Lecture 15: Intro to Graphs

built on 2019/06/09 at 21:04:21

1 Graphs

Graphs offer a great way of expressing relationships between pairs of items. Indeed, graphs are one of the most important abstractions in the study of algorithms.

A graph is also known as a network. Graphs can be very important in modeling data, and a large number of problems can be reduced to known graph problems. A graph consists of a set of vertices with connections between them. Graphs can either be directed, with the directed edges (arcs) pointing from one vertex to another, or undirected, with the edges symmetrically connecting vertices. Graphs can have weights or other values associated with either the vertices and/or the edges.

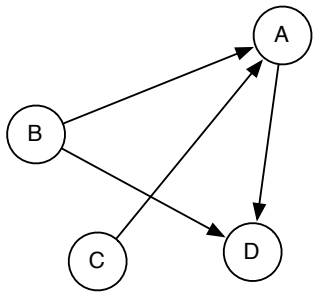
For motivation, we'll outline some of the many applications of graphs:

1. *Road networks.* Vertices are intersections and edges are the road segments between them. Such networks are used by google maps, Bing maps and other map programs to find routes between locations. They are also used for studying traffic patterns. Algorithmic questions we'll cover in this class include TSP, reachability, and shortest paths.
2. *Dependence graphs.* Graphs can be used to represent dependences or precedences among items. Such graphs are often used in large projects in laying out what components rely on other components and used to minimize the total time or cost to completion while abiding by the dependences.
3. *Utility graphs.* The power grid, the Internet, and the water network are all examples of graphs where vertices represent connection points, and edges the wires or pipes between them. Analyzing properties of these graphs is very important in understanding the reliability of such utilities under failure or attack, or in minimizing the costs to build infrastructure that matches required demands.
4. *Document link graphs.* The best known example is the link graph of the web, where each web page is a vertex, and each hyperlink a directed edge. Link graphs are used, for example, to analyze relevance of web pages, the best sources of information, and good link sites.
5. *Protein-protein interactions graphs.* Vertices represent proteins and edges represent interactions between them that carry out some biological function in the cell. These graphs can be used, for example, to study molecular pathways—chains of molecular interactions in a cellular process. Humans have over 120K proteins with millions of interactions among them.
6. *Neural networks.* Vertices represent neurons and edges the synapses between them. Neural networks are used to understand how our brain works and how connections change when we learn. The human brain has about 10^{11} neurons and close to 10^{15} synapses.
7. *Network traffic graphs.* Vertices are IP (Internet protocol) addresses and edges are the packets that flow between them. Such graphs are used for analyzing network security, studying the spread of worms, and tracking criminal or non-criminal activity.
8. *Social network graphs.* Graphs that represent who knows whom, who communicates with whom, who influences whom or other relationships in social structures. An example is the twitter graph of who tweeted whom. These can be used to determine how information flows, how topics become hot, how communities develop, or even who might be a good match for who, or is that whom.
9. *Graphs in quantum field theory.* Vertices represent states of a quantum system and the edges the transitions between them. The graphs can be used to analyze path integrals and summing these up generates a quantum amplitude (yes, I have no idea what that means).
10. *Semantic networks.* Vertices represent words or concepts and edges represent the relationships among the words or concepts. These have been used in various models of how humans organize their knowledge, and how machines might simulate such an organization.
11. *Scene graphs.* In graphics and computer games scene graphs represent the logical or spacial relationships between objects in a scene. Such graphs are very important in the computer games industry.

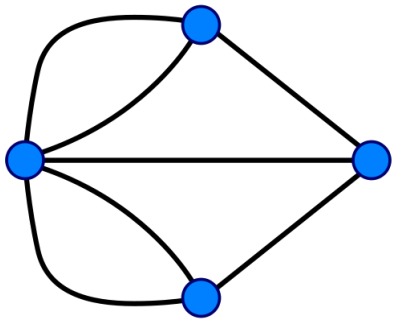
12. *Finite element meshes.* In engineering many simulations of physical systems, such as the flow of air over a car or airplane wing, the spread of earthquakes through the ground, or the structural vibrations of a building, involve partitioning space into discrete elements. The elements along with the connections between adjacent elements forms a graph that is called a finite element mesh.
13. *Robot planning.* Vertices represent states the robot can be in and the edges the possible transitions between the states. This requires approximating continuous motion as a sequence of discrete steps. Such graph plans are used, for example, in planning paths for autonomous vehicles.
14. *Graphs in epidemiology.* Vertices represent individuals and directed edges the transfer of an infectious disease from one individual to another. Analyzing such graphs has become an important component in understanding and controlling the spread of diseases.
15. *Graphs in compilers.* Graphs are used extensively in compilers. They can be used for type inference, for so called data flow analysis, register allocation and many other purposes. They are also used in specialized compilers, such as query optimization in database languages.
16. *Constraint graphs.* Graphs are often used to represent constraints among items. For example the GSM network for cell phones consists of a collection of overlapping cells. Any pair of cells that overlap must operate at different frequencies. These constraints can be modeled as a graph where the cells are vertices and edges are placed between cells that overlap.

