"I can't find an efficient algorithm, but neither can all these famous people."

Algorithms for Decision Support

Computational Complexity

How hard is optimization?

Outline

- Definitions
- **P** vs. **NP**
- **NP-completeness**
  - Why is it relevant?
  - What is it exactly?
  - How is it proven? **Reduction**
- More reductions
- Just a few animals from the complexity zoo
- Exercises

Today's Lecture

- Definitions
- **P** vs. **NP**
- **NP-completeness**
  - Why is it relevant?
  - What is it exactly?
  - How is it proven? **Reduction**
- 3-SAT

How Hard is a Problem?

How Hard is a Problem?

from computer's aspect

How Hard is a Problem?

from computer's aspect

In order to solve some problem, how many resources is needed?

How Hard is a Problem?

from computer's aspect

In order to solve some problem,
how many resources is needed?

time, memory, I/O access, information……

How Hard is a Problem?

from computer's aspect

In order to solve some problem,
how many resources is needed?

time, memory, I/O access, information……

Definition: Problem

- A *problem* just contains a description of the problem together with the parameters that describe the input of the problem.
- **Values have not been assigned to the parameters.**

- For example: the **PARTITION** problem
  - Given n non-negative integral values $a_1, a_2, \cdots a_n$ does there exist a subset S of the index-set $\{1,2,\cdots,n\}$, such that

$$\sum_{j \in S} a_j = \frac{1}{2} \sum_{j=1}^{n} a_j$$

  - The input parameters are n and $a_1, a_2, \cdots a_n$

Different versions of problems

- Decision problems.
  - Answer is yes or no.
- Optimization problems.
  - Answer is a number representing an objective value.
- Construction problems.
  - Answer is some object (set of vertices, function, …).
- Counting problems.
  - How many objects of some kind exists?

First focus on decision problems

Decision vs optimization problems

- An optimization problem can be turned into a decision problem by introducing a threshold value *y*.
- In case of a minimization problem *M* , the decision variant becomes:
  - Given an instance of *M* together with a threshold value *y* , does there exist a feasible solution with outcome value ≤y?

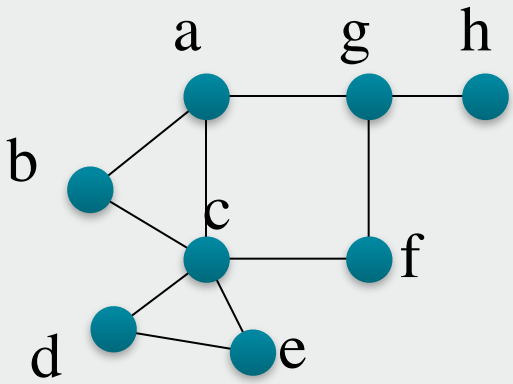
Definition: Instance

- A description of the problem together with the parameters that describe the input of the problem.
- **Values have been assigned to the parameters.**

Definition: Input size

- The **input size** is the number of elements (e.g. bits) in the formal encoding
- For the Partition problem, you must encode n together with the numbers $a_1, a_2, \cdots a_n$.
- If you use a **binary encoding**, then you need approximately $\log A$ bits with value 0 or 1 encode an integer A.
- The input size of an instance of **PARTITION** is then something like $\mathrm{n} \log(a_{max})$.
- The input size is used for measuring the running time of an algorithm

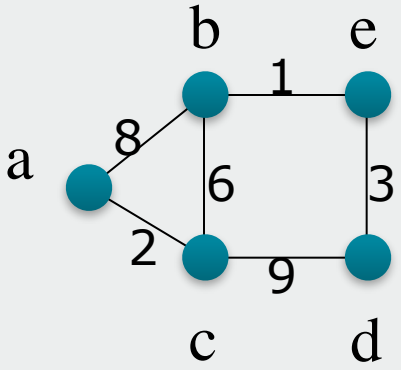
Binary encoding problem instances



0	1	1	0	0	0	1	0
1	0	1	0	0	0	0	0
1	1	0	1	1	1	0	0
0	0	1	0	1	0	0	0
0	0	1	1	0	0	0	0
0	0	1	0	0	0	1	0
1	0	0	0	0	1	0	1
0	0	0	0	0	0	1	0

If there are n vertices, what is the input size?

Binary encoding problem instances



	8	2		
8		6	9	1
2	6		9	
		9		3
	1		3	

If there are n vertices, what is the input size?

Running time of an algorithm

- Suppose we have chosen some problem, and for this problem we have defined an algorithm to solve it.
- We want to have a (rough) estimate of the number of elementary computations (additions, multiplications, comparisons, etc.) that are required by the algorithm to solve the problem *in the worst case*
- This expression will depend on the values of the parameters input size.

Definition: Polynomial versus exponential

- An algorithm with running time $O(n^k)$, where k is a given constant, is called a **polynomial** algorithm.
- An algorithm with running time, $O(c^n)$ where $c>1$ is a given constant, is called an **exponential** algorithm.

Practice

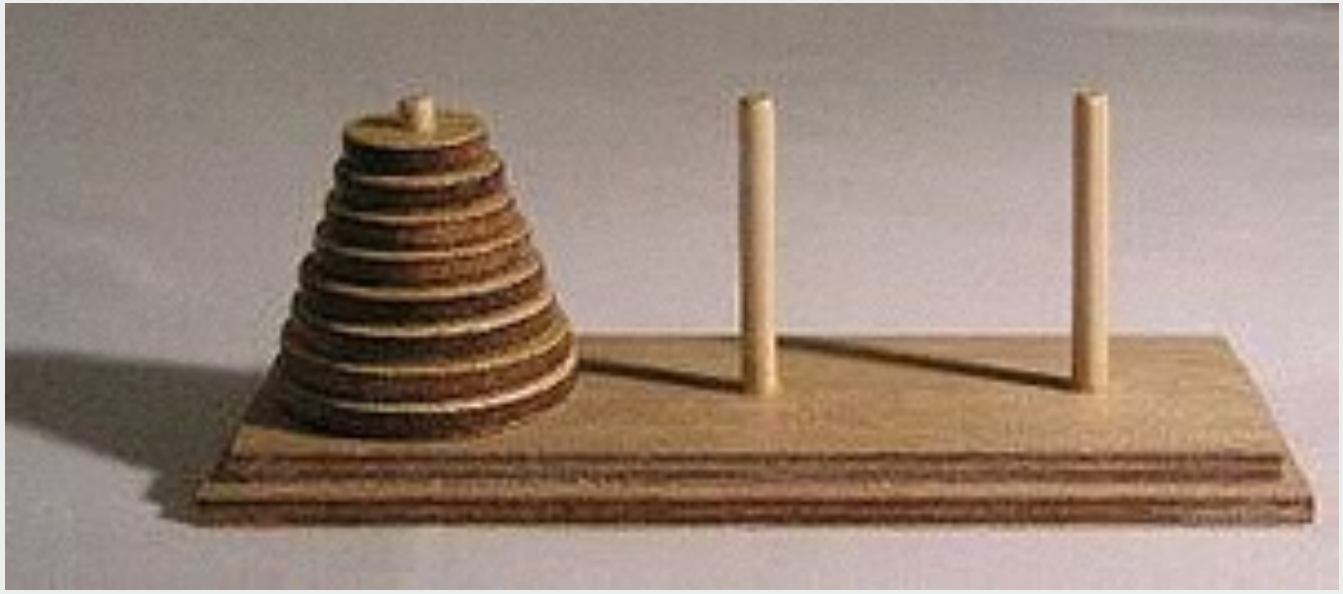
- Recall some **GRAPH** problem:
  - n vertices

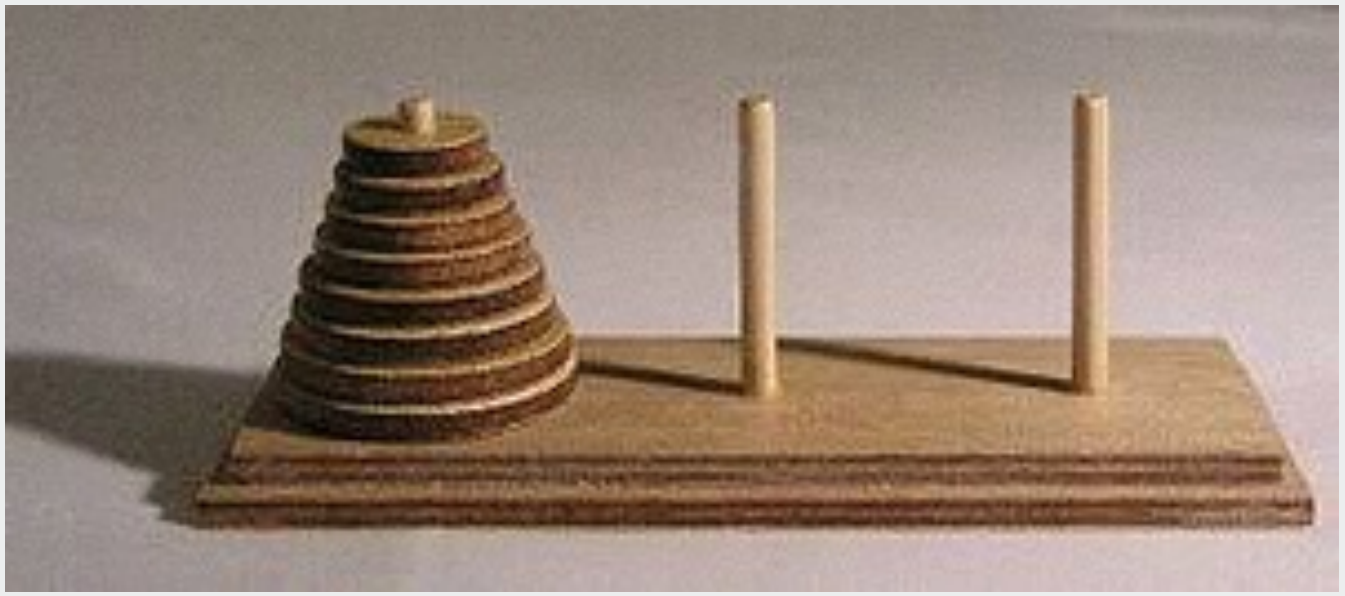
Suppose there is an algorithm with running time $O(n^3)$. Is it a polynomial algorithm or an exponential one?

