NP-Completeness

CS 3110

Textbook readings:

Chapter 8
Chapter 9

Overview

- Polynomial-time algorithms
- Optimization problems vs. decision problems
- The class P
- Decision vs. verification
- The class NP
- Polynomial-time reductions
- NP-completeness

NP-complete problems:

- Satisfiability
- Vertex cover
- Hamiltonian cycle
- Subset sum

Polynomial-Time Algorithms

A ***polynomial-time algorithm*** is an algorithm that runs in $\mathcal{O}(n^k)$ time, for some constant k, where n is the size of the input.

Polynomial-Time Algorithms

A ***polynomial-time algorithm*** is an algorithm that runs in $\mathcal{O}(n^k)$ time, for some constant k, where n is the size of the input.

Examples:

- Quicksort (sorting): $\mathcal{O}(n \lg n)$
- Strassen's algorithm (matrix multiplication): $\mathcal{O}(n^{2.81})$
- Bellman-Ford algorithm (shortest paths): $\mathcal{O}(|V||E|) = \mathcal{O}(n^2)$
- ...

Polynomial-Time Algorithms

A ***polynomial-time algorithm*** is an algorithm that runs in $\mathcal{O}(n^k)$ time, for some constant k, where n is the size of the input.

Examples:

- Quicksort (sorting): $\mathcal{O}(n \lg n)$
- Strassen's algorithm (matrix multiplication): $\mathcal{O}(n^{2.81})$
- Bellman-Ford algorithm (shortest paths): $\mathcal{O}(|V||E|) = \mathcal{O}(n^2)$
- ...

Problems with polynomial-time algorithms are considered ***tractable***.

Polynomial-Time Algorithms

A ***polynomial-time algorithm*** is an algorithm that runs in $\mathcal{O}(n^k)$ time, for some constant k, where n is the size of the input.

Examples:

- Quicksort (sorting): $\mathcal{O}(n \lg n)$
- Strassen's algorithm (matrix multiplication): $\mathcal{O}(n^{2.81})$
- Bellman-Ford algorithm (shortest paths): $\mathcal{O}(|V||E|) = \mathcal{O}(n^2)$
- ...

Problems with polynomial-time algorithms are considered ***tractable***.

Problems without such algorithms are considered ***intractable***.

Optimization Problems vs. Decision Problems (1)

Many hard problems are optimization problems:

- Given a text to be encoded, find a code that encodes this text in the minimum number of bits.
- Given a graph G, find a minimum-size vertex set S such that every edge has at least one endpoint in S.

Optimization Problems vs. Decision Problems (1)

Many hard problems are optimization problems:

- Given a text to be encoded, find a code that encodes this text in the minimum number of bits.
- Given a graph G, find a minimum-size vertex set S such that every edge has at least one endpoint in S.

Decision problems are problems where the answer is "yes" or "no".

Optimization Problems vs. Decision Problems (1)

Many hard problems are optimization problems:

- Given a text to be encoded, find a code that encodes this text in the minimum number of bits.
- Given a graph G, find a minimum-size vertex set S such that every edge has at least one endpoint in S.

Decision problems are problems where the answer is "yes" or "no".

Every optimization problem has a corresponding decision problem:

- Given a text and an integer k, decide whether there exists a code that encodes the text in at most k bits.
- Given a graph G and an integer k, decide whether there exists a set S of at most k vertices such that every edge has at least one vertex in S.

Optimization Problems vs. Decision Problems (2)

Lemma: *If an optimization problem is solvable in polynomial time, the corresponding decision problem is also solvable in polynomial time.*

Optimization Problems vs. Decision Problems (2)

Lemma: *If an optimization problem is solvable in polynomial time, the corresponding decision problem is also solvable in polynomial time.*

Corollary: *If we can provide evidence that a certain decision problem is unlikely to have a polynomial-time algorithm, the corresponding optimization problem is equally unlikely to have a polynomial-time algorithm.*

The Complexity Class P

The complexity class ***P*** consists of all decision problems $\mathcal{P}$ that have polynomial-time algorithms that decide these languages.

The Complexity Class P

The complexity class ***P*** consists of all decision problems $\mathcal{P}$ that have polynomial-time algorithms that decide these languages.

Members of P:

- Shortest paths
- Minimum spanning tree
- Minimum-length prefix codes
- ...

The Complexity Class P

The complexity class ***P*** consists of all decision problems $\mathcal{P}$ that have polynomial-time algorithms that decide these languages.

Members of P:

- Shortest paths
- Minimum spanning tree
- Minimum-length prefix codes
- ...

Unlikely to be in P:

- Vertex cover
- Satisfiability
- Hamiltonian cycle
- ...

Note: More formally, the classes P and NP are defined as sets of formal languages. Formal languages and decision problems are essentially the same thing. We won't go into this here.

Decision Problems As Sets

It is often convenient to consider a decision problem to be a set $\mathcal{P}$ containing all yes-instances.

Example: For the problem of deciding whether a number is prime, the set $\mathcal{P}$ is the set of all prime numbers $\{2, 3, 5, 7, 11, 13, \ldots\}$.

Decision Problems As Sets

It is often convenient to consider a decision problem to be a set $\mathcal{P}$ containing all yes-instances.

Example: For the problem of deciding whether a number is prime, the set $\mathcal{P}$ is the set of all prime numbers $\{2, 3, 5, 7, 11, 13, \ldots\}$.

The problem $\mathcal{P}$ then becomes that of testing membership in the set $\mathcal{P}$.

Decision Problems As Sets

It is often convenient to consider a decision problem to be a set $\mathcal{P}$ containing all yes-instances.

Example: For the problem of deciding whether a number is prime, the set $\mathcal{P}$ is the set of all prime numbers $\{2, 3, 5, 7, 11, 13, \ldots\}$.

The problem $\mathcal{P}$ then becomes that of testing membership in the set $\mathcal{P}$.

Note: If you make this more rigorous, you obtain formal languages and their equivalence to decision problems.

Verification Algorithms

An algorithm $\mathcal{A}$ ***verifies*** a decision problem $\mathcal{P}$ if

$$\mathcal{P} = \{x : \text{there exists a } y \text{ such that } \mathcal{A}(x, y) = \text{“yes”}\}.$$

Think about $\mathcal{A}$ as a proof checker. I claim that $x \in \mathcal{P}$, and I provide a “proof” y. Algorithm $\mathcal{A}$ verifies whether y proves that $x \in \mathcal{P}$. If $\mathcal{A}(x, y) =$ “no”, this does not mean that $x \notin \mathcal{P}$. It only means that y does not prove that $x \in \mathcal{P}$.

