

# CS711008Z Algorithm Design and Analysis

## Lecture 3. Problem hardness and polynomial-time reduction <sup>1</sup>

Dongbo Bu

Institute of Computing Technology  
Chinese Academy of Sciences, Beijing, China

---

<sup>1</sup>The slides are prepared based on Introduction to algorithms, Algorithm design, and Computer and Intractability.

- Problem intrinsic property: hardness
- Reduction: to identify the relationship between different problems;

# NP-Completeness Cartoon: Bandersnatch problem

- One day your boss calls you into his office and confides that the company is about to enter the highly competitive "bandersnatch" market.
- A good method is needed for determining whether or not any given set of specifications for a new bandersnatch component can be met and, if so, for constructing a design that meets them.
- Since you are the company's chief algorithm official, your charge is to find an efficient algorithm for doing this.

2

---

<sup>2</sup>Excerpted from Computer and Intractability (by Garey and Johnson) ▶

## Trial 1: attempt to solve this problem

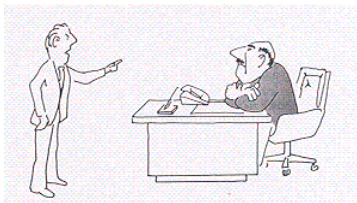
- Some weeks later, you have not been able to come up with any algorithm substantially better than searching through all possible designs. This would involve years of computation time for just one set of specifications.
- You simply return to your boss's office and report: "I can't find an efficient algorithm, I guess I'm just too dumb. "



- But perhaps this is unfair to you: **the problem might be intrinsically hard.**

## Trial 2: try to prove the hardness directly

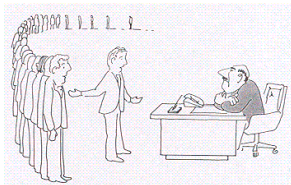
- So it would be much better if you could prove that the bandersnatch problem is inherently intractable, that no algorithm could possibly solve it quickly.
- Then you could stride confidently into the boss's office and proclaim: " I can't find an efficient algorithm, because no such algorithm is possible. "



- Unfortunately, proving inherent intractability can be just as hard as finding efficient algorithms.

# Trial 3: to show the relative hardness with other hard problems

- For such a “grey area” problem, an alternative way is to prove that the problem is “just as hard as” a large number of other problems that are widely recognize as being difficult and that have been confounding the experts for years.
- “I can’t find an efficient algorithm, but neither can all these famous people. ”



## Trial 3: to show the relative hardness with other hard problems. cont'd

- Two advantages:
  - 1 At the very least, this would inform your boss that it would do no good to fire you and hire another expert on algorithms.
  - 2 More importantly, you can spend your time looking for efficient algorithms that solve various special cases of the general problem.

## Problem and its hardness



# Hardness or complexity: an intrinsic property of problem

- Problems can be:
  - Easy (existing polynomial-time algorithm):  
STABLEMATCHING problem;
  - NP-hard: SATISFIABILITY problem;
  - Truly hard (provably non-polynomial, exponential): Given a Turing machine, does it halt in at most  $k$  steps?
  - Impossible: HALT problem

- Problem: consisting of INPUT and OUTPUT parts.  
Formally, a problem can be described as a *relation*  $P \subseteq I \times S$ , where  $I$  denotes the **problem input**, and  $S$  denotes the set of problem **solutions**.
- Instance: a particular INPUT.

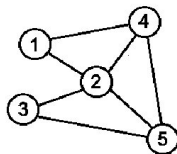
# An example

- S-T CONNECTIVITY problem

**INPUT:** a graph  $G = \langle V, E \rangle$ , two vertices  $s$ , and  $t$ ;

**OUTPUT:** a path from  $s$  to  $t$ ; ( or “NO” if there is no such path)

- Instance: a particular INPUT, including  $G$ , and  $s, t$ .



# Two types of problems: Optimization problem

- Optimization problem: given an instance  $x \in I$ , to find the “best” solution  $y^*$  according to some measure.
- Example: SHORTEST-PATH problem

**INPUT:** a graph  $G = \langle V, E \rangle$ , two vertices  $s$ , and  $t$ ;

**OUTPUT:** the shortest path from  $s$  to  $t$ , or “NO” if there is no path from  $s$  to  $t$ ;

# Two types of problems: Decision problem

- Decision problem: the relation  $P \subseteq I \times S$  reduces to a function  $f : I \mapsto S$ , where  $S = \{\text{YES}, \text{NO}\}$ .
- Example: PATH problem

**INPUT:** a graph  $G = \langle V, E \rangle$ , two vertices  $s$ , and  $t$ , and an integer  $k$ ;

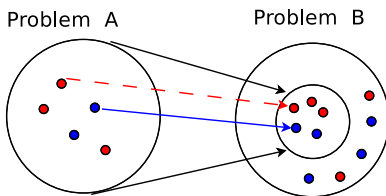
**OUTPUT:** a path from  $s$  to  $t$  with length at most  $k$ ;

# The relationship between decision problem and optimization problem

- The decision problem is in a sense “easier”.
- An optimization problem can be cast to a related decision problem by **imposing a bound of the value to be optimized.**
- For example, we can solve the PATH problem by solving the SHORTEST PATH problem, and then comparing the length of the shortest path with the decision problem parameter  $k$ .

# Reduction: uncovering the connection between two problems

- **POLYNOMIAL-TIME REDUCTION**: a procedure  $f$  to transform **any** instance  $\alpha$  of problem  $A$  to some instance  $\beta = f(\alpha)$  of problem  $B$  with the following characteristics:
  - 1 **Transformation**: The transformation takes polynomial time;
  - 2 **Equivalence**: The answers are the same, i.e. the answer for  $\alpha$  is YES iff the answer to  $\beta = f(\alpha)$  is also YES.
- Denoted as  $A \leq_P B$ , read as “**A is reducible to B**”.



# The significance of polynomial-time reduction

## Theorem

*If  $B$  can be solved in polynomial time,  $A$  is also polynomially solvable.*

The algorithm to problem  $A$  is described as follows:

- 1: Given an instance  $\alpha$  of problem  $A$ , use polynomial-time reduction to transform it to  $\beta = f(\alpha)$ ;
- 2: Run the polynomial-time algorithm for  $B$  on the instance  $\beta = f(\alpha)$ ;
- 3: Use the answer to  $\beta$  as the answer to  $\alpha$ .

## Theorem

*Conversely, if  $A$  is hard, then  $B$  is hard, too.*



A simple reduction:  $\text{INDEPENDENTSET} \leq_P \text{VERTEXCOVER}$

# INDEPENDENT SET Problem

- Practical problem:

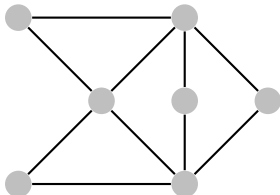
*Suppose you have  $n$  friends, and some pairs of them don't get along. How to invite at least  $k$  of them to dinner if you don't want any interpersonal tension?*

## Formalized Definition:

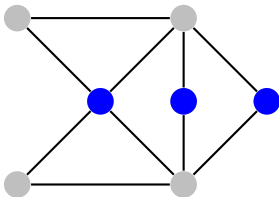
**Input:** Given a graph  $G = \langle V, E \rangle$ , and an integer  $k$ ,

**Output:** is there a set of nodes  $S \subseteq V$ ,  $|S| = k$ , such that no two nodes in  $S$  are joined by an edge?

