conjecture (Cnt'd)

- Suppose we have a program Π, which decides whether testConjecture() terminates.
- By running Π on a computer we can determine the validity of GC
  - If Π outputs "no", then the conjecture is true.
  - If Π outputs "yes", then the conjecture is disproven, i.e. there exists even $n > 2$ that is not the sum of two primes.
- We don't know how long it will take to execute Π, but since Π is an algorithm, we are guaranteed the resolution of GC.
- Numerous other mathematical problems could be solved using an algorithm for termination (consider Fermat's Theorem).

We next show that it is impossible to write a program that decides if another program terminates.

Proving Undecidability of the Halting Problem

HALTING PROBLEM

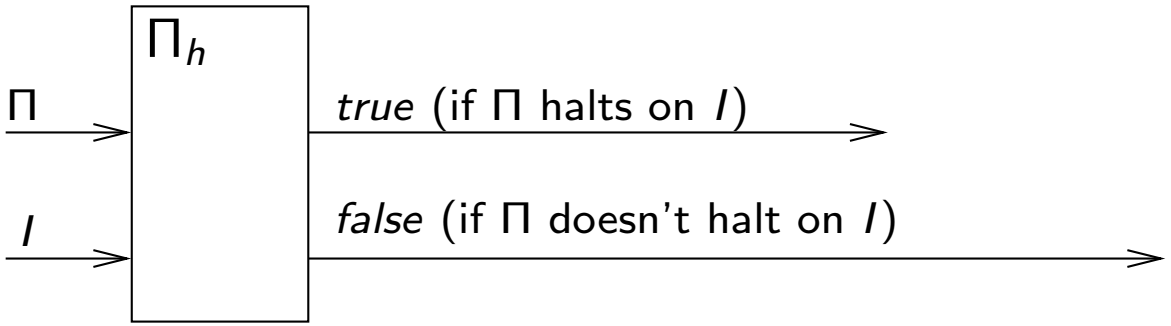
INSTANCE: A source code S (of a SIMPLE program), an input string I.
QUESTION: Does the program S terminate on input I?

- We show that there is no program Π that would decide the Halting problem.
- The proof will be by contradiction, i.e. we assume that the Halting problem is decidable, and then show that the assumption leads to a contradiction.
- We assume that the input to a program is a string.
- The source code of a program is also a string.

Proving Undecidability of the Halting Problem (Step 1)

Assume there is a program Π_h such that:

- Π_h takes two strings as input:
  - Π (the source code of a program)
  - I (an input for the program Π)
- Π_h outputs:
  - *true* if Π terminates on I
  - *false* if Π does not terminate on I

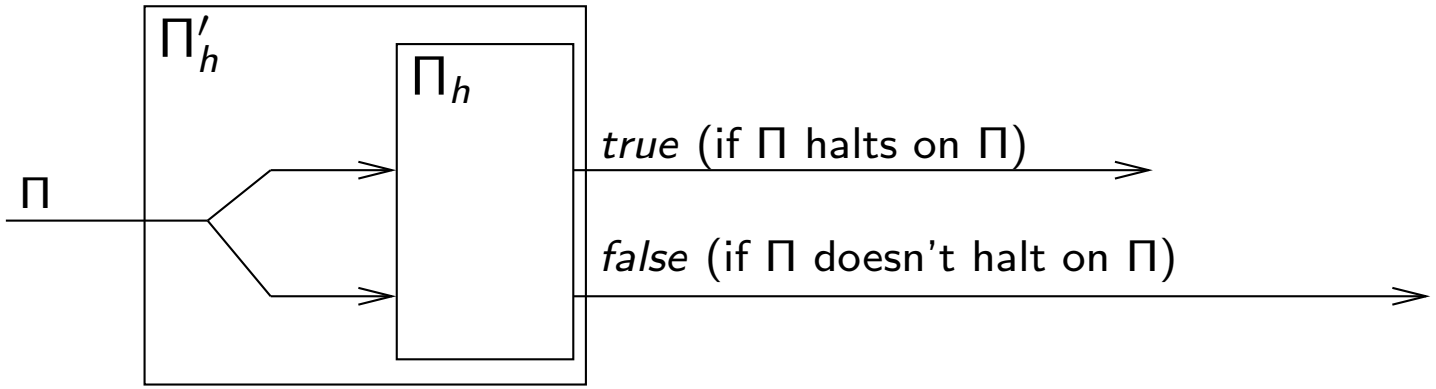



```
Boolean Π_h (String Π, String I)
  /* code to check if Π terminates on I. */
```

