Graphs

Slides by Christopher M. Bourke
Instructor: Berthe Y. Choueiry

Spring 2006

Computer Science & Engineering 235
Introduction to Discrete Mathematics
Sections 8.1-8.5 of Rosen
cse235@cse.unl.edu

Introduction I

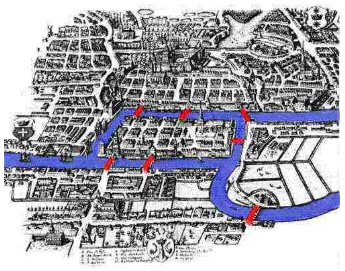
Graph theory was introduced in the 18th century by Leonhard Euler via the *Königsberg bridge problem*.

In Königsberg (old Prussia), a river ran through town that created an island and then split off into two parts.

Seven bridges were built so that people could easily get around.

Euler wondered, is it possible to walk around Königsberg, crossing every bridge exactly once?

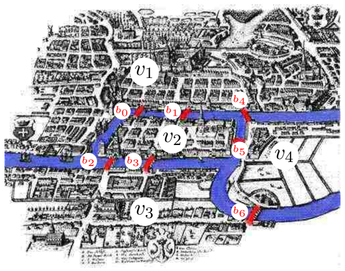
Introduction II



Introduction III

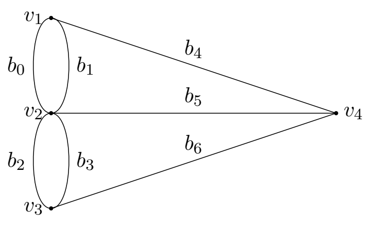
To solve this problem, we need to model it mathematically.

Specifically, we can define a *graph* whose vertices are the land areas and whose edges are the bridges.



Introduction IV

The question now becomes, does there exist a path in the following graph such that every edge is traversed exactly once?



Definitions I

Definition

A *simple graph* $G = (V, E)$ is a 2-tuple with

- $V = \{v_1, v_2, \ldots, v_n\}$ – a finite set of vertices.
- $E = V \times V = \{e_1, e_2, \ldots, e_m\}$ – an unordered set of edges where each $e_i = (v, v')$ is an unordered pair of vertices, $v, v' \in V$.

Since V and E are sets, it makes sense to consider their cardinality. As is standard, $|V| = n$ denotes the number of vertices in G and $|E| = m$ denotes the number of edges in G.

Definitions II

- A *multigraph* is a graph in which the edge set E is a multiset. Multiple distinct (or *parallel*) edges can exist between vertices.
- A *pseudograph* is a graph in which the edge set E can have edges of the form (v, v) called *loops*
- A *directed graph* is one in which E contains *ordered* pairs. The orientation of an edge (v, v') is said to be "from v to v'".
- A *directed multigraph* is a multigraph whose edges set consists of ordered pairs.

Definitions III

If we look at a graph as a *relation* then, among other things,

- Undirected graphs are *symmetric*.
- Non-pseudographs are *irreflexive*.
- Multigraphs have nonnegative integer entries in their matrix; this corresponds to *degrees* of relatedness.

Other types of graphs can include labeled graphs (each edge has a uniquely identified label or weight), colored graphs (edges are colored) etc.

Terminology
Adjacency

For now, we will concern ourselves with simple, undirected graphs. We now look at some more terminology.

Definition

Two vertices u, v in an undirected graph $G = (V, E)$ are called *adjacent* (or *neighbors*) if $e = (u, v) \in E$.

We say that e is *incident with* or *incident on* the vertices u and v.

Edge e is said to *connect* u and v.

u and v are also called the *endpoints* of e.

Terminology
Degree

Definition

The degree of a vertex in an undirected graph $G = (V, E)$ is the number of edges incident with it.

The degree of a vertex $v \in V$ is denoted

$$\deg(v)$$

In a multigraph, a loop contributes to the degree twice.

A vertex of degree 0 is called *isolated*.

Terminology
Handshake Theorem

Theorem

Let $G = (V, E)$ be an undirected graph. Then

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

The handshake lemma applies even in multi and pseudographs.

proof By definition, each $e = (v, v')$ will contribute 1 to the degree of each vertex, $\deg(v), \deg(v')$. If $e = (v, v)$ is a loop then it contributes 2 to $\deg(v)$. Therefore, the total degree over all vertices will be twice the number of edges. $\square$

Terminology
Handshake Lemma

Corollary

An undirected graph has an even number of vertices of odd degree.