- An example: are there 3 independent nodes in the following graph?

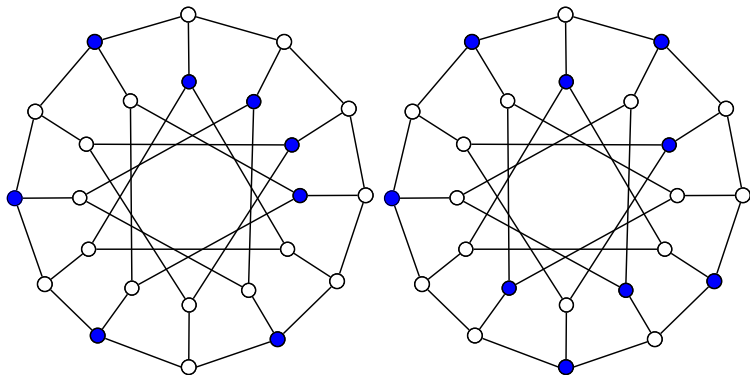


## An example



- The three nodes in blue are independent.

## Independent Set – another interesting instance



The nodes in blue form an independent set. Left: 8 nodes. Right: 9 nodes.

# VERTEX COVER Problem

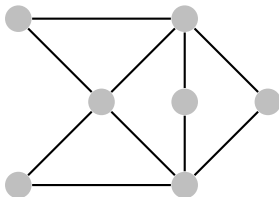
- Practical problem:  
*Given  $n$  sites connected with paths, how many guards (or cameras) should be deployed on sites to surveille **all** the paths?*

## Formalized Definition:

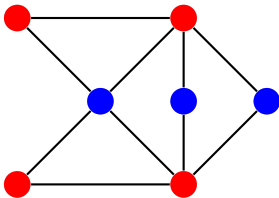
**Input:** Given a graph  $G = \langle V, E \rangle$ , and an integer  $k$ ,

**Output:** is there a set of nodes  $S \subseteq V$ ,  $|S| = k$ , such that each edge has at least one of its endpoints in  $S$ ?

- An example: are there 4 nodes to cover all edges in the following graph?



## An example



- Observation: the complement of an independent set (in blue) forms a vertex cover (in red).

# Reduction: INDEPENDENT SET $\leq_P$ VERTEX COVER

- **Transformation:** map an INDEPENDENT SET instance  $\langle G, k \rangle$  to a VERTEX COVER instance  $\langle G', k' \rangle$ , where  $G' = G$ , and  $k' = n - k$ ;

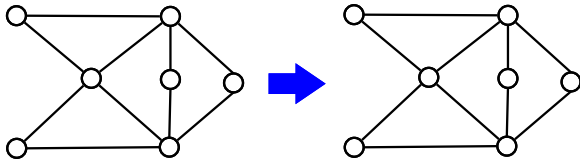


Figure: Transformation from an INDEPENDENT SET instance  $(G, k = 3)$  into a VERTEX COVER instance  $(G' = G, k' = 4)$

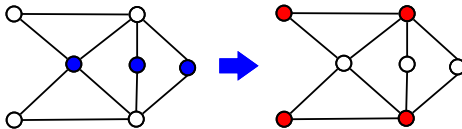
# Reduction: INDEPENDENT SET $\leq_P$ VERTEX COVER

## Theorem

**Equivalence:**  $G$  has an independent set  $S$  ( $|S| = k$ )  $\Leftrightarrow G'$  has a vertex cover  $S'$  ( $|S'| = n - k$ ).

## Proof.

- Let  $S$  be an independent set of  $G$  (in blue);
- For an arbitrary edge  $e = (u, v)$ , we have  $u \notin S$  or  $v \notin S$ ;
- Thus,  $u \in V - S$  or  $v \in V - S$ ;
- Define  $S' = V - S$  (in red).  $S'$  is a vertex cover of  $G' = G$ , and  $|S'| = n - k$ .



**Figure:** The complement of an independent set (in blue) form a vertex



# How to do reduction?

C. Papadimitriou said:

*To show the problem NP-complete we start by toying with small instances of the problem,*

*until we isolate one with an interesting behavior.*

*Sometimes the properties of this instance immediately enable a simple NP-complete proof .... sometimes called “gadget construction”...*

(Excerpted from Computer and Complexity)

Another simple reduction: VERTEX COVER  $\leq_P$  SET COVER.

- Practical problem:

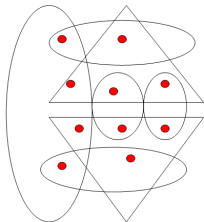
*An anti-virus package identifies a virus based on its characteristic “keywords” set. A keyword might correspond to several viruses. To reduce the size of anti-virus software, it is interesting to detect all viruses using a set of “representative” keywords rather than all keywords.*

## Formalized Definition:

**Input:** a set  $U$  of  $n$  elements, a collection  $S_1, S_2, \dots, S_m$  of subsets of  $U$ , and a number  $k$ .

**Output:** does there exist a collection of at most  $k$  of these sets whose union is equal to  $U$ ?

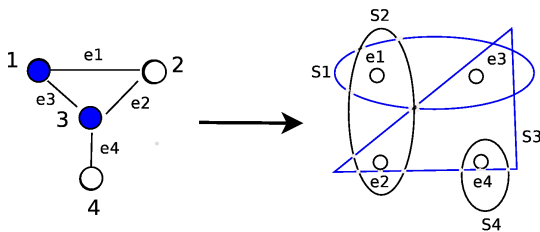
# SET COVER problem: an interesting instance



In this instance, there is a collection of three of the sets whose union is equal to all of  $U$ : We can choose the tall thin oval on the left, together with the two polygons.

# Reduction: VERTEX COVER $\leq_P$ SET COVER

- Key observation: a special case of SET COVER, where each element is covered by exactly two subsets, is in fact VERTEX COVER.
  - Transformation: Given a VERTEX COVER instance  $\langle G, k \rangle$ , create a SET COVER instance:  $k' = k$ ,  $U = E$ ,  $S_v = \{ e : e \text{ incident to } v \}$ ;



- Equivalence:  $G$  has a vertex cover  $C$  ( $|C| = k$ ) (in blue)  $\Leftrightarrow S_c$  ( $c \in C$ ) describe a set cover (in blue).

A reduction via “Gadget”:  $3\text{-SAT} \leq_P \text{INDEPENDENT SET}$ .