Proving Undecidability of the Halting Problem (Step 2)

Build a program Π'_h from Π_h:

- Π'_h takes one string as input, duplicates it, and passes it to Π_h.
- Hence, Π'_h checks whether an input program Π halts on Π.

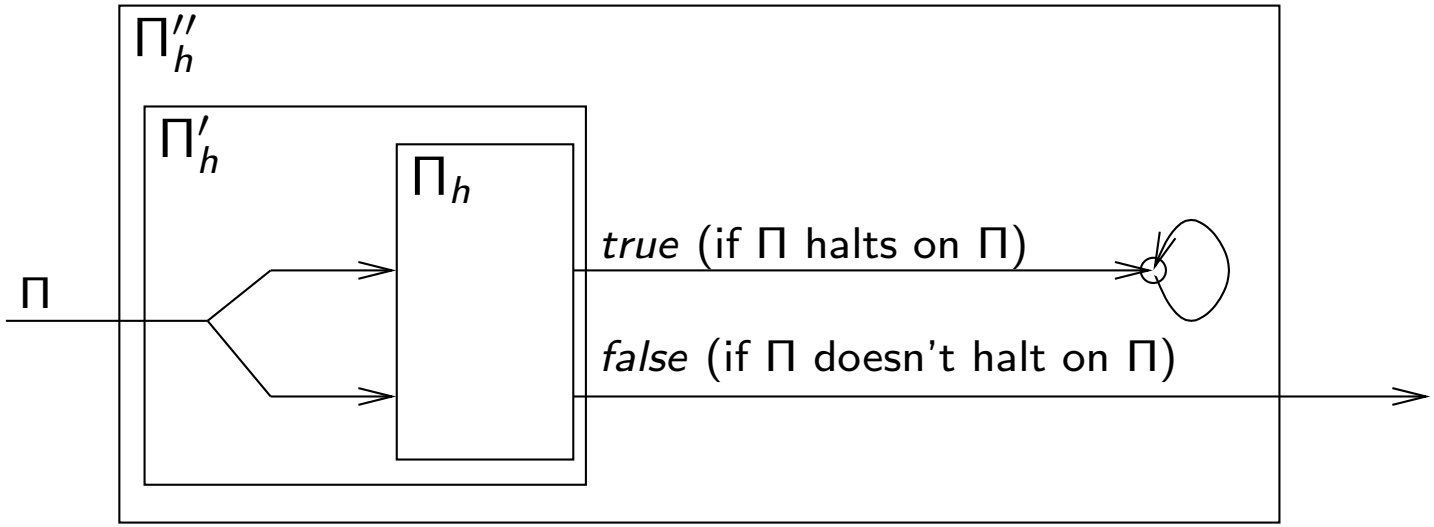


Boolean Π'_h (String Π)
return $\Pi_h(\Pi, \Pi)$;

Proving Undecidability of the Halting Problem (Step 3)

Build another program Π''_h form Π'_h:

- if the code of Π'_h returns *true*, then Π''_h enters an infinite loop!



Boolean Π''_h (String Π)
if $\Pi'_h(\Pi) = \mathit{true}$ **then** { **while** (true) **do** {} }
else return *false*;

Observation 1:

1. if Π halts on Π, then Π''_h does not halt on Π.
2. if Π does not halt on Π, then Π''_h halts on Π.

