Graph Algorithms: Lecture 1

Introduction

Michael Lampis

August 29, 2025

Before we begin

Before we begin

- This is a **Theoretical Computer Science** course. . .
  - Focus will be on algorithms and their theoretical analysis.
  - We will discuss algorithms that solve problems on **graphs**.
  - **Proofs** will be important (just like in a math course).

Before we begin

- This is a **Theoretical Computer Science** course...
  - Focus will be on algorithms and their theoretical analysis.
  - We will discuss algorithms that solve problems on **graphs**.
  - **Proofs** will be important (just like in a math course).
- ...with a bit of an applied side
  - The course includes a (small) programming component.
  - We will be (somewhat) interested in real-world relevance (applications).

Administrative Stuff

- Course Instructor: Michael Lampis (michail.lampis AT dauphine.fr)
- Course Web page:
  `https://www.lamsade.dauphine.fr/~mlampis/GraphAlgs/`
- Grade Calculation:
  - Midterm Exam: 30% of grade (date: TBD)
  - Final Exam: 70% of grade
- Material to Study:
  - Slides (posted on web page)
  - TD exercises and solutions (posted on web page)
  - Further reading material linked on web page

Administrative Stuff

- Course Instructor: Michael Lampis (michail.lampis AT dauphine.fr)
- Course Web page:
  `https://www.lamsade.dauphine.fr/~mlampis/GraphAlgs/`
- Grade Calculation:
  - Midterm Exam: 30% of grade (date: TBD)
  - Final Exam: 70% of grade
- Material to Study:
  - Slides (posted on web page)
  - TD exercises and solutions (posted on web page)
  - Further reading material linked on web page
- **Please** come to class and participate actively!
- **NB**: Programming component is **NOT** graded.

Motivation

Graphs

Definition

(Informal) A graph is a mathematical object that models **identical pair-wise symmetric relations** between objects.

Graphs

Definition

(Informal) A graph is a mathematical object that models **identical pair-wise symmetric relations** between objects.

Application Examples:

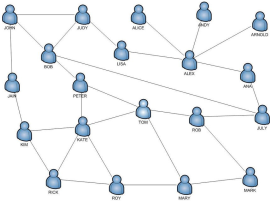
Graphs

Definition

(Informal) A graph is a mathematical object that models **identical pair-wise symmetric relations** between objects.

Application Examples:

Social Network



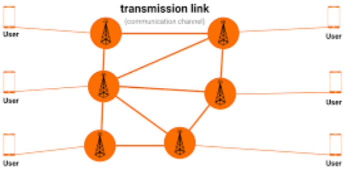
Graphs

Definition

(Informal) A graph is a mathematical object that models **identical pair-wise symmetric relations** between objects.

Application Examples:

Telecommunication Network



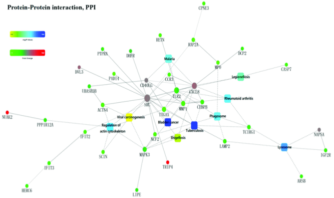
Graphs

Definition

(Informal) A graph is a mathematical object that models **identical pair-wise symmetric relations** between objects.

Application Examples:

Protein-Protein Interactions



Graphs

Definition

(Informal) A graph is a mathematical object that models **identical pair-wise symmetric relations** between objects.

Definition

A **simple graph** $G = (V, E)$ is a pair of a set of **vertices** and **edges**, with $E \subseteq \binom{V}{2}$.

- Pair-wise. $e = \{u, v\}$, for $e \in E, u, v \in V$. We write simply $e = uv$.
  - Otherwise: hypergraph
- Identical.
  - Otherwise: weighted graph, multi-graph
- Symmetric.
  - Otherwise: directed graph

Wolf-Goat-Cabbage

Wolf

Goat

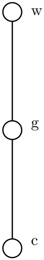
Cabbage

Wolf-Goat-Cabbage

Wolf

Goat

Cabbage



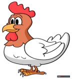
Wolf-Goat-Cabbage

Wolf

Fox

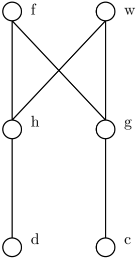
Goat

Hen

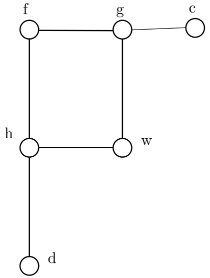


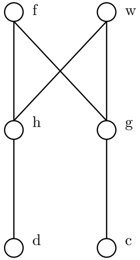
Cabbage

Corn



Wolf-Goat-Cabbage

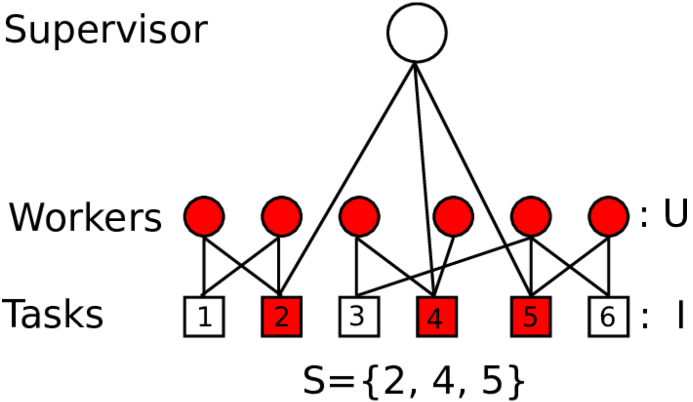




Mathematical definition:

