# NP-complete Problems: Search Problems

Alexander S. Kulikov

Steklov Institute of Mathematics at St. Petersburg
Russian Academy of Sciences

## Advanced Algorithms and Complexity
## Data Structures and Algorithms

# Outline

1. Brute Force Search
2. Search Problems
3. Easy and Hard Problems
   - Traveling Salesman Problem
   - Hamiltonian Cycle Problem
   - Longest Path Problem
   - Integer Linear Programming Problem
   - Independent Set Problem
4. P and NP

# Polynomial vs Exponential

Polynomial time

| running time: | $n$ | $n^2$ | $n^3$ | $2^n$ exponential time |
|---|---|---|---|---|
| less than $10^9$: | $10^9$ | $10^{4.5}$ | $10^3$ | 29 |

Second line is max value for n for which the total number of steps performed by the algorithm stays below 10^9 . Why 10^9 because most modern computers can perform 10^9 operations in 1 sec

as n is very small here for exponential function and exp function grows fastest these algo's are considered impractical

# Improving Brute Force Search

Usually, an efficient (polynomial) algorithm searches for a solution among an exponential number of candidates:

- there are $n!$ permutations of $n$ objects

For many computational problems the corresponding set of all candidate solutions is exponential.

For example if we have to find an optimal permutation of this object then the naive way is to go through all permutations (n!) and pick the optimal solution this takes time n! which grows even faster than exponential function

# Improving Brute Force Search

Usually, an efficient (polynomial) algorithm searches for a solution among an exponential number of candidates:

- there are $n!$ permutations of $n$ objects
- there are $2^n$ ways to partition $n$ objects into two sets

Another example is if we are given an object and we have to divide it into two sets for example we need to partition a set of vertices of a graph in two sets to find a cut. Then again naive way is to go through all vertices and select optimal. However, there are 2 to n ways to split the n objects into two sets. hence running time is 2^n which allows us to handle instances of size roughly 30 in less than 1 second.

# Improving Brute Force Search

Usually, an efficient (polynomial) algorithm searches for a solution among an exponential number of candidates:

- there are $n!$ permutations of $n$ objects
- there are $2^n$ ways to partition $n$ objects into two sets
- there are $n^{n-2}$ spanning trees in a complete graph on $n$ vertices

Another example is finding a min spanning tree in a complete graph. Naive way is to go through all spanning tree and select one with min weight but there are total n^(n-2) spanning trees

# This module

- For thousands of practically important problems we don't have an efficient algorithm yet

hence a polynomial algorithm is called efficient because it avoids going through all candidate solutions which usually has exponential size.

There are many computational problems for which we still don't know efficient (Polynomial time) algorithm. Hence we naively go through all possible candidate solution and select the best solution. This is the best we can do now.

# This module

- For thousands of practically important problems we don't have an efficient algorithm yet
- An efficient algorithm for one such problem automatically gives efficient algorithms for all these problems!

# This module

- For thousands of practically important problems we don't have an efficient algorithm yet
- An efficient algorithm for one such problem automatically gives efficient algorithms for all these problems!
- \$1M prize for constructing such an algorithm or proving that this is impossible!

# Outline

# Boolean Formulas

We will now give a formal definition of a search problem. And we will do this by considering the famous boolean satisfiability problem

## Formula in conjunctive normal form

This clause says that either x or y or z = 1

This clause says that x or y or z = 0

$$(x \vee y \vee z)(x \vee \overline{y})(y \vee \overline{z})(z \vee \overline{x})(\overline{x} \vee \overline{y} \vee \overline{z})$$

- $x, y, z$ are Boolean variables (values: `true`/`false` or 1/0)
- literals are variables ($x, y, z$) and their negations ($\overline{x}, \overline{y}, \overline{z}$)
- clauses are disjunctions (logical or) of literals

Input for this problem is formula in CNF

So a formal in conjunctive number form is just a set of clauses. In this eg we have five clauses

## Satisfiability (SAT)

**Input:** Formula $F$ in conjunctive normal form (CNF).

**Output:** An assignment of Boolean values to the variables of $F$ satisfying all clauses, if exists.

# Examples

- The formula $(x \vee \overline{y})(\overline{x} \vee \overline{y})(x \vee y)$ is satisfiable: set $x = 1$, $y = 0$. just one set satisfies
- The formula $(x \vee y \vee z)(x \vee \overline{y})(y \vee \overline{z})$ is satisfiable: set $x = 1, y = 1, z = 1$ or $x = 1, y = 0, z = 0$. more than one set satisfies
- The formula $(x \vee y \vee z)(x \vee \overline{y})(y \vee \overline{z})(z \vee \overline{x})(\overline{x} \vee \overline{y} \vee \overline{z})$ is unsatisfiable. 8 assignments for x,y,z possible from 000 to 111
None of them satisfies

# Satisfiability

This Problem of satisfiability is called canonical hard problems

**Question**

Is the following formula satisfiable?

