The classes P and NP

Decision problem

- Problem with only two possible answers „yes“ or „no“
- Examples: Is n a prime number? Does a solution path in the Solitair-game exist?

Optimization problem

- given.: an implicitely or exolicitely described set Ω of possible solutions
  and an evaluation function $f : \Omega \rightarrow IR$.
  wanted: a solution x with $f(x) = \max\{ g(x) \mid x \in \Omega \}$
- Examples: Find a best possible fleet assignment.

Decision- and Optimization problems can be transformed to each other.

The classes P and NP

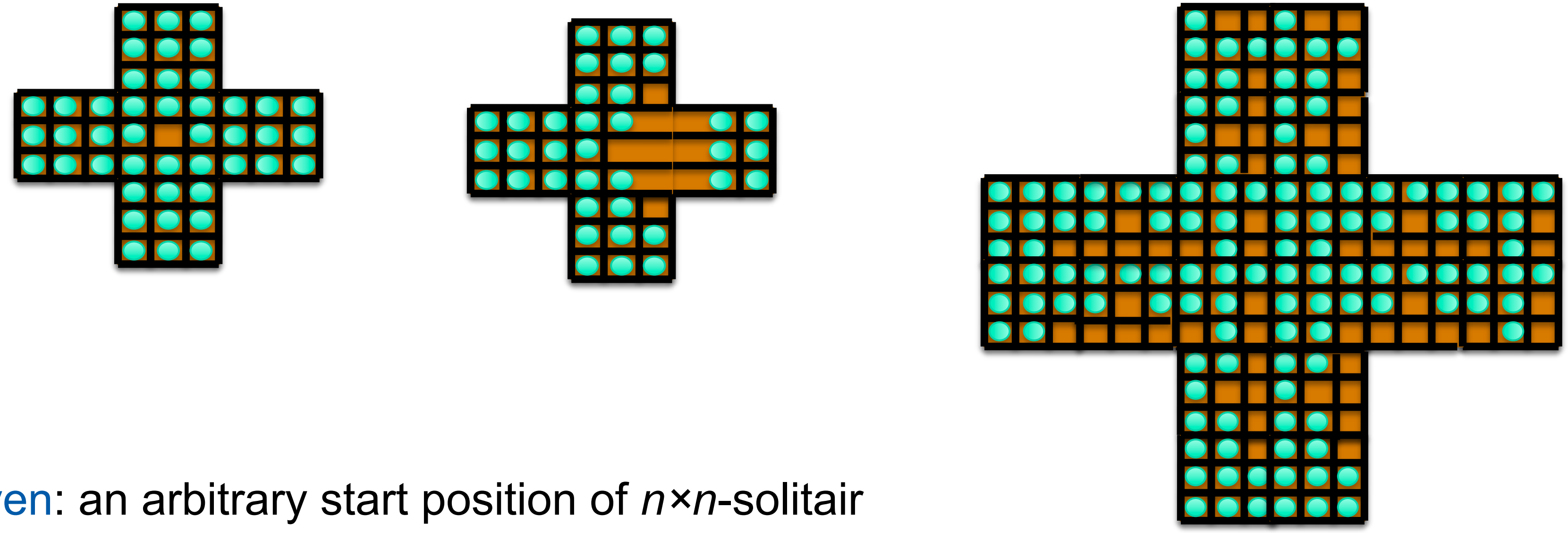
The class *P*: informal description

- set of those Decision problems, for that an algorithm exists, which solves the problem and which consumes no more than polynomial runtime.

The class P: formal definition

- Let an encoding scheme *E* and a computational model *M* be given.
- Let *Π* be a decision problem, and let each instance be encoded with the help of the encoding scheme *E.*
- *Π* belongs to the class *P* (with regard to *E* and *M*), if there is an on M implementable algorithm that solves all instances of *Π* , with a worst-case runtime function which is bounded by a polynomial.

The classes P and NP, examples



given: an arbitrary start position of *n*×*n*-solitair
wanted: yes/no with yes, if more than half of the stones have left the board.

→ simple
→ in P

given: an arbitrary start position of *n*×*n*-solitair
wanted: yes/no with yes, when tit is possible to play in such a way that exactly one stone remains in the middle.

→ intuitively not that easy
→ in „NP“

The classes P and NP

***NP*, definition 1:**

A decision problem Π belongs to class *NP*, if it is valid:

- For each instance $I \in \Pi$ with answer „yes“, there is (at least) one object Q that helps to verify the answer „yes“.
- There exists an algorithm which accepts an instance $I \in \Pi$ and an additional object *Q* as its input and verifies the answer „yes“ with runtime polynomial in $<I>+<Q>$.