Terminology - Directed Graphs I

In a directed graph (digraph), $G = (V, E)$, we have analogous definitions.

- Let $e = (u, v) \in E$.
- u is *adjacent to* or *incident on* v.
- v is *adjacent from* or *incident from* u.
- u is the *initial vertex*.
- v is the *terminal vertex*.
- For a loop, these are the same.

Terminology - Directed Graphs II

We make a distinction between incoming and outgoing edges with respect to degree.

- Let $v \in V$.
- The *in-degree* of v is the number of edges incident on v

$$\deg^-(v)$$

- The *out-degree* of v is the number of edges incident from v.

$$\deg^+(v)$$

Terminology - Directed Graphs III

Every edge $e = (u, v)$ contributes 1 to the out-degree of u and 1 to the in-degree of v. Thus, the sum over all vertices is the same.

Theorem

Let $G = (V, E)$ be a directed graph. Then

$$\sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v) = |E|$$

More Terminology I

A *path* in a graph is a sequence of vertices,

$$v_1 v_2 \cdots v_k$$

such that $(v_i, v_{i+1}) \in E$ for all $i = 1, \ldots, k - 1$.

We can denote such a path by $p : v_1 \rightsquigarrow v_k$.

The *length* of p is the number of edges in the path,

$$|p| = k - 1$$

More Terminology II

A *cycle* in a graph is a path that begins and ends at the same vertex.

$$v_1 v_2 \cdots v_k v_1$$

Cycles are also called *circuits*.

We define paths and cycles for directed graphs analogously.

A path or cycle is called *simple* if no vertex is traversed more than once. From now on we will only consider simple paths and cycles.

Classes Of Graphs

- Complete Graphs – Denoted K_n are simple graphs with n vertices where *every* possible edge is present.
- Cycle Graphs – Denoted C_n are simply cycles on n vertices.
- Wheels – Denoted W_n are cycle graphs (on n vertices) with an additional vertex connected to all other vertices.
- n-cubes – Denoted Q_n are graphs with 2^n vertices corresponding to each bit string of length n. Edges connect vertices whose bit strings differ by a single bit.
- Grid Graphs – finite graphs on the $\mathbb{N} \times \mathbb{N}$ grid.

Bipartite Graphs

Definition

A graph is called *bipartite* if its vertex set V can be partitioned into two disjoint subsets L, R such that no pair of vertices in L (or R) is connected.

We often use $G = (L, R, E)$ to denote a bipartite graph.

Bipartite Graphs

Theorem

A graph is bipartite if and only if it contains no odd-length cycles.

Another way to look at this theorem is as follows. A graph G can be *colored* (here, we color vertices) by at most 2 colors such that no two adjacent vertices have the same color if and only if G is bipartite.

Bipartite Graphs

A bipartite graph is complete if every $u \in L$ is connected to every $v \in R$. We denote a complete bipartite graph as

$$K_{n_1,n_2}$$

which means that $|L| = n_1$ and $|R| = n_2$.

Examples?

Decomposing & Composing Graphs I

We can (partially) decompose graphs by considering *subgraphs*.

Definition

A *subgraph* of a graph $G = (V, E)$ is a graph $H = (V', E')$ where

- $V' \subseteq V$ and
- $E' \subseteq E$.

Subgraphs are simply *part(s)* of the original graph.

Decomposing & Composing Graphs II

Conversely, we can combine graphs.

Definition

The *union* of two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_1, E_1)$ is defined to be $G = (V, E)$ where

- $V = V_1 \cup V_2$ and
- $E = E_1 \cup E_2$.

Data Structures I

A graph can be implemented as a data structure using one of three representations:

1. Adjacency list (vertices to list of vertices)
2. Adjacency matrix (vertices to vertices)
3. Incidence matrix (vertices to edges)

These representations *can greatly affect* the running time of certain graph algorithms.

Data Structures II

Adjacency List – An adjacency list representation of a graph $G = (V, E)$ maintains $|V|$ linked lists. For each vertex $v \in V$, the head of the list is v and subsequent entries correspond to adjacent vertices $v' \in V$.

