10 Basic Graph Properties (February 25)

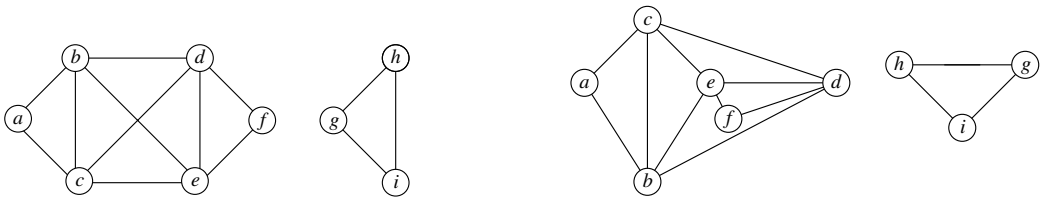
10.1 Definitions

A *graph* G is a pair of sets (V, E). V is a set of arbitrary objects which we call *vertices*[1] or *nodes*. E is a set of vertex pairs, which we call *edges* or occasionally *arcs*. In an *undirected* graph, the edges are unordered pairs, or just sets containing two vertices. In a *directed* graph, the edges are ordered pairs of vertices. We will only be concerned with *simple* graphs, where there is no edge from a vertex to itself and there is at most one edge from any vertex to any other.

Following standard (but admittedly confusing) practice, I'll also use V to denote the *number* of vertices in a graph, and E to denote the *number* of edges. Thus, in an undirected graph, we have $0 \leq E \leq \binom{V}{2}$, and in a directed graph, $0 \leq E \leq V(V-1)$.

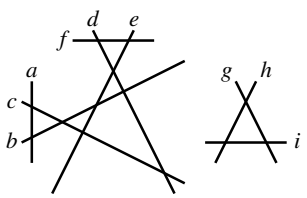
If (u, v) is an edge in an undirected graph, then u is a *neighbor* or v and vice versa. The *degree* of a node is the number of neighbors. In directed graphs, we have two kinds of neighbors. If $u \to v$ is a directed edge, then u is a *predecessor* of v and v is a *successor* of u. The *in-degree* of a node is the number of predecessors, which is the same as the number of edges going into the node. The *out-degree* is the number of successors, or the number of edges going out of the node.

We usually visualize graphs by looking at an *embedding*. An embedding of a graph maps each vertex to a point in the plane and each edge to a curve or straight line segment in the plane. A graph is *planar* if it has an embedding where no two edges cross. The same graph can have many different embeddings, so it is important to remember not to confuse a particular embedding with the graph itself. In particular, planar graphs can have non-planar embeddings!



A non-planar embedding of a planar graph with 9 vertices, 12 edges, and two connected components, and a planar embedding of the same graph.

However, there are other ways of visualizing graphs that are occasionally also useful. For example, the *intersection graph* of a collection of objects has a node for every object and an edge for every intersecting pair. (Whether a particular graph can be represented as an intersection graph depends on what kind of object you want to use for the vertices. Different types of objects — line segments, rectangles, circles, convex polygons, etc. — define different classes of graphs.)



The example graph is also the intersection graph of a set of line segments.

[1]The singular of 'vertices' is **vertex**. The singular of 'matrices' is **matrix**. There is no such thing as a vertice, a matrice, an indice, an appendice, a helice, an apice, a vortice, a radice, a simplice, a directrice, a dominatrice, a Unice, or Jimi Hendrice!

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

A *path* is a sequence of edges, where each successive pair of edges shares a vertex, and all other edges are disjoint. A graph is *connected* if there is a path from any vertex to any other vertex. A disconnected graph consists of several *connected components*, which are maximal connected subgraphs. Two vertices are in the same connected component if and only if there is a path between them.

A graph is *acyclic* if no subgraph is a cycle. *Trees* are special graphs that can be defined in several different ways. You can easily prove by induction (hint, hint, hint) that the following definitions are equivalent.

- A tree is a connected acyclic graph.
- A tree is a connected graph with V vertices and $V - 1$ edges.
- A tree is a minimal connected graph; removing any edge makes the graph disconnected.
- A tree is a maximal acyclic graph; adding an edge between any two vertices creates a cycle.

A *spanning tree* of a graph G is a subgraph that is a tree and contains every vertex of G. Of course, a graph can only have a spanning tree if it's connected. A *spanning forest* of G is a collection of spanning trees, one for each connected component of G.

10.2 Explicit Representations of Graphs

There are two common data structures used to explicitly represent graphs: *adjacency matrices*[2] and *adjacency lists*.

