Chapter 9

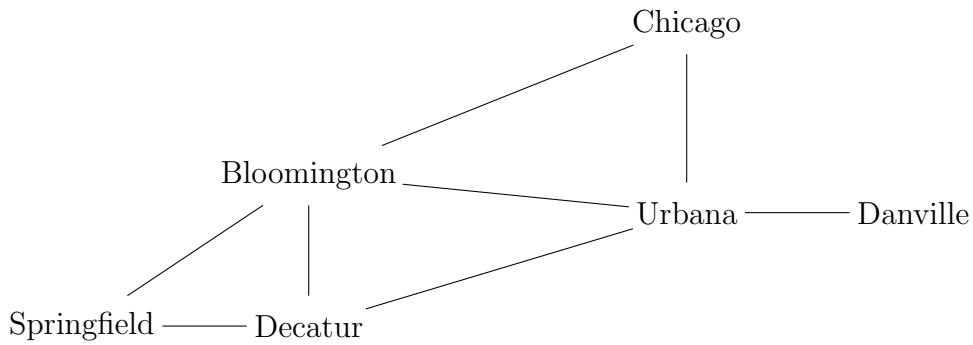
Graphs

Graphs are a very general class of object, used to formalize a wide variety of practical problems in computer science. In this chapter, we'll see the basics of (finite) undirected graphs, including graph isomorphism, connectivity, and graph coloring.

9.1 Graphs

A graph consists of a set of nodes V and a set of edges E. We'll sometimes refer to the graph as a pair of sets (V, E). Each edge in E joins two nodes in V. Two nodes connected by an edge are called **neighbors** or **adjacent**.

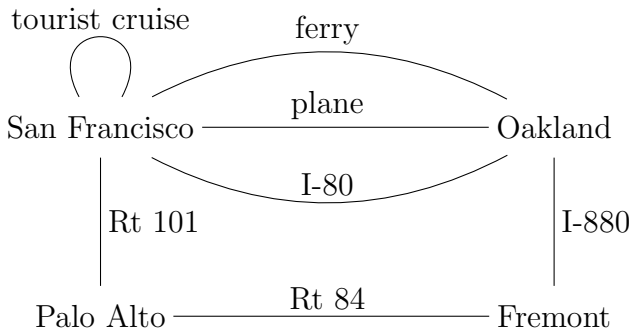
For example, here is a graph in which the nodes are Illinois cities and the edges are roads joining them:



A graph edge can be traversed in both directions, as in this street example, i.e. the edges are **undirected**. When discussing relations earlier, we used **directed graphs**, in which each edge had a specific direction. Unless we explicitly state otherwise, a "graph" will always be undirected. Concepts for undirected graphs extend in straightforward ways to directed graphs.

When there is only one edge connecting two nodes x and y, we can name the edge using the pair of nodes. We could call the edge xy or (since order doesn't matter) yx or $\{x, y\}$. So, in the graph above, the Urbana-Danville edge connects the node Urbana and the node Danville.

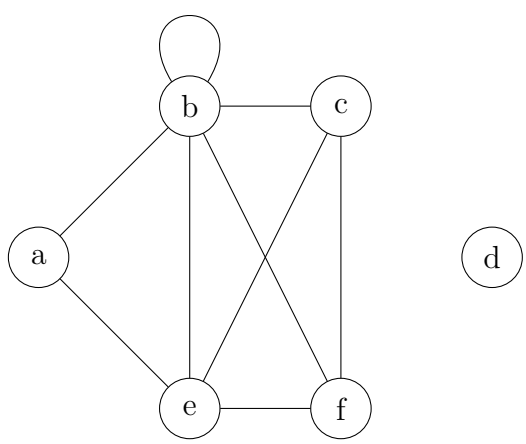
In some applications, we need graphs in which two nodes are connected by **multiple edges**, i.e. parallel edges with the same endpoints. For example, the following graph shows ways to travel among four cities in the San Francisco Bay Area. It has three edges from San Francisco to Oakland, representing different modes of transportation. When multiple edges are present, we typically label the edges rather than trying to name edges by their endpoints. This diagram also illustrates a **loop** edge which connects a node to itself.