Verification Algorithms

An algorithm $\mathcal{A}$ ***verifies*** a decision problem $\mathcal{P}$ if

$$\mathcal{P} = \{x : \text{there exists a } y \text{ such that } \mathcal{A}(x, y) = \text{“yes”}\}.$$

Think about $\mathcal{A}$ as a proof checker. I claim that $x \in \mathcal{P}$, and I provide a “proof” y. Algorithm $\mathcal{A}$ verifies whether y proves that $x \in \mathcal{P}$. If $\mathcal{A}(x, y) =$ “no”, this does not mean that $x \notin \mathcal{P}$. It only means that y does not prove that $x \in \mathcal{P}$.

Examples:

Given a graph G and a simple cycle C, it is easy to verify whether C contains all vertices of G and contains only edges of G. An algorithm that does this verifies the class of Hamiltonian graphs.

Verification Algorithms

An algorithm $\mathcal{A}$ ***verifies*** a decision problem $\mathcal{P}$ if

$$\mathcal{P} = \{x : \text{there exists a } y \text{ such that } \mathcal{A}(x, y) = \text{“yes”}\}.$$

Think about $\mathcal{A}$ as a proof checker. I claim that $x \in \mathcal{P}$, and I provide a “proof” y. Algorithm $\mathcal{A}$ verifies whether y proves that $x \in \mathcal{P}$. If $\mathcal{A}(x, y) =$ “no”, this does not mean that $x \notin \mathcal{P}$. It only means that y does not prove that $x \in \mathcal{P}$.

Examples:

Given a graph G and a simple cycle C, it is easy to verify whether C contains all vertices of G and contains only edges of G. An algorithm that does this verifies the class of Hamiltonian graphs.

Given a graph G and a set S of vertices of G, it is easy to verify whether any two vertices in S are adjacent in G and whether $|S| \geq k$. An algorithm that does this verifies the class of graphs with independent sets of size at least k.

The Class NP

The complexity class ***NP*** consists of all decision problems $\mathcal{P}$ that satisfy the following condition:

There exists a polynomial-time algorithm $\mathcal{A}$ and a constant c such that

$$\mathcal{P} = \{x : \text{there exists a } y \text{ with } |y| \leq |x|^c \text{ and such that } \mathcal{A}(x, y) = \text{“yes”}\}.$$

The Class NP

The complexity class ***NP*** consists of all decision problems $\mathcal{P}$ that satisfy the following condition:

There exists a polynomial-time algorithm $\mathcal{A}$ and a constant c such that

$$\mathcal{P} = \{x : \text{there exists a } y \text{ with } |y| \leq |x|^c \text{ and such that } \mathcal{A}(x, y) = \text{“yes”}\}.$$

$$\textbf{\textit{NP}} = \{\mathcal{P} : \mathcal{P} \textbf{ \textit{can be verified in polynomial time}}\}$$

The Class NP

The complexity class ***NP*** consists of all decision problems $\mathcal{P}$ that satisfy the following condition:

There exists a polynomial-time algorithm $\mathcal{A}$ and a constant c such that

$$\mathcal{P} = \{x : \text{there exists a } y \text{ with } |y| \leq |x|^c \text{ and such that } \mathcal{A}(x, y) = \text{“yes”}\}.$$

$$\textbf{\textit{NP}} = \{\mathcal{P} : \mathcal{P} \textbf{ \textit{can be verified in polynomial time}}\}$$

Lemma: *$P \subseteq NP$.*

Note: Measuring the size of an input is tricky without the notion of formal languages. Just think about it as the number of bits in a reasonable binary encoding of the input.

Polynomial-Time Reductions

An algorithm $\mathcal{A}$ that ***reduces*** a decision problem $\mathcal{P}_1$ to another decision problem $\mathcal{P}_2$ computes a function $\mathcal{A}(x)$, for every x, such that

$$x \in \mathcal{P}_1 \Leftrightarrow \mathcal{A}(x) \in \mathcal{P}_2.$$

Polynomial-Time Reductions

An algorithm $\mathcal{A}$ that ***reduces*** a decision problem $\mathcal{P}_1$ to another decision problem $\mathcal{P}_2$ computes a function $\mathcal{A}(x)$, for every x, such that

$$x \in \mathcal{P}_1 \Leftrightarrow \mathcal{A}(x) \in \mathcal{P}_2.$$

Lemma: *If $\mathcal{A}$ is a polynomial-time algorithm and $\mathcal{P}_2 \in$ P, then $\mathcal{P}_1 \in$ P.*

Polynomial-Time Reductions

An algorithm $\mathcal{A}$ that ***reduces*** a decision problem $\mathcal{P}_1$ to another decision problem $\mathcal{P}_2$ computes a function $\mathcal{A}(x)$, for every x, such that

$$x \in \mathcal{P}_1 \Leftrightarrow \mathcal{A}(x) \in \mathcal{P}_2.$$

Lemma: *If $\mathcal{A}$ is a polynomial-time algorithm and $\mathcal{P}_2 \in$ P, then $\mathcal{P}_1 \in$ P.*

Corollary: *If $\mathcal{A}$ is a polynomial-time algorithm and there is no polynomial-time algorithm for $\mathcal{P}_1$, there cannot be a polynomial-time algorithm for $\mathcal{P}_2$.*

NP-Hardness and NP-Completeness

A decision problem $\mathcal{P}$ is ***NP-hard*** if

$$\mathcal{P} \in \mathsf{P} \Rightarrow \mathcal{P}' \in \mathsf{P} \quad \forall\, \mathcal{P}' \in \mathsf{NP}.$$

NP-Hardness and NP-Completeness

A decision problem $\mathcal{P}$ is ***NP-hard*** if

$$\mathcal{P} \in \mathsf{P} \Rightarrow \mathcal{P}' \in \mathsf{P} \quad \forall\, \mathcal{P}' \in \mathsf{NP}.$$

A decision problem $\mathcal{P}$ is ***NP-complete*** if

- $\mathcal{P} \in \mathsf{NP}$
- $\mathcal{P}$ is NP-hard

NP-Hardness and NP-Completeness

A decision problem $\mathcal{P}$ is ***NP-hard*** if

$$\mathcal{P} \in \mathsf{P} \Rightarrow \mathcal{P}' \in \mathsf{P} \quad \forall\, \mathcal{P}' \in \mathsf{NP}.$$

