NetworkX: Network Analysis Subgraph Isomorphism

Stepan Kuznetsov

Discrete Math Bridging Course, HSE University

Social Network Analysis

- The study of social structures using graph theory is called *social network analysis* (SNA).

Social Network Analysis

- The study of social structures using graph theory is called *social network analysis* (SNA).
- Thus, SNA is an area on the border of discrete maths and sociology.

Social Network Analysis

- The study of social structures using graph theory is called *social network analysis* (SNA).
- Thus, SNA is an area on the border of discrete maths and sociology.
- Vertices in social network graphs represent *actors:* people, social entities etc.

Social Network Analysis

- The study of social structures using graph theory is called *social network analysis* (SNA).
- Thus, SNA is an area on the border of discrete maths and sociology.
- Vertices in social network graphs represent *actors:* people, social entities etc.
- Edges (also called *ties* or *links*) represent various *relations* between actors.

Social Network Analysis

- The study of social structures using graph theory is called *social network analysis* (SNA).
- Thus, SNA is an area on the border of discrete maths and sociology.
- Vertices in social network graphs represent *actors:* people, social entities etc.
- Edges (also called *ties* or *links*) represent various *relations* between actors.
- The standard example is the friendship relation in social networks.

Parameters of Social Network Graphs

- Graph parameters of social network graphs are important for sociologists studying these networks.

Parameters of Social Network Graphs

- Graph parameters of social network graphs are important for sociologists studying these networks.
- We are going to get acquainted with specialized software for calculating them.

Parameters of Social Network Graphs

- Notice how some parameters of the graph behave specifically in the social network case (if compared to a random graph, for example).

Parameters of Social Network Graphs

- Notice how some parameters of the graph behave specifically in the social network case (if compared to a random graph, for example).
- We shall see that the so called *clustering coefficients* tend to be quite high.

Parameters of Social Network Graphs

- Notice how some parameters of the graph behave specifically in the social network case (if compared to a random graph, for example).
- We shall see that the so called *clustering coefficients* tend to be quite high.
- This reflects the fact that friends of one person are much more likely to be friends also.

Parameters of Social Network Graphs

- On the other hand, being highly clusterized, the social network happens to be tightly connected.

Parameters of Social Network Graphs

- On the other hand, being highly clusterized, the social network happens to be tightly connected.
- The well-known theory of *six degrees of separation* (“six handshakes”) claims that any two people in the world are no more than six social connections from each other.

Parameters of Social Network Graphs

- On the other hand, being highly clusterized, the social network happens to be tightly connected.
- The well-known theory of *six degrees of separation* ("six handshakes") claims that any two people in the world are no more than six social connections from each other.
- In graph-theoretic terms, this means that the **diameter** of the social connections graph should be ≤ 6.

Dataset

- In our examples, we are going to use the SNAP dataset.

Dataset

- In our examples, we are going to use the SNAP dataset.
- SNAP = Stanford Network Analysis Project.

Dataset

- In our examples, we are going to use the SNAP dataset.
- SNAP = Stanford Network Analysis Project.
- The dataset we use includes friendship relations between friends of given 10 Facebook users (so-called *ego networks*).

Dataset

- In our examples, we are going to use the SNAP dataset.
- SNAP = Stanford Network Analysis Project.
- The dataset we use includes friendship relations between friends of given 10 Facebook users (so-called *ego networks*).
- This makes the dataset relatively small.

Dataset

- In our examples, we are going to use the SNAP dataset.
- SNAP = Stanford Network Analysis Project.
- The dataset we use includes friendship relations between friends of given 10 Facebook users (so-called *ego networks*).
- This makes the dataset relatively small.
- All data is of course anonymized.

NetworkX

- NetworkX is a Python library for graph analysis and visualization.

NetworkX

- NetworkX is a Python library for graph analysis and visualization.
- Free software, released under BSD-new license.

NetworkX

- NetworkX is a Python library for graph analysis and visualization.
- Free software, released under BSD-new license.
- Capable of handling big graphs (real-world datasets): 10M nodes / 100M edges and more.

NetworkX

- NetworkX is a Python library for graph analysis and visualization.
- Free software, released under BSD-new license.
- Capable of handling big graphs (real-world datasets): 10M nodes / 100M edges and more.
- Highly portable and scalable.

Getting NetworkX

- NetworkX, along with libraries necessary for visualization, can be installed with `pip`:

```
pip install networkx
pip install matplotlib
pip install scipy
```

Getting NetworkX

- NetworkX, along with libraries necessary for visualization, can be installed with `pip`:

```
pip install networkx
pip install matplotlib
pip install scipy
```

- NetworkX is then imported:

```
import networkx as nx
```

Getting NetworkX

- NetworkX, along with libraries necessary for visualization, can be installed with `pip`:

```
pip install networkx
pip install matplotlib
pip install scipy
```

- NetworkX is then imported:

```
import networkx as nx
```

- We've renamed `networkx` to `nx` for convenience.

Defining a Graph: Manual

- In NetworkX, one can define a graph manually, by adding edges one by one.

```
mygraph = nx.Graph()

mygraph.add_edge('A','B')
mygraph.add_edge('B','C')
mygraph.add_edge('C','A')
mygraph.add_edge('B','D')
```

Defining a Graph: Manual

- In NetworkX, one can define a graph manually, by adding edges one by one.

```
mygraph = nx.Graph()

mygraph.add_edge('A','B')
mygraph.add_edge('B','C')
mygraph.add_edge('C','A')
mygraph.add_edge('B','D')
```

- Vertices can be of arbitrary type (strings, numbers, ...).