Practice

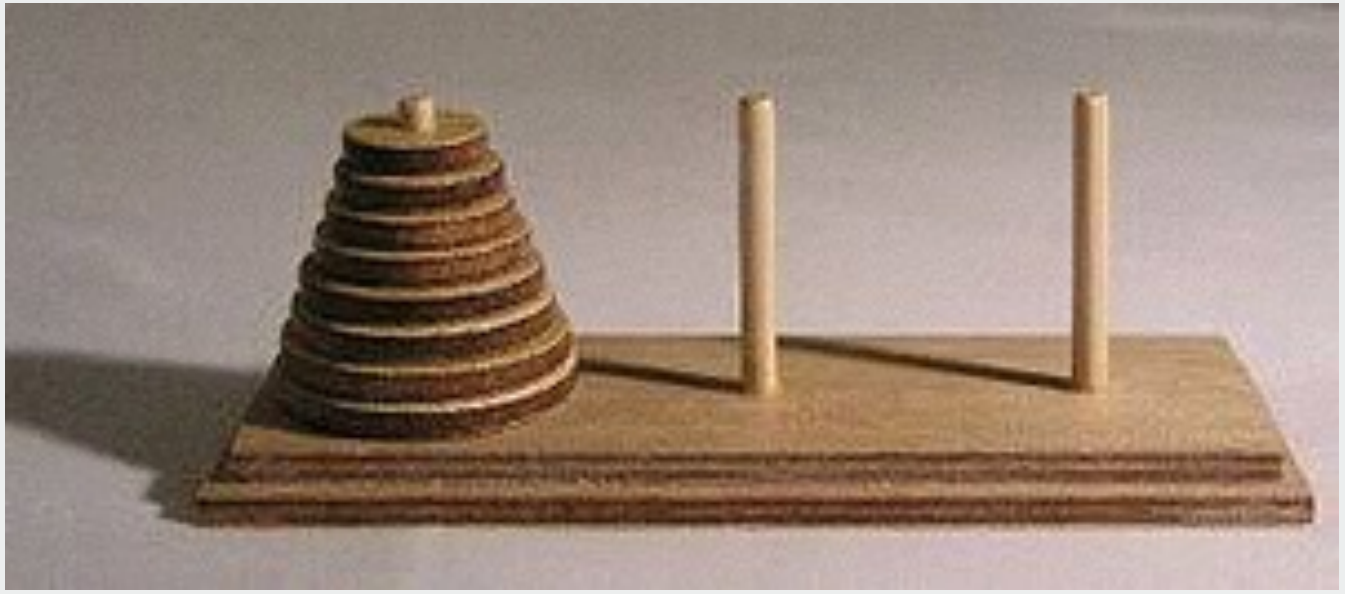
- Recall the **PARTITION** problem:
  - n numbers
  - $a_1, a_2, \ldots, a_n$

Suppose there is an algorithm with running time $O(a_{max}^3)$. Is it a polynomial algorithm or an exponential one?

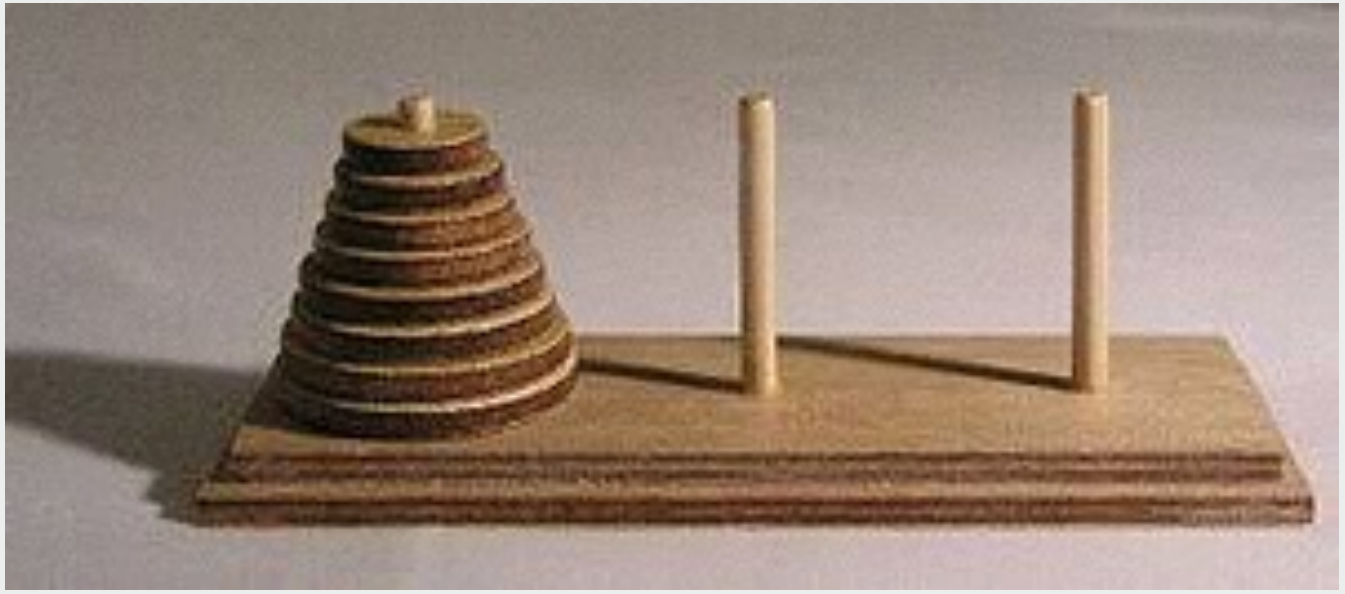




Legend: There is a 64-disk Hanoi Tower.
When the last move of the puzzle is completed,
the world will end

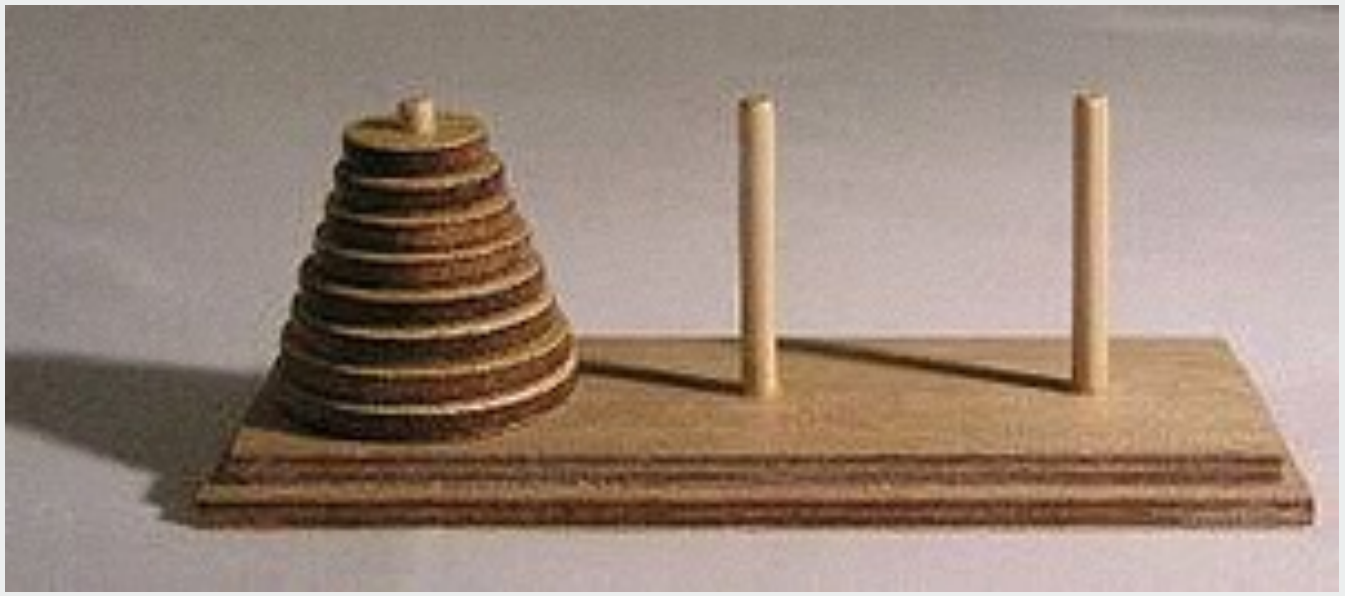


When there are n disks and 3 pegs,
it needs at least 2^n-1 moves.



When there are n disks and 3 pegs,
it needs at least 2^n-1 moves.

If one can move a disk within a second,
completing 64-disk Hanoi Tower needs 585 billion years.



When there are n disks and 3 pegs,
it needs at least 2^n-1 moves.

If one can move a disk within a second,
completing 64-disk Hanoi Tower needs 585 billion years.
(42 times the current age of the universe)

Polynomial versus exponential

- In general a polynomial algorithm is preferred over an exponential algorithm, because of the scalability (effect of increasing the size of the problem on the running time).
- For many problems polynomial algorithms exist, but for many others they have not been found yet. Major question: *are we to blame when we cannot find a polynomial algorithm for some problem?*
- This has been a major topic of research in the area of **Computational Complexity**.

Complexity class *P* and *NP*

Turing machine

- Wikipedia:

"A Turing machine is a device that manipulates symbols on a strip of tape according to a table of rules. Despite its simplicity, a Turing machine can be adapted to simulate the logic of any computer algorithm, and is particularly useful in explaining the functions of a CPU inside a computer."

Lego Turing machine

- https://www.youtube.com/watch?v=FTSAiF9AHN4
- See also http://www.legoturingmachine.org/

Turing machine: mathematical model

A Turing Machine TM is a mathematical model:

- which consists of an infinite length tape divided into cells on which input is given.
- It has a head which reads the input tape.
- A state register stores the state of the Turing machine.
- After reading an input symbol, it is replaced with another symbol, its internal state is changed, and it moves from one cell to the right or left. (this transition also depends on the state)
- If the TM reaches the final state `accept', the input string is accepted, otherwise rejected.

Complexity class *P*

- A decision problem belongs to the class ***P*** if there is a solution algorithm with a running time that is polynomial in the input size
  - In practice: encoding and corresponding concrete problem is assumed very implicitly.

- Alternative definition of **P**
  Class of decision problems, for which there exists a **Deterministic Turing Machine** that can solve any instance in polynomial time.

Complexity class *NP*

A decision problem belongs to the class ***NP*** if:

- Any solution *y* leading to `yes' can be encoded in polynomial space with respect to the size of the input *x*.
- Checking whether a given solution leads to `yes' can be done in polynomial time with respect to the size of *(x,y).*

Complexity classes *P* and *NP*

- Were originally and formally defined in terms of Turing machines

Alternative definition of ***NP***

- Class of decision problems, for which there exists a **Non-Deterministic Turing Machine** that can solve any yes instance in polynomial time.
  - The machine guesses a yes solution and then verifies that it is a yes solution

Never tell to an expert in Computational Complexity that you think that ***NP*** stands for Non Polynomial

NP stands for Non-deterministic Polynomial

Many problems are in NP

- Hamiltonian Path,
- Maximum Independent Set,
- Vertex Cover,
- Satisfiability,
- Integer Linear Programming
  - Easy to show for 0/1 programming,
  - non-trivial in general

Solution methods for linear programming

- Simplex method
  - Slower than polynomial
  - Practical
- Ellipsoid method
  - Polynomial (Khachian, 1979)
  - Not practical
- Interior points methods
  - Polynomial (Karmakar, 1984)
  - Outperforms Simplex for very large instances

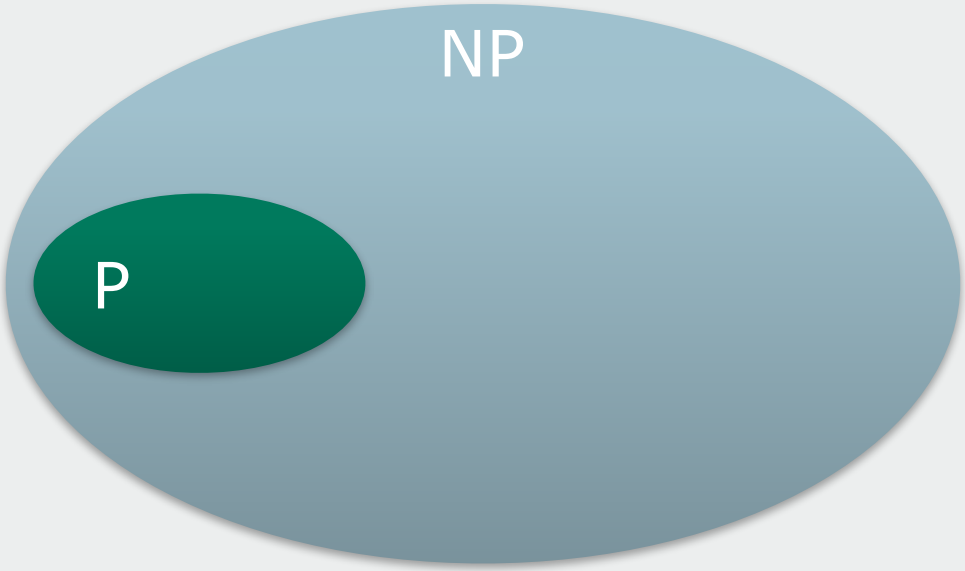
$$LP \in P$$

$P \subseteq NP$

- *P vs NP*

  $ 1 million **Millenium Prize** problem

  http://www.claymath.org/millennium/P_vs_NP



Reducibility

- Problem A is ***polynomial time reducible*** to problem B (or: **$A \leq_P B$**), if there exists a polynomial time computable function *f* from the instance set of I(A) of A to the instance set I(B) of B
  - $x \in$ I(A) is a yes-instance for A , if and only if, $f(x) \in$ I(B) is a yes-instance for B.