$(x_1 \vee \bar{x}_2 \vee \bar{x}_3)(x_3)(x_1 \vee x_2)(\bar{x}_1 \vee \bar{x}_2)$

- Yes, it is satisfiable.
- No, it is not satisfiable.

Correct

That's right! To satisfy it, set $x_1 = 1, x_2 = 0, x_3 = 1$.

- Classical hard problem
- Many applications: e.g., hardware/software verification, planning, scheduling in particular because many hard combinatorial problem is reduced to satisfiability problem
- Many hard problems are stated in terms of SAT naturally
- SAT solvers (will see later), SAT competition SAT solvers are programs which solves satisfiability problem

- SAT is a typical search problem
- Search problem: given an instance $I$, find a solution $S$ or report that none exists

For example in case of the SAT problem an instance I is a formula in CNF and solution S is a satisfying assignment

- Main property: one must be able to check quickly whether $S$ is indeed a solution for $I$

For example In case of SAT it is easy if we are given a truth assignment of values to all the variables, we can quickly check whether it satisfies all clauses. We go through all clauses from left to right and find a literal which satisfies all the clauses

- By saying quickly, we mean in time polynomial in the length of $I$. In particular, the length of $S$ should be polynomial in the length of $I$

Another natural property is we require the length of S to be polynomial in the length of I.Hence S should not be very large in particular we do not want S to have for example exponential size in the length of I. If this is the case then it would require us an exponential time just to write down a solution for instance I

and not exponential

## Definition

A search problem is defined by an algorithm $\mathcal{C}$ that takes an instance $I$ and a candidate solution $S$, and runs in time polynomial in the length of $I$. We say that $S$ is a solution to $I$ iff $\mathcal{C}(S, I) = \texttt{true}$.

## Example

For SAT, $I$ is a Boolean formula, $S$ is an assignment of Boolean constants to its variables. The corresponding algorithm $\mathcal{C}$ checks whether $S$ satisfies all clauses of $I$.

## Next part

A few practical search problems for which polynomial algorithms remain unknown

# Outline

1. Brute Force Search
2. Search Problems
3. Easy and Hard Problems
   - Traveling Salesman Problem
   - Hamiltonian Cycle Problem
   - Longest Path Problem
   - Integer Linear Programming Problem
   - Independent Set Problem
4. P and NP

# Outline

## Traveling salesman problem (TSP)

Input: Pairwise distances between $n$ cities and a budget $b$.

Output: A cycle that visits each vertex exactly once and has total length at most $b$.

