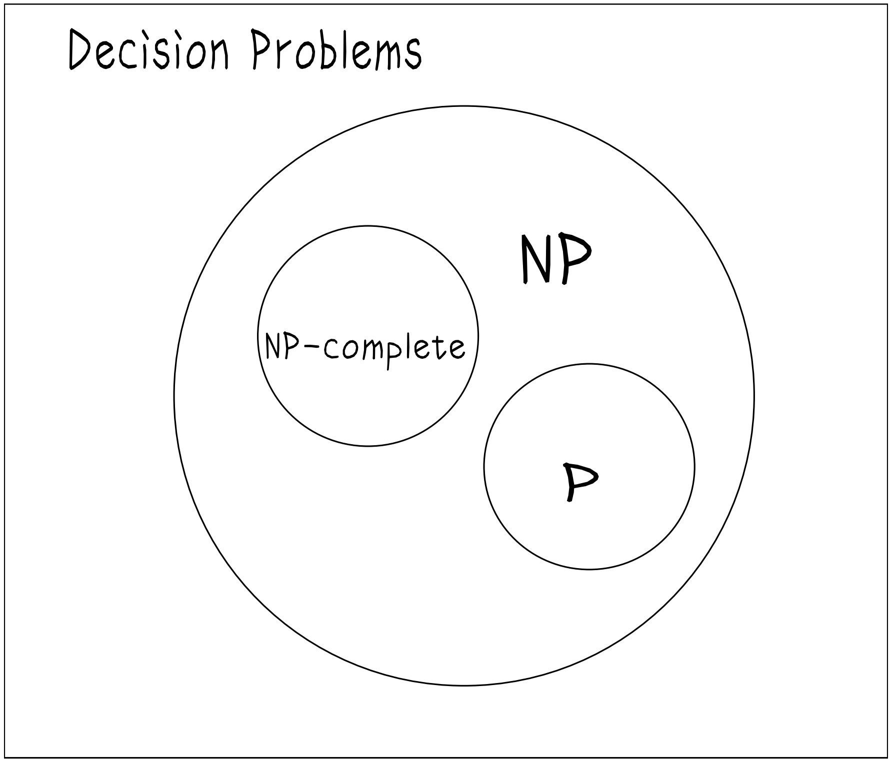
# Further Mathematics and Algorithms

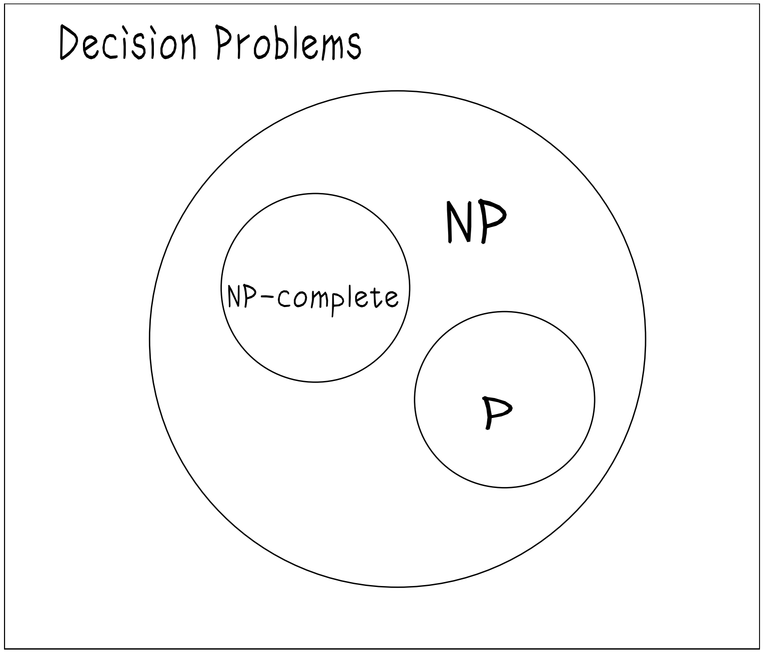
## Lesson 20: *Know What's Possible*




*Combinatorial optimisation, NP-completeness, polynomial reduction*

# Outline

1. **Motivation**
2. P, NP and NP-complete
3. Polynomial Reduction



# Exponentially Large Search Spaces

- We have seen a large number of decision problems and optimisation problems involving an exponentially large search space
- For some of these we have found efficient algorithms (greedy algorithms, divide and conquer, dynamic programming, . . . )
- For other problems we have found good algorithms (backtracking, branch and bound), but they are not necessarily polynomial
- Can we say anything general about how easy they are to solve

# Exponentially Large Search Spaces

- We have seen a large number of decision problems and optimisation problems involving an exponentially large search space
- For some of these we have found efficient algorithms (greedy algorithms, divide and conquer, dynamic programming, . . . )
- For other problems we have found good algorithms (backtracking, branch and bound), but they are not necessarily polynomial
- Can we say anything general about how easy they are to solve

# Exponentially Large Search Spaces

- We have seen a large number of decision problems and optimisation problems involving an exponentially large search space
- For some of these we have found efficient algorithms (greedy algorithms, divide and conquer, dynamic programming, . . . )
- For other problems we have found good algorithms (backtracking, branch and bound), but they are not necessarily polynomial
- Can we say anything general about how easy they are to solve

# Exponentially Large Search Spaces

- We have seen a large number of decision problems and optimisation problems involving an exponentially large search space
- For some of these we have found efficient algorithms (greedy algorithms, divide and conquer, dynamic programming, . . . )
- For other problems we have found good algorithms (backtracking, branch and bound), but they are not necessarily polynomial
- Can we say anything general about how easy they are to solve

# Types of Problems

- We concentrate here on two types of problems
  - ⋆ Decision Problems
  - ⋆ Combinatorial Optimisation Problems
- Decision problems are problems with a true/false answer, e.g. is it possible to cross all the bridges of Königsberg once?
- We showed earlier that backtracking can be used to find a solution which answers the decision problems, e.g. Hamiltonian circuit problem
- There are many other decision problems, but the most famous is satisfiability or SAT

# Types of Problems

- We concentrate here on two types of problems
  - ⋆ Decision Problems
  - ⋆ Combinatorial Optimisation Problems
- Decision problems are problems with a true/false answer, e.g. is it possible to cross all the bridges of Königsberg once?
- We showed earlier that backtracking can be used to find a solution which answers the decision problems, e.g. Hamiltonian circuit problem
- There are many other decision problems, but the most famous is satisfiability or SAT

# Types of Problems

- We concentrate here on two types of problems
  - ⋆ Decision Problems
  - ⋆ Combinatorial Optimisation Problems
- Decision problems are problems with a true/false answer, e.g. is it possible to cross all the bridges of Königsberg once?
- We showed earlier that backtracking can be used to find a solution which answers the decision problems, e.g. Hamiltonian circuit problem
- There are many other decision problems, but the most famous is satisfiability or SAT

# Types of Problems

- We concentrate here on two types of problems
  - ⋆ Decision Problems
  - ⋆ Combinatorial Optimisation Problems
- Decision problems are problems with a true/false answer, e.g. is it possible to cross all the bridges of Königsberg once?
- We showed earlier that backtracking can be used to find a solution which answers the decision problems, e.g. Hamiltonian circuit problem
- There are many other decision problems, but the most famous is satisfiability or SAT

# SAT

- Given $n$ Boolean variables $X_i \in \{T, F\}$
- $m$ disjunctive (or's) clauses, e.g.

$$
\begin{aligned}
c_1 &= X_1 \vee \neg X_2 \vee X_3 \\
c_2 &= \neg X_2 \vee X_3 \vee X_5 \\
&\vdots \\
c_m &= X_2 \vee \neg X_4 \vee \neg X_5
\end{aligned}
$$

- Find an assignment, $\boldsymbol{X} \in \{T, F\}^n$ which satisfies all the clauses
- We can view this as finding an assignment that makes the formula $f(\boldsymbol{X})$ true where

$$f(\boldsymbol{X}) = c_1 \wedge c_2 \wedge \cdots \wedge c_m$$

# SAT

- Given $n$ Boolean variables $X_i \in \{T, F\}$
- $m$ disjunctive (or's) clauses, e.g.

$$
\begin{aligned}
c_1 &= X_1 \vee \neg X_2 \vee X_3 \\
c_2 &= \neg X_2 \vee X_3 \vee X_5 \\
&\vdots \\
c_m &= X_2 \vee \neg X_4 \vee \neg X_5
\end{aligned}
$$

- Find an assignment, $\boldsymbol{X} \in \{T, F\}^n$ which satisfies all the clauses
- We can view this as finding an assignment that makes the formula $f(\boldsymbol{X})$ true where

$$
f(\boldsymbol{X}) = c_1 \wedge c_2 \wedge \cdots \wedge c_m
$$

# SAT

- Given $n$ Boolean variables $X_i \in \{T, F\}$
- $m$ disjunctive (or's) clauses, e.g.

$$\begin{aligned}
c_1 &= X_1 \vee \neg X_2 \vee X_3 \\
c_2 &= \neg X_2 \vee X_3 \vee X_5 \\
&\vdots \\
c_m &= X_2 \vee \neg X_4 \vee \neg X_5
\end{aligned}$$

- Find an assignment, $\boldsymbol{X} \in \{T, F\}^n$ which satisfies all the clauses
- We can view this as finding an assignment that makes the formula $f(\boldsymbol{X})$ true where

$$f(\boldsymbol{X}) = c_1 \wedge c_2 \wedge \cdots \wedge c_m$$

# SAT

- Given $n$ Boolean variables $X_i \in \{T, F\}$
- $m$ disjunctive (or's) clauses, e.g.

$$\begin{aligned} c_1 &= X_1 \vee \neg X_2 \vee X_3 \\ c_2 &= \neg X_2 \vee X_3 \vee X_5 \\ \vdots & \qquad \vdots \\ c_m &= X_2 \vee \neg X_4 \vee \neg X_5 \end{aligned}$$

- Find an assignment, $\boldsymbol{X} \in \{T, F\}^n$ which satisfies all the clauses
- We can view this as finding an assignment that makes the formula $f(\boldsymbol{X})$ true where

$$f(\boldsymbol{X}) = c_1 \wedge c_2 \wedge \cdots \wedge c_m$$

# Decisions and Optimisation Problems

- Often we can cast a decision problem as an optimisation problem
- E.g. the MAX-SAT problem is to find an assignment of variables that satisfies the most clauses
- If we can solve the MAX-SAT optimisation problem we can solve the decision problem
- We can also cast optimisation problems as decision problems

  *Does there exist a TSP tour shorter than 200 miles?*

# Decisions and Optimisation Problems

- Often we can cast a decision problem as an optimisation problem
- E.g. the MAX-SAT problem is to find an assignment of variables that satisfies the most clauses
- If we can solve the MAX-SAT optimisation problem we can solve the decision problem
- We can also cast optimisation problems as decision problems

  *Does there exist a TSP tour shorter than 200 miles?*

# Decisions and Optimisation Problems

- Often we can cast a decision problem as an optimisation problem
- E.g. the MAX-SAT problem is to find an assignment of variables that satisfies the most clauses
- If we can solve the MAX-SAT optimisation problem we can solve the decision problem
- We can also cast optimisation problems as decision problems

  *Does there exist a TSP tour shorter than 200 miles?*

# Decisions and Optimisation Problems

- Often we can cast a decision problem as an optimisation problem
- E.g. the MAX-SAT problem is to find an assignment of variables that satisfies the most clauses
- If we can solve the MAX-SAT optimisation problem we can solve the decision problem
- We can also cast optimisation problems as decision problems

  *Does there exist a TSP tour shorter than 200 miles?*

# Combinatorial Optimisation Problems

- In the set of discrete optimisation problems an important class are those that involve *combinatorial objects* such as permutations, binary string, etc.

- Optimisation problems involving such objects are termed **combinatorial optimisation problems**

- Classical examples of such problems include
  - Travelling Salesperson Problem (TSP)
  - Graph colouring
  - Maximum Satisfiability (MAX-SAT)
  - Many scheduling problems
  - Bin-packing
  - Quadratic integer problems

# Combinatorial Optimisation Problems

- In the set of discrete optimisation problems an important class are those that involve *combinatorial objects* such as permutations, binary string, etc.

