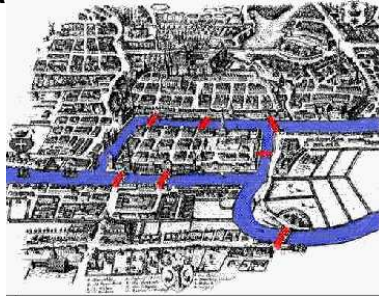


Graphs and Graph Representation



- 1736 L. Euler asks about a “touristic” question:
- Street or Computer networks
- Railway connections (space and time)
- Citation or Coauthorship \rightsquigarrow social networks
- Job precedences \rightsquigarrow scheduling problems
- Values and arithmetic operations \rightsquigarrow compiler construction
- ...

1.1.2 Degrees

out-degree(v): $|\{(v, u) \in E\}|$

in-degree(v): $|\{(u, v) \in E\}|$

1 Mathematical View: Just Sets and Pairs

Often terminology is already reason enough for a graph model

1.1 Directed Graphs

$G = (V, E)$ describes a **directed graph** with **vertex set** V and **edge set** $E \subseteq V \times V$.

Convention: $n = |V|$ and $m = |E|$

1.1.1 Special Cases (usually disallowed)

Multigraphs: parallel edges allowed, i.e. E is a multiset.

self loops: edges (v, v)

Bidirected graph: $\forall (u, v) \in E : (v, u) \in E$

1.2 Undirected Graphs

streamlined description of a bidirected graph where **edges** are **two element vertex sets**.

$\{u, v\} \leftrightarrow (u, v), (v, u)$

degree(v) = in-degree(v) = out-degree(v)

What about self loops?

1.3 Subgraphs

$G' = (V', E')$ is **subgraph** of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$

G' is **vertex induced** by V' if $E' = \{(u, v) \in E : u \in V', v \in V'\}$

1.4 Associated Information

edge weights $w : E \rightarrow \mathbb{R}$

Example: Length or traffic capacity of a street,

we will see more things like **node potentials**, **node/vertex colors**, . . .

1.5 Paths

A **path** $p = \langle v_0, \dots, v_k \rangle$

of **length** k connects nodes v_0 and v_k if

subsequent nodes in p are connected by edges in E , i.e.,

$e_1 = (v_0, v_1) \in E, e_2 = (v_1, v_2) \in E, \dots, e_k = (v_{k-1}, v_k) \in E$.

edge representation: $p = \langle e_1, \dots, e_k \rangle$.

simple path: v_0, \dots, v_k **different**.

Nonsimple paths are usually called **walks**.

In **weighted graphs**, **length** is $\sum_{i=1}^k w(e_i)$.

bottleneck weight: $\min_{i=1}^k w(e_i)$.

Applications: What is the shortest route from Saarbrücken to Paris?

How much traffic can it carry?

1.6 Cycles

Cycles are paths that share the first and the last node

A cycle is **simple** if all other nodes are different.

Hamiltonian cycle: simple cycle that visits all **nodes**.

Euler cycle: cycle (cyclic walk) that visits all **edges** once

\rightsquigarrow back to the bridge problem. But we need one more term.

1.7 Connectedness

G is strongly connected $\Leftrightarrow \forall u, v \in V : \exists \text{ path } p : p \text{ connects } u \text{ and } v$.

drop the “strongly” for undirected graphs

A **connected component** is a maximal connected vertex induced subgraph

1.9 Trees

An undirected graph is a **tree** if there is **exactly** one path between any pair of nodes.

Exercise: Prove that the following properties are equivalent

1. G is a tree.
2. G is connected and has exactly $n - 1$ edges.
3. G is connected and contains no cycles.

Forest: undirected acyclic graph

Directed acyclic graph (DAG): no cycles.

Exercise: How many edges are possible?

1.8 Euler Tours

Theorem 1. G is **Eulerian** (has an Euler tour) if and only if G is **connected** and every node has **even degree**.

Proof: Necessity: connectedness is clear.

Consider a node v with degree $2k - 1$.

The tour is stuck when it visits v the k -th time.

Sufficiency:

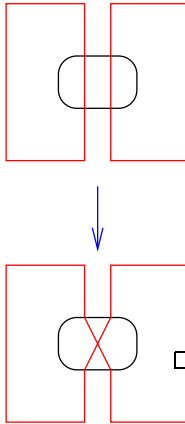
Step 1, Construct **Euler Partitioning** of E into a **set of cycles**:

Greedily build any walk. Remove visited edges.