A graph is called a **simple graph** if it has neither multiple edges nor loop edges. Unless we explicitly state otherwise, a "graph" will always be a simple graph. Also, we'll assume that it has at least one node and that it has only a finite number of edges and nodes. Again, most concepts extend in a reasonable way to infinite and non-simple graphs.

9.2 Degrees

The degree of a node v, written $deg(v)$ is the number of edges which have v as an endpoint. Self-loops, if you are allowing them, count twice. For example, in the following graph, a has degree 2, b has degree 6, d has degree 0, and so forth.



Each edge contributes to two node degrees. So the sum of the degrees of all the nodes is twice the number of edges. This is called the Handshaking Theorem and can be written as

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

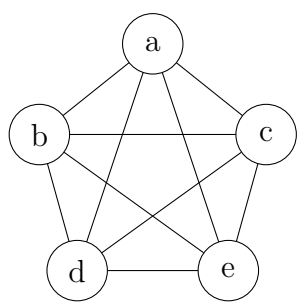
This is a slightly different version of summation notation. We pick each node v in the set V, get its degree, and add its value into the sum. Since V is finite, we could also have given names to the nodes $v_1, \ldots, v_n$ and then written

$$\sum_{k=1}^{n} v_k \in V deg(v) = 2|E|$$

The advantage to the first, set-based, style is that it generalizes well to situations involving infinite sets.

9.3 Complete graphs

Several special types of graphs are useful as examples. First, the complete graph on n nodes (shorthand name K_n), is a graph with n nodes in which every node is connected to every other node. K_5 is shown below.



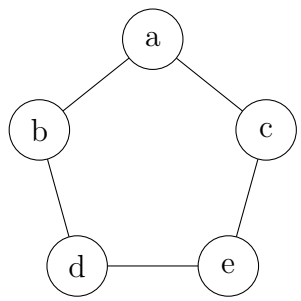
To calculate the number of edges in K_n, think about the situation from the perspective of the first node. It is connected to $n-1$ other nodes. If we look at the second node, it adds $n-2$ more connections. And so forth. So we have $\sum_{k=1}^{n}(n-k) = \sum_{k=0}^{n-1} k = \frac{n(n-1)}{2}$ edges.

9.4 Cycle graphs and wheels

Suppose that we have n nodes named $v_1, \ldots, v_n$, where $n \geq 3$. Then the cycle graph C_n is the graph with these nodes and edges connecting v_i to v_{i+1}, plus an additional edge from v_n to v_1. That is, the set of edges is:

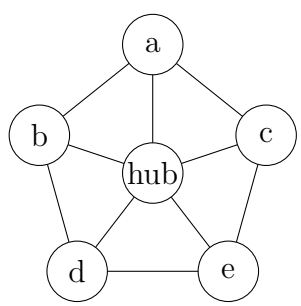
$$E = \{v_1v_2,\ v_2v_3,\ \ldots,\ v_{n-1}v_n,\ v_nv_1\}$$

So C_5 looks like



C_n has n nodes and also n edges. Cycle graphs often occur in networking applications. They could also be used to model games like "telephone" where people sit in a circle and communicate only with their neighbors.

The wheel W_n is just like the cycle graph C_n except that it has an additional central "hub" node which is connected to all the others. Notice that W_n has $n+1$ nodes (not n nodes). It has $2n$ edges. For example, W_5 looks like

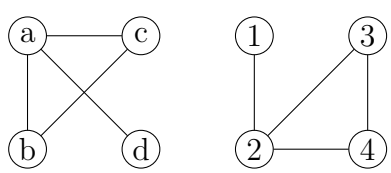


9.5 Isomorphism

In graph theory, we only care about how nodes and edges are connected together. We don't care about how they are arranged on the page or in space, how the nodes and edges are named, and whether the edges are drawn as straight or curvy. We would like to treat graphs as interchangeable if they have the same abstract connectivity structure.

