GRAPH THEORY
[8]

Introduction to
NP complete

Documents are here:

https://www-l2ti.univ-paris13.fr/~viennet/ens/2024-USTH-Graphs

Emmanuel Viennet
emmanuel.viennet@univ-paris13.fr

USTH

Université Sorbonne Paris Nord

Polynomial Time

Most (but not all !) of the algorithms we have studied so far are easy, in that they can be solved in *polynomial time*, be it linear, quadratic, cubic, etc.

Cubic may not sound very fast, and isn't when compared to linear, but compared to exponential we have seen that it has a much better asymptotic behavior.

There are **many algorithms which are not polynomial**. In general, if the space of possible solutions grows exponentially as *n* increases, then we should not hope for a polynomial time algorithm.

Polynomial time algorithms are the exception rather than the rule.

Hard problems

Problems with no known polynomial solution are called *Hard problems*.

Another class of problems (NP) seem *like* they should be easy. They have the following property:

- A solution can be **verified** in polynomial time

• **NP** stands for "*Nondeterministic Polynomial time* ". It's a **complexity class** in computer science that represents problems for which a given solution can be *verified* quickly (in polynomial time).
• Think of NP as a set of problems where, if you're handed a solution, you can check its correctness relatively easily and quickly.

The SAT problem (Satisfiability)

The SAT problem is a question about **logic** and decision-making.
Imagine you have a series of statements, each involving a yes/no decision, like "*it's raining*" or "*the light is on*".
These statements can be combined using logical operators like AND, OR, and NOT.
For instance, "it's raining AND the light is on."

The **SAT problem** asks: **Is there a way to assign truth values (TRUE or FALSE) to each statement so that the entire combination becomes TRUE**?

The SAT problem

Is there a way to assign truth values (TRUE or FALSE) to each statement so that the entire combination becomes TRUE?

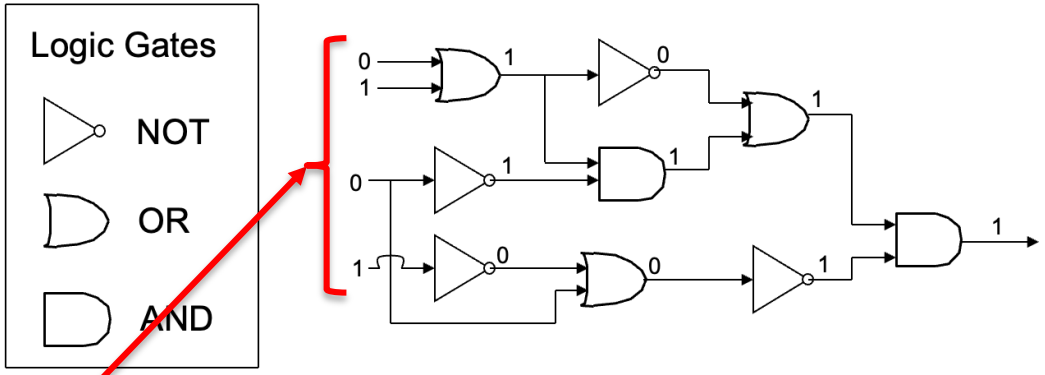
Example

Let's say you have statements *A* and *B* and a logical formula like

(*A* OR *B*) AND (NOT *A* OR *B*)

The SAT problem is asking whether we can assign TRUE or FALSE to *A* and *B* in such a way that the whole formula is TRUE. In this case, if we set *A*=FALSE and *B*=TRUE, the formula becomes TRUE, so this is a '**satisfiable**' case. Now imagine you have 1000 variables A1, …, A1000 : not so easy to find the combination !

The SAT problem : circuit



Take a Boolean circuit with a single output node and ask whether there is an assignment of values to the circuit's inputs so that the output is "1"

SAT and NP

- The SAT problem is a classic example of an NP problem.

Why? Because if I give you a specific assignment of TRUE/FALSE values for *A* and *B*, you can **quickly check** whether this assignment makes the whole formula TRUE.

However, finding that assignment in the first place is not necessarily easy or quick. In fact, as the number of statements increases, the time it takes to **find a solution** (if one exists) can grow **exponentially**.

SAT and NP-Completeness

SAT is not just in NP; it's one of the special problems known as **NP-complete**.
These are the toughest problems in NP. They are like the "bosses" of NP problems.

A problem is NP-complete if **every problem in NP can be transformed into it** in polynomial time.

What makes SAT so interesting is that if you can find a polynomial-time algorithm to solve SAT, you would effectively solve all problems in NP quickly.

Transformation of problems

Transformation: In computational theory, to transform one problem into another means to take an instance of the first problem and convert it into an instance of the second problem.l

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

- It is in NP, meaning we can verify a given solution in polynomial time.
- Every other problem in NP can be transformed into it in polynomial time.

Therefore, if we can solve an NP-complete problem in polynomial time, we can solve all problems in NP in polynomial time, effectively proving *P=NP*.

Transformation of problems

Transformation: In computational theory, to transform one problem into another means to take an instance of the first problem and convert it into an instance of the second problem.l

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

- It is in NP, meaning we can verify a given solution in polynomial time.
- Every other problem in NP can be transformed into it in polynomial time.