When you get stuck, you have closed a cycle.

Step 2, Glue the cycles together.

Possible since any cycle intersects some other cycle.



1.10 Directed Trees?

we need a special **root** node r .

$\forall v \in V$ exactly one $r - v$ path or

$\forall v \in V$ exactly one $v - r$ path

CS trees have the root at the top.

a **parent** is immediately above its **children** or **successors** which are **siblings**.
(ancestors, . . .) Nodes without children are **leaves**.

Nonroot nonleaf nodes are **interior**.

ordered trees: order of children matters. Example: search trees.

2 Graph Representation

we mostly represent undirected graphs as bidirected graphs

~> focus on directed graphs

Overview:

- Operations are what matters
- A trivial representation
- Arrays
- Linked Lists
- Matrices
- Implicit Representation
- Discussion

Example: Recognizing DAGs

while $\exists v \in V : \text{out-degree}(v) = 0$ **do delete** v

if $V = \emptyset$ **then output** “ G is a DAG”

else output “ G contains a cycle”

// to find a cycle, pick any node and follow edges

works because we never destroy or introduce cycles

Exercise: implement it in time $\mathcal{O}(m + n)$

2.1 Operations

Goal: $\mathcal{O}(1)$ time for all elementary operations

Access associated information. Use arrays. Perhaps hash tables.

One reason for $V = 1..n$.

Navigation: Given v find outgoing edges. (We may also want incoming edges.)

Edge queries: $(u, v) \in E$? Adjacency matrices, hash tables.

Reverse access: Given (u, v) find (v, u)

Construction conversion and output ($\mathcal{O}(m + n)$ time)

Update: Insert/delete nodes/edges. This is the hard part.

2.2 Edge Sequence Representation

Sequence of node pairs (or triples with edge weight)

+ **Compact**

+ Good for I/O

– Almost no useful operations except **scanning** all edges

Not so bad with some additional node arrays.

Examples: Find isolated nodes, several MST algorithms, **conversion**.

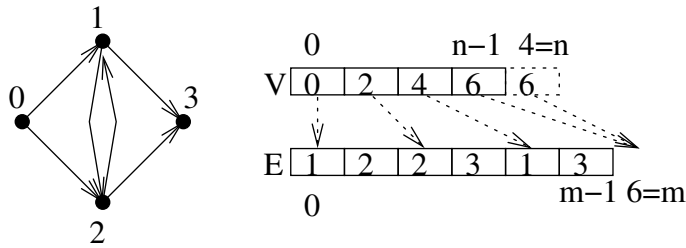
2.3 Adjacency Arrays

$V = 0..n - 1$

Edge array E stores **targets**
grouping edges leaving a node

Node array V stores index of first outgoing edge

dummy entry $V[n]$ stores $m + 1$



Example: $\text{out-degree}(v) = V[v + 1] - V[v]$

Integrated Solution

Function $\text{adjacencyArray}(\text{EdgeList})$

$V = \langle 0, \dots, 0 \rangle : \text{Array}[0..n]$ of \mathbb{N}

foreach $(u, v) \in \text{EdgeList}$ **do** $V[u]++$ // count

for $v := 1$ **to** n **do** $V[v] += V[v - 1]$ // prefix sums

foreach $(u, v) \in \text{EdgeList}$ **do** $E[V[u] - -] = v$ // place

return (V, E)

Example on Blackboard

2.4 Edge List \rightarrow Adjacency Array

A Reminder: Sorting Small Integers

// make b a sorted permutation of a

Procedure $\text{KSortArray}(a, b : \text{Array}[1..n]$ of $0..K-1$)

$c = \langle 0, \dots, 0 \rangle : \text{Array}[0..K-1]$ of \mathbb{N} // counters for each bucket

for $i := 1$ **to** n **do** $c[a[i]]++$ // Count bucket sizes

$C := 0$

for $k := 0$ **to** $K - 1$ **do** $(C, c[k]) := (C + c[k], C)$ // Store $\sum_{i < k} c[i]$ in C

for $i := 1$ **to** n **do** // Distribute $a[i]$

$b[c[a[i]]] := a[i]$

$c[a[i]]++$

Idea: Sort by starting node

Operations for Adjacency Arrays

Navigation: easy. we will see several applications

Edge weights: E becomes array of records

Incoming Edges: another (reverse) edge array

Delete Edges: explicit end indices

Batched Updates: rebuild

2.5 Adjacency Lists

store (doubly linked) **list** of adjacent edges for each node
 (a bit more restricted with singly linked lists)