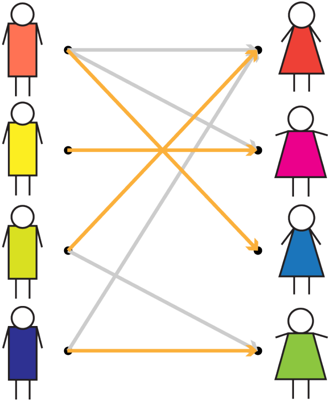
- $V = \{f, w, g, h, d, c\}$
- $E = \{wg, gc, wh, fg, fh, hd\}$

Other graphs



Matching Problems

Other graphs



Matching Problems

Other graphs

$w, g, c \leftarrow \emptyset$

$\emptyset \rightarrow w, g, c$

Configuration graph

Other graphs

$w, g, c \to \emptyset$

$c \leftarrow w, g$

$w, g \leftarrow c$

$c \to w, g$

$w, g, c \leftarrow \emptyset$ $\quad$ $w, g \to c$ $\quad$ $\emptyset \to w, g, c$

$w \leftarrow g, c$

$w, c \leftarrow g$

$w \to g, c$

$w, c \to g$

$g \leftarrow w, c$

$g, c \leftarrow w$

$g \to w, c$

$g, c \to w$

Configuration graph

Other graphs

$\cancel{w, g, c \to \emptyset}$

$c \leftarrow w, g$

$w, g \leftarrow c$

$c \to w, g$

$w, g, c \leftarrow \emptyset$ $w, g \to c$ $\emptyset \to w, g, c$

$w \leftarrow g, c$

$w, c \leftarrow g$

$w \to g, c$

$w, c \to g$

$g \leftarrow w, c$

$g, c \leftarrow w$

$g \to w, c$

$g, c \to w$

Configuration graph

Other graphs

$\cancel{w, g, c \rightarrow \emptyset}$

$w, g, c \leftarrow \emptyset$ $\quad$ $w, g \rightarrow c$ $\quad$ $w, g \leftarrow c$ $\quad$ $c \rightarrow w, g$ $\quad$ $\cancel{c \leftarrow w, g}$ $\quad$ $\emptyset \rightarrow w, g, c$

$w, c \rightarrow g$ $\quad$ $w, c \leftarrow g$ $\quad$ $w \rightarrow g, c$ $\quad$ $w \leftarrow g, c$

$g, c \rightarrow w$ $\quad$ $g, c \leftarrow w$ $\quad$ $g \rightarrow w, c$ $\quad$ $g \leftarrow w, c$

