Carleton University

Alina Shaikhet

Graphs

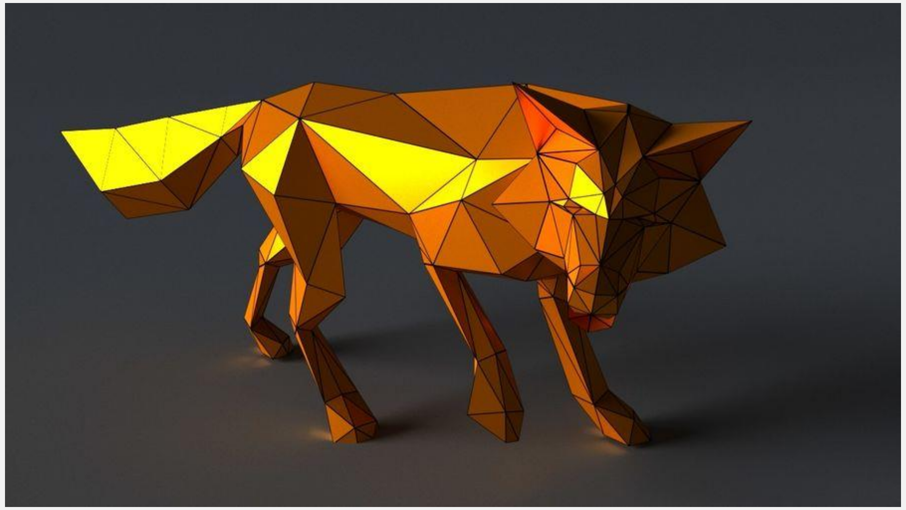
part 1

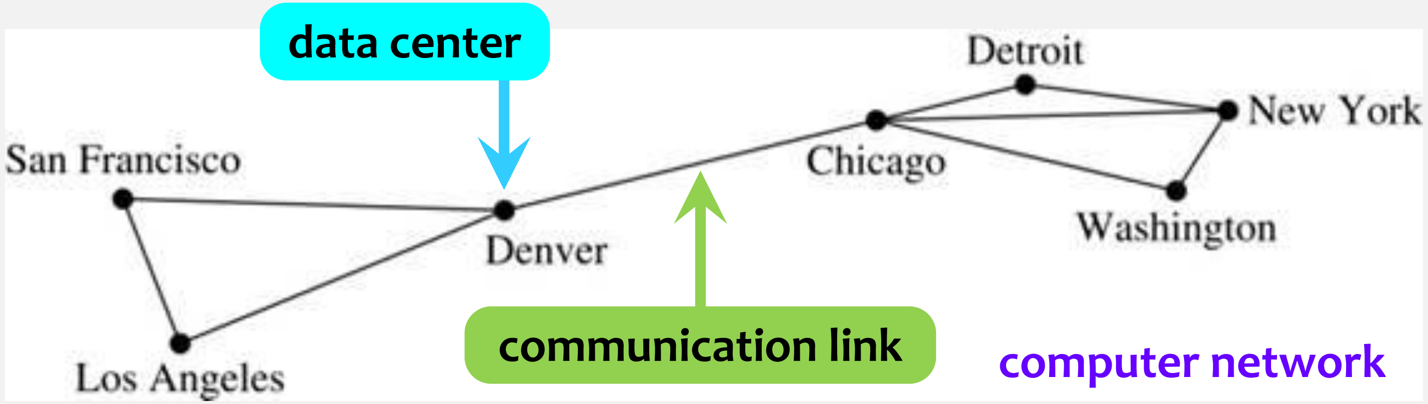
What is a graph?

A collection of “things” called **vertices** (or **nodes**) and some relationship between the things.

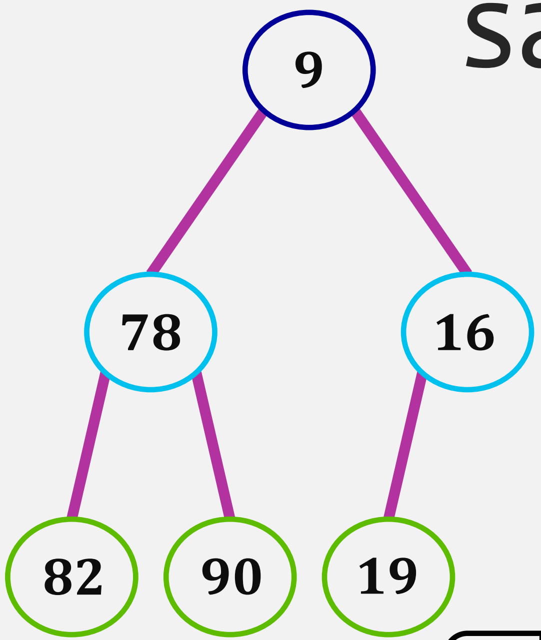
If two things are related then we have an **edge** that connects them in the graph.

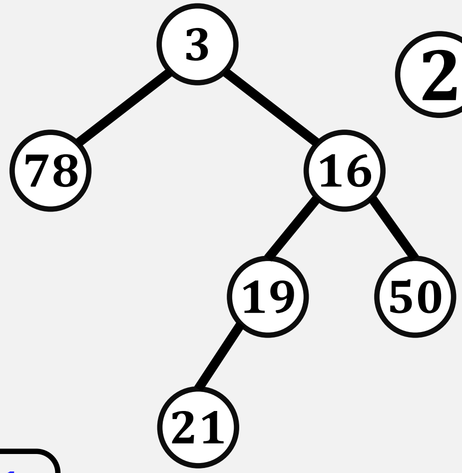
vertices	edges
people	friends
cities	highways
websites	links

