Introduction to Graph Theory

Tandy Warnow

January 20, 2017

Graphs
Tandy Warnow

Graphs

A graph $G = (V, E)$ is an object that contains a vertex set V and an edge set E.
We also write $V(G)$ to denote the vertex set of G and $E(G)$ to denote the edge set of G.
Edges are pairs of vertices.
Most graphs we will discuss will not have directions on the edges.
However, **directed graphs** have directions on the edges (so an edge $u \to v$ from u to v is not the same as an edge $v \to u$ from v to u).
We often refer to directed graphs as **digraphs**.
Most graphs are finite (so $|V| < \infty$), but sometimes graphs are infinite.
In this class we'll only talk about finite graphs and unless otherwise specified the graphs will be undirected.

Notation for edges in undirected graphs

For undirected graphs, an edge between two vertices has no direction.
There are several ways to write an edge between vertices u and v in an undirected graph.
I tend to use (u, v) to denote the edge between u and v, and Margaret Fleck writes uv. You may well find other UIUC professors writing edges using $\{u, v\}$!

1. $uv = vu$
2. $(u, v) = (v, u)$
3. $\{u, v\}$

Edges in directed graphs

In a directed graph, the orientation of the edge has to indicated, and so we usually write this as $u \to v$.
In this case u is called the tail and v is called the head.
Sometimes these edges are referred to as **arcs**.

Simple Graphs

A **self-loop** is an edge from a vertex to itself (i.e., edges (v, v)).
Parallel edges are multiple occurances of the same pair (i.e., (u, v) appearing more than once).
A **simple graph** is one that has no parallel edges and no self-loops.

Unless stated otherwise, all graphs in this class should be assumed to be finite, simple, undirected graphs.

Terminology

- For a given graph G, if $(v, w) \in E$, then v and w are said to be **adjacent** or **neighbors**.
- The **degree** of a vertex v in a graph is the number of its neighbors, and is denoted $deg(v)$.
- The **degree sequence** of a graph G is the list of degrees of the vertices in G (note a degree can appear more than once).
- The **neighborhood** of a vertex v, denoted $\Gamma(v)$, is the set of neighbors of v.
- The **endpoints** of an edge $e = (v, w)$ are v and w.
- Two edges are said to be **incident** if they have a common endpoint.
- The **empty graph** has no vertices (and hence obviously no edges).
- Some people say "vertices" and some people say "nodes", and these refer to the same thing. (The singular version of "vertices" is "vertex".)

Paths and walks

- A **path** in a graph is a sequence of distinct vertices so that every two successive vertices are adjacent. Thus, a path can be written as $v_1, v_2, \ldots, v_k$ so that $(v_i, v_{i+1}) \in E$ for all $i = 1, 2, \ldots, k-1$.
- In a directed graph, the edges are all directed, and so $(v_i, v_{i+1}) \in E$ reflects the orientation of the edge.
- The **length** of the path is the number of edges in the path.
- The **distance** from a vertex v to a vertex w is the number of edges in the *shortest* path from v to w in G, and is denoted $d(v, w)$.
- The **diameter** of a graph G is the largest distance between any two vertices in the graph.
- A **walk** is a sequence of vertices (where repeated vertices are allowed), where every two successive vertices are adjacent.

Cycles and circuits

- A **cycle** is a "closed path" (i.e., a sequence of vertices that begins and ends at the same vertex). Thus, a cycle can be written as $v_1, v_2, \ldots, v_k, v_{k+1}$ so that $(v_i, v_{i+1}) \in E$ for all $i = 1, 2, \ldots, k$, $v_{k+1} = v_1$, and otherwise there are no repeated vertices.
- An **acyclic** graph is one that has no cycles.
- A **circuit** in a graph G is a "closed walk" (i.e., a walk that begins and ends at the same vertex). Thus, a circuit can be written as a sequence of vertices $v_1, v_2, \ldots, v_k, v_{k+1}$ so that $(v_i, v_{i+1}) \in E$ for all $i = 1, 2, \ldots, k$, $v_{k+1} = v_1$, and where a vertex can appear more than once.

Connected graphs and connected components

- A graph G is **connected** if for every two distinct vertices v, w there is a path in G starting in v and ending at w. (Note that a graph that contains a single node is by definition connected.)
- The **connected components** of a graph G are the *maximal* subgraphs of G that are connected. (Note: maximal just means that it cannot be enlarged by adding more vertices.)