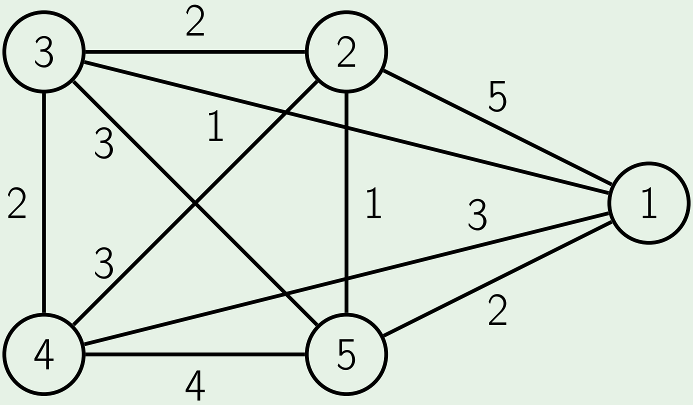
# Delivery Company

https://simple.wikipedia.org/wiki/Travelling_salesman_problem

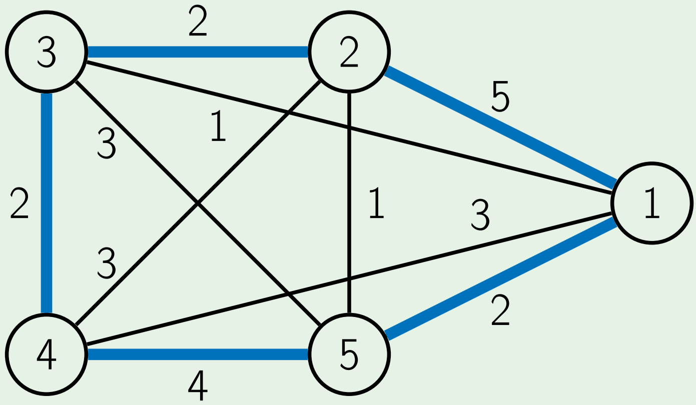
# Drilling Holes in a Circuit Board

https://developers.google.com/optimization/routing/tsp

# Example

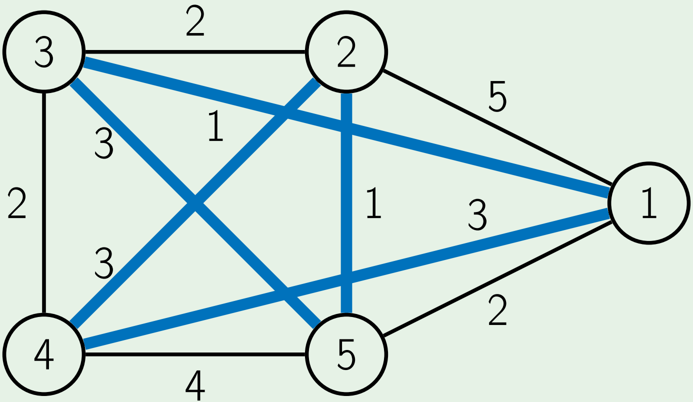


## Example



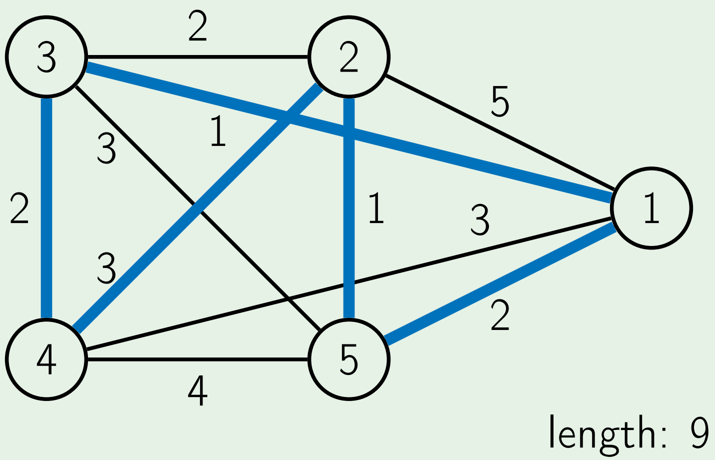

length: 15

## Example




length: 13

# Example



# Search Problem

- TSP is a search problem: given a sequence of vertices, it is easy to check whether it is a cycle visiting all the vertices of total length at most $b$
- TSP is usually stated as an optimization problem; we stated its decision version to guarantee that a candidate solution can be efficiently checked for correctness

Note that optimization Problem and decision version are very different. If we have algorithm which solves optimization problem (shortest path) then it obviously solves the decision problem (path length < b) and if we have algo to solve the decision problem we can solve the optimization problem as well by assuming path length some number say 100 and then check if there is a cycle /path length (with constraints atmost b) if yes we check for 50 if no then we check for 75 and likewise and finally coming to the optimal cycle which visits each vertex exactly once through some iterations. This is done by calling algo logarithmic no of times

# Algorithms

- Check all permutations: about $O(n!)$, extremely slow

Naive way. Checking all possible permutations. Just for n=15 running time becomes n! = 15! = 10^12

- Dynamic programming: $O(n^2 2^n)$

This is the best algo available so far. In particular there is no algo which can solve this problem in 1.99^n

- No significantly better upper bound is known
- There are heuristic algorithms and approximation algorithms

These heuristic algorithm solves these problems for practical instances quite fast however there is no guarantee on the running time of such algorithm. There are also approximation algorithms. For such algorithms we have guarantee on the running time but what they return is not an optimal solution but the solution which is not much worse than optimal

# Comparing to MST

## MST

Decision version: given $n$ cities, connect them by $(n-1)$ roads of minimal total length

**Question**

As we've discussed in the lecture, the brute force search algorithm for the traveling salesman problem makes roughly $n!$ steps for a graph with $n$ vertices. Later in this class, we'll see a dynamic programming based algorithm with running time about $n^2 2^n$. To see the dramatic difference between these two running times, let's estimate how much the second algorithm is faster than the first one for $n = 100$. What is $\frac{100!}{100^2 \cdot 2^{100}}$?

- About $10^{10}$.
- About $10^{40}$.
- About $10^{80}$.
- About $10^{123}$.
- About $10^{239}$.