Example

Data Structures III

What is the associated graph of the following adjacency list?

$$\begin{tikzcd}
\boxed{v_0} \arrow[r] & \boxed{v_2} \arrow[r] & \boxed{v_3} \arrow[r] & \boxed{v_4} & \\
\boxed{v_1} \arrow[r] & \boxed{v_0} \arrow[r] & \boxed{v_2} & & \\
\boxed{v_2} \arrow[r] & \boxed{v_0} \arrow[r] & \boxed{v_1} \arrow[r] & \boxed{v_3} \arrow[r] & \boxed{v_4} \\
\boxed{v_3} \arrow[r] & \boxed{v_1} \arrow[r] & \boxed{v_4} & & \\
\boxed{v_4} \arrow[r] & \boxed{v_1} & & &
\end{tikzcd}$$

Data Structures IV

- **Advantages:** Less storage
- **Disadvantages:** Adjacency look up is $\mathcal{O}(|V|)$, extra work to maintain vertex ordering (lexicographic)

Adjacency Matrix – An adjacency matrix representation maintains an $n \times n$ sized matrix with entries

$$a_{i,j} = \begin{cases} 0 & \text{if } (v_i, v_j) \notin E \\ 1 & \text{if } (v_i, v_j) \in E \end{cases}$$

for $0 \leq i, j \leq (n-1)$.

Data Structures V

Example

For the same graph in the previous example, we have the following adjacency matrix.

$$\begin{bmatrix} 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

- **Advantages:** Adjacency/Weight look up is constant
- **Disadvantages:** Extra storage

Data Structures VI

The entry of 1 for edges $e = (v_i, v_j)$ can be changed to a weight function $\mathrm{wt} : E \to \mathbb{N}$. Alternatively, entries can be used to represent pseudographs.

Note that either representation is equally useful for directed and undirected graphs.

Sparse vs Dense Graphs

We say that a graph is *sparse* if $|E| \in \mathcal{O}(|V|)$ and *dense* if $|E| \in \mathcal{O}(|V|^2)$.

A complete graph K_n has precisely $|E| = \frac{n(n-1)}{2}$ edges.

Thus, for sparse graphs, Adjacency lists tend to be better while for dense graphs, adjacency matrices are better *in general*.

Graph Isomorphism I

An *isomorphism* is a bijection (one-to-one and onto) that preserves the *structure* of some object.

In some sense, if two objects are isomorphic to each other, they are essentially the same.

Most properties that hold for one object hold for any object that it is isomorphic to.

An isomorphism of graphs preserves adjacency.

Graph Isomorphism II

Definition

Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are *isomorphic* if there exists a bijection

$$\varphi : V_1 \to V_2$$

such that $(v, v') \in E_1$ if and only if

$$\big(\varphi(v), \varphi(v')\big) \in E_2$$

for all vertices $v \in V_1$.

If G_1 is isomorphic to G_2 we use the notation

$$G_1 \cong G_2$$

Graph Isomorphism III

Lemma

Isomorphism of graphs is an equivalence relation.

Proof?

Graph Isomorphism I

Computability

Problem

Given: *Two graphs, G_1, G_2.*
Question: *Is $G_1 \cong G_2$?*

The obvious way of solving this problem is to simply try to find a bijection that preserves adjacency. That is, search through all $n!$ of them.

Wait: Do we really need to search all $n!$ bijections?

There are smarter, but more complicated ways. However, the best known algorithm for general graphs is still only

$$\mathcal{O}(\exp(\sqrt{n \log n}))$$

Graph Isomorphism II

Computability

The graph isomorphism problem is of great theoretical interest because it is believed to be a problem of 'intermediate complexity.'

Conversely, it is sometimes easier (though not in general) to show that two graphs are *not* isomorphic.

In particular, it suffices to show that the pair (G_1, G_2) do not have a property that isomorphic graphs should. Such a property is called *invariant* wrt isomorphism.

Graph Isomorphism III

Computability

Examples of invariant properties:

- $|V_1| = |V_2|$
- $|E_1| = |E_2|$
- Degrees of vertices must be preserved.
- Lengths of paths & cycles.