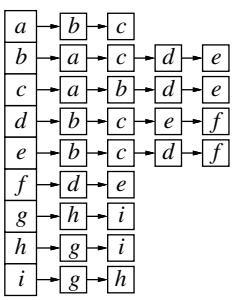
The adjacency matrix of a graph G is a $V \times V$ of indicator variables. Each entry in the matrix encodes whether a particular edge is or is not in the graph:

$$A[i,j] = \big[(i,j) \in E\big].$$

For undirected graphs, the adjacency matrix is always *symmetric*: $A[i,j] = A[j,i]$. Since we don't allow edges from a vertex to itself, the diagonal elements of the adjacency matrix are all zeros.

Given an adjacency matrix, we can decide in $\Theta(1)$ time whether two vertices are connected by an edge just by looking in the appropriate slot in the matrix. We can also list all the neighbors of a vertex in $\Theta(V)$ time by scanning the corresponding row (or column). This is optimal in the worst case, since a vertex can have up to $V-1$ neighbors; however, if a vertex has few neighbors, we may still have to examine every entry in the row to see them all. Similarly, adjacency matrices require $\Theta(V^2)$ space, regardless of how many edges the graph actually has, so it is only space-efficient for very *dense* graphs.

	a	b	c	d	e	f	g	h	i
a	0	1	1	0	0	0	0	0	0
b	1	0	1	1	1	0	0	0	0
c	1	1	0	1	1	0	0	0	0
d	0	1	1	0	1	1	0	0	0
e	0	1	1	1	0	1	0	0	0
f	0	0	0	1	1	0	0	0	0
g	0	0	0	0	0	0	0	1	0
h	0	0	0	0	0	0	1	0	1
i	0	0	0	0	0	0	1	1	1



Adjacency matrix and adjacency list representations for the example graph.

[2] See footnote 1.

For *sparse* graphs—graphs with relatively few edges—we're better off using adjacency lists. An adjacency list is an array of linked lists, one list per vertex. Each linked list stores the neighbors of the corresponding vertex.

For undirected graphs, each edge (u, v) is stored twice, once in u's neighbor list and once in v's neighbor list; for directed graphs, each edge is stores only once. Either way, the overall space required for an adjacency list is $O(V+E)$. Listing the neighbors of a node v takes $O(1+\deg(v))$ time; just scan the neighbor list. Similarly, we can determine whether (u, v) is an edge in $O(1+\deg(u))$ time by scanning the neighbor list of u. For undirected graphs, we can speed up the search by simultaneously scanning the neighbor lists of both u and v, stopping either we locate the edge or when we fall of the end of a list. This takes $O(1 + \min\{\deg(u), \deg(v)\})$ time.

The adjacency list structure should immediately remind you of hash tables with chaining. Just as with hash tables, we can make adjacency list structure more efficient by using something besides a linked list to store the neighbors. For example, if we use a hash table with constant load factor, when we can detect edges in $O(1)$ expected time, just as with an adjacency list. In practice, this will only be useful for vertices with large degree, since the constant overhead in both the space and search time is larger for hash tables than for simple linked lists.

Graphs can also be represented *implicitly*. For example, any data structure build from records with pointers between them can be seen as a directed graph. Algorithms for searching graphs can be applied to these data structures as though the graph were represented explicitly using an adjacency list. On the other hand, intersection graphs can be represented implicitly by simply storing the list of objects. As long as we can test whether two objects overlap in constant time, we can apply any graph algorithm to an intersection graph by pretending that it is stored explicitly as an adjacency matrix.

To keep things simple, we'll consider only undirected graphs for the rest of this lecture, although the algorithms I'll describe also work for directed graphs.

10.3 Traversing connected graphs

Suppose we want to visit every node in a connected graph, which may me represented implicitly. The simplest method to do this is an algorithm called *depth-first search*, which can be written either recursively or non-recursively. It's exactly the same algorithm either way; the only difference is that we can see the "recursion" stack in the non-recursive version. Both versions are initially passed a *source* vertex s.

```
RecursiveDFS(v):
    if v is unmarked
        mark v
        for each edge (v, w)
            RecursiveDFS(w)
```

```
IterativeDFS(s):
    Push(s)
    while stack not empty
        v ← Pop
        if v is unmarked
            mark v
            for each edge (v, w)
                Push(w)
```

Depth-first search is one (perhaps the most common) instance of a general family of graph traversal algorithms. The generic graph traversal algorithm stores a set of candidate edges in some data structure—in the case of depth-first search, a stack. Here's the algorithm:

```
TRAVERSE(s):
  put (∅, s) in bag
  while bag is not empty
    take (p, v) from bag        (⋆)
    if v is unmarked
      mark v
      parent(v) ← p
      for each edge (v, w)      (†)
        put (v, w) in bag       (⋆⋆)
```

Notice that we're keeping *edges* in the bag instead of *vertices*. This is because we want to remember, whenever we visit a vertex v for the first time, which previously-visited vertex p put v into the bag. The vertex p is called the *parent* of v.

Lemma 1. TRAVERSE(s) *marks every vertex in any connected graph exactly once, and the set of edges* $(v, \text{parent}(v))$ *with* $\text{parent}(v) \neq \varnothing$ *form a spanning tree of the graph.*

Proof: First, it should be obvious that no node is marked more than once.