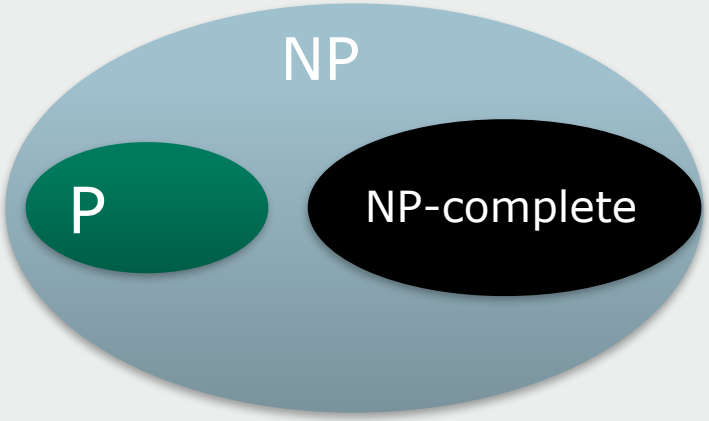
Reducibility (2)

- **Lemma:** If $A \leq_P B$ then: if $B \in \mathbf{P}$, then $A \in \mathbf{P}$.

  Proof-idea:
  - Let *x* be an instance of problem A.
  - Determine *f(x)*
  - run an algorithm for B on *f*(*x*).

NP-complete

- There are certain problems in **NP** whose individual complexity is related to that of the entire class
- If a polynomial time algorithm exists for any of the **NP-complete** problems, all problems in **NP** would be polynomial time solvable.



Definition: NP-hard and NP-complete

A problem A is **NP-hard**, if:

1. For every B $\in$ ***NP***: B $\leq_P$ A.

A problem A is **NP-complete**, if:

1. A $\in$ ***NP***.
2. For every B $\in$ ***NP***: B $\leq_P$ A.

NP-hardness/NP-completeness also used as term

- for problems that are not a decision problem, e.g. the optimization version of an NP-complete decision problem
- for problems that are '*harder than NP*'.

Decision vs optimization problems

- An optimization problem can be turned into a decision problem by introducing a threshold value *y*.
- In case of a minimization problem *M* , the decision variant becomes:
  - Given an instance of *M* together with a threshold value *y* , does there exist a feasible solution with outcome value ≤y?
- If optimization problem *M* can be solved in polynomial time, then its decision variant can be decided in polynomial time.
- If an optimization problem *H* has an NP-complete decision version *L*, then *H* is called NP-hard.

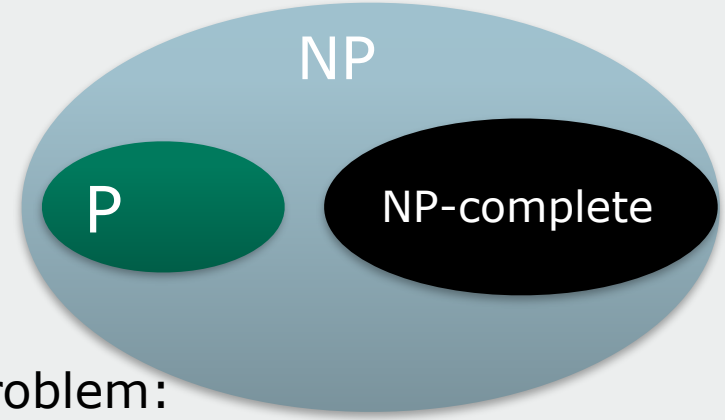
What does it mean to be NP-Complete?

"I can't find an efficient algorithm, but neither can all these famous people."

lecture complexity

What does it mean to be NP-Complete?

- Evidence that it is (very probably) hard to find an algorithm that solves the problem.
  - Always.
  - Exactly.
  - In polynomial time.

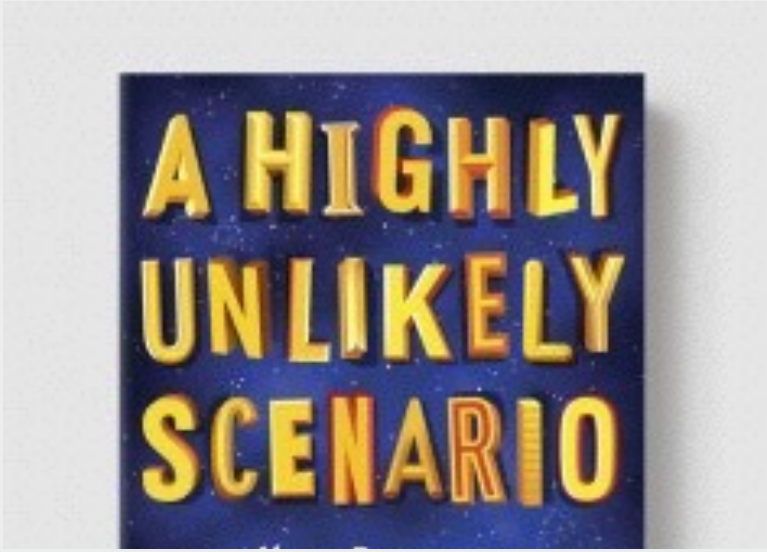


- For an NP-hard optimization problem:

If you want to find the optimal solution,
we cannot guarantee anything better than
checking all possibilities

What does it mean to be NP-Complete?

- Suppose there is a polynomial algorithm for some **NP-complete** problem X.
- In other words, X is **NP-complete** and X ∈ **P.**
- Then for every A ∈ ***NP***: $A \leq_P X$.
- So for every A ∈ ***NP***: A ∈ **P**
- All NP problems can be solved in polynomial time



CNF-Satisfiability: SAT

- **Input**: Expression over Boolean variables in conjunctive normal form (CNF).
  - "and" of clauses;
  - each clause "or" of variables or negations (x_i or $\underline{\neg(x_j)}$)

- **Question**: Is the expression satisfiable?
  I.e., can we give each variable a value (true or false) such that the expression becomes true?

Cook-Levin theorem

Theorem: CNF-Satisfiability is NP-complete.

- Most well known is Cook's proof, using Turing machine characterization of NP.
- It design a Turing machine that verifies yes-instances of SAT

Proving problems NP-complete

Lemma (key in the proof)

1. Let $A \leq_P B$ and let A be NP-complete. Then B is NP-hard.

2. Let $A \leq_P B$ and let A be NP-complete, and $B \in NP$. Then B is NP-complete.

Proving problems NP-complete: General recipe for a reduction

Suppose that you want to show NP-completeness of problem *B*;

1. Show that *B* belongs to the class NP.
   - Give/explain the encoding of a solution leading to yes,
   - Show/state that it is polynomial in the size of instance and
   - Explain how (or state that, if trivial) a yes-solution can verified in polynomial time w.r.t. the length of instance and solution.

2. Assume that you know that problem *A* is NP-complete. Show that A is reducible to B: $A \leq_p B$.
   - Take an arbitrary instance *I* of problem *A*.
   - Indicate how you can construct a special instance *f(I)* of problem *B* on basis of the instance of *A* that you selected:
   - **Show the answers to the instance of *A* and the special instance of *B* are equal**. (yes in A⇔ yes in B)
   - This transformation (or construction) must be possible in polynomial time.

Reduction: first example

Theorem: **SUBSET-SUM** is NP-complete

Reduction: first example

- The **SUBSET-SUM** problem
  - Given n non-negative integral values $a_1, a_2, \cdots a_n$ and a nonnegative integer B does there exist a subset S of the index-set $\{1,2,\cdots,n\}$, such that $\sum_{j \in S} a_j = B$
- Theorem: **SUBSET-SUM is NP-complete**

Reduction: first example

- The **SUBSET-SUM** problem
  - Given n non-negative integral values $a_1, a_2, \cdots a_n$ and a nonnegative integer B does there exist a subset S of the index-set $\{1,2,\cdots,n\}$, such that $\sum_{j \in S} a_j = B$
- Theorem: **SUBSET-SUM is NP-complete**
- the **PARTITION** problem
  - Given n non-negative integral values $a_1, a_2, \cdots a_n$ does there exist a subset S of the index-set $\{1,2,\cdots,n\}$, such that

$$\sum_{j \in S} a_j = \frac{1}{2} \sum_{j=1}^{n} a_j$$

- Suppose that we know that the **PARTITION** problem is **NP-complete**

Proof SUBSET-SUM is NP-complete

- **SUBSET-SUM** is in ***NP*:**
  - Input size $O(n \log B)$.
  - A solution leading to yes is a subset of $\{1,2,\cdots,n\}$. Can be encoded in polynomial time
  - Checking if a solution leads to yes is adding the included numbers a_i and comparing to B: polynomial
- Reduction from **PARTITION**
  - Let $a_1, a_2, \cdots a_n$ an instance from **PARTITION** Construct an instance from Subset sum with the same values $a'_i = a_i \; \forall i$ and $B = \frac{1}{2}\sum_{j=1}^{n} a_j$.
  - We have yes in Partition if and only if we have yes in Subset sum.
  - Transformation can be performed in polynomial time.

"I can't find an efficient algorithm, but neither can all these famous people."

Recap

- Class **P**: set of decision problems that can be solved in polynomial time
- Class **NP**: set of decision problems whose yes solution can be verified in polynomial time
- Polynomial time reduction $\leq_P$: $X \leq_P Y$ implies that Y is at least as hard as X! That is, Y cannot be easier than X.
- Problem A is **NP-hard** if every problem in NP can be reduced to A. (That is, for all $B \in NP$, $B \leq_P A$.)
- Problem A is **NP-complete** if it is in NP and it is NP-hard. That is, 1) any yes solution of A can be verified in polynomial time and 2) for all $B \in NP$, $B \leq_P A$.

Theorem: **3-SAT** is NP-complete

3-SAT is NP-complete

- A **3-SAT** is a special case of CNF-SAT where each clause has exactly three literals.

3-SAT is NP-complete

- A **3-SAT** is a special case of CNF-SAT where each clause has exactly three literals.

- Theorem: 3-SAT is NP-complete

Proof:

1. Membership in **NP** (easy to check).
2. Reduction (next slide).

- 3-SAT is important starting problem for many NP-completeness proofs.

Theorem: **3-SAT** is NP-complete

- Lemma: CNF-SAT $\leq_p$ 3-SAT
  - Clauses with less than 3 literals:
    - Replace $(x_1 \vee x_2)$ by $(x_1 \vee x_2 \vee x_1)$ (or $(x_1 \vee x_2 \vee x_2)$)
    - Replace (x_1) by $(x_1 \vee x_1 \vee x_1)$
  - Clauses with more than 3 literals:
    - For $(x_1 \vee x_2 \vee x_3 \vee \ldots \vee x_n)$, add new variable y and replace the clause by $(x_1 \vee x_2 \vee y) \wedge (\neg y \vee x_3 \vee \ldots \vee x_n)$, repeating this procedure.

Clique

Clique

- Clique: set of vertices $W \subseteq V$, such that for all $v,w \in W$: $\{v,w\} \in E$.

Clique

- Clique: set of vertices $W \subseteq V$, such that for all $v,w \in W$: $\{v,w\} \in E$.

