Introduction to Optimization

Basic Flavors of Complexity Theory

September 28, 2015

École Centrale Paris, Châtenay-Malabry, France

Dimo Brockhoff

INRIA Lille – Nord Europe

Course Overview

Date		Topic
Mon, 21.9.2015		Introduction
Mon, 28.9.2015	**D**	**Basic Flavors of Complexity Theory**
Mon, 5.10.2015	D	Greedy algorithms
Mon, 12.10.2015	D	Dynamic programming
Mon, 2.11.2015	D	Branch and bound/divide&conquer
Fri, 6.11.2015	D	Approximation algorithms and heuristics
Fri, 9.11.2015	C	Introduction to Continuous Optimization I
Fri, 13.11.2015	C	Introduction to Continuous Optimization II
Fri, 20.11.2015	C	Gradient-based Algorithms
Fri, 27.11.2015	C	End of Gradient-based Algorithms + Linear Programming
Fri, 4.12.2015	C	Stochastic Optimization and Derivative Free Optimization
Tue, 15.12.2015		Exam

all classes + exam last 3 hours (incl. a 15min break)

Motivation: Analyzing Algorithm Runtimes

- we want to analyze algorithms for discrete problems
- to be more precise: want to know runtime to find the optimum

Not realistic:

- do this for any input sequence
- do this for any machine, programming language, compiler, ...

Instead:

- abstract from a real implementation to the algorithm run on an abstract machine model

  [use a model which makes useful predictions in the real world]
- analyze the algorithm runtime for all instances of a given input size (worst case, average case, ...)

Motivation: Analyzing the Optimal Algorithms

- want to know how quick an optimal algorithm would run
  - how much slower is my own one?
- want to know the general difficulty of problems
  - why can’t I find an efficient algorithm for my problem?

Complexity Theory

A part of theoretical computer science that is concerned about:

- comparison of (optimization) problems regarding their difficulty
- classes of difficulties
- computability in general

Complexity Theory: Lecture Overview

- deterministic machine models
- computability
  - an example of a problem which cannot be solved by a computer
- non-determinism and the class NP
- difficult problems:
  - the classes NP-complete, NP-hard, etc.
  - polynomial reductions
- the complexity zoo

Note: complexity theory is often a full lecture by itself!

Algorithm Runtimes in Reality

Algorithm runtimes depend on

- hardware (cpu, RAM, ...)
- the used programming language
- the used compiler/interpreter
- other load on the machine
- implementation “tricks” (running on GPU, compiler options, ...)

But still, we often make general statements like

- “Quicksort is a good sorting algorithm.”
- “My algorithm is quicker than yours.”
- “Algorithm A is the best possible algorithm for problem P.”

how comes? what does it mean?

Abstractions for Algorithm Runtime Considerations

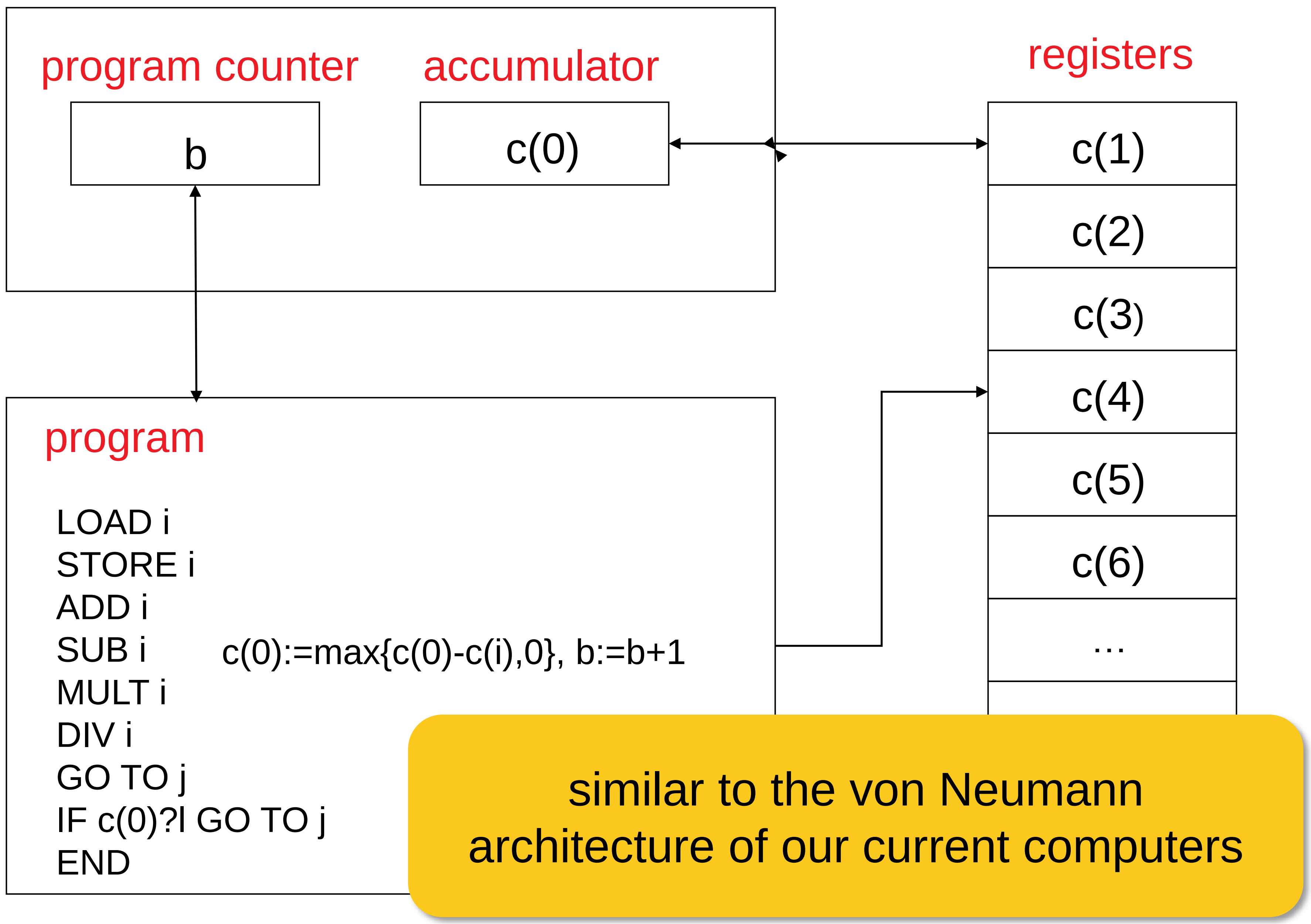
...because we abstract!

- for SORTING for example: number of comparisons as basic operation (actual runtime will again depend on hard- and software)
- often basic calculations as basic model (addition, multiplication, division, ...)
  - but what model is good?
  - are addition and multiplication e.g. equally difficult?

Important Aspects:

- relation to our real-world computers
- optimally, the choice of the model does not matter!

The Random Access Machine (RAM)



LOAD i
STORE i
ADD i
SUB i c(0):=max{c(0)-c(i),0}, b:=b+1
MULT i
DIV i
GO TO j
IF c(0)?l GO TO j
END

similar to the von Neumann architecture of our current computers

The Random Access Machine

is similar to the von Neumann architecture of our current computers

But:

- simpler (no pipelining, caches, ...)
- registers can contain non-negative natural numbers!

Last point not too much of a restriction:

- general natural numbers simulated by 2 registers
- rational numbers simulated by 4 registers

But probably too optimistic for measuring performance:

operations on arbitrarily large numbers might cost much more on an actual computer!

Cost Measures

Uniform Cost Measure:

- each operation costs 1

Logarithmic Cost Measure:

- each operation costs relative to the length of the arguments
- log(ARG) is cost measure if we assume binary representations of the numbers

Problem Complexity

- for example for Random Access Machine and a given cost measure

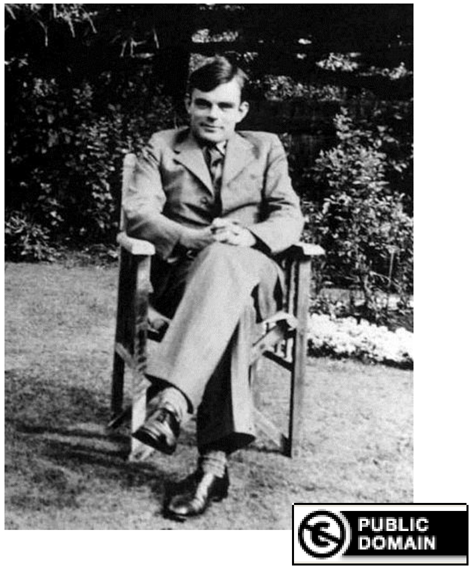
Complexity of problem Π

= number of operations needed for an optimal algorithm to solve each instance of Π

- important question: how much does this complexity depend on the machine model and the cost measure?
- moreover, independent of the existance of actual computers?

The Turing Machine (TM)

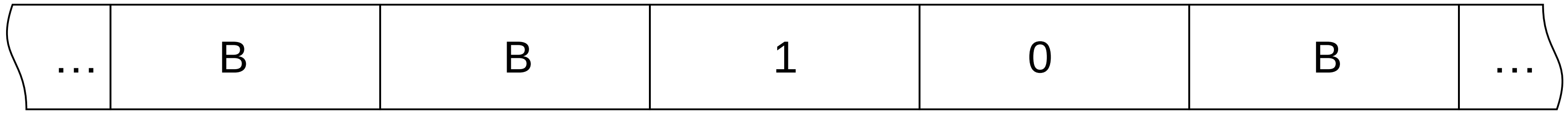
- Alan Turing (1912—1954)
- simplest ~~computer~~ computation model



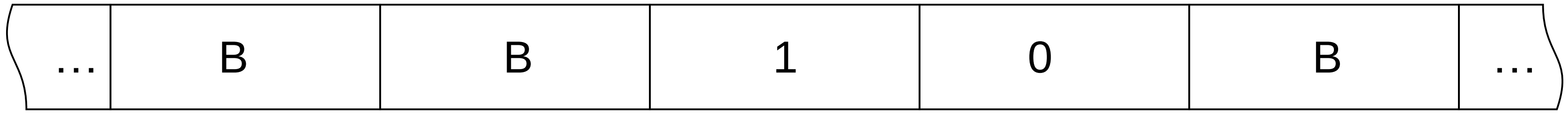
Formal definition:

$$(Q, \Sigma, \Gamma \supset \Sigma, B \in \Gamma \setminus \Sigma, q_0 \in Q, \delta, F \subset Q)$$

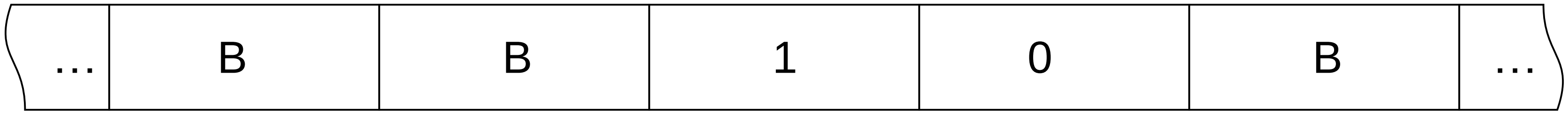
$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{R, L, N\}$$


…
B
B
1
0
B
…

$$(Q, \Sigma, \Gamma \supset \Sigma, B \in \Gamma \setminus \Sigma, q_0 \in Q, \delta, F \subset Q)$$

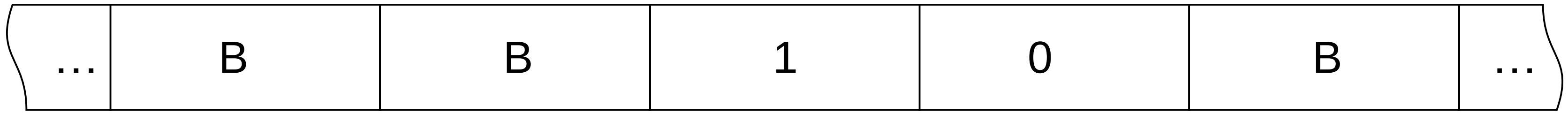


$$\left(\quad \Sigma, \qquad B \qquad\qquad\qquad\qquad \right)$$



$$\Big(\quad \underset{\substack{\big\uparrow \\ \text{input symbols}}}{\Sigma,} \qquad \underset{\substack{\big\uparrow \\ \text{blank}}}{B} \qquad\qquad\qquad\qquad \Big)$$

$\Sigma = \{0, 1\}$

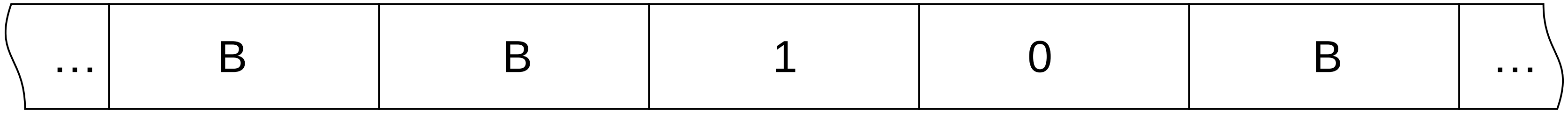


$$\Big(\quad \Sigma, \Gamma \supset \Sigma, B \in \Gamma \setminus \Sigma, \qquad\qquad\qquad\qquad \Big)$$