Therefore, if we can solve an NP-complete problem in polynomial time, we can solve all problems in NP in polynomial time, effectively proving *P=NP*.

P vs NP

The question of whether *P* is equal to *NP* remains one of the most important open problems in theoretical computer science and mathematics.
It's known as the ***P* vs *NP* problem.**
The problem asks whether every problem whose solution can be quickly **verified** (in polynomial time, hence *NP*) can also be quickly **solved** (in polynomial time, hence *P*).

Despite many efforts by mathematicians and computer scientists, no one has been able to prove either *P*=*NP* or *P* !=*NP*.

P vs NP

This problem is not just theoretical; it has practical implications in various fields including **cryptography**, algorithm design, artificial intelligence, and more.
If *P* were equal to *NP*, it would mean that numerous problems considered computationally difficult would have efficiently computable solutions.

Millennium Prize: The Clay Mathematics Institute has listed the *P* vs *NP* problem as one of the seven "Millennium Prize Problems" and has offered a prize of *one million dollars* for a correct solution.

P vs NP

If it is easy to check that a solution to a problem is correct, is it also easy to solve the problem? This is the essence of the P vs NP question. Typical of the NP problems is that of the Hamiltonian Path Problem: given N cities to visit, how can one do this without visiting a city twice? If you give me a solution, I can easily check that it is correct. But I cannot so easily find a solution.

https://www.claymath.org/millennium-problems

P vs NP

This problem is not just theoretical; it has practical implications in various fields including **cryptography**, algorithm design, artificial intelligence, and more.
If *P* were equal to *NP*, it would mean that numerous problems considered computationally difficult would have efficiently computable solutions.
The general *consensus* in the computer science community is that *P likely* does not equal *NP* (*P* != *NP*).

Millennium Prize: The Clay Mathematics Institute has listed the *P* vs *NP* problem as one of the seven "Millennium Prize Problems" and has offered a prize of *one million dollars* for a correct solution. https://www.claymath.org/millennium-problems

P vs NP

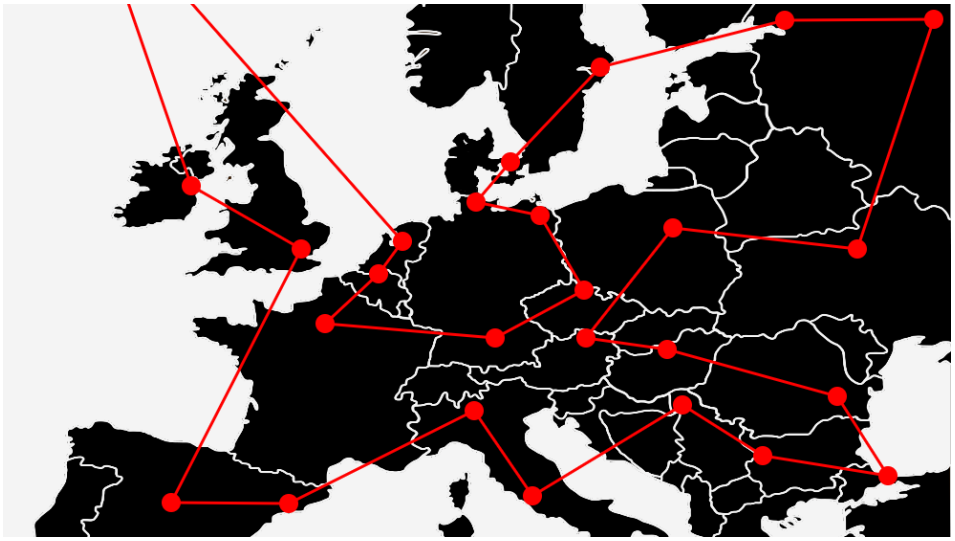
If it is easy to check that a solution to a problem is correct, is it also easy to solve the problem? This is the essence of the P vs NP question. Typical of the NP problems is that of the Hamiltonian Path Problem: given N cities to visit, how can one do this without visiting a city twice? If you give me a solution, I can easily check that it is correct. But I cannot so easily find a solution.

NP-complete

It is also important to know these problems, as they occur frequently in real applications and tackling them in a **brute force** fashion may be disastrous:

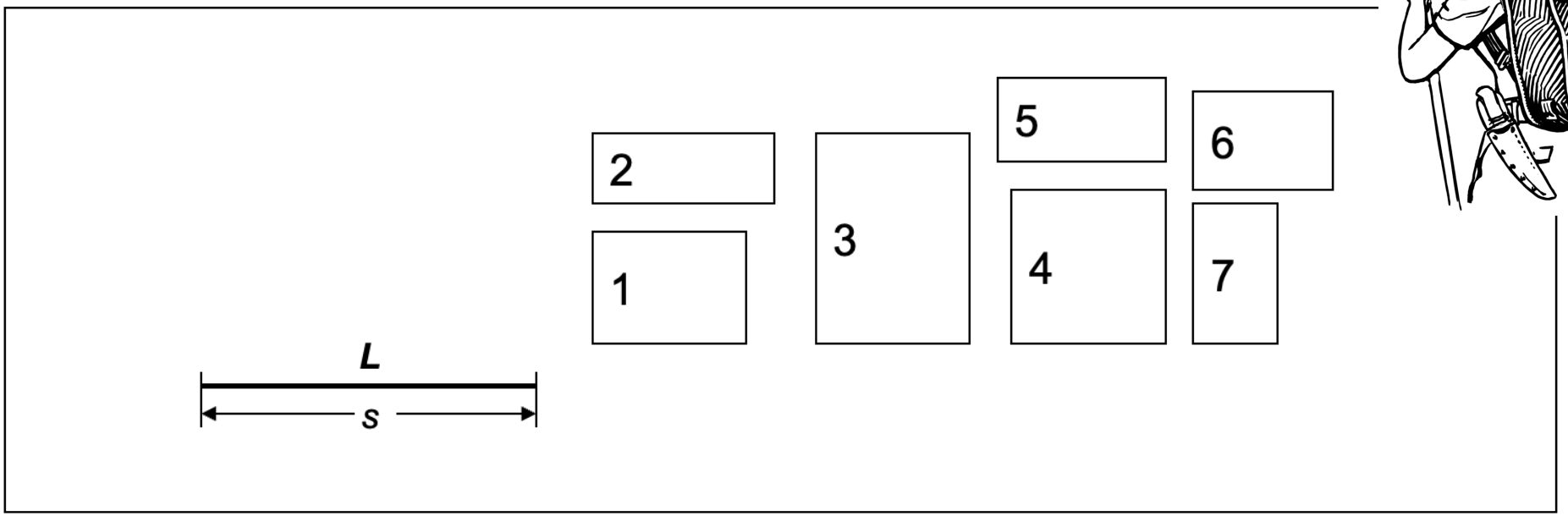
- SAT problem
- Traveling Salesman problem
- Knapsack Problem
- Longest Path
- Graph Clique

Traveling Salesman Problem (TSP)



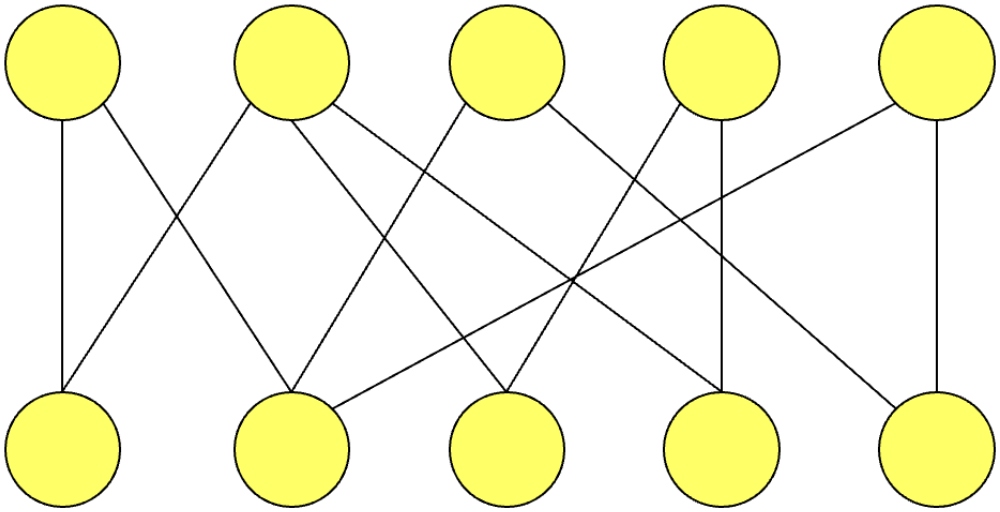
For each two cities, an integer cost is given to travel from one of the two cities to the other. The salesperson wants to make a minimum cost circuit visiting each city exactly once

Knapsack

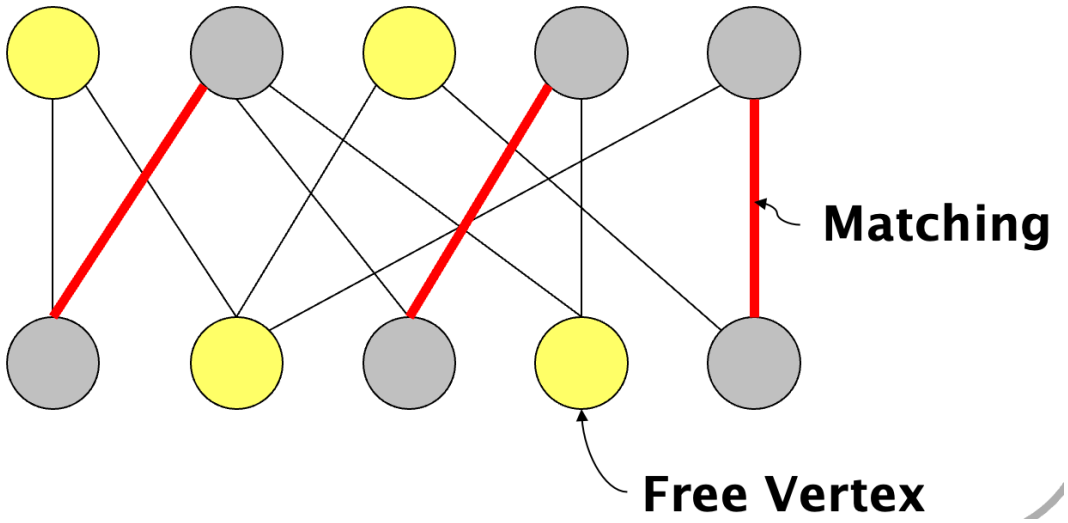


Given s and w can we translate a subset of rectangles to have their bottom edges on L so that the total area of the rectangles touching L is at least w?