CLIQUE (decision problem)

- Given: graph G=(V,E), integer *k.*

Clique

- Clique: set of vertices $W \subseteq V$, such that for all $v,w \in W$: $\{v,w\} \in E$.

CLIQUE (decision problem)

- Given: graph G=(V,E), integer *k.*
- Question: does G have a clique with at least / most *k* vertices?

Which one makes more sense?

Clique

- Clique: set of vertices $W \subseteq V$, such that for all $v,w \in W$: $\{v,w\} \in E$.

CLIQUE (decision problem)

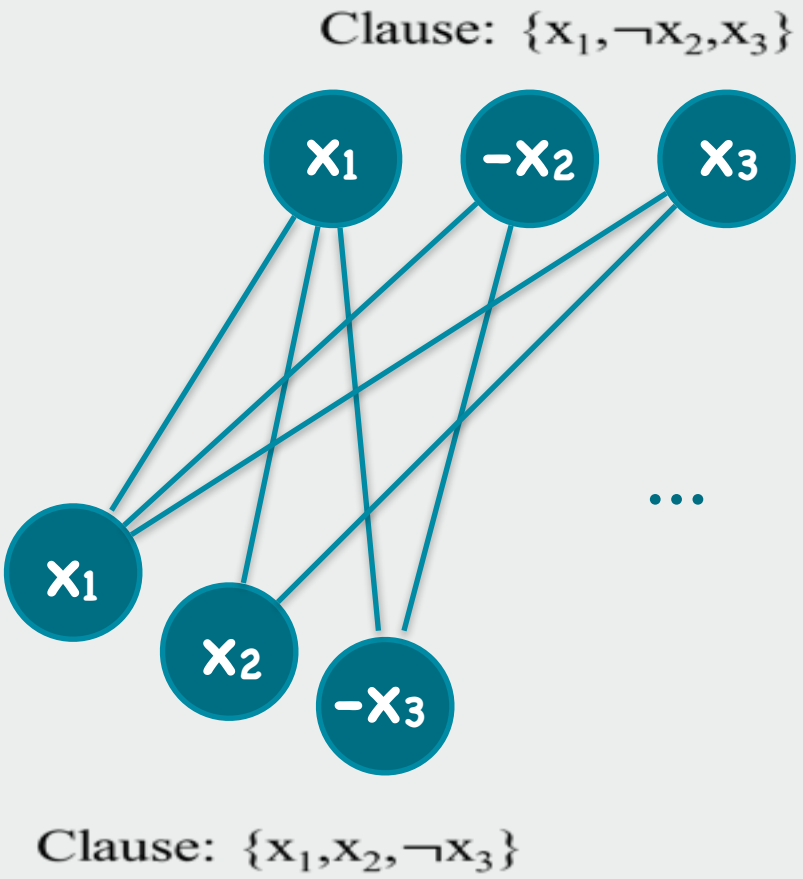
- Given: graph G=(V,E), integer *k.*
- Question: does G have a clique with at least *k* vertices?

Clique is NP-complete

- CLIQUE is in ***NP.***
  - Graph (V,E) with *n* nodes can be encoded in $O(n^2)$ bits; *k* can be encoded in *log(n)* bits
  - A solution is a set of nodes *S*, can be encoded in *O(n)* bits
  - Checking if *S* is a clique of size at least *k* , takes $O(n^2)$
- Reduction from 3-SAT
  - Let *x* be an instance from 3-SAT
  - Define instance f(x) from CLIQUE
  - Make clear that *f* can be performed in polynomial time
  - Show x is yes-instance iff f(x) is yes-instance
  - *All detailed in next slides*

Reduction from 3-SAT to CLIQUE

- Let *x* be an instance of 3-SAT
- Define the following instance of CLIQUE:
  - One vertex per literal per clause.
  - Edges between vertices in different clauses, except edges between x_i and $\neg x_i$.
  - If *x has m* clauses, look for clique of size *m*.
  - *Idea: you can make a clique from the literals that are true.*



Correctness

- There is a satisfying truth assignment, if and only if, f(x) has a clique with m vertices
  - ⇒ :
    - Let *x* be a satisfying truth assignment.
    - Select from each clause one true literal (there must be at least one since *x* is true).
      - Since vertices in different clauses, except x_i and $\neg x_i$ are connected, the corresponding vertices form a clique with *m* vertices.
  - ⇐:
    - Suppose f(x) has a clique of size *m*
    - Set variable x_i to true, if a vertex representing x_i is in the clique, otherwise set it to false. This is a satisfying truth assignment:
    - It cannot contain a vertex representing x_i and a vertex representing $\neg x_i$, so well-defined
    - The clique must contain one vertex from each 3 vertices representing a clause (vertices within a clause are not connected), so true

Independent set

Independent set

- Independent set: set of vertices $W \subseteq V$, such that for all $v,w \in W$: $\{v,w\} \notin E$.

- **INDEPENDENT-SET** (decision problem):

Independent set

- Independent set: set of vertices $W \subseteq V$, such that for all $v,w \in W$: $\{v,w\} \notin E$.

- **INDEPENDENT-SET** (decision problem):
  - Given: graph G, integer *k.*

Independent set

- Independent set: set of vertices $W \subseteq V$, such that for all $v,w \in W$: $\{v,w\} \notin E$.

- **INDEPENDENT-SET** (decision problem):
  - Given: graph G, integer *k.*
  - Question: Does G have an independent set of size at least *k*?

Independent set

- Independent set: set of vertices $W \subseteq V$, such that for all $v,w \in W$: $\{v,w\} \notin E$.

- **INDEPENDENT-SET** (decision problem):
  - Given: graph G, integer *k.*
  - Question: Does G have an independent set of size at least *k*?

- **INDEPENDENT-SET** is NP-complete

INDEPENDENT-SET is NP-complete

- In NP.
- Reduction: transform from CLIQUE.
- W is a clique in G, if and only if, W is an independent set in the complement of G (there is an edge in G^c iff. there is no edge in G).

How do I write down this proof?

Theorem: INDEPENDENT-SET is NP-complete.

Proof:

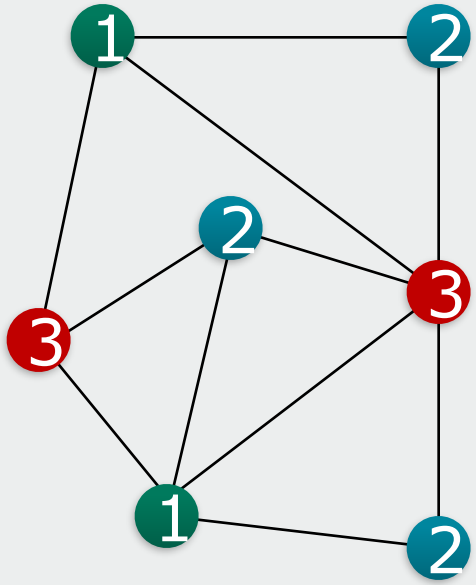
- The problem belongs to NP:
  - Solutions are subsets of vertices of the input graph; polynomial size
  - We can check in polynomial time for a given subset of vertices that it is an independent set and that its size is at least k.
- We use a reduction from CLIQUE.
  - Let (G,k) be an instance of the clique problem.
  - Transform this to instance (G^c,k) of the independent set problem with G^c the complement of G.
  - As G has a clique with k vertices, if and only if, G^c has an independent set with k vertices, this is a correct transformation.
  - The transformation can clearly be carried out in polynomial time.

Writing an NP-Completeness proof

- Proof starts with showing that problem belongs to NP.
  - Give/explain the encoding of a solution leading to yes,
  - show/state that it is polynomial in the size of instance and
  - explain how (or state that, if trivial) a yes-solution can verified in polynomial time w.r.t. the length of instance and solution.
- State which known NP-complete problem you reduce from.
- Explain the transformation (also called reduction).
- Give the proof: instance to original problem is Yes-instance, if and only if, transformed instance is Yes-instance for the known NP-complete problem.
  - Remember: you need to proof this in two directions.
- Phrase (or prove if not trivial): transformation can be carried out in polynomial time, hence problem is NP-complete.

3-Colouring Problem

- **3-COLOURING**
  - Given: Graph G=(V,E)
  - Question: Can we colour the vertices with 3 colours, such that for all edges {v,w} in E, the colour of v differs from the colour of w.
- 3-colouring is NP-Complete.



Proof: 3-COLOURING is NP-Complete

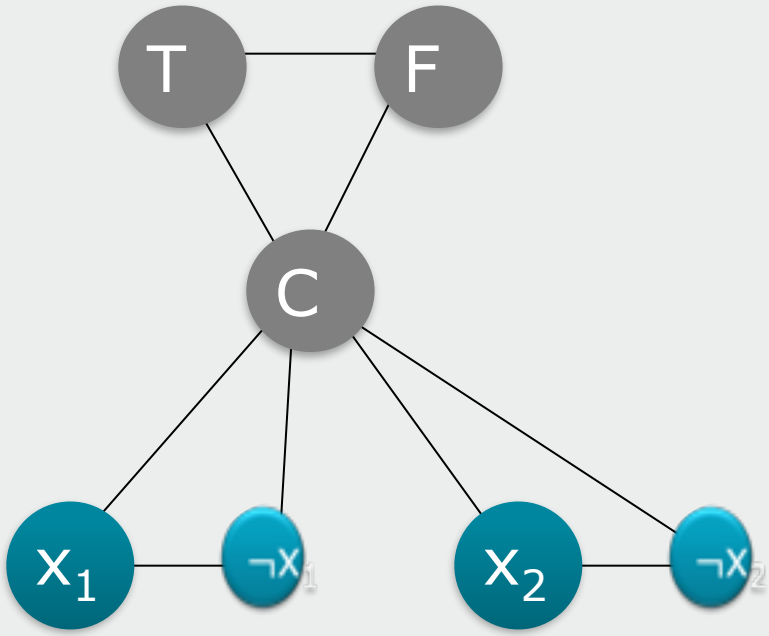
In NP:

- Encoding a solution: colour for each vertex (O(n))
- Checking if a solution is a yes instance: $O(n^2)$.

Reduction from 3-SAT:

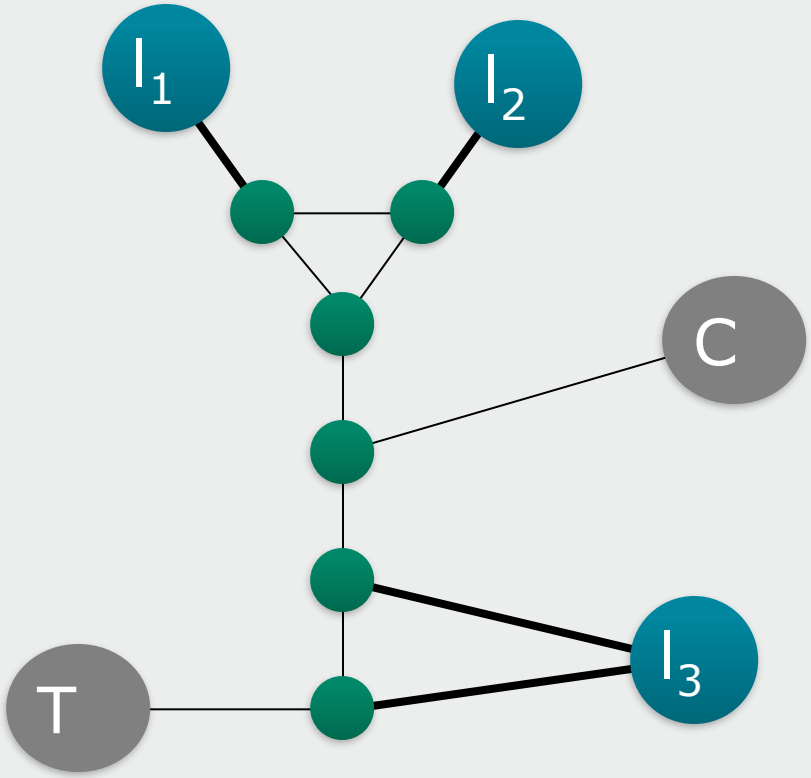
- **Given an instance from 3-SAT**
- **We build a graph in 3 steps:**
  1. Take a clique with 3 vertices True, False, C.
  2. Take two adjacent vertices for each variable *x*.