$\uparrow$

band alphabet

$\Gamma = \{0, 1, B\}$

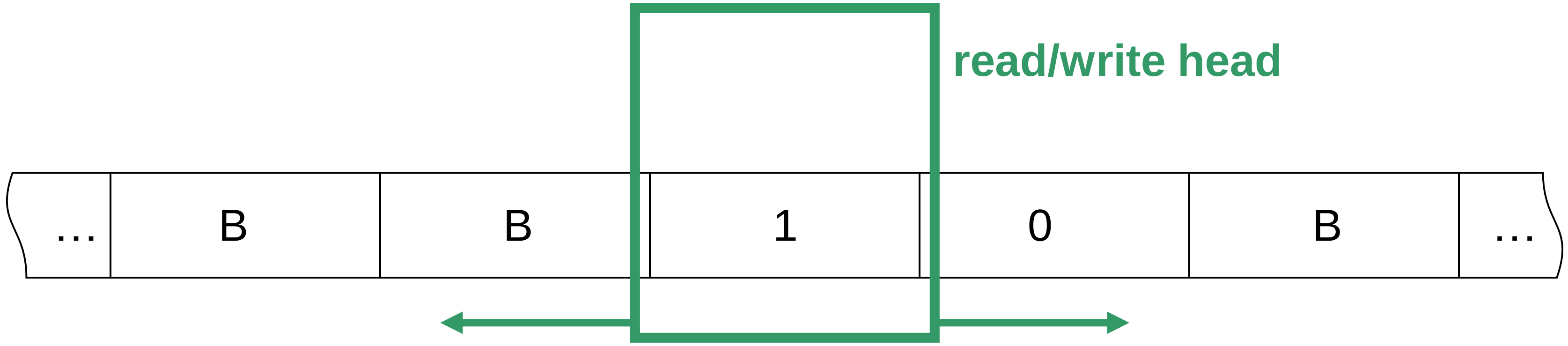


$$(\quad \Sigma, \Gamma \supset \Sigma, B \in \Gamma \setminus \Sigma, \qquad\qquad)$$

↑

band alphabet

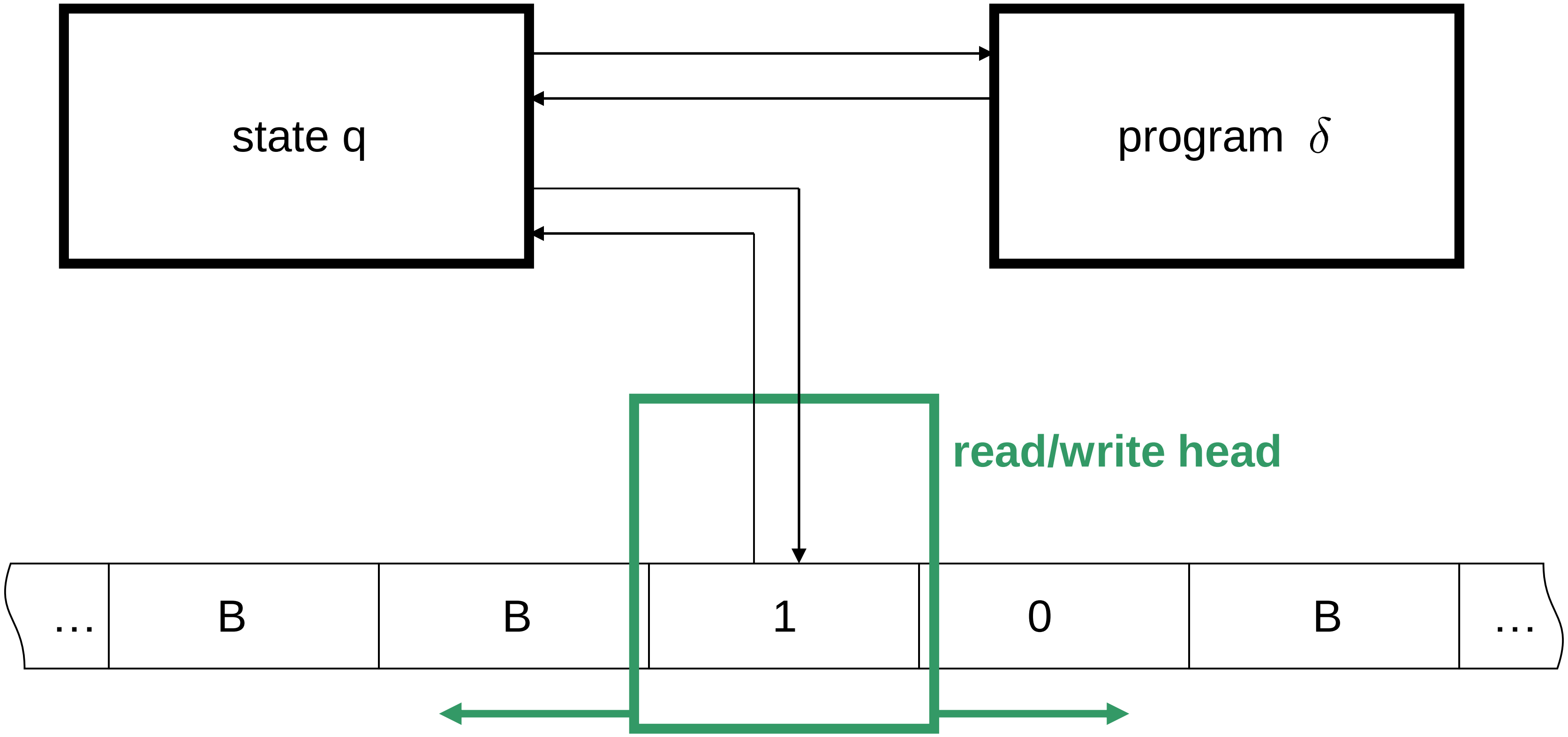
$$\Gamma = \{0, 1, B\}$$



$$(\quad \Sigma, \Gamma \supset \Sigma, B \in \Gamma \setminus \Sigma, \quad)$$

band alphabet

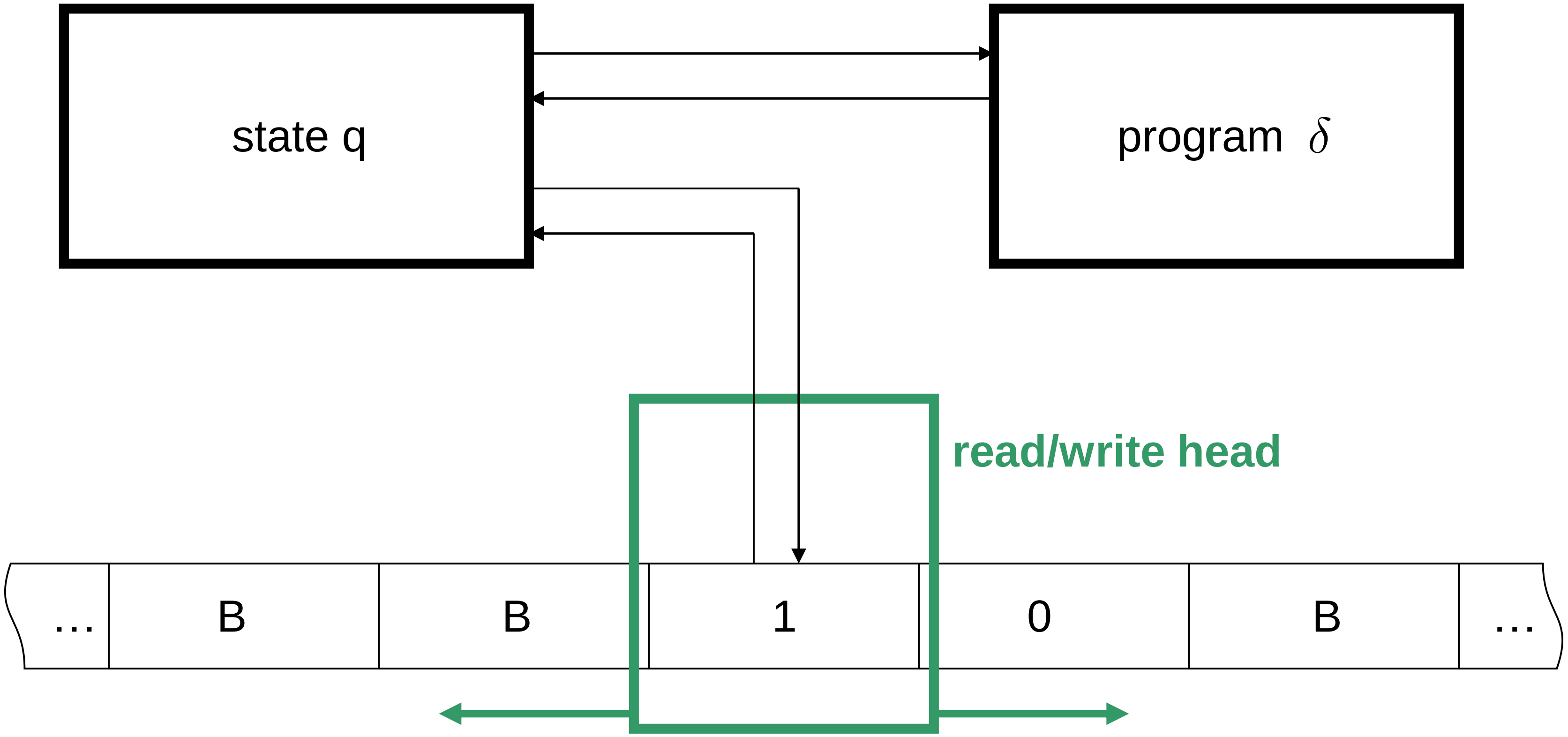
$\Gamma = \{0, 1, B\}$



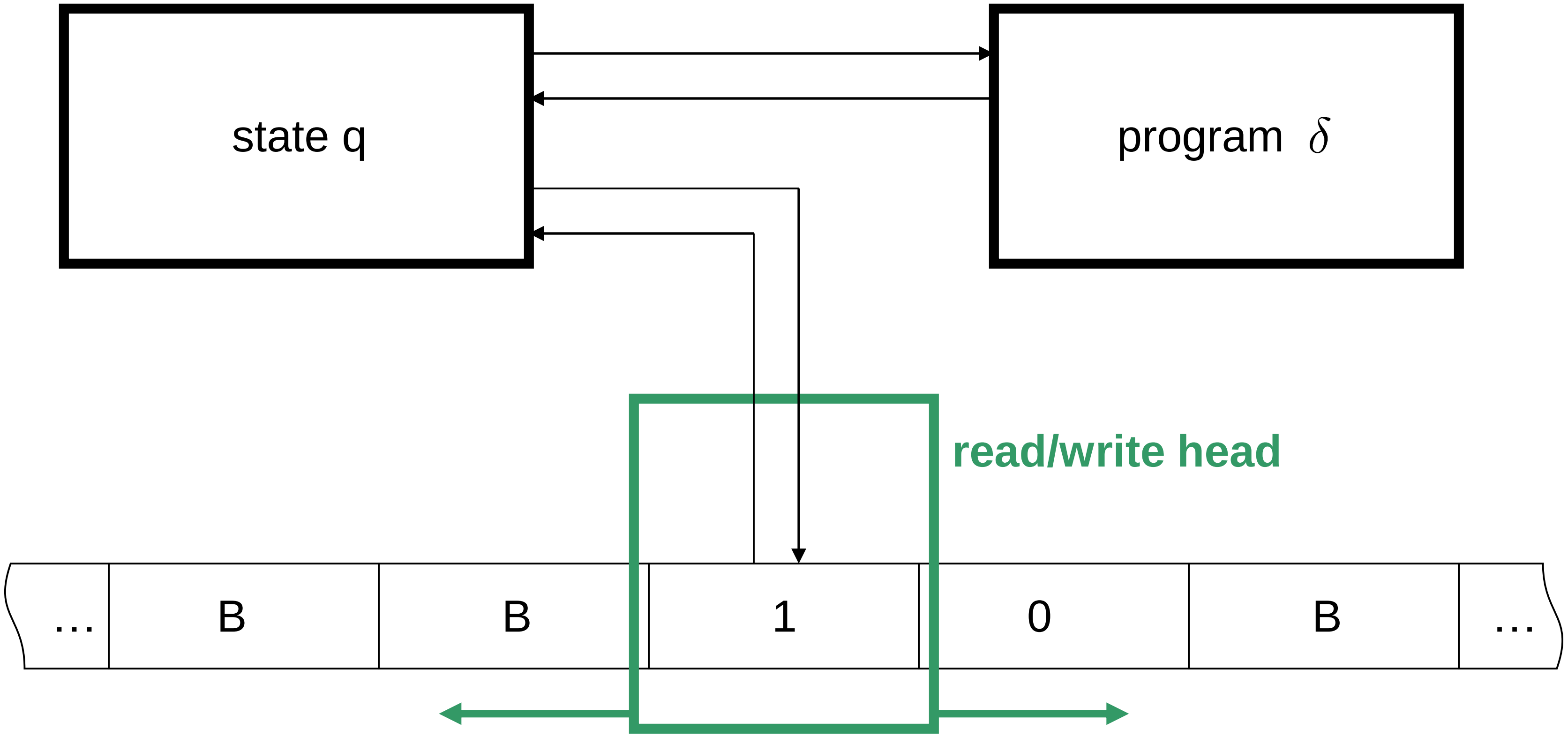
$$(Q, \Sigma, \Gamma \supset \Sigma, B \in \Gamma \setminus \Sigma, \qquad\qquad\qquad)$$