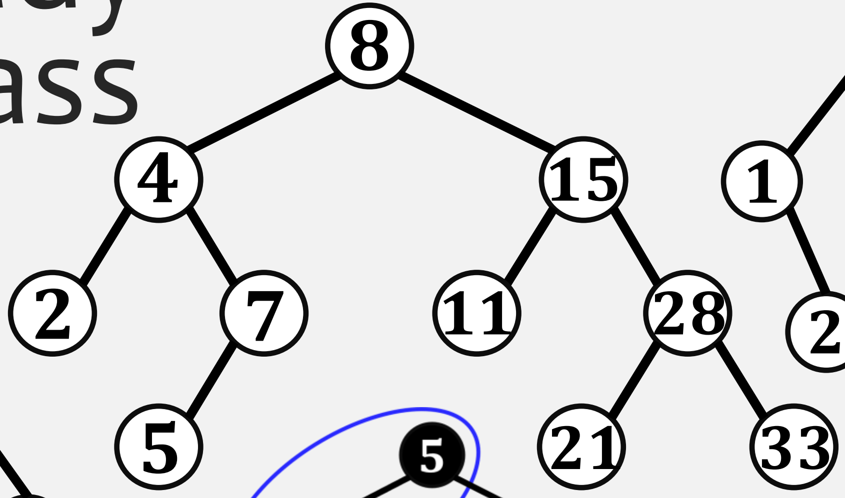


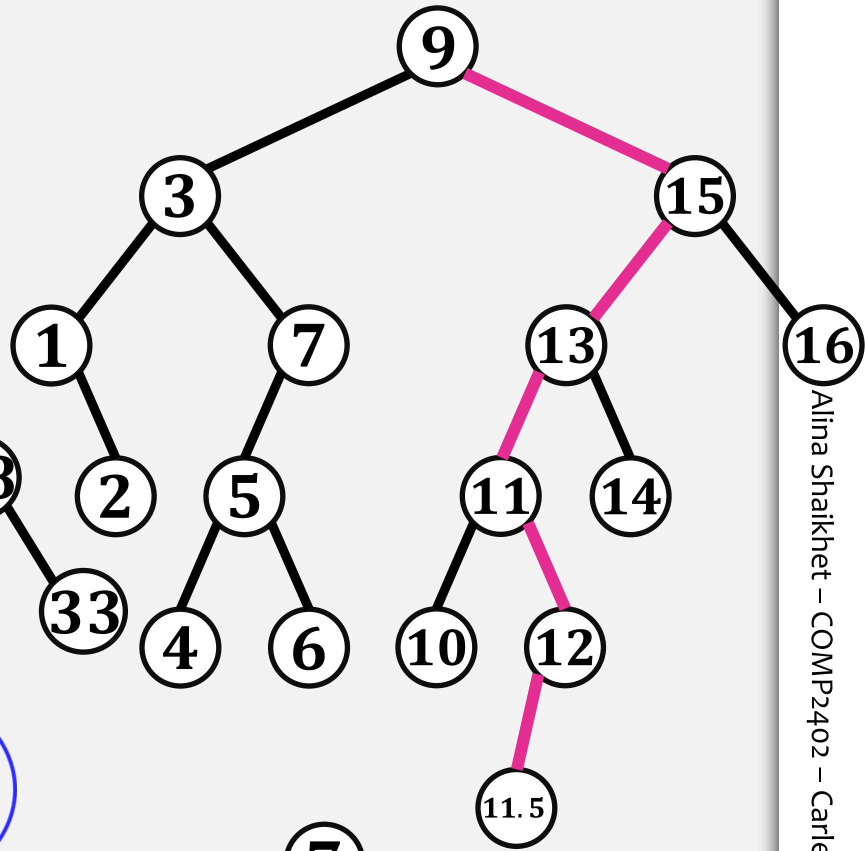


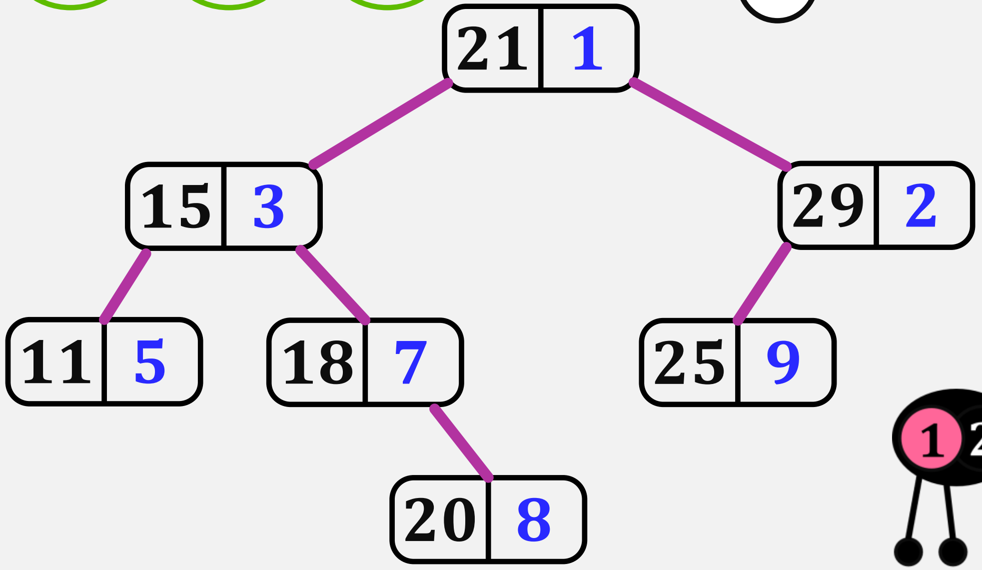
Graphs we already saw in class

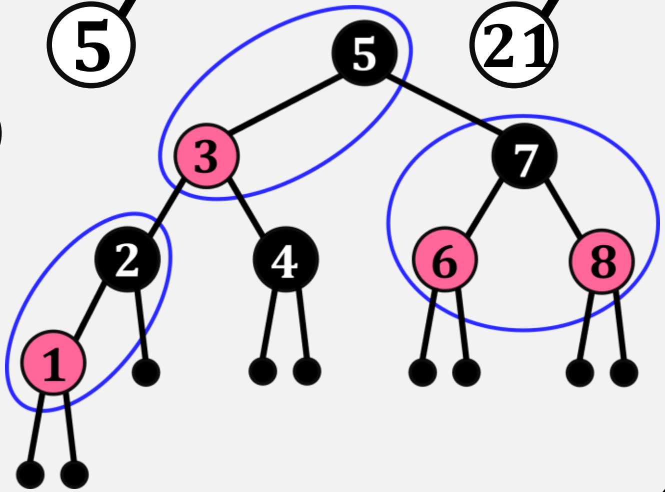


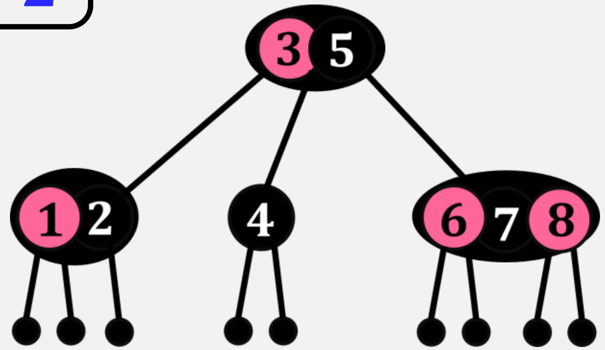


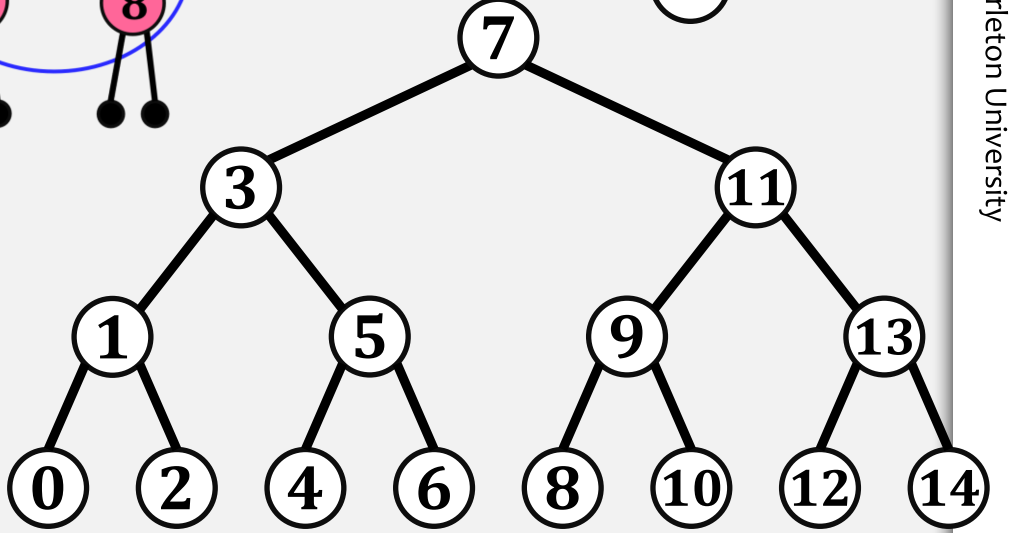












What problems can graphs solve?