- Optimisation problems involving such objects are termed **combinatorial optimisation problems**

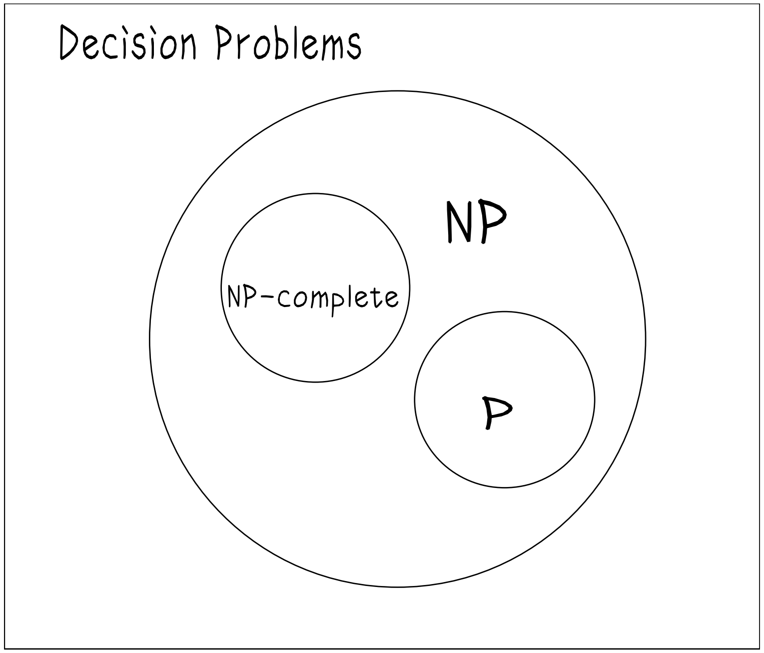
- Classical examples of such problems include
  - ⋆ Travelling Salesperson Problem (TSP)
  - ⋆ Graph colouring
  - ⋆ Maximum Satisfiability (MAX-SAT)
  - ⋆ Many scheduling problems
  - ⋆ Bin-packing
  - ⋆ Quadratic integer problems

# Combinatorial Optimisation Problems

- In the set of discrete optimisation problems an important class are those that involve *combinatorial objects* such as permutations, binary string, etc.
- Optimisation problems involving such objects are termed **combinatorial optimisation problems**
- Classical examples of such problems include
  - ⋆ Travelling Salesperson Problem (TSP)
  - ⋆ Graph colouring
  - ⋆ Maximum Satisfiability (MAX-SAT)
  - ⋆ Many scheduling problems
  - ⋆ Bin-packing
  - ⋆ Quadratic integer problems

# Outline

1. Motivation
2. **P, NP and NP-complete**
3. Polynomial Reduction



# Polynomial Problems

- Some optimisation problems are "easy"—there are known polynomial time algorithms to solve them
  - ⋆ Minimum spanning tree
  - ⋆ Shortest path
  - ⋆ Linear assignment problem
  - ⋆ Maximum flow between any two vertices of a directed graph
  - ⋆ Linear programming
- Many apparently different problems can be mapped onto these problems

# Polynomial Problems

- Some optimisation problems are “easy”—there are known polynomial time algorithms to solve them
  - ⋆ Minimum spanning tree
  - ⋆ Shortest path
  - ⋆ Linear assignment problem
  - ⋆ Maximum flow between any two vertices of a directed graph
  - ⋆ Linear programming
- Many apparently different problems can be mapped onto these problems

# Polynomial Problems

- Some optimisation problems are "easy"—there are known polynomial time algorithms to solve them
  - ⋆ Minimum spanning tree
  - ⋆ Shortest path
  - ⋆ Linear assignment problem
  - ⋆ Maximum flow between any two vertices of a directed graph
  - ⋆ Linear programming
- Many apparently different problems can be mapped onto these problems

# NP-Hard Problems

- Is it possible to solve the TSP in polynomial time?
- Answer:
- However, no one has discovered such an algorithm and if they do it will have huge implications
- TSP is an example of a class of problems called NP-Hard
- If you can solve one of these problems then you can solve a whole class of problems
- To understand what NP-Hard means we must backtrack

# NP-Hard Problems

- Is it possible to solve the TSP in polynomial time?
- Answer:
- However, no one has discovered such an algorithm and if they do it will have huge implications
- TSP is an example of a class of problems called NP-Hard
- If you can solve one of these problems then you can solve a whole class of problems
- To understand what NP-Hard means we must backtrack

# NP-Hard Problems

- Is it possible to solve the TSP in polynomial time?
- Answer: **maybe**
- However, no one has discovered such an algorithm and if they do it will have huge implications
- TSP is an example of a class of problems called NP-Hard
- If you can solve one of these problems then you can solve a whole class of problems
- To understand what NP-Hard means we must backtrack

# NP-Hard Problems

- Is it possible to solve the TSP in polynomial time?
- Answer: **maybe**
- However, no one has discovered such an algorithm and if they do it will have huge implications
- TSP is an example of a class of problems called NP-Hard
- If you can solve one of these problems then you can solve a whole class of problems
- To understand what NP-Hard means we must backtrack

# NP-Hard Problems

- Is it possible to solve the TSP in polynomial time?
- Answer: **maybe**
- However, no one has discovered such an algorithm and if they do it will have huge implications
- TSP is an example of a class of problems called NP-Hard
- If you can solve one of these problems then you can solve a whole class of problems
- To understand what NP-Hard means we must backtrack

# NP-Hard Problems

- Is it possible to solve the TSP in polynomial time?
- Answer: **maybe**
- However, no one has discovered such an algorithm and if they do it will have huge implications
- TSP is an example of a class of problems called NP-Hard
- If you can solve one of these problems then you can solve a whole class of problems
- To understand what NP-Hard means we must backtrack

# NP-Hard Problems

- Is it possible to solve the TSP in polynomial time?
- Answer: **maybe**
- However, no one has discovered such an algorithm and if they do it will have huge implications
- TSP is an example of a class of problems called NP-Hard
- If you can solve one of these problems then you can solve a whole class of problems
- To understand what NP-Hard means we must backtrack

# NP-Hard Problems

- Is it possible to solve the TSP in polynomial time?
- Answer: **maybe**
- However, no one has discovered such an algorithm and if they do it will have huge implications
- TSP is an example of a class of problems called NP-Hard
- If you can solve one of these problems then you can solve a whole class of problems
- To understand what NP-Hard means we must backtrack

# Decision Problems

- Decision problems are problems with a true or false answer
- E.g. does a TSP have a tour less than $2\,000$ miles?
- Algorithmic complexity theory deals with classes of decision problems that have some characteristic size, $n$
- E.g. $n$ is the number of cities in a TSP
- Any decision problem that can be answered with an algorithm that runs in polynomial time ($n^a$) on a normal computer (Turing machine) is said to be in class P
- E.g. The decision problem “Is there a path of length less that 450 miles between Southampton and Glasgow?” is in class P

# Decision Problems

- Decision problems are problems with a true or false answer
- E.g. does a TSP have a tour less than $2\,000$ miles?
- Algorithmic complexity theory deals with classes of decision problems that have some characteristic size, $n$
- E.g. $n$ is the number of cities in a TSP
- Any decision problem that can be answered with an algorithm that runs in polynomial time ($n^a$) on a normal computer (Turing machine) is said to be in class P
- E.g. The decision problem "Is there a path of length less that 450 miles between Southampton and Glasgow?" is in class P

# Decision Problems

- Decision problems are problems with a true or false answer
- E.g. does a TSP have a tour less than $2\,000$ miles?
- Algorithmic complexity theory deals with classes of decision problems that have some characteristic size, $n$
- E.g. $n$ is the number of cities in a TSP
- Any decision problem that can be answered with an algorithm that runs in polynomial time ($n^a$) on a normal computer (Turing machine) is said to be in class P
- E.g. The decision problem "Is there a path of length less that 450 miles between Southampton and Glasgow?" is in class P

# Decision Problems

- Decision problems are problems with a true or false answer
- E.g. does a TSP have a tour less than $2\,000$ miles?
- Algorithmic complexity theory deals with classes of decision problems that have some characteristic size, $n$
- E.g. $n$ is the number of cities in a TSP
- Any decision problem that can be answered with an algorithm that runs in polynomial time ($n^a$) on a normal computer (Turing machine) is said to be in class P
- E.g. The decision problem "Is there a path of length less that 450 miles between Southampton and Glasgow?" is in class P

# Decision Problems

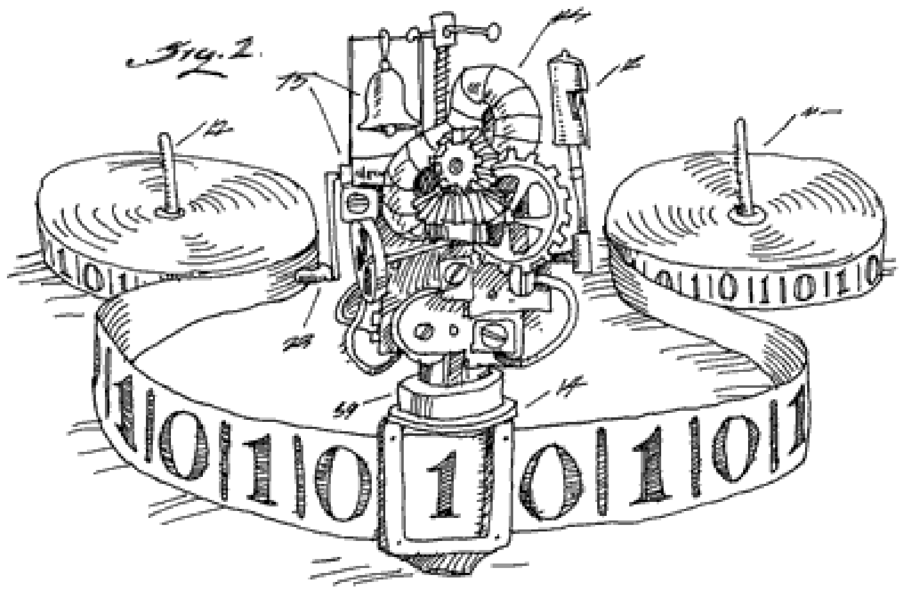
- Decision problems are problems with a true or false answer
- E.g. does a TSP have a tour less than $2\,000$ miles?
- Algorithmic complexity theory deals with classes of decision problems that have some characteristic size, $n$
- E.g. $n$ is the number of cities in a TSP
- Any decision problem that can be answered with an algorithm that runs in polynomial time ($n^a$) on a normal computer (Turing machine) is said to be in class P
- E.g. The decision problem "Is there a path of length less that 450 miles between Southampton and Glasgow?" is in class P

# Decision Problems

- Decision problems are problems with a true or false answer
- E.g. does a TSP have a tour less than $2\,000$ miles?
- Algorithmic complexity theory deals with classes of decision problems that have some characteristic size, $n$
- E.g. $n$ is the number of cities in a TSP
- Any decision problem that can be answered with an algorithm that runs in polynomial time ($n^a$) on a normal computer (Turing machine) is said to be in class P
- E.g. The decision problem "Is there a path of length less that 450 miles between Southampton and Glasgow?" is in class P

# Turing Machines

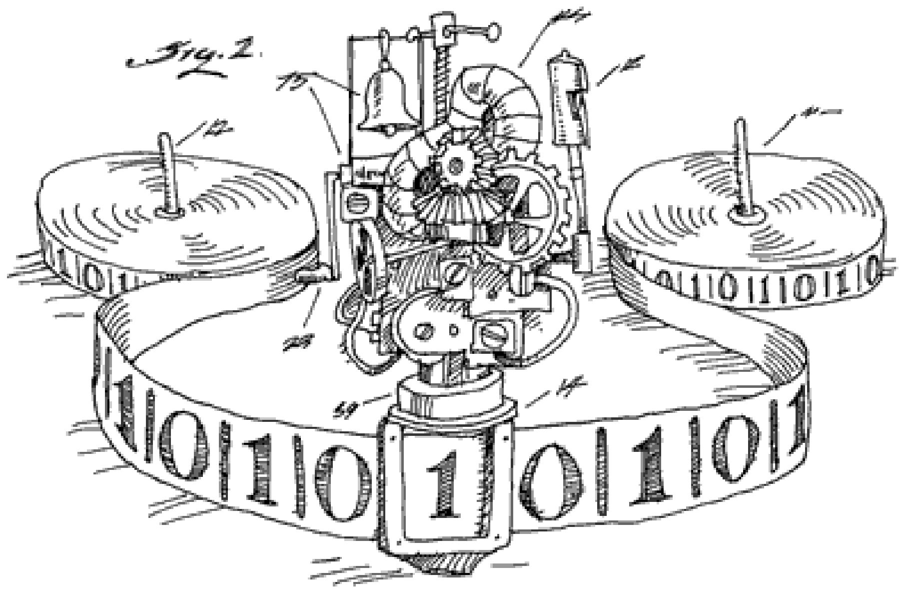
- A Turing machine can be viewed as a very dumb computer which computes by having a number of states and reading and writing to a tape



- Although dumb, it can do anything that any other computer can do (although it may take polynomially more time)

# Turing Machines

- A Turing machine can be viewed as a very dumb computer which computes by having a number of states and reading and writing to a tape



- Although dumb, it can do anything that any other computer can do (although it may take polynomially more time)

# Non-Deterministic Turing Machines

- A non-deterministic Turing machine is a magic machine that can guess the answer and then use a normal Turing machine to verify the answer
- We assume that it guesses right in the first go
- No one knows how to simulate a non-deterministic Turing machine in polynomial time
- We can simulate it in exponential time by trying all possible guesses

# Non-Deterministic Turing Machines

- A non-deterministic Turing machine is a magic machine that can guess the answer and then use a normal Turing machine to verify the answer
- We assume that it guesses right in the first go
- No one knows how to simulate a non-deterministic Turing machine in polynomial time
- We can simulate it in exponential time by trying all possible guesses

# Non-Deterministic Turing Machines

- A non-deterministic Turing machine is a magic machine that can guess the answer and then use a normal Turing machine to verify the answer
- We assume that it guesses right in the first go
- No one knows how to simulate a non-deterministic Turing machine in polynomial time
- We can simulate it in exponential time by trying all possible guesses

# Non-Deterministic Turing Machines

- A non-deterministic Turing machine is a magic machine that can guess the answer and then use a normal Turing machine to verify the answer
- We assume that it guesses right in the first go
- No one knows how to simulate a non-deterministic Turing machine in polynomial time
- We can simulate it in exponential time by trying all possible guesses