- No statement how Q is computed. Q can be guessed by an oracle.
- The only statement for „no“ instances is that there has to be an algorithm which correctly outputs „yes“ or „no“ in finite time.

The classes P and NP

TECHNISCHE UNIVERSITÄT DARMSTADT

***NP*, definition 2 (equivalent to previous one):**
The class *NP* is defined via a so called non-deterministic RAM. Such a machine possesses an additional instruction „goto L1 or goto L2;“.

A problem Π is in *NP* if there is a (non-deterministic) algorithm *A* (for the non-deterministic RAM) such that for any instance $I \in \Pi$ with answer „yes“ there is a computation-path of polynomial length in $<I>$. A must halt on all instances.

P, NP, PSPACE

TECHNISCHE UNIVERSITÄT DARMSTADT

- **P:** Class of problems which can be solved with he help of a deterministic RAM in polynomial time

- **NP:** Class of problems which can be solved with the help of a non-deterministic RAM in polynomial time.

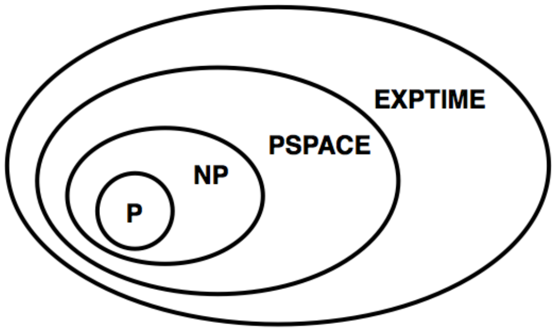
- **PSPACE :** Class of problems which can be solved with the help of a deterministic RAM with no more than polynomial space

$$\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} \subseteq \mathbf{EXPTIME}$$

- Only known: P ≠ EXPTIME and

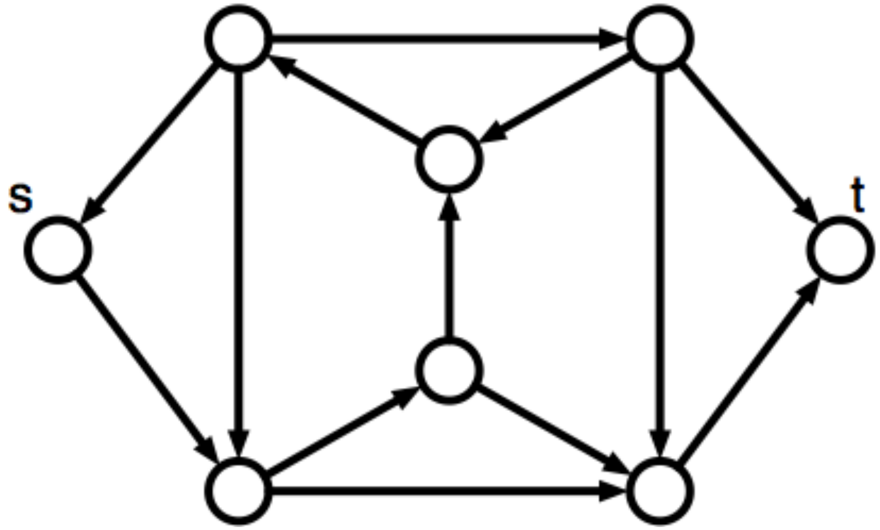
- EXPTIME = $\bigcup_k \mathbf{TIME}(2^{n^k})$

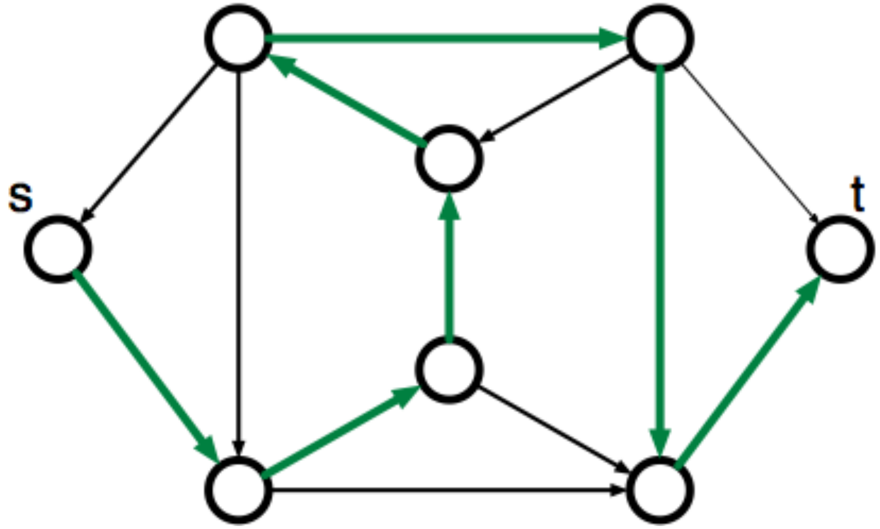
- Most researchers assume that the inclusions are strict.



