Lecture 10: Graph Data Structures

Steven Skiena

Department of Computer Science
State University of New York
Stony Brook, NY 11794–4400

http://www.cs.sunysb.edu/∼skiena

Sort Yourselves

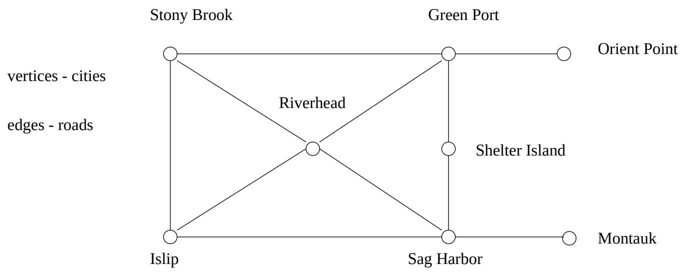
Sort yourselves in alphabetical order so I can return the midterms efficiently!

Graphs

Graphs are one of the unifying themes of computer science.
A graph $G = (V, E)$ is defined by a set of *vertices* V, and a set of *edges* E consisting of ordered or unordered pairs of vertices from V.

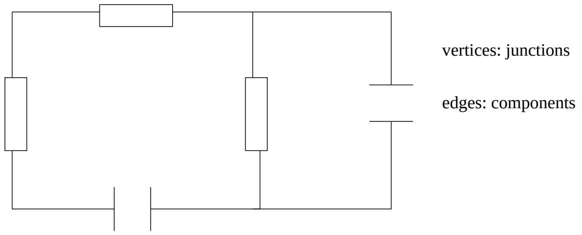
Road Networks

In modeling a road network, the vertices may represent the cities or junctions, certain pairs of which are connected by roads/edges.



Electronic Circuits

In an electronic circuit, with junctions as vertices as components as edges.



Flavors of Graphs

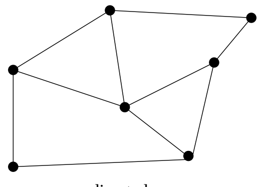
The first step in any graph problem is determining which flavor of graph you are dealing with.

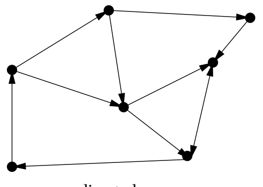
Learning to talk the talk is an important part of walking the walk.

The flavor of graph has a big impact on which algorithms are appropriate and efficient.

Directed vs. Undirected Graphs

A graph $G = (V, E)$ is *undirected* if edge $(x, y) \in E$ implies that (y, x) is also in E.

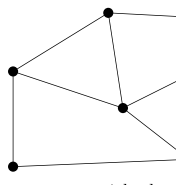

undirected

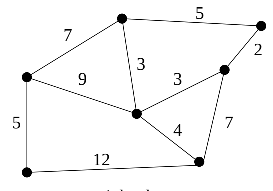

directed

Road networks *between* cities are typically undirected.
Street networks *within* cities are almost always directed because of one-way streets.
Most graphs of graph-theoretic interest are undirected.

Weighted vs. Unweighted Graphs

In *weighted* graphs, each edge (or vertex) of G is assigned a numerical value, or weight.





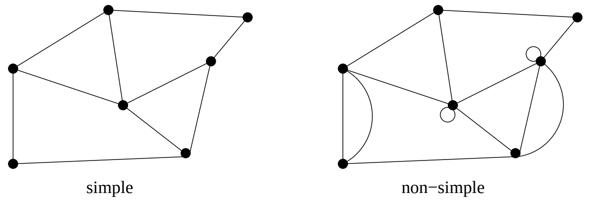
The edges of a road network graph might be weighted with their length, drive-time or speed limit.

In *unweighted* graphs, there is no cost distinction between various edges and vertices.

Simple vs. Non-simple Graphs

Certain types of edges complicate the task of working with graphs. A *self-loop* is an edge (x, x) involving only one vertex.

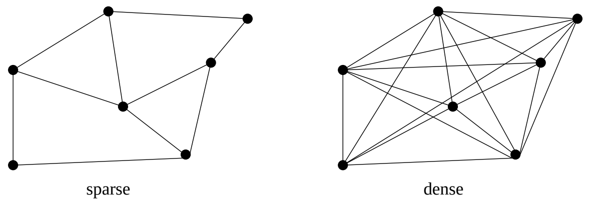
An edge (x, y) is a *multi-edge* if it occurs more than once in the graph.



