Sets contain individual elements, and a set by itself has no extra structure—it holds its elements like a bag of loose sand. Things start to take off when we study ways that *pairs* of elements relate to one another, and our first example is the ***graph***.

Graphs come in many flavors; each one connects the elements of some finite set (called its ***vertices*** or ***nodes***) in some way via a finite set of ***edges***, each of which has a vertex at each of its two ***ends***—we call the vertices at the ends of an edge ***adjacent***. The rules for these edges determine the flavor.

- Directed and undirected graphs:
  - In an ***[undirected] graph***, the edges have no direction; they just connect vertices.
  - In a ***directed graph*** (also called a ***digraph***), each edge has an ***orientation***, or direction, which we usually indicate with an arrowhead; we consider each edge to run from the vertex at its ***tail*** to the vertex at its ***head***, just like the tail and head of an arrow.
- Loops and parallel edges:

  Be very careful to clearly identify what sort of graph you're working with in a given context, as some authors differ on their application of these terms!
  - A ***loop*** is an edge from a vertex to itself; a ***loopless*** graph has none.
  - A ***multigraph*** (which could be directed or undirected) may have multiple "parallel" edges connecting the same two vertices (in the same direction, for a digraph—two "opposite" edges aren't counted as parallel).
- A ***walk*** on a graph G from some vertex v_0 to some vertex v_n is an ordered list alternating between vertices and edges, beginning with v_0 and ending with v_n, such that each edge connects the two vertices adjacent to it in the list (and has the correct orientation, in the case of a digraph).

Incidence and Degree

- An edge is called ***incident to*** the vertices at its ends.
- The ***degree*** of a vertex v is the total number of edge-ends incident to it (loop-edges count twice!).
  - For a digraph, we have a more refined count: the ***in-degree*** of a vertex counts the number of heads of edges incident to the vertex, and the ***out-degree*** counts the tails at the vertex.
- A vertex having degree zero is called an ***isolated vertex***.
- A vertex having degree one is called a ***leaf vertex***, and the unique edge coming out of it is called a ***pendant edge*** (which literally "hangs from" the rest of the graph).
- An n***-regular*** graph is one in which *every* vertex has degree n.

Graph Isomorphism and Subgraphs

- Two graphs G and G' are called ***isomorphic*** (literally the adjective form of "same shape") if we can pair up the vertices of G one-to-one with those of G' and the edges of G one-to-one with those of G', such that the ends of each edge respect these pairings (as well as their direction, in the case of a digraph).
  - It is not always obvious whether or not two graphs are isomorphic—don't be misled by things (position, distance) not intrinsic to graphs when considering isomorphism—all that matters are the vertices and edges (and the edges' orientations, in the case of a digraph).
- A ***subgraph*** of a graph G is a smaller graph H living inside G, formed by taking some subset of the vertices and edges of the ***containing*** graph G; specifically:
  - the vertices of H can be *any* subset of the vertices of G, and
  - the edges of H can be any subset of the edges of G *whose ends are both in* H.

    H must be its own little graph—so its edges must connect *its* vertices.
- A subgraph H of G is called ***proper*** if it is not all of G.
- We'll often speak of some shape of graph (e.g., a path, cycle, or tree, as below) existing ***in a graph*** G—this means that there is a *subgraph* H of G that is *isomorphic* to that shape.
- A subgraph H of G with some property is called ***maximal*** if there is no subgraph of G with the same property that *properly contains* H
  - In other words, there is no way to add vertices and/or edges to H without losing that property.
- It's called a ***maximum*** if there is no larger subgraph with that property (regardless of containment).

Important graph shapes

- A ***path*** P_n consists of a $(n+1)$ vertices connected into a single chain by (n) edges, ***connecting*** its first vertex to its last vertex.
  - A ***directed path*** has all of its edges pointing int the direction from first to last vertex.
  - The ***length*** of a path (or cycle, below) is the number of *edges* it contains.
- A ***cycle*** C_n consists of (n) vertices connected into a single circuit by (n) edges.
  - A ***directed cycle*** has all of its edges pointing in a consistent direction.
- A ***complete graph*** K_n is a simple graph with n vertices and an edge between each pair of vertices.
  - A subgraph of a graph G isomorphic to K_n is called an n***-clique***.

Graph properties

- A ***simple*** graph is one that is undirected and has no loops or parallel edges; we can think of its edges as being given by a set of *unordered pairs* of vertices.
- A ***connected*** graph is one such that every pair of vertices can be connected by some path in the graph.
  - A digraph is called ***strongly connected*** if this can be done using *directed* paths in the graph.
- A ***component*** of a graph G is a *nonempty maximal connected subgraph* of G. Note that:
  - a connected graph has zero or one components, while a disconnected graph has at least two; and
  - these components being *maximal* has nothing to do with being the *biggest*—a graph could have one very small component and another very large one!
- An ***acyclic*** graph is one containing no cycles.
  - A ***DAG (directed acyclic graph)*** is a directed graph with no *directed* cycles.
- A ***bipartite*** graph is one whose vertices can be divided into *two* sets in such a way that all edges are between vertices of different sets (i.e., no edges occur between vertices of the same set!).
  - An n***-partite graph***'s vertices can be split into (n) sets in the same way.