Proving Undecidability of the Halting Problem (Step 4)

What happens if we feed (the source code of) Π''_h to Π''_h?

There can only be two possibilities:

1. Π''_h halts on Π''_h. It follows from Observation 1 that Π''_h does not halt on Π''_h. Contradiction!
2. Π''_h does not halt on Π''_h. It follows from Observation 2 that Π''_h halts on Π''_h. Contradiction!

Hence any attempt to write a program Π_h will lead to failure!

Theorem

The Halting problem is undecidable.

Other Examples of Undecidable Problems

CORRECTNESS

INSTANCE: A source code S for a function that takes a string and outputs a string, and a pair of strings I_1, I_2.
QUESTION: Does S return I_2 when run on input I_1?

Intuitively, **CORRECTNESS** is undecidable because answering the question involves checking if S terminates on I_1.

REACHABLE-CODE

INSTANCE: A source code S, a number n of a line in S.
QUESTION: Is there an input I for S such that the run of S on I will reach the code on line n?

Optimization potential: unreachable code can be safely removed!

Undecidability (intuitively): if we let n be the "last" line in S, then the problem is equivalent to checking if S terminates on some input.

Remark. These informal arguments for the undecidability of the above problems will be made precise later when we introduce "reductions".

Semi-decidable Problems

We relax the notion of decidability and introduce semi-decidable problems

Definition

A decision problem $\mathcal{P}$ is called semi-decidable if we can build a program Π such that:

- Π takes as input instances I of $\mathcal{P}$;
- if I is a "yes" instance, then Π returns *true*;
- if I is a "no" instance, then Π returns *false* or does not terminate;

In other words:

- Π works correctly on all positive instances of $\mathcal{P}$, but
- Π may not terminate on the negative instances of $\mathcal{P}$.

Is the Halting problem semi-decidable?

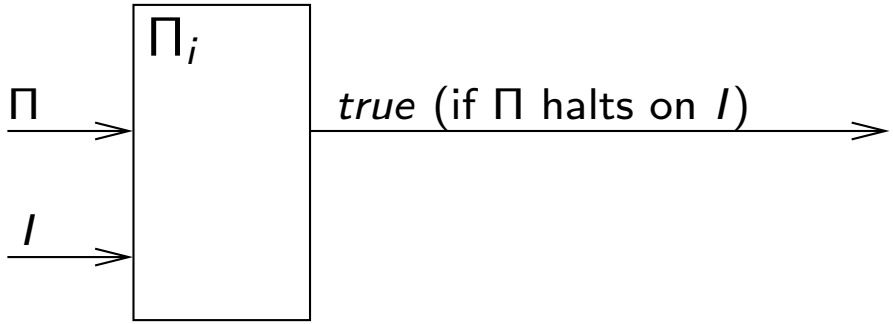
The Halting Problem is Semi-decidable because...

...we can construct a program that returns *true* for every positive instance of the Halting problem!

The positive instances of the Halting problem are pairs (Π, I) of a program and an input such that Π halts on I.

For instance, we can write an interpreter program Π_i such that:

- Π_i takes as input a source code Π and an input I for Π;
- Π_i parses Π and simulates a run of Π on I;
- If the simulation of Π on I reaches the end, then Π_i returns *true*;
- If the simulation of Π on I does not end, then, clearly, Π_i cannot return any value;



Other Examples of Semi-decidable Problems

REACHABLE-CODE is semi-decidable, but the proof is more involved.

Recall: **REACHABLE-CODE**

INSTANCE: A source code S, a number n of a line in S.
QUESTION: Is there an input I for S such that the run of S on I will reach the code on line n?

- Observation: given a pair (I, i), where I is an input for S and i is a natural number, we can
  - execute in an interpreter the first i steps of S running on I, and
  - decide whether the code on line n is reached within i steps.
- Simply enumerate all pairs (I, i) and for each pair check whether S on I reaches the line n within i steps:
- if (S, n) is a positive instance, we will eventually find (I, i) such that S on I reaches the line n within i steps (then output *true*).
- if (S, n) is a negative instance, the enumeration will never end.