Configuration graph

Other graphs

$w, g, c \leftarrow \emptyset$

$\cancel{w, g, c \rightarrow \emptyset}$

$\cancel{w, g \rightarrow c}$

$w, c \rightarrow g$

$\cancel{g, c \rightarrow w}$

$w, g \leftarrow c$

$w, c \leftarrow g$

$g, c \leftarrow w$

$c \rightarrow w, g$

$w \rightarrow g, c$

$g \rightarrow w, c$

$\cancel{c \leftarrow w, g}$

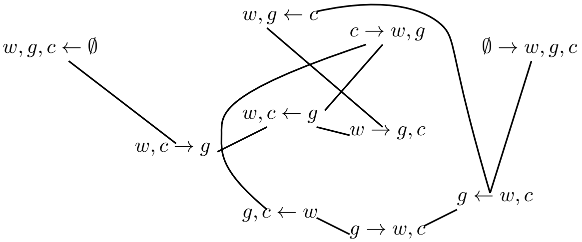
$\cancel{w \leftarrow g, c}$

$g \leftarrow w, c$

$\emptyset \rightarrow w, g, c$

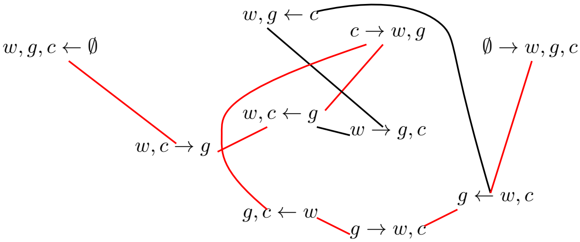
Configuration graph

Other graphs



Configuration graph

Other graphs



Configuration graph

Directed graph example

Angela

Directed graph example

Angela

Boris

Directed graph example

Angela

Boris

Christine

Directed graph example

Angela

Boris

Christine

Donald

Directed graph example

Angela

Boris

Christine

Donald

Emmanuel

Directed graph example


Angela

$\succeq$


Boris

$\succeq$


Christine

$\succeq$


Donald

$\succeq$


Emmanuel

Directed graph example

 $\succeq$  $\succeq$  $\succeq$  $\succeq$ 

Angela Boris Christine Donald Emmanuel

 $\succeq$  $\succeq$  $\succeq$  $\succeq$ 

Directed graph example

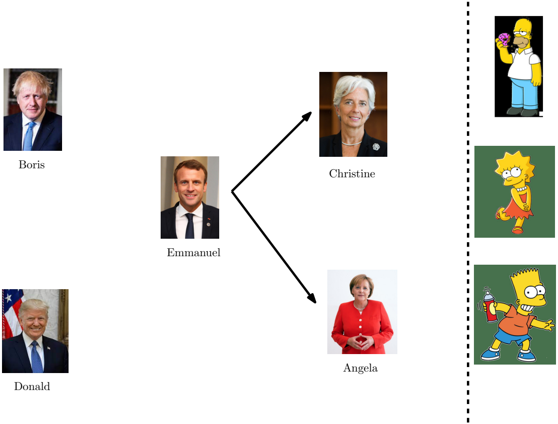
 $\succeq$  $\succeq$  $\succeq$  $\succeq$ 

Angela Boris Christine Donald Emmanuel

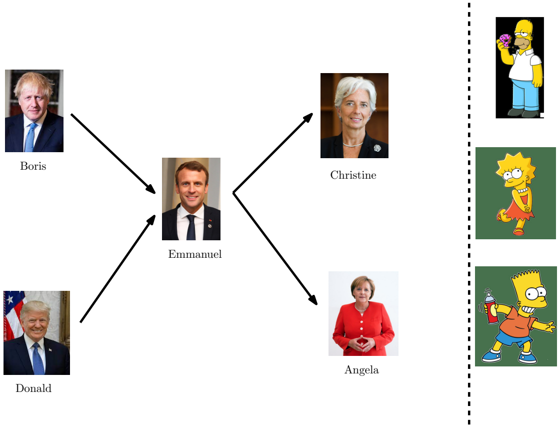
 $\succeq$  $\succeq$  $\succeq$  $\succeq$ 