Every relational system is a graph.

It is more difficult to find an area where graphs are not involved

- Geospatial intelligence (GIS, GPS navigation and roadmaps, traffic organization, …)
- Social networking (Instagram, Netflix recommendations, Amazon suggestions, …)
- Biology (Disease outbreak/transmission, competition of species in an ecological niche, …)
- Social Science (Information and decision-making. Who influences whom in an organization. Assignment of jobs to employees. Collaborations. …)
- Neurology (studying the brain, …)
- Computer Graphics, Animation, & Visualization
- Scheduling, Coloring, Searching Algorithms
- Software design (Dependency and Precedence graphs, concurrent processing, …)
- Internet routing, Links between Websites
- Currency trading and Fraud detection (detecting financial crimes by tracking the flow of money in directed graphs)
- Industry (Circuit boards, differentiating between chemical compounds,…)

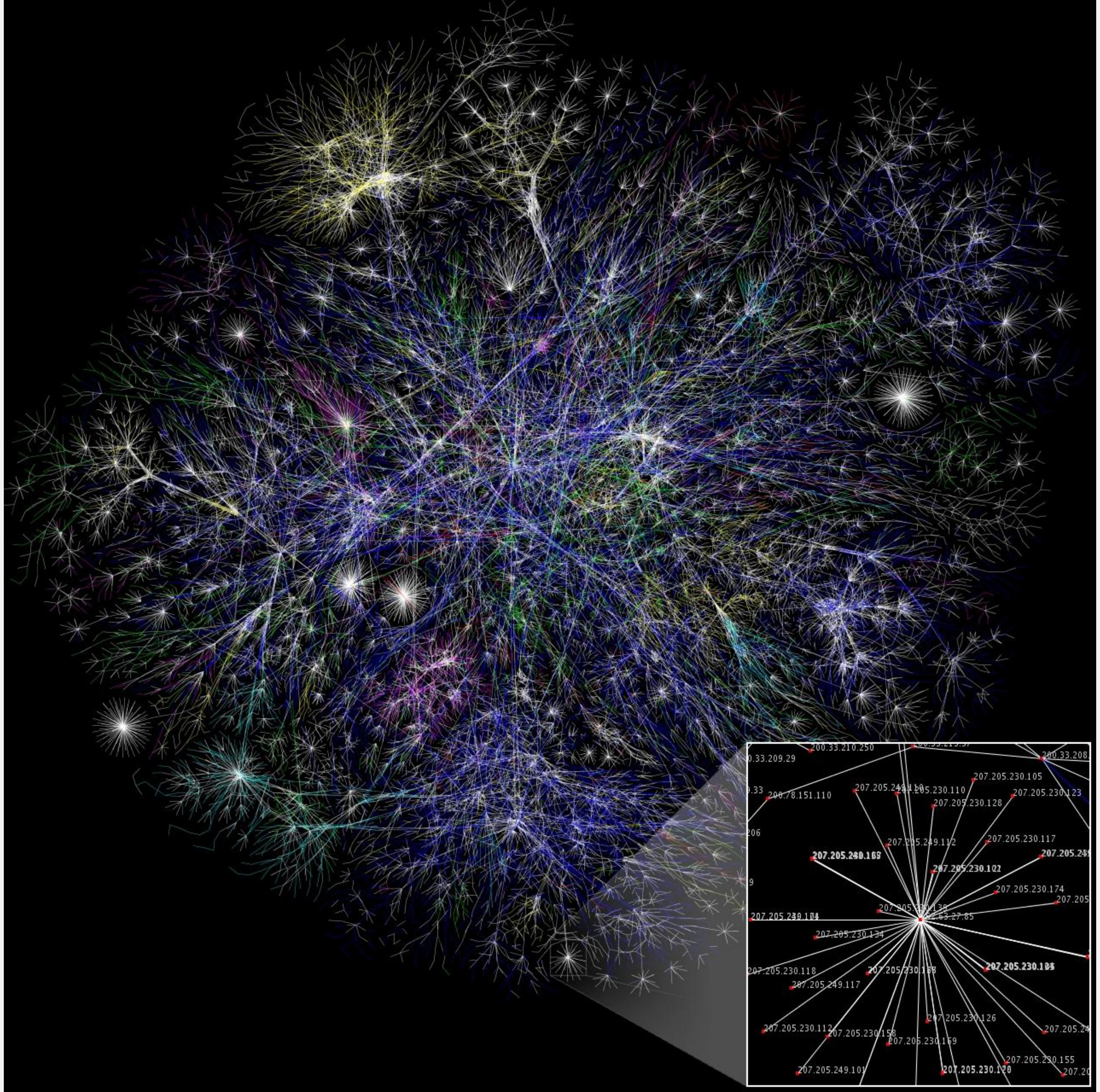
Opte Project

Visualization of the various routes through a portion of the Internet

2005

https://www.opte.org/the-internet

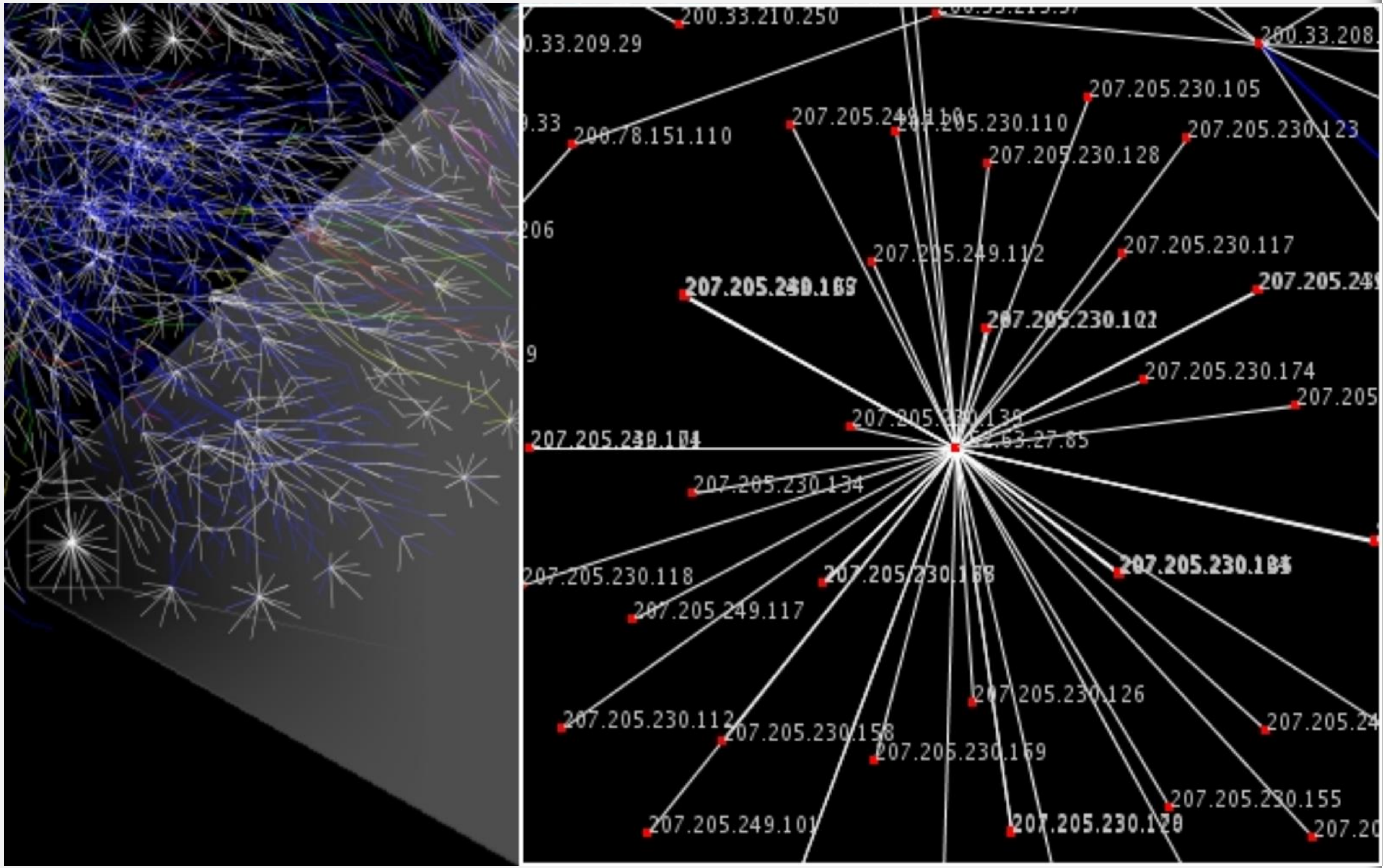
https://en.wikipedia.org/wiki/Opte_Project



Opte Project

Visualization of the various routes through a portion of the Internet

2005



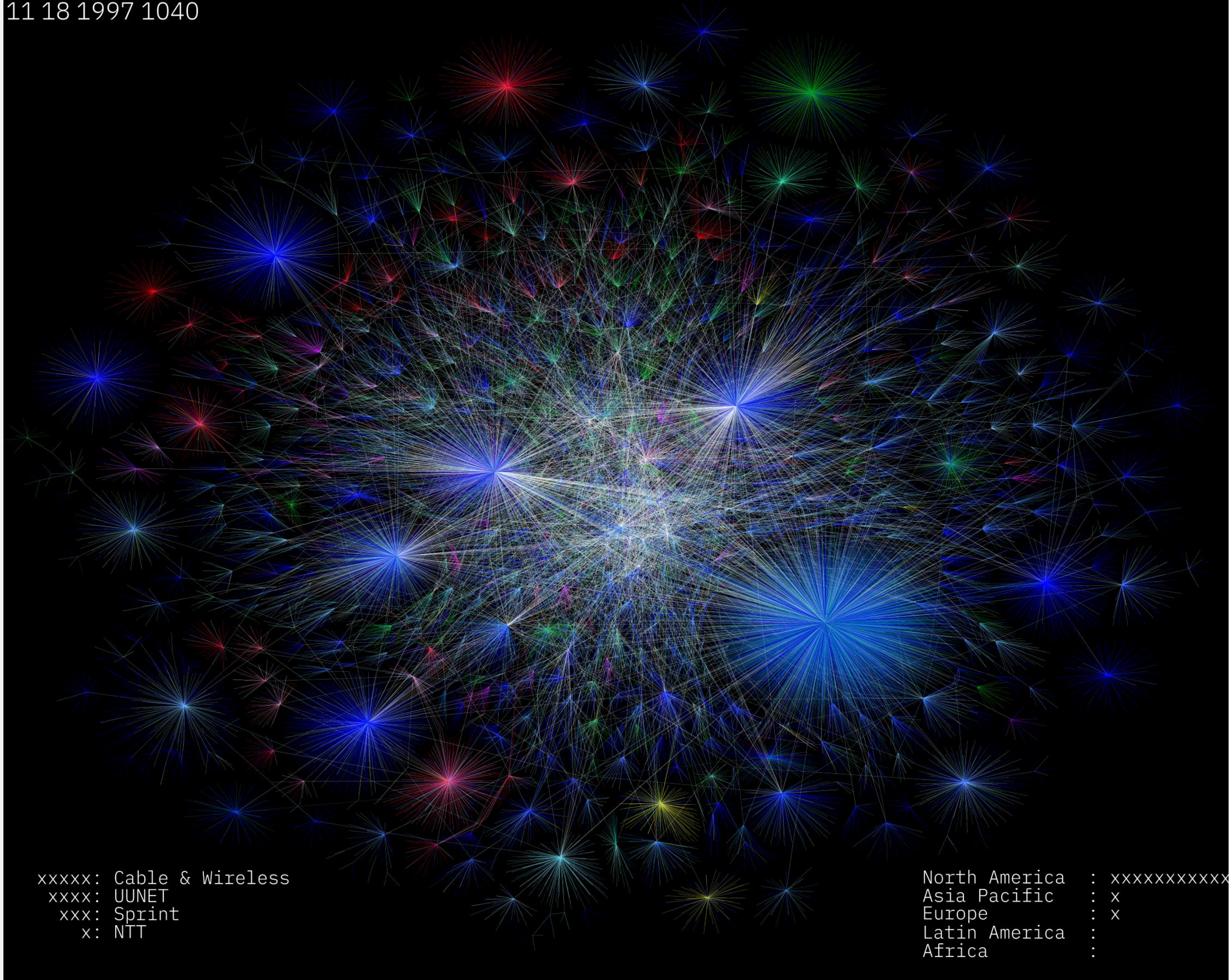
https://en.wikipedia.org/wiki/Opte_Project