Specifically, suppose that $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are graphs. An **isomorphism** from G_1 to G_2 is a bijection $f : V_1 \to V_2$ such that nodes a and b are joined by an edge if and only if $f(a)$ and $f(b)$ are joined by an edge. The graphs G_1 and G_2 are **isomorphic** if there is an isomorphism from G_1 to G_2.

For example, the following two graphs are isomorphic. We can prove this by defining the function f so that it maps 1 to d, 2 to a, 3 to c, and 4 to b. The reader can then verify that edges exist in the left graph if and only if the corresponding edges exist in the right graph.



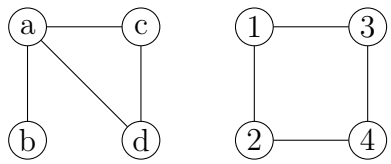
Graph isomorphism is another example of an equivalence relation. Each equivalence class contains a group of graphs which are superficially different (e.g. different names for the nodes, drawn differently on the page) but all represent the same underlying abstract graph.

To prove that two graphs are not isomorphic, we could walk through all possible functions mapping the nodes of one to the nodes of the other. However, that's a huge number of functions for graphs of any interesting size. An exponential number, in fact. Instead, a better technique for many examples is to notice that a number of graph properties are "invariant," i.e. preserved by isomorphism.

- The two graphs must have the same number of nodes and the same number of edges.

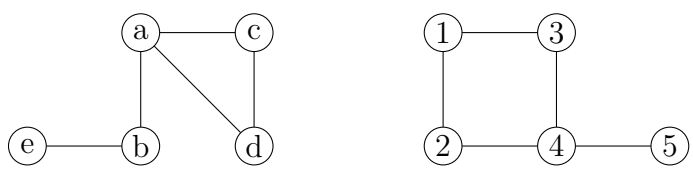
- For any node degree k, the two graphs must have the same number of nodes of degree k. For example, they must have the same number of nodes with degree 3.

We can prove that two graphs are not isomorphic by giving one example of a property that is supposed to be invariant but, in fact, differs between the two graphs. For example, in the following picture, the lefthand graph has a node of degree 3, but the righthand graph has no nodes of degree 3, so they can't be isomorphic.



9.6 Subgraphs

It's not hard to find a pair of graphs that aren't isomorphic but where the most obvious properties (e.g. node degrees) match. To prove that such a pair isn't isomorphic, it's often helpful to focus on certain specific local features of one graph that aren't present in the other graph. For example, the following two graphs have the same node degrees: one node of degree 1, three of degree 2, one of degree 3. However, a little experimentation suggests they aren't isomorphic.



To make a convincing argument that these graphs aren't isomorphic, we need to define the notion of a **subgraph**. If G and G' are graphs, then G' is a subgraph of G if and only if the nodes of G' are a subset of the nodes of

G and the edges of G' are a subset of the edges of G. If two graphs G and F are isomorphic, then any subgraph of G must have a matching subgraph somewhere in F.

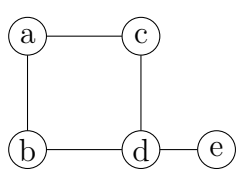
A graph has a huge number of subgraphs. However, we can usually find evidence of non-isomorphism by looking at small subgraphs. For example, in the graphs above, the lefthand graph has C_3 as a subgraph, but the righthand graph does not. So they can't be isomorphic.

9.7 Walks, paths, and cycles

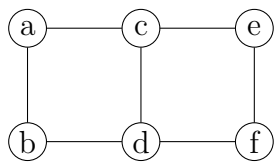
In a graph G, a walk of length k from node a to node b is a finite sequence of nodes $a = v_1, v_2, \ldots, v_n = b$ and a finite sequence of edges $e_1, e_2, \ldots, e_{n-1}$ in which e_i connects v_i and v_{i+1}, for all i. Under most circumstances, it isn't necessary to give both the sequence of nodes and the sequence of edges: one of the two is usually sufficient. The length of a walk is the number of edges in it. The shortest walks consist of just a single node and have length zero.