**Correct**
That's right! See here

# Comparing to MST

## MST

Decision version: given $n$ cities, connect them by $(n-1)$ roads of minimal total length

Can be solved efficiently

# Comparing to MST

## MST

Decision version: given $n$ cities, connect them by $(n-1)$ roads of minimal total length

Can be solved efficiently

## TSP

Decision version: given $n$ cities, connect them in a path by $(n-1)$ roads of minimal total length

TSP is actually a MST with additional restriction that connection between the vertices (tree) should be a path

# Comparing to MST

## MST

Decision version: given $n$ cities, connect them by $(n-1)$ roads of minimal total length

Can be solved efficiently

## TSP

Decision version: given $n$ cities, connect them in a path by $(n-1)$ roads of minimal total length

No polynomial algorithm known!

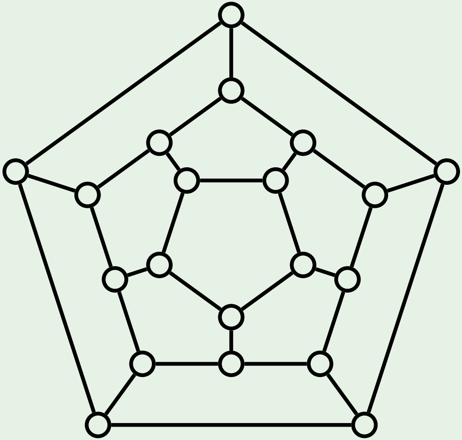
# Outline

## Hamiltonian cycle

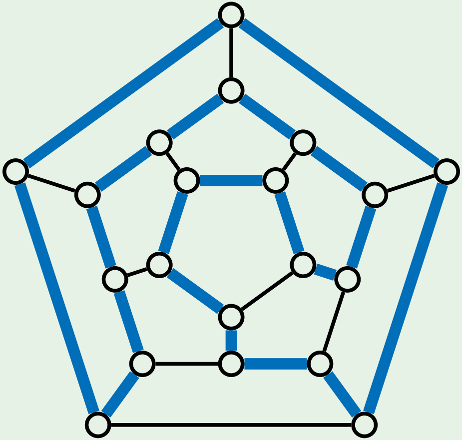
**Input:** A graph. directed or undirected without weight of the edges

**Output:** A cycle that visits each vertex of the graph exactly once.

# Example



## Example



## Eulerian cycle

Input: A graph.

Output: A cycle that visits each edge of the graph exactly once.

## Eulerian cycle

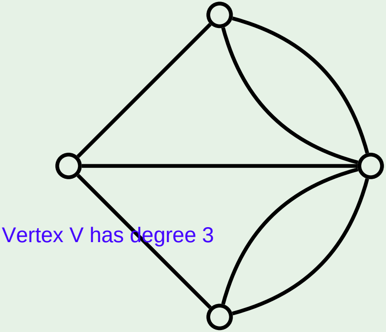
**Input:** A graph.

**Output:** A cycle that visits each edge of the graph exactly once.

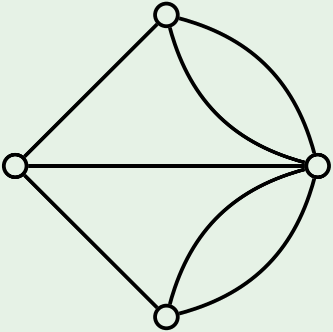
## Theorem

A graph has an Eulerian cycle if and only if it is connected and the degree of each vertex is even.

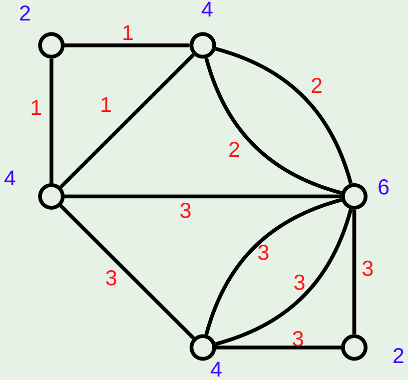
# Non-Eulerian graph



## Non-Eulerian graph



## Eulerian graph



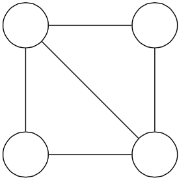

Here what we have is not just one single cycle but a bunch of cycles. but one property is such that if we have several cycles it is easy to glue them together into a single cycle

# Eulerian cycle

Find a cycle visiting each edge exactly once

Is it true that every graph that has a Hamiltonian cycle also has an Eulerian cycle?