Opte Project

Visualization of the various routes through a portion of the Internet

1997

https://www.opte.org/the-internet

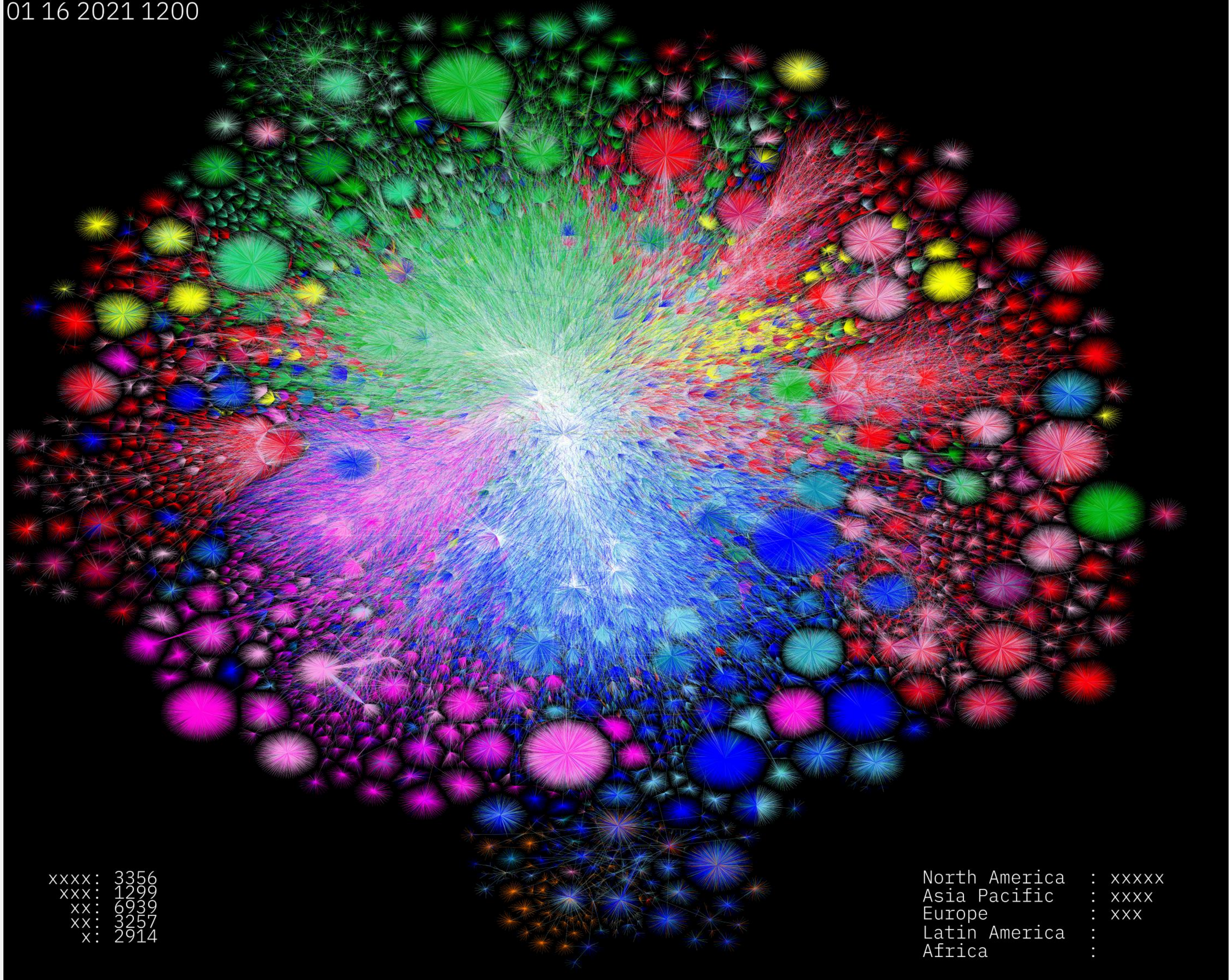


Opte Project

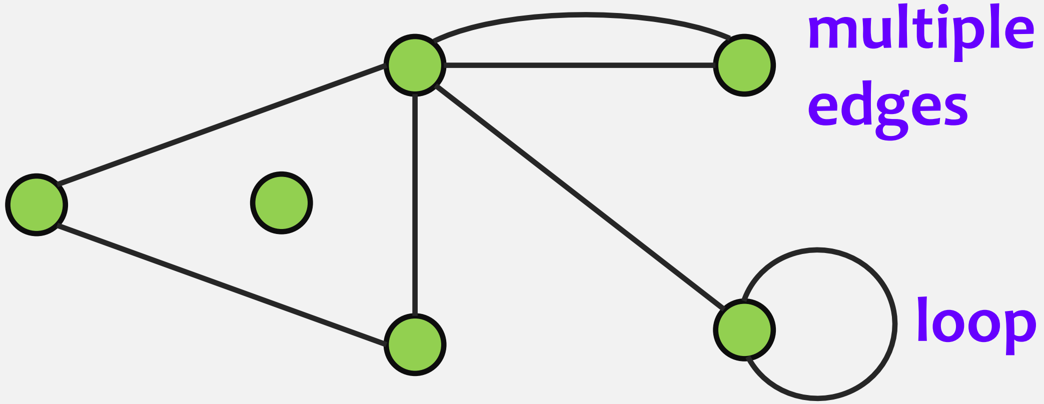
Visualization of the various routes through a portion of the Internet

2021

https://www.opte.org/the-internet



What is a graph?



$G = (V, E)$ is a **graph**

V is a set of **vertices** (**nodes**)

E is a set of **edges**

Each edge has either one or two vertices associated with it, called its **endpoints**.

An edge is said to **connect** its endpoints.

Vertices, connected by an edge are called **adjacent** (also called **neighbours**).

If edge e connects vertices u and v, then e is **incident on** u, **incident on** v, or **incident with** u and v.

In a **simple graph** each edge connects two different vertices (**no loops**), and no two edges connect the same pair of vertices (**no multiple edges**).

Intro to Graphs

$G = (V, E)$ is a **graph**

V is a set of **vertices** (**nodes**)

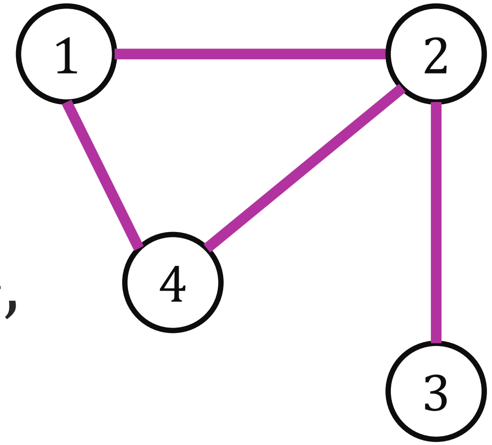
E is a set of **edges**

Undirected Graph

Each edge is an unordered pair $\{u, v\}$, where $u \in V, v \in V, u \neq v$.