A decision problem $\mathcal{P}$ is ***NP-complete*** if

- $\mathcal{P} \in \mathsf{NP}$
- $\mathcal{P}$ is NP-hard

Lemma: *If we can reduce an NP-hard decision problem $\mathcal{P}$ to another decision problem $\mathcal{P}'$ using a polynomial-time algorithm, then $\mathcal{P}'$ is also NP-hard.*

NP-Hardness and NP-Completeness

A decision problem $\mathcal{P}$ is ***NP-hard*** if

$$\mathcal{P} \in \mathsf{P} \Rightarrow \mathcal{P}' \in \mathsf{P} \quad \forall\, \mathcal{P}' \in \mathsf{NP}.$$

A decision problem $\mathcal{P}$ is ***NP-complete*** if

- $\mathcal{P} \in \mathsf{NP}$
- $\mathcal{P}$ is NP-hard

Lemma: *If we can reduce an NP-hard decision problem $\mathcal{P}$ to another decision problem $\mathcal{P}'$ using a polynomial-time algorithm, then $\mathcal{P}'$ is also NP-hard.*

Corollary: *If we can reduce an NP-hard decision problem $\mathcal{P}$ to another decision problem $\mathcal{P}' \in$ NP using a polynomial-time algorithm, then $\mathcal{P}'$ is NP-complete.*

Satisfiability

The Satisfiability Problem (SAT):

Given a logical formula formed using $\wedge$ (and), $\vee$ (or), and $\neg$ (not); decide whether it is satisfiable, that is, whether there exists a truth assignment to its variables that makes the formula true.

Satisfiability

The Satisfiability Problem (SAT):

Given a logical formula formed using $\wedge$ (and), $\vee$ (or), and $\neg$ (not); decide whether it is satisfiable, that is, whether there exists a truth assignment to its variables that makes the formula true.

A formula is said to be in ***3-CNF*** if

$$F = C_1 \wedge C_2 \wedge \cdots \wedge C_k,$$

where

$$C_i = \lambda_{i,1} \vee \lambda_{i,2} \vee \lambda_{i,3}$$

and

$$\lambda_{i,j} = x_k \text{ or } \lambda_{i,j} = \bar{x}_k.$$

Satisfiability

The Satisfiability Problem (SAT):

> Given a logical formula formed using $\wedge$ (and), $\vee$ (or), and $\neg$ (not); decide whether it is satisfiable, that is, whether there exists a truth assignment to its variables that makes the formula true.

A formula is said to be in ***3-CNF*** if

$$F = C_1 \wedge C_2 \wedge \cdots \wedge C_k,$$

where

$$C_i = \lambda_{i,1} \vee \lambda_{i,2} \vee \lambda_{i,3}$$

and

$$\lambda_{i,j} = x_k \text{ or } \lambda_{i,j} = \bar{x}_k.$$

3-SAT: Given a logical formula in 3-CNF, decide whether it is satisfiable.

Satisfiability is NP-Complete

Theorem: *SAT and 3-SAT are NP-complete problems.*

Satisfiability is NP-Complete

Theorem: *SAT and 3-SAT are NP-complete problems.*

Proof sketch:

- Membership in NP is easy: given a formula F and a truth assignment to its variables, it takes linear time to verify whether F is satisfied by this assignment.

Satisfiability is NP-Complete

Theorem: *SAT and 3-SAT are NP-complete problems.*

Proof sketch:

- Membership in NP is easy: given a formula F and a truth assignment to its variables, it takes linear time to verify whether F is satisfied by this assignment.
- To prove hardness, one shows that the computation of every non-deterministic Turing machine $\mathcal{M}$ with polynomial running time on an input x can be expressed as a logical formula $F_{\mathcal{M}}$ of polynomial length.

Satisfiability is NP-Complete

Theorem: *SAT and 3-SAT are NP-complete problems.*

Proof sketch:

- Membership in NP is easy: given a formula F and a truth assignment to its variables, it takes linear time to verify whether F is satisfied by this assignment.
- To prove hardness, one shows that the computation of every non-deterministic Turing machine $\mathcal{M}$ with polynomial running time on an input x can be expressed as a logical formula $F_{\mathcal{M}}$ of polynomial length.
- More precisely, $F_{\mathcal{M}}$ is satisfiable if and only if $\mathcal{M}$ accepts input x.

Satisfiability is NP-Complete

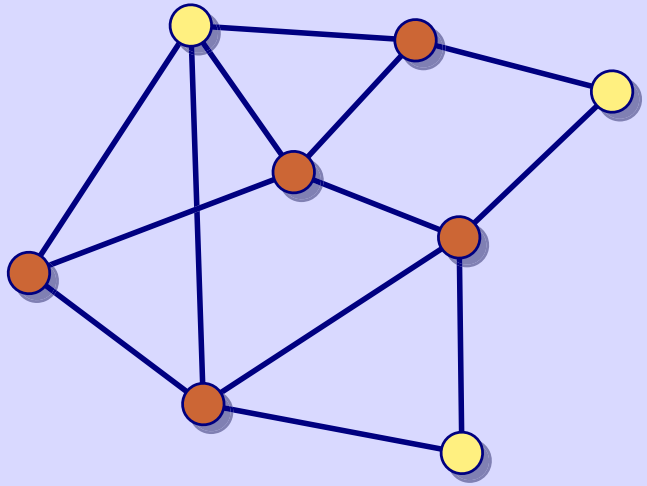
Theorem: *SAT and 3-SAT are NP-complete problems.*

Proof sketch:

- Membership in NP is easy: given a formula F and a truth assignment to its variables, it takes linear time to verify whether F is satisfied by this assignment.
- To prove hardness, one shows that the computation of every non-deterministic Turing machine $\mathcal{M}$ with polynomial running time on an input x can be expressed as a logical formula $F_{\mathcal{M}}$ of polynomial length.
- More precisely, $F_{\mathcal{M}}$ is satisfiable if and only if $\mathcal{M}$ accepts input x.
- The relation between non-deterministic polynomial-time computations and verification, and the construction of $F_{\mathcal{M}}$ are beyond the scope of this course.