- Yes, this is true.
- No, there exists a graph that has a Hamiltonian cycle, but has no Eulerian cycle.



Correct

That's right! One of such graphs is shown below.

## Eulerian cycle

Find a cycle visiting each edge exactly once

Can be solved efficiently

## Eulerian cycle

Find a cycle visiting each edge exactly once

## Hamiltonian cycle

Find a cycle visiting each vertex exactly once

Can be solved efficiently

**Eulerian cycle**

Find a cycle visiting each edge exactly once

**Hamiltonian cycle**

Find a cycle visiting each vertex exactly once

Can be solved efficiently

No polynomial algorithm known!

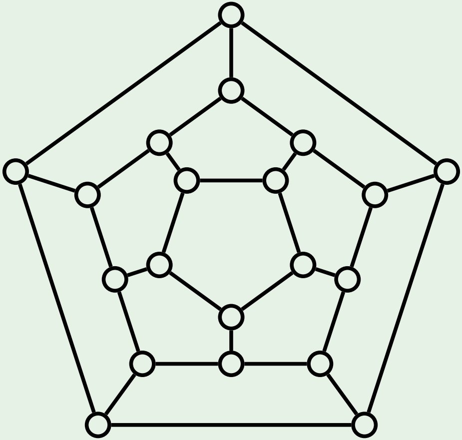
# Outline

1. Brute Force Search
2. Search Problems
3. Easy and Hard Problems
   - Traveling Salesman Problem
   - Hamiltonian Cycle Problem
   - Longest Path Problem
   - Integer Linear Programming Problem
   - Independent Set Problem
4. P and NP

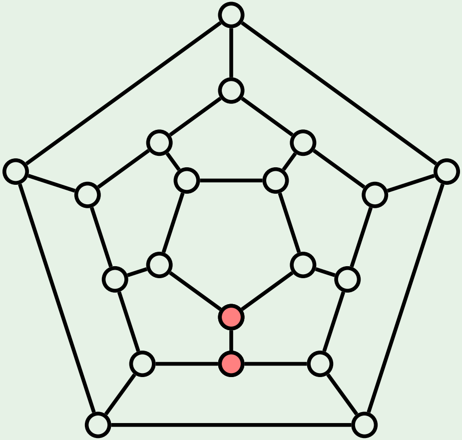
## Longest path

| | |
|---|---|
| Input: | A weighted graph, two vertices $s, t$, and a budget $b$. |
| Output: | A simple path (containing no repeated vertices) of total length at least $b$. |

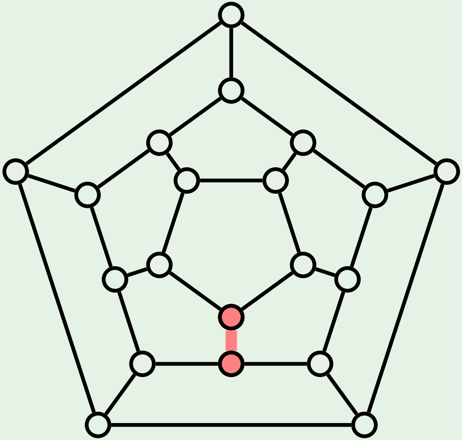
# Example



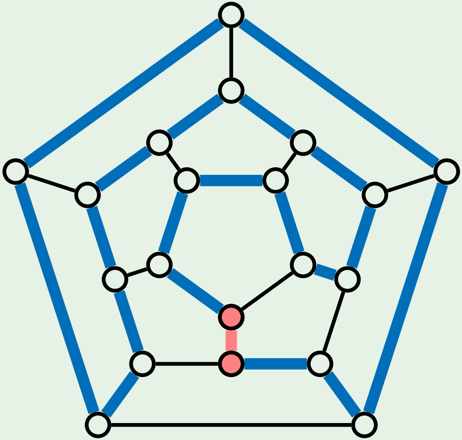
# Example



## Example



## Example



## Shortest path

Find a simple path from $s$ to $t$ of total length at most $b$

## Shortest path

Find a simple path from $s$ to $t$ of total length at most $b$

Can be solved efficiently

## Shortest path

Find a simple path from $s$ to $t$ of total length at most $b$

## Longest path

Find a simple path from $s$ to $t$ of total length at least $b$

Can be solved efficiently

## Shortest path

Find a simple path from $s$ to $t$ of total length at most $b$

## Longest path

Find a simple path from $s$ to $t$ of total length at least $b$

Can be solved efficiently

No polynomial algorithm known!