# SAT (SATISFIABILITY) Problem

- Practical problems:  
*expressing constraints on a set of variables (in AI), verifying whether a circuit has the desired functionality (in VLSI), etc.*

## Formalized Definition:

**Input:** Given a CNF  $\phi = C_1 \wedge C_2 \dots \wedge C_k$ ;

**Output:** Is there an assignment of all  $x_i$  such that all clauses  $C_j$  are satisfied?

- Notations:
  - Boolean variable:  $x_1, x_2, \dots, x_n, x_i = \text{TRUE/FALSE}$ ;
  - Literal (or term): a variable  $x_i$ , or its negative  $\neg x_i$ ;
  - Clause: a disjunction of literals:  $C_1 = x_1 \vee \neg x_2 \vee \dots \vee x_3$ ;
  - CNF( Conjunctive normal form): the conjunctions of clauses;  
 $\phi = C_1 \wedge C_2 \dots \wedge C_k$ ;
- Example:
  - CNF:  $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_2 \vee \neg x_3)$
  - TRUE assignment:  $x_1 = \text{FALSE}, x_2 = \text{FALSE}, x_3 = \text{FALSE}$ ;

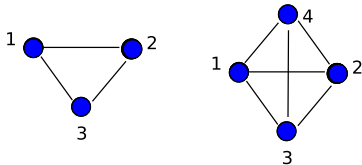
# SAT (SATISFIABILITY) Problem: Two viewpoints of TRUE assignment

- Example:
  - CNF:  $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_2 \vee \neg x_3)$
  - True assignment:  $x_1 = \text{FALSE}, x_2 = \text{FALSE}, x_3 = \text{FALSE}$ ;
- Two viewpoints of a TRUE assignment:
  - 1 Variables: giving each variable a TRUE/FALSE to satisfy all clauses;
  - 2 Clauses: In each clause, we select a literal and set it to be TRUE to make the clause satisfied. However, there should be no conflict among the selected literals from different clauses, e.g., select  $x_i$  from one clause and select  $\neg x_i$  from another one.



# INDEPENDENT SET: designing gadget

- **Gadget**: a small, useful, and cleverly-designed machine or tool; a piece with specific functionality, which can be used to simulate another problem, say, to simulate **variables** and **clauses** in SAT problem.
- For example, in INDEPENDENT SET problem, clique is a gadget with functionality **OR**. Consider the three nodes in the following INDEPENDENT SET instance:



- We can **choose 1 OR choose 2 OR choose 3** since only one vertex of a clique can appear in an independent set.
- Thus, we can use it to simulate the **OR** operator in a clause.

# $3SAT \leq_P$ INDEPENDENT SET: Transformation

- **Transformation:**

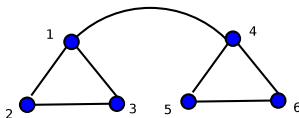
- For a given  $SAT$  instance  $\phi$  with  $k$  clauses, constructing an INDEPENDENT SET instance  $(G, k')$  as follows:
  - ①  $G$  consists of  $k$  triangles: each triangle corresponds to a clause  $C_i$ ; the nodes are labeled with the literals; connecting  $x_i$  and  $\neg x_i$  with an edge;
  - ② Set  $k' = k$ ;
- Example:

**A SAT Instance**

( X1 OR X2 OR X3 ) AND  
( NOT X1 OR X5 OR X6 )



**Independent Set Instance**



- Intuition: edge represents “conflicts”; we should identify  $k$  nodes (each node from a triangle) without connections (no conflict);

# $3SAT \leq_P$ INDEPENDENT SET

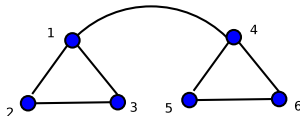
- **Equivalence:**

**A SAT Instance**

( X1 OR X2 OR X3 ) AND  
( NOT X1 OR X5 OR X6 )



**Independent Set Instance**



## Proof.

- Suppose  $\phi$  is satisfiable;
- There is an assignment such that in each clause, at least one literal is satisfied;
- Choose exactly one satisfied literal from each clause;
- The corresponding nodes form an independent set;



# $3SAT \leq_P$ INDEPENDENT SET: Equivalence

- Equivalence:

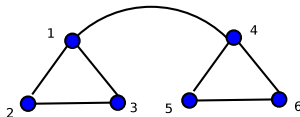
**A SAT Instance**

(  $X_1$  OR  $X_2$  OR  $X_3$  ) AND

( NOT  $X_1$  OR  $X_5$  OR  $X_6$  )



**Independent Set Instance**



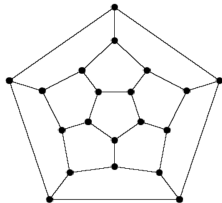
Proof.

- Suppose  $S$  is an independent set, and  $|S| = k$ ;
- $S$  contains exactly one node from each triangle; (?)
- Constructing an assignment: if  $v_i \in S$ ,  $x_i = TRUE$ , and  $x_i = FALSE$  otherwise;
- Notice that  $v_i$  and  $\neg v_i$  would not appear in  $S$  simultaneously.
- $\phi$  is satisfied by this assignment.



Another reduction via “gadget”:  $\text{SAT} \leq_P \text{HAMILTON CYCLE}$ .

# HAMILTON CYCLE Problem



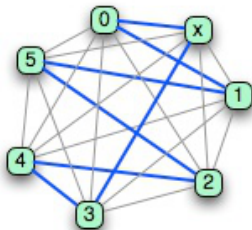
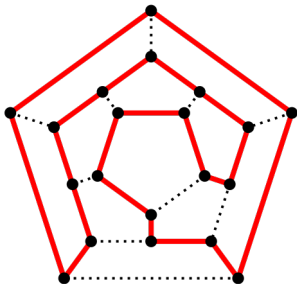
A game invented by Sir William Hamilton in 1857.

**Formalized Definition:**

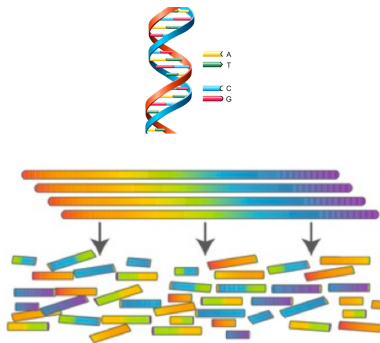
**Input:** Given a graph  $G = \langle V, E \rangle$

**Output:** Is there a cycle visiting every node exactly once?

# HAMILTON CYCLE: two examples



# DNA sequencing: an application of HAMILTONIAN CYCLE

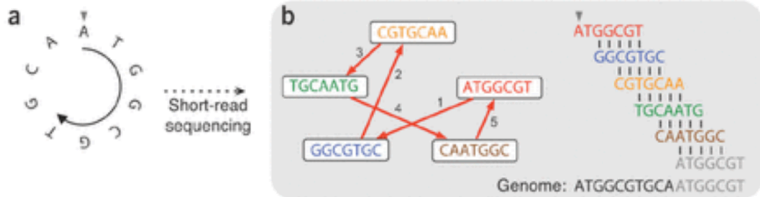


- Multiple copies of a DNA  $\Rightarrow$  small sequenced fragments called reads (say 500 bp).
- Challenge: how to restore the whole genome from the short fragments?