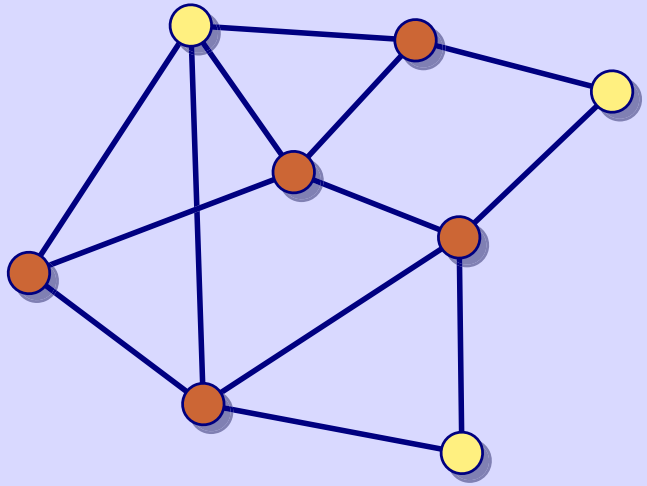
Vertex Cover

A ***vertex cover*** of a graph $G = (V, E)$ is a subset S of V such that every edge in E has at least one endpoint in S.



Vertex Cover

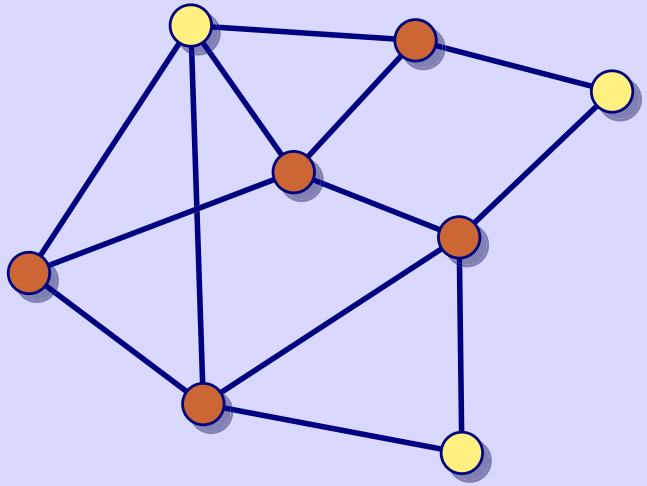
A ***vertex cover*** of a graph $G = (V, E)$ is a subset S of V such that every edge in E has at least one endpoint in S.



Optimization problem (minimum vertex cover): Given a graph G, determine the size of the smallest vertex cover of G.

Vertex Cover

A ***vertex cover*** of a graph $G = (V, E)$ is a subset S of V such that every edge in E has at least one endpoint in S.



Optimization problem (minimum vertex cover): Given a graph G, determine the size of the smallest vertex cover of G.

Decision problem: Given a graph G and an integer k, determine whether G has a vertex cover of size k.

Vertex Cover is NP-Complete (1)

Reduction from 3-SAT:

- Build a graph G_F for a given formula F in 3-CNF such that G_F has a vertex cover of a certain size if and only if F is satisfiable.

Vertex Cover is NP-Complete (1)

Reduction from 3-SAT:

- Build a graph G_F for a given formula F in 3-CNF such that G_F has a vertex cover of a certain size if and only if F is satisfiable.
- Building blocks for G_F are ***widgets*** (subgraphs that enforce certain properties).

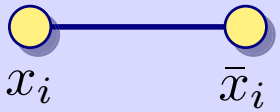
Vertex Cover is NP-Complete (1)

Reduction from 3-SAT:

- Build a graph G_F for a given formula F in 3-CNF such that G_F has a vertex cover of a certain size if and only if F is satisfiable.
- Building blocks for G_F are ***widgets*** (subgraphs that enforce certain properties).

Variable widgets:

- Two vertices x_i and $\bar{x}_i$
- One edge $(x_i, \bar{x}_i)$



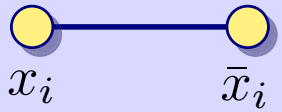
Vertex Cover is NP-Complete (1)

Reduction from 3-SAT:

- Build a graph G_F for a given formula F in 3-CNF such that G_F has a vertex cover of a certain size if and only if F is satisfiable.
- Building blocks for G_F are ***widgets*** (subgraphs that enforce certain properties).

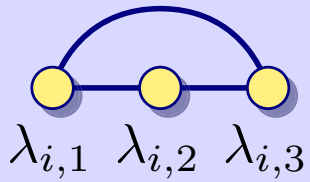
Variable widgets:

- Two vertices x_i and $\bar{x}_i$
- One edge $(x_i, \bar{x}_i)$



Clause widgets:

- Three literal vertices $\lambda_{i,1}$, $\lambda_{i,2}$, and $\lambda_{i,3}$
- Three edges $(\lambda_{i,1}, \lambda_{i,2})$, $(\lambda_{i,2}, \lambda_{i,3})$, and $(\lambda_{i,3}, \lambda_{i,1})$



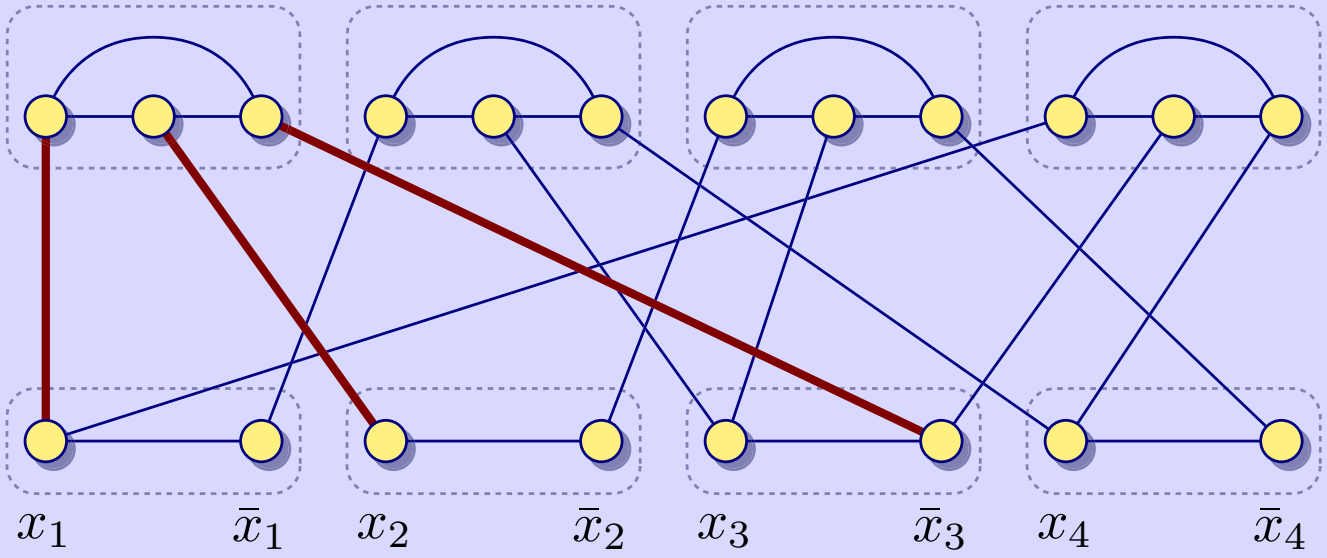
Vertex Cover is NP-Complete (2)