 $\succeq$  $\succeq$  $\succeq$  $\succeq$ 

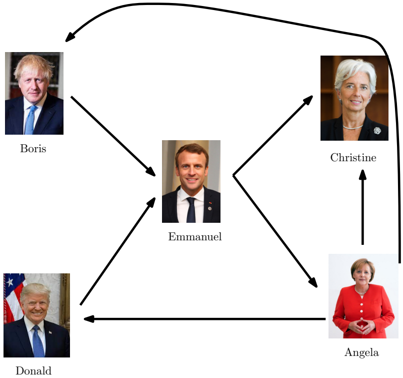
Directed graph example



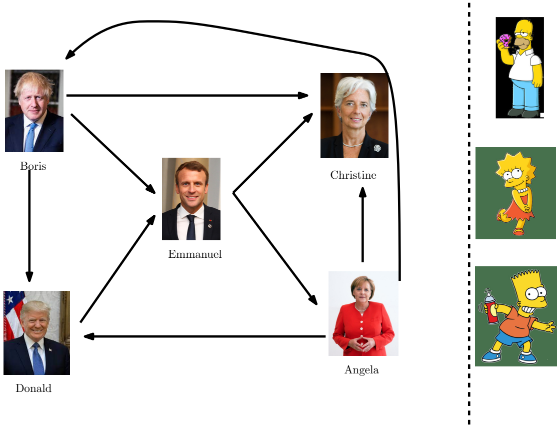
Directed graph example



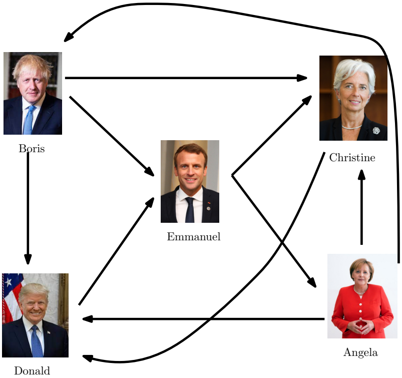
Directed graph example



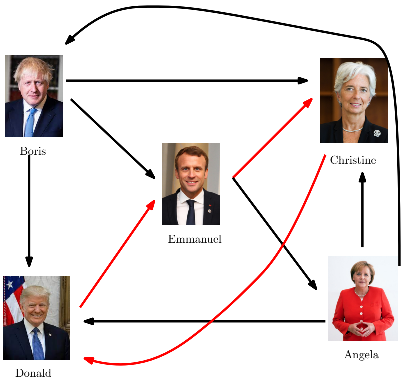
Directed graph example



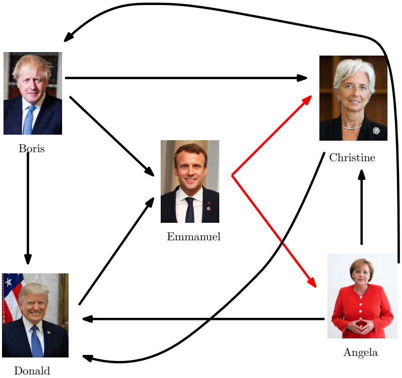
Directed graph example



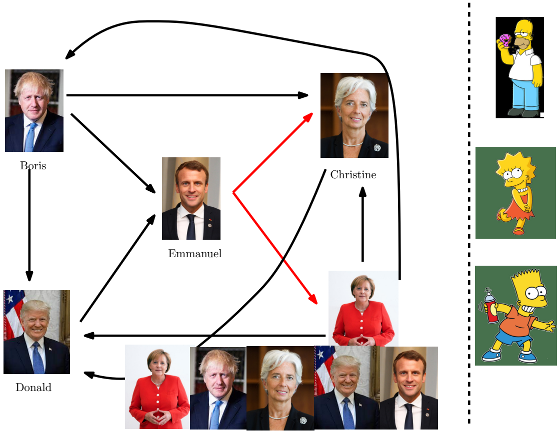
Directed graph example



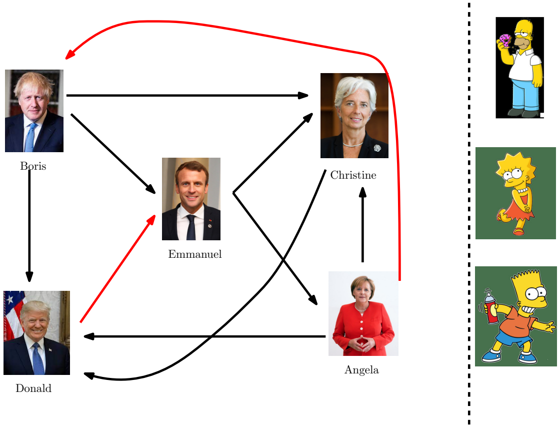
Directed graph example



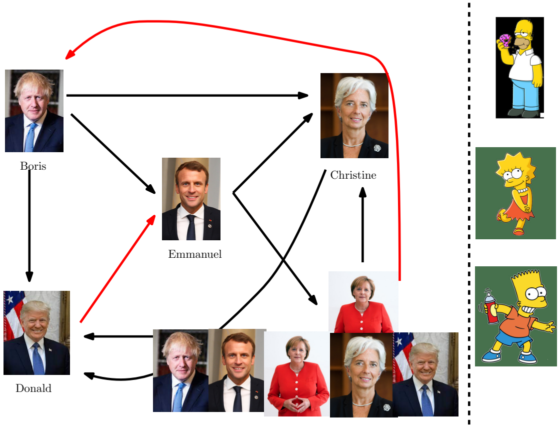
Directed graph example



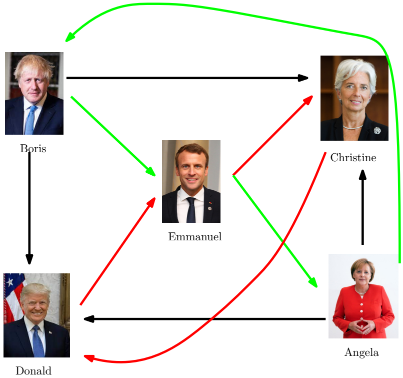
Directed graph example



Directed graph example



Directed graph example



Basic Definitions

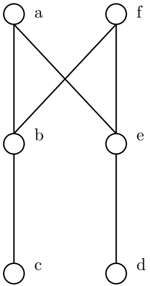
Graph Representations – Isomorphism

Adjacency Matrix:

	a	b	c	d	e	f
a	0	1	0	0	1	0
b	1	0	1	0	0	1
c	0	1	0	0	0	0
d	0	0	0	0	1	0
e	1	0	0	1	0	1
f	0	1	0	0	1	0

- $n \times n$ **symmetric** matrix
- 0 diagonal
- Number of 1's $= 2m$

Graph:



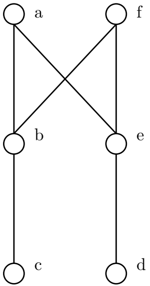
Graph Representations – Isomorphism

Incidence Matrix:

	ab	ae	bf	bc	de	ef
a	1	1	0	0	0	0
b	1	0	1	1	0	0
c	0	0	0	1	0	0
d	0	0	0	0	1	0
e	0	1	0	0	1	1
f	0	0	1	0	0	1

- $n \times m$ matrix
- Two 1's per column
- Number of 1's $= 2m$

Graph:

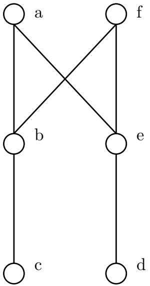


Graph Representations – Isomorphism

Adjacency Matrix:

	a	b	c	d	e	f
a	0	1	0	0	1	0
b	1	0	1	0	0	1
c	0	1	0	0	0	0
d	0	0	0	0	1	0
e	1	0	0	1	0	1
f	0	1	0	0	1	0

Graph:



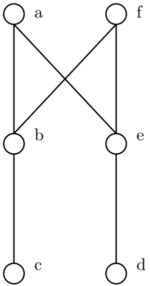
- Several different matrices could represent the same graph!
- Permuting rows/columns does not change the graph.