band alphabet

$$\Gamma = \{0, 1, B\}$$



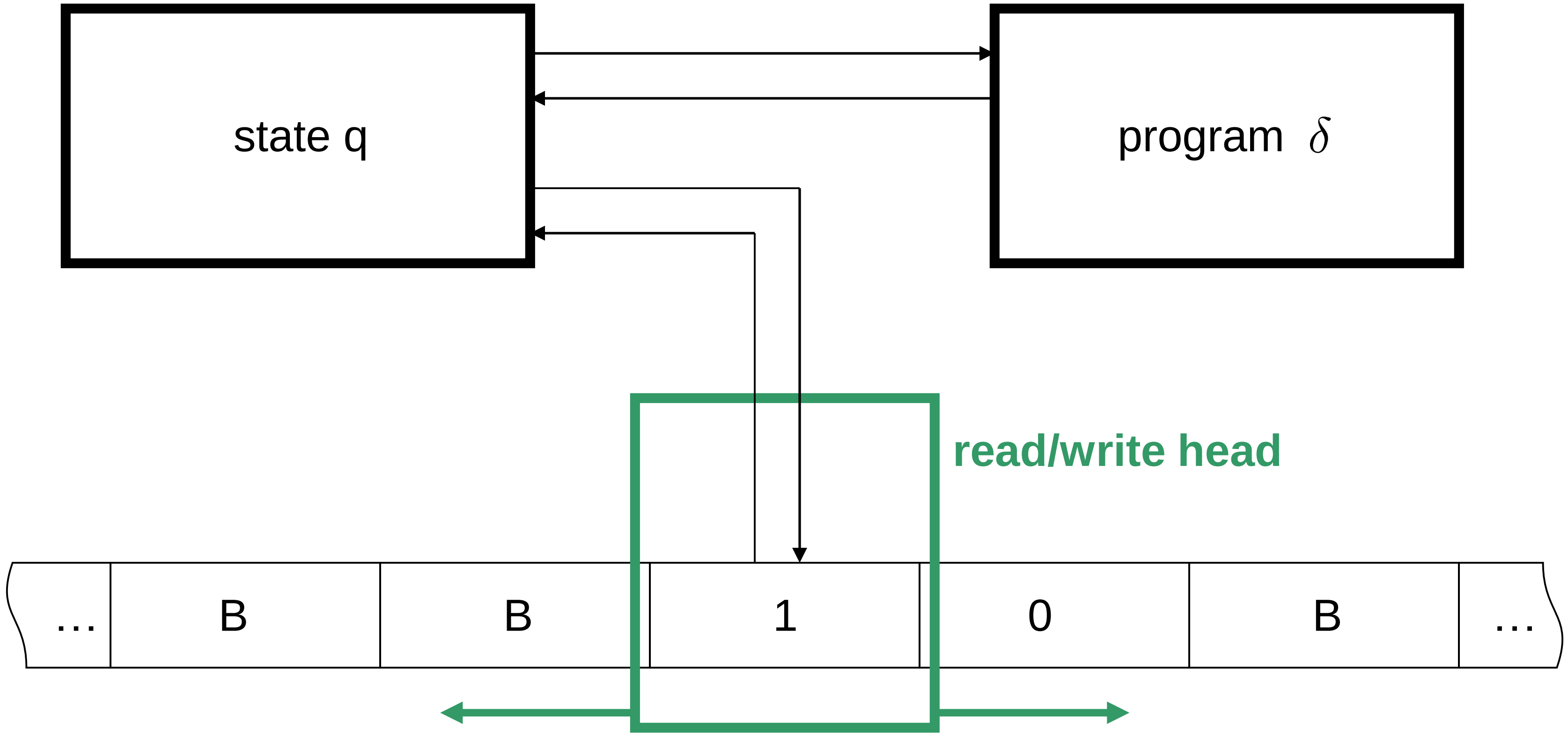
$$(Q, \Sigma, \underset{\substack{\uparrow\\ \text{band alphabet}\\ \Gamma = \{0,1,B\}}}{\Gamma} \supset \Sigma, B \in \Gamma \setminus \Sigma, q_0 \in Q, \qquad\qquad)$$



$$(Q, \Sigma, \Gamma \supset \Sigma, B \in \Gamma \setminus \Sigma, q_0 \in Q, \quad F \subset Q)$$

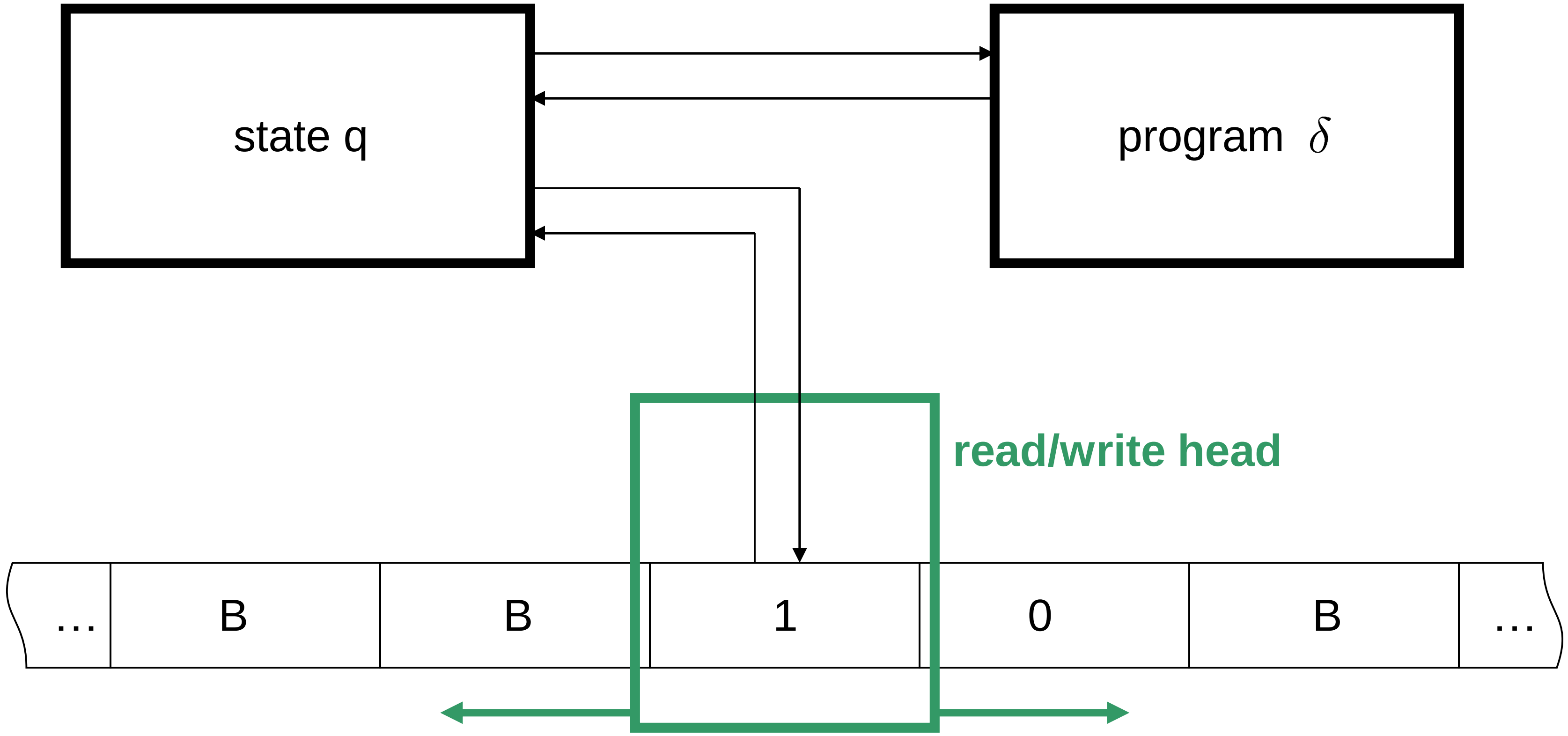
band alphabet

$\Gamma = \{0, 1, B\}$



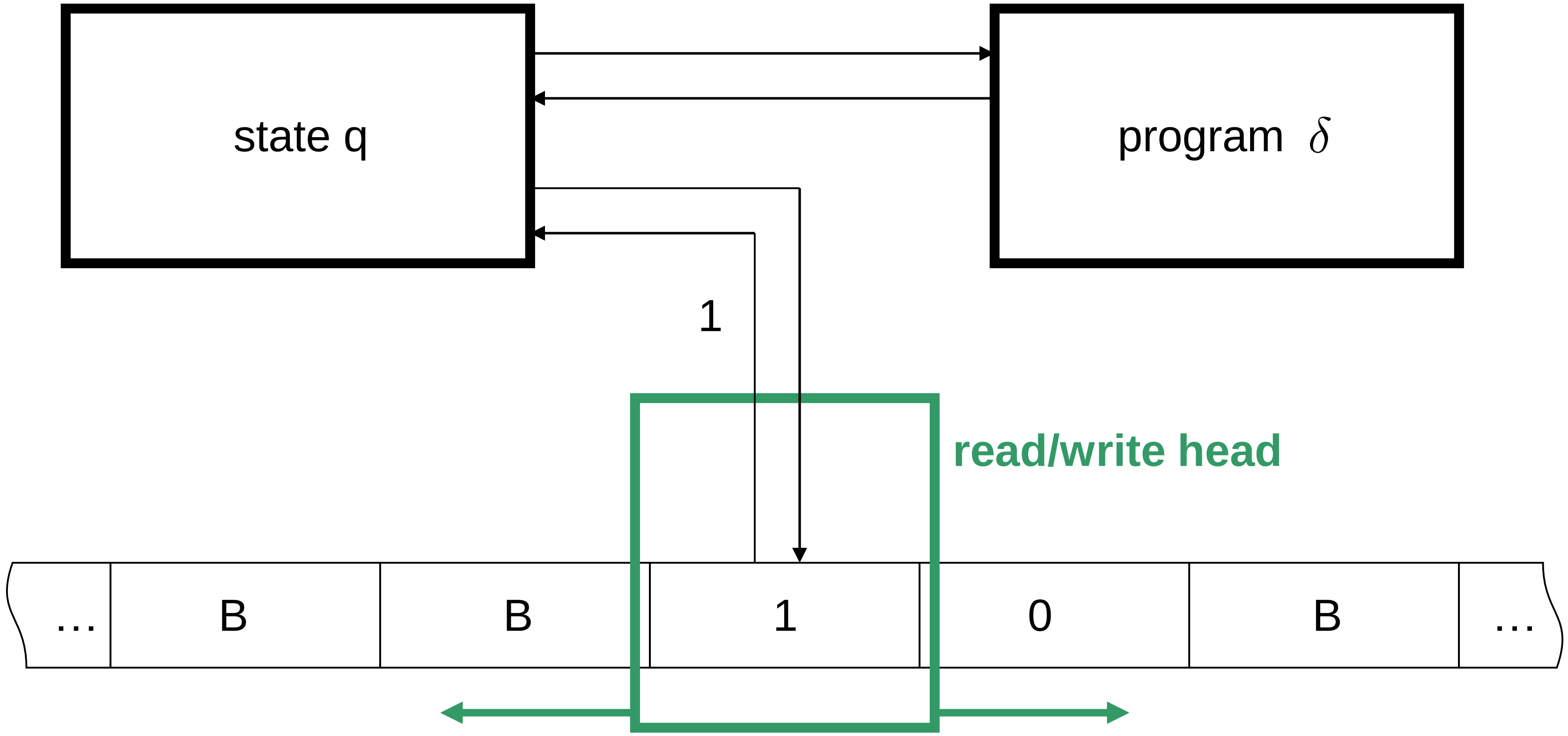
$$(Q, \Sigma, \Gamma \supset \Sigma, B \in \Gamma \setminus \Sigma, q_0 \in Q, \delta, F \subset Q)$$