Other Types of Graphs

- NetworkX can also handle directed graphs, multigraphs etc.

Other Types of Graphs

- NetworkX can also handle directed graphs, multigraphs etc.
- For a directed graph, use `nx.DiGraph` instead of `nx.Graph`.

Other Types of Graphs

- NetworkX can also handle directed graphs, multigraphs etc.
- For a directed graph, use `nx.DiGraph` instead of `nx.Graph`.
- Graphs in NetworkX can also be *weighted*.

Other Types of Graphs

- NetworkX can also handle directed graphs, multigraphs etc.
- For a directed graph, use `nx.DiGraph` instead of `nx.Graph`.
- Graphs in NetworkX can also be *weighted*.
- In a weighted graph, each edge receives a number called its weight.

Other Types of Graphs

- NetworkX can also handle directed graphs, multigraphs etc.
- For a directed graph, use `nx.DiGraph` instead of `nx.Graph`.
- Graphs in NetworkX can also be *weighted*.
- In a weighted graph, each edge receives a number called its weight.
- E.g., time (or cost) of driving along a road.

Other Types of Graphs

- NetworkX can also handle directed graphs, multigraphs etc.
- For a directed graph, use `nx.DiGraph` instead of `nx.Graph`.
- Graphs in NetworkX can also be *weighted*.
- In a weighted graph, each edge receives a number called its weight.
- E.g., time (or cost) of driving along a road.
- Weight is added just as an optional parameter to `add_edge`:

```
mygraph.add_edge('A','B', weight=6)
```

Reading a Graph from File

- NetworkX is also capable of reading graphs from files (datasets).

Reading a Graph from File

- NetworkX is also capable of reading graphs from files (datasets).
- In our example, we use SNAP's Facebook dataset (10 ego networks combined).

Reading a Graph from File

- NetworkX is also capable of reading graphs from files (datasets).
- In our example, we use SNAP's Facebook dataset (10 ego networks combined).
- In the file `facebook_combined.txt` one finds the list of edges as pairs of numbers (vertices are numbered).

Reading a Graph from File

- NetworkX is also capable of reading graphs from files (datasets).
- In our example, we use SNAP's Facebook dataset (10 ego networks combined).
- In the file `facebook_combined.txt` one finds the list of edges as pairs of numbers (vertices are numbered).
- The data gets imported by the `nx.read_edgelist` method.

Visualizing Graphs

- Graphs are abstract objects, but they have nice geometric representations.

Visualizing Graphs

- Graphs are abstract objects, but they have nice geometric representations.
- In many cases, it is very helpful to **see** how the graph looks like.

Visualizing Graphs

- Graphs are abstract objects, but they have nice geometric representations.
- In many cases, it is very helpful to **see** how the graph looks like.
- Rendering an abstract graph to a picture is called *visualization*.

Visualizing Graphs

- Graphs are abstract objects, but they have nice geometric representations.
- In many cases, it is very helpful to **see** how the graph looks like.
- Rendering an abstract graph to a picture is called *visualization*.
- NetworkX is capable of visualizing graphs, both in 2D and 3D.

Visualization: Small Example

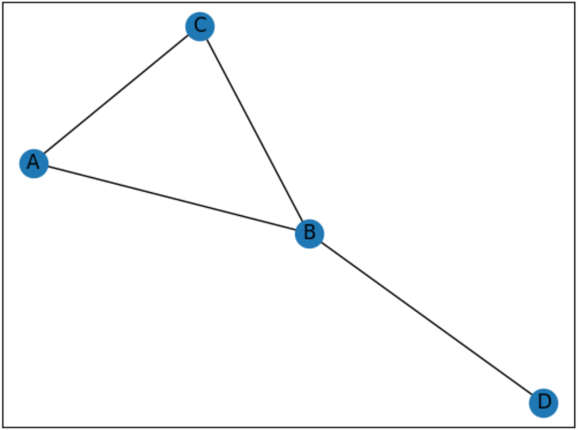
- NetworkX visualizes graphs via Matplotlib (a Python library for plotting).

Visualization: Small Example

- NetworkX visualizes graphs via Matplotlib (a Python library for plotting).
- The method is called `nx.draw_networkx`:

```
nx.draw_networkx(mygraph)
matplotlib.pyplot.savefig("mygraph.png")
```

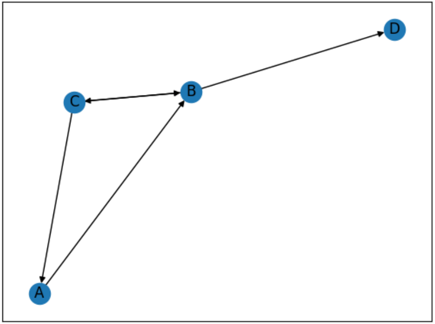
Visualization: Small Example



NetworkX output

Visualization: Small Example

This is how a directed graph is visualized. Two opposite edges between B and C are drawn as one edge with two arrows.