Graph Representations – Isomorphism

Adjacency lists:

a	b,e
b	a,c,f
c	b
d	e
e	a,d,f
f	b,e

Graph:



A graph may also be represented by:

- n lists of neighbors
- . . . or even a list of edges

Which representation is “better”?

Algorithmic Background

Polynomial Time

Algorithmic Efficiency: we care about

- Time/Space Complexity
- In the worst case
- As function of input size (n)
- Polynomial in n is good!

Polynomial Time

Algorithmic Efficiency: we care about

- Time/Space Complexity
- In the worst case
- As function of input size (n)
- Polynomial in n is good!
- And we care about which polynomial!

Reminder: O-notation

Definitions:

- $f(n) = O(g(n)) \Leftrightarrow \exists C, n_0 \forall n > n_0 : f(n) \leq Cg(n)$
- $f(n) = \Omega(g(n)) \Leftrightarrow \exists C, n_0 \forall n > n_0 : f(n) \geq Cg(n)$
- $f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$
- $f(n) = o(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
- $f(n) = \omega(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

Reminder: O-notation

(Slightly Inaccurate) Definitions:

- $f(n) = O(g(n)) \Leftrightarrow f(n) \leq g(n)$
- $f(n) = \Omega(g(n)) \Leftrightarrow f(n) \geq g(n)$
- $f(n) = \Theta(g(n)) \Leftrightarrow f(n) \approx g(n)$
- $f(n) = o(g(n)) \Leftrightarrow f(n) << g(n)$
- $f(n) = \omega(g(n)) \Leftrightarrow f(n) >> g(n)$

Reminder: O-notation

Intuition:

- We care what happens when n is huge ($\rightarrow$ asymptotically):
- n^2 and $3n^2 + 25n$ are "roughly" the same
- $500n^2$ is "much less" than $\frac{n^3}{10}$
- $\log n$ is negligible compared to n which is negligible compared to 2^n
- (We do care about the distinction between n and $n \log n$)

Reminder: Basic data structures

- Array/Matrix
  - $O(1)$ time to access/modify an arbitrary element
  - $O(n)$ time to add/remove an element
- Linked List
  - $O(1)$ time to access first or last element
  - $O(n)$ time to access arbitrary element
  - $O(1)$ time to add/remove first/last element

Reminder: Basic data structures

- Array/Matrix
  - $O(1)$ time to access/modify an arbitrary element
  - $O(n)$ time to add/remove an element
- Linked List
  - $O(1)$ time to access first or last element
  - $O(n)$ time to access arbitrary element
  - $O(1)$ time to add/remove first/last element

Other data structures: Stack, Queue, Priority Queue...

Graph Representations

Which graph representation is more efficient?

- In graph G with n vertices and m edges:
  - $m = O(n^2)$
  - $m = \Omega(n)$ (unless we have many isolated vertices, which is silly)

Graph Representations

Which graph representation is more efficient?

- In graph G with n vertices and m edges:
  - $m = O(n^2)$
  - $m = \Omega(n)$ (unless we have many isolated vertices, which is silly)
- Adjacency matrix size: $\Theta(n^2)$
- Incidence matrix size: $\Theta(nm)$
- Adjacency lists: $\Theta(n + m)$

Graph Representations

Which graph representation is more efficient?

- In graph G with n vertices and m edges:
  - $m = O(n^2)$
  - $m = \Omega(n)$ (unless we have many isolated vertices, which is silly)
- Adjacency matrix size: $\Theta(n^2)$
- Incidence matrix size: $\Theta(nm)$
- Adjacency lists: $\Theta(n + m)$
- Adjacency lists are always best!

Graph Representations

Which graph representation is more efficient?

- In graph G with n vertices and m edges:
  - $m = O(n^2)$
  - $m = \Omega(n)$ (unless we have many isolated vertices, which is silly)
- Adjacency matrix size: $\Theta(n^2)$

-)
-
- st!

Log n is a constant?!

- Total size of an adjacency list representation is $O(m)$...
- ...if we assume that storing the index of a vertex takes $O(1)$ space.

Log n is a constant?!

- Total size of an adjacency list representation is $O(m)$...
- ...if we assume that storing the index of a vertex takes $O(1)$ space.
- If the index of a vertex takes k bits, then there are at most 2^k vertices
- If $k = O(1)$, then $2^k = O(1)$, so n is a constant!?!?

Log n is a constant?!