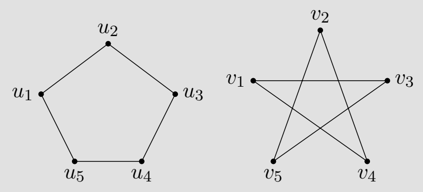
Such properties are a *necessary* condition of being isomorphic, but are not a *sufficient* condition.

Graph Isomorphism I

Example

Example

(8.3.35) Are the following two graphs isomorphic?



Graph Isomorphism II

Example

All of the invariant properties previously mentioned hold.

However, we still need to give an explicit bijection φ if they are isomorphic.

Consider the following bijection.

$$\begin{array}{rcl} \varphi(u_1) & = & v_1 \\ \varphi(u_2) & = & v_3 \\ \varphi(u_3) & = & v_5 \\ \varphi(u_4) & = & v_2 \\ \varphi(u_5) & = & v_4 \end{array}$$

We still need to verify that φ preserves adjacency.

Graph Isomorphism III

Example

The original edges were

$$\begin{array}{lcccl} (u_1, u_2) & \to & (\varphi(u_1), \varphi(u_2)) & = & (v_1, v_3) \in E_2? \\ (u_2, u_3) & \to & (\varphi(u_2), \varphi(u_3)) & = & (v_3, v_5) \in E_2? \\ (u_3, u_4) & \to & (\varphi(u_3), \varphi(u_4)) & = & (v_4, v_2) \in E_2? \\ (u_4, u_5) & \to & (\varphi(u_4), \varphi(u_5)) & = & (v_2, v_4) \in E_2? \\ (u_5, u_1) & \to & (\varphi(u_5), \varphi(u_1)) & = & (v_4, v_1) \in E_2? \end{array}$$

Thus, they *are* isomorphic. Note that there are several bijections that show these graphs are isomorphic.

Using Paths & Cycles in Isomorphisms I

Recall that the lengths of paths & cycles are invariant properties for isomorphisms.

Moreover, they can be used to find potential isomorphisms.

For example, say there is a path of length k in G_1

$$v_0 v_1 \cdots v_k$$

Now consider the degree sequence of each vertex;

$$\deg(v_0), \deg(v_1), \ldots, \deg(v_k)$$

Since both of these properties are invariants, we could try looking for a path (of length k) in G_2 that has the same degree sequence.

Using Paths & Cycles in Isomorphisms II

If we can find such a path, say

$$u_0 u_1 \cdots u_k$$

it may be a good (partial) candidate for an isomorphic bijection.

Connectivity I

An undirected graph is called *connected* if for every pair of vertices, u, v there exists a path connecting u to v.

A graph that is not connected is the union of two or more subgraphs called *connected components*.

We have analogous (but more useful) notions for directed graphs as well.

Definition

A directed graph is *strongly connected* if for every pair of vertices u, v

- There exists $p_1 : u \rightsquigarrow v$ and
- There exists $p_2 : v \rightsquigarrow u$.

Connectivity II

Even if a graph is not strongly connected, it can still be (graphically) "one piece".

Definition

A directed graph is *weakly connected* if there is a path between every two vertices in the underlying undirected graph (i.e. the symmetric closure).

The subgraphs of a directed graph that are strongly connected are called *strongly connected components*.

Such notions are useful in applications where we want to determine what individuals can communicate in a network (here, the notion of *condensation graphs* is useful).

Example?

Counting Paths I

Often, we are concerned as to *how connected* two vertices are in a graph.

That is, how many unique, paths (directed or undirected, but not necessarily simple) there are between two vertices, u, v?

An easy solution is to use matrix multiplication on the adjacency matrix of a graph.

Theorem

Let G be a graph with adjacency matrix A. The number of distinct paths of length r from $v_i \rightsquigarrow v_j$ equals the entry a_{ij} in the matrix A^r.

The proof is a nice proof by induction.

Euler Paths & Cycles I

Recall the Königsberg Bridge Problem. In graph theory terminology, the question can be translated as follows.

Given a graph G, does there exist a cycle traversing every edge exactly once? Such a cycle is known as an *Euler cycle*.

Definition

An *Euler cycle* in a graph G is a cycle that traverses every edge exactly once. An *Euler path* is a path in G that traverses every edge exactly once.