We name the colours: T, F, C



NP-Completeness of 3-Colouring

3. For each clause $\{l_1, l_2, l_3\}$, take the following gadget:



NP-Completeness of 3-Colouring

- The transformation takes polynomial time.
- Suppose the formula is satisfiable.
  - Colour the variables *T* or *F* according to their truth value. By making proper case distinction you can show that this can be extended to a 3-colouring of G.
- Suppose there is a 3-colouring of G.
  - Consider the following solution for the SAT formula. Give the variables with colour T assignment true and the ones with colour F assignment false.
  - You can check that $l_1 = l_2 = l_3 =$ F is not a feasible 3-colouring. So SAT formula must be true
- Note: In both cases, the intuition is that a literal vertex is coloured T (true), if and only if, we take it to be true in the formula.

THREE APPROACHES TO PROVE NP-COMPLETENESS

Approaches to proving NP-Hardness

Three approaches to prove NP-hardness:

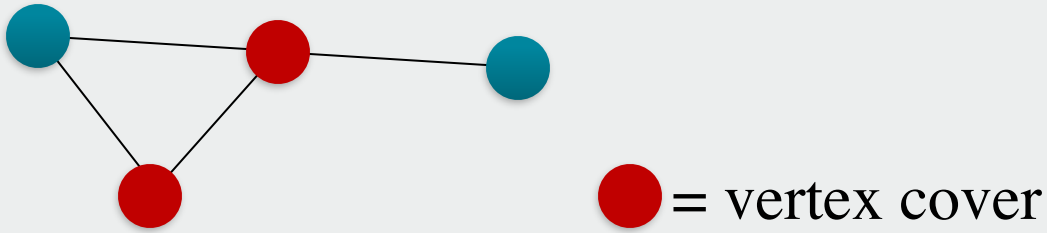
1) Restriction
2) Local replacement
3) Component Design

Approach 1: Restriction

- Take the problem.
- Add a restriction to *the set of instances*.
  Define a special case.
- If this restricted problem is a known NP-complete problem, then your original problem is also NP-complete.

Vertex Cover

- Set of vertices $W \subseteq V$ with for all $\{x,y\} \in E$: $x \in W$ or $y \in W$.
- **Vertex Cover** problem:
  - Given graph G, find vertex cover of minimum size.
- Decision version **VERTEX-COVER**:
  - Given graph G, number k
  - Find a vertex cover with size at most k
- **Vertex Cover** is NP-complete



Restriction: Weighted Vertex Cover

- **Weighted vertex cover**
  - Given: Graph G=(V,E), for each vertex $v \in V$, a positive integer weight $w(v)$, integer k.
  - Question: Does G have a vertex cover of total weight at most k?
- NP-Complete.
  - In NP.
  - NP-Hardness: reduction from Vertex Cover to Weighted Vertex Cover (set all weights to 1).
- This is the easiest form of reduction.
- Vertex cover is NP-complete in exercises

Restriction: Subset sum

- Suppose that we know that the Partition problem is NP-complete;
- the **PARTITION** problem:
  - Given n non-negative integral values $a_1, a_2, \cdots a_n$ does there exist a subset S of the index-set $\{1,2,\cdots,n\}$, such that

$$\sum_{j \in S} a_j = \frac{1}{2} \sum_{j=1}^{n} a_j$$

- the **SUBSET-SUM** problem:
  - Given n non-negative integral values $a_1, a_2, \cdots a_n$ and a nonnegative integer B does there exist a subset S of the index-set $\{1,2,\cdots,n\}$, such that $\sum_{j \in S} a_j = B$

- SUBSET-SUM is a generalization of PARTITION (or in other words, PARTITION is a special case of SUBSET-SUM), and hence SUBSET-SUM is NP-complete as well.

Approach 2: Local replacement

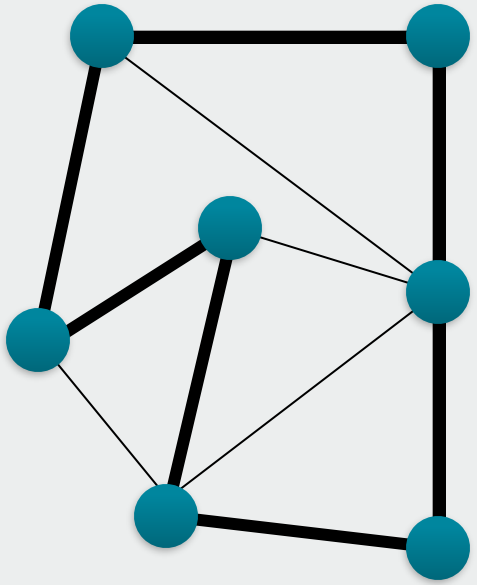
- Form an instance of our problem by
  - Taking an instance of a known NP-Complete problem.
  - Making some change "everywhere".
  - Such that we get an equivalent instance, but now of the problem we want to show NP-complete.

Examples of Local Replacement

- We saw:
  - 3-SAT (from SAT).
  - CLIQUE (from 3-SAT).
  - INDEPENDENT-SET (from CLIQUE).
- We will see:
  - Travelling Salesman Problem (from Hamiltonian Circuit).
  - Planar 3-Colouring (from 3-Colouring).

Hamiltonian Circuit

- Given: Graph G
- Question: does G have a simple cycle that contains all vertices?



NP-Completeness of Travelling Salesman Problem by local replacement

- In NP.
- Reduction from HAMILTONIAN-CIRCUIT:
  - Take city for each vertex.
  - Take cost(i,j) = 1 if $\{i,j\} \notin E$.
  - Take cost(i,j) = 0, if $\{i,j\} \in E$.
  - G has HC, if and only if, there is a TSP-tour of length 0.

Remark

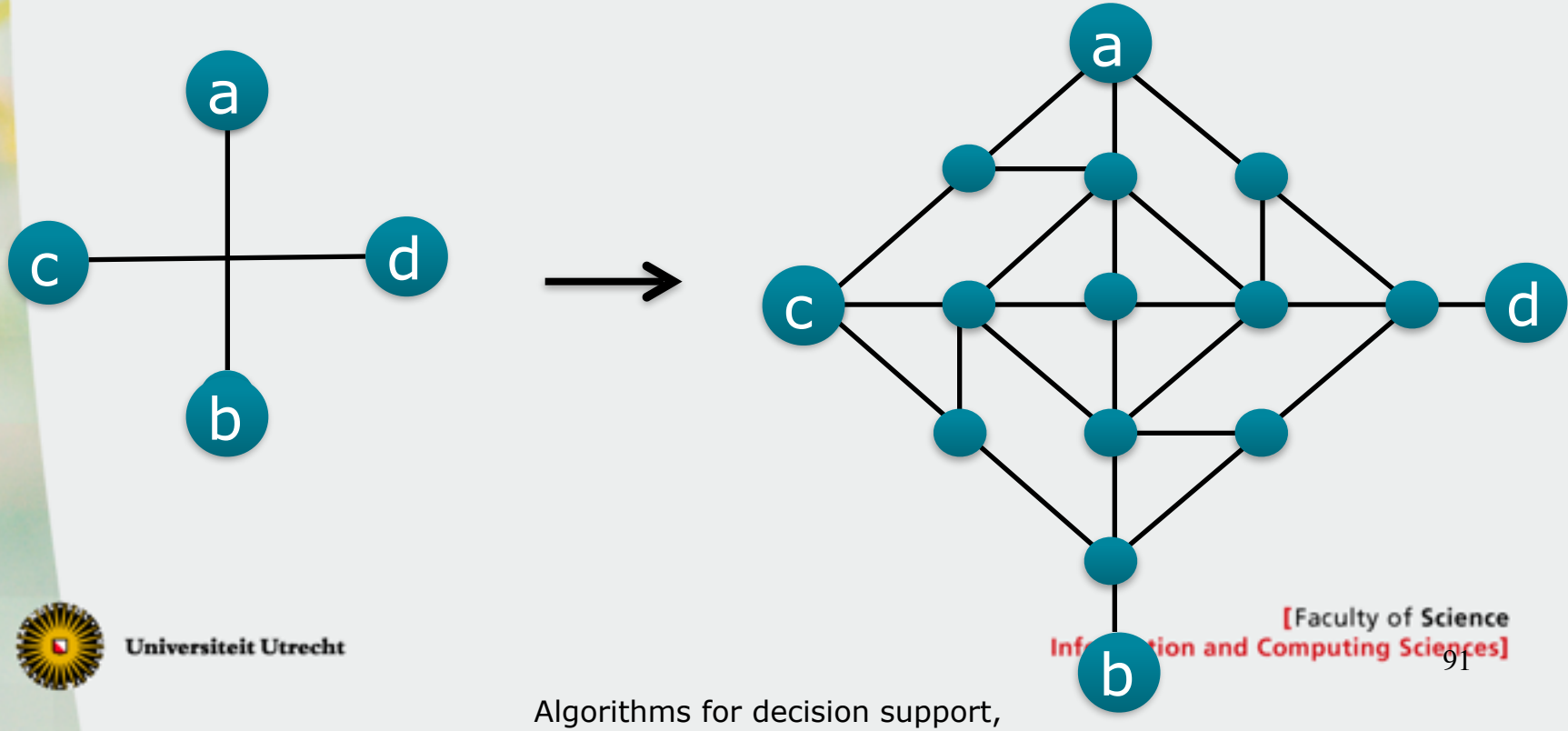
- TSP variant with triangle inequality:
  - Use weights 2 and 1.
  - G has HC, if and only if, there is a TSP-tour of length n.

NP-Completeness of Planar 3-Colouring by Local Replacement

- **Planar 3-COLOURING**
  - Given: Planar graph G=(V,E).
  - A **planar graph** is a **graph** that can be embedded in the **plane**, i.e., it can be drawn on the **plane** in such a way that its edges intersect only at their endpoints.
  - Question: Can we colour the vertices of G with 3 colours, such that adjacent vertices have different colours?
- Plan: reduction from 3-COLOURING.
  - Take arbitrary graph G.
  - Draw G on the plane.
  - So, we possibly get some crossings.
  - Replace the crossings by something clever.

Clever reduction

- Garey and Johnson (1979) found the following:



Property of the gadget

- If *a* and *b* have the same colour, we cannot colour the gadget with 3 colours.
- If *c* and *d* have the same colour, we cannot colour the gadget with 3 colours.
- Otherwise, we can.

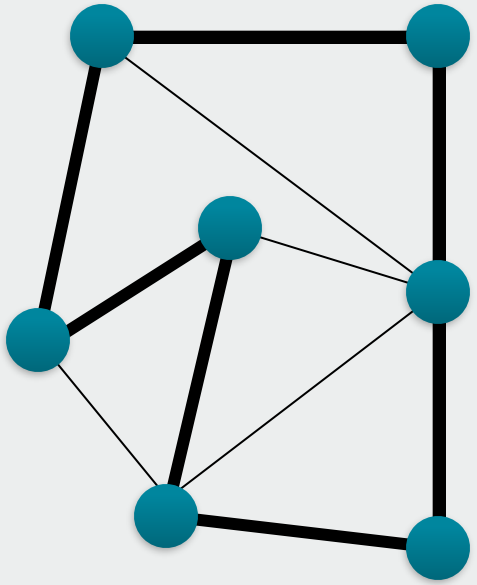
- ... Some additional details (basically, repeat the step) when an edge has more than one crossing …

Approach 3: Component design

- Component design (gadgeteering).
- Build (often complicated) parts of an instance with certain properties.
  - Often we call these parts components, gadgets, or widgets.
- Glue them together in such a way that the proof works.
- Examples:
  - 3-COLORING (from 3-SAT).
  - Hamiltonian Circuit (from Vertex Cover).