NetworkX output

Visualization of Real Data

- We remove labels, because there are too many vertices:

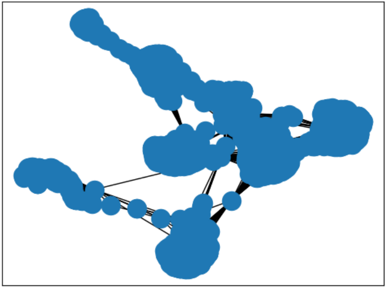
```
nx.draw_networkx(fb_gr, with_labels=False);
```

Visualization of Real Data

- We remove labels, because there are too many vertices:

```
nx.draw_networkx(fb_gr, with_labels=False);
```

- Visualization makes clustering visible:



NetworkX output

Some Graphs Tend to Cluster

- Social network graph: vertices = users, edges = friendship relations.

Some Graphs Tend to Cluster

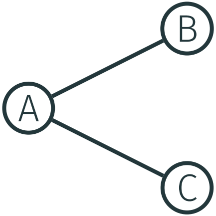
- Social network graph: vertices = users, edges = friendship relations.
- The probability, for two random vertices, to be connected, is generally quite low.

Some Graphs Tend to Cluster

- Social network graph: vertices = users, edges = friendship relations.
- The probability, for two random vertices, to be connected, is generally quite low.
- However, if Alex is a friend with Bob and Carl, a friendship relation between Bob and Carl becomes *much more probable.*

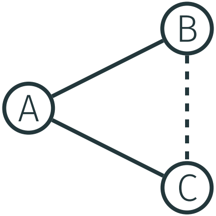
Some Graphs Tend to Cluster

- Social network graph: vertices = users, edges = friendship relations.
- The probability, for two random vertices, to be connected, is generally quite low.
- However, if Alex is a friend with Bob and Carl, a friendship relation between Bob and Carl becomes *much more probable*.



Some Graphs Tend to Cluster

- Social network graph: vertices = users, edges = friendship relations.
- The probability, for two random vertices, to be connected, is generally quite low.
- However, if Alex is a friend with Bob and Carl, a friendship relation between Bob and Carl becomes *much more probable.*

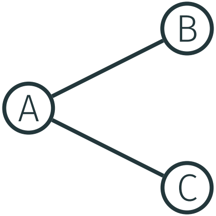


Global Clustering Coefficient

- One can measure clustering of a graph as a whole using the *global clustering coefficient*.

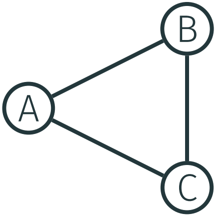
Global Clustering Coefficient

- One can measure clustering of a graph as a whole using the *global clustering coefficient*.
- A *triplet* is a pair of edges going from one vertex A:



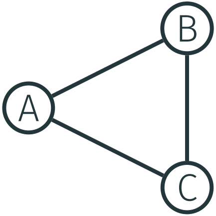
Global Clustering Coefficient

- A *triangle* is a triple of interconnected vertices:



Global Clustering Coefficient

- A *triangle* is a triple of interconnected vertices:



- $GCC(\mathcal{G}) = \dfrac{3 \cdot (\text{number of triangles})}{\text{number of triplets}}.$

Global Clustering Coefficient

- $GCC(\mathcal{G}) = \frac{3 \cdot (\text{number of triangles})}{\text{number of triplets}}.$

Global Clustering Coefficient

- $GCC(\mathcal{G}) = \frac{3 \cdot (\text{number of triangles})}{\text{number of triplets}}$.
- Why multiply by 3?

Global Clustering Coefficient

- $GCC(\mathcal{G}) = \frac{3 \cdot (\text{number of triangles})}{\text{number of triplets}}$.
- Why multiply by 3?
- Answer: each triangle includes three triplets.

Global Clustering Coefficient

- $GCC(\mathcal{G}) = \dfrac{3 \cdot (\text{number of triangles})}{\text{number of triplets}}.$
- Why multiply by 3?
- Answer: each triangle includes three triplets.
- Thus, the GCC is the *probability* for a random triplet A, B, C in $\mathcal{G}$ to be closed (that is, B and C connected).

Local Clustering Coefficient

- In the local clustering coefficient, we count only triplets with a given A as the central vertex.

Local Clustering Coefficient

- In the local clustering coefficient, we count only triplets with a given A as the central vertex.
- Suppose, the degree of A is k.

Local Clustering Coefficient

- In the local clustering coefficient, we count only triplets with a given A as the central vertex.
- Suppose, the degree of A is k.
- The total number of triplets with A as the central is $k \cdot (k-1)/2$.

Local Clustering Coefficient

- In the local clustering coefficient, we count only triplets with a given A as the central vertex.
- Suppose, the degree of A is k.
- The total number of triplets with A as the central is $k \cdot (k-1)/2$.
- $LCC(A)$ is the ratio

$$\frac{\text{number of pairs (B, C) which form a triangle with A}}{k \cdot (k-1)/2}$$

Local Clustering Coefficient

- In the local clustering coefficient, we count only triplets with a given A as the central vertex.
- Suppose, the degree of A is k.
- The total number of triplets with A as the central is $k \cdot (k-1)/2$.
- $LCC(A)$ is the ratio

$$\frac{2 \cdot (\text{number of pairs (B, C) which form a triangle with A}}{k \cdot (k-1)}$$