Typical examples from NP

- **Definition: *HAMPATH***
  - The Hamiltonian path problem
    - given.:
      - a directed graph
      - two nodes s,t
    - wanted.: does a path from s ti t exist, such that all nodes are visited once, but no edge twice?
- **Algorithm for Hamiltonian path:**
  - Guess a permutation $(s,v_1,v_2,...,v_{n-2},t)$
  - Check, whether the permutation describes a path
    - If yes, do accept
    - If no, throw it away
- **Therefore: HamPath $\in$ NP**





Typical examples from NP

TECHNISCHE UNIVERSITÄT DARMSTADT

The SAT problem

- A boolean function $f(x_1,x_2,..,x_n)$ is satisfiable, if there is an assignment for $x_1,x_2,..,x_n$ such that $f(x_1,x_2,..,x_n) = 1$
    - $(x \vee y) \wedge (z \vee \neg x \vee \neg y) \wedge (x \vee \neg z)$ is satisfiable, because
        - the assignment $x = 1$, $y = 0$, $z = 0$
    - delivers $(1 \vee 0) \wedge (0 \vee 0 \vee 1) \wedge (1 \vee 1) = 1 \wedge 1 \wedge 1 = 1$.

- Definition (SAT problem, the origin of all NPc problems)
    - **Given**:
        - Boolean Function ϕ
    - **Wanted**:
        - Is there $x_1,x_2,..,x_n$ such that $\phi(x_1,x_2,..,x_n) = 1$

- SAT is in NP. It is supposed that SAT is not in P.

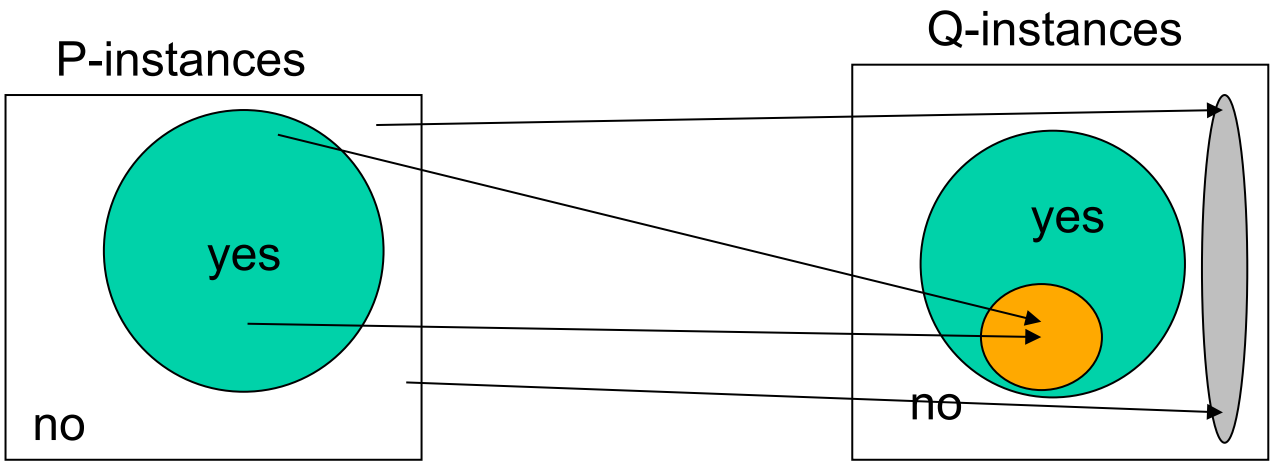
Classification of problems in P, NP, PSPACE

The reduction technique

Definition: Let *P and Q* be problems. Let L_P (or L_Q) be the set of Instances of the problem *P* (or *Q*) with answer „yes“. Additionally, let Σ be an alphabet for problem encoding and Σ^* the set of all possible strings over the alphabet. *P* is said to be polynomialy reducible to Q ($P \leq_p Q$) if there is an in polynomial time computable function $f\colon \Sigma^* \rightarrow \Sigma^*$ such that