# Outline

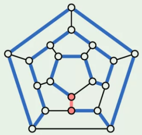
1. Brute Force Search
2. Search Problems
3. **Easy and Hard Problems**
   - Traveling Salesman Proble
   - Hamiltonian Cycle Probler
   - Longest Path Problem
   - **Integer Linear Programm**
   - Independent Set Problem
4. P and NP

**Longest path**

Input: A weighted graph, two vertices $s$, $t$, and a budget $b$.

Output: A simple path (containing no repeated vertices) of total length at least $b$.

**Example**



Longest Path Problem

| Shortest path | Longest path |
|---|---|
| Find a simple path from $s$ to $t$ of total length at most $b$ | Find a simple path from $s$ to $t$ of total length at least $b$ |
| Can be solved efficiently | No polynomial algorithm known! |

Shortest path can be solved using BFS

## Integer linear programming

Input: A set of linear inequalities $\mathbf{A}\mathbf{x} \leq \mathbf{b}$.

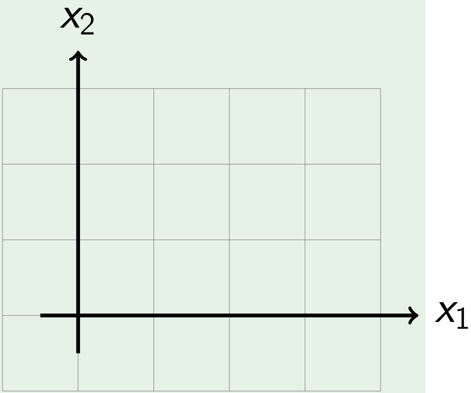
Output: Integer solution.

## Example

$$
\begin{aligned}
x_1 &\geq 0.5 \\
-x_1 + 8x_2 &\geq 0 \\
-x_1 - 8x_2 &\geq -8
\end{aligned}
$$

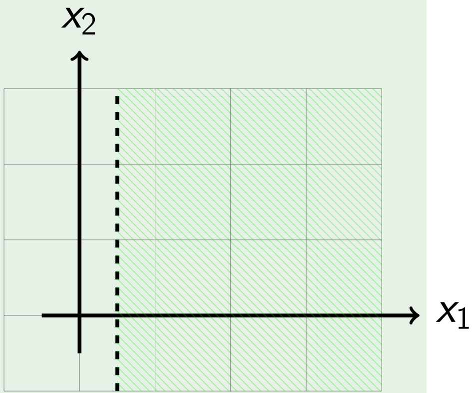
# Example

$$x_1 \geq 0.5$$
$$-x_1 + 8x_2 \geq 0$$
$$-x_1 - 8x_2 \geq -8$$



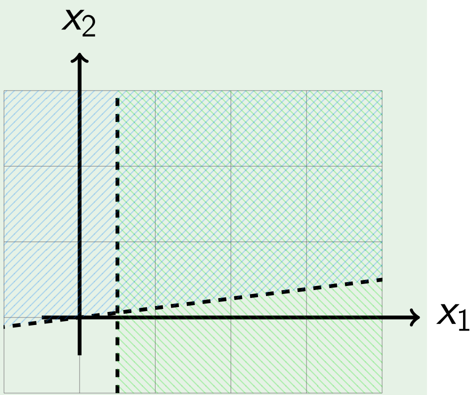
## Example

$$x_1 \geq 0.5$$
$$-x_1 + 8x_2 \geq 0$$
$$-x_1 - 8x_2 \geq -8$$



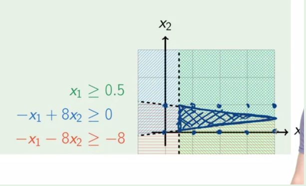
# Example

$$x_1 \geq 0.5$$
$$-x_1 + 8x_2 \geq 0$$
$$-x_1 - 8x_2 \geq -8$$

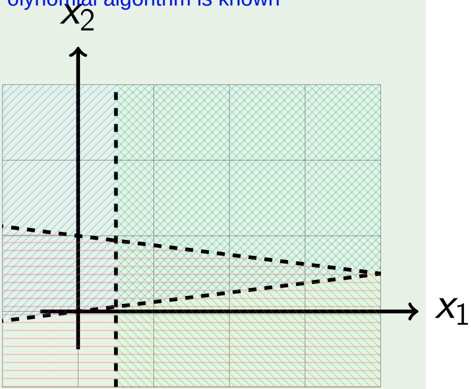


# Example

Note that this shaded feasible region does not have any integer points and it turns out that this additional restriction in which solution has to be integer gives us a very hard problem for which no Polynomial algorithm is known