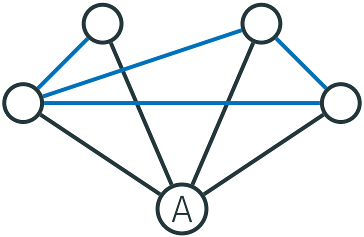
Local Clustering Coefficient

- In the local clustering coefficient, we count only triplets with a given A as the central vertex.
- Suppose, the degree of A is k.
- The total number of triplets with A as the central is $k \cdot (k-1)/2$.
- $LCC(A)$ is the ratio

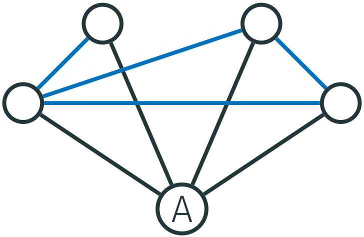
$$\frac{2 \cdot (\text{number of pairs (B, C) which form a triangle with A}}{k \cdot (k-1)}$$

- If A is an isolated vertex (degree = 0), then $LCC(A)$ is undefined (zero-by-zero division)

Local Clustering Coefficient



Local Clustering Coefficient



In this example, $LCC(A) = \frac{2 \cdot 4}{4 \cdot 3} = \frac{2}{3}$.

Graph Parameters in NetworkX

- NetworkX provides a convenient interface to algorithms computing graph parameters.

Graph Parameters in NetworkX

- NetworkX provides a convenient interface to algorithms computing graph parameters.
- Global parameters of the graph are just functions of it.

Graph Parameters in NetworkX

- NetworkX provides a convenient interface to algorithms computing graph parameters.
- Global parameters of the graph are just functions of it.
- For example, if we wish to calculate the *average clustering coefficient* (the average value of local clustering coefficients), we just run

```
av_clust = nx.average_clustering(fb_gr)
```

Clustering: Real vs Random

```
import networkx as nx
from random import random

G_fb = nx.read_edgelist("facebook_combined.txt")

av_clust_coeff =  nx.average_clustering(G_fb)
print ("acc = "+str(av_clust_coeff))

edges = G_fb.number_of_edges()
nodes = G_fb.number_of_nodes()
max_edges = nodes*(nodes-1)/2
edge_probab = edges / max_edges
G_rand = nx.Graph();
k = nodes-1
for i in range(0,k) :
    for j in range(0,i) :
        if (random() <= edge_probab) :
            G_rand.add_edge(i,j)

av_clust_coeff = nx.average_clustering(G_rand)
print("rgraph_acc = " + str(av_clust_coeff));
```

Clustering: Real vs Random

- This experiment yields the following results:

```
acc = 0.6055467186200876
rgraph_acc = 0.010822469487992627
```

Clustering: Real vs Random

- This experiment yields the following results:

```
acc = 0.6055467186200876
rgraph_acc = 0.010822469487992627
```

- This shows that (unsurprisingly) the social network tends to cluster more than the random graph (with the same probability of edge).

Clustering: Real vs Random

- This experiment yields the following results:

```
acc = 0.6055467186200876
rgraph_acc = 0.010822469487992627
```

- This shows that (unsurprisingly) the social network tends to cluster more than the random graph (with the same probability of edge).
- Thus, one has to be cautious with stochastic modelling of social graphs.

Graph Parameters in NetworkX

- Suppose we want to check "six handshakes."

Graph Parameters in NetworkX

- Suppose we want to check "six handshakes."
- That is, we have to calculate the diameter of our graph:

```
diam = nx.diameter(fb_gr)
```

Graph Parameters in NetworkX

- Suppose we want to check “six handshakes.”
- That is, we have to calculate the diameter of our graph:

```
diam = nx.diameter(fb_gr)
```

- The calculation takes quite long… and on our data it yields 8.

Graph Parameters in NetworkX

- Suppose we want to check “six handshakes.”
- That is, we have to calculate the diameter of our graph:

```
diam = nx.diameter(fb_gr)
```

- The calculation takes quite long… and on our data it yields 8.
- This is quite a good result, recalling that we have just a fusion of 10 ego nets, not the full Facebook graph.

Distances

- By definition, the *distance* between two vertices is the length of the shortest path connecting them.

Distances

- By definition, the *distance* between two vertices is the length of the shortest path connecting them.
- This can be computed by `nx.shortest_path_length`

Distances

- By definition, the *distance* between two vertices is the length of the shortest path connecting them.
- This can be computed by `nx.shortest_path_length`
- In directed graphs, the path should also be directed—thus, sometimes $d(a, b) \neq d(b, a)$.

Distances

- By definition, the *distance* between two vertices is the length of the shortest path connecting them.
- This can be computed by `nx.shortest_path_length`
- In directed graphs, the path should also be directed—thus, sometimes $d(a,b) \neq d(b,a)$.
- **Caveat!** If there is no path, NetworkX throws an exception.

Distances

- By definition, the *distance* between two vertices is the length of the shortest path connecting them.
- This can be computed by `nx.shortest_path_length`
- In directed graphs, the path should also be directed—thus, sometimes $d(a, b) \neq d(b, a)$.
- **Caveat!** If there is no path, NetworkX throws an exception.
- To be on the safe side, use `nx.has_path` before.

Preparing for HW 3

- More information is available in NetworkX documentation.

Preparing for HW 3

- More information is available in NetworkX documentation.
- Please consult it when accomplishing the programming task.

Preparing for HW 3

- More information is available in NetworkX documentation.
- Please consult it when accomplishing the programming task.
- Good luck!

Graph and Subgraph Isomorphism

- We are going to discuss algorithmic problems connected to isomorphism and subgraphs.