Unweighted bipartite matching



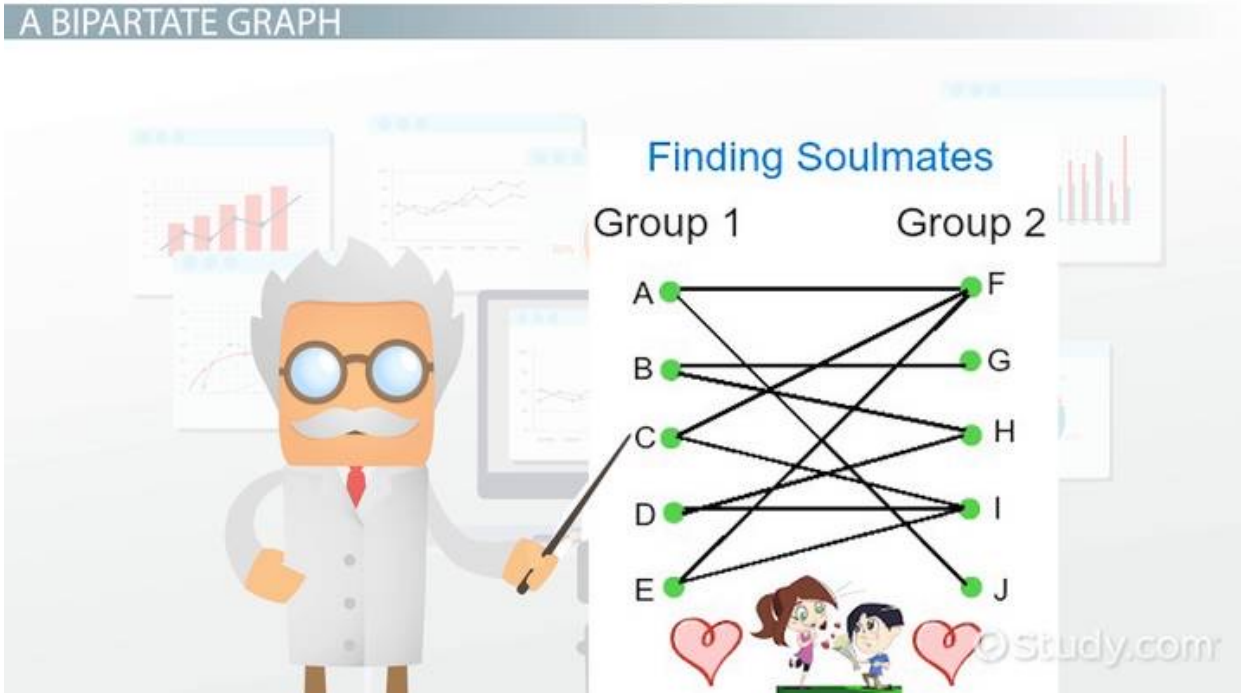
Unweighted bipartite matching



Maximum Matching: matching with the largest number of edges

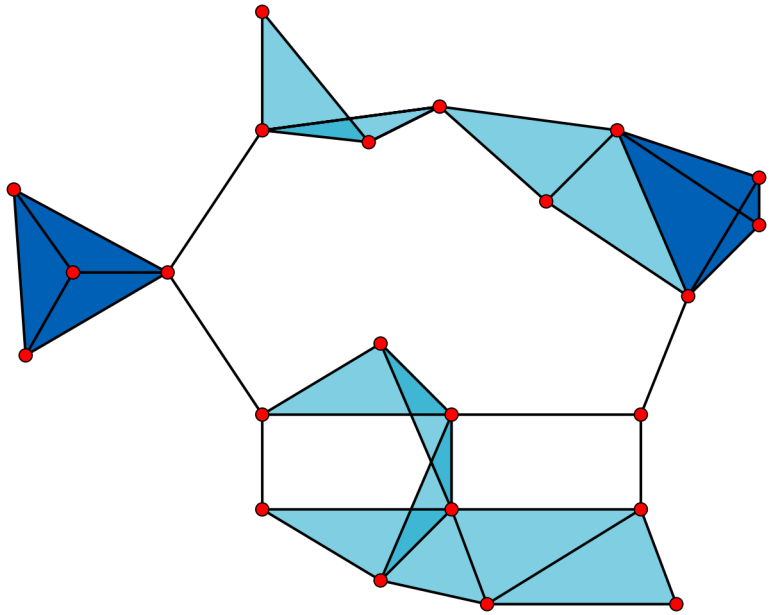
Note: the maximum matching is not unique