Any graph which avoids these structures is called *simple*.

Sparse vs. Dense Graphs

Graphs are *sparse* when only a small fraction of the possible number of vertex pairs actually have edges defined between them.

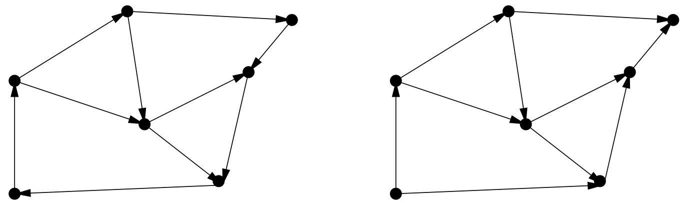


Graphs are usually sparse due to application-specific constraints. Road networks must be sparse because of road junctions.

Typically dense graphs have a quadratic number of edges while sparse graphs are linear in size.

Cyclic vs. Acyclic Graphs

An *acyclic* graph does not contain any cycles. *Trees* are connected acyclic *undirected* graphs.

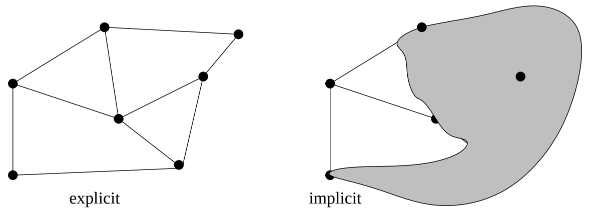


cyclic acyclic

Directed acyclic graphs are called *DAGs*. They arise naturally in scheduling problems, where a directed edge (x, y) indicates that x must occur before y.

Implicit vs. Explicit Graphs

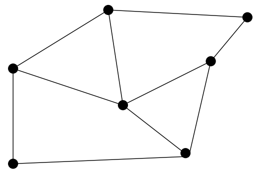
Many graphs are not explicitly constructed and then traversed, but built as we use them.

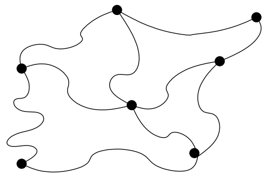


A good example arises in backtrack search.

Embedded vs. Topological Graphs

A graph is *embedded* if the vertices and edges have been assigned geometric positions.

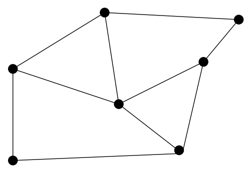

embedded

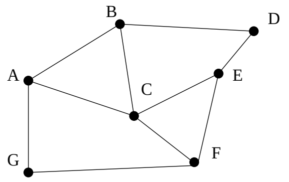

topological

Example: TSP or Shortest path on points in the plane.
Example: Grid graphs.
Example: Planar graphs.

Labeled vs. Unlabeled Graphs

In *labeled* graphs, each vertex is assigned a unique name or identifier to distinguish it from all other vertices.


unlabeled

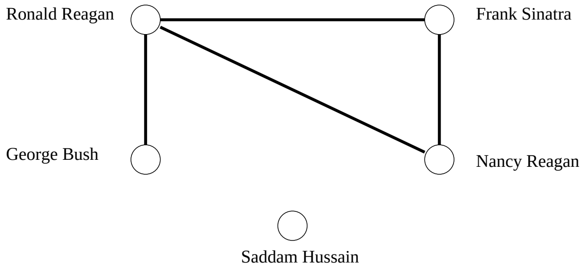


labeled

An important graph problem is *isomorphism testing*, determining whether the topological structure of two graphs are in fact identical if we ignore any labels.

The Friendship Graph

Consider a graph where the vertices are people, and there is an edge between two people if and only if they are friends.



This graph is well-defined on any set of people: SUNY SB, New York, or the world.

What questions might we ask about the friendship graph?

If I am your friend, does that mean you are my friend?

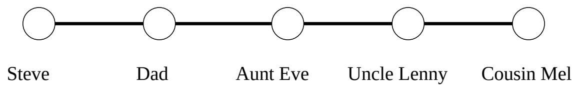
A graph is *undirected* if (x, y) implies (y, x). Otherwise the graph is directed.
The “heard-of” graph is directed since countless famous people have never heard of me!
The “had-sex-with” graph is presumably undirected, since it requires a partner.

Am I my own friend?

An edge of the form (x, x) is said to be a *loop*.
If x is y's friend several times over, that could be modeled using *multiedges*, multiple edges between the same pair of vertices.
A graph is said to be *simple* if it contains no loops and multiple edges.