Graph and Subgraph Isomorphism

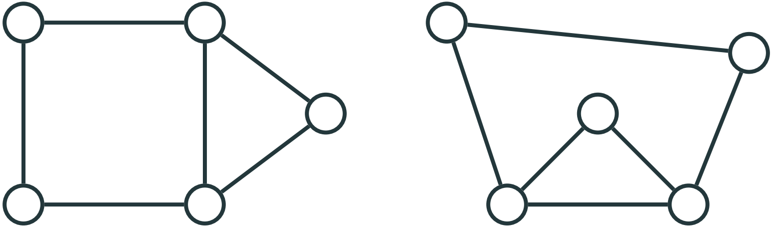
- We are going to discuss algorithmic problems connected to isomorphism and subgraphs.
- First, let us recall the notion of isomorphic graphs.

Graph Isomorphism

Sometimes graphs look different, but essentially are the same…

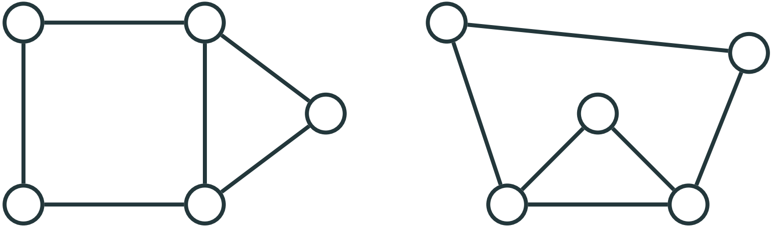
Graph Isomorphism

Sometimes graphs look different, but essentially are the same...



Graph Isomorphism

Sometimes graphs look different, but essentially are the same…



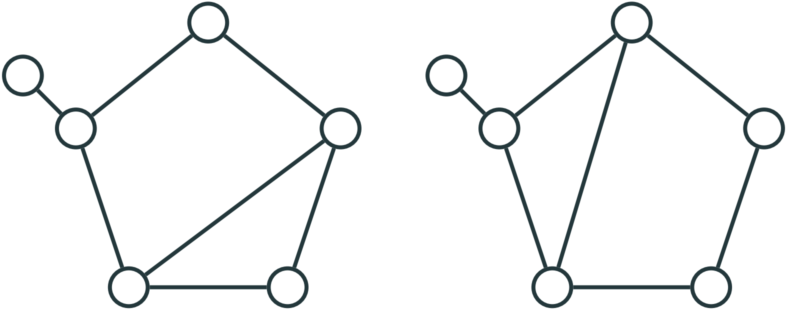
Here both graphs can be described as “a triangle and a quadrangle sharing a common edge.”

Graph Isomorphism

… and sometimes similarly looking graphs are different.

Graph Isomorphism

… and sometimes similarly looking graphs are different.



Graph Isomorphism

Isomorphic Graphs

Two graphs, $\mathcal{G}$ and $\mathcal{H}$, are *isomorphic*, if they have the same number n of vertices and vertices of each graph can be enumerated by numbers from 1 to n, so that vertices with numbers i and j are connected in $\mathcal{G}$ if and only if vertices with these numbers are connected in $\mathcal{H}$.

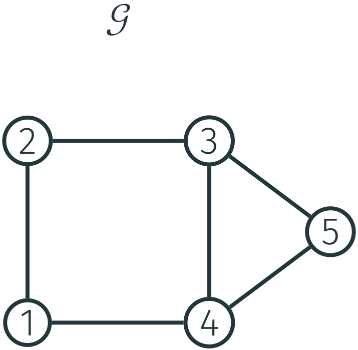
Graph Isomorphism

Isomorphic Graphs

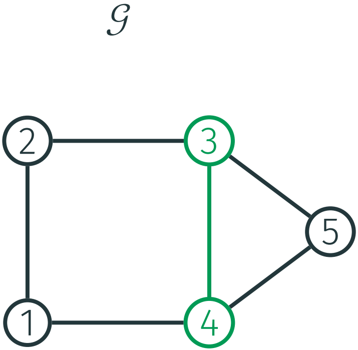
Two graphs, $\mathcal{G}$ and $\mathcal{H}$, are *isomorphic*, if they have the same number n of vertices and vertices of each graph can be enumerated by numbers from 1 to n, so that vertices with numbers i and j are connected in $\mathcal{G}$ if and only if vertices with these numbers are connected in $\mathcal{H}$.

Isomorphic graphs can be seen as *different representations of the same graph.*

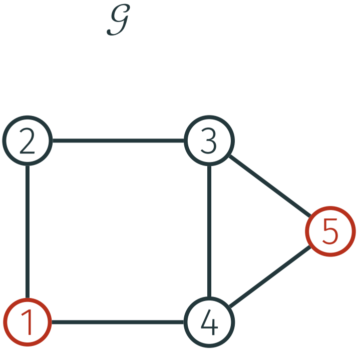
Isomorphic Graphs



Isomorphic Graphs



Isomorphic Graphs

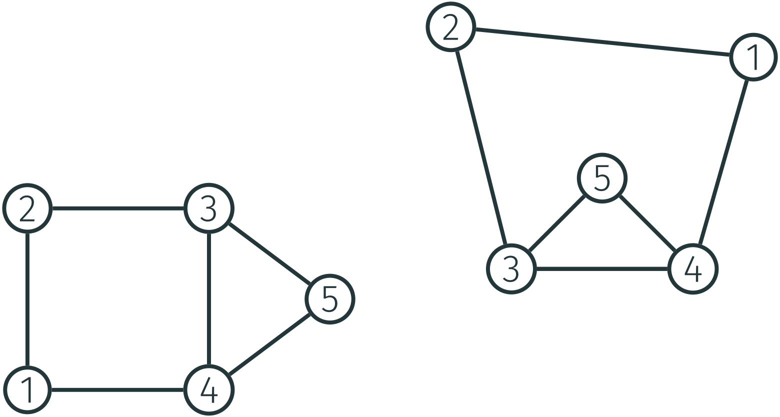


Isomorphism

The *isomorphism* itself is the correspondence between vertices with the same number.