Cut edges and cut vertices

- A **cut edge** in a graph $G = (V, E)$ is an edge e such that if you delete the edge (but not its endpoints) the graph becomes disconnected.
- A **cut vertex** in a graph $G = (V, E)$ is a vertex v such that if you delete the vertex (and its incident edges) the graph becomes disconnected.

Subgraphs and Induced Subgraphs

- A graph $G' = (V', E')$ is a **subgraph** of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$.
- A graph $G' = (V', E')$ is an **induced subgraph** of $G = (V, E)$ if G' is a subgraph of G and *also* $E' = E \cap (V' \times V')$.

Handshaking Theorem

Theorem: For all graphs $G = (V, E)$, $\sum_{v \in V} deg(v) = 2|E|$.

Proof: Every edge in the graph has two endpoints, and so contributes 2 to this sum.

Eulerian Graphs

A graph $G = (V, E)$ is **Eulerian** if it has a circuit that covers every edge exactly once. Note – vertices can be repeated, but not edges. Such a circuit is called an *Eulerian Circuit*.

Theorem: A connected simple graph $G = (V, E)$ is Eulerian if and only if every vertex in G has even degree. (Note we assume G is simple and finite.)
Proof: (Will be done later in the semester.)
Notes:

- Determining if a graph G is Eulerian can be performed in polynomial time.
- We also are interested in graphs that have *Eulerian walks*, i.e., walks that cover every edge exactly once.
- Finding Eulerian Circuits and Eulerian Walks is used in Assembling Genomes!

Eulerian Graphs, continued

Special graphs

- K_n. The graph on n vertices where every two vertices are adjacent. This is also called the **complete graph** on n vertices.
- $K_{n,m}$. The graph where $V = A \cup B$, $|A| = n, |B| = m$, and $E = \{(v, w) : v \in A, w \in B\}$. This is called a **complete bipartite graph**.
- P_n. The graph with $V = \{v_1, v_2, \ldots, v_n\}$ and $E = \{(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n)\}$. (This is the **path graph** on n vertices.)
- C_n. The graph with $V = \{v_1, v_2, \ldots, v_n\}$ and $E = \{(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n), (v_n, v_1)\}$ (This is the **cycle graph** on n vertices.)
- W_n. The graph with formed by taking C_n and adding an extra vertex w that is adjacent to all the vertices in C_n. This is the **wheel graph** on $n + 1$ vertices (n in the ring outside, and then 1 in the middle).

VERTEX COLORING

Definition: A proper **vertex coloring** of a graph is an assignment of colors to the vertices so that no two adjacent vertices get the same color. Thus, a proper vertex coloring is a function $f : V \to \{1, 2, \ldots, k\}$ such that $f(v) = f(w) \Rightarrow (v, w) \notin E$.

Decision problem:

- Input: Graph $G = (V, E)$ and integer k
- Question: Does G have a proper k-coloring?

Optimization problem: Find the minimum number of colors needed to vertex-color an input graph G. (This number is called the **chromatic number**.)

Construction problem: Find a vertex coloring of input graph G that uses the minimum number of colors.

Note: Vertex Coloring is NP-complete (one of Karp's original 21 problems)

CLIQUE

Definition: A **clique** in $G = (V, E)$ is a subset V_0 of the vertices of G such that every pair of vertices in V_0 are adjacent. In other words, $V_0 \subseteq V$ such that $\forall \{v, w\} \subseteq V_0, (v, w) \in E$.

Decision problem:

- Input: Graph $G = (V, E)$ and integer k
- Question: Does G have a clique of size k?

Optimization problem: Find the size of the largest clique in the input graph G.
Construction problem: Find the largest clique in the input graph G.

NOTE: CLIQUE is NP-complete (one of Karp's original 21 problems)

MAXIMUM INDEPENDENT SET

Definition: An **independent set** in a graph $G = (V, E)$ is a subset of the vertices so that no two vertices in the subset are adjacent. In other words, $V_0 \subseteq V$ such that $\forall \{v, w\} \subseteq V_0, (v, w) \notin E$.

Decision problem:

- Input: Graph $G = (V, E)$ and integer k
- Question: Does G have an independent set of size k?

Optimization problem: Find the size of the largest independent set in the input graph G.
Construction problem: Find the largest independent set in the input graph G.

NOTE: MAXIMUM INDEPENDENT SET is NP-hard (easy reduction from CLIQUE).