$$x_1 \geq 0.5$$
$$-x_1 + 8x_2 \geq 0$$
$$-x_1 - 8x_2 \geq -8$$



# LP (decision version)

Find a real solution of a system of linear inequalities

**Question**

The satisfiability problem can be easily reduced to the integer linear programming problem. For this, for each variable $x_i$ of the input formula, add two inequalities $0 \leq x_i$ and $x_i \leq 1$. Then, for each clause like $(x_i \vee \overline{x}_j \vee \overline{x}_k)$ add an inequality $x_i + (1 - x_j) + (1 - x_k) \geq 1$. More formally, for each clause of the form $(\ell_1 \vee \ell_2 \vee \cdots \vee \ell_k)$ add an inequality $m_1 + m_2 + \cdots + m_k \geq 1$. Here, $m_i$ is equal to $\ell_i$, if $\ell_i$ is a variable $x_j$, and $m_i = (1 - x_j)$, if $l_i$ is the negation of a variable $x_j$.

What integer linear program corresponds to the formula

$(x_1 \vee x_2)(x_1 \vee \overline{x}_2)$?

- ○ $x_1 + x_2 \geq 1, x_1 - x_2 \geq 0$
- ◉ $0 \leq x_1 \leq 1, 0 \leq x_2 \leq 1, x_1 + x_2 \geq 1, x_1 - x_2 \geq 0$
- ○ $0 \leq x_1 \leq 1, 0 \leq x_2 \leq 1, x_1 + x_2 \geq 1, x_1 + x_2 \geq 0$

✓ **Correct**

That's right!

## LP (decision version)

Find a real solution of a system of linear inequalities

Can be solved efficiently

| LP (decision version) | ILP |
| --- | --- |
| Find a real solution of a system of linear inequalities | Find an integer solution of a system of linear inequalities |

| |
| --- |
| Can be solved efficiently |

Note that LP can be solved using Simplex method for which too running time is not bounded by polynomial and for some pathological cases it can have exponential running time but methods like ellipsoid method and Interior point method have polynomial upper bounds on the running time

| LP (decision version) | ILP |
| --- | --- |
| Find a real solution of a system of linear inequalities | Find an integer solution of a system of linear inequalities |

| | |
| --- | --- |
| Can be solved efficiently | No polynomial algorithm known! |

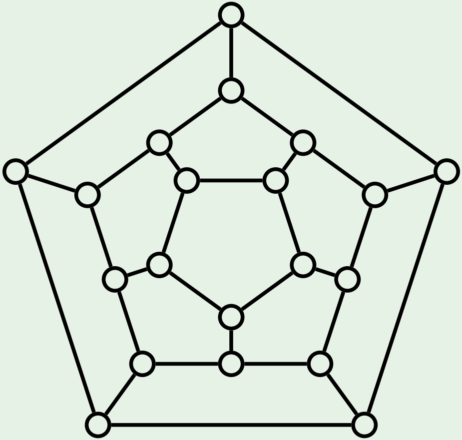
# Outline

## Independent set

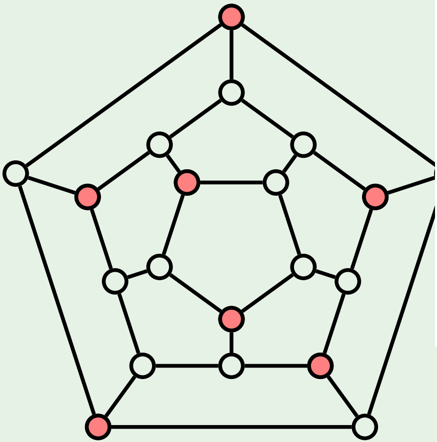
Input: A graph and a budget $b$.

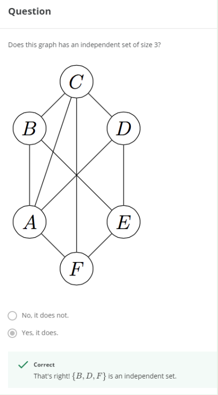
Output: A subset of vertices of size at least $b$ such that no two of them are adjacent.