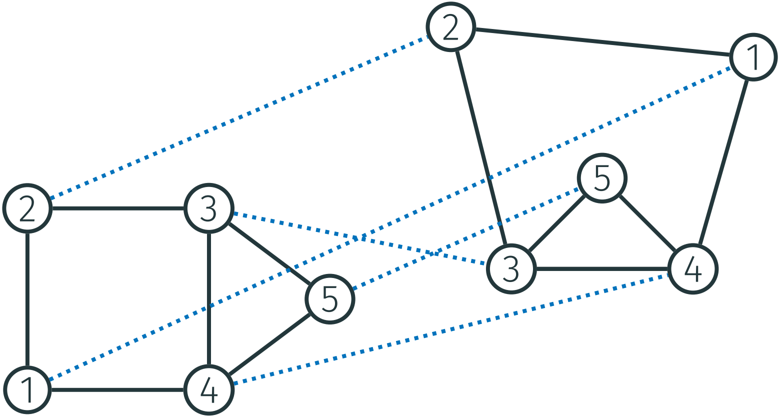
Isomorphism

The *isomorphism* itself is the correspondence between vertices with the same number.



Isomorphism

The *isomorphism* itself is the correspondence between vertices with the same number.



Graph Isomorphism Algorithmically

- What is the algorithmic complexity of checking whether two given graphs, $\mathcal{G}$ and $\mathcal{H}$, are isomorphic?

Graph Isomorphism Algorithmically

- What is the algorithmic complexity of checking whether two given graphs, $\mathcal{G}$ and $\mathcal{H}$, are isomorphic?
- First, this problem is obviously in NP: one can just non-deterministically guess the isomorphism.

Graph Isomorphism Algorithmically

- What is the algorithmic complexity of checking whether two given graphs, $\mathcal{G}$ and $\mathcal{H}$, are isomorphic?
- First, this problem is obviously in NP: one can just non-deterministically guess the isomorphism.
- However, graph isomorphism is a quite rare species of NP problem: we know neither that it is NP-complete, nor that it belongs to P.

Subgraphs

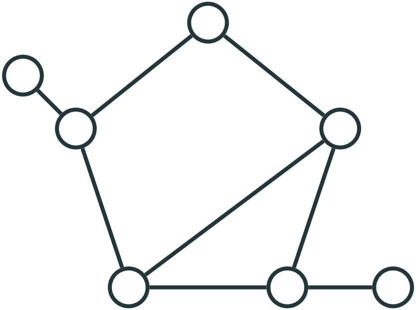
- A *subgraph* is a part of a graph which is obtained by taking a subset of vertices and a subset of edges.

Subgraphs

- A *subgraph* is a part of a graph which is obtained by taking a subset of vertices and a subset of edges.
- The vertex subset should cover the edge subset.

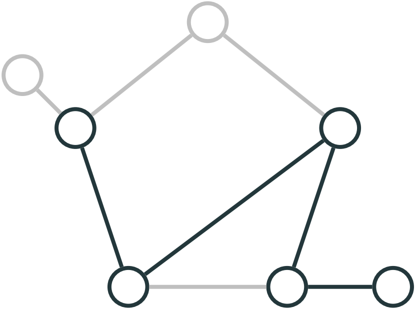
Subgraphs

- A *subgraph* is a part of a graph which is obtained by taking a subset of vertices and a subset of edges.
- The vertex subset should cover the edge subset.



Subgraphs

- A *subgraph* is a part of a graph which is obtained by taking a subset of vertices and a subset of edges.
- The vertex subset should cover the edge subset.



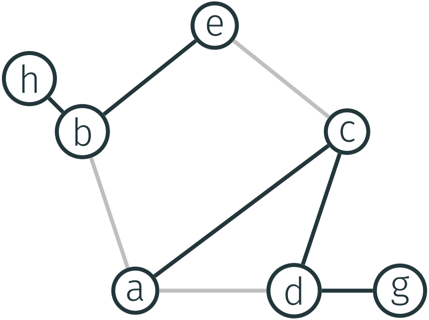
Subgraphs

- An *induced* subgraph includes all the edges of the original graph, whose endpoints are in the vertex subset.

Subgraphs

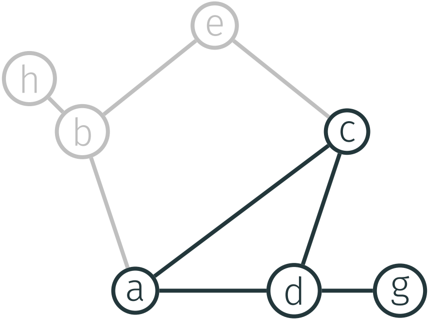
- An *induced* subgraph includes all the edges of the original graph, whose endpoints are in the vertex subset.
- A *spanning* subgraph includes all vertices of the original graph (but maybe not all edges).