REACHABLE-CODE continued

Enumeration of the pairs (I, i)

Observation 1. The set of finite strings over a finite alphabet is countably infinite. Hence, the set of possible inputs I is countable.

Observation 2. The cartesian product of two countable sets is countable.

REACHABLE-CODE continued

Enumeration of the pairs (I, i)

Observation 1. The set of finite strings over a finite alphabet is countably infinite. Hence, the set of possible inputs I is countable.

Observation 2. The cartesian product of two countable sets is countable.

Cantor's enumeration principle

enumeration of $A \times B$ with $A = \{a_1, a_2, a_3, \dots\}$ and $B = \{b_1, b_2, b_3, \dots\}$

$$\begin{pmatrix} (a_1, b_1) & (a_1, b_2) & (a_1, b_3) & \dots, \\ (a_2, b_1) & (a_2, b_2) & (a_2, b_3) & \dots, \\ (a_3, b_1) & (a_3, b_2) & (a_3, b_3) & \dots, \\ (a_4, b_1) & (a_4, b_2) & (a_4, b_3) & \dots, \\ \vdots & \vdots & \vdots & \end{pmatrix}$$

REACHABLE-CODE continued

Enumeration of the pairs (I, i)

Observation 1. The set of finite strings over a finite alphabet is countably infinite. Hence, the set of possible inputs I is countable.

Observation 2. The cartesian product of two countable sets is countable.

Cantor's enumeration principle

enumeration of $A \times B$ with $A = \{a_1, a_2, a_3, \dots\}$ and $B = \{b_1, b_2, b_3, \dots\}$

$$\begin{pmatrix} (a_1, b_1) & (a_1, b_2) & (a_1, b_3) & \dots, \\ (a_2, b_1) & (a_2, b_2) & (a_2, b_3) & \dots, \\ (a_3, b_1) & (a_3, b_2) & (a_3, b_3) & \dots, \\ (a_4, b_1) & (a_4, b_2) & (a_4, b_3) & \dots, \\ \vdots & \vdots & \vdots & \end{pmatrix} \quad \begin{pmatrix} 1 & 4 & 9 & 16 & \dots, \\ 2 & 3 & 8 & 15 & \dots, \\ 5 & 6 & 7 & 14 & \dots, \\ 10 & 11 & 12 & 13 & \dots, \\ \vdots & \vdots & \vdots & \vdots & \end{pmatrix}$$

REACHABLE-CODE continued

Enumeration of the pairs (I, i)

Observation 1. The set of finite strings over a finite alphabet is countably infinite. Hence, the set of possible inputs I is countable.

Observation 2. The cartesian product of two countable sets is countable.

Cantor's enumeration principle

enumeration of $A \times B$ with $A = \{a_1, a_2, a_3, \dots\}$ and $B = \{b_1, b_2, b_3, \dots\}$

$$\begin{pmatrix} (a_1, b_1) & (a_1, b_2) & (a_1, b_3) & \dots, \\ (a_2, b_1) & (a_2, b_2) & (a_2, b_3) & \dots, \\ (a_3, b_1) & (a_3, b_2) & (a_3, b_3) & \dots, \\ (a_4, b_1) & (a_4, b_2) & (a_4, b_3) & \dots, \\ \vdots & \vdots & \vdots & \end{pmatrix} \quad \begin{pmatrix} 1 & 4 & 9 & 16 & \dots, \\ 2 & 3 & 8 & 15 & \dots, \\ 5 & 6 & 7 & 14 & \dots, \\ 10 & 11 & 12 & 13 & \dots, \\ \vdots & \vdots & \vdots & \vdots & \end{pmatrix}$$

REACHABLE-CODE continued

Enumeration of the pairs (I, i)

Observation 1. The set of finite strings over a finite alphabet is countably infinite. Hence, the set of possible inputs I is countable.