(see <http://www.learner.org/courses/mathilluminated/interactives/dna/> for an animation)



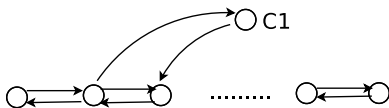
# HAMILTONIAN CYCLE and genome assembly



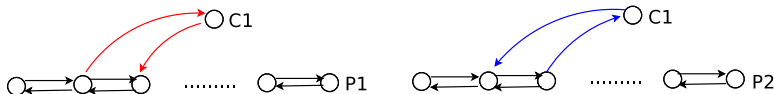
- Let's construct a graph as follows:
  - node: a short fragment
  - edge: if two fragments overlap, then an edge is added between the corresponding nodes;
- Observation: HAMILTONIAN CYCLE  $\Leftrightarrow$  the genome

# HAMILTON CYCLE problem: designing gadget

- Consider the following graph:



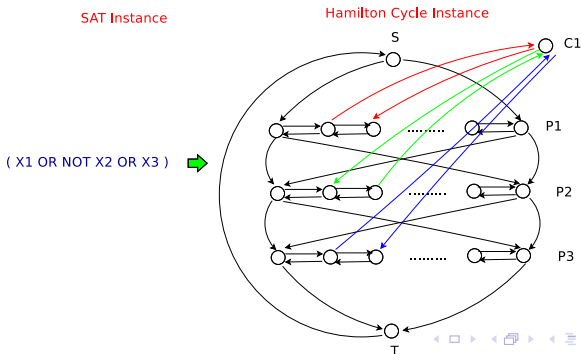
- Suppose that we were required to visit all nodes. It is obvious that:
  - To visit all nodes in the line, we can visit from left to right, **OR** from right to left.
  - However, in order to visit node  $C_i$ , the connection manner of  $C_i$  defines the possible traveling direction of the line of nodes. For example, if we want to visit node  $C_1$ , we have to travel the line  $P_1$  from left to right, or travel  $P_2$  from right to left.



# Reduction $3SAT \leq_P \text{HAMILTON CYCLE}$

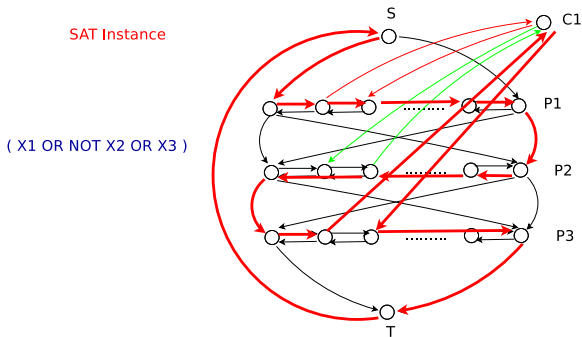
- **Transformation:**

- For a given SAT instance  $\phi$ , we construct a HAMILTON CYCLE instance  $G$  as follows:
  - 1 Variable  $\Rightarrow$  a line of nodes for each variable;
  - 2 Clause  $\Rightarrow$  a special node  $C_i$ .  $C_i$  connects to the line  $j$  in “clockwise” direction if it contains  $x_j$  and connects in “counter-clockwise” direction if it contains  $\neg x_j$ .
  - 3 Two special nodes  $s$  and  $t$ :



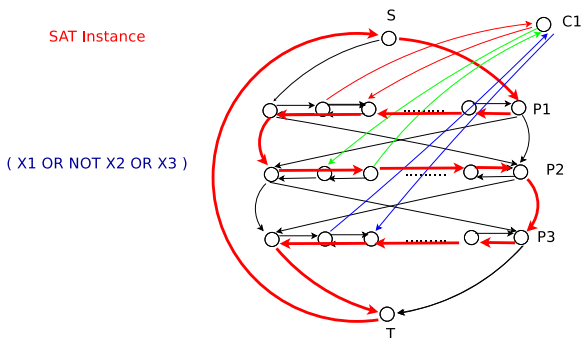
# How does a gadget work?

- TRUE assignment  $\Rightarrow$  a cycle. We travel line  $P_i$  from left to right if  $x_i = \text{TRUE}$ , and travel from right to left if  $x_i = \text{FALSE}$ .
- For example:  $x_1 = \text{TRUE}$ ,  $x_2 = \text{FALSE}$ ,  $x_3 = \text{TRUE}$



# How does a gadget work?

- FALSE assignment  $\Rightarrow$  cannot visit  $C$  node. We travel line  $P_i$  from left to right if  $x_i = \text{TRUE}$ , and travel from right to left if  $x_i = \text{FALSE}$ .
- For example:  $x_1 = \text{FALSE}$ ,  $x_2 = \text{TRUE}$ ,  $x_3 = \text{FALSE}$ . We have no chance to visit  $C$ .

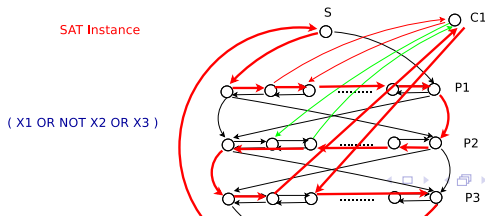


# Reduction $3SAT \leq_P \text{HAMILTON CYCLE}$

- **Equivalence:**

Proof.

- Suppose  $\phi$  can be satisfied by an assignment;
- Starting from  $s$ ; travel line  $i$  from left to right if  $x_i = \text{TRUE}$ ; otherwise from right to left;
- If  $C_j$  is satisfied by literal  $x_i$  or  $\neg x_i$ , then travel  $C_j$  when traveling line  $i$ ;
- Return to  $s$  from  $t$  finally;
- This way, all nodes will be visited exactly once.

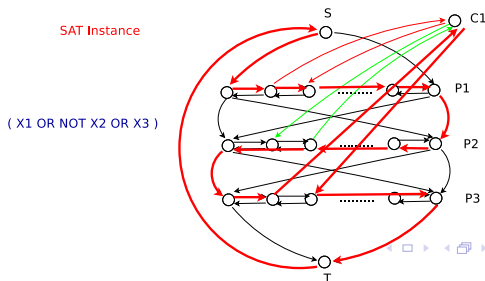


# Reduction $3SAT \leq_P \text{HAMILTON CYCLE}$ : Equivalence

- **Equivalence:**

Proof.

- Suppose there is a Hamilton cycle;
- Assign  $x_i = \text{TRUE}$  iff line  $i$  are visited from left to right;
- This assignment satisfies  $\phi$ ;
- Each clause  $C_j$  is satisfied. Why?

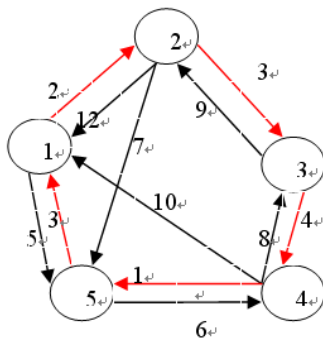


A simple reduction:  $\text{HAMILTON CYCLE} \leq_P \text{TSP (TRAVELING SALESMAN PROBLEM)}$  .



# TSP (TRAVELING SALESMAN PROBLEM)

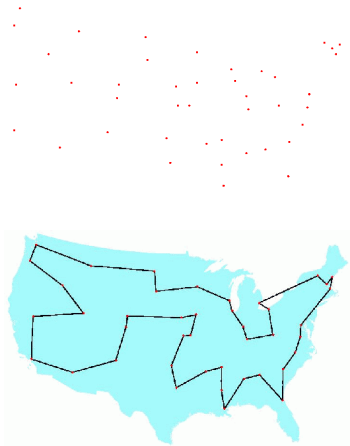
Intuition: A salesman tries to travel  $n$  cities with the shortest tour.



The tour path with the shortest length is 1, 2, 3, 4, 5, 1 with red color.