$$x \in L_P \Leftrightarrow f(x) \in L_Q$$

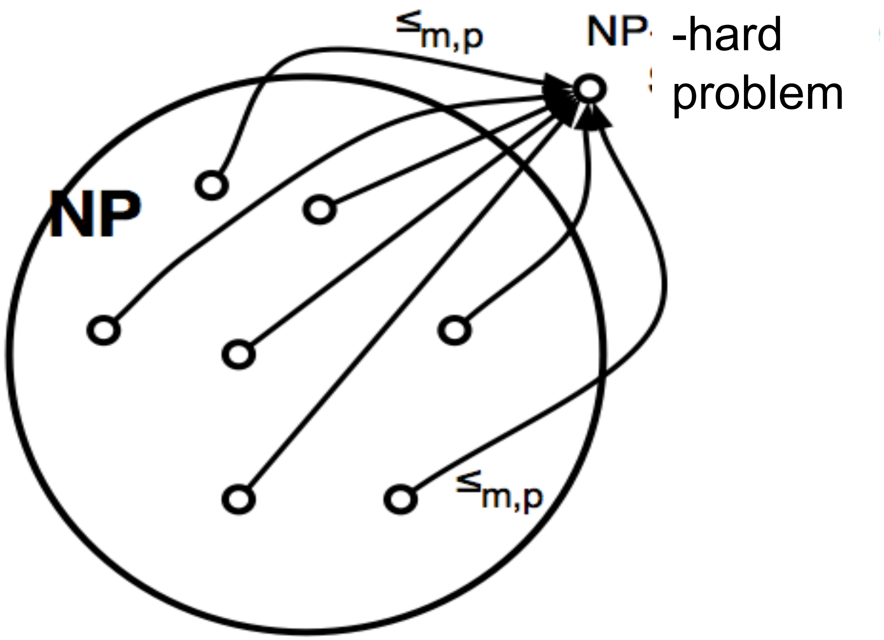
E.g.:



NP-hardness

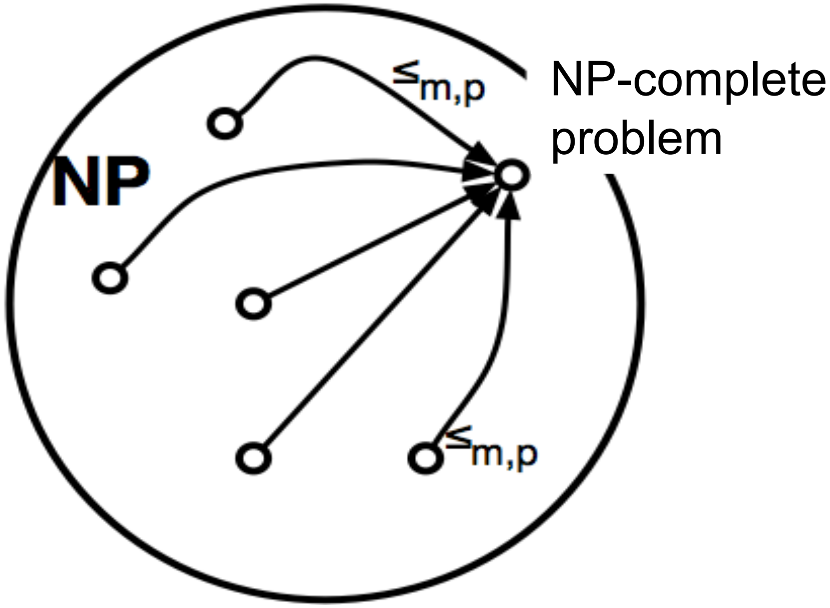
TECHNISCHE UNIVERSITÄT DARMSTADT

- Definition:
  - A problem S is called **NP-hard** if:
    - every problem from NP can be reduced to S with the help of a polynomial time reduction, i.e.
    - for all L ∈NP: $L \leq_p S$
- Theorem
  - if any NP-hard problem is in P, it will P=NP
- Proof
  - If S ∈ P and for all L: $L \leq_p S$ → L ∈ P.



NP-completeness

- Definition:
  - A problem S is **NP-complete** if:
    - $S \in NP$
    - S is NP-hard
- Corollary:
  - If any NP-complete problem is in P, it will hold P=NP
- Proof:
  - Follows from NP-hardness of an NP-complete problem.



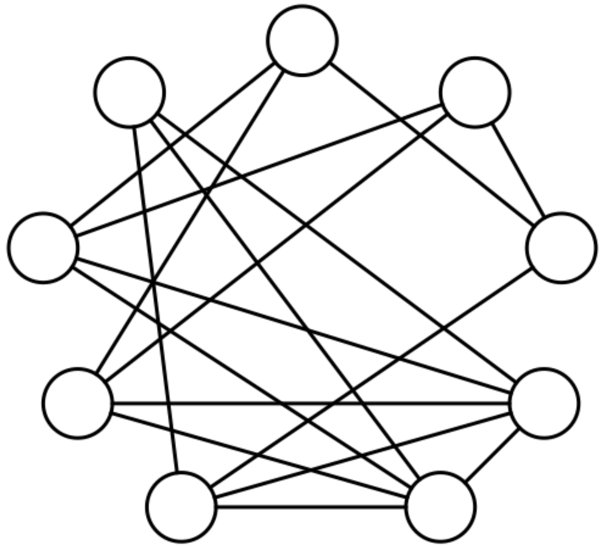
The 3-SAT-problem and the Clique-problem

$$\psi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$

- 3-SAT:
  - **Given**:
    - A boolean formula in 3-CNF
  - **Wanted**:
    - A satisfying assignment