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{R, L, N\}$$



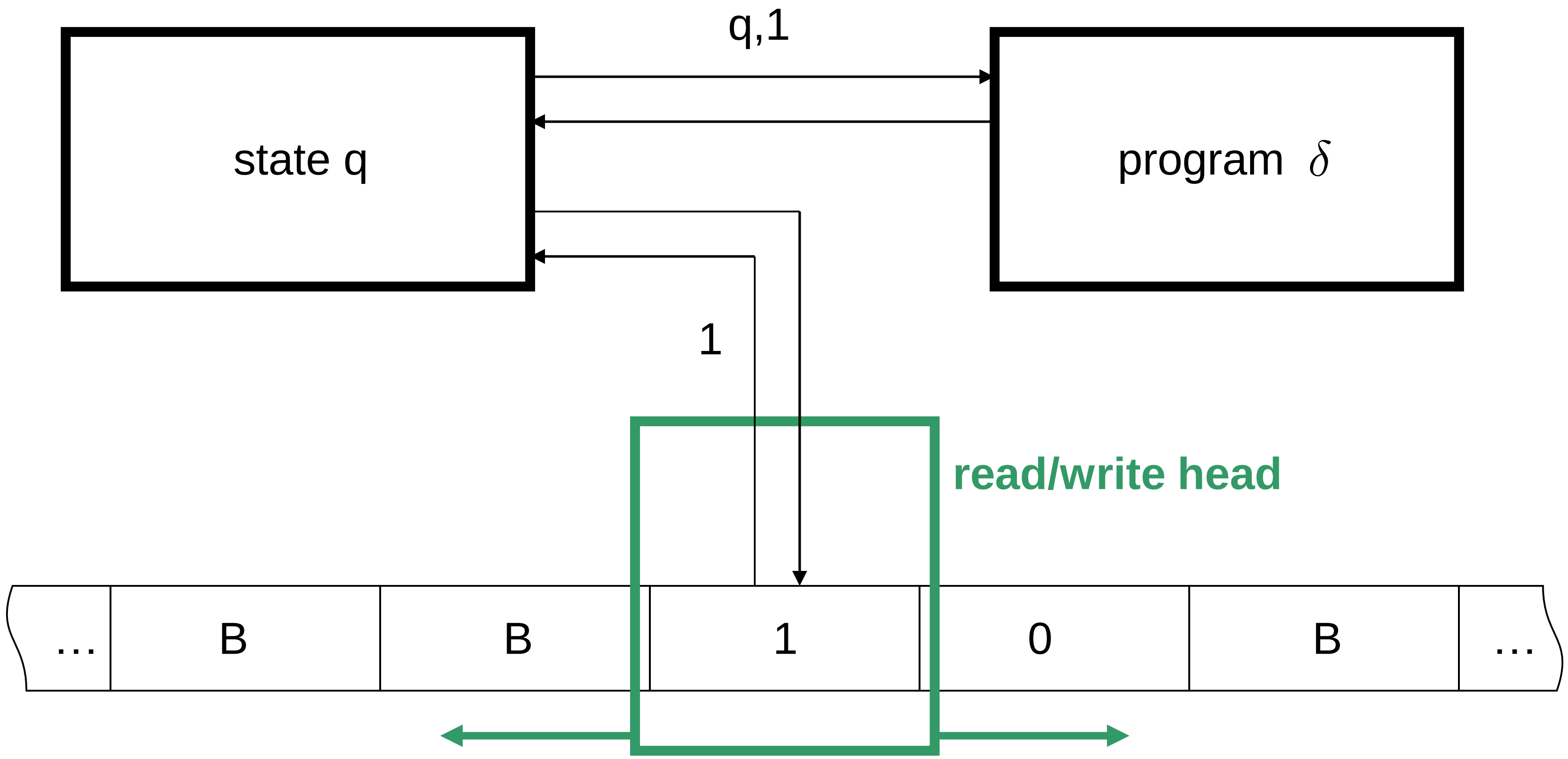
$$(Q, \Sigma, \Gamma \supset \Sigma, B \in \Gamma \setminus \Sigma, q_0 \in Q, \delta, F \subset Q)$$

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{R, L, N\}$$



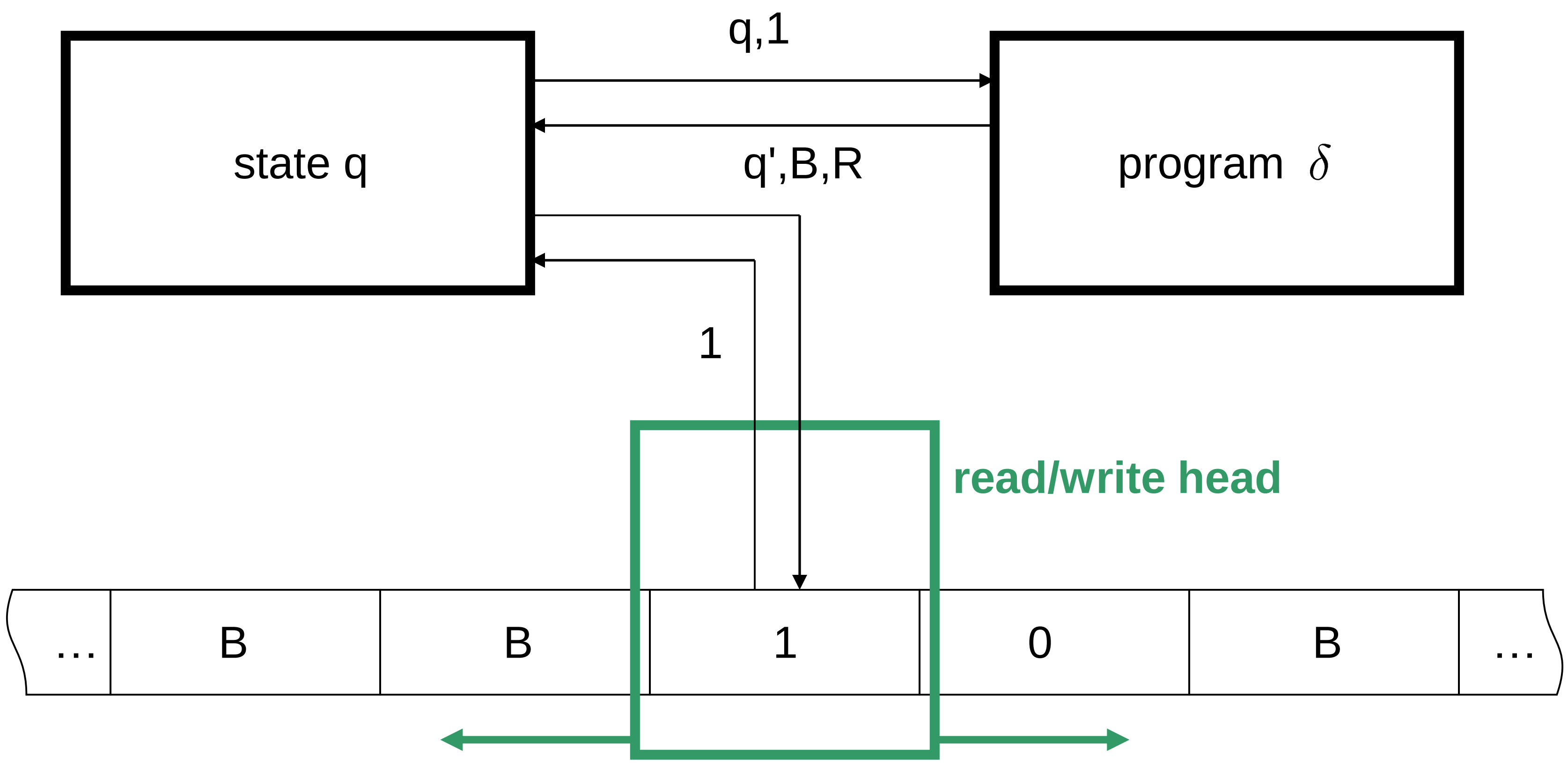
$$(Q, \Sigma, \Gamma \supset \Sigma, B \in \Gamma \setminus \Sigma, q_0 \in Q, \delta, F \subset Q)$$

$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{R, L, N\}$$



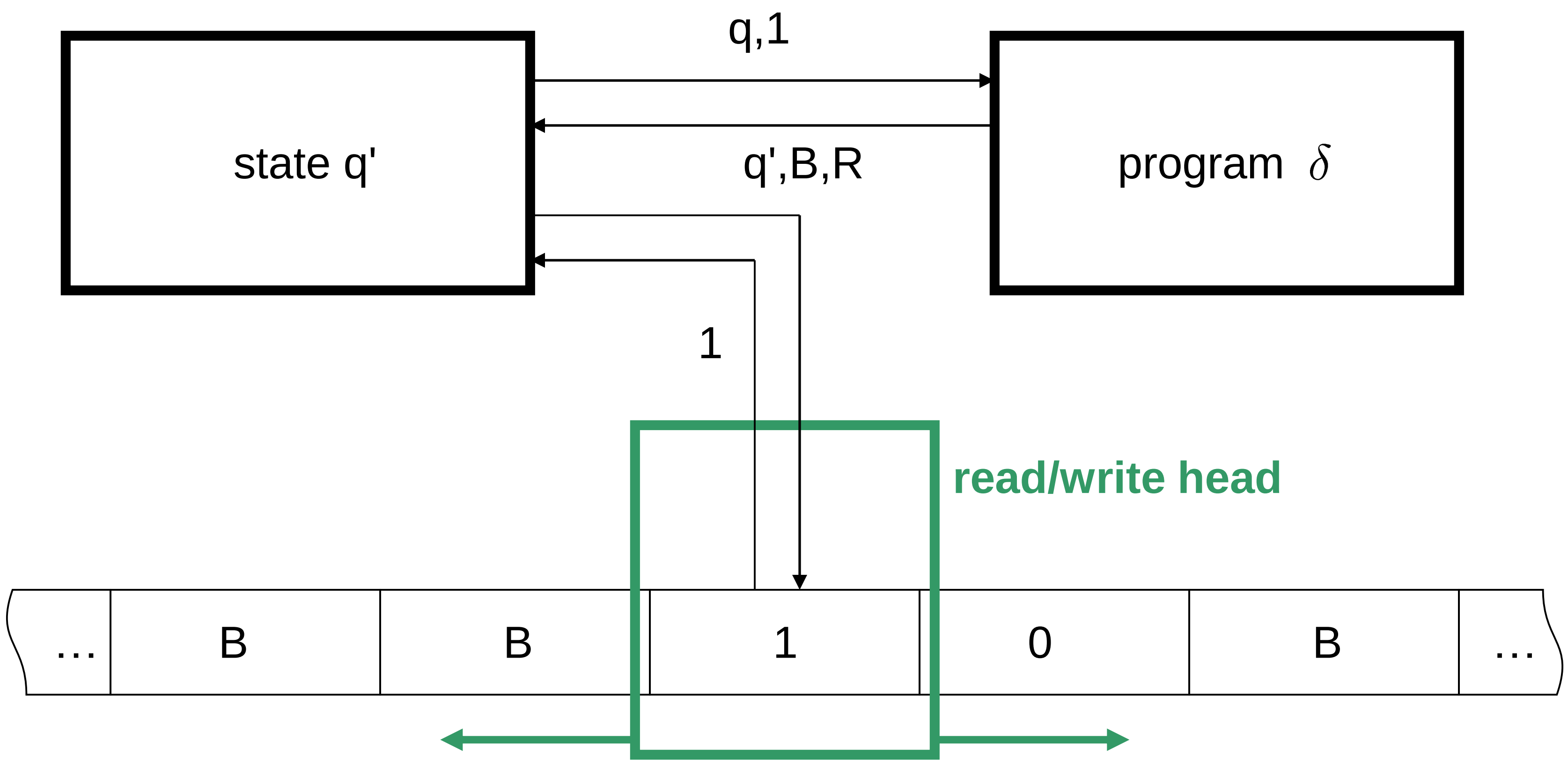
$$(Q, \Sigma, \Gamma \supset \Sigma, B \in \Gamma \setminus \Sigma, q_0 \in Q, \delta, F \subset Q)$$

$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{R, L, N\}$$



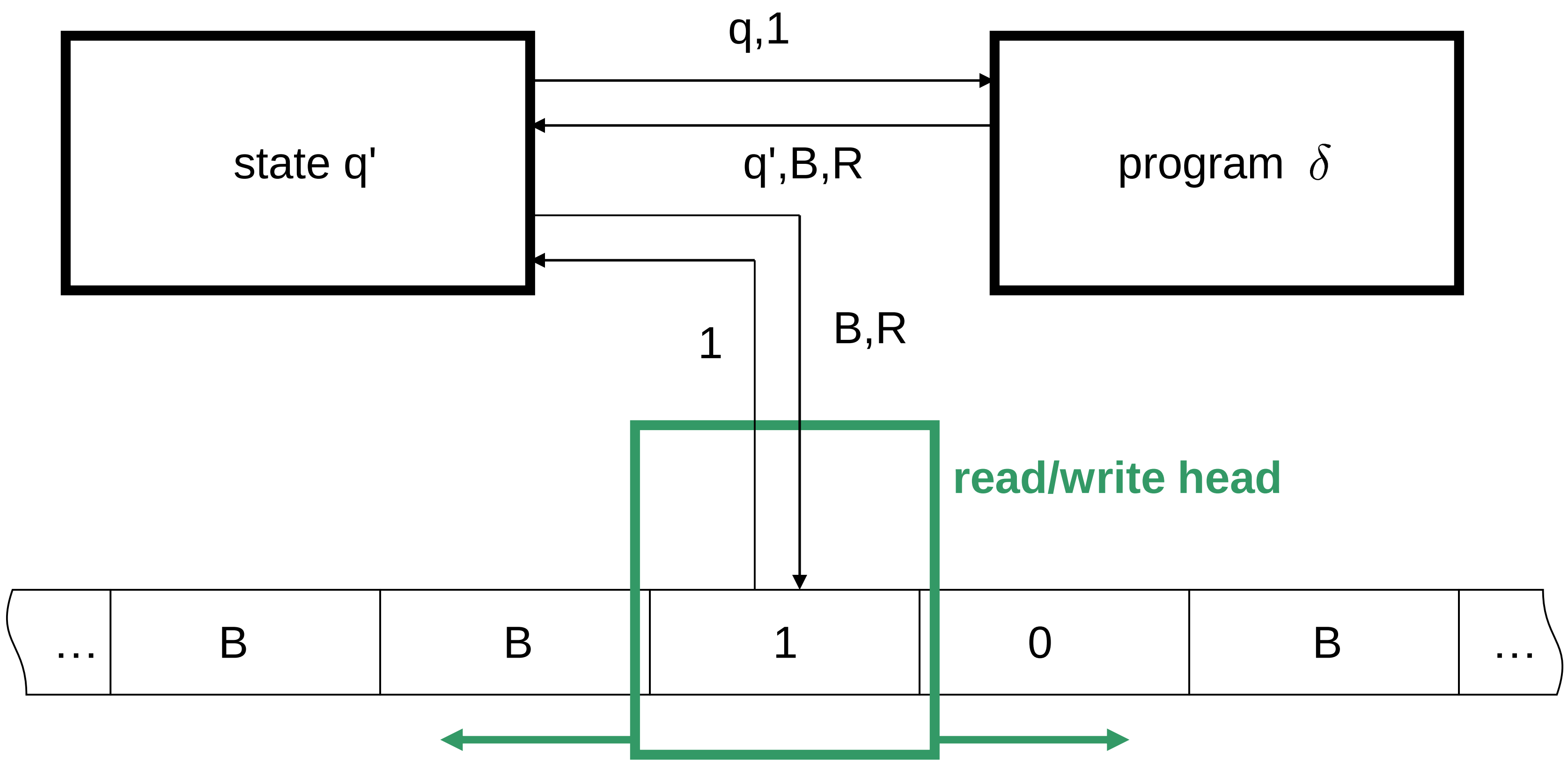
$$(Q, \Sigma, \Gamma \supset \Sigma, B \in \Gamma \setminus \Sigma, q_0 \in Q, \delta, F \subset Q)$$