# Class NP

- Any decision problem that can be answered with an algorithm that runs in polynomial time on a non-deterministic Turing machine is said to be in class NP

- A whole lot of decision problems belong to class NP

- All decision problems with polynomial algorithms are also in class NP (P$\subset$NP)

- "*Is the game of chess winnable by white?*" is **not** in class NP

# Class NP

- Any decision problem that can be answered with an algorithm that runs in polynomial time on a non-deterministic Turing machine is said to be in class NP
- A whole lot of decision problems belong to class NP
- All decision problems with polynomial algorithms are also in class NP (P$\subset$NP)
- "*Is the game of chess winnable by white?*" is **not** in class NP

# Class NP

- Any decision problem that can be answered with an algorithm that runs in polynomial time on a non-deterministic Turing machine is said to be in class NP

- A whole lot of decision problems belong to class NP

- All decision problems with polynomial algorithms are also in class NP (P$\subset$NP)

- "*Is the game of chess winnable by white?*" is **not** in class NP

# Class NP

- Any decision problem that can be answered with an algorithm that runs in polynomial time on a non-deterministic Turing machine is said to be in class NP

- A whole lot of decision problems belong to class NP

- All decision problems with polynomial algorithms are also in class NP (P$\subset$NP)

- "*Is the game of chess winnable by white?*" is **not** in class NP

# Belonging to NP

- For a problem to belong to class NP it must
  - ⋆ be a decision problem (true of false)
  - ⋆ describable by some polynomial sized string in the length of the input $n$
  - ⋆ be verifiable if true in polynomial time on a normal Turing machine (e.g. a computer)
- To be verifiable it is sufficient for the decision problem to have a "**witness**" which is usually a solution that can be checked in polynomial time
- E.g. in TSP the witness would be a tour which satisfies the condition

# Belonging to NP

- For a problem to belong to class NP it must
  - be a decision problem (true of false)
  - describable by some polynomial sized string in the length of the input $n$
  - be verifiable if true in polynomial time on a normal Turing machine (e.g. a computer)
- To be verifiable it is sufficient for the decision problem to have a "**witness**" which is usually a solution that can be checked in polynomial time
- E.g. in TSP the witness would be a tour which satisfies the condition

# Belonging to NP

- For a problem to belong to class NP it must
  - ⋆ be a decision problem (true of false)
  - ⋆ describable by some polynomial sized string in the length of the input $n$
  - ⋆ be verifiable if true in polynomial time on a normal Turing machine (e.g. a computer)
- To be verifiable it is sufficient for the decision problem to have a "**witness**" which is usually a solution that can be checked in polynomial time
- E.g. in TSP the witness would be a tour which satisfies the condition

# Belonging to NP

- For a problem to belong to class NP it must
  - ⋆ be a decision problem (true of false)
  - ⋆ describable by some polynomial sized string in the length of the input $n$
  - ⋆ be verifiable if true in polynomial time on a normal Turing machine (e.g. a computer)
- To be verifiable it is sufficient for the decision problem to have a "**witness**" which is usually a solution that can be checked in polynomial time
- E.g. in TSP the witness would be a tour which satisfies the condition

# Belonging to NP

- For a problem to belong to class NP it must
  - be a decision problem (true of false)
  - describable by some polynomial sized string in the length of the input $n$
  - be verifiable if true in polynomial time on a normal Turing machine (e.g. a computer)
- To be verifiable it is sufficient for the decision problem to have a "**witness**" which is usually a solution that can be checked in polynomial time
- E.g. in TSP the witness would be a tour which satisfies the condition

# Using Non-Deterministic Turing Machines

- A witness is sufficient for a problem to be in NP since a non-deterministic Turing machine can guess the witness in one step and then proceed to check the witness is true in polynomial time

- Thus TSP is also in NP

- As are graph-colouring, SAT, Max-Clique, Hamilton cycle and countless others

# Using Non-Deterministic Turing Machines

- A witness is sufficient for a problem to be in NP since a non-deterministic Turing machine can guess the witness in one step and then proceed to check the witness is true in polynomial time
- Thus TSP is also in NP
- As are graph-colouring, SAT, Max-Clique, Hamilton cycle and countless others

# Using Non-Deterministic Turing Machines

- A witness is sufficient for a problem to be in NP since a non-deterministic Turing machine can guess the witness in one step and then proceed to check the witness is true in polynomial time
- Thus TSP is also in NP
- As are graph-colouring, SAT, Max-Clique, Hamilton cycle and countless others

# Class NP-complete

- In 1971 Cook showed that you could represent any NP problem on a non-deterministic Turing machine by a polynomially sized SAT (Satisfaction) problem
- Thus if you could solve SAT in polynomial time you can use that to simulate a non-deterministic Turing machine in polynomial time
- SAT is therefore an example of one of the hardest problems in NP since if you can solve SAT in polynomial time you can solve all problems in NP in polynomial time
- These hardest problems in NP are said to be in NP-complete
- If there existed a polynomial time algorithm for SAT then all problems in NP could be performed in polynomial time so that NP=P

# Class NP-complete

- In 1971 Cook showed that you could represent any NP problem on a non-deterministic Turing machine by a polynomially sized SAT (Satisfaction) problem
- Thus if you could solve SAT in polynomial time you can use that to simulate a non-deterministic Turing machine in polynomial time
- SAT is therefore an example of one of the hardest problems in NP since if you can solve SAT in polynomial time you can solve all problems in NP in polynomial time
- These hardest problems in NP are said to be in NP-complete
- If there existed a polynomial time algorithm for SAT then all problems in NP could be performed in polynomial time so that NP=P

# Class NP-complete

- In 1971 Cook showed that you could represent any NP problem on a non-deterministic Turing machine by a polynomially sized SAT (Satisfaction) problem
- Thus if you could solve SAT in polynomial time you can use that to simulate a non-deterministic Turing machine in polynomial time
- SAT is therefore an example of one of the hardest problems in NP since if you can solve SAT in polynomial time you can solve all problems in NP in polynomial time
- These hardest problems in NP are said to be in NP-complete
- If there existed a polynomial time algorithm for SAT then all problems in NP could be performed in polynomial time so that NP=P

# Class NP-complete

- In 1971 Cook showed that you could represent any NP problem on a non-deterministic Turing machine by a polynomially sized SAT (Satisfaction) problem
- Thus if you could solve SAT in polynomial time you can use that to simulate a non-deterministic Turing machine in polynomial time
- SAT is therefore an example of one of the hardest problems in NP since if you can solve SAT in polynomial time you can solve all problems in NP in polynomial time
- These hardest problems in NP are said to be in NP-complete
- If there existed a polynomial time algorithm for SAT then all problems in NP could be performed in polynomial time so that NP=P

# Class NP-complete

- In 1971 Cook showed that you could represent any NP problem on a non-deterministic Turing machine by a polynomially sized SAT (Satisfaction) problem
- Thus if you could solve SAT in polynomial time you can use that to simulate a non-deterministic Turing machine in polynomial time
- SAT is therefore an example of one of the hardest problems in NP since if you can solve SAT in polynomial time you can solve all problems in NP in polynomial time
- These hardest problems in NP are said to be in NP-complete
- If there existed a polynomial time algorithm for SAT then all problems in NP could be performed in polynomial time so that NP=P

# Idea Behind Cook's Theorem

- Cook showed that a non-deterministic Turing machine could be encoded as a big SAT formula
- The evolution of the state and tape was represented by a big tableau ($n^k \times n^k$-table where $n^k$ is the time it takes for the Turing machine to verify the answer)
- The structure of the clauses reflect the rules the Turing machine operates
- If the clauses are simultaneously satisfiable then there exists an input that satisfies the conditions

# Idea Behind Cook's Theorem

- Cook showed that a non-deterministic Turing machine could be encoded as a big SAT formula
- The evolution of the state and tape was represented by a big tableau ($n^k \times n^k$-table where $n^k$ is the time it takes for the Turing machine to verify the answer)
- The structure of the clauses reflect the rules the Turing machine operates
- If the clauses are simultaneously satisfiable then there exists an input that satisfies the conditions

# Idea Behind Cook's Theorem

- Cook showed that a non-deterministic Turing machine could be encoded as a big SAT formula

- The evolution of the state and tape was represented by a big tableau ($n^k \times n^k$-table where $n^k$ is the time it takes for the Turing machine to verify the answer)

- The structure of the clauses reflect the rules the Turing machine operates

- If the clauses are simultaneously satisfiable then there exists an input that satisfies the conditions

# Idea Behind Cook's Theorem

- Cook showed that a non-deterministic Turing machine could be encoded as a big SAT formula

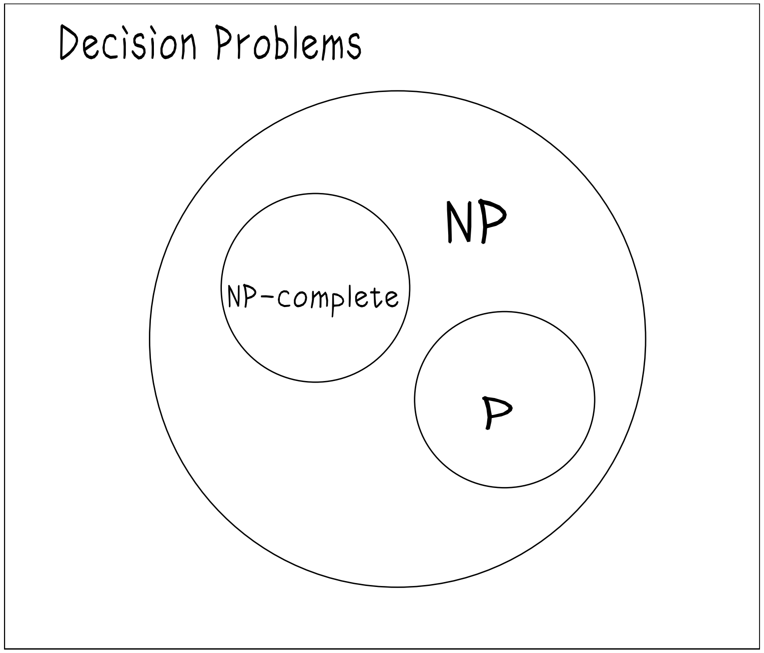
- The evolution of the state and tape was represented by a big tableau ($n^k \times n^k$-table where $n^k$ is the time it takes for the Turing machine to verify the answer)

- The structure of the clauses reflect the rules the Turing machine operates

- If the clauses are simultaneously satisfiable then there exists an input that satisfies the conditions

# Outline

1. Motivation
2. P, NP and NP-complete
3. **Polynomial Reduction**



# Polynomial Reductions

- Given two decision problems A and B we say there is a **polynomial reduction** from A to B if
  - ⋆ Every instance of A can be mapped to an instance of B:
  - ⋆ The truth of the instance A is the same as the corresponding instance B
- We can therefore use B to solve A
- So: $B \in P \rightarrow A \in P$
- The contrapositive of this statement is

  $$A \notin P \rightarrow B \notin P$$

# Polynomial Reductions

- Given two decision problems A and B we say there is a **polynomial reduction** from A to B if
  - ⋆ Every instance of A can be mapped to an instance of B:
  - ⋆ The truth of the instance A is the same as the corresponding instance B
- We can therefore use B to solve A
- So: $B \in P \rightarrow A \in P$
- The contrapositive of this statement is

  $$A \notin P \rightarrow B \notin P$$

# Polynomial Reductions

- Given two decision problems A and B we say there is a **polynomial reduction** from A to B if
  - ⋆ Every instance of A can be mapped to an instance of B:
  - ⋆ The truth of the instance A is the same as the corresponding instance B
- We can therefore use B to solve A
- So: $B \in P \rightarrow A \in P$
- The contrapositive of this statement is

  $$A \notin P \rightarrow B \notin P$$

# Polynomial Reductions

- Given two decision problems A and B we say there is a **polynomial reduction** from A to B if
  - ⋆ Every instance of A can be mapped to an instance of B:
  - ⋆ The truth of the instance A is the same as the corresponding instance B
- We can therefore use B to solve A
- So: $B \in P \rightarrow A \in P$
- The contrapositive of this statement is

  $$A \notin P \rightarrow B \notin P$$

# Polynomial Reductions

- Given two decision problems A and B we say there is a **polynomial reduction** from A to B if
  - ⋆ Every instance of A can be mapped to an instance of B:
  - ⋆ The truth of the instance A is the same as the corresponding instance B
- We can therefore use B to solve A
- So: $B \in P \to A \in P$
- The contrapositive of this statement is

  $A \notin P \to B \notin P$

# Polynomial Reductions

- Given two decision problems A and B we say there is a **polynomial reduction** from A to B if
  - ⋆ Every instance of A can be mapped to an instance of B:
  - ⋆ The truth of the instance A is the same as the corresponding instance B
- We can therefore use B to solve A
- So: $B \in P \rightarrow A \in P$
- The contrapositive of this statement is

  $$A \notin P \rightarrow B \notin P$$

# SAT to 3-SAT

- We can reduce a clause with 4 variable to a clause with 3

$$X_1 \vee \neg X_3 \vee X_6 \vee \neg X_{10} \equiv (X_1 \vee \neg X_3 \vee Z) \wedge (\neg Z \vee X_6 \vee \neg X_{10})$$