# TSP (TRAVELING SALESMAN PROBLEM)

Optimal tour of 49 cities in the USA (solved by Dantzig, Fulkerson, and Johnson in 1954).



Origin: Karl Menger (1920), Mahalanobis (1940), Jensen (1942), Gosh (1948), Marks (1948)

# TSP (TRAVELING SALESMAN PROBLEM) RESEARCH PROGRESS

Year	Research Team	Size of Instance	Name
1954	G. Dantzig, R. Fulkerson, and S. Johnson	49 cities	dantzig42
1971	M. Held and R.M. Karp	64 cities	64 random poi
1975	P.M. Camerini, L. Fratta, and F. Maffioli	67 cities	67 random poi
1977	M. Grötschel	120 cities	gr120
1980	H. Crowder and M.W. Padberg	318 cities	lin318
1987	M. Padberg and G. Rinaldi	532 cities	att532
1987	M. Grötschel and O. Holland	666 cities	gr666
1987	M. Padberg and G. Rinaldi	2,392 cities	pr2392
1994	D. Applegate, R. Bixby, V. Chvátal, and W. Cook	7,397 cities	pla7397
1998	D. Applegate, R. Bixby, V. Chvátal, and W. Cook	13,509 cities	usa13509
2001	D. Applegate, R. Bixby, V. Chvátal, and W. Cook	15,112 cities	d15112
2004	D. Applegate, R. Bixby, V. Chvátal, W. Cook, and K. Helsgaun	24,978 cities	sw24798

(Excerpted from

<http://www.tsp.gatech.edu/history/milestone.html>)

- Practical problem:  
path planning: the most efficient motion of a robotic arm that drill  $n$  holes on a VLSI chip;

## Formalized Definition:

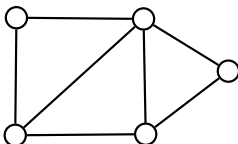
**Input:** Given a graph  $G = \langle V, E \rangle$ , distance  $d : E \rightarrow R$ , and a bound  $B$ ;

**Output:** is there a Hamilton cycle with total distance less than  $B$ ?

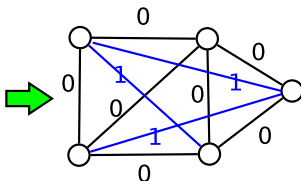
# Reduction: HAMILTON CYCLE $\leq_P$ TSP

- **Transformation:** for a HAMILTON CYCLE instance  $G = \langle V, E \rangle$ , we construct a TSP instance as follows:  $G'$ : a complete graph with  $n$  node,  $d(u, v) = 0$  if  $(u, v) \in E$ ; otherwise  $d(u, v) = +\infty$ . Let  $B = 0$ ;

Hamilton Cycle  
Instance



TSP Instance



- **Equivalence:** a tour with total distance  $\leq 0$  in  $G'$  corresponds to a Hamilton cycle in  $G$ .

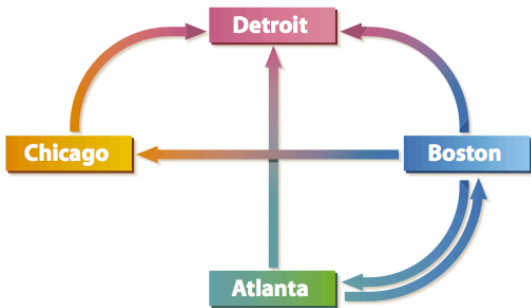
Extended reading: Computing Hamiltonian path using DNA computer

# Molecular Computation of Solutions to Combinatorial Problems, Leonard M. Adleman [1994]



- The tools of molecular biology were used to solve an instance of the directed Hamiltonian path problem.
- A small graph was encoded in molecules of DNA, and the "operations" of the computation were performed with standard protocols and enzymes.
- This experiment demonstrates the feasibility of carrying out computations at the molecular level.

# Representing a Hamiltonian instance using DNA

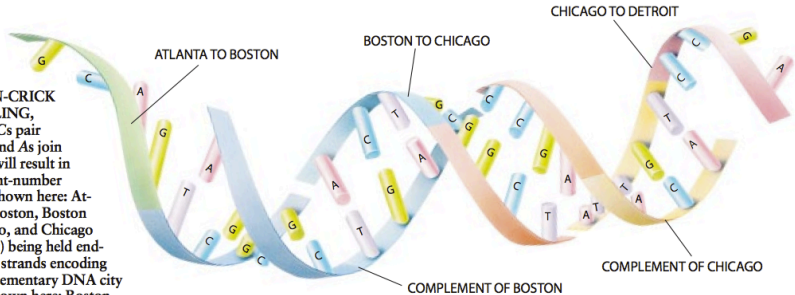


CITY	DNA NAME	COMPLEMENT
ATLANTA	ACTTGCAG	TGAACGTC
BOSTON	TCGGACTG	AGCCTGAC
CHICAGO	GGCTATGT	CCGATACA
DETROIT	CCGAGCAA	GGCTCGTT
FLIGHT		DNA FLIGHT NUMBER
ATLANTA - BOSTON		GCAGTCGG
ATLANTA - DETROIT		GCAGCCGA
BOSTON - CHICAGO		ACTGGGCT
BOSTON - DETROIT		ACTGCCGA
BOSTON - ATLANTA		ACTGACTT
CHICAGO - DETROIT		ATGTCCGA



# A path generated during DNA synthesis process

**WATSON-CRICK ANNEALING**, in which Cs pair with Gs and As join with Ts, will result in DNA flight-number strands (shown here: Atlanta to Boston, Boston to Chicago, and Chicago to Detroit) being held end-to-end by strands encoding the complementary DNA city names (shown here: Boston and Chicago).



A simple reduction:  $\text{SAT} \leq_P \text{GRAPH COLORING}$

# GRAPH COLORING problem

- Practical problem:

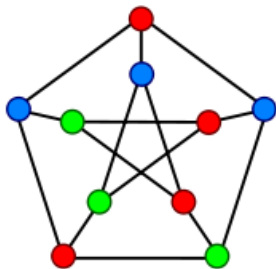
*Consider assigning one of the  $k$  wavelengths to  $n$  wireless devices. If two devices are sufficiently close to each other, we should assign them with different wavelengths to prevent interference.*

## Formalized Definition:

**Input:** A graph  $G = \langle V, E \rangle$ , an integer  $k$ ;

**Output:** Is there a  $k$  – coloring of  $G$  such that each node has a color, but the two endpoints of an edge have different colors?

# An example



A proper vertex coloring of the Petersen graph with 3 colors, the minimum number possible.

# Graph coloring: a brief introduction

There are three types of coloring:

- 1 Vertex coloring: coloring the vertices of a graph such that no two adjacent vertices share the same color;
- 2 Edge coloring: coloring edges so that no two adjacent edges share the same color
- 3 Face coloring: (planar graph) coloring faces or region so that no two faces that share a boundary have the same color.