Am I linked by some chain of friends to the President?

A *path* is a sequence of edges connecting two vertices. Since *Mel Brooks* is my father's-sister's-husband's cousin, there is a path between me and him!



How close is my link to the President?

If I were trying to impress you with how tight I am with Mel Brooks, I would be much better off saying that Uncle Lenny knows him than to go into the details of how connected I am to Uncle Lenny.

Thus we are often interested in the *shortest path* between two nodes.

Is there a path of friends between any two people?

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

A directed graph is *strongly connected* if there is a directed path between any two vertices.

Who has the most friends?

The *degree* of a vertex is the number of edges adjacent to it.

Data Structures for Graphs: Adjacency Matrix

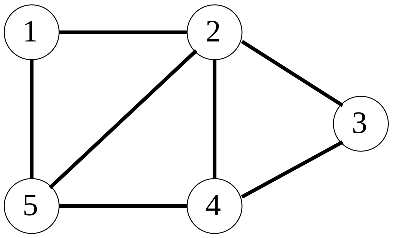
There are two main data structures used to represent graphs. We assume the graph $G = (V, E)$ contains n vertices and m edges.

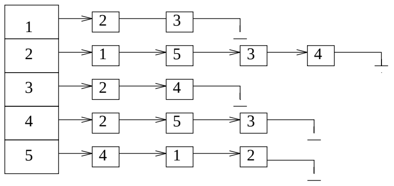
We can represent G using an $n \times n$ matrix M, where element $M[i, j]$ is, say, 1, if (i, j) is an edge of G, and 0 if it isn't. It may use excessive space for graphs with many vertices and relatively few edges, however.

Can we save space if (1) the graph is undirected? (2) if the graph is sparse?

Adjacency Lists

An *adjacency list* consists of a $N \times 1$ array of pointers, where the ith element points to a linked list of the edges incident on vertex *i*.





To test if edge (i, j) is in the graph, we search the ith list for j, which takes $O(d_i)$, where d_i is the degree of the ith vertex. Note that d_i can be much less than n when the graph is sparse. If necessary, the two *copies* of each edge can be linked by a pointer to facilitate deletions.

Tradeoffs Between Adjacency Lists and Adjacency Matrices

Comparison	Winner
Faster to test if (x, y) exists?	matrices
Faster to find vertex degree?	lists
Less memory on small graphs?	lists $(m + n)$ vs. (n^2)
Less memory on big graphs?	matrices (small win)
Edge insertion or deletion?	matrices $O(1)$
Faster to traverse the graph?	lists $m + n$ vs. n^2
Better for most problems?	lists

Both representations are very useful and have different properties, although adjacency lists are probably better for most problems.

Adjancency List Representation

```
#define MAXV 100

typedef struct {
        int y;
        int weight;
        struct edgenode *next;
} edgenode;
```

Edge Representation

```
typedef struct {
        edgenode *edges[MAXV+1];
        int degree[MAXV+1];
        int nvertices;
        int nedges;
        bool directed;
} graph;
```

The degree field counts the number of meaningful entries for the given vertex. An undirected edge (x, y) appears twice in any adjacency-based graph structure, once as y in x's list, and once as x in y's list.

Initializing a Graph

```
initialize_graph(graph *g, bool directed)
{
        int i;

        g - > nvertices = 0;
        g - > nedges = 0;
        g - > directed = directed;

        for (i=1; i<=MAXV; i++) g- >degree[i] = 0;
        for (i=1; i<=MAXV; i++) g- >edges[i] = NULL;
}
```

Reading a Graph

A typical graph format consists of an initial line featuring the number of vertices and edges in the graph, followed by a listing of the edges at one vertex pair per line.

```
read_graph(graph *g, bool directed)
{       int i;
        int m;
        int x, y;

        initialize_graph(g, directed);

        scanf("%d %d",&(g->nvertices),&m);

        for (i=1; i<=m; i++) {
                scanf("%d %d",&x,&y);
                insert_edge(g,x,y,directed);
        }
}
```

Inserting an Edge

```
insert_edge(graph *g, int x, int y, bool directed)
{
        edgenode *p;

        p = malloc(sizeof(edgenode));

        p− >weight = NULL;
        p− >y = y;
        p− >next = g− >edges[x];

        g− >edges[x] = p;

        g− >degree[x] ++;

        if (directed == FALSE)
                insert_edge(g,y,x,TRUE);
        else
                g− >nedges ++;
}
```