- In doing so we increase the number of variables and the number of clauses to satisfy
- We can similarly reduce a clause with more variables

$$X_1 \vee \neg X_3 \vee X_6 \vee \neg X_{10} \vee \neg X_{11} \vee X_{15} \equiv$$
$$(X_1 \vee \neg X_3 \vee Z_1) \wedge (\neg Z_1 \vee X_6 \vee Z_2) \wedge (\neg Z_2 \vee \neg X_{10} \vee Z_3) \wedge (\neg Z_3 \vee \neg X_{11} \vee X_{15})$$

- Because every instance of SAT can be written as a 3-SAT problem which is only polynomially larger than the SAT problem, 3-SAT is also NP-complete

# SAT to 3-SAT

- We can reduce a clause with 4 variable to a clause with 3

$$X_1 \vee \neg X_3 \vee X_6 \vee \neg X_{10} \equiv (X_1 \vee \neg X_3 \vee Z) \wedge (\neg Z \vee X_6 \vee \neg X_{10})$$

- In doing so we increase the number of variables and the number of clauses to satisfy

- We can similarly reduce a clause with more variables

$$X_1 \vee \neg X_3 \vee X_6 \vee \neg X_{10} \vee \neg X_{11} \vee X_{15} \equiv$$
$$(X_1 \vee \neg X_3 \vee Z_1) \wedge (\neg Z_1 \vee X_6 \vee Z_2) \wedge (\neg Z_2 \vee \neg X_{10} \vee Z_3) \wedge (\neg Z_3 \vee \neg X_{11} \vee X_{15})$$

- Because every instance of SAT can be written as a 3-SAT problem which is only polynomially larger than the SAT problem, 3-SAT is also NP-complete

# SAT to 3-SAT

- We can reduce a clause with 4 variable to a clause with 3

$$X_1 \vee \neg X_3 \vee X_6 \vee \neg X_{10} \equiv (X_1 \vee \neg X_3 \vee Z) \wedge (\neg Z \vee X_6 \vee \neg X_{10})$$

- In doing so we increase the number of variables and the number of clauses to satisfy
- We can similarly reduce a clause with more variables

$$X_1 \vee \neg X_3 \vee X_6 \vee \neg X_{10} \vee \neg X_{11} \vee X_{15} \equiv$$
$$(X_1 \vee \neg X_3 \vee Z_1) \wedge (\neg Z_1 \vee X_6 \vee Z_2) \wedge (\neg Z_2 \vee \neg X_{10} \vee Z_3) \wedge (\neg Z_3 \vee \neg X_{11} \vee X_{15})$$

- Because every instance of SAT can be written as a 3-SAT problem which is only polynomially larger than the SAT problem, 3-SAT is also NP-complete

# SAT to 3-SAT

- We can reduce a clause with 4 variable to a clause with 3

$$X_1 \vee \neg X_3 \vee X_6 \vee \neg X_{10} \equiv (X_1 \vee \neg X_3 \vee Z) \wedge (\neg Z \vee X_6 \vee \neg X_{10})$$

- In doing so we increase the number of variables and the number of clauses to satisfy

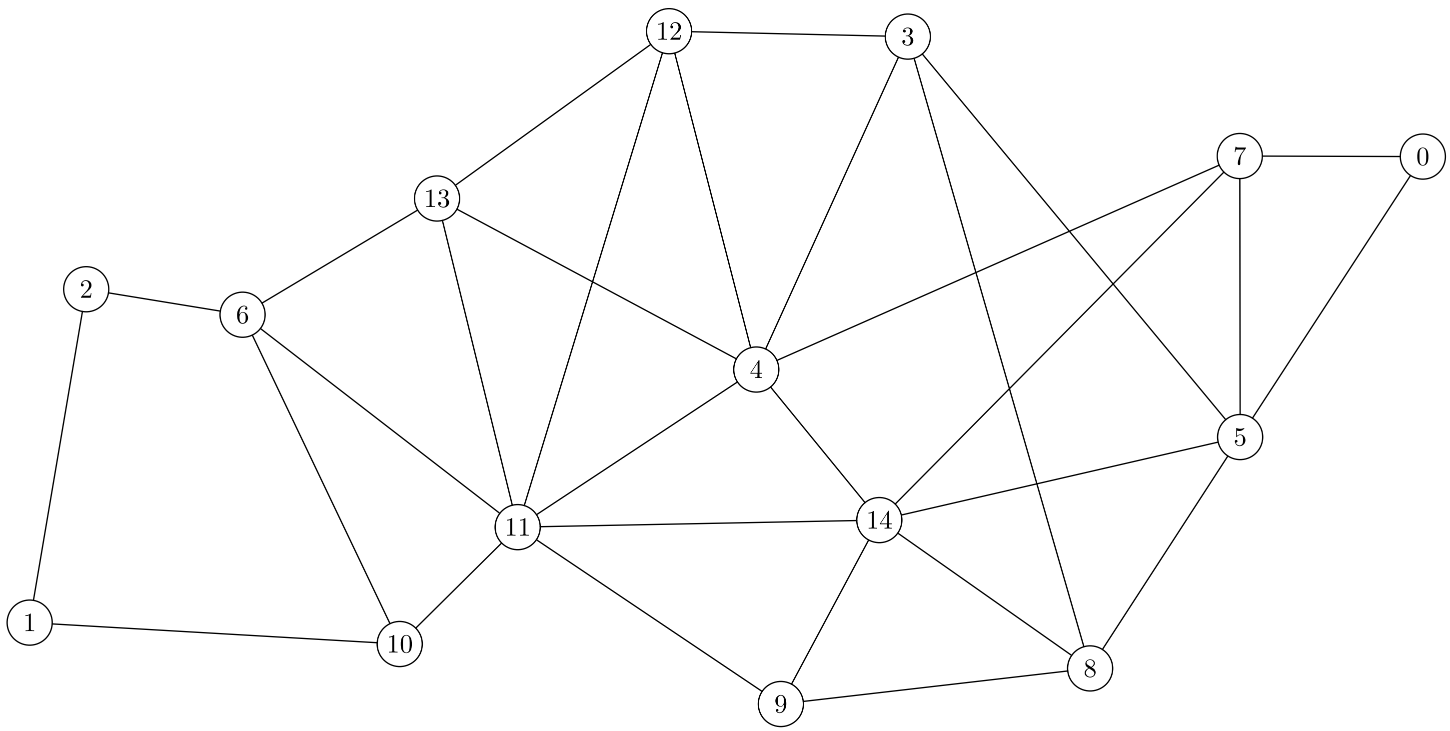
- We can similarly reduce a clause with more variables

$$X_1 \vee \neg X_3 \vee X_6 \vee \neg X_{10} \vee \neg X_{11} \vee X_{15} \equiv$$
$$(X_1 \vee \neg X_3 \vee Z_1) \wedge (\neg Z_1 \vee X_6 \vee Z_2) \wedge (\neg Z_2 \vee \neg X_{10} \vee Z_3) \wedge (\neg Z_3 \vee \neg X_{11} \vee X_{15})$$

- Because every instance of SAT can be written as a 3-SAT problem which is only polynomially larger than the SAT problem, 3-SAT is also NP-complete

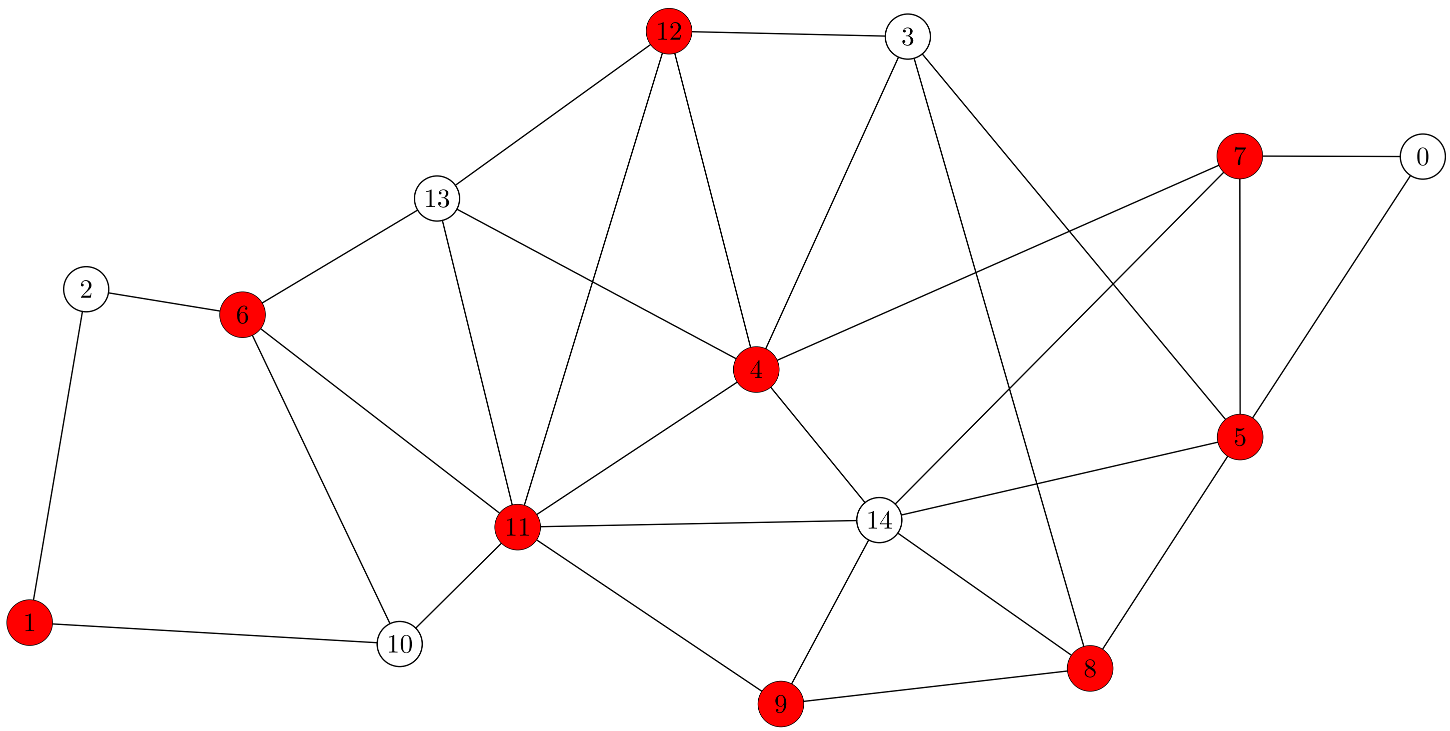
# Vertex Cover

- The vertex cover problem is: "can we choose $K$ (9) vertices of a graph such that every edge is connected to a chosen vertex?"



# Vertex Cover

- The vertex cover problem is: "can we choose $K$ (9) vertices of a graph such that every edge is connected to a chosen vertex?"



# Vertex Cover is NP-complete

- Vertex cover is obviously in NP as a set of $K$ vertices acts as a witness, i.e. it can be checked that it covers all edges
- To show vertex cover is NP-complete we show that every instance of 3-SAT is reducible to vertex cover
- The idea is to show that any 3-SAT problem with variables $\{X_1, X_2, \ldots, X_n\}$ and clauses $\{c_1, c_2, \ldots, c_m\}$ can be encoded as a vertex cover problem

# Vertex Cover is NP-complete

- Vertex cover is obviously in NP as a set of $K$ vertices acts as a witness, i.e. it can be checked that it covers all edges
- To show vertex cover is NP-complete we show that every instance of 3-SAT is reducible to vertex cover
- The idea is to show that any 3-SAT problem with variables $\{X_1, X_2, \ldots, X_n\}$ and clauses $\{c_1, c_2, \ldots, c_m\}$ can be encoded as a vertex cover problem

# Vertex Cover is NP-complete

- Vertex cover is obviously in NP as a set of $K$ vertices acts as a witness, i.e. it can be checked that it covers all edges
- To show vertex cover is NP-complete we show that every instance of 3-SAT is reducible to vertex cover
- The idea is to show that any 3-SAT problem with variables $\{X_1, X_2, \ldots, X_n\}$ and clauses $\{c_1, c_2, \ldots, c_m\}$ can be encoded as a vertex cover problem

# 3-SAT to Vertex Cover

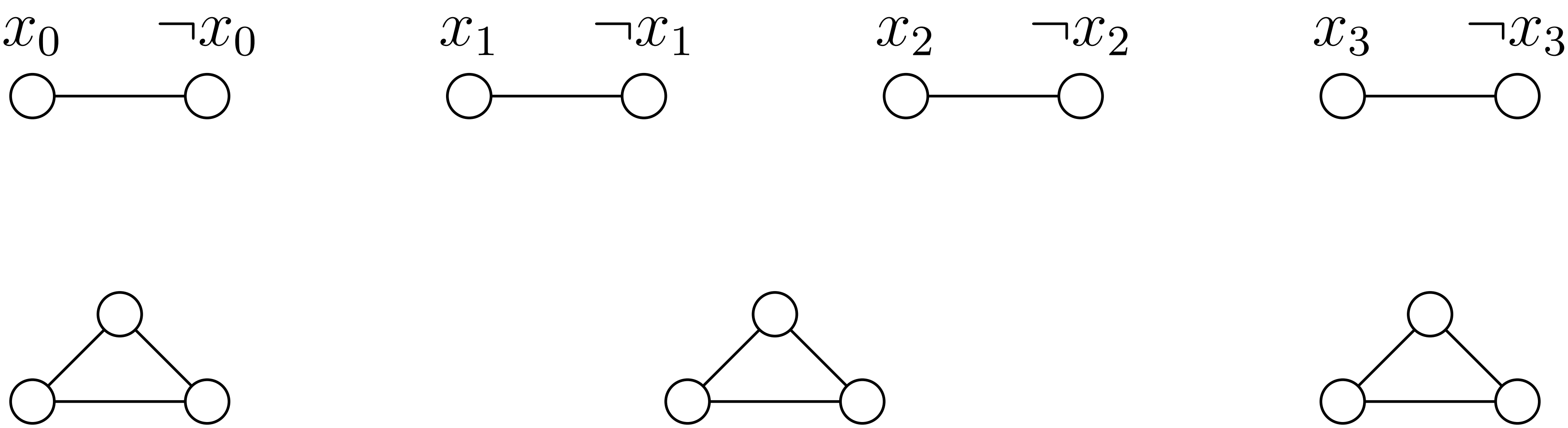
$$(x_0 \vee \neg x_1 \vee x_2) \wedge (\neg x_0 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3)$$