Given: Formula F with n variables and m clauses

$$F = (\boldsymbol{x_1} \vee \boldsymbol{x_2} \vee \boldsymbol{\bar{x}_3}) \wedge (\bar{x}_1 \vee x_3 \vee x_4) \wedge (\bar{x}_2 \vee x_3 \vee \bar{x}_4) \wedge (x_1 \vee \bar{x}_3 \vee x_4)$$

Reduction: Construct a graph G_F with n variable widgets and m clause widgets.

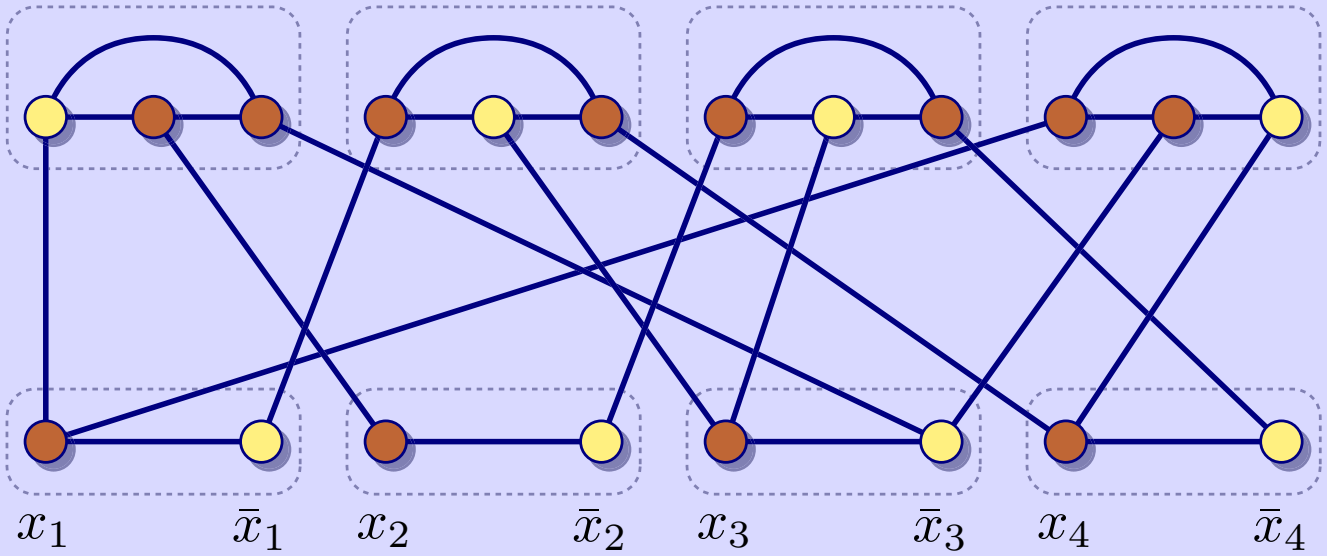
- Connect every literal node $\lambda_{i,j}$ to its corresponding node x_k or $\bar{x}_k$.



Vertex Cover is NP-Complete (3)

Lemma: *Formula F is satisfiable if and only if G_F has a vertex cover of size $n + 2m$.*

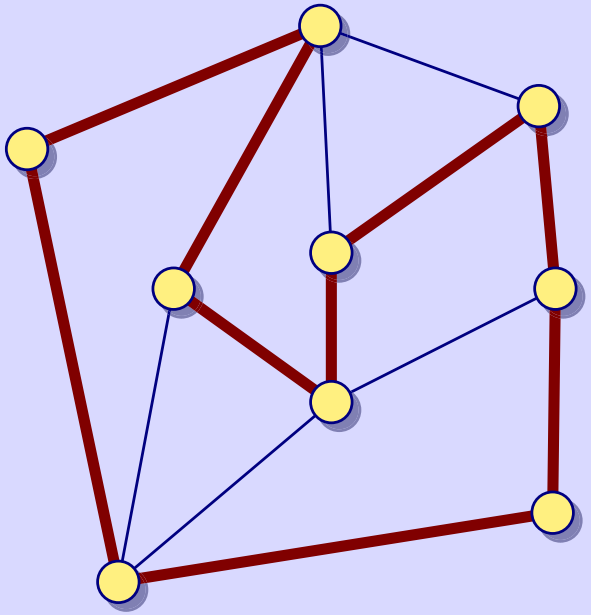
$$F = (\boldsymbol{x_1} \vee \boldsymbol{x_2} \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \boldsymbol{x_3} \vee \boldsymbol{x_4}) \wedge (\bar{x}_2 \vee \boldsymbol{x_3} \vee \bar{x}_4) \wedge (\boldsymbol{x_1} \vee \bar{x}_3 \vee \boldsymbol{x_4})$$



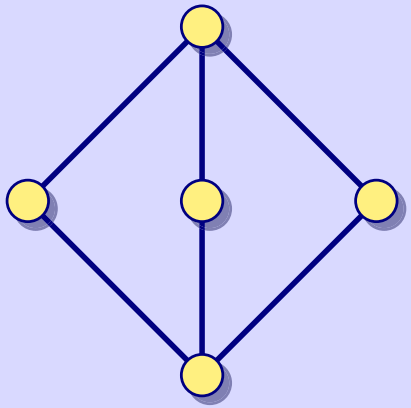
Hamiltonian Cycle

A ***Hamiltonian cycle*** of a graph G is a simple cycle that contains all vertices of G and whose edges are edges of G.

A graph G is ***Hamiltonian*** if it contains a Hamiltonian cycle.