A walk is **closed** if its starting and ending nodes are the same. Otherwise it is **open**. A **path** is a walk in which no node is used more than once. A **cycle** is a closed walk with at least three nodes in which no node is used more than once except that the starting and ending nodes are the same.

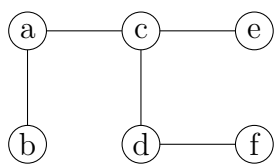
For example, in the following graph, there is a length-3 walk from a to e: ac, cd, de. Another walk of length 3 would have edges: ab, bd, de. These two walks are also paths. There are also longer walks from a to e, which aren't paths because they re-use nodes, e.g. the walk with a node sequence a, c, d, b, d, e.



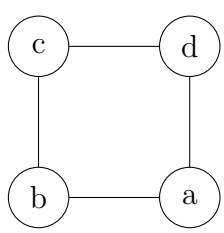
In the following graph, one cycle of length 4 has edges: ab, bc, ce, da. Other closely-related cycles go through the same nodes but with a different starting point or in the opposite direction, e.g. da, ab, bc, ce. Unlike cycles, closed walks can re-use nodes, e.g. ab, ba, ac, ce, ec, ca is a closed walk but not a cycle.



The following graph is **acyclic**, i.e. it doesn't contain any cycles.

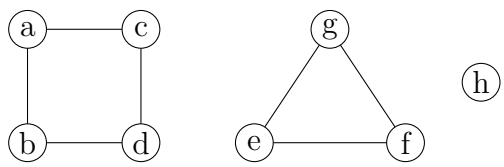


Notice that the cycle graph C_n contains $2n$ different cycles. For example, if the vertices of C_4 are labelled as shown below, then one cycle is ab, bc, cd, da, another is cd, bc, ab, da, and so forth.



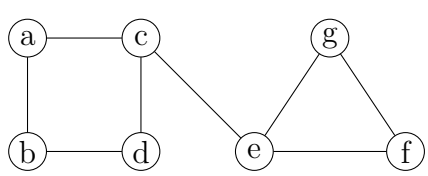
9.8 Connectivity

A graph G is **connected** if there is a walk between every pair of nodes in G. Our previous examples of graphs were connected. The following graph is not connected, because there is no walk from (for example), a to g.



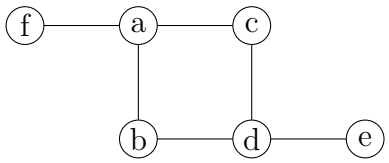
If we have a graph G that might or might not be connected, we can divide G into **connected components.** Each connected component contains a maximal (i.e. biggest possible) set of nodes that are all connected to one another, plus all their edges. So, the above graph has three connected components: one containing nodes a,b,c, and d, a second containing nodes e, f, and g, and a third that contains only the node h.

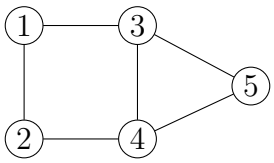
Sometimes two parts of a graph are connected by only a single edge, so that the graph would become disconnected if that edge were removed. This is called a **cut edge.** For example, in the following graph, the edge ce is a cut edge. In some applications, cut edges are a problem. E.g. in networking, they are places where the network is vulnerable. In other applications (e.g. compilers), they represent opportunities to divide a larger problem into several simpler ones.



9.9 Distances

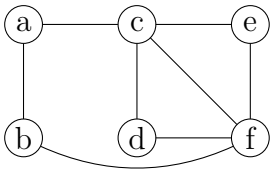
In graphs, distances are based on the lengths of paths connecting pairs of nodes. Specifically, the distance $d(a, b)$ between two nodes a and b is the length of the shortest path from a to b. The diameter of a graph is the maximum distance between any pair of nodes in the graph. For example, the lefthand graph below has diameter 4, because $d(f, e) = 4$ and no other pair of nodes is further apart. The righthand graph has diameter 2, because $d(1, 5) = 2$ and no other pair of nodes is further apart.