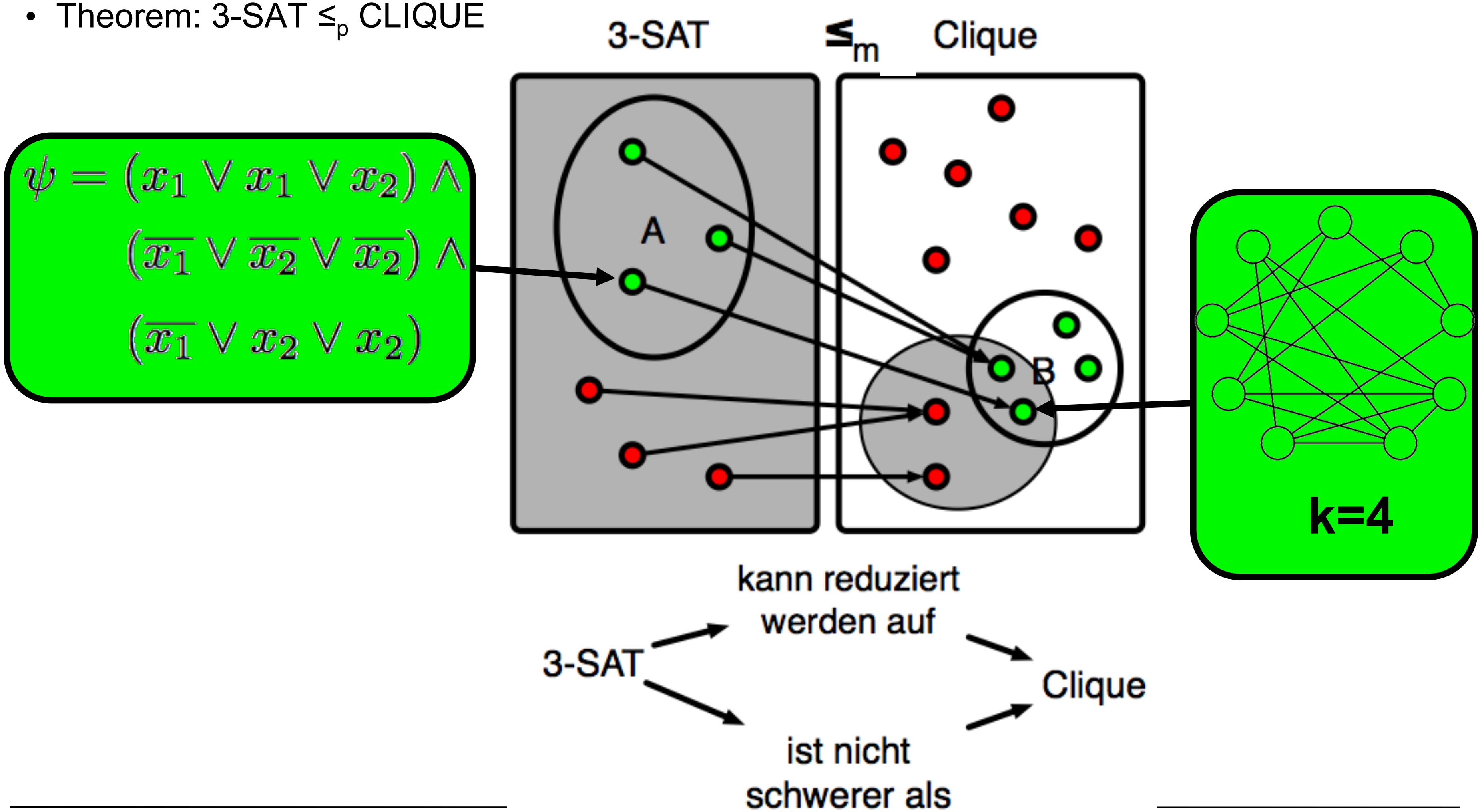
- Definition k-clique
  - An undirected graph Graph G=(V,E) contains a k-clique,
    - If it contains k nodes , such that
    - Each of the k nodes is connected with each other one in G
- CLIQUE:
  - **Given**:
    - An undirected graph G
    - A natural number k
  - **Wanted**:
    - Does G contain a clique of size k?