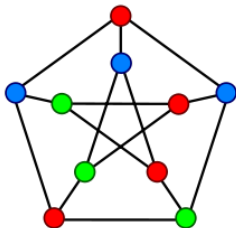


Figure: Peterson graph

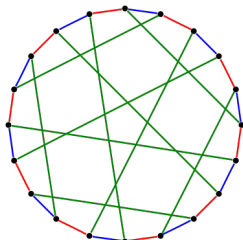


Figure: Desargues graph:  
the complement of Peterson graph

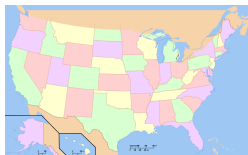
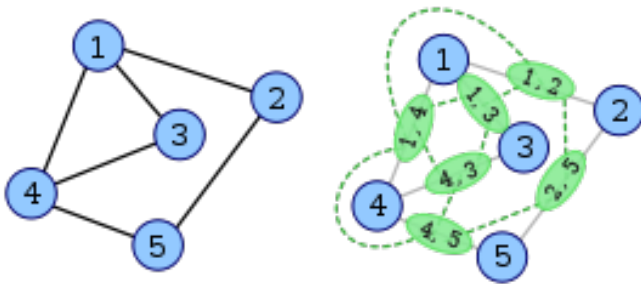




Figure: USA map

# Transforming EDGE COLORING into VERTEX COLORING

Edge coloring  $\Rightarrow$  vertex coloring (line graph ): an edge coloring of a graph is just a vertex coloring of its line graph.

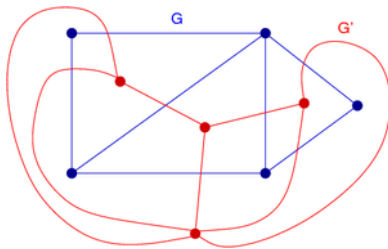


3

<sup>3</sup>The 3 slides were excerpted from Wiki::Graph Coloring  

# Transforming FACE COLORING into VERTEX COLORING

Face coloring  $\Rightarrow$  vertex coloring (graph dual ): a face coloring of a planar graph is just a vertex coloring of its planar dual.



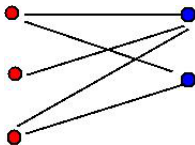
# Planar graph: 4 coloring

- The four-color theorem was proven in 1976 by Kenneth Appel and Wolfgang Haken.
- It was the first major theorem to be proved using a computer.
- Appel and Haken's approach started by showing that there is a particular set of 1,936 maps, each of which cannot be part of a smallest-sized counterexample to the four color theorem.
- Appel and Haken used a special-purpose computer program to confirm that each of these maps had this property.



## General graph: 2-coloring.

- 2-coloring is in  $P$ .
- A graph  $G$  can be 2-colored iff  $G$  is a bi-partite.



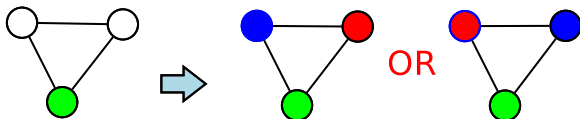
## General graph: $k$ -coloring ( $k \geq 3$ )

- Using dynamic programming and a bound on the number of maximal independent sets,  $k$ -colorability can be decided in time and space  $O(2.445^n)$ .
- Using the principle of inclusionexclusion and Yates algorithm for the fast zeta transform,  $k$ -colorability can be decided in time  $O(2^n n)$  for any  $k$ .
- Faster algorithms are known for 3- and 4-colorability, which can be decided in time  $O(1.3289^n)$  and  $O(1.7504^n)$ , respectively.

# Designing gadget: triangle

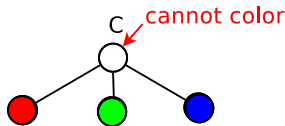
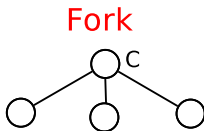
- **Triangle:** Suppose a node has already been colored in Green, one of the other nodes should be Red OR Blue;
- Hint: can be used to express a Boolean variable since  $x_i = \text{TRUE OR FALSE}$ . For example: Red: True, Blue: False.

Triangle



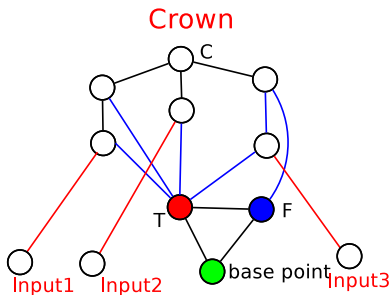
# Designing gadget: fork

- **Fork:** If the three endpoints use all RGB colors, the root cannot be colored.

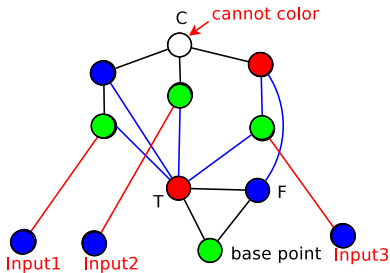


# Designing gadget: crown

- **Crown:**  $C$  can be colored iff one of the three input is colored in Red.
- Hint: can be used to express a clause since at least one of the three literals should be TRUE.

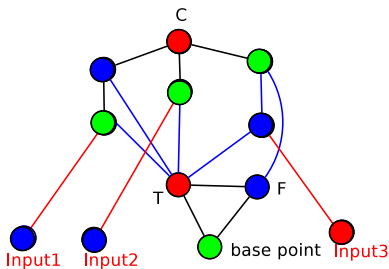


## Case 1: the three input of crown = BBB



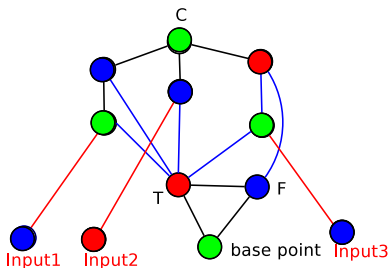
$I_1 = B, I_2 = B, I_3 = B \Rightarrow C$  cannot be colored.

## Case 2: the three input of crown = BBR



$I_1 = B, I_2 = B, I_3 = R \Rightarrow C$  can be colored.

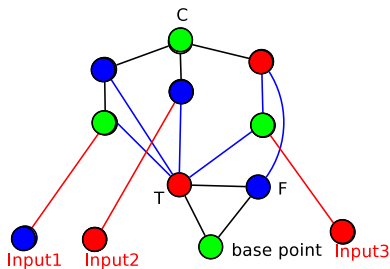
### Case 3: the three input of crown = BRB



$I_1 = B, I_2 = R, I_3 = B \Rightarrow C$  can be colored.

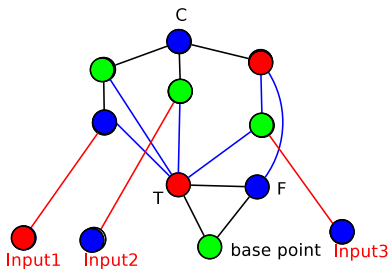


## Case 4: the three input of crown = BRR



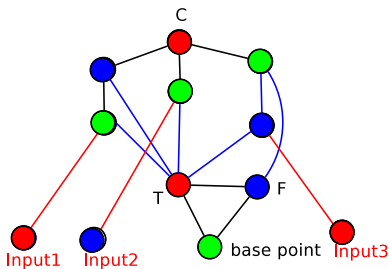
$I_1 = B, I_2 = R, I_3 = R \Rightarrow C$  can be colored.

## Case 5: the three input of crown = RBB



$I_1 = R, I_2 = B, I_3 = B \Rightarrow C$  can be colored.