Hamiltonian

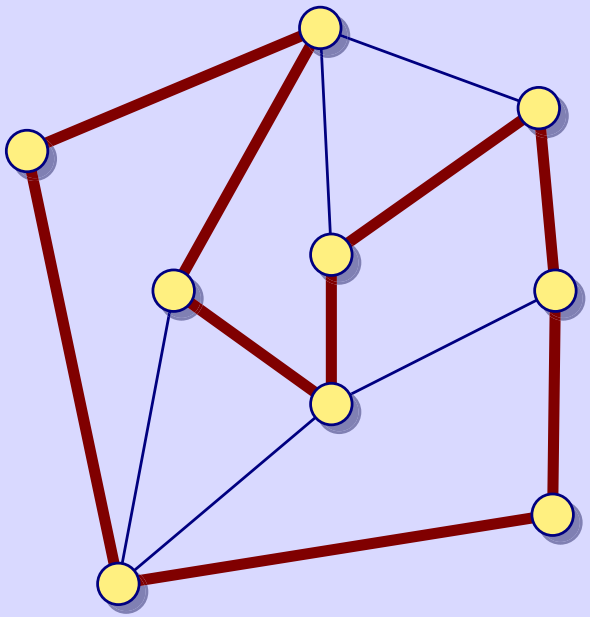


non-Hamiltonian

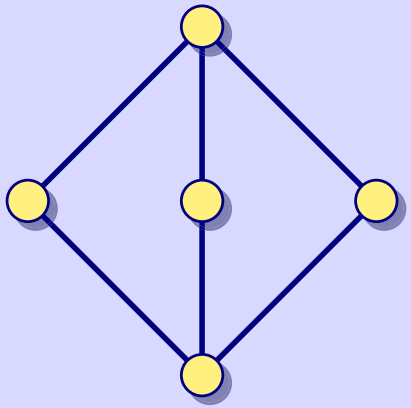
Hamiltonian Cycle

A ***Hamiltonian cycle*** of a graph G is a simple cycle that contains all vertices of G and whose edges are edges of G.

A graph G is ***Hamiltonian*** if it contains a Hamiltonian cycle.



Hamiltonian



non-Hamiltonian

Decision problem: Given a graph G, decide whether it is Hamiltonian.

Hamiltonian Cycle is NP-Complete (1)

Reduction from vertex cover:

- Build a graph G' from an instance (G, k) such that G has a vertex cover of size k if and only if G' has a Hamiltonian cycle.

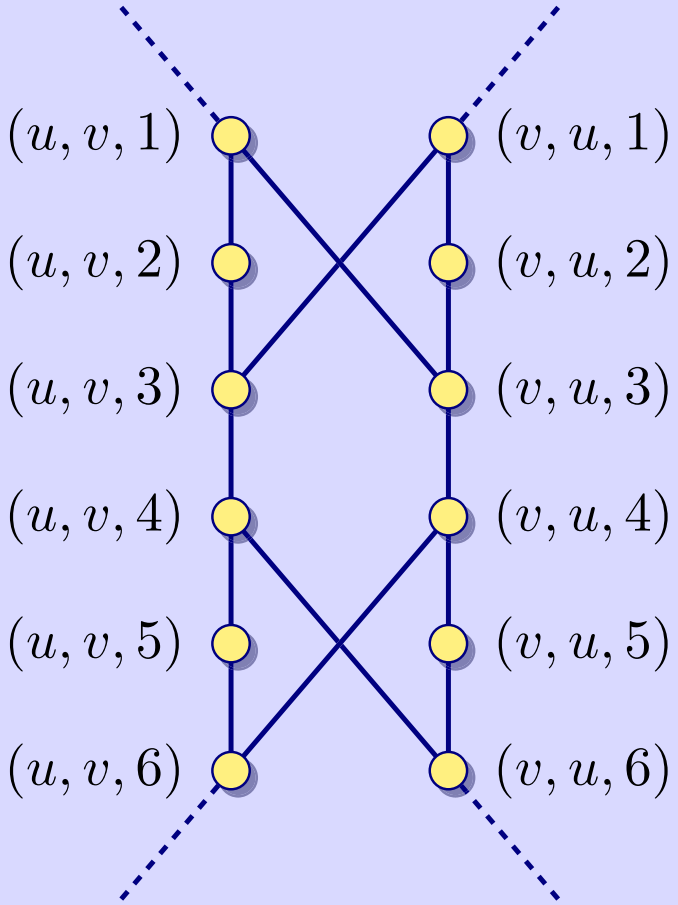
Hamiltonian Cycle is NP-Complete (1)

Reduction from vertex cover:

- Build a graph G' from an instance (G, k) such that G has a vertex cover of size k if and only if G' has a Hamiltonian cycle.