Hamiltonian circuit

- Given: Graph G
- Question: does G have a simple cycle that contains all vertices?

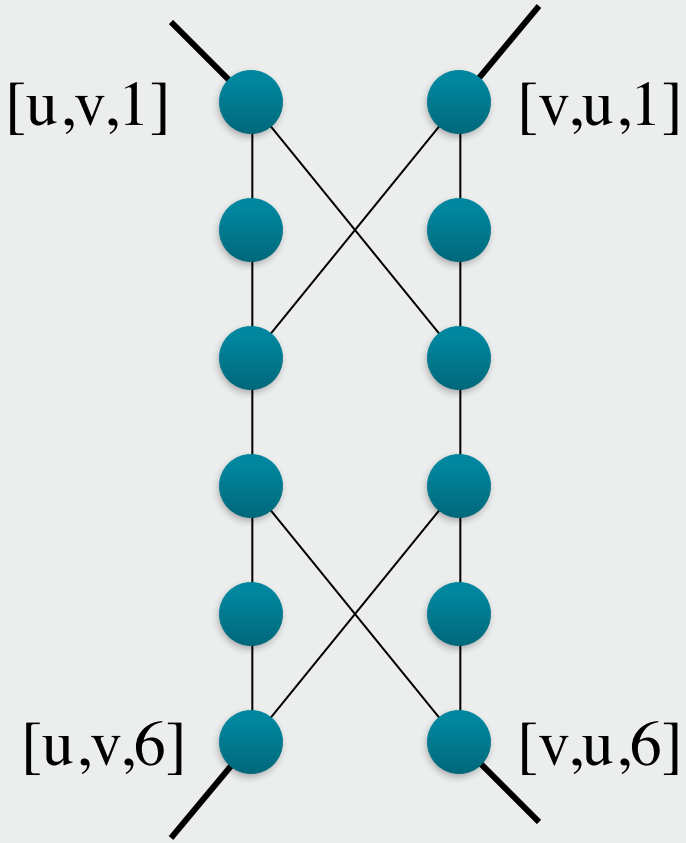


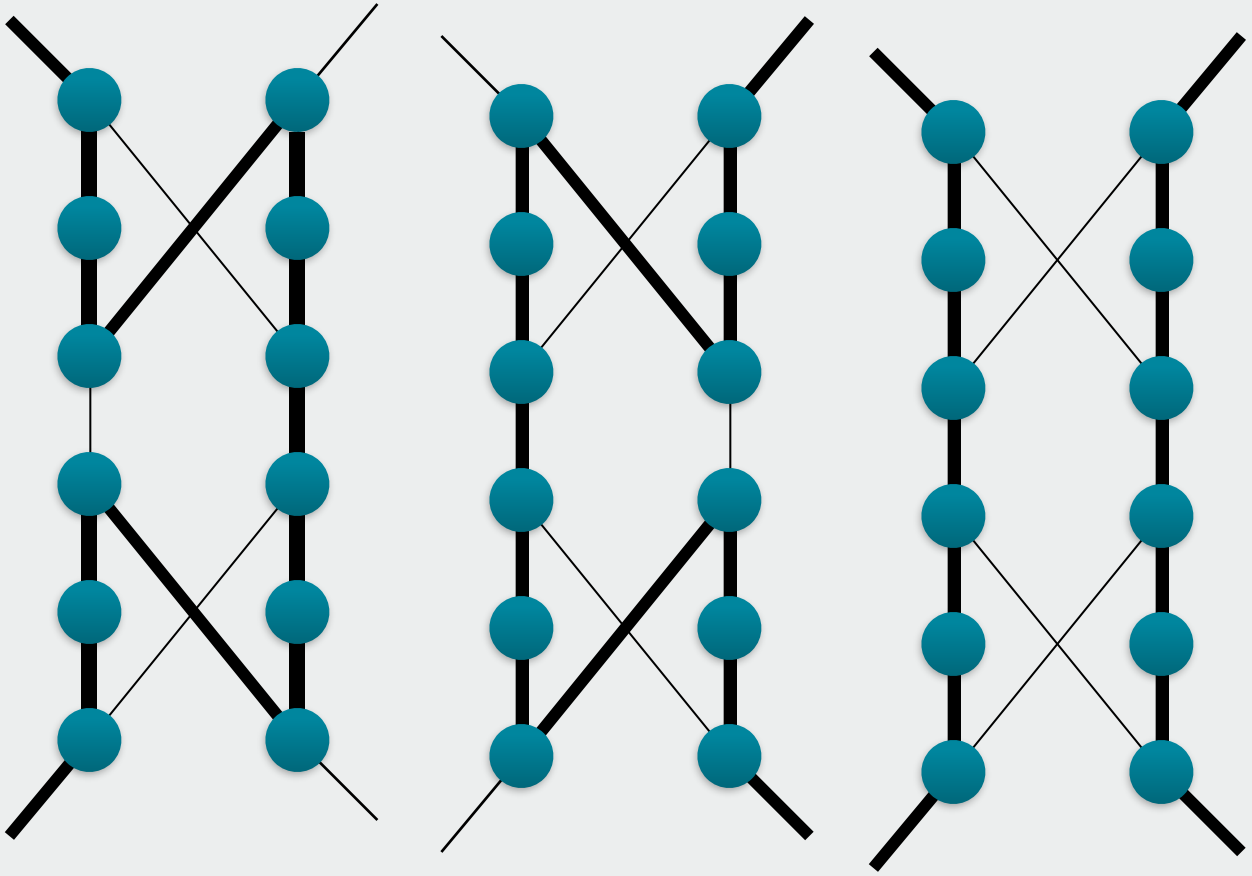
NP-Completeness of Hamiltonian Circuit

- Hamiltonian Circuit is in NP.
- Vertex Cover $\leq_P$ Hamiltonian Circuit: complicated proof (*component design*)
  - Given a graph *G* and an integer *k*, we construct a graph H, such that H has a Hamilton Circuit, if and only if G has a Vertex Cover C of size *k.*
  - Widgets (one for each edge of graph *G)*
  - Selector vertices: *k*.

Widget

- For each edge $\{u,v\}$ we have a widget W_{uv}
- Details for interested reader





Only possible ways to visit all vertices in widget

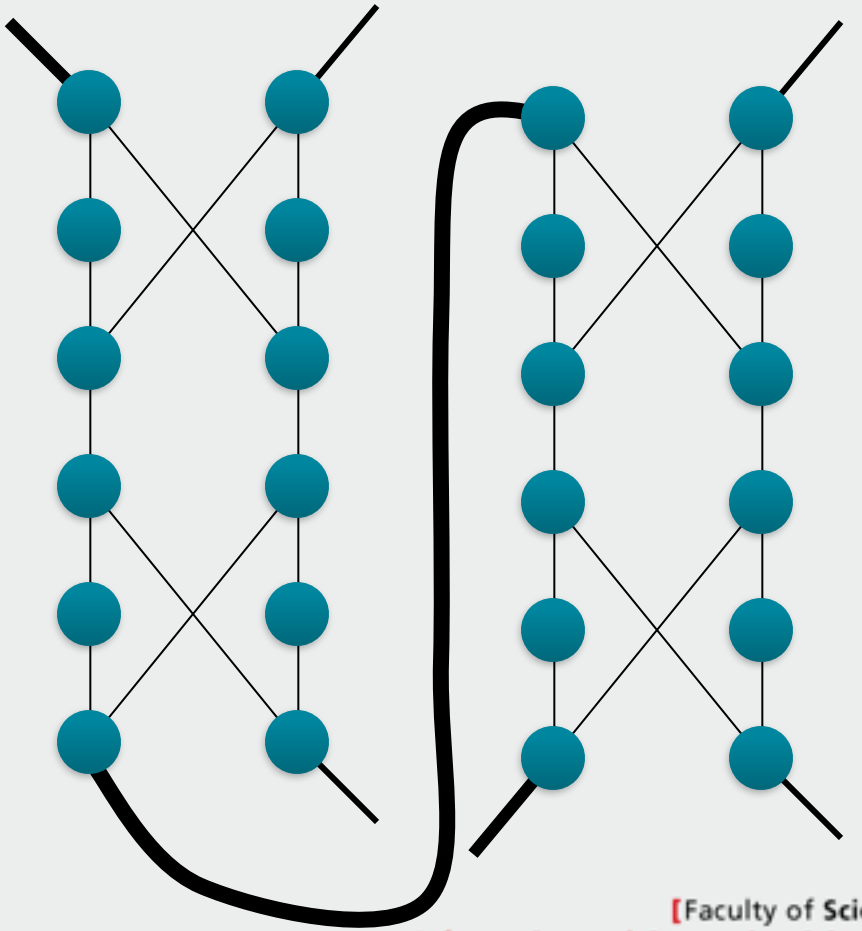
- One visit to the widget from the u side
- One visit from the v side
- Two visits
- You never switch to another side

Connecting the widgets

- For each vertex v we connect the widgets of all its edges $\{v,w\}$.
- Suppose v has neighbors x_1, …, x_r:
  - add edges $\{[v,x_1,6], [v,x_2,1]\}$, $\{[v,x_2,6], [v,x_3,1]\}$,

    …,

    $\{[v,x_{r-1},6],[v,x_r,1]\}$.



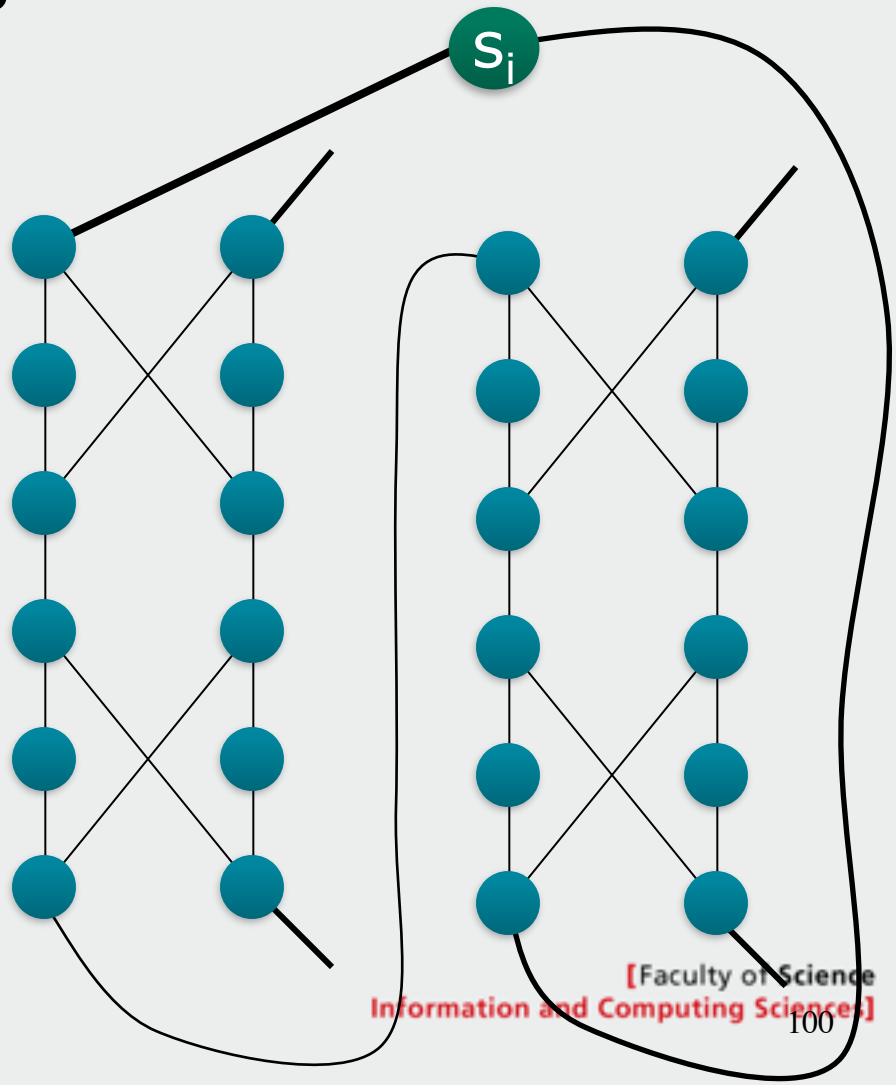
Selector vertices

- We have k selector vertices $s_1, \ldots, s_k$.
- They form a clique
- These will represent the vertices selected for the vertex cover.