MAXIMUM MATCHING

Definition: A **matching** in a graph is a subset of the edges that do not share any endpoints. In other words, $E_0 \subseteq E$ such that $\forall (u, v) \in E_0$ and $\forall (w, x) \in E_0$, if $u = w$ then $v = x$.

Decision problem:

- Input: Graph $G = (V, E)$ and integer k
- Question: Does G have a matching of size k?

Optimization problem: Find the size of the largest matching in the input graph G. This is called the **matching number** of G.
Construction problem: Find the largest matching in the input graph G.

NOTE: MAXIMUM MATCHING can be solved in polynomial time!

VERTEX COVER

Definition: A **vertex cover** in a graph is a set V_0 of vertices so that every edge in the graph has at least one of its endpoints in V_0. In other words, a vertex cover is a set $V_0 \subseteq V$ so that $\forall (v, w) \in E, v \in V_0$ or $w \in V_0$.

Decision problem:

- Input: Graph $G = (V, E)$ and integer k
- Question: Does G have a vertex cover of size k?

Optimization problem: Find the size of the smallest vertex cover in the input graph G.
Construction problem: Find the smallest vertex cover in the input graph G.

NOTE: VERTEX COVER is NP-complete (one of Karp's original problems)

DOMINATING SET

Definition: A **dominating set** in a graph in G is a subset V_0 of the vertices so that every other vertex in G is adjacent to at least one element of V_0. In other words, $V_0 \subseteq V$ such that $\forall v \in V - V_0, \exists w \in V_0$ so that $(v, w) \in E$.

Decision problem:

- Input: Graph $G = (V, E)$ and integer k
- Question: Does G have a dominating set of size k?

Optimization problem: Find the size of the smallest dominating set in the input graph G.
Construction problem: Find the smallest domiating set in the input graph G.

NOTE: Dominating Set is NP-complete.

Travelling salesman

The Travelling Salesman is generally stated as an optimization problem:

- Input: Complete graph $G = (V, E)$ with positive weights on the edges
- Output: Hamilton circuit in G (circuit that visits every vertex exactly once) and has total minimum weight

However we can formulate this as a decision problem, as follows:

- Input: Graph $G = (V, E)$ with positive weights on the edges, and bound B
- Output: Does there exist a circuit in G that visits every vertex at least once, and has total weight at most B?

And of course we could make it into a construction problem.

Note: The decision problem for Travelling Salesman is NP-complete.

Trees: connected acyclic graphs

Thus, trees are just undirected graphs that are connected and acyclic. Nothing more special than that.

More about trees

- If a tree $T = (V, E)$ has n vertices, then it has $n - 1$ edges (proof by induction on n).
- Every tree can be 2-colored.
- The maximum clique size is 2.
- Generally NP-hard problems become polynomial on trees.

Formulating real world problems as graph problems

Consider the following problems, all based on the set of people in this class, and under the assumption where you know who likes who in the class (and you assume this is mutual).

- You want to find a set of people in this class so that between them they know everyone in the class, and the set is as small as possible.
- You want to partition the set of people into subsets so that every two people in any subset both like each other, and make the number of subsets as small as possible.
- You want to pair people off in the class so that they will study together. You want to have as large a number of people able to be in study groups, but you have the following rules: study groups must have pairs of people who like each other, and no person can be in two study groups.
- You want to have a party and invite as many people as you can to it, subject to (a) they all like you, and (b) they all like each other.

Formulating real world problems as graph problems

You want to find a set of people in this class so that between them they know everyone in the class, and the set is as small as possible. Assume you know who likes who.
Challenge: describe this as a graph problem.

Solution:
The graph $G = (V, E)$ is defined by:

- Let V denote all the people in the class.
- Put an edge between v and w if v and w like each other.

We are looking for the smallest $V_0 \subseteq V$ such that $\forall v \in V - V_0$, $\exists w \in V_0$ such that $(v, w) \in E$.

- Does a solution always exist?
- Does it have to be unique?
- What graph problem does this look like?

Describing a real world problem

You want to partition the set of people into subsets so that every two people in any subset both like each other, and make the number of subsets as small as possible.

Solution: The graph $G = (V, E)$ is defined by

- V is the set of people in the class
- E contains (v, w) if and only if v and w like each other

We are looking for a partition of V into a small number of sets so that every one of the sets is a clique.
In other words, we want to write $V = V_1 \cup V_2 \cup \ldots \cup V_k$, where V_i is a clique in G and where k is minimized.

- Does a solution always exist?
- Does it have to be unique?
- What graph problem does this look like?