- Total size of an adjacency list representation is $O(m)$...
- ...if we assume that storing the index of a vertex takes $O(1)$ space.
- If the index of a vertex takes k bits, then there are at most 2^k vertices
- If $k = O(1)$, then $2^k = O(1)$, so n is a constant!?!?
- Actually, adjacency lists take space $O(m \log n)$, because we need $\log n$ bits to give the index of a vertex.
- But, for any reasonable input $n < 2^{100}$, so $\log n$ is basically a constant.

Log n is a constant?!

- Total size of an adjacency list representation is $O(m)$...
- ...if we assume that storing the index of a vertex takes $O(1)$ space.
- If the index of a vertex takes k bits, then there are at most 2^k vertices
- If $k = O(1)$, then $2^k = O(1)$, so n is a constant!?!?
- Actually, adjacency lists take space $O(m \log n)$, because we need $\log n$ bits to give the index of a vertex.
- But, for any reasonable input $n < 2^{100}$, so $\log n$ is basically a constant.
- Take home message: this is why we don't worry too much about $\log n$ factors. n factors are another story...

Notation Basics

Conventions and Interesting Graphs

- $n = |V|$, $m = |E|$
- $uv \in E \Rightarrow u, v$ are adjacent or neighbors
- $N(v)$: set of neighbors of v
- $e = uv \in E \Rightarrow e$ is incident on u
- Degree $d(v)$: number of edges incident on v
- Δ: maximum degree

Conventions and Interesting Graphs

- $n = |V|$, $m = |E|$
- $uv \in E \Rightarrow u, v$ are adjacent or neighbors
- $N(v)$: set of neighbors of v
- $e = uv \in E \Rightarrow e$ is incident on u
- Degree $d(v)$: number of edges incident on v
- Δ: maximum degree

- Clique K_n: all n vertices adjacent

Conventions and Interesting Graphs

- $n = |V|$, $m = |E|$
- $uv \in E \Rightarrow u, v$ are adjacent or neighbors
- $N(v)$: set of neighbors of v
- $e = uv \in E \Rightarrow e$ is incident on u
- Degree $d(v)$: number of edges incident on v
- Δ: maximum degree

- Clique K_n: all n vertices adjacent
- Path P_n: path on n vertices

Conventions and Interesting Graphs

- $n = |V|$, $m = |E|$
- $uv \in E \Rightarrow u, v$ are adjacent or neighbors
- $N(v)$: set of neighbors of v
- $e = uv \in E \Rightarrow e$ is incident on u
- Degree $d(v)$: number of edges incident on v
- Δ: maximum degree

- Clique K_n: all n vertices adjacent
- Path P_n: path on n vertices
- Cycle C_n: cycle on n vertices

Conventions and Interesting Graphs

- $n = |V|$, $m = |E|$
- $uv \in E \Rightarrow u, v$ are adjacent or neighbors
- $N(v)$: set of neighbors of v
- $e = uv \in E \Rightarrow e$ is incident on u
- Degree $d(v)$: number of edges incident on v
- Δ: maximum degree

- Clique K_n: all n vertices adjacent
- Path P_n: path on n vertices
- Cycle C_n: cycle on n vertices
- Wheel W_n: C_n plus a universal vertex

Conventions and Interesting Graphs

- $n = |V|$, $m = |E|$
- $uv \in E \Rightarrow u, v$ are adjacent or neighbors
- $N(v)$: set of neighbors of v
- $e = uv \in E \Rightarrow e$ is incident on u
- Degree $d(v)$: number of edges incident on v
- Δ: maximum degree

- Clique K_n: all n vertices adjacent
- Path P_n: path on n vertices
- Cycle C_n: cycle on n vertices
- Wheel W_n: C_n plus a universal vertex

Q: Is there a polynomial-time algorithm to decide if a graph belongs in one of these classes?

Simple facts about Degrees

Theorem

For all $G = (V, E)$ *we have* $\sum_{v \in V} \deg(v) = 2|E|$.

Theorem

For all $G = (V, E)$ *the number of vertices of odd degree in* G *is even.*

Theorem

Every graph G *has two vertices with the same degree.*

Degrees and Digraphs

In directed graphs:

- $e = uv$ is an **arc** from u to v.
- The outdegree of u, $\deg^+(u)$, is the number of arcs going out of u.
- The indegree of u, $\deg^-(u)$, is the number of arcs going into u.

Theorem

For all digraphs $G = (V, A)$ we have
$\sum_{v \in V} \deg^+(v) = |A| = \sum_{v \in V} \deg^-(v)$.

Theorem

In all digraphs, there exist two vertices with the same outdegree.

Paths and Connectivity

Definition

A path is an ordered sequence of **distinct** vertices $v_1, v_2, \ldots, v_k$ such that for all $i \in [k-1]$ we have $v_i v_{i+1} \in E$.

Definition

A graph is connected if there is a path between any two of its vertices.

Paths and Connectivity

Definition

A path is an ordered sequence of **distinct** vertices $v_1, v_2, \ldots, v_k$ such that for all $i \in [k-1]$ we have $v_i v_{i+1} \in E$.