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{R, L, N\}$$



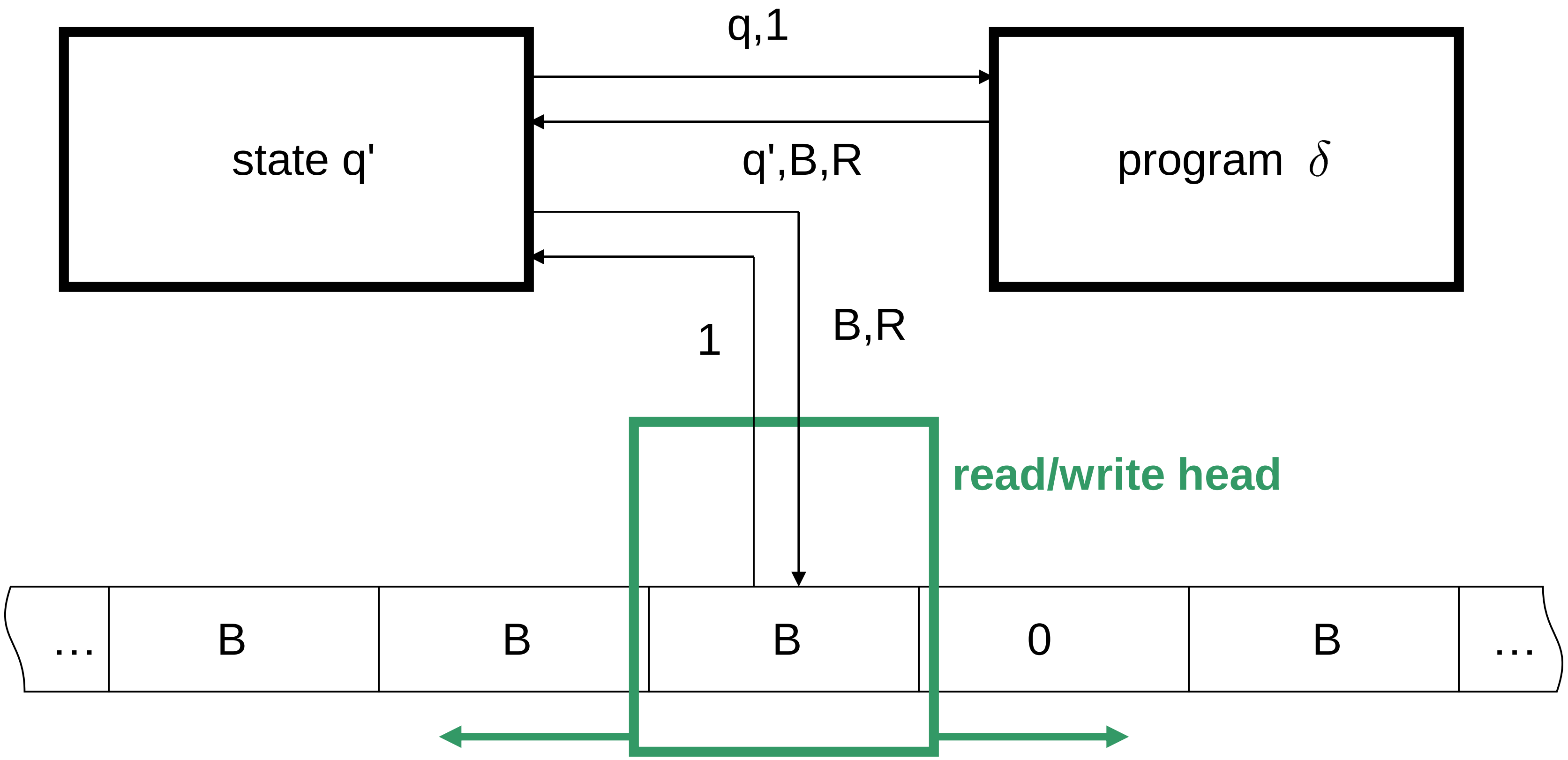
$$(Q, \Sigma, \Gamma \supset \Sigma, B \in \Gamma \setminus \Sigma, q_0 \in Q, \delta, F \subset Q)$$

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{R, L, N\}$$



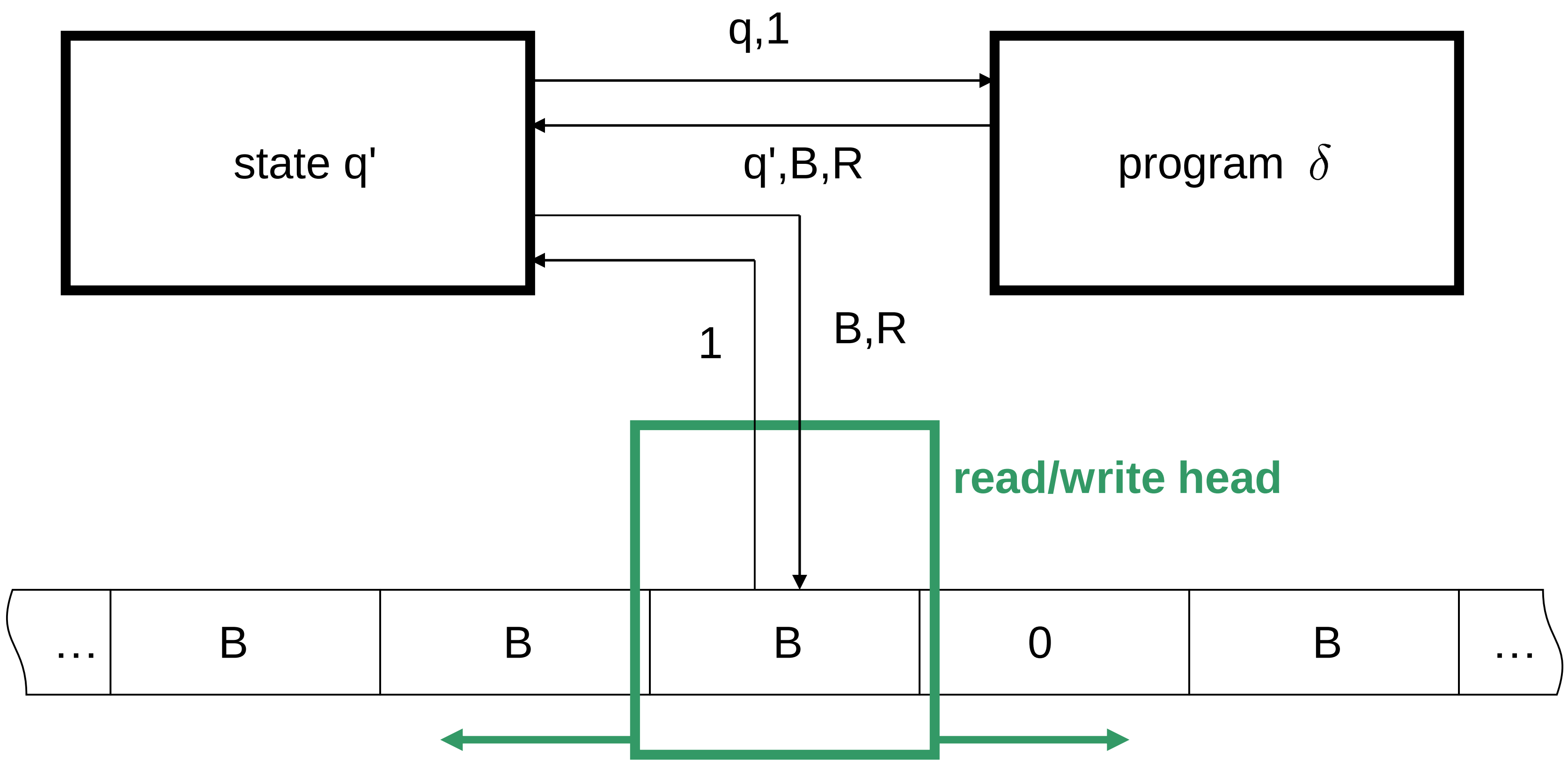
$$(Q, \Sigma, \Gamma \supset \Sigma, B \in \Gamma \setminus \Sigma, q_0 \in Q, \delta, F \subset Q)$$

$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{R, L, N\}$$



$$(Q, \Sigma, \Gamma \supset \Sigma, B \in \Gamma \setminus \Sigma, q_0 \in Q, \delta, F \subset Q)$$

$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{R, L, N\}$$



Interesting Facts

- instead of a RAM's random access computation is local
- Deterministic TM (DTM) as powerful as RAM
    - except polynomial overhead

Universal Turing machines:

- get program and data as input
- simulate δ' of the program with general transition function

Church-Turing Thesis

- Every function which would naturally be regarded as computable can be computed by a Turing machine.
- not provable
- most surprising: there are functions that are not computable (undecidable)
  - halting problem: given a program P, does the universal TM halts on P?
- related to
  - incompleteness theorem
  - Entscheidungsproblem

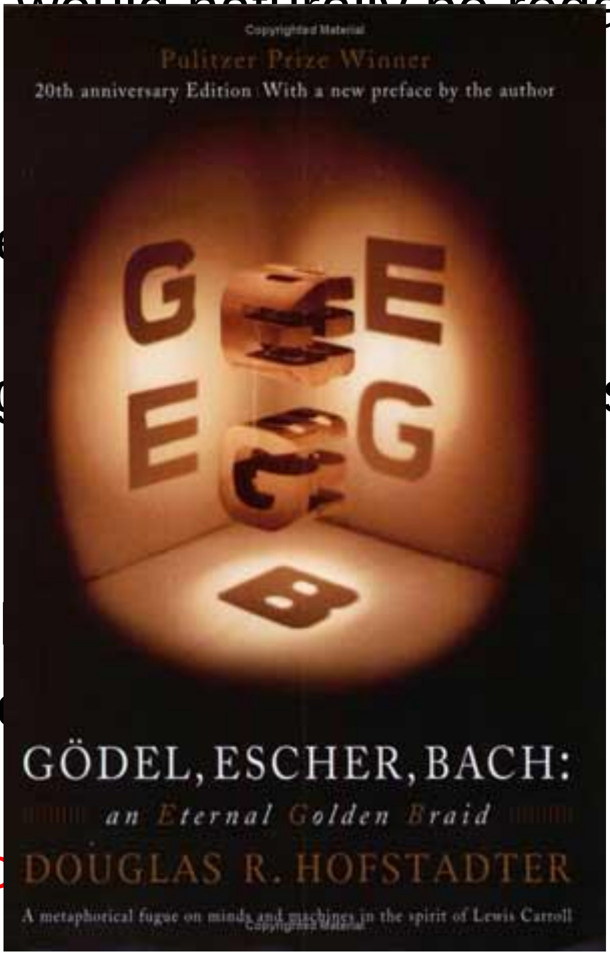
now from undecidable to decidable problems

Kurt Gödel (1906-78)

Church-Turing Thesis

- Every function which would naturally be regarded as computable can be computed by
- not provable
- most surprising: there not computable (undecidable)
  - halting problem: s the universal TM halts on P?
- related to
  - incompleteness t
  - Entscheidungspr

now from undecidable to


Copyrighted Material
Pulitzer Prize Winner
20th anniversary Edition With a new preface by the author
GÖDEL, ESCHER, BACH:
an Eternal Golden Braid
DOUGLAS R. HOFSTADTER
A metaphorical fugue on minds and machines in the spirit of Lewis Carroll
Copyrighted Material

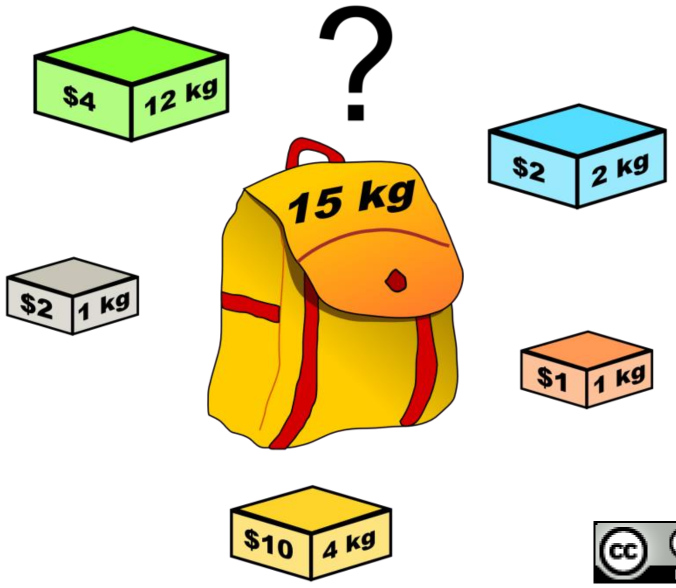
Kurt Gödel (1906-78)

Remains for today...