Observation 2. The cartesian product of two countable sets is countable.

Cantor's enumeration principle

enumeration of $A \times B$ with $A = \{a_1, a_2, a_3, \dots\}$ and $B = \{b_1, b_2, b_3, \dots\}$

$$\begin{pmatrix} (a_1, b_1) & (a_1, b_2) & (a_1, b_3) & \dots, \\ (a_2, b_1) & (a_2, b_2) & (a_2, b_3) & \dots, \\ (a_3, b_1) & (a_3, b_2) & (a_3, b_3) & \dots, \\ (a_4, b_1) & (a_4, b_2) & (a_4, b_3) & \dots, \\ \vdots & \vdots & \vdots & \end{pmatrix} \quad \begin{pmatrix} 1 & 4 & 9 & 16 & \dots, \\ 2 & 3 & 8 & 15 & \dots, \\ 5 & 6 & 7 & 14 & \dots, \\ 10 & 11 & 12 & 13 & \dots, \\ \vdots & \vdots & \vdots & \vdots & \end{pmatrix}$$

REACHABLE-CODE continued

Enumeration of the pairs (I, i)

Observation 1. The set of finite strings over a finite alphabet is countably infinite. Hence, the set of possible inputs I is countable.

Observation 2. The cartesian product of two countable sets is countable.

Cantor's enumeration principle

enumeration of $A \times B$ with $A = \{a_1, a_2, a_3, \dots\}$ and $B = \{b_1, b_2, b_3, \dots\}$

$$\begin{pmatrix} (a_1,b_1) & (a_1,b_2) & (a_1,b_3) & \dots, \\ (a_2,b_1) & (a_2,b_2) & (a_2,b_3) & \dots, \\ (a_3,b_1) & (a_3,b_2) & (a_3,b_3) & \dots, \\ (a_4,b_1) & (a_4,b_2) & (a_4,b_3) & \dots, \\ \vdots & \vdots & \vdots & \end{pmatrix} \quad \begin{pmatrix} 1 & 4 & 9 & 16 & \dots, \\ 2 & 3 & 8 & 15 & \dots, \\ 5 & 6 & 7 & 14 & \dots, \\ 10 & 11 & 12 & 13 & \dots, \\ \vdots & \vdots & \vdots & \vdots & \end{pmatrix}$$

REACHABLE-CODE continued

Enumeration of the pairs (I, i)

Observation 1. The set of finite strings over a finite alphabet is countably infinite. Hence, the set of possible inputs I is countable.

Observation 2. The cartesian product of two countable sets is countable.

Cantor's enumeration principle

enumeration of $A \times B$ with $A = \{a_1, a_2, a_3, \ldots\}$ and $B = \{b_1, b_2, b_3, \ldots\}$

$$\begin{pmatrix} (a_1, b_1) & (a_1, b_2) & (a_1, b_3) & \ldots, \\ (a_2, b_1) & (a_2, b_2) & (a_2, b_3) & \ldots, \\ (a_3, b_1) & (a_3, b_2) & (a_3, b_3) & \ldots, \\ (a_4, b_1) & (a_4, b_2) & (a_4, b_3) & \ldots, \\ \vdots & \vdots & \vdots & \end{pmatrix} \quad \begin{pmatrix} 1 & 4 & 9 & 16 & \ldots, \\ 2 & 3 & 8 & 15 & \ldots, \\ 5 & 6 & 7 & 14 & \ldots, \\ 10 & 11 & 12 & 13 & \ldots, \\ \vdots & \vdots & \vdots & \vdots & \end{pmatrix}$$

Other Examples of Semi-decidable Problems (2)

CORRECTNESS is semi-decidable (analogous to the Halting problem).

Entscheidungsproblem

INSTANCE: A formula φ in First-Order Logic.
QUESTION: Is φ valid?