Definition

A graph is connected if there is a path between any two of its vertices.

Can we decide in polynomial time if there is a path from s to t? $\rightarrow$ Graph traversal algorithms (in a bit...)

Basic Questions

Basic Questions

- Subgraph Containment
- Short-Long Paths
- Interesting Sets
- Coloring

- G_1 is a subgraph of G_2 if it can be obtained from G_2 by deleting vertices and edges.
- G_1 is an **induced** subgraph of G_2 if we only delete vertices.
- G_1 is a **spanning** subgraph of G_2 if we only delete edges.
- Typical question: does G contain a given graph H?

Basic Questions

- Subgraph Containment
- Short-Long Paths
- Interesting Sets
- Coloring

- A Hamiltonian Path is a path that visits every vertex exactly once.
- An Eulerian Walk is a walk (path that may repeat vertices) that visits every edge exactly once.
- Typical question: find the shortest/longest path between two vertices.
- Related: Is G Hamiltonian? Eulerian?

Basic Questions

- Subgraph Containment
- Short-Long Paths
- Interesting Sets
- Coloring

- An independent set is a set of vertices inducing no edges.
- A vertex cover is a set of vertices that intersects all edges.
- A dominating set is a set of vertices that is adjacent to all vertices.
- ...
- Typical question: Find the smallest/largest set of vertices satisfying some property.

Basic Questions

- Subgraph Containment
- Short-Long Paths
- Interesting Sets
- Coloring

- A coloring is a partitioning of a graph into independent sets.
- Typical question: How many colors do we need to color the vertices of this graph?

Basic Questions

- Subgraph Containment
- Short-Long Paths
- Interesting Sets
- Coloring

- A coloring is a partitioning of a graph into independent sets.
- Typical question: How many colors do we need to color the vertices of this graph?

Many of these questions are **Hard**! Which are easy and for which classes of graphs? This is something we will discuss...

Examples

Find same degree vertices

Problem

Given graph G as an adjacency matrix, design an algorithm that finds two vertices of G with the same degree.

Find same degree vertices

Problem

Given graph G as an adjacency matrix, design an algorithm that finds two vertices of G with the same degree.

Solution:

- For $i \in \{1, \ldots, n\}$
- For $j \in \{i + 1, \ldots, n\}$
  - Check if $\deg(i) = \deg(j)$. If yes, output (i, j).

Complexity?

Find same degree vertices

Problem

Given graph G as an adjacency matrix, design an algorithm that finds two vertices of G with the same degree.

Solution:

- For $i \in \{1, \ldots, n\}$
- For $j \in \{i+1, \ldots, n\}$
  - Check if $\deg(i) = \deg(j)$. If yes, output (i, j).

Complexity? $O(n^3)$, because computing $\deg(i)$ takes time $O(n)$

Find same degree vertices

Problem

Given graph G as an adjacency matrix, design an algorithm that finds two vertices of G with the same degree.

Solution:

- For $i \in \{1, \dots, n\}$
  - Compute $\deg(i)$
- Sort array of degrees D
- For $i \in \{1, \dots, n\}$ check if $D[i] = D[i+1]$

Complexity? $O(n^2)$ time.

Find same degree vertices

Problem

Given graph G as an adjacency matrix, design an algorithm that finds two vertices of G with the same degree.

Solution:

- For $i \in \{1, \ldots, n\}$
    - Compute $\deg(i)$
- Sort array of degrees D
- For $i \in \{1, \ldots, n\}$ check if $D[i] = D[i+1]$

Complexity? $O(n^2)$ time.
$O(n^2)$ time is optimal (why?). However, we are now using $O(n)$ space, whereas previously we were using $O(1)$...

Find vertices at distance at least 3

Problem

Given graph G as an adjacency matrix, find two vertices u, v (if they exist) which are at distance at least 3.

Find vertices at distance at least 3

Problem

Given graph G as an adjacency matrix, find two vertices u, v (if they exist) which are at distance at least 3.

Solution:

- For $i \in \{1, \dots, n\}$
- For $j \in \{i+1, \dots, n\}$
  - if $ij \notin E$ and $N(i) \cap N(j) = \emptyset$ then output (i, j)
- Otherwise, Output Not Found!

Complexity?

Find vertices at distance at least 3

Problem

Given graph G as an adjacency matrix, find two vertices u, v (if they exist) which are at distance at least 3.

Solution:

- For $i \in \{1, \ldots, n\}$
- For $j \in \{i+1, \ldots, n\}$
  - if $ij \notin E$ and $N(i) \cap N(j) = \emptyset$ then output (i, j)
- Otherwise, Output Not Found!

Complexity? $O(n^3)$ assuming $N(i) \cap N(j) = \emptyset$ can be checked in $O(n)$ (how?)

Find vertices at distance at least 3

Problem

Given graph G as an adjacency matrix, find two vertices u, v (if they exist) which are at distance at least 3.