- complexity classes (in particular the famous P and NP)
- polynomial and Turing reductions
- hardness and completeness

What is P and NP?

- Complexity classes
- Set of problems with similar complexity
- Complexity = asymptotic running time of the best algorithm wrt. a given computation model (for the worst-case instance)
- Decision problems vs search problems vs optimization problems
  - Example: KP



Different Problem Types

Optimization problem:

find the best solution among all feasible ones!

- KP: “find packing with maximal value”

Search problem:

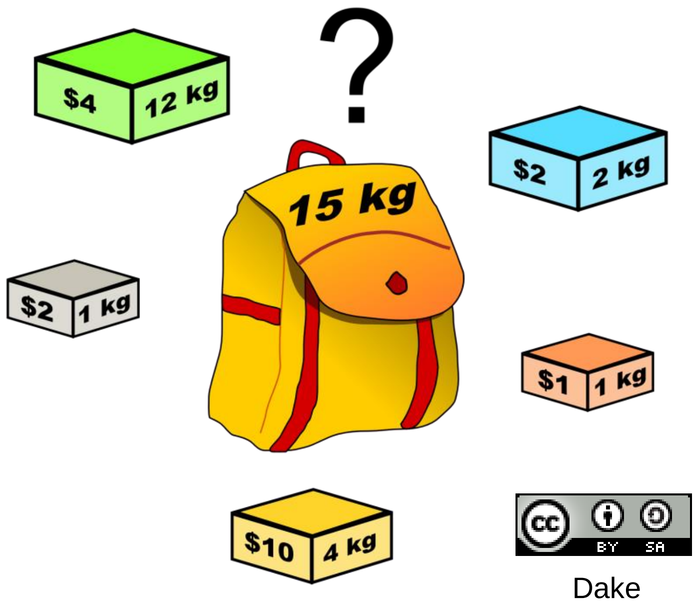
output a solution with a given structure!

- KP: “give a packing with value V”

Decision problem:

is there a solution with a certain property?

- KP: “is there a packing with value ≥V”



Dake

A decision problem is solved by a TM when it halts in an “accepting state” iff the given instance has the desired property

The Classes DTIME(t(n)) and P

$$\begin{aligned}\mathrm{DTIME}(t(n)) \quad := \quad & \{P \,|\, P \text{ is a (decision) problem} \\ & \text{s.t. there exist an algorithm } A \\ & \text{that solves } P \text{ in time } O(t(n))\}\end{aligned}$$

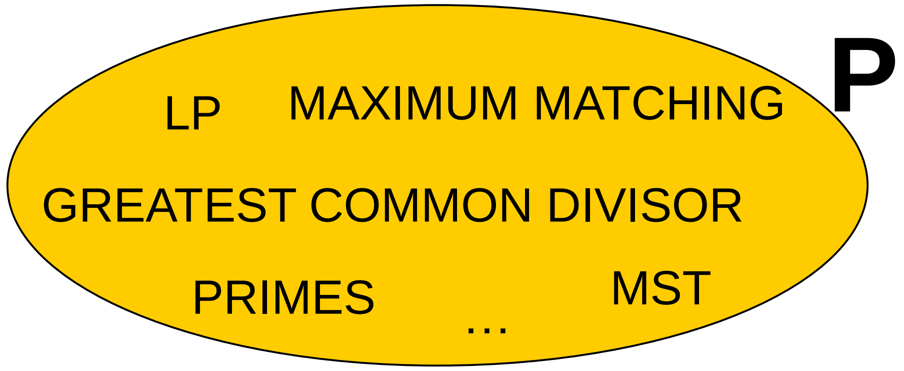
$$P = \bigcup_{k\geq 1} \mathrm{DTIME}(n^k)$$

- Why is P defined like that? And why is P important?
  - Independent of computation model

    $P_{TM} = P_{RAM} = P_{\mu\text{-}recursive\ functions} = \dots$
  - Also independent of whether the TM has
    - one or more tracks
    - one or more tapes

Intuition about P

- P is the set of all problems which have polynomial time (deterministic) algorithms
- i.e., for a given problem p2P, there exists a DTM which
  - always halts in polynomial time and
  - ends in an accepting state iff the instance belongs to p, i.e., the answer to the problem p is "yes"
- P is the set of all "efficiently solvable" or "tractable" problems
  - This set is robust against changes of the computing model
  - But also not all problems in P are *practically* solvable, e.g., if the running time is $n^{1{,}000{,}000}$



Nondeterministic Turing Machines

Deterministic TM (DTM) have a deterministic transition *function:*

$$\delta_{\text{det}} : Q \times \Gamma \to Q \times \Gamma \times \{R, L, N\}$$

Nondeterministic TM (NTM) have only a transition *relation:*

$$\delta_{\text{non-det.}} \subseteq (Q \times \Gamma) \times (Q \times \Gamma \times \{R, L, N\})$$

Which transitions will be actually performed?

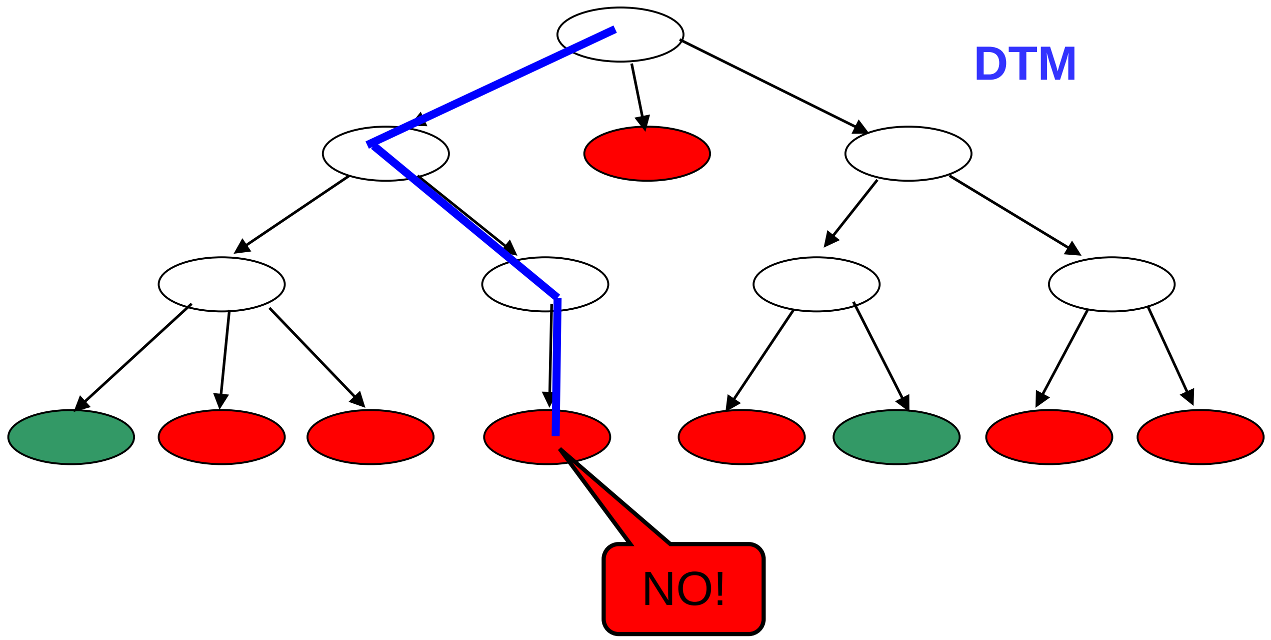
- "lucky guesser": nondet. TM guesses the right transition
- "parallel computation": nondet. TM branches into many copies and accepts if one of the branches reaches an accepting state

Nondeterminism and the Class NP

NP is the set of all problems which have polynomial time nondeterministic (!) algorithms $\mathcal{NP} = \bigcup_{k\geq 1} \mathrm{NDTIME}(n^k)$

Intuition:

- If I know a solution I can proof in deterministic polynomial time whether it belongs to the answer "yes" or "no"
- "Guess" the right solution and proof it in polynomial time

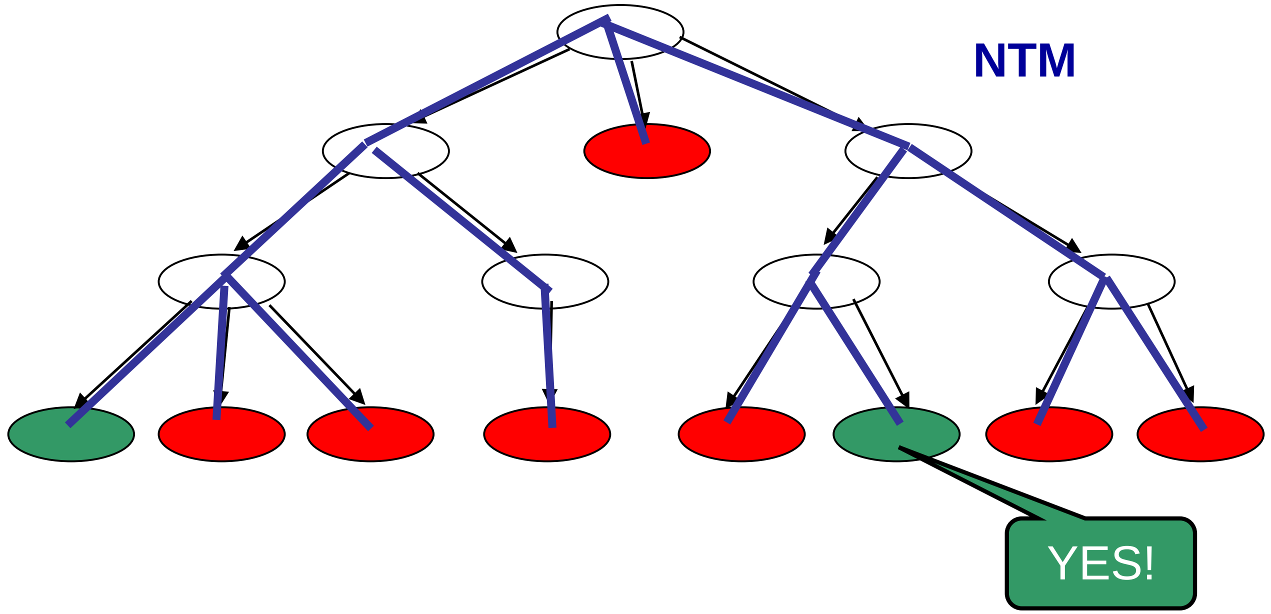


Nondeterminism and the Class NP

NP is the set of all problems which have polynomial time nondeterministic (!) algorithms $\mathcal{NP} = \bigcup_{k\geq 1} \mathrm{NDTIME}(n^k)$

Intuition:

- If I know a solution I can proof in deterministic polynomial time whether it belongs to the answer "yes" or "no"
- "Guess" the right solution and proof it in polynomial time



Problems in NP

KP

- Guess which items to choose, check that the knapsack constraint is fulfilled, and sum up all profits

TSP

- Guess a tour and sum up all edge weights

SAT

- Guess an assignment of variables and compute boolean value of the DNF

SCP

- Guess the subset, check that all items are covered, and count the number of selected sets

Bin Packing

- Guess the assignment of items to bins, check that the size restrictions are fulfilled, and count the number of bins used

Facts about P=NP Hypothesis

- Clear: PµNP
- Not clear: P½NP
- What is the difference between, e.g., KP and PRIMES?
- For PRIMES, we know a polynomial time algorithm*, for KP, we don't
- Is KP "harder to solve" than PRIMES?
- Idea: classify the hardest problems in NP
    - NP-complete problems (NPCµ NP)
    - Cook (1971), Levin (1973): SAT 2 NPC
    - Reductions

*Agrawal, Kayal, Saxena (2004): "Primes is in P", Annals of Mathematics, 160 (2004), 781–793
S. Cook (1971): "The Complexity of Theorem Proving Procedures", Proc. ACM symp. on Theory of computing, 151–158.
L. Levin (1973): "Universal'nye perebornye zadachi". Problemy Peredachi Informatsii 9 (3): 265–266.