The **Entscheidungsproblem** is semi-decidable because:

- For each valid formula there is a finite derivation in the Sequential Calculus (or any other complete deduction system like Natural Deduction).
- Simply enumerate all derivations in the deduction system.
  - if a formula φ is valid, we will eventually find a derivation for φ
  - if φ is not valid, the program would not terminate (no problem – we are not asking for decidability).

The **Entscheidungsproblem** is semi-decidable but not decidable.

Complement of a Decision Problem

- Let $\mathcal{P}$ be a decision problem, i.e. a "yes/no" question with a set of possible instances of the problem.
- The complement of $\mathcal{P}$ is obtained by "inverting" the question of $\mathcal{P}$.

Recall

REACHABILITY

INSTANCE: A graph (V, E) and nodes $u, v \in V$.
QUESTION: Is there a path in the graph from u to v?

The complement of **REACHABILITY** is

UNREACHABILITY (or **co-REACHABILITY**)

INSTANCE: A graph (V, E) and nodes $u, v \in V$.
QUESTION: Is there no path in the graph from u to v?

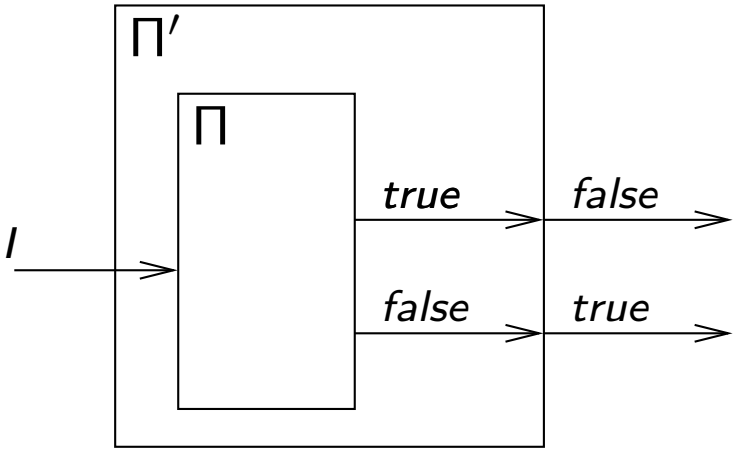
Properties of Complementation

Theorem

If $\mathcal{P}$ is a decidable decision problem and co-$\mathcal{P}$ is the complement of $\mathcal{P}$, then co-$\mathcal{P}$ is also decidable.

Proof:

- If $\mathcal{P}$ is a decidable decision problem, then there is a program Π that returns *true* on all positive instances of $\mathcal{P}$ and *false* on all negative instances of $\mathcal{P}$.
- Take a program Π' that is exactly as Π but inverts the output value.
- Π' is a decision procedure for co-$\mathcal{P}$.



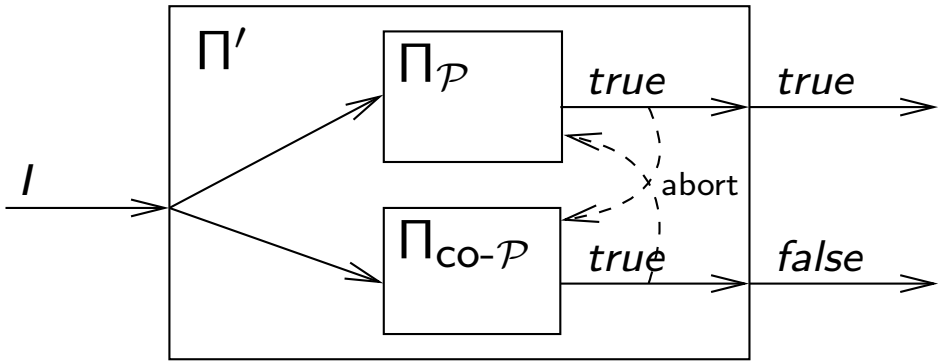

```
Boolean Π′ (String I)
    if Π(I) = true
            then return false
            else return true;
```

Theorem

If $\mathcal{P}$ is a decision problem, co-$\mathcal{P}$ is the complement of $\mathcal{P}$, and both $\mathcal{P}$ and co-$\mathcal{P}$ are semi-decidable, then $\mathcal{P}$ is decidable.