k=4



3-SAT can be reduced to clique

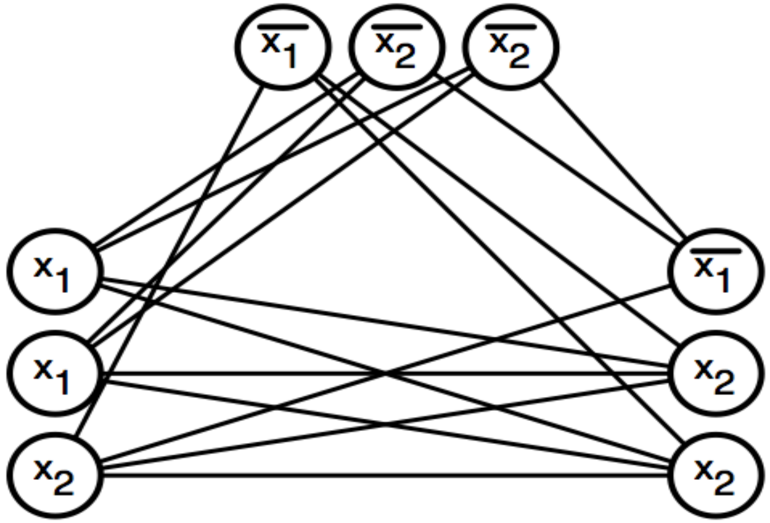
- Theorem: 3-SAT $\leq_p$ CLIQUE



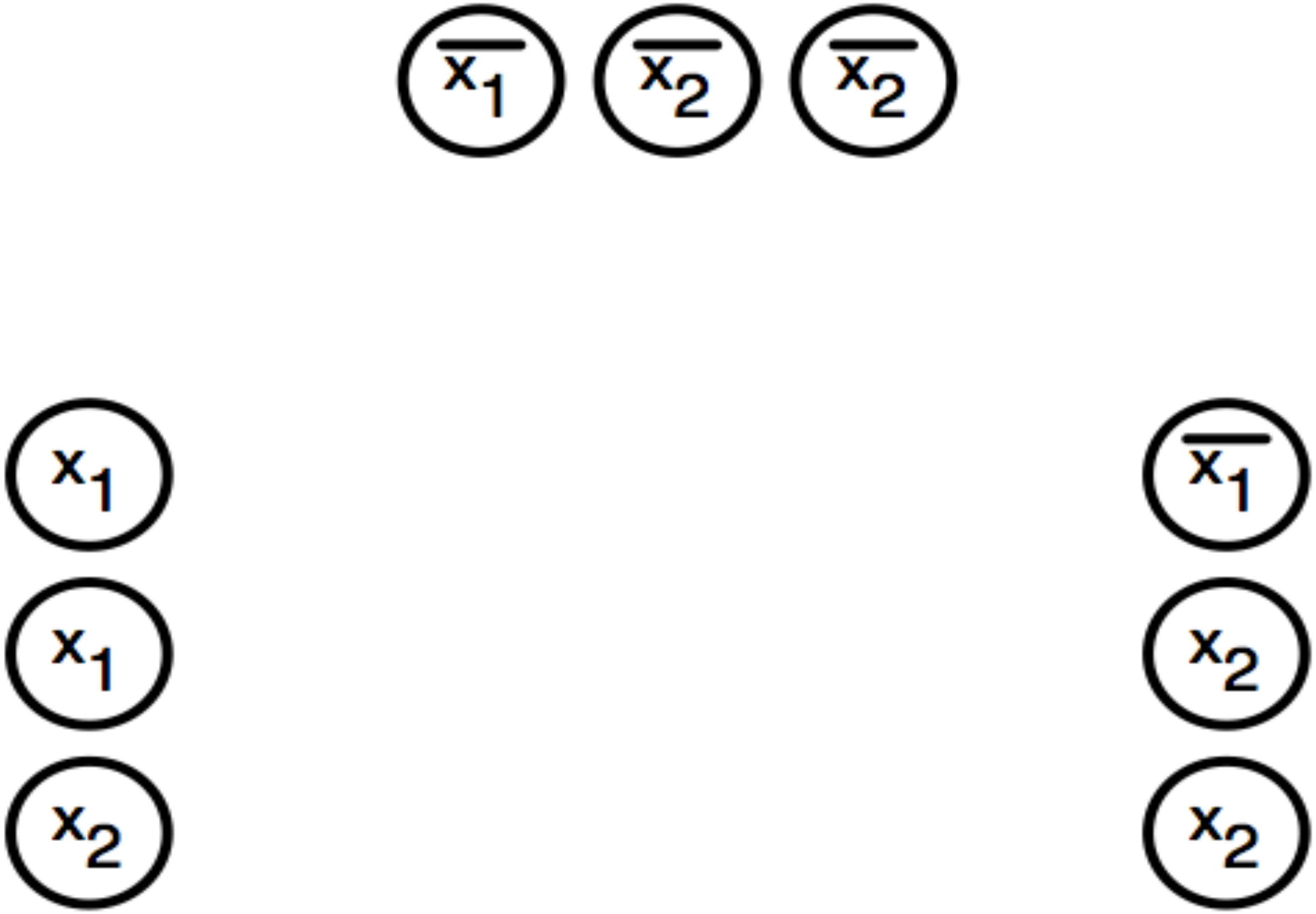
3-SAT läßt sich auf Clique reduzieren

TECHNISCHE UNIVERSITÄT DARMSTADT

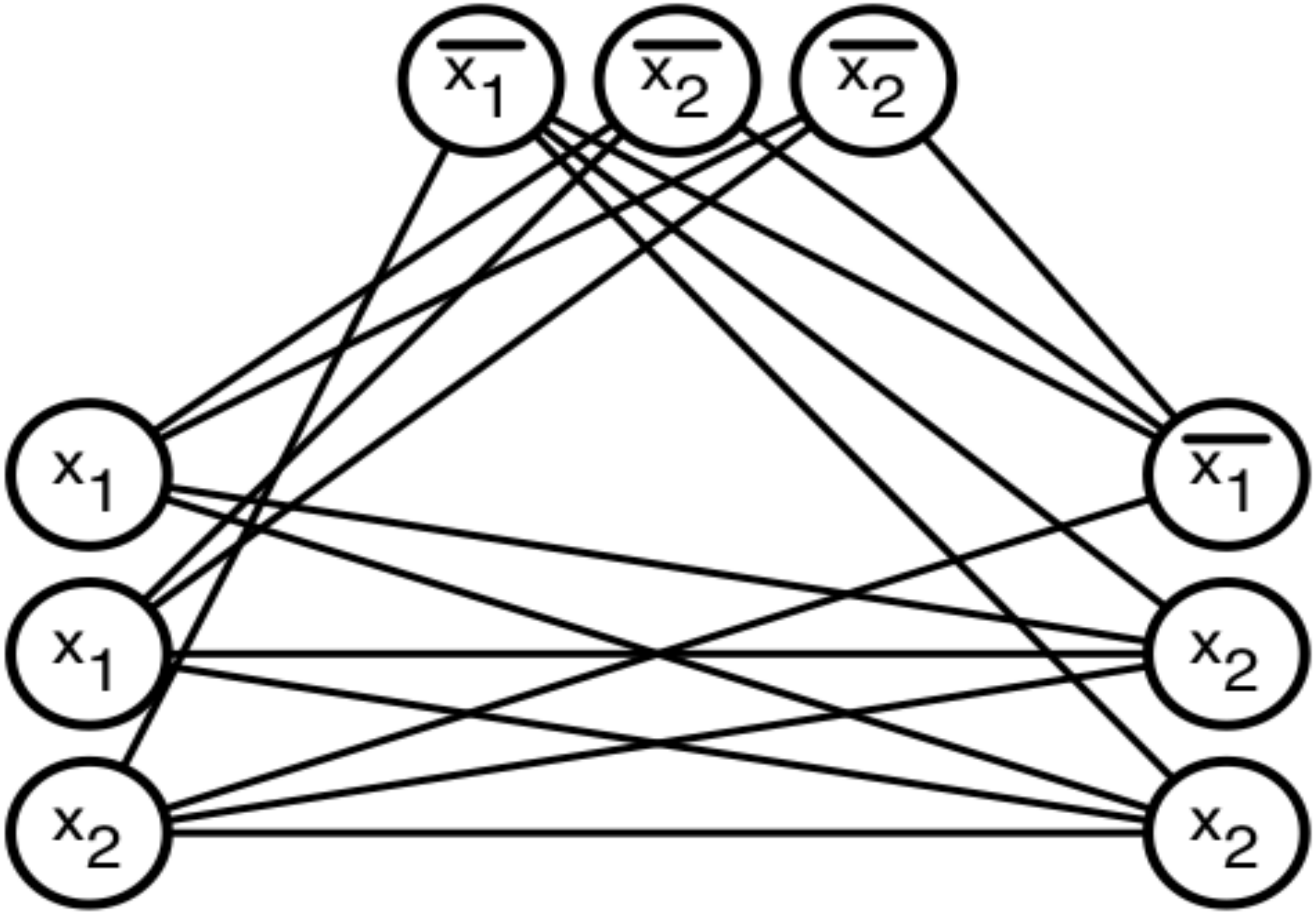
- Theorem: 3-SAT $\leq_{m,p}$ CLIQUE
- Proof
  - Construct a reduction function f as follows :
  - $f(\phi)$ = <G,k>
  - k = number of clauses
  - For each clause C in ϕ, 3 nodes are created, assigned with the names of the literals of that clause
  - Add an edge between a pair of nodes if and only if
    - The two nodes do not belong to the same clause and
    - The two nodes do not correspond to the same variable, once negated and once not.