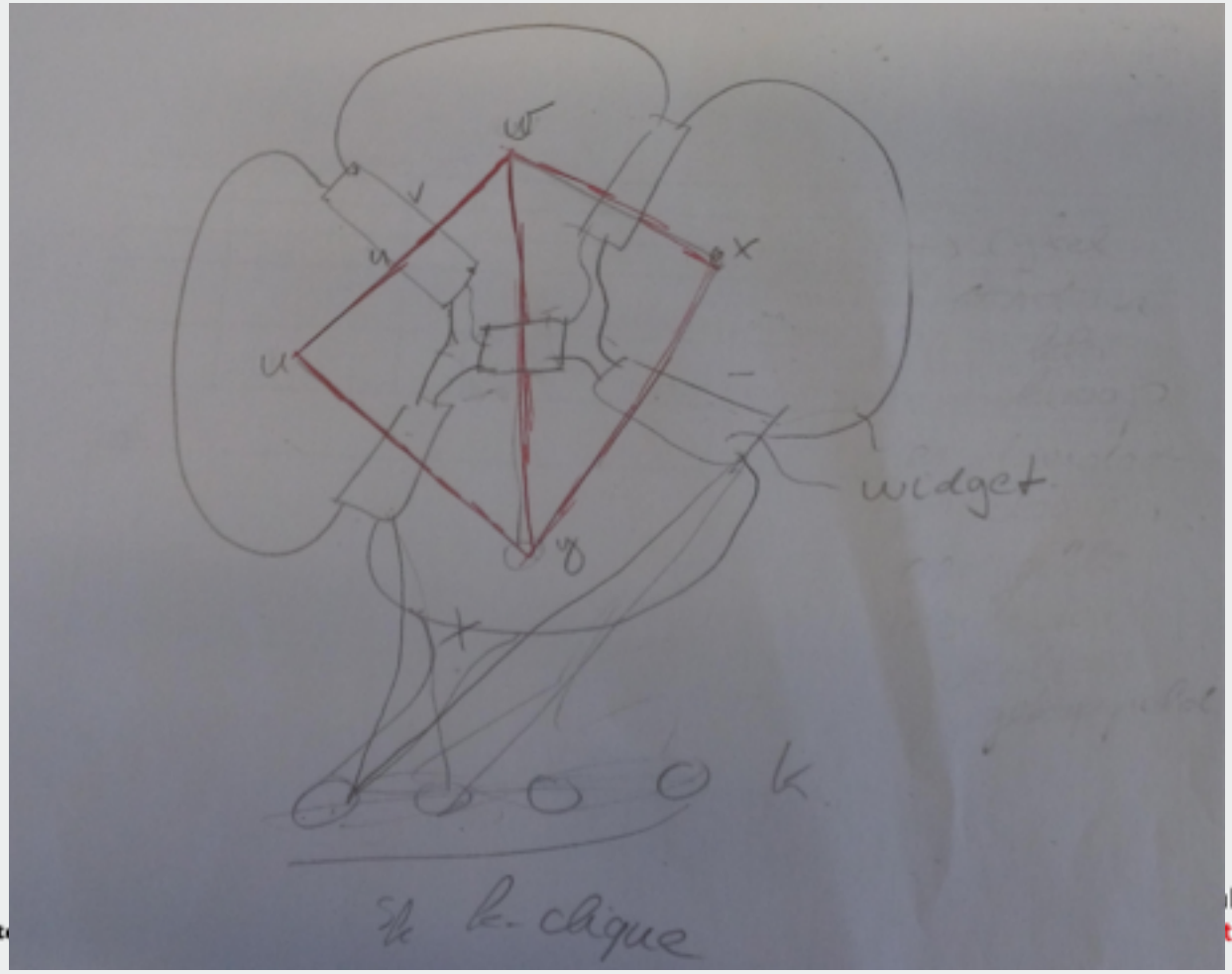
Connecting the selector vertices to the widgets

- **Each** selector vertex is attached to the first neighbor widget of each vertex, i.e. to vertex $[v,x_1,1]$ and to the last neighbor widget $[v,x_r,6]$
- We form a cycles around each node v, which are interrupted by selector vertices

Vertex in example has degree 2



A (draft, draft) drawing to give you an idea



Correctness of reduction

- **Lemma**: G has a vertex cover of size (at most) *k,* if and only if H has a Hamiltonian circuit.
- The idea of the proof is that the Hamiltonian circuit has to contain exactly *k* circuits around nodes (since there are *k* selector vertices. These *k* nodes form a vertex cover
- The reduction takes polynomial time.
- So, we can conclude that Hamiltonian Circuit is NP-complete.

Problem with numbers: example

- **Knapsack**
  - Given: Set S of items, each with integer value c_i and integer weight a_i, integers B and V.
  - Question: is there a subset of S of weight no more than B, with total value at least V?
- Solvable using dynamic programming in O(nB) time, that is, solvable in *pseudo-polynomial time.*
- *Pseudo-polynomial means polynomial in the unary encoding of the input.*
- *Dynamic programming:*
  - $W(j,w) = max\{\sum_{i \in S} c_i | S \subseteq \{1,2,\dots,j\}, \sum_{i \in S} a_i \leq w\}$
  - $W(j+1,w) = max(W(j,w), W(j,w-a_{j+1}) + c_{j+1})$
  - Look-up details in Algorithms course
- We say, Knapsack is **Weak NP-complete**.

Problems with numbers

- Weak NP-complete (NP-complete in the ordinary sense):
  - Problem is NP-complete if numbers are given in binary, *but* polynomial time solvable when numbers are given in unary encoding.
  - Algorithms are known which solve them in time bounded by a polynomial in the input length and the magnitude of the largest number in the given problem instance.
  - The problem is solvable in pseudo-polynomial time
- Strong NP-complete (NP-complete in the strong sense)
  - Problem is NP-complete if numbers are given in unary encoding
  - Problem is NP-complete even when the numerical parameters are bounded by a polynomial in the input size

Examples

- **3-Partition**
  - Given: set of positive integers S.
  - Question: can we partition S into sets of exactly 3 elements each, such that each has the same sum (t)?
- 3-Partition is Strong NP-complete.
  - *t* must be the sum of S divided by the number of groups (|S|/3).
  - Starting point for many reductions.

Let P' be NP-complete in the strong sense. If P' can be reduced to P by a pseudo-polynomial reduction, and P in NP, then P is also NP-complete in the strong sense.

Pitfalls in NP-completeness proofs

- Uncapacitated facility location is NP-complete by a reduction from ILP
- Uncapacitated facility location
  - Data:
    - m customers, n possible locations of depot
    - Each customer is assigned to one depot
    - d_{ij} cost of serving customer i by depot j
    - Fixed cost for opening depot DC: F_j
  - Which depots are opened and which customer is served by which depot?
  - Decision variant: is there a solution with cost at most M
- Given an instance from Uncapacitated Facility Location: formulate the problem as ILP

Recall: Proving problems NP-complete: General recipe for a reduction

- Suppose that you want to show NP-completeness of problem *B*;
  1. Show that *B* belongs to the class NP.
  2. Assume that you know that problem *A* is NP-complete. Show that A is reducible to B: $A \leq_p B$.
     - Take an arbitrary instance *I* of problem *A*.
     - Indicate how you can construct a special instance *f(I)* of problem *B* on basis of the instance of *A* that you selected: **the answers to the instance of *A* and the special instance of *B* must be equal**.
     - This transformation (or construction) must be possible in polynomial time.

Pitfalls in NP-completeness proofs

- Finding the largest integer in an array is NP-complete. Proof by a reduction from a Knapsack

- **Largest number:** given an array $\{q_1, q_2, \ldots, q_k\}$ of integers and a value *Q*.

  Does the array contain a number with value at least *Q*?

- **Knapsack:** Given *n* items with weight a_i and value c_i a knapsack volume *B*, and a number *V*. Is there a feasible knapsack with value at least *V*?

Pitfalls in NP-completeness proofs

- **Reduction:**
  - Given an instance of Knapsack
  - Let $S_1, S_2, \ldots$ be the subsets of $\{1,2,\ldots n\}$
  - Define $q_{S_i} = \begin{cases} \sum_{j \in S_i} c_i & \text{if } \sum_{j \in S_i} a_j \leq B \\ -\infty \text{ (or omit)} & \text{otherwise} \end{cases}$
  - Set Q=V
- Yes in Knapsack iff Yes in Largest number

Pitfalls in NP-completeness proofs

- Going in the wrong direction
- Non-polynomial reduction

Reducibility

- Informal rule to remember (Ezelsbruggetje, 'Bridge for a Donkey' disclaimer: do not sue me for bad English)

$$A \leq_p B$$

indicates that B is at least as difficult as A

More specifically:

- $B \in P \implies A \in P$
- $A\ NP\text{-complete} \wedge B \in NP \implies B\ NP\text{-complete}$

NP-Completeness

SOME ANIMALS FROM THE COMPLEXITY ZOO

Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Much more complexity classes

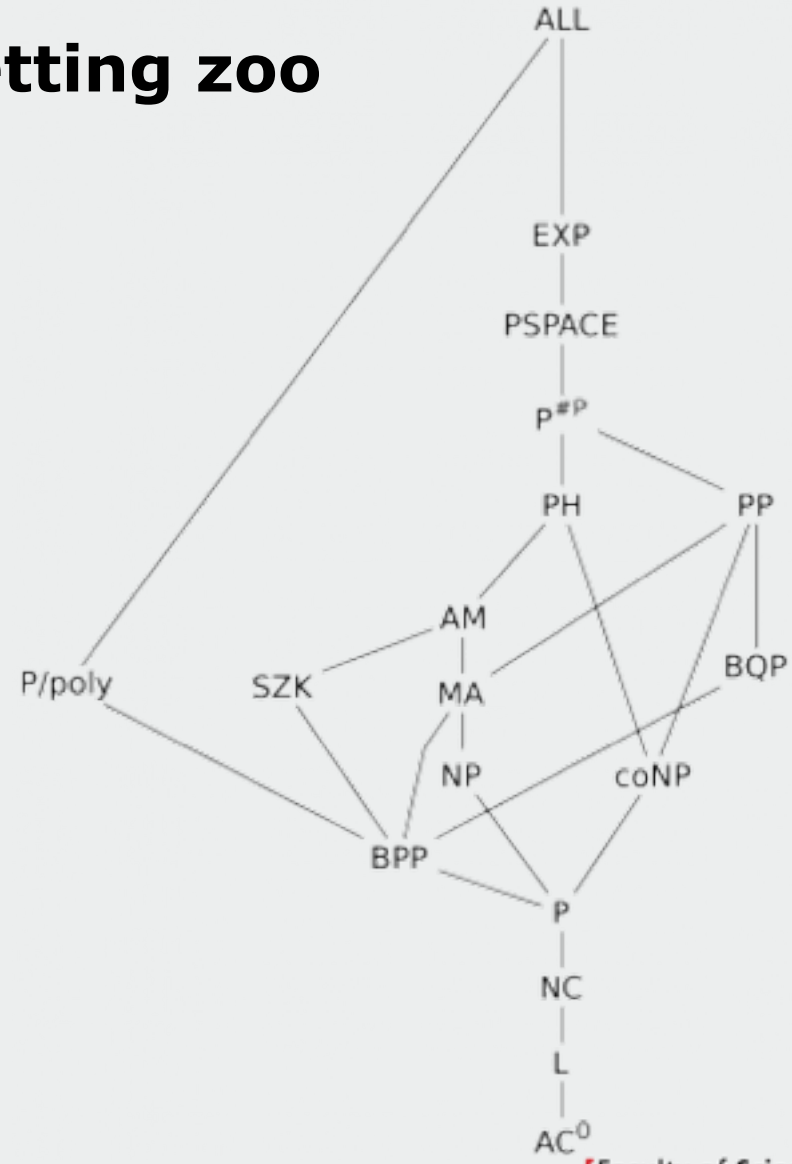
- In Theoretical Computer Science, a large number of other complexity classes have been defined.