Euler Paths & Cycles II

Theorem (Euler)

A graph G contains an Euler cycle if and only if every vertex has even degree.

This theorem also holds more generally for multigraphs.

Euler Paths & Cycles III

Therefore, the answer to the Königsberg Bridge problem is, *no*, does there does not exist an Euler cycle. In fact, there is not even an Euler path.

Theorem

A graph G contains an Euler path (not a cycle) if and only if it has exactly two vertices of odd degree.

Constructing Euler Cycles I

Constructing Euler paths is simple. Given a (multi)graph G, we can start at an arbitrary vertex.

We then find any arbitrary cycle c_1 in the graph.

Once this is done, we can look at the *induced subgraph*; the graph created by eliminating the cycle c_1.

We can repeat this step (why?) until we have found a collection of cycles that involves every edge; $c_1, \ldots, c_k$.

Constructing Euler Cycles II

The Euler cycle can then be constructed from these cycles as follows. Starting with c_1, traverse the cycle until we reach a vertex in common with another cycle, c_i; then we continue our tour on this cycle until we reach a vertex in common with another cycle, etc.

We are always guaranteed a way to return to the original vertex by completing the tour of each cycle.

Hamiltonian Paths & Circuits I

Euler cycles & paths traverse every *edge* exactly once.

Cycles and paths that traverse every *vertex* exactly once are *Hamiltonian* cycles and paths.

Definition

A path $v_0, v_1, \ldots, v_n$ in a graph $G = (V, E)$ is called a *Hamiltonian Path* if $V = \{v_0, \ldots, v_n\}$ and $v_i \neq v_j$ for $i \neq j$. A *Hamiltonian* cycle is a Hamiltonian path with $(v_n, v_0) \in E$.

Hamiltonian Paths & Circuits II

Exercise

Show that K_n has a Hamiltonian Cycle for all $n \geq 3$.

Hamiltonian Paths & Circuits III

For general graphs, however, there is no known simple necessary and sufficient condition for a Hamiltonian Cycle to exist.

This is a stark contrast with Euler Cycles: we have a simple, efficiently verifiable condition for such a cycle to exist.

There are no known efficient algorithms for determining whether or not a graph G contains a Hamiltonian Cycle.

This problem is NP-complete. When the edges are weighted, we get Traveling Salesperson Problem, which is NP-hard.

Hamiltonian Paths & Circuits IV

Nevertheless, there are sufficient conditions.

Theorem (Dirac Theorem)

If G is a graph with n vertices with $n \geq 3$ such that the degree of every vertex in G is at least $n/2$, then G has a Hamiltonian cycle.

Theorem (Ore's Theorem)

If G is a graph with n vertices with $n \geq 3$ such that $\deg(u) \geq \deg(v) \geq n$ for every pair of nonadjacent vertices u, v in G then G has a Hamiltonian cycle.

Application: Gray Codes I

Electronic devices often report state by using a series of switches which can be thought of as bit strings of length n. (corresponding to 2^n states).

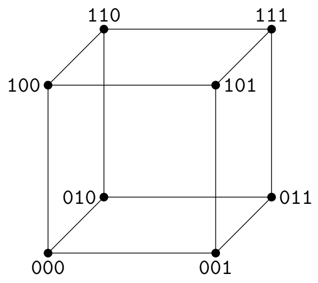
If we use the usual binary enumeration, a state change can take a long time—going from 01111 to 10000 for example.

It is much better to use a scheme (a *code*) such that the change in state can be achieved by flipping a *single* bit.

A *Gray Code* does just that.

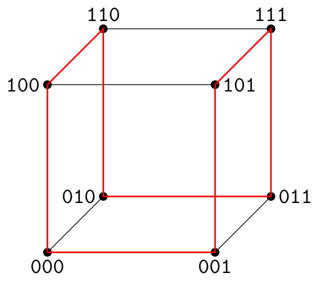
Recall Q_n, the cube graph.

Application: Gray Codes II



Each edge connects bit strings that differ by a single bit. To define a Gray Code, it suffices to find a Hamiltonian cycle in Q_n.

Application: Gray Codes III



A Hamiltonian Path

Application: Gray Codes IV

So our code is as follows.

000
001
101
111
011
010
110
100