Trees and Forests

- A ***tree*** is a graph that, equivalently:
  - can be constructed from a single vertex by iteratively adding a pendant edge and a new leaf vertex;
  - can be reduced to a single isolated vertex by iteratively removing one pendant edge and its leaf vertex; or
  - is nonempty, connected, and acyclic.
- In a ***rooted tree***, we choose one special vertex as its ***root***.
  - The root serves as the initial (or final) vertex in the constructive (or reductive) formulation above, respectively.
    - Note that *any* vertex of a given tree could serve as a root for it!
  - Each non-root vertex of a rooted tree has a unique ***parent*** leading toward the root and some number of ***children*** leading away from the root.
  - We typically don't consider the root to be a leaf vertex, even if it has valence 1, because we would never remove it in the constructive/reductive definitions of tree in this context.
  - In a ***binary [rooted] tree***, each parent has *exactly two* children; more generally, in an n***-ary tree***, each child has exactly (n) children.
- A ***spanning tree*** for a graph G is a is a tree T in G that contains all vertices of G.
  - A graph must be connected to have a spanning tree!
  - Every connected graph has a spanning tree.
  - Spanning trees are *maximal* trees in a connected graph G.
- A ***forest*** is a union of *disjoint* trees.
- A ***spanning forest*** for a graph G is a *maximal forest* in G.
  - Every graph has a spanning forest.
  - Spanning forests for G are made from one spanning tree for each component of the graph G.

Additional structures on graphs

We can attach all sorts of extra information to the vertices and/or edges of a graph: counts, probabilities, etc., This useful abstraction of the graph gives us a lens through which we can view *any problem* having to do with relationships between pairs of a set's elements through the lens of graphs.

At the same time, remember that a graph is dictated only by its set of vertices and edges, and how the edges connect those vertices—anything else we see in a diagram (position, length, intersections of edges, etc.) is an additional structure!

- Common vertex structures
  - We can *mark* a vertex (or set of vertices) as special, e.g., the root of a rooted tree.
  - We can attach a *color* (i.e., label) to each vertex of a vertex, in which case we usually insist that no adjacent vertices are assigned the same color.
    - If we use (n) colors, such a coloring is possible just when the graph is n-partite.
  - We can attach a *position* to each vertex, e.g., if each vertex represents a geographic location.
- Common edge structures
  - Attaching a scalar *weight* (or length, cost, etc.) to each edge of a graph gives a ***[edge-]weighted graph***, attach be attached to each edge of a graph in optimization problems.
  - Assigning a *probability* between 0 and 1 to each edge of a digraph, in such a way that at each vertex the sum of the outgoing probabilities is 1, gives a graph that can be used to represent a ***Markov chain***.
  - Assigning another label (e.g., a character) to each edge is used, e.g., in ***trie's*** and diagrams for ***finite automata***.

Important graph problems

- The ***isomorphism*** problem, determining whether or not two given graphs are isomorphic, *seems* quite difficult—but just how difficult it is is an open problem!
- ***Searching*** a graph is typically performed via one of two general strategies, ***depth-first (DFS)*** and ***breadth-first (BFS)***.
- Finding a valid n***-coloring*** of the vertices of a graph, or finding a graph's ***chromatic number*** (the minimum value of n for which the graph is n-colorable).
- *Combinatorial optimization* problems on a graph involve minimizing or maximizing some quantity, typically on a *weighted* graph, with the weights used to represent some quantity such as distance or capacity. Some of these can be solved efficiently:
  - finding a ***minimal spanning tree (MST)*** (a spanning tree with minimal total weight of its edges); [e.g., Prim's MST algorithm]
  - finding a ***shortest path*** from one vertex to another on an edge-weighted graph; [e.g., Dijkstra's SPT algorithm]
  - finding a ***maximum matching*** (a 1-regular subgraph with as many edges as possible); and [not in this course]
  - Finding ***maximum flow*** possible from one vertex to another on a weighted graph (with the weights representing flow capacity). [also not in this course]
- On the other extreme, the ***traveling salesman problem (TSP)*** on a weighted simple graph asks to find a *minimal-length cycle* containing all vertices of the graph, where the weights on edges give their individual lengths—this is a very hard problem!

Computer implementation

In some way or another, the vertices and edge information of a graph—and any additional structures—must be stored.

The two most common vertex-centric implemenations are:

- ***Adjacency matrices***: number the vertices $1, 2, \ldots, n$ and store adjacency information in an $n \times n$ matrix.
  - Simple to implement, but horribly inefficient for large, ***sparse*** graphs (which yield sparse matrices, i.e., matrices with lots of 0's)!
  - Matrix entries can store one piece of numerical information per vertex-pair (weight, probability, etc.).
- ***Adjacency lists****: for each vertex, store a list* of adjacent vertices. * Lists are one of many possible containers!
  - More efficient for sparse matrices, as only the adjacencies that exist are stored.

More complete graph implementations store both vertices and edges in separate containers, better reflecting the formal structure of a graph and allowing independent data to be stored for each.