Chapter 9

Graphs: Definition, Applications, Representation

9.1 Graphs and Relations

Graphs (sometimes referred to as networks) offer a way of expressing relationships between pairs of items, and are one of the most important abstractions in computer science.

Question 9.1. *What makes graphs so special?*

What makes graphs special is that they represent relationships. As you will (or might have) discover (discovered already) relationships between things from the most abstract to the most concrete, e.g., mathematical objects, things, events, people are what makes everything interesting. Considered in isolation, hardly anything is interesting. For example, considered in isolation, there would be nothing interesting about a person. It is only when you start considering his or her relationships to the world around, the person becomes interesting. Challenge yourself to try to find something interesting about a person in isolation. You will have difficulty. Even at a biological level, what is interesting are the relationships between cells, molecules, and the biological mechanisms.

Question 9.2. *Trees captures relationships too, so why are graphs more interesting?*

Graphs are more interesting than other abstractions such as tree, which can also represent certain relationships, because graphs are more expressive. For example, in a tree, tree cannot be cycles, and multiple paths between two nodes.

Question 9.3. *What do we mean by a "relationship"? Can you think of a mathematical way to represent relationships.*

What we mean by a relationship, is essentially anything that we can represent abstractly by

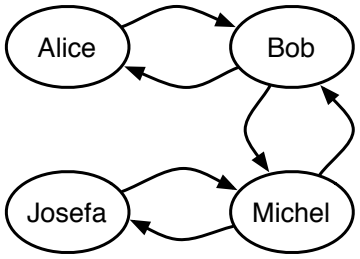


Figure 9.1: Friendship relation {(Alice, Bob), (Bob, Alice), (Bob, Michel), (Michel, Bob), (Josefa, Michel), (Michel, Josefa)} as a graph.

the mathematical notion of a relation. A *relation* is defined as a subset of the Cartesian product of two sets.

Exercise 9.4. *You can represent the friendship relation(ship) between people as a subset of the Cartesian product of the people, e.g, {(Alice, Bob), (Bob, Alice), (Josefa, Michel), (Michel, Josefa), (Bob, Michel), (Michel,Bob), . . . }.*

Now what is cool about graphs is that, they can represent any mathematical relation.

Question 9.5. *Can you see how to represent a (mathematical) relation with a graph?*

To represent a relation with a graph, we construct a graph, whose vertices represent the domain and the range of the relationship and connect the vertices with edges as described by the relation. Figure 9.1 shows an example. We will clarify what we mean by vertices and edges momentarily.

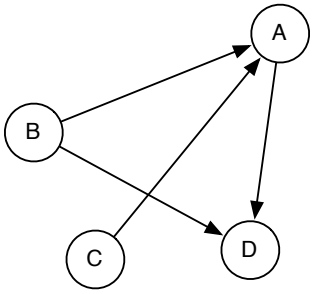
9.2 Defining Graphs

In order to be able to use graph abstractions, it is important for you to become acquainted with the terminology of graphs. In this section, we define graphs and summarize some of the terminology.

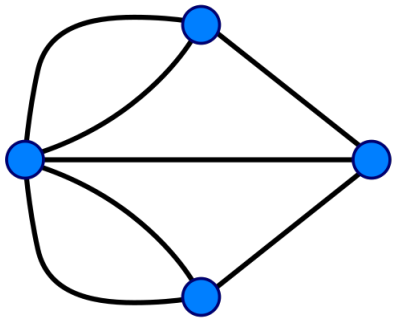
Directed graphs. Formally, a *directed graph* or (*digraph*) is a pair $G = (V, A)$ where

- 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 $e = (u, v)$. A digraph can have *self loops* (u, u). Directed graphs represent asymmetric relationships, e.g., my web page points to yours, but yours does not necessarily point back.



An example of a directed graph on 4 vertices.



An undirected graph on 4 vertices[1]

Figure 9.2: Example Graphs.

Exercise 9.6. *Identify the vertices and the arcs in the example digraph shown in Figure 9.1.*

Undirected graphs. Formally, an *undirected graph* is a pair $G = (V, E)$ where

- V is a set of *vertices* (or nodes), and
- $E \subseteq \binom{V}{2}$ is a set of edges.

In this case each edge is an unordered pair $e = \{u, v\}$ (or equivalently $\{v, u\}$). By this definition an undirected graph cannot have self loops since $\{v, v\} = \{v\} \notin \binom{V}{2}$. Undirected graphs represent symmetric relationships.

Question 9.7. *Can you think of a way to represent undirected graphs with digraphs.*

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 undirected graphs in data structures.

Graphs come with a lot of terminology, but 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. In Chapter 13 we will talk about weighted graphs, where the weights on edges can represent a distance, a capacity or the strength of the connection.