- + easy edge **insertion**
- + easy edge **deletion** (order preserving)
- more space (factor up to 3) than adj. arrays
- more cache faults than adj. arrays

Enhancing Adjacency Lists

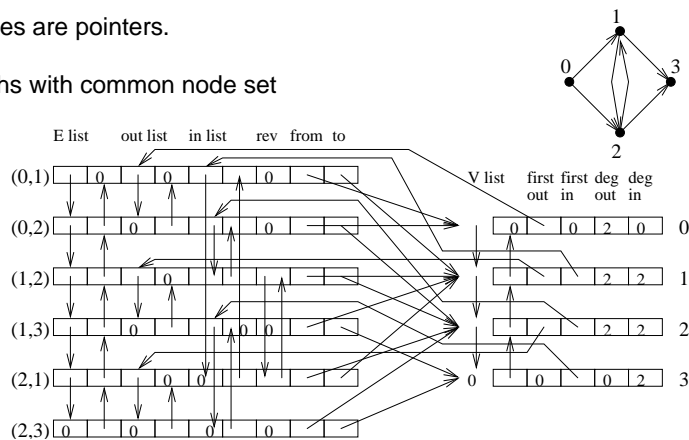
For **reverse** edge access or edge **weight updates in undirected** graphs:
 explicit **edge objects**

- stores weights
- pred/succ-pointers for all adjacency lists containing this edge
- reverse pointer for directed graphs
- cute trick: store only XOR of incident nodes

Node insertion/deletion

mark as deleted \vee swap in node n \vee list of nodes
 \Rightarrow node handles are pointers.

Problem: graphs with common node set



$\approx 8\times$ more space than vanilla adjacency arrays

2.6 Customization

Customize data structure for application for maximum speed/compactness.

Software Engineering nightmare

Separating algorithm from their representation may be a way out

Example: Recognizing dags

while $\exists v \in V : \text{out-degree}(v) = 0$ **do** delete v

- Use adjacency array storing **incoming edges**
- Store **out-degrees**
- Maintain a stack of out-degree zero nodes
- delete (u, v) by decrementing $\text{out-degree}(u)$
- no explicit node deletion**
- At the end test whether all nodes have $\text{out-degree}(0)$

Example where graph theory helps LA

Problem: solve $\mathbf{B}\mathbf{x} = \mathbf{c}$ Consider $G = (1..n, E = \{(i, j) : B_{ij} \neq 0\})$

Assume G has two connected components \Rightarrow

we swap rows and cols such that

$$\begin{pmatrix} \mathbf{B}_1 & \mathbf{0} \\ \mathbf{0} & \mathbf{B}_2 \end{pmatrix} \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{c}_1 \\ \mathbf{c}_2 \end{pmatrix} .$$

Exercise: What if G is a DAG?

2.7 Adjacency Matrix

$A \in \{0, 1\}^{n \times n}$ with $A(i, j) = [(i, j) \in E]$

- + space efficient for very dense matrices
- space **inefficient** otherwise. Exercise: what should "very dense" mean?
- + Easy edge queries
- Slow navigation (except for dense graphs)
- ++ joins linear algebra and graph theory

Example: $\mathbf{C} = \mathbf{A}^k$. $\mathbf{C}_{ij} = \#$ k -edge paths from i to j

Exercise: count $\leq k$ -edge paths

Important accelerators:

$\mathcal{O}(\log k)$ mat-mults for mat-power

matrix multiplication in subcubic time, e.g., Strassen's algorithm

2.8 Implicit Representation

Compact representations of possibly very dense graphs

Implement algorithms **directly** using this representation

Example: Connectedness of Interval Graphs

$$V = \{[a_1, b_1], \dots, [a_n, b_n]\}$$

$$E = \{\{[a_i, b_i], [a_j, b_j]\} : [a_i, b_i] \text{ and } [a_j, b_j] \text{ overlap}\}$$

Idea: **sweep** intervals from left to right. The number of overlapping intervals may never drop to zero.

Function $isConnected(L : SortedListOfIntervalEndPoints) : \{0, 1\}$

remove first element of L

overlap := 1

foreach $p \in L$ **do**

if $overlap = 0$ **return** 0

if p is a start point **then** $overlap++$

else $overlap--$ *// end point*

$\mathcal{O}(n \log n)$ algorithm for up to $\mathcal{O}(n^2)$ edges! Exercise: find all connected components