9.10 Euler circuits

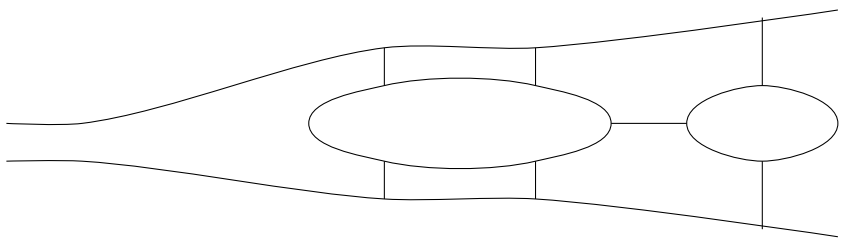
An Euler circuit of a graph G is a closed walk that uses each edge of the graph exactly once. For example, one Euler circuit of the following graph would be ac, cd, df, fe, ec, cf, fb, ba.



An Euler circuit is possible exactly when the graph is connected and each node has even degree. Each node has to have even degree because, in order to complete the circuit, you have to leave each node that you enter. If the

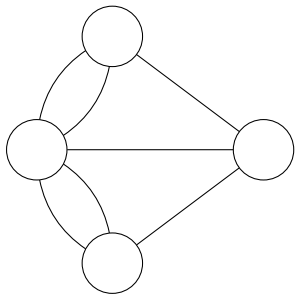
node has odd degree, you will eventually enter a node but have no unused edge to go out on.

Fascination with Euler circuits dates back to the 18th century. At that time, the city of Königberg, in Prussia, had a set of bridges that looked roughly as follows:



Folks in the town wondered whether it was possible to take a walk in which you crossed each bridge exactly once, coming back to the same place you started. This is the kind of thing that starts long debates late at night in pubs, or keeps people amused during boring church services. Leonard Euler was the one who explained clearly why this isn't possible.

For our specific example, the corresponding graph looks as follows. Since all of the nodes have odd degree, there's no possibility of an Euler circuit.

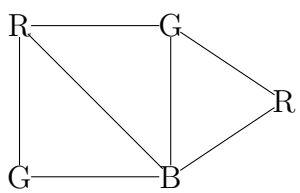


9.11 Graph coloring

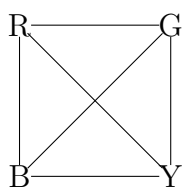
A *coloring* of a graph G assigns a color to each node of G, with the restriction that two adjacent nodes never have the same color. If G can be colored with

k colors, we say that G is k-colorable. The *chromatic number* of G, written $\chi(G)$, is the smallest number of colors needed to color G.

For example, only three colors are required for this graph:



But the complete graph K_n requires n colors, because each node is adjacent to all the other nodes. E.g. K_4 can be colored as follows:



To establish that n is the chromatic number for a graph G, we need to establish two facts:

- $\chi(G) \leq n$: G can be colored with n colors.
- $\chi(G) \geq n$: G cannot be colored with less than n colors

For small finite graphs, the simplest way to show that $\chi(G) \leq n$ is to show a coloring of G that uses n colors.

Showing that $\chi(G) \geq n$ can sometimes be equally straightforward. For example, G may contain a copy of K_n, which can't be colored with less than n colors. However, sometimes it may be necessary to step carefully through all the possible ways to color G with $n-1$ colors and show that none of them works out.

So determining the chromatic number of a graph can be relatively easy for graphs with a helpful structure, but quite difficult for other sorts of graphs. A fully general algorithm to find chromatic number will take time roughly exponential in the size of the graph, for the most difficult input graphs.[1] Such a slow running time isn't practical except for small examples, so applications tend to focus on approximate solutions.

9.12 Why should I care?

Graph coloring is required for solving a wide range of practical problems. For example, there is a coloring algorithm embedded in most compilers. Because the general problem can't be solved efficiently, the implemented algorithms use limitations or approximations of various sorts so that they can run in a reasonable amount of time.

For example, suppose that we want to allocate broadcast frequencies to local radio stations. In the corresponding graph problem, each station is a node and the frequencies are the "colors." Two stations are connected by an edge if they are geographically too close together, so that they would interfere if they used the same frequency. This graph should not be too bad to color in practice, so long as we have a large enough supply of frequencies compared to the numbers of stations clustered near one another.