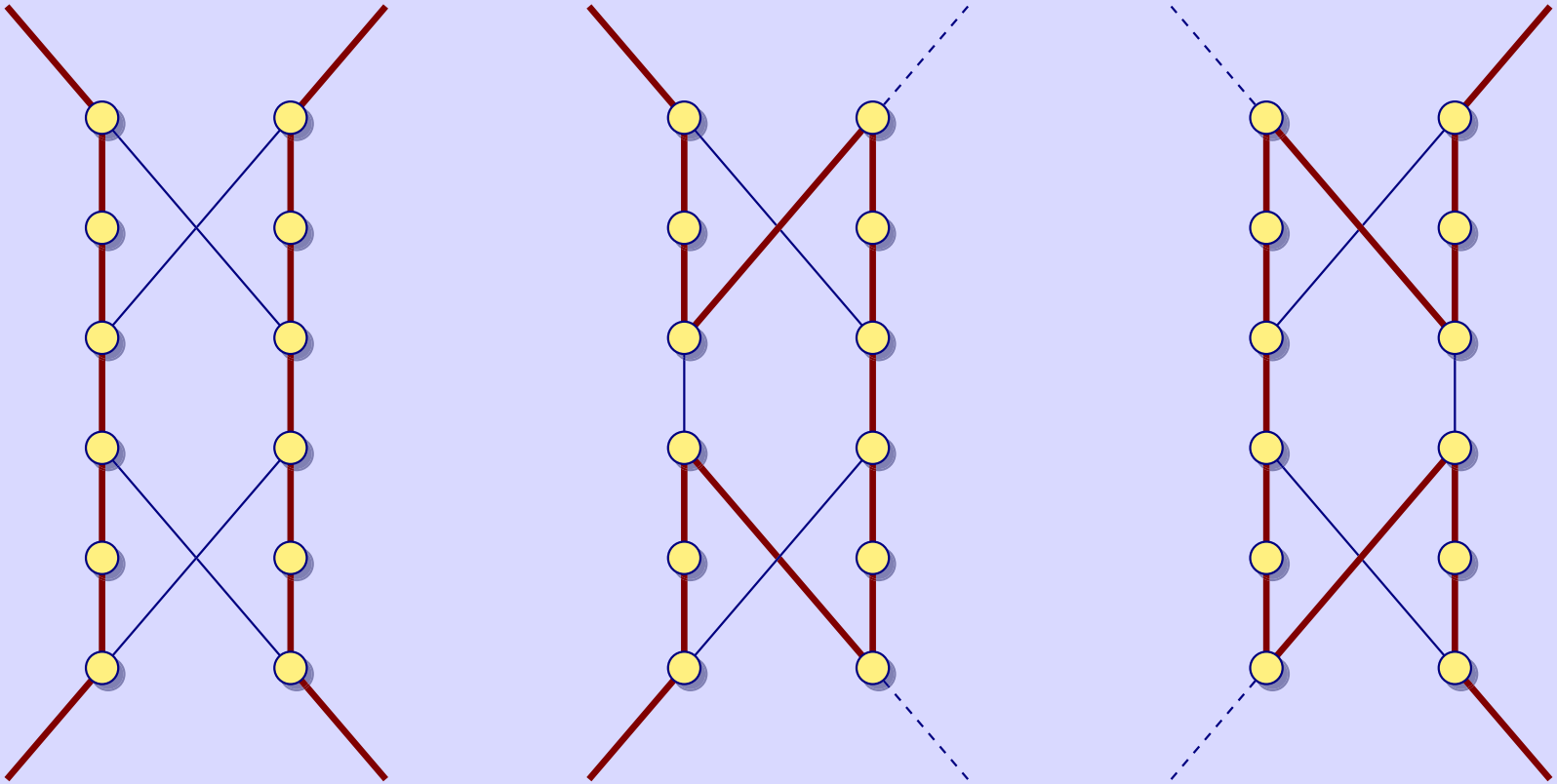
Edge widget for edge (u, v):

- 12 vertices $(u, v, 1), (u, v, 2), \ldots, (u, v, 6)$ and $(v, u, 1), (v, u, 2), \ldots, (v, u, 6)$.
- 14 edges as shown.
- Only $(u, v, 1)$, $(u, v, 6)$, $(v, u, 1)$, and $(v, u, 6)$ have "external" edges (dotted).



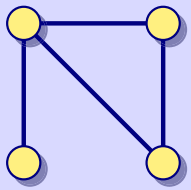
Hamiltonian Cycle is NP-Complete (2)

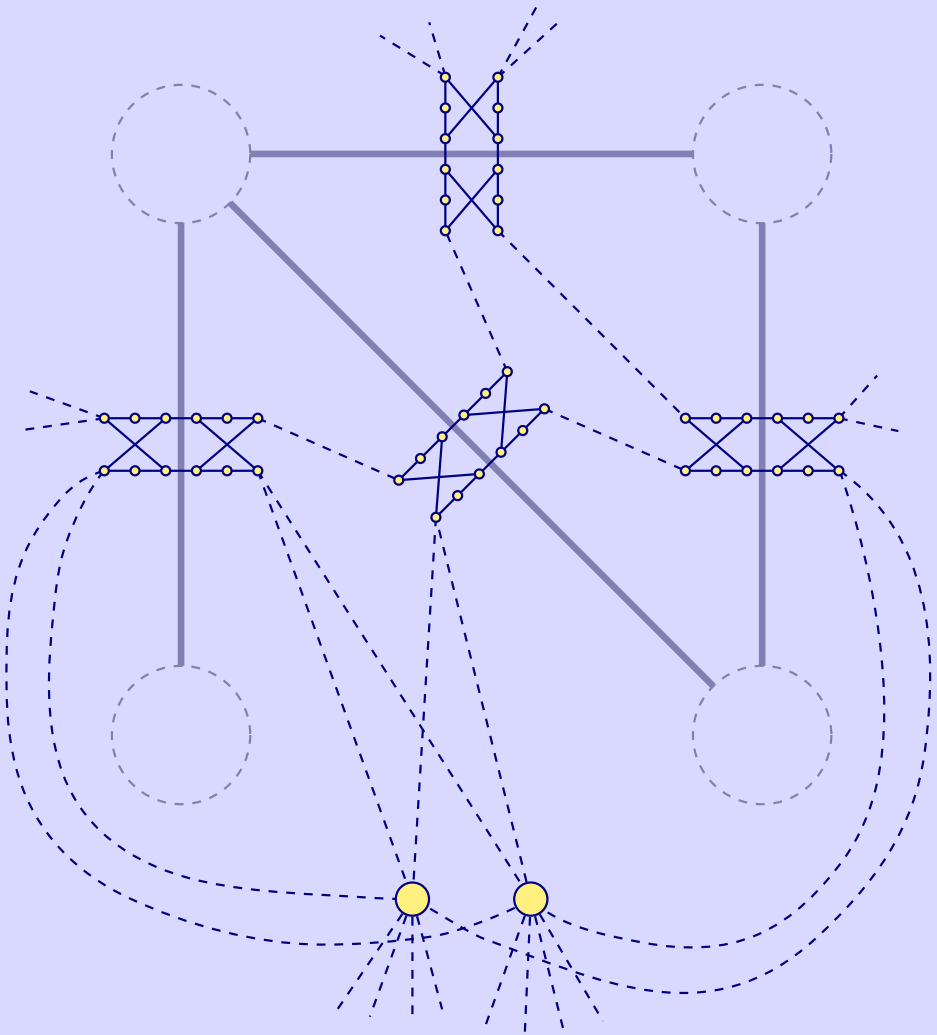
Observation: *A Hamiltonian cycle of a graph built from edge widgets traverses every widget in one of three different ways.*



Hamiltonian Cycle is NP-Complete (3)

Claim: *The graph G' contains a Hamiltonian cycle if and only if G has a vertex cover of size k.*





Hamiltonian Cycle is NP-Complete (3)

Claim: *The graph G' contains a Hamiltonian cycle if and only if G has a vertex cover of size k.*

Subset Sum

Decision problem:

Given a set S of distinct numbers, $S = \{x_1, x_2, \ldots, x_n\}$, and a parameter k, decide whether there exists a subset $S' = \{y_1, y_2, \ldots, y_m\}$ of S such that $\sum_{i=1}^{m} y_i = k$.