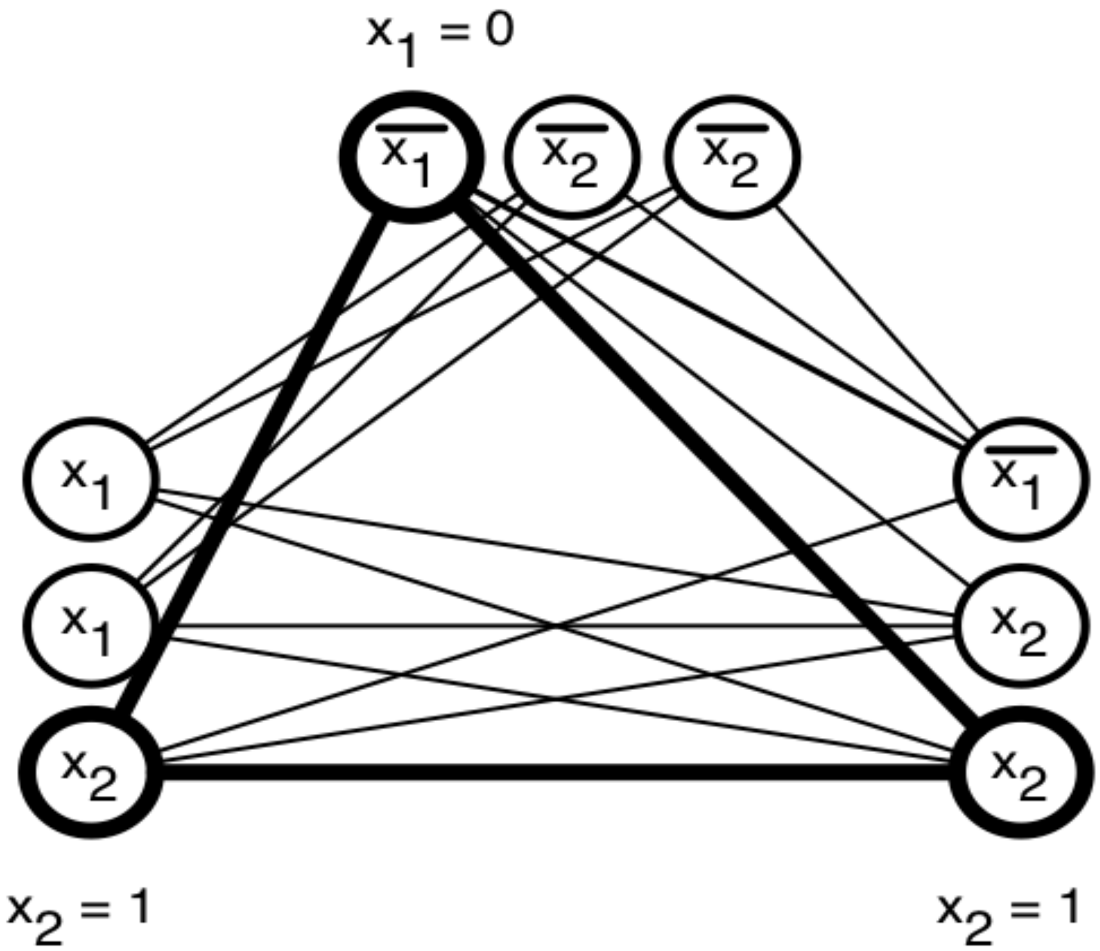
$$\psi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$



$$\psi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$



$$\psi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$



$$\psi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$

0	0	**1**	**1**	0	0	**1**	**1**	**1**

Correctness

- The reduction function is correct:
- Claim;
  - There is a true assignment of variables in ϕ if and only if there is a k-clique in G
- 1. case: a true assignment exists in ϕ
  - Then, this assignment forces at least one literal to true, in each clause
  - Choose such a literal from the node set for all clauses
  - The chosen node set then consists of k nodes
  - Between all these nodes exists an edge, because a variable and its negation cannot be both true
- 2. case: a k-clique exists in G
  - Each node of the clique belongs to another clause
  - Set the corresponding literals to true 1
  - Determine the corresponding variables
  - No contradiction occurs, because there is no edge between any literal and its negation
- runtime:
  - Construction of the graph and the edges consume no more than quadratic time.