Things to note

When you formulate a real world problem as a graph problem, you have to:

- Describe the graph precisely. What are the vertices? What are the edges? Do the edges have weights? Use correct terminology (don't be sloppy about language).
- Once the graph is defined, the problem (whether a decision problem, optimization problem, or construction problem) is then defined only in terms of the graph and not in terms of the original problem.

The power in making this formulation is that there are many algorithms (and software) for most natural graph problems, and so you can use those programs to solve your problem.

Many graph problems are NP-hard, but sometimes you have extra structure in your problem that allows you to solve the problem in polynomial time. (For example, MIN VERTEX COLORING is NP-hard, but solvable in polynomial time on trees.)

Class Exercise

You are trying to pick a jury of 12 people from a group of people. The group has already been screened so that everyone is legally acceptable. However, you cannot have any two people on the jury who know each other. So the question is whether or not it's even possible to do this.

Question: Formulate this as a graph problem.

Practice problems

You should be able to do the following tasks:

- Give definitions for NP, Co-NP, P, NP-hard, NP-complete, and Karp reduction
- Know various graph-theoretic terms (tree, degree, path, cycle, induced subgraph, K_n, $K_{n,m}$, etc.) and be able to answer questions based on these terms
- Define decision, optimization, and construction problems defined in today's lecture
- Find solutions to these decision problems on small graphs (with at most 6 vertices)
- Formulate a real world problem as a graph problem
- Write down the proof that 4-colorability is NP-complete (given that 3-colorability is NP-complete)

Isomorphism

Two graphs G and G' are **isomorphic** if they differ only in their vertex names (i.e., renaming the vertices of G makes it G').

Equivalently, $G = (V, E)$ and $G' = (V', E')$ are isomorphic if and only if $\exists$ bijection $f : V \to V'$ such that $(v, w) \in E$ if and only if $(f(v), f(w)) \in E'$.

Determining if two graphs are isomorphic

The only way to be sure if two graphs $G = (V, E)$ and $G' = (V', E')$ are isomorphic is to find the bijection $f : V \to V'$ satisfying $(v, w) \in E$ if and only if $(f(v), f(w)) \in E'$.

This can be difficult.

Proving two graphs not isomoprhic

To prove that G and G' are not isomorphic can also be difficult. However sometimes it's easy. For example, if any of the following properties hold, then G and G' **cannot be isomorphic!**:

- $|V| \neq |V'|$
- $|E| \neq |E'|$
- the multi-set of vertex degrees of G is not the same as the multi-set of vertex degrees of G'.
- The number of components of G is not the same as the number of components of G'.
- G has a cycle of length k and G' does not have a cycle of length k.
- G has a path of length k and G' does not have a path of length k.
- G has some specific subgraph that G' does not have.

Adjacency Matrices

We can represent a graph $G = (V, E)$, where $V = \{v_1, v_2, \ldots, v_n\}$, in several ways. Here we describe one of the two most popular methods, **adjacency matrices**.

Assuming that the graph is undirected and does not have any parallel edges (nor weights on the edges), the adjacency matrix representation is very simple: we define a matrix M where

- $M[i, j] = 1$ if $(v_i, v_j) \in E$
- $M[i, j] = 0$ if $(v_i, v_j) \notin E$

If the graph is simple (so no self-loops), then $M[i, i,] = 0$ for all $i = 1, 2, \ldots, n$.

For undirected graphs, $M[i, j] = M[j, i]$; in other words, the adjacency matrix for an undirected graph is symmetric. For directed graphs, we distinguish between edges from v_i to v_j and from v_j to v_i; hence, we can get asymmetric matrices.

Note that this representation inherently requires $\Theta(n^2)$ space, even for graphs that don't have many edges. But checking if an edge is present is $O(1)$ time.

See https://en.wikipedia.org/wiki/Adjacency_matrix.

Adjacency Lists

An alternative representation of graphs that is sometimes more useful is an Adjacency List.
In an adjacency list, for each vertex in the graph, you have a list of its neighbors. If the graph is undirected, the list for vertex v is the set of all x such that $(v, x) \in E$. For directed graphs, you list only those x such that $v \to x \in E$.
Note that an adjacency list requires $\Theta(m)$ space, where m is the number of edges. This can be much more space efficient than an adjacency matrix for sparse graphs, but some operations take more time (e.g., checking if an edge is present).
See https://en.wikipedia.org/wiki/Adjacency_list.