Proof:

- Since $\mathcal{P}$ is semi-decidable, there is a program $\Pi_{\mathcal{P}}$ that terminates and returns *true* on all positive instances of $\mathcal{P}$.
- Since co-$\mathcal{P}$ is semi-decidable, there is a program $\Pi_{\text{co-}\mathcal{P}}$ that terminates and returns *true* on all positive instances of co-$\mathcal{P}$, or equivalently, on all negative instances of $\mathcal{P}$.
- Define Π' that runs $\Pi_{\mathcal{P}}$ and $\Pi_{\text{co-}\mathcal{P}}$ in parallel, and outputs the corresponding result when one of the two programs terminates (termination of one triggers termination of the other).



Beyond Semi-decidability

- We have seen problems which are undecidable but semi-decidable.
- Question: are there problems which are not even semi-decidable?

Theorem

If $\mathcal{P}$ is a decision problem such that

1. *$\mathcal{P}$ is undecidable, and*
2. *$\mathcal{P}$ is semi-decidable, then*

the complement co-$\mathcal{P}$ of $\mathcal{P}$ is not semi-decidable.

Proof.

Proof by contradiction. Suppose co-$\mathcal{P}$ is semi-decidable. By the previous theorem, since $\mathcal{P}$ and co-$\mathcal{P}$ are both semi-decidable, then $\mathcal{P}$ is decidable. This contradicts the assumption (1) that $\mathcal{P}$ is undecidable. □

We can infer from the theorem that the complement of the Halting problem is not semi-decidable. The same is true for CORRECTNESS, REACHABLE-CODE and the Entscheidungsproblem.

Beyond Semi-decidability (2)

- We have seen problems which are not semi-decidable; but their complements are semi-decidable.
- Question: are there problems $\mathcal{P}$ such that neither $\mathcal{P}$ nor its complement co-$\mathcal{P}$ is semi-decidable?
- Such problems exist:

SAME-OUTPUT

INSTANCE: A pair Π_1, Π_2 of programs that take a single string as input, an input string I.
QUESTION: Do Π_1 and Π_2 behave the same on input I? That is, Π_1 on I and Π_2 on I both return the same value or both do not terminate?

- Intuitively, we cannot recognize positive instances where Π_1 and Π_2 both don't terminate on I. If we could, non-termination would be semi-decidable, leading to a contradiction.
- Neither can we recognize negative instances such that Π_1 terminates on I and Π_2 doesn't terminate on I. Again, if we could, then non-termination would be semi-decidable.

Beyond Semi-decidability (3)

The following problems have the same undecidability properties as SAME-OUTPUT:

ALL-HALTING

INSTANCE: A program Π that takes a single string as input.
QUESTION: Does Π halt on all input strings I?

- Intuitively, positive instances of ALL-HALTING are unrecognizable because we need to check termination on infinitely many strings I.
- Negative instances are unrecognizable because we cannot spot non-termination on some string I.

PROGRAM-EQUIVALENCE

INSTANCE: A pair Π_1, Π_2 of programs that take a single string as input.
QUESTION: Are Π_1 and Π_2 equivalent?
That is, is it true that for all inputs I, Π_1 on I and Π_2 on I both return the same value or both do not terminate?

Learning Objectives

- Ability to read and formulate decision/optimization problems
- Several kinds of problems (decision p., function p., optimization p., enumeration p., counting p.)
- Problem vs. problem instance
- Problem vs. algorithm vs. program
- Church-Turing thesis
- Halting problem
- Decidability vs. undecidability vs. semi-decidability
- Complement of a decision problem
- Properties of complementation