Subgraphs



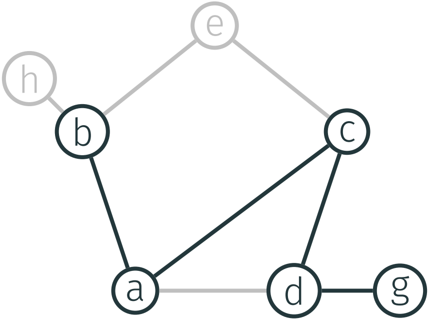
spanning

Subgraphs



induced

Subgraphs



neither

Subgraph Isomorphism Problem

- The (algorithmic) problem is as follows: given a “big” graph $\mathcal{H}$ and a “small” graph $\mathcal{G}_0$, determine whether there exists an induced subgraph in $\mathcal{H}$, which is isomorphic to $\mathcal{G}_0$.

Subgraph Isomorphism Problem

- The (algorithmic) problem is as follows: given a “big” graph $\mathcal{H}$ and a “small” graph $\mathcal{G}_0$, determine whether there exists an induced subgraph in $\mathcal{H}$, which is isomorphic to $\mathcal{G}_0$.
- There is also a variant of this problem without requiring the subgraph to be an induced one.

Subgraph Isomorphism Problem

- The subgraph isomorphism problem is a problem of **pattern matching / search,** but for structures rather than words.

Subgraph Isomorphism Problem

- The subgraph isomorphism problem is a problem of **pattern matching / search,** but for structures rather than words.
- One asks for existence (or to find) a given small pattern $\mathcal{G}_0$ in a huge structure $\mathcal{H}$.

Subgraph Isomorphism Problem

- The subgraph isomorphism problem is a problem of **pattern matching / search,** but for structures rather than words.
- One asks for existence (or to find) a given small pattern $\mathcal{G}_0$ in a huge structure $\mathcal{H}$.
- Applications: chem- / bioinformatics, graph mining (structure mining), etc

Subgraph Isomorphism Problem

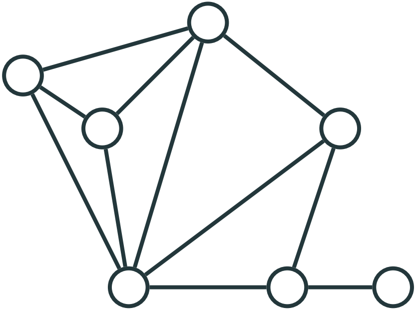
- The subgraph isomorphism problem is a problem of **pattern matching / search,** but for structures rather than words.
- One asks for existence (or to find) a given small pattern $\mathcal{G}_0$ in a huge structure $\mathcal{H}$.
- Applications: chem- / bioinformatics, graph mining (structure mining), etc
- The subgraph isomorphism problem is NP-complete (in both variants).

Special Subgraphs

- A *clique* is a complete subgraph.

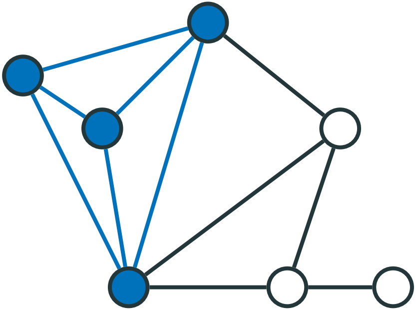
Special Subgraphs

- A *clique* is a complete subgraph.
- Example of a clique on 4 vertices:



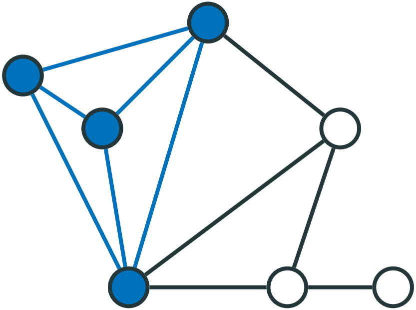
Special Subgraphs

- A *clique* is a complete subgraph.
- Example of a clique on 4 vertices:



Special Subgraphs

- A *clique* is a complete subgraph.
- Example of a clique on 4 vertices:



- Clique in social network graph = group of users who are friends with each other.

Independent Sets

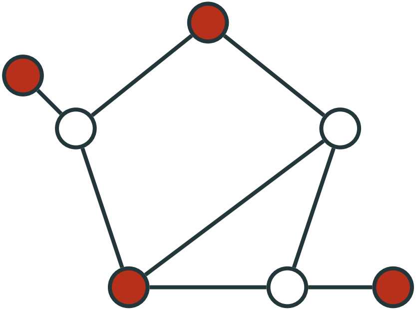
- An *independent set* is an empty induced subgraph.

Independent Sets

- An *independent set* is an empty induced subgraph.
- No pair of vertices from an independent set could be connected.

Independent Sets

- An *independent set* is an empty induced subgraph.
- No pair of vertices from an independent set could be connected.
- Example: independent set of 4 vertices.



NP-Completeness of INDSET

- INDSET, the problem of existence, in a given graph $\mathcal{G}$, an independent set of a given size k, is NP-complete.