There are many other applications of graphs.

1.1 Formalities



An example of a directed graph on 4 vertices.



An undirected graph on 4 vertices[1]

Formally, a *directed graph* or (*digraph*) is a pair $G = (V, A)$ where

- V is a set of *vertices* (or nodes), and
- $A \subseteq V \times V$ is a set of *directed edges* (or arcs).

Each arc is an ordered pair $e = (u, v)$ and a graph can have *self loops* (u, u). Directed graphs represent asymmetric relationships. An *undirected graph* is a pair $G = (V, E)$ where E is a set of unordered pairs over V (i.e., $E \subseteq \binom{V}{2}$). Each edge can be written as $e = \{u, v\}$, which is the same as $\{v, u\}$, and self loops are generally *not* allowed. Undirected graphs represent symmetric relationships.

Note that directed graphs are in some sense more general than undirected graphs since we can easily represent an undirected graph by a directed graph by placing an arc in each direction. Indeed, this is often the way we represent directed graphs in data structures.

Unfortunately, graphs come with a lot of terminology. Though, fortunately, most of it is intuitive once we understand the concept. At this point, we will just talk about graphs that do not have any data associated with edges, such as weights. We will later talk about weighted graphs.

- A vertex u is a *neighbor* of or equivalently *adjacent* to a vertex v if there is an edge between them. For a directed graph we use the terms *in-neighbor* (if the arc points to u) and *out-neighbor* (if the arc points from u).

[1] representing the famous Königsberg problem (figure credit, Wikipedia)

- The *degree* of a vertex is its number of neighbors and will be denoted as $d_G(v)$. For directed graphs we use *in-degree* ($d_G^-(v)$) and *out-degree* ($d_G^+(v)$) with the presumed meanings. We will drop the subscript when it is clear from the context which graph we're talking about.
- For an undirected graph $G = (V, E)$, the *neighborhood* $N_G(v)$ of a vertex $v \in V$ is its set of neighbors, i.e. $N(v) = \{u \mid \{u, v\} \in E\}$. For a directed graph we use $N_G^+(v)$ to indicate the set of out-neighbors and $N_G^-(v)$ to indicate the set of in-neighbors of v. If we use $N_G(v)$ for a directed graph, we mean the out neighbors. The neighborhood of a set of vertices $U \subseteq V$ is the union of their neighborhoods, e.g. $N_G(V) = \cup_{v \in V} N_G(v)$, or $N_G^+(V) = \cup_{v \in V} N_G^+(v)$.
- A *path* in a graph is a sequence of adjacent vertices. More formally for a graph $G = (V, E)$, a path p is a sequence of vertices such that p_i, p_{i+1} is an edge in G for all $i = 1, \ldots, |p| - 1$. The length of a path is one less than the number of vertices in the path—i.e., it is the number of edges in the path.
- A vertex v is *reachable* from a vertex u in G if there is a path starting at v and ending at u in G. An undirected graph is *connected* if all vertices are reachable from all other vertices.
- A *simple path* is a path with no repeated vertices. Often, however, the term simple is dropped, making it sometimes unclear whether path means simple or not (sorry). In this course we will almost exclusively be talking about simple paths and so unless stated otherwise our use of path means simple path.
- A *cycle* is a path that starts and ends at the same vertex. In a directed graph a cycle can have length 1 (i.e. a *self loop*). In an undirected graph we require that a cycle must have length at least three. In particular going from v to u and back to v does not count. A *simple cycle* is a cycle that has no repeated vertices other than the start vertex being the same as the end. In this course we will exclusively talk about simple cycles and hence, as with paths, we will often drop simple.
- An undirected graph with no cycles is a *forest* and if it is connected it is called a *tree*. A directed graph is a forest (or tree) if when all edges are converted to undirected edges it is undirected forest (or tree). A *rooted tree* is a tree with one vertex designated as the root. For a directed graph the edges are typically all directed toward the root or away from the root.

By convention we will use the following definitions:

$$\begin{aligned} n &= |V| \\ m &= |E| \end{aligned}$$

Note that a directed graph can have at most n^2 edges (including self loops) and an undirected graph at most $n(n-1)/2$. We informally say that a graph is *sparse* if $m \ll n^2$ (formally, $m = o(n^2)$; yup, that's a little-oh) and *dense* otherwise. In most applications graphs are very sparse, often with only a handful of neighbors per vertex when averaged across vertices, although some vertices could have high degree. Therefore, the emphasis in the design of graph algorithms, at least for this class, is typically on algorithms that work well for sparse graphs.