## Example



# Example

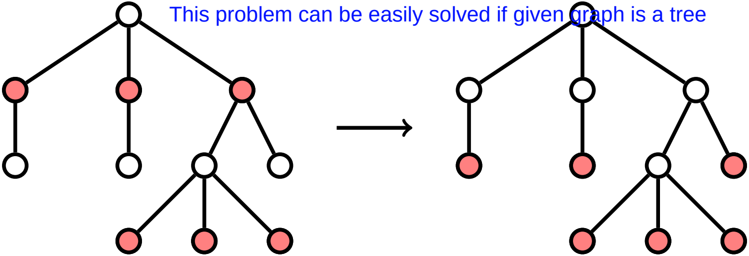



Question
Does this graph has an independent set of size 3?
$C$
$B$
$D$
$A$
$E$
$F$
No, it does not.
Yes, it does.
Correct
That's right! $\{B, D, F\}$ is an independent set.

# Independent Sets in a Tree

A maximal independent set in a tree can be found by a simple greedy algorithm: it is safe to take into a solution all the leaves.

This problem can be easily solved if given graph is a tree



Given a tree if we want to find independent set of maximum size it can be solved using greedy method just remove all the leaves from the parents and then remove new leaves likewise continue the process

## Independent set in a tree

Find an independent set of size at least $b$ in a given tree

## Independent set in a tree

Find an independent set of size at least $b$ in a given tree

Can be solved efficiently

## Independent set in a tree

Find an independent set of size at least $b$ in a given tree

## Independent set in a graph

Find an independent set of size at least $b$ in a given graph

Can be solved efficiently

### Independent set in a tree

Find an independent set of size at least $b$ in a given tree

Can be solved efficiently

### Independent set in a graph

Find an independent set of size at least $b$ in a given graph

No polynomial algorithm known!

## Next part

It turns out that all these hard problems are in a sense a single hard problem:
a polynomial time algorithm for any of these problems can be used to solve all of them in polynomial time!

# Outline

1. Brute Force Search
2. Search Problems
3. Easy and Hard Problems
   - Traveling Salesman Problem
   - Hamiltonian Cycle Problem
   - Longest Path Problem
   - Integer Linear Programming Problem
   - Independent Set Problem
4. P and NP

# Class NP

## Definition

A search problem is defined by an algorithm $\mathcal{C}$ that takes an instance $I$ and a candidate solution $S$, and runs in time polynomial in the length of $I$. We say that $S$ is a solution to $I$ iff $\mathcal{C}(S, I) = \texttt{true}$.

# Class NP

## Definition

A search problem is defined by an algorithm $\mathcal{C}$ that takes an instance $I$ and a candidate solution $S$, and runs in time polynomial in the length of $I$. We say that $S$ is a solution to $I$ iff $\mathcal{C}(S, I) = \texttt{true}$.

## Definition

**NP** is the class of all search problems.

- **NP** stands for "non-deterministic polynomial time": one can guess a solution, and then verify its correctness in polynomial time
- In other words, the class **NP** contains all problems whose solutions can be efficiently verified

**Question**

The Subset sum problem is defined as follows: Given a set of $n$ integers, check whether some subset of them sums up to 0. For example, for the input $-2, -3, 15, 14, 7, -10$ the answer is yes, because $(-2) + (-3) + 15 + (-10) = 0$. Does this problem belong to the class NP?

- (•) Yes, it is in NP.
- ( ) No, it is not in NP.
- ( ) It is not known.

✓ Correct

That's right! To show that it is in NP we need to guarantee that there is a polynomial time algorithm that verifies a solution. This is straightforward: given a subset of numbers, we check whether it is indeed a subset of the input numbers and whether its sum is zero.

# Class $\mathbf{P}$

**Definition**

$\mathbf{P}$ is the class of all search problems that can be solved in polynomial time.

## Class P

Problems whose solution can be found efficiently

## Class P

Problems whose solution can be found efficiently

- MST
- Shortest path
- LP
- IS on trees

## Class P

Problems whose solution can be found efficiently

## Class NP

Problems whose solution can be verified efficiently

- MST
- Shortest path
- LP
- IS on trees

## Class **P**

Problems whose solution can be found efficiently

- MST
- Shortest path
- LP
- IS on trees

## Class **NP**

Problems whose solution can be verified efficiently

- TSP
- Longest path
- ILP
- IS on graphs

## The main open problem in Computer Science

Is $\mathbf{P}$ equal to $\mathbf{NP}$?

**The main open problem in Computer Science**

Is **P** equal to **NP**?

**Millenium Prize Problem**

Clay Mathematics Institute: $1M prize for solving the problem

- If **P**=**NP**, then all search problems can be solved in polynomial time.

- If P$=$NP, then all search problems can be solved in polynomial time.
- If P$\neq$NP, then there exist search problems that cannot be solved in polynomial time.

## Next part

We'll show that the satisfiability problem, the traveling salesman problem, the independent set problem, the integer linear programming are the hardest problems in **NP**.