We can model a sudoku puzzle by setting up one node for each square. The colors are the 9 numbers, and some are pre-assigned to certain nodes. Two nodes are connected if their squares are in the same block or row or column. The puzzle is solvable if we can 9-color this graph, respecting the pre-assigned colors.

We can model exam scheduling as a coloring problem. The exams for two courses should not be put at the same time if there is a student who is in both courses. So we can model this as a graph, in which each course is a node and courses are connected by edges if they share students. The question is then whether we can color the graph with k colors, where k is the number of exam times in our schedule.

In the exam scheduling problem, we actually expect the answer to be

[1] In CS theory jargon, this problem is "HP-hard."

"no," because eliminating conflicts would require an excessive number of exam times. So the real practical problem is: how few students do we have to take out of the picture (i.e. give special conflict exams to) in order to be able to solve the coloring problem with a reasonable value for k. We also have the option of splitting a course (i.e. offering a scheduled conflict exam) to simplify the graph.

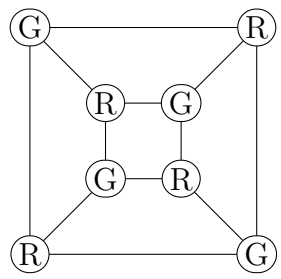
A particularly important use of coloring in computer science is register allocation. A large java or C program contains many named variables. But a computer has a smallish number (e.g. 32) of fast registers which can feed basic operations such as addition. So variables must be allocated to specific registers.

The nodes in this coloring problem are variables. The colors are registers. Two variables are connected by an edge if they are in use at the same time and, therefore, cannot share a register. As with the exam scheduling problem, we actually expect the raw coloring problem to fail. The compiler then uses so-called "spill" operations to break up the dependencies and create a graph we can color with our limited number of registers. The goal is to use as few spills as possible.

9.13 Bipartite graphs

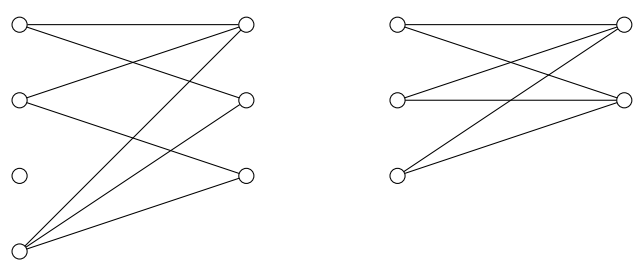
Another special type of graph is a bipartite graph. A graph $G = (V, E)$ is **bipartite** if we can split V into two non-overlapping subsets V_1 and V_2 such that every edge in G connects an element of V_1 with an element of V_2. That is, no edge connects two nodes from the same part of the division. Or, alternatively, a graph is bipartite if and only if it is 2-colorable.

For example, the cube graph is bipartite because we can color it with two colors:



Bipartite graphs often appear in matching problems, where the two subsets represent different types of objects. For example, one group of nodes might be students, the other group of nodes might be workstudy jobs, and the edges might indicate which jobs each student is interested in.

The complete bipartite graph $K_{m,n}$ is a bipartite graph with m nodes in V_1, n nodes in V_2, and which contains all possible edges that are consistent with the definition of bipartite. The diagram below shows a partial bipartite graph on a set of 7 nodes, as well as the complete bipartite graph $K_{3,2}$.



The complete bipartite graph $K_{m,n}$ has $m + n$ nodes and mn edges.

9.14 Variation in terminology

Although the core ideas of graph theory are quite stable, terminology varies a lot and there is a huge range of specialized terminology for specific types of graphs. In particular, nodes are often called "vertices" Notice that the singular of this term is "vertex" (not "vertice"). A complete graph on some

set of vertices is also known as a clique.

What we're calling a "walk" used to be widely called a "path." Authors who still use this convention would then use the term "simple path" to exclude repetition of vertices. Terms for closely-related concepts, e.g. cycle, often change as well.