Unweighted bipartite matching



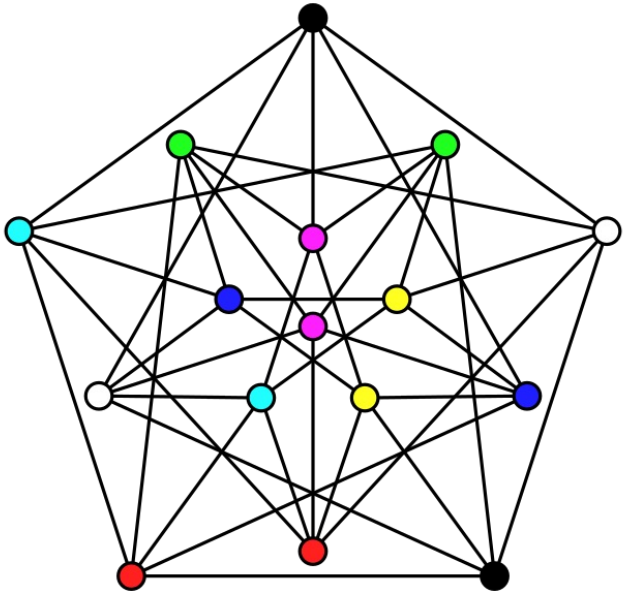
maximize the number of couples!

Another NP-complete problem: finding cliques in a graph



see next lesson

Back to graph coloring



Graph Coloring and NP

In NP: The graph coloring problem is in NP.

Because if someone gives you a solution to the problem (i.e., a coloring of the graph), *you can quickly verify whether it's correct*. You just need to check two things:

1. Every vertex has a color.
2. No two adjacent vertices have the same color.

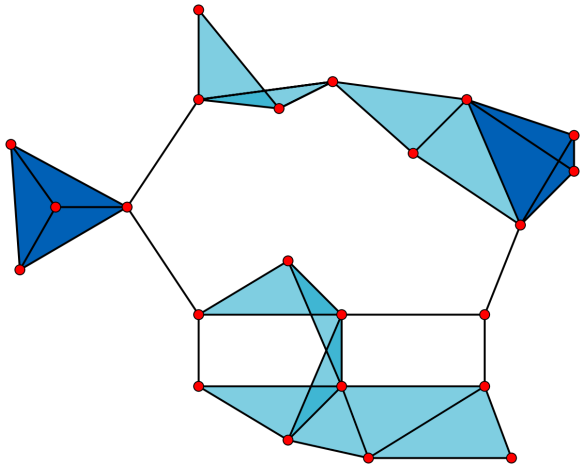
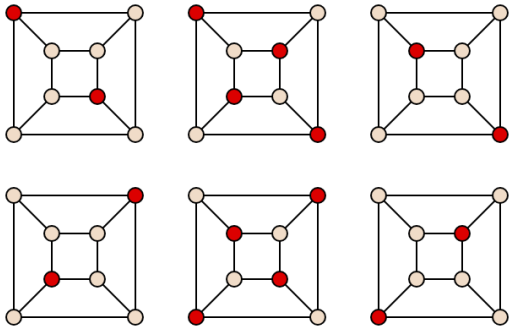
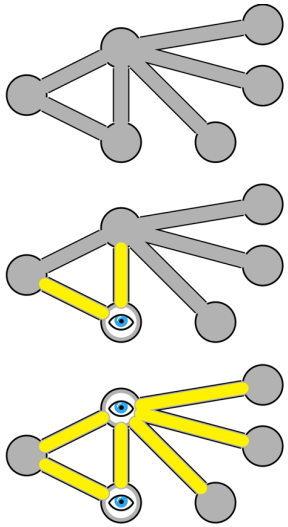
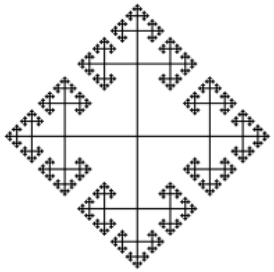
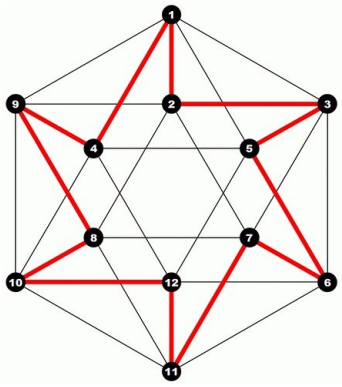
These checks can be done in polynomial time relative to the number of vertices and edges in the graph.

The decision version asks a yes/no question:

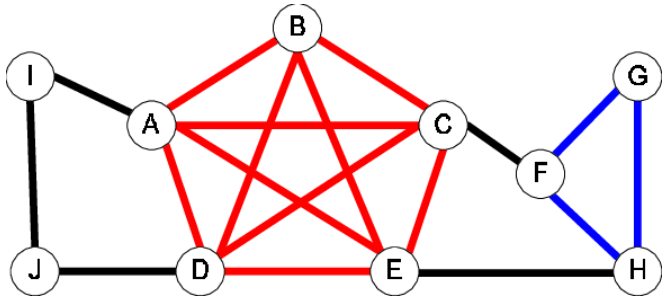
"**Is it possible to color the graph with k colors**?"