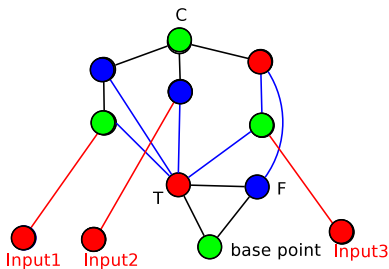
## Case 6: the three input of crown = RBR



$I_1 = R, I_2 = B, I_3 = R \Rightarrow C$  can be colored.



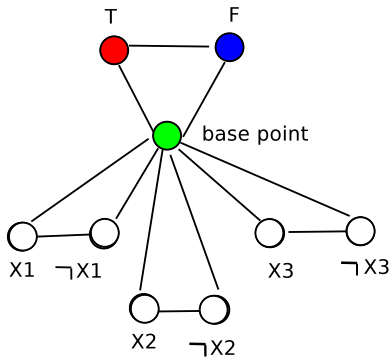
## Case 8: the three input of crown = RRR



$I_1 = R, I_2 = R, I_3 = R \Rightarrow C$  can be colored.

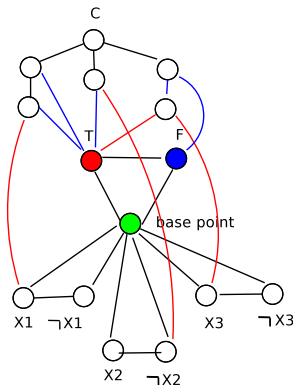
# $\text{SAT} \leq_P \text{3-COLORING}$ : Transformation

- Transformation: a variable  $\Rightarrow$  a triangle;



# SAT $\leq_P$ 3-COLORING: Transformation cont'd

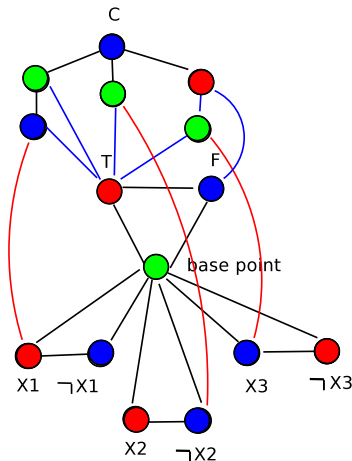
- Transformation: Clause  $\Rightarrow$  connecting inputs with literals.
- Example:  $C = (x_1 \vee \neg x_2 \vee x_3)$



Note: The node  $C$  can be colored iff one of the three input is colored in Red, i.e., at least one literal is satisfied.

# True assignment $\Rightarrow$ 3-Coloring

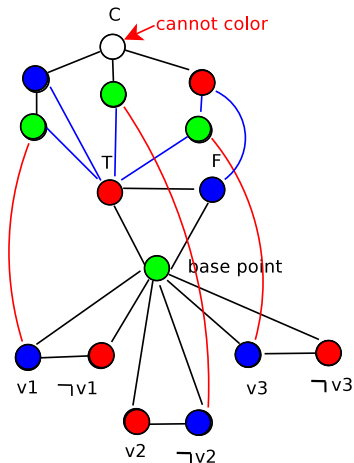
- Clause  $\Rightarrow$  connecting inputs with literals.
- Example:  $C = (x_1 \vee \neg x_2 \vee x_3)$
- True assignment:  $x_1 = T, x_2 = T, x_3 = F$ .





False assignment  $\Rightarrow$  No 3-Coloring

- Clause  $\Rightarrow$  connecting inputs with literals.
- Example:  $C = (x_1 \vee \neg x_2 \vee x_3)$
- False assignment:  $x_1 = F, x_2 = T, x_3 = F$ .



# Reduction: $SAT \leq_P 3\text{-COLORING}$

## Proof.

$\Rightarrow$

- Consider a true assignment;
- Color  $v_i$  Red if  $x_i = TRUE$ ; otherwise *Blue*;
- This is a 3-Coloring. (  $C$  can be colored unless ALL 3 input are in Blue. )

$\Leftarrow$

- Consider a 3-Coloring; (w.l.o.g, suppose node  $T$  is in Red,  $F$  is in Blue, and  $base$  is in Green.)
- Let  $x_i = TRUE$  if  $v_i$  is in Red; otherwise,  $x_i = FALSE$ ;
- This is a true assignment. (Each clause  $C_j$  has a satisfied term.)



Another reduction via gadget:  $\text{SAT} \leq_P \text{SUBSETSUM}$ .

# SUBSETSUM problem

## Formalized Definition:

**Input:** Given  $n$  numbers  $S = w_1, w_2, \dots, w_n$ , and an objective value  $W$ ;

**Output:** is there a subset  $S' \subseteq S$  such that the sum of  $S'$  is  $W$ ?

Example:  $S = \{1, 3, 5, 7, 11, 13, 17, 19\}$ ,  $W = 33$ ;

Solution:  $S' = \{3, 13, 17\}$

# SUBSETSUM problem: Gadget

- Suppose we are given a set of numbers as follows:

	0	X	X	X
	0	X	X	X
	.	.	.	.
	0	X	X	X
$w_1$	1	X	X	X
$w_2$	1	X	X	X
-----				
$W$	1	X	X	X

- Observation: we should choose either  $w_1$  OR  $w_2$ .
- Hint: can be used to express a Boolean variable.

# 3SAT $\leq_P$ SUBSETSUM: Transformation

- Variable  $x_i \Rightarrow$  two numbers  $v_i$  and  $v'_i$ . (Intuition: to assure that exactly one of  $v_i$  and  $v'_i$  should be selected. )
- Clause  $C_j \Rightarrow$  two numbers  $s_j$  and  $s'_j$ , and set the final column as follows:  $v_i = 1$  if  $C_j$  contains  $x_i$ , and  $v'_i = 1$  otherwise. (Intuition: the number denotes how many literals were satisfied.)

SAT Instance

$C1=(X1 \text{ OR } \text{NOT } X2 \text{ OR } X3)$

Subset Sum Instance

$S=\{1001, 1000, 100, 101, 11, 10, 1, 2 \}$   
 $W = 1114$

	x1	x2	x3	C1
v1	1	0	0	1 $\leftarrow$ X1
v1'	1	0	0	0
v2	0	1	0	0
v2'	0	1	0	1 $\leftarrow$ NOT X2
v3	0	0	1	1 $\leftarrow$ X3
v3'	0	0	1	0
s1	0	0	0	1
s1'	0	0	0	2
W	1	1	1	4

# Transformation: an example with two clauses

## SAT Instance

$C1 = (X1 \text{ OR NOT } X2 \text{ OR } X3)$

$C2 = (\text{NOT } X1 \text{ OR } X2 \text{ OR } X3)$



## Subset Sum Instance

$S = \{10010, 10001, 1001, 1010, 111, 100, 10, 20, 1, 2\}$

$W = 11144$

	x1	x2	x3	C1	C2
v1	1	0	0	1	0
v1'	1	0	0	0	1
v2	0	1	0	0	1
v2'	0	1	0	1	0
v3	0	0	1	1	1
v3'	0	0	1	0	0
s1	0	0	0	1	0
s1'	0	0	0	2	0
s2	0	0	0	0	1
s2'	0	0	0	0	2
W	1	1	1	4	4

# 3SAT $\leq_P$ SUBSETSUM: Equivalence

$\Rightarrow$

- Consider a true assignment;
- If  $x_i = TRUE$ , then select  $v_i$ ; otherwise select  $v'_i$ ;
- If  $C_j$  has 1 satisfied term, select  $s'_j$  and  $s_j$ ;
- if  $C_j$  has 2 satisfied terms, select  $s'_j$ ;
- if  $C_j$  has 3 satisfied terms, select  $s_j$ .
- The sum of subset is exactly  $W$ .