Subset Sum

Decision problem:

Given a set S of distinct numbers, $S = \{x_1, x_2, \ldots, x_n\}$, and a parameter k, decide whether there exists a subset $S' = \{y_1, y_2, \ldots, y_m\}$ of S such that $\sum_{i=1}^{m} y_i = k$.

Example:

$$S = \{1, 2, 8, 13\}$$

Set S contains a subset S' whose elements sum to 22, namely $\{1, 8, 13\}$; but there is no subset whose elements sum to 12.

Subset Sum is NP-Complete (1)

Reduction from 3-SAT:

- Given a formula F with n variables and m clauses, we construct a set S_F of $2n + 2m$ numbers with $n + m$ digits in base 10 and a number $t = \sum_{i=0}^{n-1} 10^{i+m} + \sum_{i=0}^{m-1} 4 \cdot 10^i$:

$$t = \underbrace{11\ldots1}_{n \text{ variable digits}} \quad \underbrace{44\ldots4}_{m \text{ clause digits}}$$

Subset Sum is NP-Complete (1)

Reduction from 3-SAT:

- Given a formula F with n variables and m clauses, we construct a set S_F of $2n + 2m$ numbers with $n + m$ digits in base 10 and a number $t = \sum_{i=0}^{n-1} 10^{i+m} + \sum_{i=0}^{m-1} 4 \cdot 10^i$:

$$t = \underbrace{11\ldots1}_{n \text{ variable digits}} \quad \underbrace{44\ldots4}_{m \text{ clause digits}}$$

- There will be a subset of S_F whose numbers sum to t if and only if F is satisfiable.

Subset Sum is NP-Complete (2)

Literal numbers:

- Two numbers v_i and $\bar{v}_i$ per variable x_i.
- Both have a 1 in the i-th variable digit.
- v_i has a 1 in every digit corresponding to a clause that contains x_i.
- $\bar{v}_i$ has a 1 in every digit corresponding to a clause that contains $\bar{x}_i$.
- All other digits in v_i and $\bar{v}_i$ are 0.

Subset Sum is NP-Complete (2)

Literal numbers:

- Two numbers v_i and $\bar{v}_i$ per variable x_i.
- Both have a 1 in the i-th variable digit.
- v_i has a 1 in every digit corresponding to a clause that contains x_i.
- $\bar{v}_i$ has a 1 in every digit corresponding to a clause that contains $\bar{x}_i$.
- All other digits in v_i and $\bar{v}_i$ are 0.

Slack numbers:

- Two slack numbers s_i and s'_i per clause.
- s_i has a 1 in the digit corresponding to the clause.
- s'_i has a 2 in the digit corresponding to the clause.
- All other digits in s_i and s'_i are 0.

Subset Sum is NP-Complete (3)

Example:

$$F = (x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_3 \vee x_4) \wedge (\bar{x}_2 \vee x_3 \vee \bar{x}_4) \wedge (x_1 \vee \bar{x}_3 \vee x_4)$$

$$v_1 = 00011001$$
$$\bar{v}_1 = 00010100$$
$$v_2 = 00101000$$
$$\bar{v}_2 = 00100010$$
$$v_3 = 01000110$$
$$\bar{v}_3 = 01001001$$
$$v_4 = 10000101$$
$$\bar{v}_4 = 10000010$$

$$s_1 = 00001000$$
$$s'_1 = 00002000$$
$$s_2 = 00000100$$
$$s'_2 = 00000200$$
$$s_3 = 00000010$$
$$s'_3 = 00000020$$
$$s_4 = 00000001$$
$$s'_4 = 00000002$$

Subset Sum is NP-Complete (3)

Example:

$$F = (\boldsymbol{x_1} \vee \boldsymbol{x_2} \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \boldsymbol{x_3} \vee \boldsymbol{x_4}) \wedge (\bar{x}_2 \vee \boldsymbol{x_3} \vee \bar{x}_4) \wedge (\boldsymbol{x_1} \vee \bar{x}_3 \vee \boldsymbol{x_4})$$

- $v_1 = 00011001$
- $\bar{v}_1 = 00010100$
- $v_2 = 00101000$
- $\bar{v}_2 = 00100010$
- $v_3 = 01000110$
- $\bar{v}_3 = 01001001$
- $v_4 = 10000101$
- $\bar{v}_4 = 10000010$

- $s_1 = 00001000$
- $s'_1 = 00002000$
- $s_2 = 00000100$
- $s'_2 = 00000200$
- $s_3 = 00000010$
- $s'_3 = 00000020$
- $s_4 = 00000001$
- $s'_4 = 00000002$

$$v_1 + v_2 + v_3 + v_4 + s'_1 + s'_2 + s_3 + s'_3 + s'_4 = t$$

Subset Sum is NP-Complete (3)

Example:

$$F = (\boldsymbol{x_1} \vee \boldsymbol{x_2} \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \boldsymbol{x_3} \vee \boldsymbol{x_4}) \wedge (\bar{x}_2 \vee \boldsymbol{x_3} \vee \bar{x}_4) \wedge (\boldsymbol{x_1} \vee \bar{x}_3 \vee \boldsymbol{x_4})$$

- $v_1 = 00011001$
- $\bar{v}_1 = 00010100$
- $v_2 = 00101000$
- $\bar{v}_2 = 00100010$
- $v_3 = 01000110$
- $\bar{v}_3 = 01001001$
- $v_4 = 10000101$
- $\bar{v}_4 = 10000010$

- $s_1 = 00001000$
- $s'_1 = 00002000$
- $s_2 = 00000100$
- $s'_2 = 00000200$
- $s_3 = 00000010$
- $s'_3 = 00000020$
- $s_4 = 00000001$
- $s'_4 = 00000002$

$$v_1 + v_2 + v_3 + v_4 + s'_1 + s'_2 + s_3 + s'_3 + s'_4 = t$$

Lemma: *The Set S_F contains a subset that sums to t if and only if F is satisfiable.*

Summary

Many important problems are NP-complete.

Examples:

- Satisfiability
- Clique
- Vertex cover
- Independent set
- Hamiltonian cycle
- Subset sum
- ...

These problems are unlikely to be solvable in polynomial time.

Alternatives:

- Approximation algorithms (CS 4113)
- Parameterized algorithms