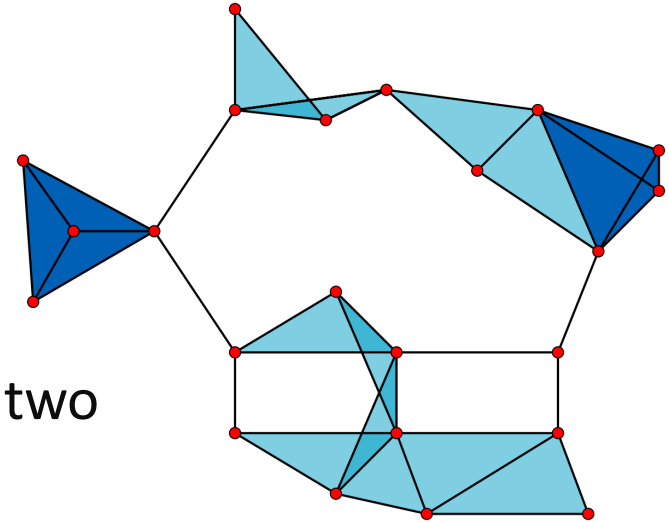
This problem is NP-complete !

Part 2
More interesting graph topics

Cliques



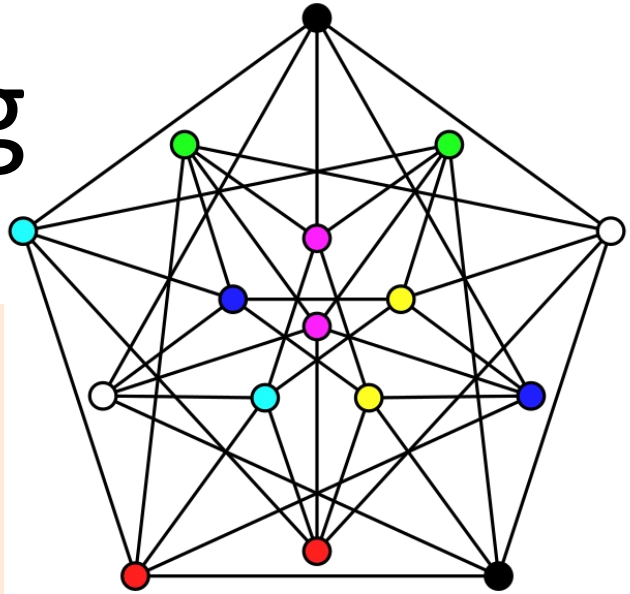


A clique in a graph is a subset of vertices where every two distinct vertices are adjacent.

It's like a group of friends where everyone knows each other.

Cliques represent complete subgraphs within a larger graph.
Finding Cliques: Finding the largest clique (**Maximum Clique Problem**) is computationally challenging (NP-Hard).
Even though it's a simple concept, finding the largest clique can be extremely difficult for large graphs.

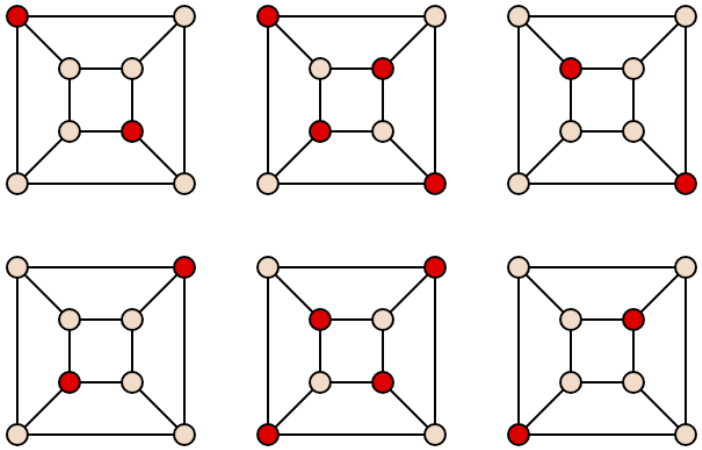
Cliques in graph coloring



The size of the largest clique is a lower bound for the graph **chromatic number**

== the minimum number of colors needed to color a graph is **at least the size of its largest clique**.

Independent Sets



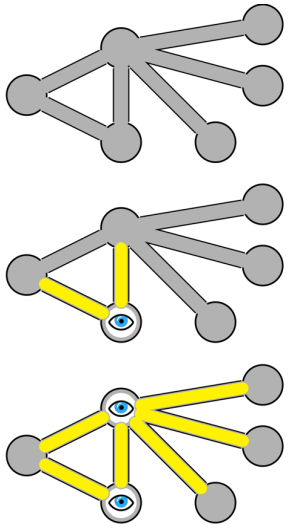
An independent set in a graph is a set of vertices with no edges connecting them.

It's like a set of people where no one knows each other.

Applications: e.g. scheduling problems, where tasks need to be performed without conflict. Helps in understanding which tasks can be executed simultaneously.

Maximum Independent Set: Finding the largest independent set is **NP-Hard**. Similar to cliques, determining the largest independent set is computationally challenging.

Vertex covers



A vertex cover of a graph is a set of vertices that includes at least one endpoint of every edge of the graph.