Reductions

Idea:

if problem A can be solved by using an algorithm for problem B, then A is not harder than B (except for a polynomial overhead)

Polynomial Reduction $A \leq_p B$ (Cook, 1971)

- Transform instance of A into one for B within polynomial time by a function f
- Use oracle for B once which computes the solution for transformed instance as solution for A
- $a \in A \Longleftrightarrow f(a) \in B$

Turing Reduction $A \leq_T B$ (Karp, 1972)

- Use oracle for problem B polynomially often to compute the solution of A
- $a \in A \Longleftrightarrow f(a) \in B$

Important: both reductions are transitive!

see http://groups.csail.mit.edu/tds/papers/Lynch/stoc74.pdf

Example: DHC $\leq_p$ HC

Hamiltonian Cycle

= A cycle in a graph which visits each vertex exactly once.

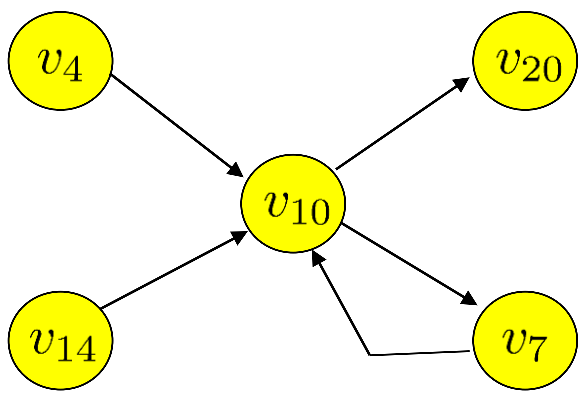
Hamiltonian Cycle Problem (HC), decision version

- given an undirected graph, is there a Hamiltonian cycle?

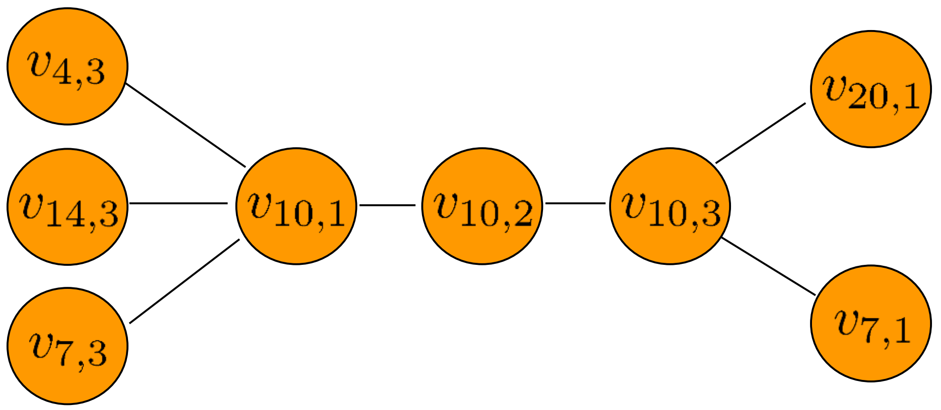
Directed Hamiltonian Cycle Problem (DHC)

- same for directed graphs

Example: DHC $\leq_p$ HC



DHC



HC

Example from I. Wegener (2003): "Komplexitätstheorie", Springer

- Transformation in polynomial time O(nm) possible
- Directed hamiltonian cycle in instance of DHC
  $\Longrightarrow$ Hamiltonian cycle in HC
- Hamiltonian cycle in instance of HC
  $\Longrightarrow$ order of HC is always ..., $v_{i,1}$, $v_{i,2}$, $v_{i,3}$, $v_{j,1}$, $v_{j,2}$, $v_{j,3}$, ... or
  ..., $v_{i,3}$, $v_{i,2}$, $v_{i,1}$, $v_{j,3}$, $v_{j,2}$, $v_{j,1}$, ...
  $\Longrightarrow$ take either HC or the inverted HC as solution for DHC $\square$

Different Types of Polynomial Reductions

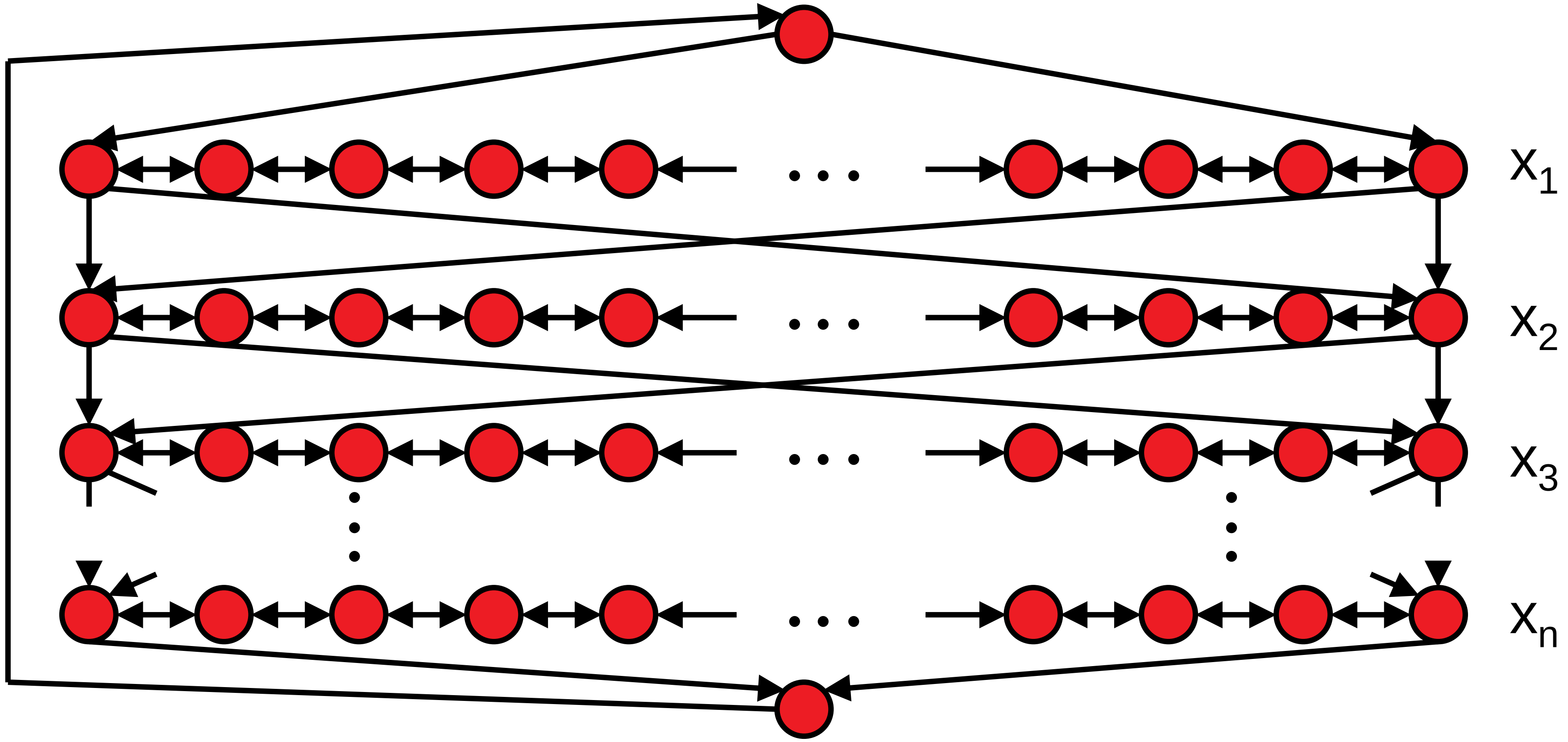
- The last example was a reduction from a special case to a general case
- Now: one slightly more complicated example (reduction from 3-SAT to DHC)
- In the exercises, we will see two more reductions

Example: 3-SAT $\leq_p$ DHC

Given a 3-SAT instance with n variables x_i and k clauses.

Construction of DHC instance:

- basic graph with 2n many Hamilton circuits (n rows, 3k+3 columns)
- intuition: set x_i to TRUE iff its row is traversed from left to right



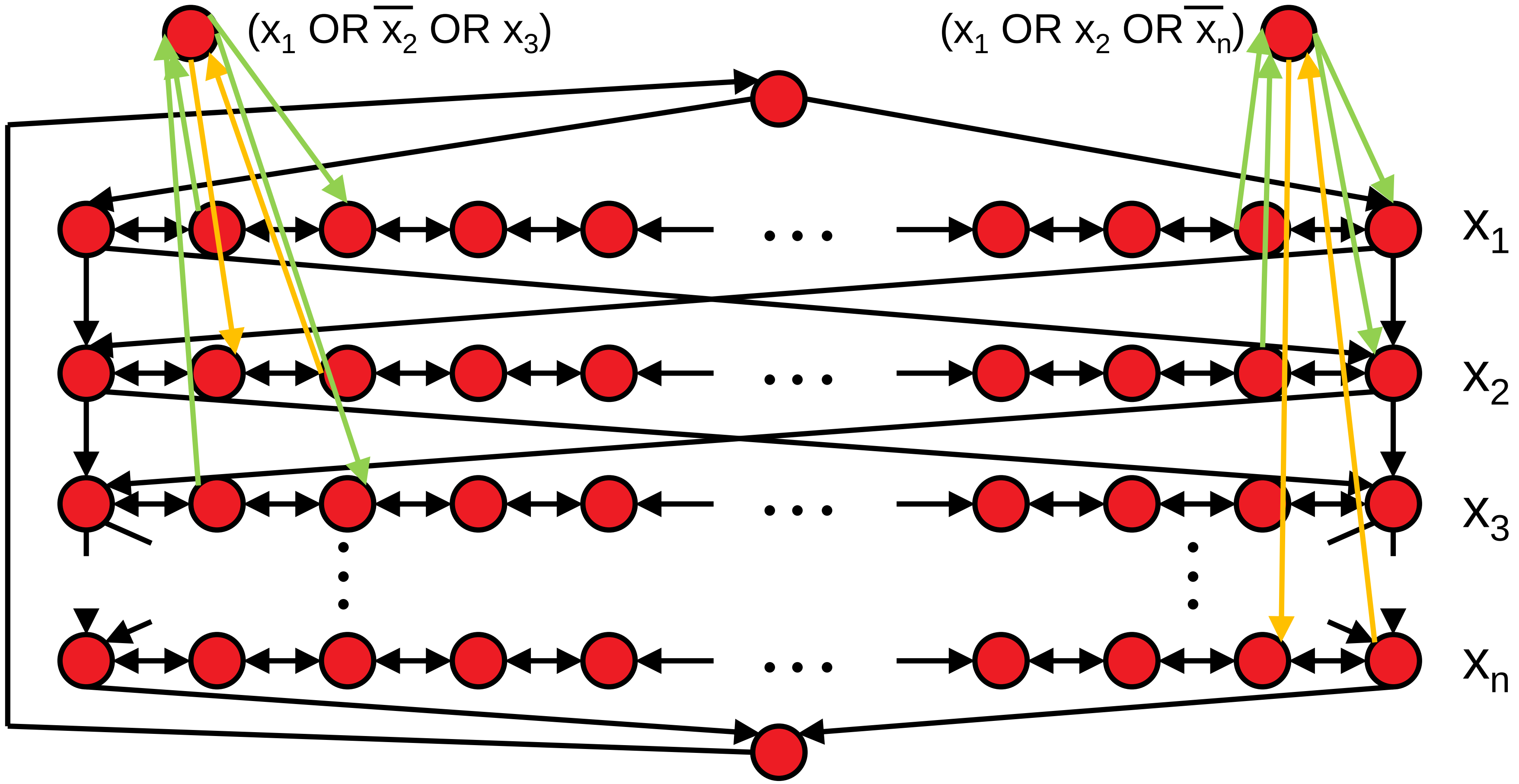
following http://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/08IntractabilityI.pdf

Example: 3-SAT $\leq_p$ DHC

Given a 3-SAT instance with n variables x_i and k clauses.