NP-Completeness of INDSET

- INDSET, the problem of existence, in a given graph $\mathcal{G}$, an independent set of a given size k, is NP-complete.
  - Input: (G, k).

NP-Completeness of INDSET

- INDSET, the problem of existence, in a given graph $\mathcal{G}$, an independent set of a given size k, is NP-complete.
  - Input: (G, k).
- This follows from the following reduction: CNF-SAT $\leq_m^P$ INDSET.

NP-Completeness of INDSET

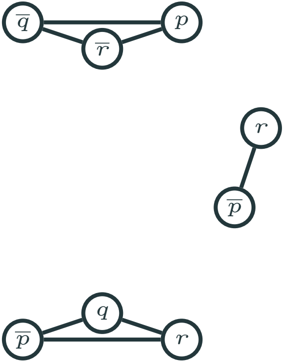
- INDSET, the problem of existence, in a given graph $\mathcal{G}$, an independent set of a given size k, is NP-complete.
  - Input: (G, k).
- This follows from the following reduction: CNF-SAT $\leq_m^P$ INDSET.
- Given a CNF A, we construct $(\mathcal{G}, k)$ such $\mathcal{G}$ has an independent set of k vertices if and only if A is satisfiable.

Reducing CNF-SAT to INDSET

$$A = (q \vee p) \wedge (\overline{p} \vee q \vee r) \wedge (\overline{q} \vee \overline{r} \vee p) \wedge (\overline{p} \vee r).$$

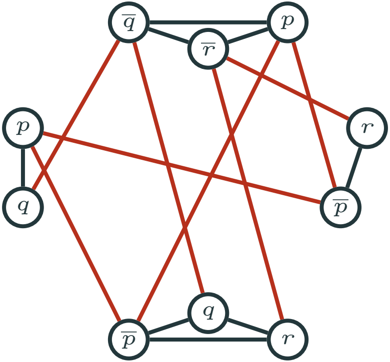
Reducing CNF-SAT to INDSET

$$A = (q \vee p) \wedge (\overline{p} \vee q \vee r) \wedge (\overline{q} \vee \overline{r} \vee p) \wedge (\overline{p} \vee r).$$



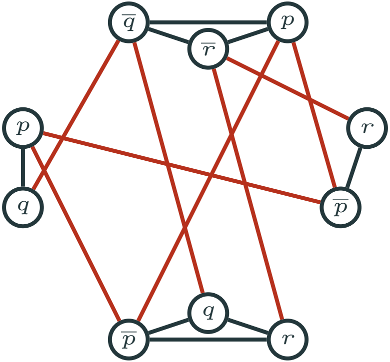
Reducing CNF-SAT to INDSET

$$A = (q \vee p) \wedge (\overline{p} \vee q \vee r) \wedge (\overline{q} \vee \overline{r} \vee p) \wedge (\overline{p} \vee r).$$



Reducing CNF-SAT to INDSET

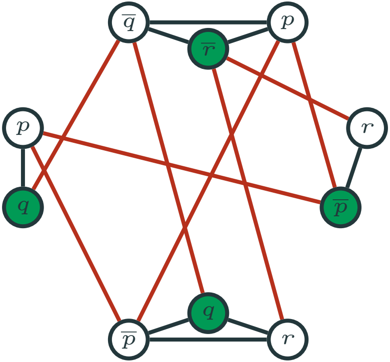
$$A = (q \vee p) \wedge (\overline{p} \vee q \vee r) \wedge (\overline{q} \vee \overline{r} \vee p) \wedge (\overline{p} \vee r).$$



$k = 4$

Reducing CNF-SAT to INDSET

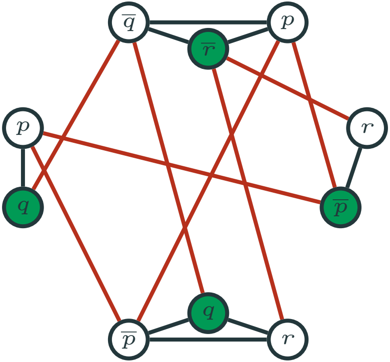
$$A = (q \vee p) \wedge (\overline{p} \vee q \vee r) \wedge (\overline{q} \vee \overline{r} \vee p) \wedge (\overline{p} \vee r).$$



$k = 4$

Reducing CNF-SAT to INDSET

$$A = (q \vee p) \wedge (\overline{p} \vee q \vee r) \wedge (\overline{q} \vee \overline{r} \vee p) \wedge (\overline{p} \vee r).$$



$k = 4$ $\qquad$ $q = 1, p = r = 0$

Hamiltonian Paths

- Another famous NP-complete problem is the existence of a Hamiltonian path, i.e., a path which visits each **vertex** exactly once.

Hamiltonian Paths

- Another famous NP-complete problem is the existence of a Hamiltonian path, i.e., a path which visits each **vertex** exactly once.
- This is also a subcase of subgraph isomorphism: whether $\mathcal{H}$ includes a subgraph (not an induced one), which is a chain of $|V|$ vertices.

Hamiltonian Paths

- Another famous NP-complete problem is the existence of a Hamiltonian path, i.e., a path which visits each **vertex** exactly once.
- This is also a subcase of subgraph isomorphism: whether $\mathcal{H}$ includes a subgraph (not an induced one), which is a chain of $|V|$ vertices.
- In this course, we omit the proof of NP-hardness for Hamiltonian path, but next week we'll sketch its applications to genomics.