# 3-SAT to Vertex Cover

$$(x_0 \vee \neg x_1 \vee x_2) \wedge (\neg x_0 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3)$$

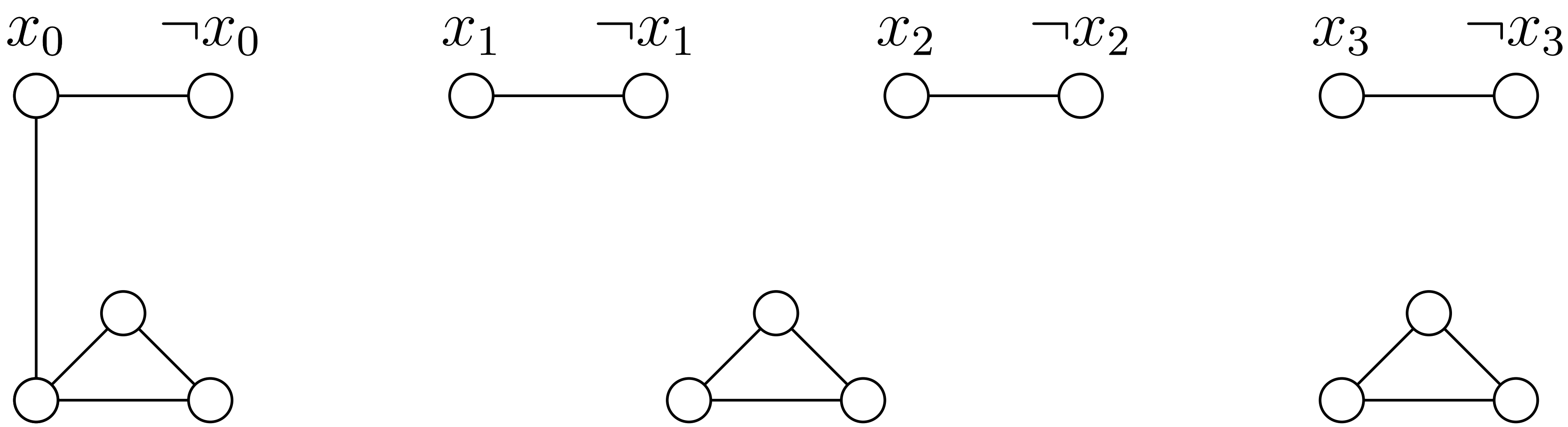
# 3-SAT to Vertex Cover

$$(x_0 \vee \neg x_1 \vee x_2) \wedge (\neg x_0 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3)$$



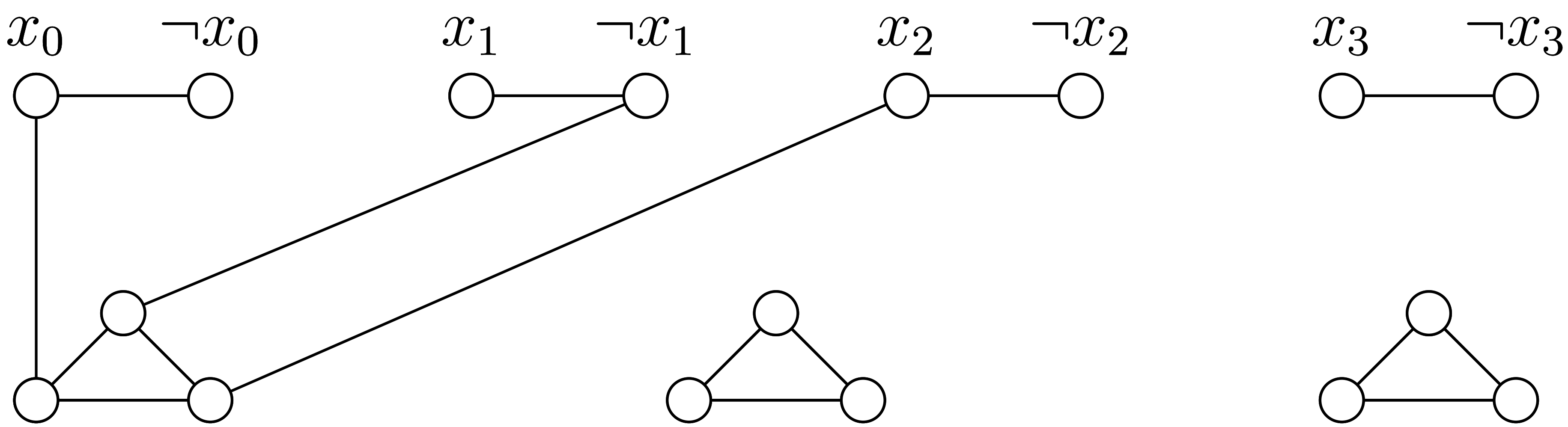
# 3-SAT to Vertex Cover

$$(x_0 \vee \neg x_1 \vee x_2) \wedge (\neg x_0 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3)$$



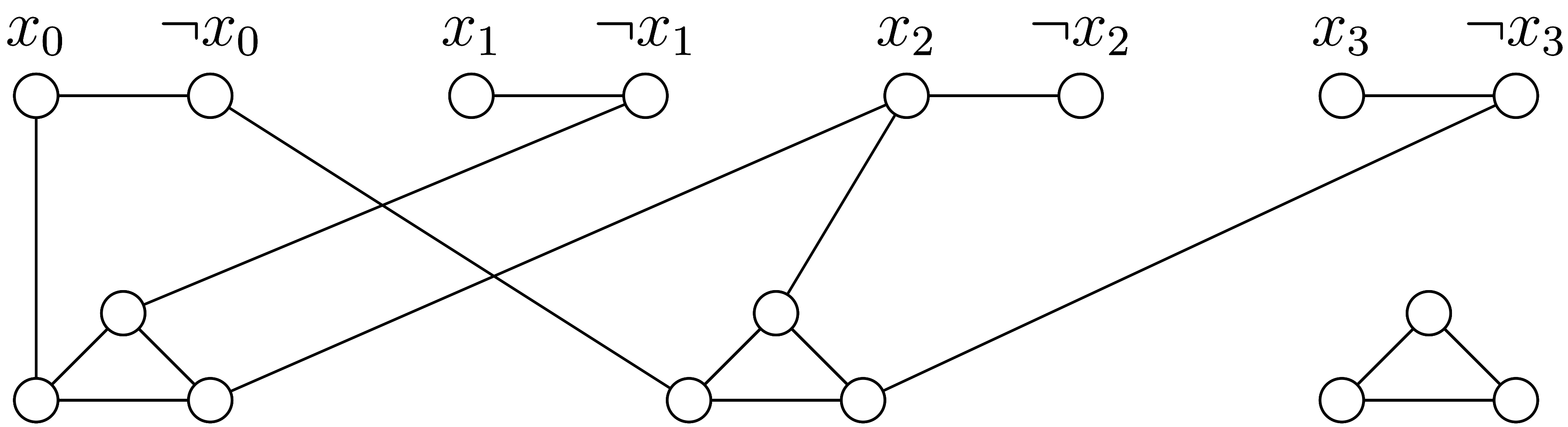
# 3-SAT to Vertex Cover

$$(x_0 \vee \neg x_1 \vee x_2) \wedge (\neg x_0 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3)$$



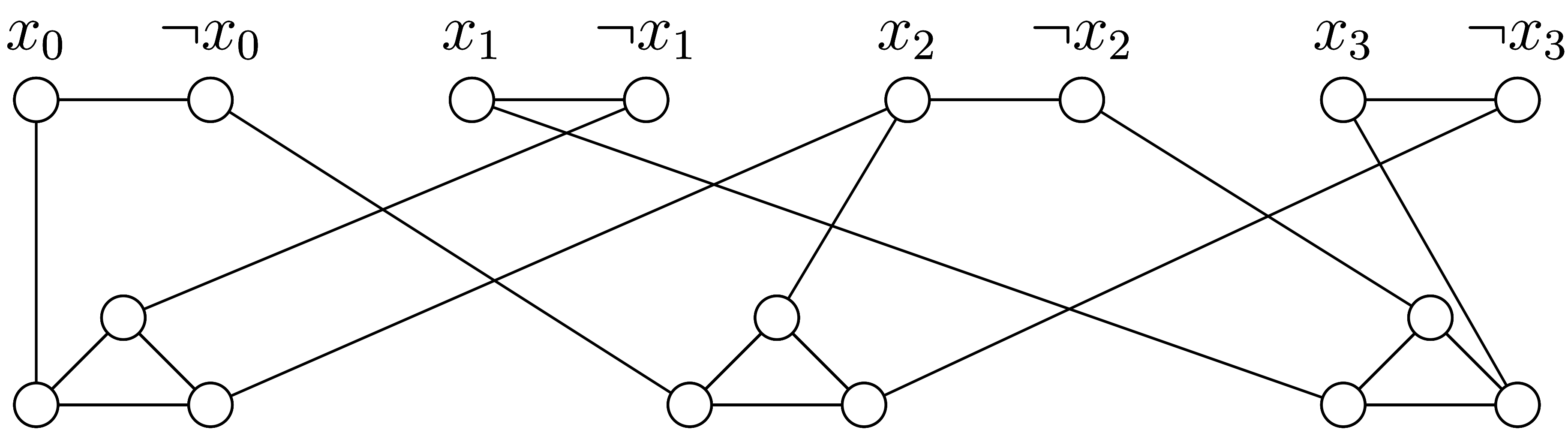
# 3-SAT to Vertex Cover

$$(x_0 \vee \neg x_1 \vee x_2) \wedge (\neg x_0 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3)$$



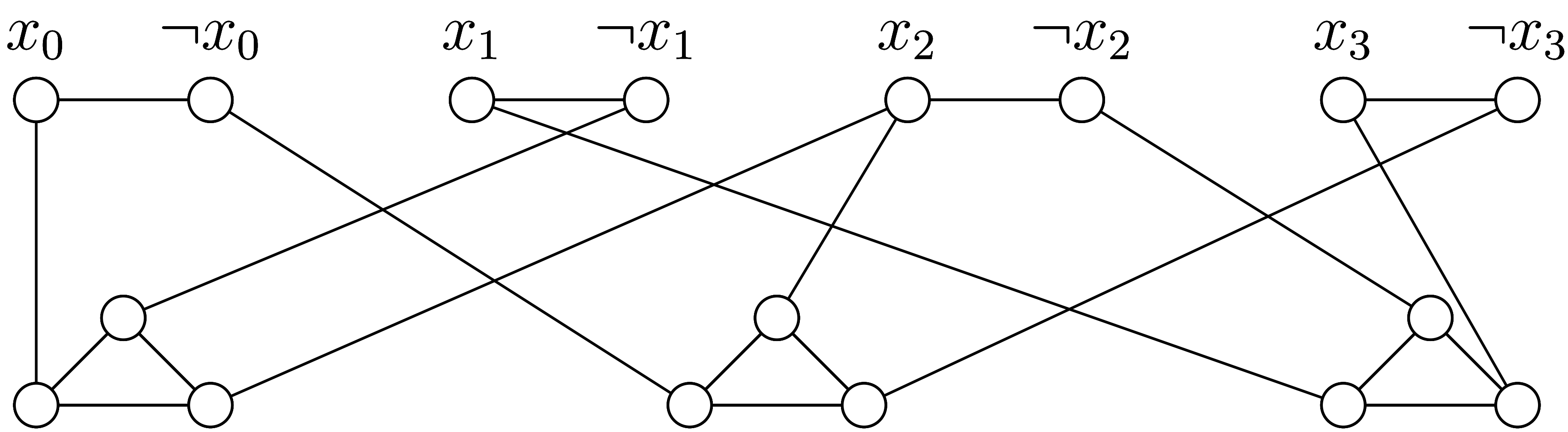
# 3-SAT to Vertex Cover

$$(x_0 \vee \neg x_1 \vee x_2) \wedge (\neg x_0 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3)$$