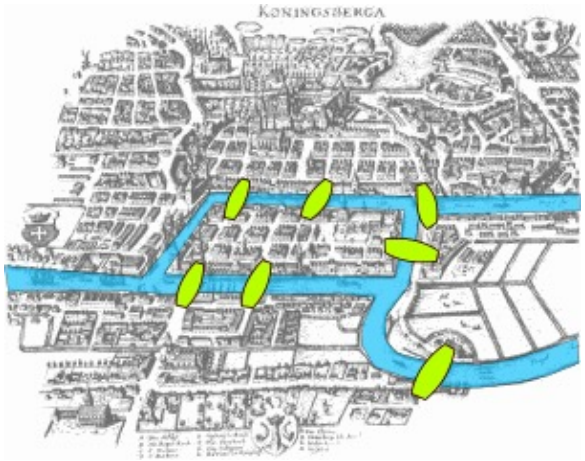
If you pick a vertex from each edge, you'll have a vertex cover.

Application: Network security (placing security cameras at intersections). Ensures observation of every pathway (edge) in the network.

The problem of finding a minimum vertex cover is a classical optimization problem. **It is NP-hard.**

See https://visualgo.net/en/mvc

Eulerian circuits



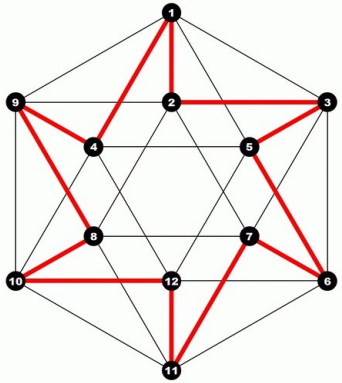
Back to Euler !

An Eulerian circuit in a graph is a path that uses each edge exactly once and returns to the starting vertex.

An Eulerian circuit allows you to traverse the graph without retracing any edge.

Exists if and only if the graph is connected and every vertex has an even degree.

Hamiltonian circuits



A Hamiltonian circuit in a graph is a path that visits each vertex exactly once and returns to the starting vertex.

Unlike Eulerian circuits, it's about visiting every **vertex** once.

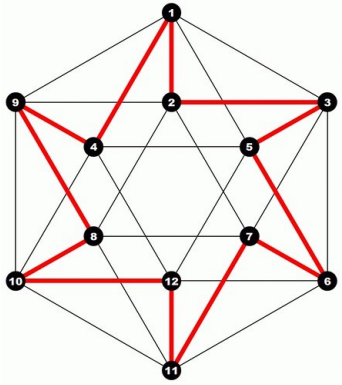
Applications: Traveling Salesman Problem, where the shortest possible route visiting each city once and returning to the origin is sought.
Central problem in logistics and travel planning.

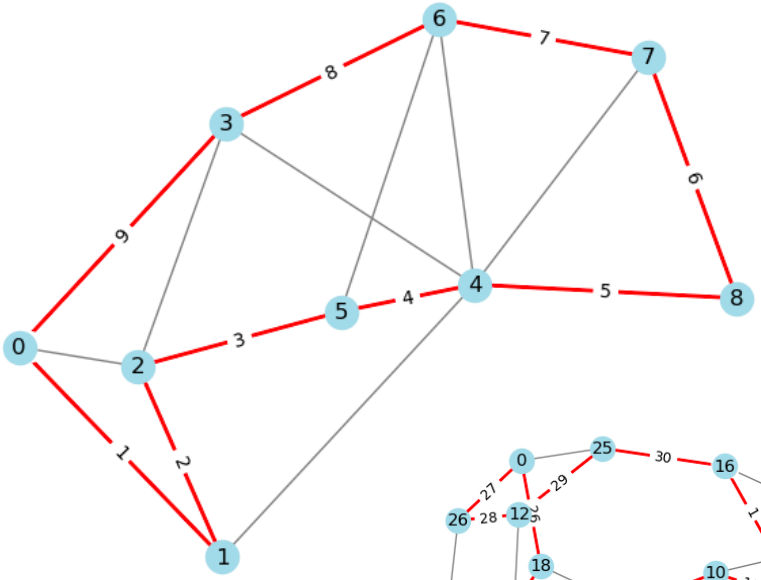
Finding Hamiltonian Circuit: Determining whether such a circuit exists is **NP-Complete.**

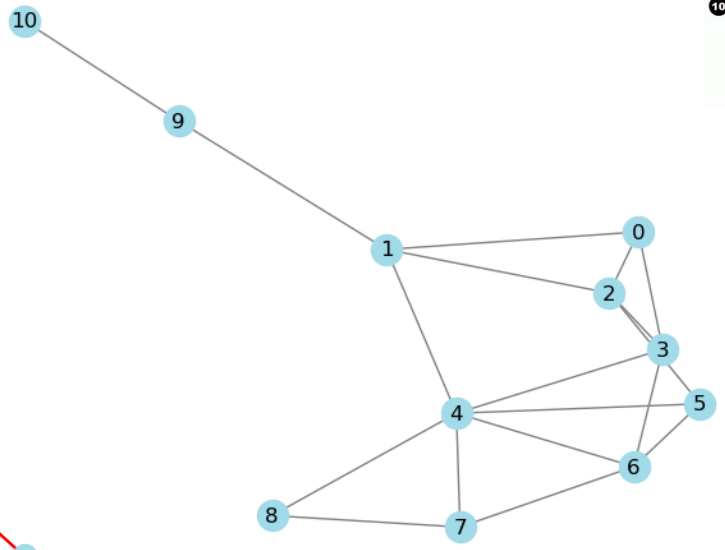
Finding the optimal Hamiltonian circuit is extremely difficult for large graphs, reflecting its computational complexity.