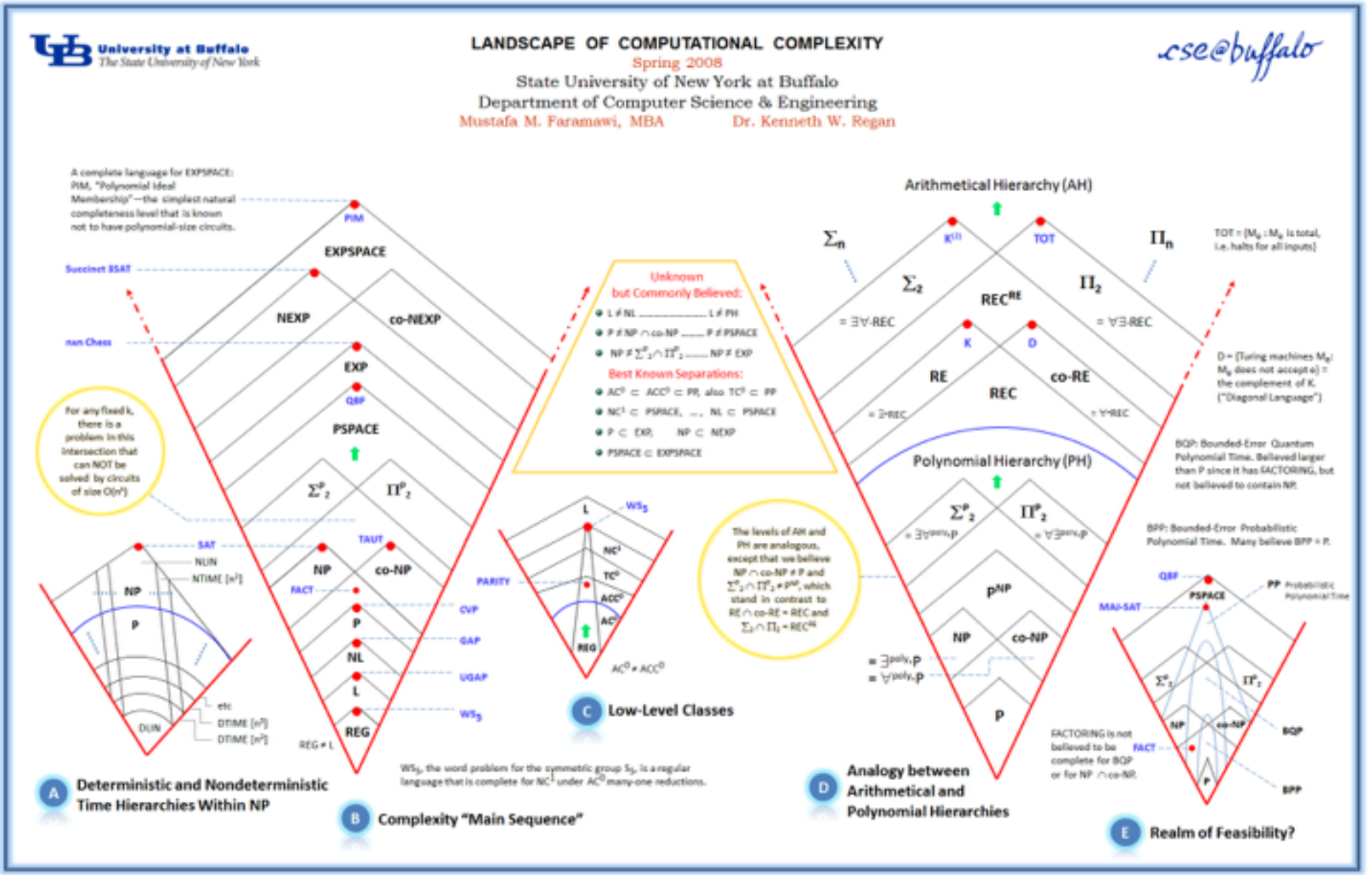
- Completeness works in the same way
- Here, we give an informal introduction to a few of the more important ones.
- There is much, much, much more…

The petting zoo

- https://complexityzoo.uwaterloo.ca/Petting_Zoo:

"Once finished, the Petting Zoo will introduce complexity theory to newcomers unready for the terrifying and complex beasts lurking in the main zoo."




University at Buffalo
The State University of New York
LANDSCAPE OF COMPUTATIONAL COMPLEXITY
Spring 2008
State University of New York at Buffalo
Department of Computer Science & Engineering
Mustafa M. Faramawi, MBA
Dr. Kenneth W. Regan
cse@buffalo
A complete language for EXPSPACE: PIM, "Polynomial Ideal Membership"—the simplest natural completeness level that is known not to have polynomial-size circuits.
PIM
EXPSPACE
Succinct 3SAT
NEXP
co-NEXP
nxn Chess
EXP
QBF
PSPACE
For any fixed k, there is a problem in this intersection that can NOT be solved by circuits of size O(n^k)
SAT
NLIN
NTIME [n^1]
NP
P
D(IN)
etc
DTIME [n^2]
DTIME [n^1]
TAUT
co-NP
FACT
CVP
GAP
NL
UGAP
L
WS3
REG
REG ≠ L
Unknown but Commonly Believed:
L ≠ NL ..... L ≠ PH
P ≠ NP ∩ co-NP ..... P ≠ PSPACE
NP ≠ Σ^P_1 ∩ Π^P_1 ..... NP ≠ EXP
Best Known Separations:
AC^0 ⊂ ACC^0 ⊂ PP, also TC^0 ⊂ PP
NC^1 ⊂ PSPACE, ..., NL ⊂ PSPACE
P ⊂ EXP, NP ⊂ NEXP
PSPACE ⊂ EXPSPACE
L
WS5
NC^1
TC^0
ACC^0
AC^0
PARITY
REG
AC^0 ≠ ACC^0
The levels of AH and PH are analogous, except that we believe NP ∩ co-NP ≠ P and Σ^P_2 ∩ Π^P_2 ≠ P^NP, which stand in contrast to RE ∩ co-RE = REC and Σ_2 ∩ Π_2 = REC^RE
Arithmetical Hierarchy (AH)
Σ_n
Σ_2
= ∃∀·REC
K^(1)
TOT
REC^RE
K
D
RE
co-RE
REC
= ∃·REC
= ∀·REC
Π_n
Π_2
= ∀∃·REC
TOT = {M_e : M_e is total, i.e. halts for all inputs}
D = {Turing machines M_e : M_e does not accept e} = the complement of K. ("Diagonal Language")
Polynomial Hierarchy (PH)
Σ^P_2
Π^P_2
= ∃∀^poly·P
= ∀∃^poly·P
P^NP
NP
co-NP
P
= ∃^poly·P
= ∀^poly·P
BQP: Bounded-Error Quantum Polynomial Time. Believed larger than P since it has FACTORING, but not believed to contain NP.
BPP: Bounded-Error Probabilistic Polynomial Time. Many believe BPP = P.
QBF
PSPACE
PP Probabilistic Polynomial Time
MAJ-SAT
Σ^P_1
Π^P_1
NP
co-NP
FACT
P
BQP
BPP
FACTORING is not believed to be complete for BQP or for NP ∩ co-NP.
A Deterministic and Nondeterministic Time Hierarchies Within NP
B Complexity "Main Sequence"
WS5, the word problem for the symmetric group S5, is a regular language that is complete for NC^1 under AC^0 many-one reductions.
C Low-Level Classes
D Analogy between Arithmetical and Polynomial Hierarchies
E Realm of Feasibility?

Complexity class *NP*

A decision problem belongs to the class ***NP*** if:

- Any solution *y* leading to `yes' can be encoded in polynomial space with respect to the size of the input *x*.
- Checking whether a given solution leads to `yes' can be done in polynomial time with respect to the size of *(x,y).*

Complexity class co-*NP*

A decision problem belongs to the class co-***NP*** if:

- Any solution *y* leading to `no' can be encoded in polynomial space with respect to the size of the input *x*.
- Checking whether a given solution leads to `no' can be done in polynomial time with respect to the size of *(x,y).*

coNP

- Complement of a class: switch “yes” and “no”.
- Polynomial time verification of solutions of no instances.
- coNP: complement of problems in NP, e.g.:

  NOT-HAMILTONIAN
  - Given: Graph G.
  - Question: Does G NOT have a Hamiltonian circuit.

  UNSATISFIABLE
  - Given: Boolean formula in CNF.
  - Question: Do all truth assignments to the variable make the formula false?

- All problems in P are in coNP.
- NP-Complete problems are most probably not in coNP.

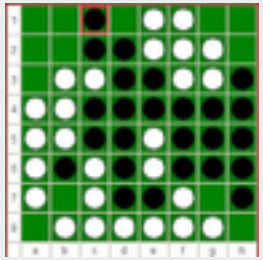
A more natural example for NP and coNP

- **Integer factorisation**:
  - Given: integer n, integer m.
  - Question: does n have a prime factor less than m?

- In NP - Yes-solution:
  - Two numbers *i, j* with *i* prime (prime testing is in P – 2002) ,
    $ij = n$, $i < m$.
- In coNP - No-solution:
  - List of all prime factors of n.
  - Verify that these are primes (prime testing is in P – 2002) and at least *m* (trivial).
  - Verify that when multiplied we have n.
- Problem in NP and coNP.
  - Also in P? Maybe not – for cryptography sake we hope not!

PSPACE

- All decision problems solvable in polynomial space.
- Unknown: is P=PSPACE?
- NP included in PSPACE
- PSPACE-complete, e.g.,
  - Strategies for generalized Tic-Tac-Toe, strategies for generalized Reversi,



  - Quantified Boolean formula's (QBF):

$$\forall x_1 \exists x_2 \forall x_3 \exists x_4 \left(x_1 \vee \neg x_2 \vee x_3\right) \wedge \left(x_2 \vee \neg x_4\right)$$

Goal

- Make sure that you are able to write down a complexity proof accurately
- To give an NP-completeness proof, you have to be able to select a problem to reduce from out off a list of given NP-complete problems.
- For the following problems you have to know the definition:
  - Partition
  - Subset Sum
  - Knapsack
  - Clique
  - Independent set
  - Vertex Cover
  - Hamiltonian Path
  - Hamiltonian Cycle