$V = \{1, 2, 3, 4\}$

$E = \{\{1, 2\}, \{1, 4\}, \{2, 3\}, \{2, 4\}\}$

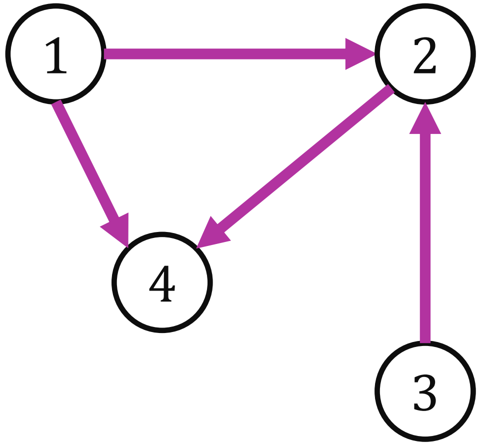


Directed Graph

Each edge is an ordered pair (u, v), where $u \in V, v \in V, u \neq v$. ("one-way street")

$V = \{1, 2, 3, 4\}$

$E = \{(1, 2), (1, 4), (2, 4), (3, 2)\}$



Intro to Graphs

$\boldsymbol{G = (V, E)}$ is a **graph**

$\boldsymbol{V}$ is a set of **vertices** (**nodes**)

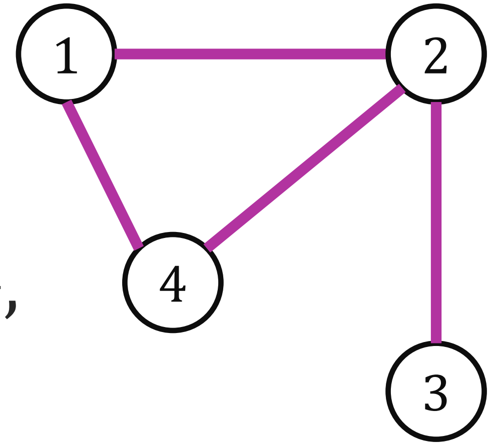
$\boldsymbol{E}$ is a set of **edges**

Undirected Graph

Each edge is an unordered pair $\{\boldsymbol{u}, \boldsymbol{v}\}$, where $\boldsymbol{u} \in \boldsymbol{V}, \boldsymbol{v} \in \boldsymbol{V}, \boldsymbol{u} \neq \boldsymbol{v}$.

$$\boldsymbol{V = \{1, 2, 3, 4\}}$$

$$\boldsymbol{E = \{\{1, 2\}, \{1, 4\}, \{2, 3\}, \{2, 4\}\}}$$



$\boldsymbol{deg(u)}$ = number of edges that contain $\boldsymbol{u}$.

$$\boldsymbol{deg(2) = 3}$$

$$\boldsymbol{\sum_{u \in V} deg(u) = 2|E|}$$

Intro to Graphs

In a directed graph, vertices have
- **inDegree** – number of edges coming into the vertex
- **outDegree** – number of edges leaving the vertex

$\boldsymbol{n = |V|}$ number of vertices of $\boldsymbol{G}$

$\boldsymbol{m = |E|}$ number of edges of $\boldsymbol{G}$

$\boldsymbol{V = \{v_0, v_1, \dots , v_{n-1}\}}$ or simply $\boldsymbol{V = \{0, \dots, n-1\}}$

Any other data that we would like to associate with the elements of $\boldsymbol{V}$ can be stored in an array of length $\boldsymbol{n}$.

$\boldsymbol{E =}$ set of (possibly ordered) pairs

addEdge$(\boldsymbol{i, j})$ – Add the edge $(\boldsymbol{i, j})$ to $\boldsymbol{E}$
removeEdge$(\boldsymbol{i, j})$ – Remove the edge $(\boldsymbol{i, j})$ from $\boldsymbol{E}$
hasEdge$(\boldsymbol{i, j})$ – Check if the edge $(\boldsymbol{i, j}) \in \boldsymbol{E}$
outEdges $(\boldsymbol{i})$ – Return a List of all integers $\boldsymbol{j}$ such that $(\boldsymbol{i, j}) \in \boldsymbol{E}$
inEdges $(\boldsymbol{i})$ – Return a List of all integers $\boldsymbol{j}$ such that $(\boldsymbol{j, i}) \in \boldsymbol{E}$

How to store a graph?

How do we represent and store a graph in a computer?

1. **Adjacency Matrix**
2. **Adjacency List**

We would like to perform some operations on the graph such as

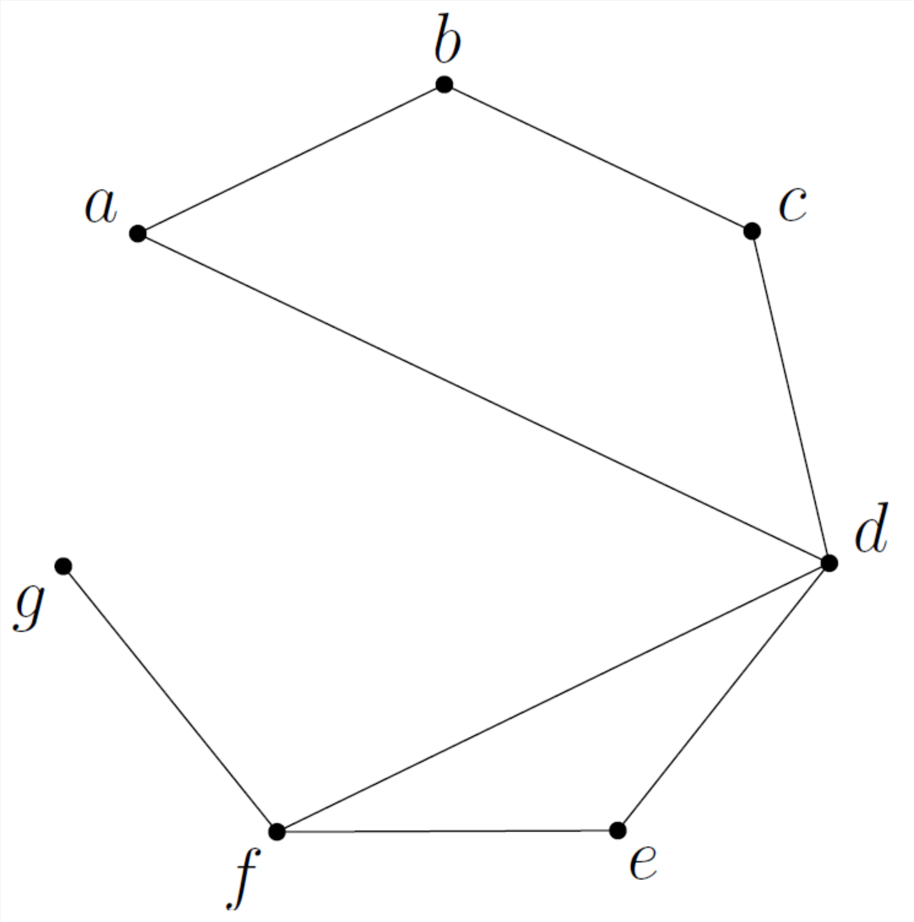
- Ask if two nodes are adjacent
- Find the degree of a vertex or list all its neighbours
- Add a vertex or an edge to a graph
- Remove a vertex or an edge from a graph

We want the operations to be fast and we don't want to waste too much memory.