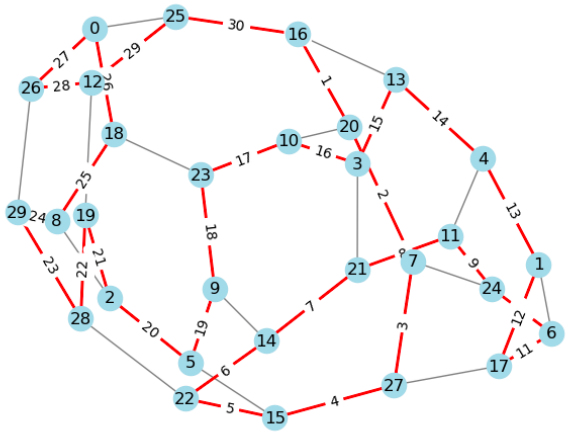
Hamiltonian circuits

Examples



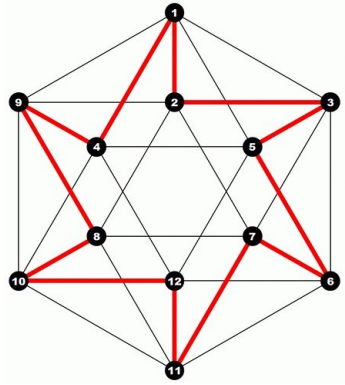


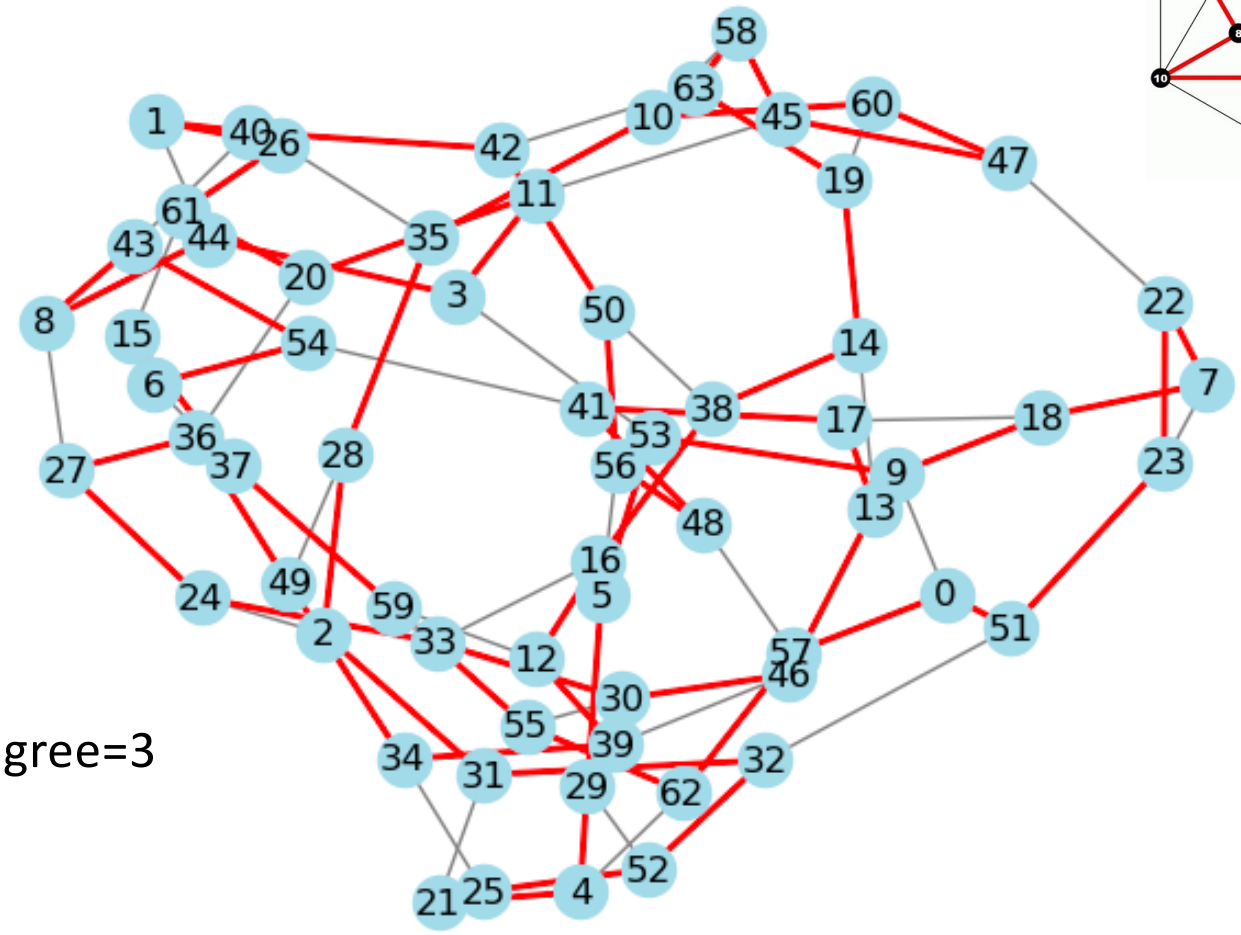




Hamiltonian circuits

Examples





64 nodes, degree=3