Clearly, the algorithm marks s. Let $v \neq s$ be a vertex, and let $s \to \cdots \to u \to v$ be the path from s to v with the minimum number of edges. Since the graph is connected, such a path always exists. (If s and v are neighbors, then $u = s$, and the path has just one edge.) If the algorithm marks u, then it must put (u, v) into the bag, so it must later take (u, v) out of the bag, at which point v must be marked (if it isn't already). Thus, by induction on the shortest-path distance from s, the algorithm marks every vertex in the graph.

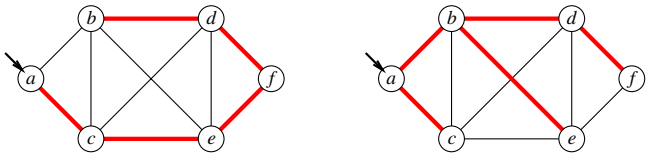
Call an edge $(v, \text{parent}(v))$ with $\text{parent}(v) \neq \varnothing$ a *parent edge*. For any node v, the path of parent edges $v \to \text{parent}(v) \to \text{parent}(\text{parent}(v)) \to \cdots$ eventually leads back to s, so the set of parent edges form a connected graph. Clearly, both endpoints of every parent edge are marked, and the number of parent edges is exactly one less than the number of vertices. Thus, the parent edges form a spanning tree. □

The exact running time of the traversal algorithm depends on how the graph is represented and what data structure is used as the "bag", but we can make a few general observations. Since each vertex is visited at most once, the for loop (†) is executed at most V times. Each edge is put into the bag exactly twice; once as (u, v) and once as (v, u), so line $(\star\star)$ is executed at most $2E$ times. Finally, since we can't take more things out of the bag than we put in, line $(\star)$ is executed at most $2E + 1$ times.

10.4 Examples

Let's first assume that the graph is represented by an adjacency list, so that the overhead of the for loop (†) is only a constant per edge.

- If we implement the "bag" by using a *stack*, we have depth-first search. Each execution of $(\star)$ or $(\star\star)$ takes constant time, so the overall running time is $O(V + E)$. Since the graph is connected, $V \leq E + 1$, so we can simplify the running time to $O(E)$. The spanning tree formed by the parent edges is called a *depth-first spanning tree*. The exact shape of the tree depends on the order in which neighbor edges are pushed onto the stack, but the in general, depth-first spanning trees are long and skinny.



A depth-first spanning tree and a breadth-first spanning tree of one component of the example graph, with start vertex a.

- If we use a *queue* instead of a stack, we have *breadth-first search*. Again, each execution of $(\star)$ or $(\star\star)$ takes constant time, so the overall running time is still $O(E)$. In this case, the *breadth-first spanning tree* formed by the parent edges contains *shortest paths* from the start vertex s to every other vertex in its connected component. The exact shape of the shortest path tree depends on the order in which neighbor edges are pushed onto the queue, but the in general, shortest path trees are short and bushy. We'll see shortest paths again next week.

- Suppose the edges of the graph are weighted. If we implement the "bag" using a *priority queue*, always extracting the minimum-weight edge in line $(\star)$, then we we have what might be called *shortest-first search*. In this case, each execution of $(\star)$ or $(\star\star)$ takes $O(\log E)$ time, so the overall running time is still $O((V + E) \log E)$, or $O(E \log E)$ if the graph is connected. In this case, the set of parent edges form the *minimum spanning tree* of the connected component of s. We'll talk more about minimum spanning trees next time.

If the graph is represented using an adjacency matrix, the finding all the neighbors of each vertex in line $(\dagger)$ takes $O(V)$ time. Thus, depth- and breadth-first search take $O(V^2)$ time overall, and "shortest-first search" takes $O(V^2 + E \log E) = O(V^2 \log V)$ time overall.

10.5 Searching disconnected graphs

If the graph is disconnected, then TRAVERSE(s) only visits the nodes in the connected component of the start vertex s. If we want to visit all the nodes in every component, we can use the following "wrapper" around our generic traversal algorithm. Since TRAVERSE computes a spanning tree of one component, TRAVERSEALL computes a spanning *forest* of the entire graph.

```
TRAVERSEALL(s):
  for all vertices v
    if v is unmarked
      TRAVERSE(v)
```

There is a rather unfortunate mistake on page 477 of [CLR]:

> Unlike breadth-first search, whose predecessor subgraph forms a tree, the predecessor subgraph produced by depth-first search may be composed of several trees, because the search may be repeated from multiple sources.

The implication is that depth-first search is always called with the TRAVERSEALL, and breadth-first search never is. **This is absolute nonsense!** The choice of whether to use a stack or a queue is completely independent of the choice of whether to use TRAVERSEALL or not.