# 3-SAT to Vertex Cover

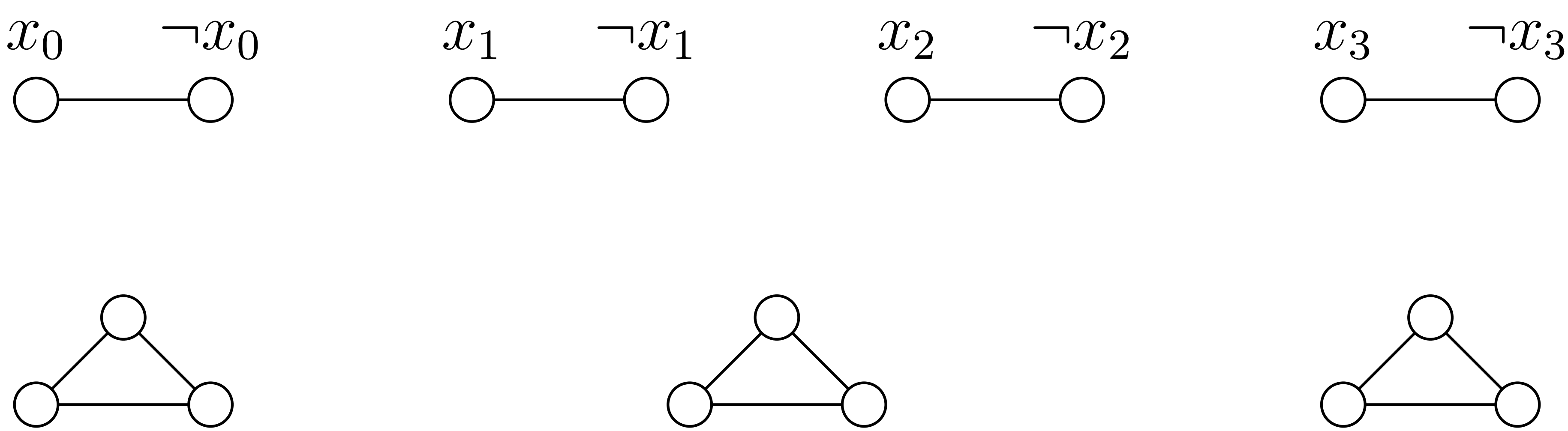
$$(x_0 \vee \neg x_1 \vee x_2) \wedge (\neg x_0 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3)$$




Cover all edges using $n + 2m = 4 + 2 \times 3 = 10$ vertices

# 3-SAT to Vertex Cover

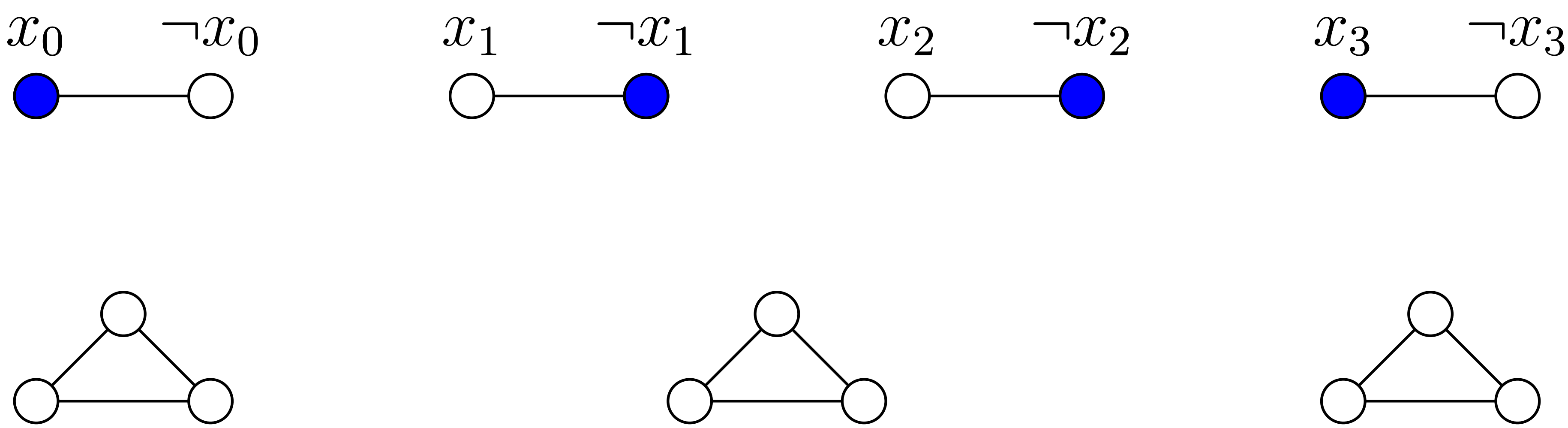
$$(x_0 \vee \neg x_1 \vee x_2) \wedge (\neg x_0 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3)$$




Cover all edges using $n + 2m = 4 + 2 \times 3 = 10$ vertices

# 3-SAT to Vertex Cover

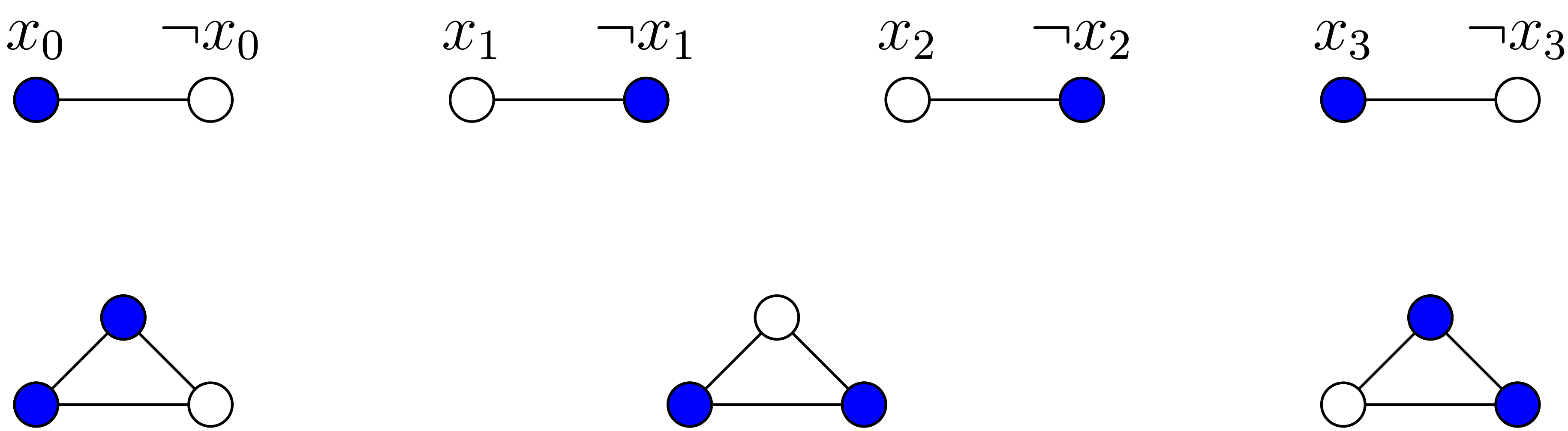
$$(x_0 \lor \neg x_1 \lor x_2) \land (\neg x_0 \lor x_2 \lor \neg x_3) \land (x_1 \lor \neg x_2 \lor x_3)$$




Cover all edges using $n + 2m = 4 + 2 \times 3 = 10$ vertices

# 3-SAT to Vertex Cover

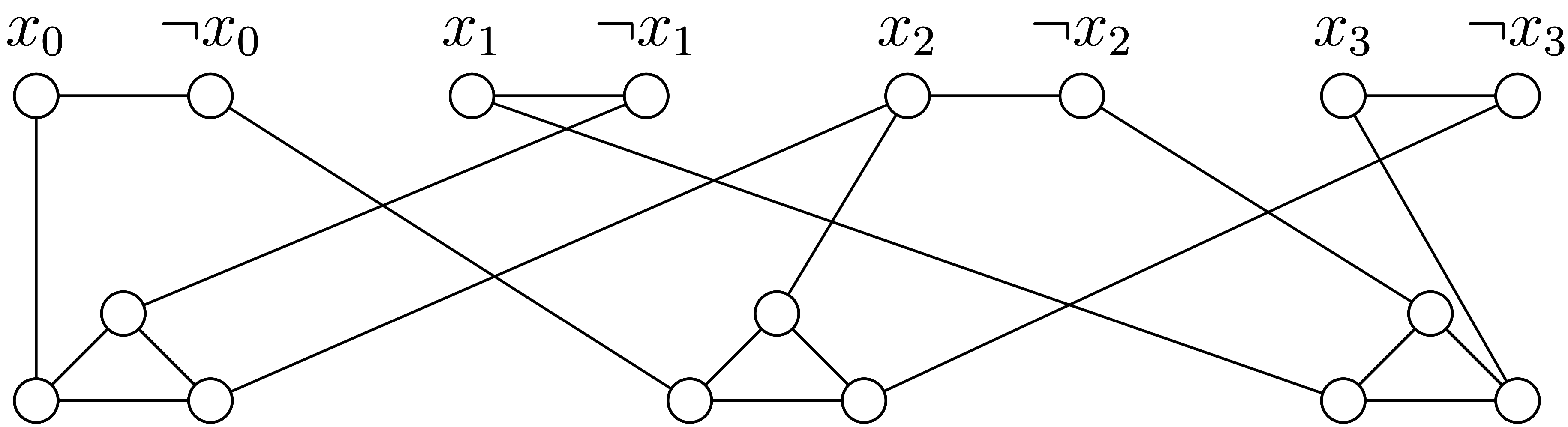
$$(x_0 \vee \neg x_1 \vee x_2) \wedge (\neg x_0 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3)$$




Cover all edges using $n + 2m = 4 + 2 \times 3 = 10$ vertices

# 3-SAT to Vertex Cover

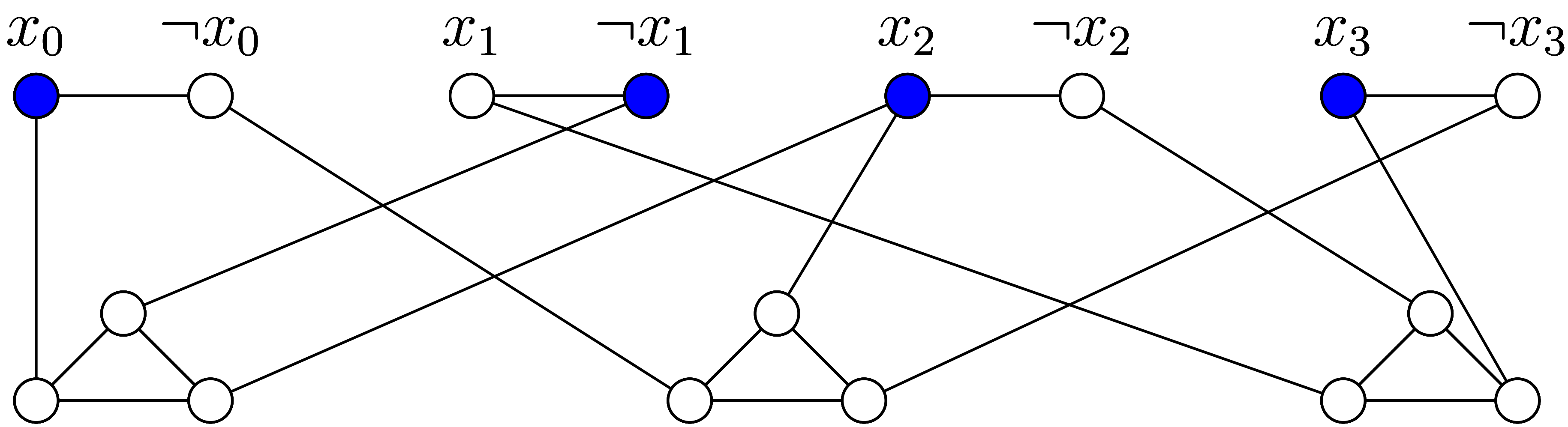
$$(x_0 \vee \neg x_1 \vee x_2) \wedge (\neg x_0 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3)$$




Cover all edges using $n + 2m = 4 + 2 \times 3 = 10$ vertices

# 3-SAT to Vertex Cover

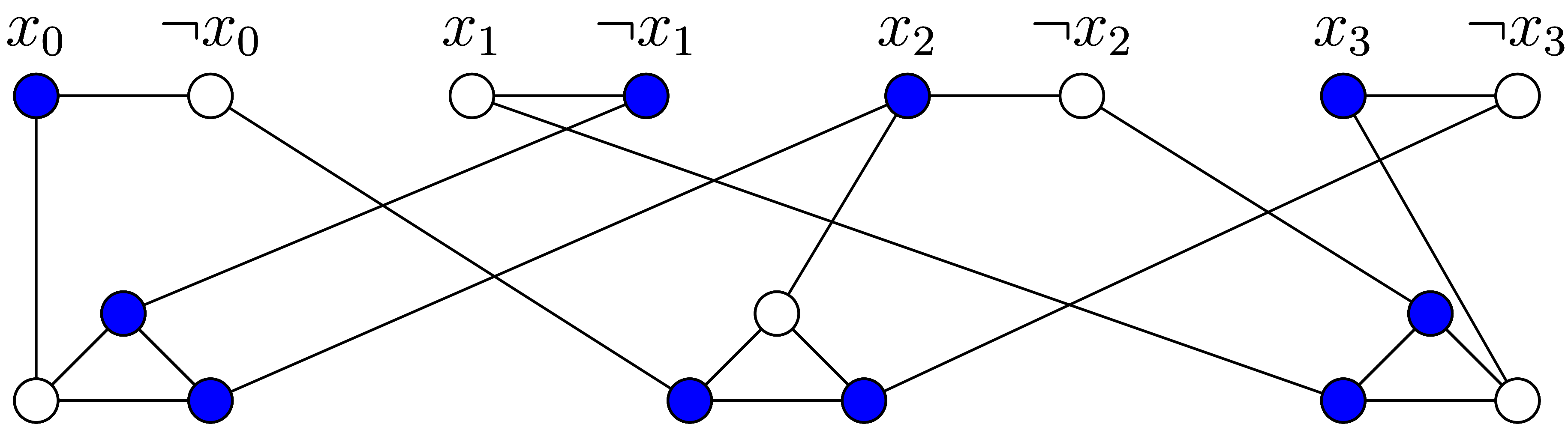
$$(x_0 \vee \neg x_1 \vee x_2) \wedge (\neg x_0 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3)$$