Solution:

- For $i \in \{1, \ldots, n\}$
- For $j \in \{i+1, \ldots, n\}$
    - if $ij \notin E$ and $N(i) \cap N(j) = \emptyset$ then output (i, j)
- Otherwise, Output Not Found!

Complexity? $O(n^3)$ assuming $N(i) \cap N(j) = \emptyset$ can be checked in $O(n)$ (how?)Can we do it in sub-cubic time???

Find vertex cover of size 2

Problem

Given graph G as an adjacency matrix, find two vertices u, v (if they exist) which touch all edges of G.

Find vertex cover of size 2

Problem

Given graph G as an adjacency matrix, find two vertices u, v (if they exist) which touch all edges of G.

Solution:

- For $i \in \{1, \ldots, n\}$
- For $j \in \{1, \ldots, n\}$
    - Check if there is an edge not incident on either i nor j.

  Complexity?

Find vertex cover of size 2

Problem

Given graph G as an adjacency matrix, find two vertices u, v (if they exist) which touch all edges of G.

Solution:

- For $i \in \{1, \ldots, n\}$
- For $j \in \{1, \ldots, n\}$
  - Check if there is an edge not incident on either i nor j.

  Complexity?

$O(n^4)$ in the obvious implementation. Better?

Find vertex cover of size 2

Problem

Given graph G as an adjacency matrix, find two vertices u, v (if they exist) which touch all edges of G.

First, let us find is G has vertex cover of size 1 in $O(n^2)$ (instead of $O(n^3)$)
Solution:

- Find an edge ij
- Check if $\{i\}$ is a vertex cover. If yes, output i
- If not, check if $\{j\}$ is a vertex cover. If yes, output j
- Otherwise, output No

Find vertex cover of size 2

Problem

Given graph G as an adjacency matrix, find two vertices u, v (if they exist) which touch all edges of G.

First, let us find is G has vertex cover of size 1 in $O(n^2)$ (instead of $O(n^3)$)
Solution:

- Find an edge ij
- Check if $\{i\}$ is a vertex cover. If yes, output i
- If not, check if $\{j\}$ is a vertex cover. If yes, output j
- Otherwise, output No

$O(n^2)$ time

Find vertex cover of size 2

Problem

Given graph G as an adjacency matrix, find two vertices u, v (if they exist) which touch all edges of G.

Now, work recursively:
Solution:

- Find an edge ij.
- Let G_1 be the graph obtained from G if we remove i. Check if G_1 has a vertex cover of size 1. If yes, output $\{i\}$ plus the vertex cover of G_1.
- If not, let G_2 be the graph obtained from G if we remove j. If G_2 has a vertex cover of size 1, output $\{j\}$ plus the vertex cover of G_2.
- Otherwise, output No

Complexity: $O(n^2)$

Digraph Transpose I

Problem

Given digraph G as an adjacency matrix, compute the adjacency matrix of G^T, which is the digraph where the direction of all arcs is reversed.

Digraph Transpose I

Problem

Given digraph G as an adjacency matrix, compute the adjacency matrix of G^T, which is the digraph where the direction of all arcs is reversed.

Solution: (easy!)

- For each $i, j \in \{1, \ldots, n\}$ set $A'[i, j] = A[j, i]$, where A is the original matrix.

Digraph Transpose II

Problem

Given digraph G as adjacency lists, compute the adjacency list representation of G^T, which is the digraph where the direction of all arcs is reversed.

Digraph Transpose II

Problem

Given digraph G as adjacency lists, compute the adjacency list representation of G^T, which is the digraph where the direction of all arcs is reversed.

Wait! What is the adjacency list representation of a digraph?

- For each $v \in V$ we have a list $N^+(v)$ with all the out-neighbors of v.

Digraph Transpose II

Problem

Given digraph G as adjacency lists, compute the adjacency list representation of G^T, which is the digraph where the direction of all arcs is reversed.

- Initialize with empty lists for all $v \in V$
- For each $v \in V$
  - For each $u \in N^+(v)$ in the original graph
  - . . . Add v to the list of outneighbors of u in the new graph.

Complexity: $O(m + n)$ time, but **NB** adjacency lists are not sorted (we never promised they were!)

Digraph degree I

Problem

Given digraph G in adjacency list represenation and a vertex v, compute the outdegree of v.

Digraph degree I

Problem

Given digraph G in adjacency list represenation and a vertex v, compute the outdegree of v.

Solution (easy):

- Output $|N^+(v)|$

Complexity: $O(\deg^+(v))$ or $O(1)$ (depending on how list is stored)

Digraph degree II

Problem

Given digraph G in adjacency list represenation and a vertex v, compute the indegree of v.

Digraph degree II

Problem

Given digraph G in adjacency list represenation and a vertex v, compute the indegree of v.

Solution (less easy):

- $c := 0$
- For each $u \in V \setminus \{v\}$
  - If $v \in N^+(u)$ then $c{+}{+}$
- Output c

Complexity: $O(n+m)$, as we have to traverse all the lists (deciding if $v \in N^+(u)$ takes time $O(|N^+(u)|)$).