# Case 1: $x_1 = T$ , $x_2 = T$ , $x_3 = T$ (True assignment)

	x1	x2	x3	C1
v1	1	0	0	1
v1'	1	0	0	0
v2	0	1	0	0
v2'	0	1	0	1
v3	0	0	1	1
v3'	0	0	1	0
s1	0	0	0	1
s1'	0	0	0	2
W	1	1	1	4

True assignment  $x_1 = T$ ,  $x_2 = T$ ,  $x_3 = T \Rightarrow$   
 $v_1 + v_2 + v_3 + s_1' = W$ .

## Case 2: $x_1 = T$ , $x_2 = T$ , $x_3 = F$ (True assignment)

	x1	x2	x3	C1
v1	1	0	0	1
v1'	1	0	0	0
v2	0	1	0	0
v2'	0	1	0	1
v3	0	0	1	1
v3'	0	0	1	0
s1	0	0	0	1
s1'	0	0	0	2
W	1	1	1	4

True assignment  $x_1 = T$ ,  $x_2 = T$ ,  $x_3 = F \Rightarrow$   
 $v_1 + v_2 + v_3 + s_1 + s_1' = W$ .

### Case 3: $x_1 = T$ , $x_2 = F$ , $x_3 = T$ (True assignment)

	x1	x2	x3	C1
v1	1	0	0	1
v1'	1	0	0	0
v2	0	1	0	0
v2'	0	1	0	1
v3	0	0	1	1
v3'	0	0	1	0
s1	0	0	0	1
s1'	0	0	0	2
W	1	1	1	4

True assignment  $x_1 = T$ ,  $x_2 = F$ ,  $x_3 = T \Rightarrow$   
 $v_1 + v_2 + v_3 + s_1 = W$ .

# Case 4: $x_1 = T$ , $x_2 = F$ , $x_3 = F$ (True assignment)

	x1	x2	x3	C1
v1	1	0	0	1
v1'	1	0	0	0
v2	0	1	0	0
v2'	0	1	0	1
v3	0	0	1	1
v3'	0	0	1	0
s1	0	0	0	1
s1'	0	0	0	2
W	1	1	1	4

True assignment  $x_1 = T$ ,  $x_2 = F$ ,  $x_3 = F \Rightarrow$   
 $v_1 + v_2 + v_3 + s_1' = W$ .

# Case 5: $x_1 = F$ , $x_2 = T$ , $x_3 = T$ (True assignment)

	x1	x2	x3	C1
v1	1	0	0	1
v1'	1	0	0	0
v2	0	1	0	0
v2'	0	1	0	1
v3	0	0	1	1
v3'	0	0	1	0
s1	0	0	0	1
s1'	0	0	0	2
W	1	1	1	4

True assignment  $x_1 = F$ ,  $x_2 = T$ ,  $x_3 = T \Rightarrow$   
 $v_1 + v_2 + v_3 + s_1 + s_1' = W$ .

# Case 6: $x_1 = F$ , $x_2 = F$ , $x_3 = T$ (True assignment)

	x1	x2	x3	C1
v1	1	0	0	1
v1'	1	0	0	0
v2	0	1	0	0
v2'	0	1	0	1
v3	0	0	1	1
v3'	0	0	1	0
s1	0	0	0	1
s1'	0	0	0	2
W	1	1	1	4

True assignment  $x_1 = F$ ,  $x_2 = F$ ,  $x_3 = T \Rightarrow$   
 $v_1 + v_2 + v_3 + s'_1 = W$ .

# Case 7: $x_1 = F$ , $x_2 = F$ , $x_3 = F$ (True assignment)

	x1	x2	x3	C1
-----				
v1	1	0	0	1
v1'	1	0	0	0
v2	0	1	0	0
v2'	0	1	0	1
v3	0	0	1	1
v3'	0	0	1	0
s1	0	0	0	1
s1'	0	0	0	2
-----				
W	1	1	1	4

True assignment  $x_1 = F$ ,  $x_2 = F$ ,  $x_3 = F \Rightarrow$   
 $v_1 + v_2 + v_3 + s_1 + s'_1 = W$ .

# Case 8: $x_1 = F$ , $x_2 = T$ , $x_3 = F$ (False assignment)

	x1	x2	x3	C1
v1	1	0	0	1
v1'	1	0	0	0
v2	0	1	0	0
v2'	0	1	0	1
v3	0	0	1	1
v3'	0	0	1	0
s1	0	0	0	1
s1'	0	0	0	2
W	1	1	1	4

False assignment  $x_1 = F$ ,  $x_2 = T$ ,  $x_3 = F \Rightarrow$  the sum of the lowest digit is 0, i.e.,  $v'_1 + v_2 + v'_3 = 1110$ . We cannot get a sum of 1114 even if we choose  $s_1$  plus  $s'_1$ .



$\Leftarrow$

- Consider a subset  $S'$ ;
- If  $v_i$  is selected, set  $x_i = T$ , and  $x_i = F$  otherwise;
- This is a true assignment. (?)
- (Hint: the lowest digit of  $W$  is 4, while the lowest digit of  $s_1 + s'_1$  is 3. This means that at least one “1” was selected in the final column. )

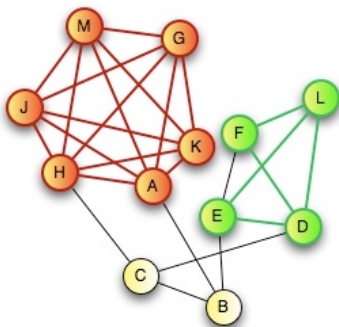
INDEPENDENTSET  $\leq_P$  CLIQUE

# CLIQUE problem

## Formalized Definition:

**Input:** Graph  $G = \langle V, E \rangle$ , an integer  $k$ ; **Output:** is there a clique of size  $k$ ? Here, a clique refers to a subset of vertices that are all connected.

Example: a graph having a clique of size 5.



- Transformation: map an INDEPENDENTSET instance  $\langle G, k \rangle$  to a CLIQUE instance  $\langle G', k' \rangle$ , where  $G'$  is the complement of  $G$ , and  $k' = k$ .
- Equivalence:  $G$  has an independent set of size  $k$  iff  $G'$  has a clique of size  $k$ .

## Easy problems vs. hard problems

# Easy problem vs. hard problems

Easy problems	Hard problems
BIPARTITEMATCHING	3D MATCHING
3SAT	2SAT
SHORESTPATH	LONGESTPATH
LINEARPROGRAMING	INTEGERLINEARPROGRAMING
MINIMUMSPANNINGTREE	TSP
MINCUT	BALANCEDCUT

So far we have the following relative hardness results.