Example

Draw the simple undirected graph $\boldsymbol{G = (V, E)}$ represented by the adjacency matrix below.

$$
\begin{array}{c}
\\ a \\ b \\ c \\ d \\ e \\ f \\ g
\end{array}
\begin{array}{c}
\begin{array}{ccccccc} a & b & c & d & e & f & g \end{array} \\
\begin{bmatrix}
0 & 1 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0
\end{bmatrix}
\end{array}
$$



What does it mean if we have **1** on the main diagonal?

Is it possible to use adjacency matrix if we have multiple (parallel) edges?

Adjacency Matrix

$G = (V, E)$ simple, $V = \{v_0, v_1, \dots, v_{n-1}\}$ Adjacency Matrix is $n \times n$ matrix

If G is undirected: entry $(i, j) = \begin{cases} 1 & \text{if } \{v_i, v_j\} \text{ is an edge} \\ 0 & \text{otherwise} \end{cases}$

symmetric matrix

If G is directed: entry $(i, j) = \begin{cases} 1 & \text{if } (v_i, v_j) \text{ is an edge} \\ 0 & \text{otherwise} \end{cases}$

Adjacency Matrix

$G = (V, E)$ simple, $V = \{v_0, v_1, \dots, v_{n-1}\}$ Adjacency Matrix is $n \times n$ matrix

If G is undirected: entry $(i, j) = \begin{cases} 1 & \text{if } \{v_i, v_j\} \text{ is an edge} \\ 0 & \text{otherwise} \end{cases}$ ← **symmetric matrix**

If G is directed: entry $(i, j) = \begin{cases} 1 & \text{if } (v_i, v_j) \text{ is an edge} \\ 0 & \text{otherwise} \end{cases}$

Advantage

In $O(1)$ time we can test if there is an edge between two given vertices.

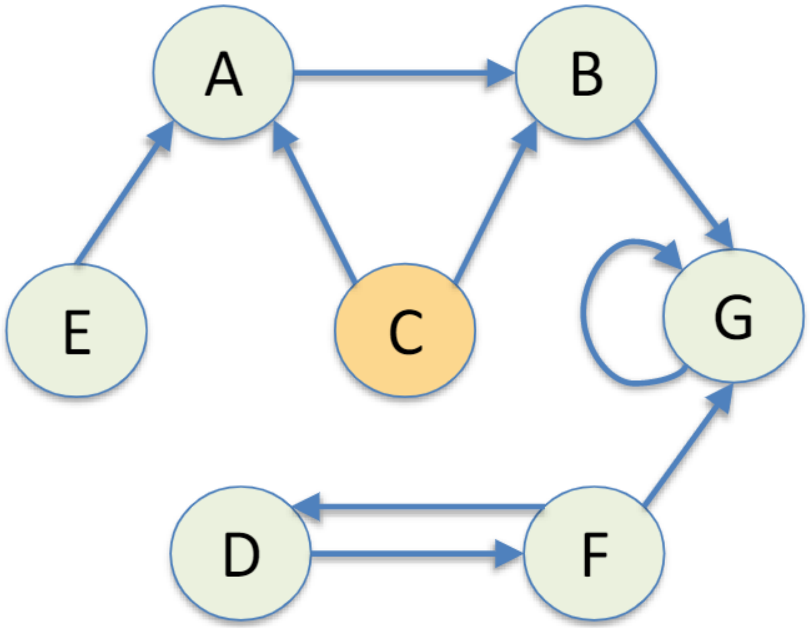
Disadvantage

- Uses $\Theta(n^2)$ space for any graph
- To find all neighbors of a given vertex takes $\Theta(n)$ time.

Graphs

Notice that each row in the matrix corresponds to the outEdges of a node.

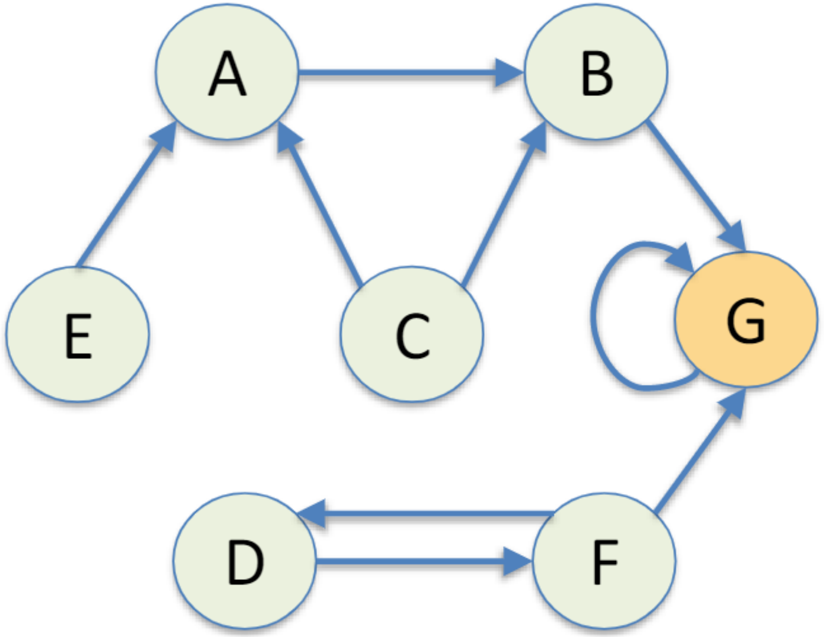
	A	B	C	D	E	F	G
A	0	1	0	0	0	0	0
B	0	0	0	0	0	0	1
C	1	1	0	0	0	0	0
D	0	0	0	0	0	1	0
E	1	0	0	0	0	0	0
F	0	0	0	1	0	0	1
G	0	0	0	0	0	0	1



Graphs

Notice that each column in the matrix corresponds to the inEdges of a node.

	A	B	C	D	E	F	G
A	0	1	0	0	0	0	0
B	0	0	0	0	0	0	1
C	1	1	0	0	0	0	0
D	0	0	0	0	0	1	0
E	1	0	0	0	0	0	0
F	0	0	0	1	0	0	1
G	0	0	0	0	0	0	1



Adjacency Matrix

$\Theta(n^2)$ space

G is a directed graph

$$a[i][j] = \begin{cases} \text{true} & \text{if } (i,j) \in E \\ \text{false} & \text{otherwise} \end{cases}$$

```
boolean[][] a;
int n;
AdjacencyMatrix(int n0) {
  n = n0;
  a = new boolean[n][n];
}
```

```
void addEdge(int i, int j) {
  a[i][j] = true;
}
void removeEdge(int i, int j) {
  a[i][j] = false;
}
boolean hasEdge(int i, int j) {
  return a[i][j];
}
```

$O(1)$

```
List<Integer> outEdges(int i) {
  List<Integer> edges = new ArrayList<Integer>();
  for (int j = 0; j < n; j++)
    if (a[i][j]) edges.add(j);
  return edges;
}
List<Integer> inEdges(int i) {
  List<Integer> edges = new ArrayList<Integer>();
  for (int j = 0; j < n; j++)
    if (a[j][i]) edges.add(j);
  return edges;
}
```

$O(n)$

Adjacency Lists

In this course we assume that each list is sorted (in either numerical or alphabetical order).