Construction of DHC instance:

- for each clause add 1 vertex and 6 edges



$(x_1 \text{ OR } \overline{x_2} \text{ OR } x_3)$
$(x_1 \text{ OR } x_2 \text{ OR } \overline{x_n})$
x_1
x_2
x_3
x_n

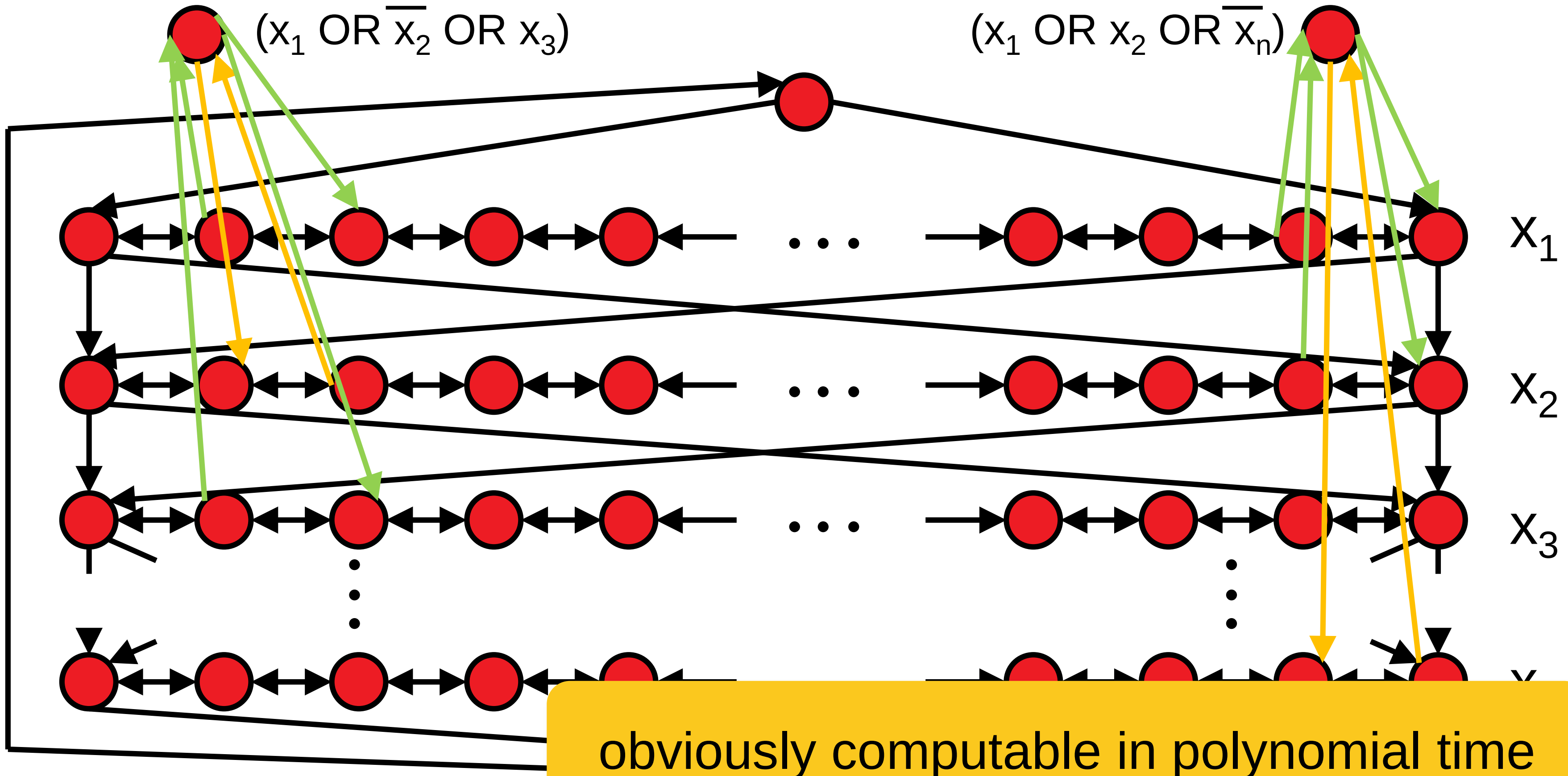
following http://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/08IntractabilityI.pdf

Example: 3-SAT $\leq_p$ DHC

Given a 3-SAT instance with n variables x_i and k clauses.

Construction of DHC instance:

- for each clause add 1 vertex and 6 edges



$(x_1 \text{ OR } \overline{x_2} \text{ OR } x_3)$
$(x_1 \text{ OR } x_2 \text{ OR } \overline{x_n})$
x_1
x_2
x_3

obviously computable in polynomial time

following http://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/08IntractabilityI.pdf

Proof of Correctness

3-SAT instance is satisfiable iff corresponding graph G has Hamilton cycle!

- let’s show “⇨” first
- assume that the 3-SAT instance has satisfying assignment x*
- construct Hamiltonian cycle in G as follows:
  - if $x^*_i = 1$, traverse row i from left to right
  - if $x^*_i = 0$, traverse row i from right to left
  - for each clause C_j, there is at least one row i in which we are going in "correct" direction to insert the corresponding C_j vertex into the tour (we do this only once per clause vertex)

following http://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/08IntractabilityI.pdf

Proof of Correctness

3-SAT instance is satisfiable iff corresponding graph G has Hamilton cycle!

- now, let us see "⇦"
- assume a Hamiltonian cycle H in G
- by construction, it has to visit node C_j from and to the same row
- replacing the part of H through C_j by the edge in between its neighbors defines a Hamilton cycle on $G \backslash C_j$
- doing this for all C_j allows to assign $x^*_i = 1$ if row i is traversed fully from left to right and $x^*_i = 0$ otherwise
- now since H traverses the clause vertex C_j originally, at least one of the paths through it is traversed in "correct" order and each clause is satisfied

□

following http://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/08IntractabilityI.pdf

The Class NPC

- NPC: set of all NP-complete problems
- The "hardest problems in NP"
- A is NP-complete if
  - A2NP
  - All problems A_{NP}2NP can be polynomially reduced to A:

$$\forall A_{\mathcal{NP}} : A_{\mathcal{NP}} \leq_p A$$

- NP-complete problems are the hardest of the ones in NP in the sense that if I can solve them in polynomial time, I can solve all problems in NP in polynomial time

Proving NP-completeness

How to prove that a problem A is NP-complete?

- Two possibilities:
  - Either prove A2NP and for all problems in NP that they can be reduced to A (complex, see Cook (1971)) or
  - Prove A2NP (simple) and a reduction from a problem B that is already known as NP-complete to A (!)

caveat: be careful of the order in the reduction!

The Cook-Levin Theorem

Theorem: 3-SAT $\in$ NPC

- proven by Cook in 1971 and independently (with a slightly different proof) by Levin in 1973
- not enough time here for the detailed proof

But idea easy to understand:

- 3-SAT $\in$ NP trivial
- Given any problem p $\in$ NPC and an instance i to that problem, construct a Boolean formula which is satisfiable iff the non-deterministic TM for p accepts instance i
- Variables for states of the TM, e.g. $T_{i,j,k}$ = true if tape cell *i* contains symbol *j* at step *k* of the computation
- Polynomially many variables and Boolean statements enough because the TM runs in polynomial time

Exercise:
Two Example Reductions

`http://researchers.lille.inria.fr/~brockhof/introoptimization/`

Example: HC $\leq_p$ TSP

Observation: Hamilton Cycle Problem is a subproblem of TSP

Transformation:

Simulate same graph for TSP as the one given for HC

- Full graph actually, but weight 1 for each edge in HC graph and weight 2 for each „non-edge“ in HC
- Asking the TSP oracle whether a weight |V| tour exists

Correctness:

- If H is a Hamilton cycle in original graph, it is also a cycle through all cities but with weight ≤|V|
- Let T be a tour in the (transformed) TSP instance with weight ≤|V|. It cannot contain an edge with weight 2. Hence, the cycle T is also a cycle in the original HC problem.

Example: VERTEX COVER $\leq_p$ CLIQUE

Observation: vertex cover in G=(V,E) of size k = clique in complementary graph G_C=(V, ExE \ E) of size |V|-k

Transformation:

- change each edge in „non-edge“ and vice versa
- use |V|-k as threshold for CLIQUE if VERTEX COVER of size k is asked
- obviously polynomial time

Correctness: first „⇨ “

- let V′ be a vertex cover of size k, i.e. for each edge (u,v) either u or v (or both) is in V′
- by definition, then for each pair u,v which are both not in V′ (and thus in V\V′): the edge (u,v) is not contained in G („contraposition“)
- but then all those edges are contained in G_C and V\V′ is a clique

Example: VERTEX COVER $\leq_p$ CLIQUE

Observation: vertex cover in G=(V,E) of size k = clique in complementary graph G_C=(V, ExE \ E) of size |V|-k

Transformation:

- change each edge in „non-edge“ and vice versa
- use |V|-k as threshold for CLIQUE if VERTEX COVER of size k is asked
- obviously polynomial time

Correctness: now „⇦“

- let V′ be a clique of size n-k in G_C
- if (u,v) is an edge in G, then both u and v can‘t be in V′ at the same time because V′ is clique in G_C
- but then either u or v is in V\V′ which means that V\V′ is a vertex cover

Difference between NP-complete and NP-hard

A is NP-complete if

- $A \in \mathcal{NP}$
- $\forall B \in \mathcal{NP} : B \leq_p A$

A is NP-hard if

- $\forall B \in \mathcal{NP} : B \leq_T A$

Implications:

- An NP-hard problem is not necessarily a decision problem
- The search and optimization versions of an NP-complete problem are NP-hard

Practical Implications of Reductions

The proof of NP-completeness is typically seen as a proof of difficulty:

“I did not find an efficient algorithm for my problem, maybe I am dumb?”

vs.

“I cannot find an efficient algorithm for my problem because there is none”

vs.

“I did not find an efficient algorithm for my problem but neither all of those famous people”

But...

Having a proof of NP-completeness or NP-hardness, does not mean that a problem is not manageable in practice:

- the average-case complexity might be reasonable
- randomized algorithms might work well
- maybe, the difficult instances are not observed

Example of success: SAT solvers

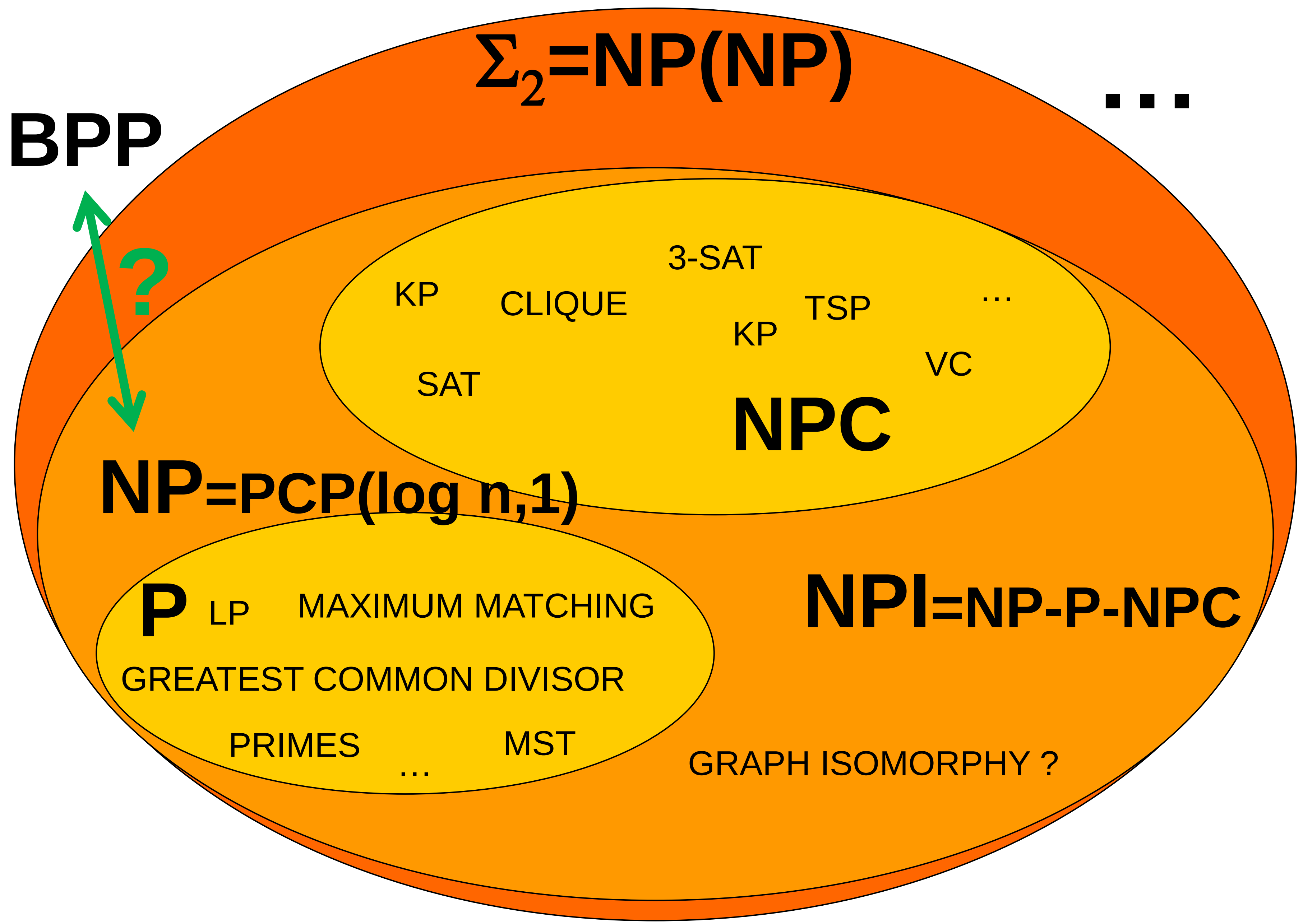
The Famous P versus NP Problem

Is P=NP?

- One of the 7 Millennium Prize problems selected by the Clay Mathematics Institute (worth 10^6 $)
- first mentioned in 1956 in letter from K. Gödel to J. von Neumann
- formalized by J. Cook in his 1971 seminal paper
- solving this problem might have significant practical implications (or not)

what do you think?

The „Complexity Zoo"



Conclusions

I hope it became clear...

...what complexity theory is about

...what is a Random Access Machine and a Turing Machine

... how a decision and an optimization problem differ

...what are the classes P, NP, and NPC

...and that complexity theory is more involved than what we could see today