Cover all edges using $n + 2m = 4 + 2 \times 3 = 10$ vertices

# 3-SAT to Vertex Cover

$$(x_0 \vee \neg x_1 \vee x_2) \wedge (\neg x_0 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3)$$




Cover all edges using $n + 2m = 4 + 2 \times 3 = 10$ vertices

# Other Examples of NP-complete

- As we can polynomial reduce any instance of SAT to vertex cover then vertex cover is also NP-complete

- Lots of problems have been shown to be in class NP-complete—some $10\,000$, or so, to date

- These include
  - TSP
  - Graph colouring
  - Many scheduling problems
  - Bin-packing
  - Quadratic integer problems

# Other Examples of NP-complete

- As we can polynomial reduce any instance of SAT to vertex cover then vertex cover is also NP-complete
- Lots of problems have been shown to be in class NP-complete—some $10\,000$, or so, to date
- These include
  - TSP
  - Graph colouring
  - Many scheduling problems
  - Bin-packing
  - Quadratic integer problems

# Other Examples of NP-complete

- As we can polynomial reduce any instance of SAT to vertex cover then vertex cover is also NP-complete

- Lots of problems have been shown to be in class NP-complete—some $10\,000$, or so, to date

- These include
  - TSP
  - Graph colouring
  - Many scheduling problems
  - Bin-packing
  - Quadratic integer problems

# Other Examples of NP-complete

- As we can polynomial reduce any instance of SAT to vertex cover then vertex cover is also NP-complete
- Lots of problems have been shown to be in class NP-complete—some $10\,000$, or so, to date
- These include
  - TSP
  - Graph colouring
  - Many scheduling problems
  - Bin-packing
  - Quadratic integer problems

# Other Examples of NP-complete

- As we can polynomial reduce any instance of SAT to vertex cover then vertex cover is also NP-complete

- Lots of problems have been shown to be in class NP-complete—some $10\,000$, or so, to date

- These include
  - TSP
  - Graph colouring
  - Many scheduling problems
  - Bin-packing
  - Quadratic integer problems

# Other Examples of NP-complete

- As we can polynomial reduce any instance of SAT to vertex cover then vertex cover is also NP-complete
- Lots of problems have been shown to be in class NP-complete—some $10\,000$, or so, to date
- These include
  - ⋆ TSP
  - ⋆ Graph colouring
  - ⋆ Many scheduling problems
  - ⋆ Bin-packing
  - ⋆ Quadratic integer problems

# Other Examples of NP-complete

- As we can polynomial reduce any instance of SAT to vertex cover then vertex cover is also NP-complete
- Lots of problems have been shown to be in class NP-complete—some $10\,000$, or so, to date
- These include
  - TSP
  - Graph colouring
  - Many scheduling problems
  - Bin-packing
  - Quadratic integer problems

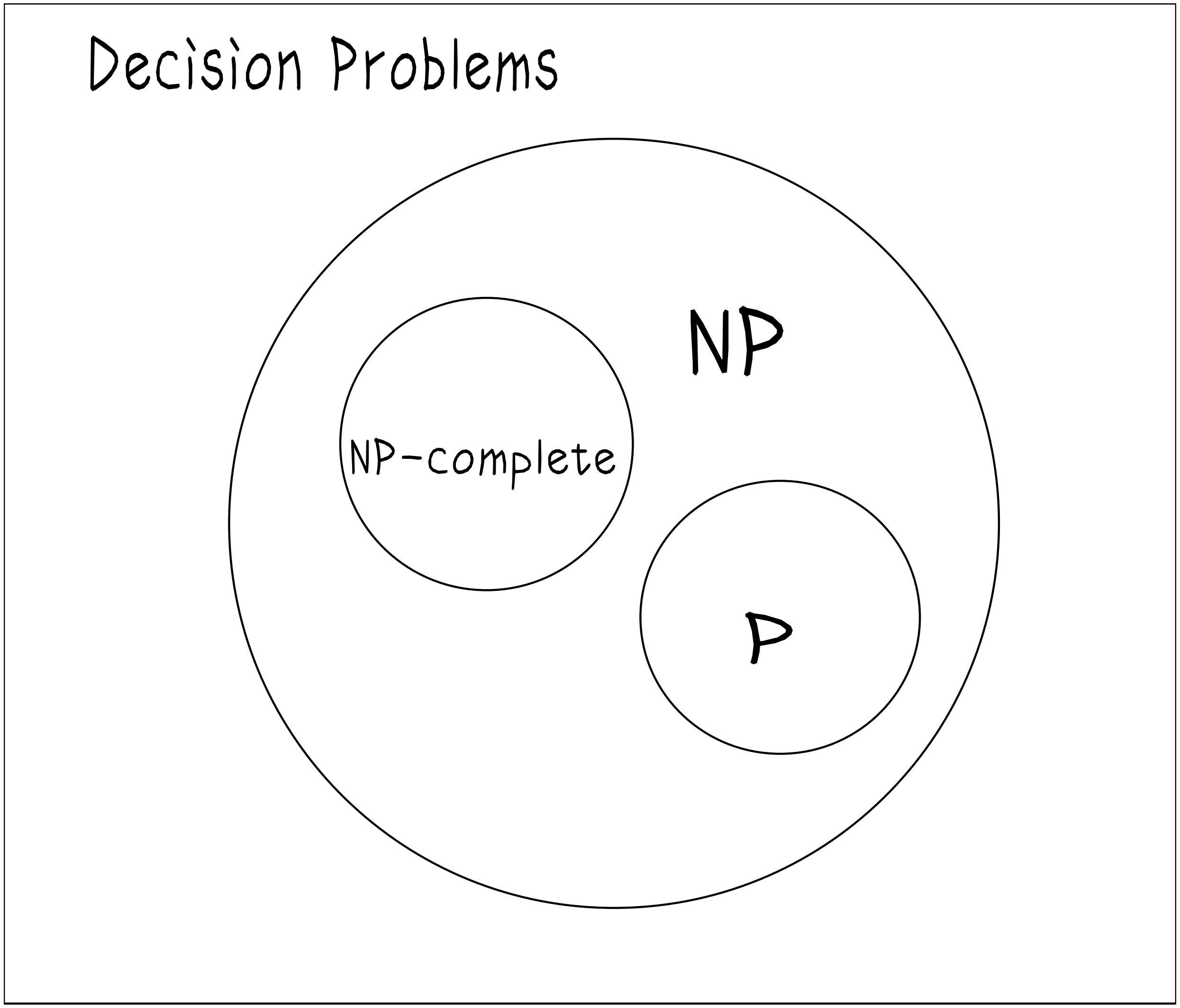
# Other Examples of NP-complete

- As we can polynomial reduce any instance of SAT to vertex cover then vertex cover is also NP-complete

- Lots of problems have been shown to be in class NP-complete—some $10\,000$, or so, to date

- These include
  - ⋆ TSP
  - ⋆ Graph colouring
  - ⋆ Many scheduling problems
  - ⋆ Bin-packing
  - ⋆ Quadratic integer problems

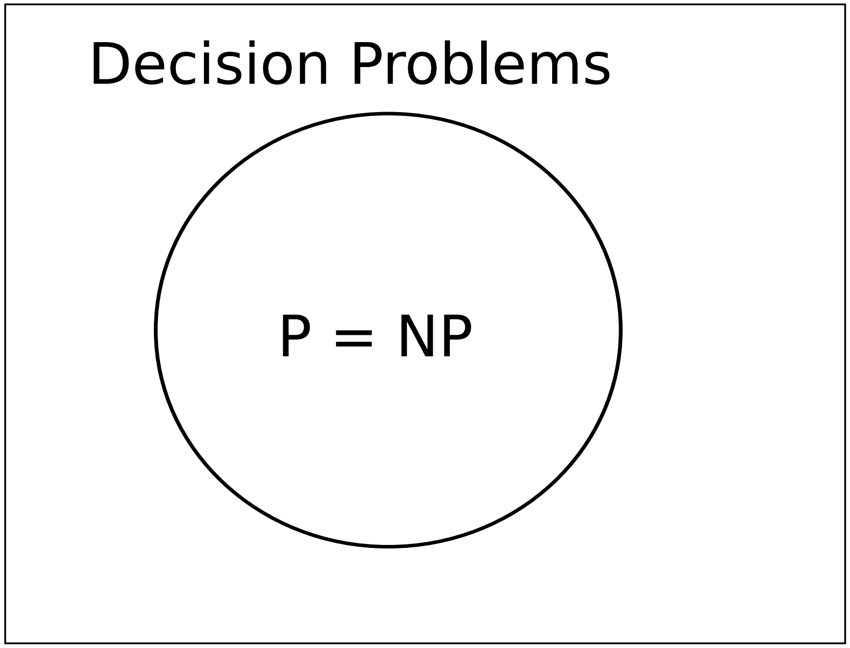
# Structure of Decision Problems



# $P \neq NP$?

- No one has proved that any problem in NP is not solvable in polynomial time

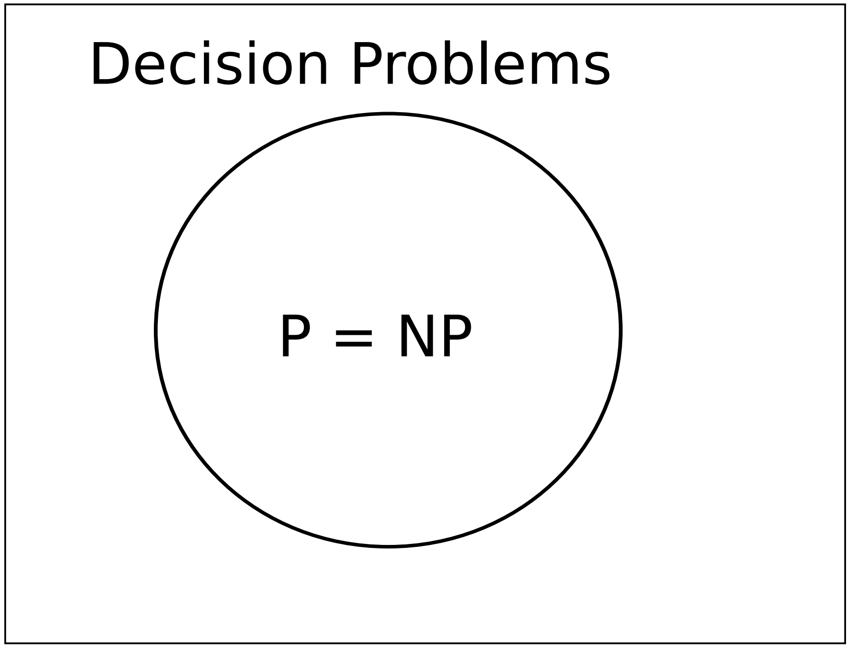
- If any NP-complete problem was solved in polynomial time all problems in NP would be solve and NP=P



# $P \neq NP$?

- No one has proved that any problem in NP is not solvable in polynomial time

- If any NP-complete problem was solved in polynomial time all problems in NP would be solve and NP=P



# NP-Hard

- TSP is not a decision problem—although we can make it into one—Is there a tour shorter that $L$?
- However, if we can find the shortest tour in polynomial time we could solve the TSP decision problem
- Thus finding the shortest tour is at least as hard as solving the decision problems
- Problems that are at least as hard as NP-complete decision problems are said to be in **NP-hard**
- Graph colouring (finding a colouring with the least number of conflicts), job scheduling, etc. are all examples of NP-hard problems

# NP-Hard

- TSP is not a decision problem—although we can make it into one—Is there a tour shorter that $L$?
- However, if we can find the shortest tour in polynomial time we could solve the TSP decision problem
- Thus finding the shortest tour is at least as hard as solving the decision problems
- Problems that are at least as hard as NP-complete decision problems are said to be in **NP-hard**
- Graph colouring (finding a colouring with the least number of conflicts), job scheduling, etc. are all examples of NP-hard problems

# NP-Hard

- TSP is not a decision problem—although we can make it into one—Is there a tour shorter that $L$?
- However, if we can find the shortest tour in polynomial time we could solve the TSP decision problem
- Thus finding the shortest tour is at least as hard as solving the decision problems
- Problems that are at least as hard as NP-complete decision problems are said to be in **NP-hard**
- Graph colouring (finding a colouring with the least number of conflicts), job scheduling, etc. are all examples of NP-hard problems