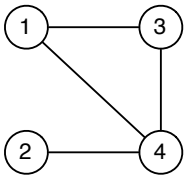
2 How should we represent a graph?

How we want to represent a graph largely depends on the operations we intend to support. To answer this overarching question, let's look at a few things we want to be able to answer efficiently about a graph. As a running example, suppose you just founded a new social networking site. Naturally, the network of friends is represented as a graph, and you're interested in supporting the following operations:

(1) Is x a friend of y?
(2) For each user x, how many friends does x have?
(3) Find friends of x who are not in group Y.
(4) Find friends of friends of x.
(5) Find the diameter of this graph.

Traditionally, the representations assume that vertices are numbered from $1, 2, \ldots, n$ (or $0, 1, \ldots, n-1$):

- **Adjacency matrix.** An $n \times n$ matrix of binary values in which location (i, j) is 1 if $(i, j) \in E$ and 0 otherwise. Note that for an undirected graph the matrix is symmetric and 0 along the diagonal.



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

- **Adjacency array.** This is where we have n arrays, each keeping the neighbors of a particular vertex. That is, if we keep an array A of length n, each $A[i]$ is itself an array of the out-neighbors of vertex i. In an undirected graph with edge $\{u, v\}$ the edge will appear in the adjacency list for both u and v. Using the example (above), we have the following arrays (abusing Java notation):

```
A = [[3, 4],    // node 1
     [2],       // node 2
     [1, 4],    // node 3
     [1, 2, 3] // node 4
    ]
```

- **Edge list.** A list of pairs $(i, j) \in E$. For example, $[(1,3),(1,4),(2,4),(3,1),(3,4),(4,1),(4,2),(4,3)]$.

But nothing stops us from using a more advanced data type to represent a graph. Importantly, when we make decisions about how to represent graphs, we should consider take into account the costs of basic operations on graphs that we intend to support. The following operations are common among standard graph algorithms:

1. $\deg(v)$ — finding the degree of a vertex;
2. *Is* $\{u, v\} \in E(G)$*?* — finding if an edge is in the graph;
3. *Iterate over edges* — iterating over all edges in the graph; and
4. *For each* $v \in N(v)$ — mapping or iterating over all neighbors of a given vertex.

In a dynamically changing graph, we will also want to insert and delete edges.

Idea 1: Suppose we want a representation that directly follows the mathematical definition of a graph and simply uses a set of edges $E \subseteq \binom{V}{2}$. A moment's thought shows that we can actually support these basic operations rather efficiently using sequences. Sets are such an important and useful concept; we should create an ADT for it and see how to best support set operations.

Idea 2: More efficient neighbor access. In our second representation, we aim to get more efficiency in accessing to the neighbors of a vertex. This representation, which we refer to as an *adjacency table*, is a table that maps every vertex to the set of its neighbors. Simply put, it is an edge-set table.

Therefore, in this representation, accessing the neighbors of a vertex v is cheap: it just requires a lookup in the table. Once the corresponding table/set has been fetched, various operations can be supported cheaply.

2.1 A Graph ADT

To support commonly used operations on graphs, we'll define an abstract data type (ADT) for undirected graphs and show how to implement it using the adjacency table idea just discussed. Notice that a similar ADT can be defined for directed graphs.

Exercise: How would you represent a directed graph?

2.1.1 Undirected Graph Interface:

```
public interface UndirectedGraph {

  public int numEdges();
  public int numVertices();

  public void addEdge(int u, int v);
  public void removeEdge(int u, int v);
```

```
  public Iterable<Integer> adj(int v);

  public int deg(int v);
  public boolean isEdge(int u, int v);
}
```

2.1.2 An Implementation Using HashMap and HashSet:

```
import java.util.*;

public class UndirectedAdjTable implements UndirectedGraph {

    Map<Integer, Set<Integer>> adjTable;
    int numEdges;

    public UndirectedAdjTable() {
        numEdges = 0;
        adjTable = new HashMap<>();
    }

    public int numEdges() {
        return numEdges;
    }

    public int numVertices() {
        return adjTable.size();
    }

    public void addVertex(int v) {
        if (!adjTable.containsKey(v))
            adjTable.put(v, new HashSet<>());
    }

    public void addEdge(int u, int v) {
        addVertex(u);
        addVertex(v);
        adjTable.get(u).add(v);
        adjTable.get(v).add(u);
    }

    public void removeEdge(int u, int v) {
        adjTable.get(u).remove(v);
        adjTable.get(v).remove(u);
    }

    public Iterable<Integer> adj(int v) {
        return adjTable.get(v);
    }

    public int deg(int v) {
        return adjTable.containsKey(v) ? adjTable.get(v).size() : 0;
    }

    public boolean isEdge(int u, int v) {
        return adjTable.containsKey(u) && adjTable.get(u).contains(v);
    }

}
```