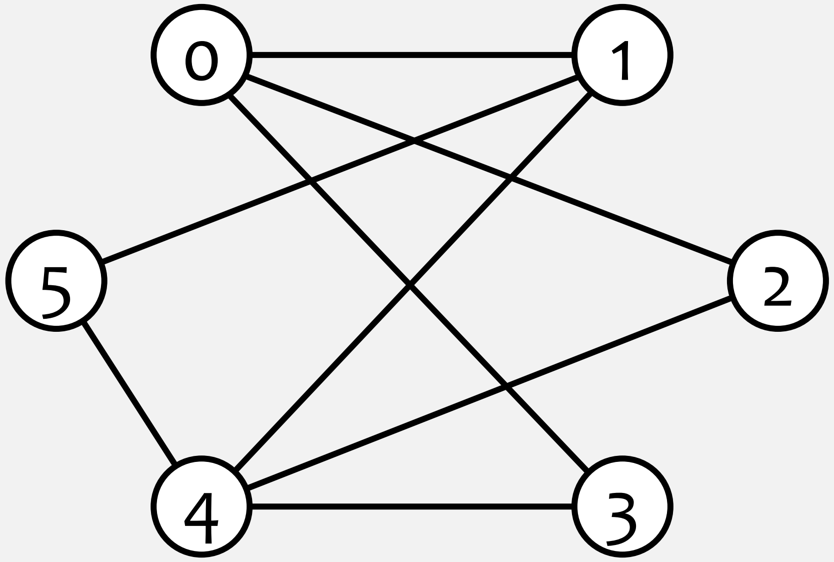
$G = (V, E)$, Each vertex u stores a list. There are $|V|$ linked lists.

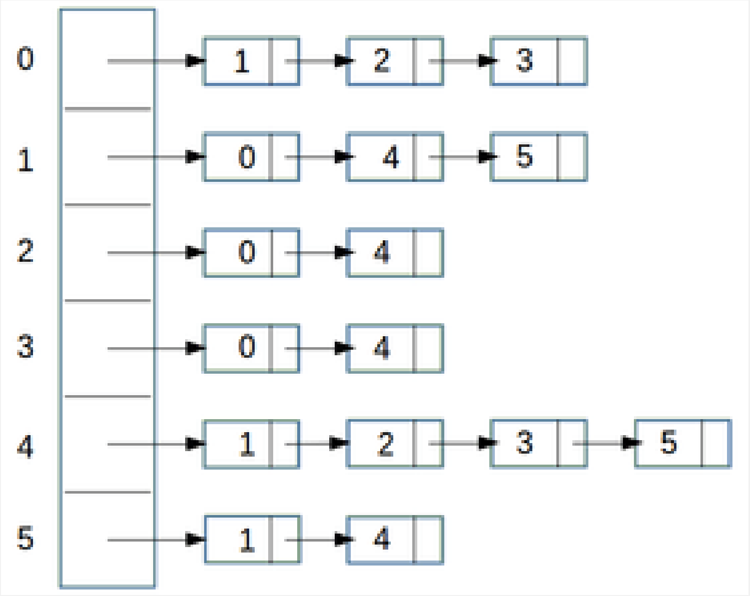
If G is undirected: the list of u stores all neighbors of u:

all v for which $\{u, v\} \epsilon E$.

If G is directed: the list of u stores all v for which $(u, v) \in E$.

outgoing edges





Adjacency Lists

$G = (V, E)$, Each vertex u stores a list. There are $|V|$ linked lists.

If G is undirected: the list of u stores all neighbors of u:

all v for which $\{u, v\} \epsilon E$.

If G is directed: the list of u stores all v for which $(u, v) \in E$.

outgoing edges

Advantage

- Uses $\Theta(|V| + |E|)$ space
- all neighbors of vertex u can be found in $O(1 + deg(u))$ time.

Disadvantage

Testing if $\{u, v\}$ (or (u, v)) is an edge takes $O(1 + deg(u))$ time.

Adjacency Lists

G is represented as an array, **adj**, of lists.
The lists are implemented using **ArrayStack**

```
List<Integer>[] adj;
int n;
AdjacencyLists(int n0) {
  n = n0;
  adj = (List<Integer>[])new List[n];
  for (int i = 0; i < n; i++)
    adj[i] = new ArrayStack<Integer>(Integer.class);
}
```

$\Theta(n+m)$ space

```
void addEdge(int i, int j) {
  adj[i].add(j);
}
```

$O(1)$

```
boolean hasEdge(int i, int j) {
  return adj[i].contains(j);
}
```

$O(1+\deg(i))$

```
void removeEdge(int i, int j) {
  Iterator<Integer> it = adj[i].iterator();
  while (it.hasNext()) {
    if (it.next() == j) {
      it.remove();
      return;
    }
  }
}
```

$O(1+\deg(i))$

```
List<Integer> inEdges(int i) {
  List<Integer> edges = new ArrayStack<Integer>(Integer.class);
  for (int j = 0; j < n; j++)
    if (adj[j].contains(i))  edges.add(j);
  return edges;
}
```

$O(n+m)$

```
List<Integer> outEdges(int i) {
  return adj[i];
}
```

$O(1)$

Which method to choose?

1 it depends on what operations we want to do with the graph

	space	access time
Adjacency Matrix	$\boldsymbol{\Theta(n^2)}$	$\boldsymbol{O(1)}$
Adjacency List	$\boldsymbol{\Theta(n+m)}$	$\boldsymbol{O(1+deg(u))}$

Consider the following operations on a graph.

- is_there_an_Edge(v_i, v_j)
- add/remove_Edge(v_i, v_j)
- list_neigbors(v_i)
- degree(v_i)
- in/out_degree(v_i)
- add/remove(v_i)

What is the runtime complexity of each?

It depends on how we store a graph.

Which method to choose?

2 it depends on the relationship between $|V| = n$ and $|E| = m$

	space	access time
Adjacency Matrix	$\Theta(n^2)$	$O(1)$
Adjacency List	$\Theta(n+m)$	$O(1+deg(u))$

How big is your graph?

Which method to choose?

	space	access time
Adjacency Matrix	$\boldsymbol{\Theta(n^2)}$	$\boldsymbol{O(1)}$
Adjacency List	$\boldsymbol{\Theta(n+m)}$	$\boldsymbol{O(1+deg(u))}$

Use Adjacency Matrix:

- when the graph $\boldsymbol{G}$ is **dense**, i.e., it has close to $\boldsymbol{n^2}$ edges, then a memory usage of $\boldsymbol{n^2}$ may be acceptable.
- to compute the shortest paths between all pairs of vertices in $\boldsymbol{G}$. This can be done using only $\boldsymbol{O}(\log \boldsymbol{n})$ matrix multiplications.

  Some properties of graphs can be computed efficiently using algebraic operations on matrices.

Use Adjacency List: in almost all other cases