# NP-Hard

- TSP is not a decision problem—although we can make it into one—Is there a tour shorter that $L$?
- However, if we can find the shortest tour in polynomial time we could solve the TSP decision problem
- Thus finding the shortest tour is at least as hard as solving the decision problems
- Problems that are at least as hard as NP-complete decision problems are said to be in **NP-hard**
- Graph colouring (finding a colouring with the least number of conflicts), job scheduling, etc. are all examples of NP-hard problems

# NP-Hard

- TSP is not a decision problem—although we can make it into one—Is there a tour shorter that $L$?
- However, if we can find the shortest tour in polynomial time we could solve the TSP decision problem
- Thus finding the shortest tour is at least as hard as solving the decision problems
- Problems that are at least as hard as NP-complete decision problems are said to be in **NP-hard**
- Graph colouring (finding a colouring with the least number of conflicts), job scheduling, etc. are all examples of NP-hard problems

# Not All Hard Problems are NP-Hard

- Graph isomorphism, GI, (are two graphs identical up to a relabelling of the vertices?) has not been proved to be NP-complete—it is postulated that

$$GI \in \text{NP} \wedge GI \notin \text{P} \wedge GI \notin \text{NP-complete}$$

- Factoring is $not$ believed to be NP-hard, but it is believed to be sufficiently hard that most banks use an encryption technique based on people not being able to factor large numbers easily

- For large problems polynomial algorithms can take too long

# Not All Hard Problems are NP-Hard

- Graph isomorphism, GI, (are two graphs identical up to a relabelling of the vertices?) has not been proved to be NP-complete—it is postulated that

$$GI \in \text{NP} \wedge GI \notin \text{P} \wedge GI \notin \text{NP-complete}$$

- Factoring is *not* believed to be NP-hard, but it is believed to be sufficiently hard that most banks use an encryption technique based on people not being able to factor large numbers easily

- For large problems polynomial algorithms can take too long

# Not All Hard Problems are NP-Hard

- Graph isomorphism, GI, (are two graphs identical up to a relabelling of the vertices?) has not been proved to be NP-complete—it is postulated that

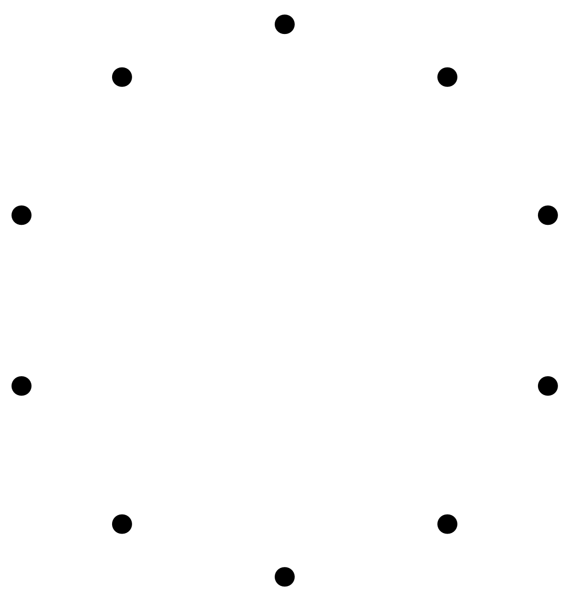
$$GI \in \text{NP} \wedge GI \notin \text{P} \wedge GI \notin \text{NP-complete}$$

- Factoring is *not* believed to be NP-hard, but it is believed to be sufficiently hard that most banks use an encryption technique based on people not being able to factor large numbers easily

- For large problems polynomial algorithms can take too long

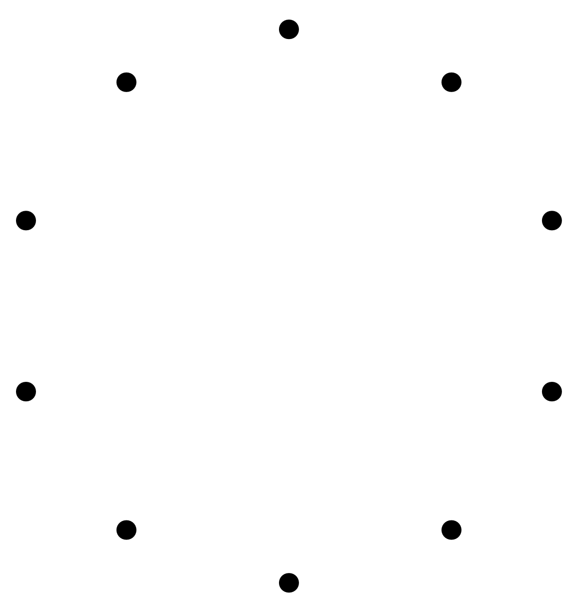
# Not All NP-Hard Problem Instances are Hard

- NP-hardness is a worst case analysis
- It means there exist some instance of the problem that we don't know how to solve in polynomial time
- Many instance of the problem might be rather easy to solve
- What is the optimal TSP tour for the problem below?



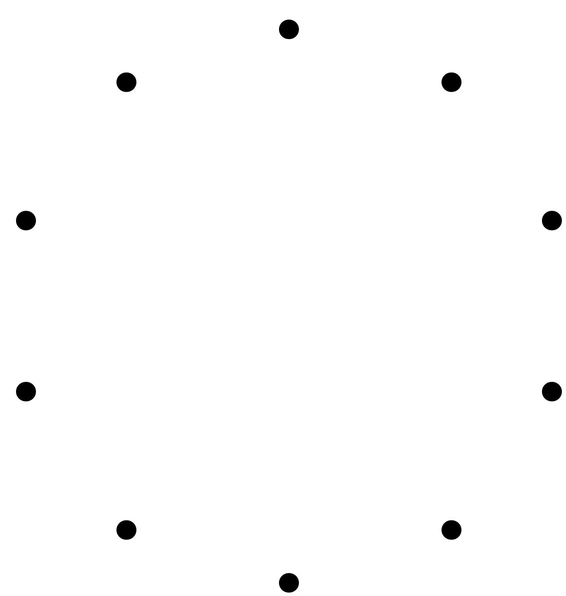
# Not All NP-Hard Problem Instances are Hard

- NP-hardness is a worst case analysis
- It means there exist some instance of the problem that we don't know how to solve in polynomial time
- Many instance of the problem might be rather easy to solve
- What is the optimal TSP tour for the problem below?



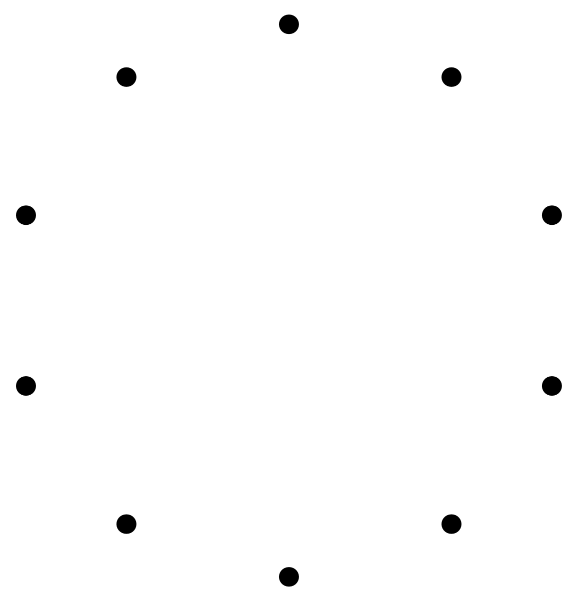
# Not All NP-Hard Problem Instances are Hard

- NP-hardness is a worst case analysis
- It means there exist some instance of the problem that we don't know how to solve in polynomial time
- Many instance of the problem might be rather easy to solve
- What is the optimal TSP tour for the problem below?



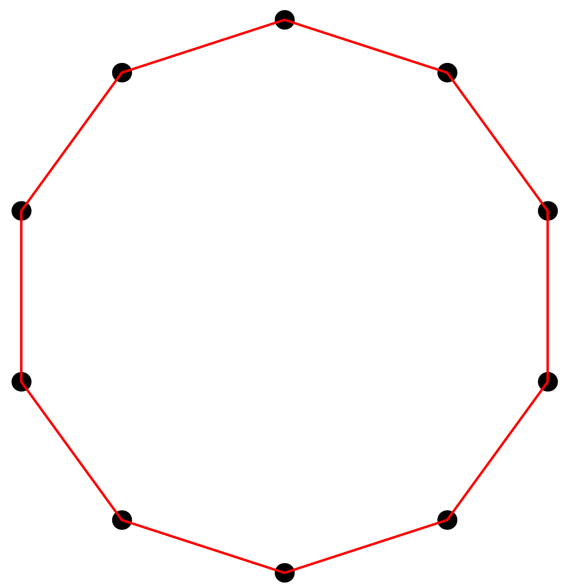
# Not All NP-Hard Problem Instances are Hard

- NP-hardness is a worst case analysis
- It means there exist some instance of the problem that we don't know how to solve in polynomial time
- Many instance of the problem might be rather easy to solve
- What is the optimal TSP tour for the problem below?



# Not All NP-Hard Problem Instances are Hard

- NP-hardness is a worst case analysis
- It means there exist some instance of the problem that we don't know how to solve in polynomial time
- Many instance of the problem might be rather easy to solve
- What is the optimal TSP tour for the problem below?



# Not All NP-Hard Problems are Hard

- For some problems almost all instances appear easy

- E.g. The subset-sum problem
  - ⋆ Given a set of numbers find a subset whose sums is as close as possible to some constant
  - ⋆ Subset-sum is in NP-hard but there exist a "pseudo-polynomial time" algorithm which solves almost every instance in polynomial time

- Many problems including subset-sum are known to be easy to approximate

- For other problems even finding a good approximation is known to be NP-hard

# Not All NP-Hard Problems are Hard

- For some problems almost all instances appear easy
- E.g. The subset-sum problem
  - ⋆ Given a set of numbers find a subset whose sums is as close as possible to some constant
  - ⋆ Subset-sum is in NP-hard but there exist a “pseudo-polynomial time” algorithm which solves almost every instance in polynomial time
- Many problems including subset-sum are known to be easy to approximate
- For other problems even finding a good approximation is known to be NP-hard

# Not All NP-Hard Problems are Hard

- For some problems almost all instances appear easy
- E.g. The subset-sum problem
  - ⋆ Given a set of numbers find a subset whose sums is as close as possible to some constant
  - ⋆ Subset-sum is in NP-hard but there exist a “pseudo-polynomial time” algorithm which solves almost every instance in polynomial time
- Many problems including subset-sum are known to be easy to approximate
- For other problems even finding a good approximation is known to be NP-hard

# Not All NP-Hard Problems are Hard

- For some problems almost all instances appear easy
- E.g. The subset-sum problem
  - ⋆ Given a set of numbers find a subset whose sums is as close as possible to some constant
  - ⋆ Subset-sum is in NP-hard but there exist a "pseudo-polynomial time" algorithm which solves almost every instance in polynomial time
- Many problems including subset-sum are known to be easy to approximate
- For other problems even finding a good approximation is known to be NP-hard

# Not All NP-Hard Problems are Hard

- For some problems almost all instances appear easy
- E.g. The subset-sum problem
  - ⋆ Given a set of numbers find a subset whose sums is as close as possible to some constant
  - ⋆ Subset-sum is in NP-hard but there exist a "pseudo-polynomial time" algorithm which solves almost every instance in polynomial time
- Many problems including subset-sum are known to be easy to approximate
- For other problems even finding a good approximation is known to be NP-hard

# Not All NP-Hard Problems are Hard

- For some problems almost all instances appear easy
- E.g. The subset-sum problem
  - ⋆ Given a set of numbers find a subset whose sums is as close as possible to some constant
  - ⋆ Subset-sum is in NP-hard but there exist a "pseudo-polynomial time" algorithm which solves almost every instance in polynomial time
- Many problems including subset-sum are known to be easy to approximate
- For other problems even finding a good approximation is known to be NP-hard

# Lessons

- There exist efficient algorithms for many problems. . .
- . . . but probably not for all
- There are no known polynomial algorithm for any NP-complete problem
- These include many famous problems: TSP, graph-colouring, scheduling, . . .
- If you could find a polynomial algorithm for any of these problems then you could use it to solve all problems in NP in polynomial time

# Lessons

- There exist efficient algorithms for many problems. . .
- . . . but probably not for all
- There are no known polynomial algorithm for any NP-complete problem
- These include many famous problems: TSP, graph-colouring, scheduling, . . .
- If you could find a polynomial algorithm for any of these problems then you could use it to solve all problems in NP in polynomial time

# Lessons

- There exist efficient algorithms for many problems. . .
- . . . but probably not for all
- There are no known polynomial algorithm for any NP-complete problem
- These include many famous problems: TSP, graph-colouring, scheduling, . . .
- If you could find a polynomial algorithm for any of these problems then you could use it to solve all problems in NP in polynomial time

# Lessons

- There exist efficient algorithms for many problems. . .
- . . . but probably not for all
- There are no known polynomial algorithm for any NP-complete problem
- These include many famous problems: TSP, graph-colouring, scheduling, . . .
- If you could find a polynomial algorithm for any of these problems then you could use it to solve all problems in NP in polynomial time

# Lessons

- There exist efficient algorithms for many problems. . .
- . . . but probably not for all
- There are no known polynomial algorithm for any NP-complete problem
- These include many famous problems: TSP, graph-colouring, scheduling, . . .
- If you could find a polynomial algorithm for any of these problems then you could use it to solve all problems in NP in polynomial time