Neighbors. A vertex u is a *neighbor* of (or equivalently *adjacent* to) a vertex v in a graph $G = (V, E)$ if there is an edge $\{u, v\} \in E$. For a directed graph a vertex u is an *in-neighbor* of a vertex v if $(u, v) \in E$ and an *out-neighbor* if $(v, u) \in E$.

Neighborhood. For an undirected graph $G = (V, E)$, the *neighborhood* $N_G(v)$ of a vertex $v \in V$ is its set of all neighbors of v, i.e., $N_G(v) = \{u \mid \{u, v\} \in E\}$. 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. $N_G(U) = \bigcup_{u \in U} N_G(y)$, or $N_G^+(U) = \bigcup_{u \in U} N_G^+(u)$.

Degree. The *degree* $d_G(v)$ of a vertex $v \in V$ in a graph $G = (V, E)$ is the size of the neighborhood $|N_G(v)|$. For directed graphs we use *in-degree* $d_G^-(v) = |N_G^-(v)|$ and *out-degree* $d_G^+(v) = |N_G^+(v)|$. We will drop the subscript G when it is clear from the context which graph we're talking about.

Paths. A *path* in a graph is a sequence of adjacent vertices. More formally for a graph $G = (V, E)$, *Paths*$(G) = \{P \in V^+ \mid 1 \leq i < |P|, (P_i, P_{i+1}) \in E\}$ is the set of all paths in G, where V^+ indicates all positive length sequences of vertices (allowing for repeats). 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 path in a finite graph can have infinite length. A *simple path* is a path with no repeated vertices. Please see the remark below, however.

Remark 9.8. *Some authors use the terms walk for path, and path for simple path. Even in this book when it is clear from the context we will sometimes drop the "simple" from simple path.*

Reachability and connectivity. 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. We use $R_G(v)$ to indicate the set of all vertices reachable from v in G. An undirected graph is *connected* if all vertices are reachable from all other vertices. A directed graph is *strongly connected* if all vertices are reachable from all other vertices.

Cycles. In a directed graph a *cycle* is a path that starts and ends at the same vertex. A cycle can have length one (i.e. a *self loop*). A *simple cycle* is a cycle that has no repeated vertices other than the start and end vertices being the same. In an undirected graph a (simple) *cycle* is a path that starts and ends at the same vertex, has no repeated vertices other than the first and last, and has length at least three. In this course we will exclusively talk about simple cycles and hence, as with paths, we will often drop simple.

Exercise 9.9. *Why is important in a undirected graph to require that a cycle has length at least three? Why is important that we do not allow repeated vertices?*

Trees and forests. 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.

Directed acyclic graphs. A directed graph with no cycles is a *directed acyclic graph* (DAG).

Distance. The *distance* $\delta_G(u, v)$ from a vertex u to a vertex v in a graph G is the shortest path (minimum number of edges) from u to v. It is also referred to as the *shortest path length* from u to v.

Diameter. The *diameter* of a graph is the maximum shortest path length over all pairs of vertices: $\mathsf{diam}(G) = \max\{\delta_G(u, v) : u, v \in V\}$.

Multigraphs. Sometimes graphs allow multiple edges between the same pair of vertices, called *multi-edges*. Graphs with multi-edges are called *multi-graphs*. We will allow multi-edges in a couple algorithms just for convenience.

Sparse and dense graphs. 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 $m \ll n^2$ 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 book, is typically on algorithms that work well for sparse graphs.

9.3 Applications of Graphs

Since they are powerful abstractions, graphs can be very important in modeling data. In fact, many problems can be reduced to known graph problems. Here we outline just some of the many applications of graphs.

1. *Social network graphs: to tweet or not to tweet.* 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 follows 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.

2. *Transportation networks.* In road networks vertices are intersections and edges are the road segments between them, and for public transportation networks vertices are stops and edges are the links between them. Such networks are used by many map programs such as Google maps, Bing maps and now Apple IOS 6 maps (well perhaps without the public transport) to find the best routes between locations. They are also used for studying traffic patterns, traffic light timings, and many aspects of transportation.

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. *Network packet 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.

7. *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.

8. *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.

9. *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.

10. *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.

11. *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).

12. *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.

13. *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.

14. *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.

15. *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.

16. *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.

9.4 Representing Graphs

How we want to represent a graph largely depends on the operations we intend to support. For example we might want to do the following on a graph $G = (V, E)$:

(1) Map over the vertices $v \in V$.

(2) Map over the edges $(u, v) \in E$.

(3) Map over the neighbors of a vertex $v \in V$, or in a directed graph the in-neighbors or out-neighbors.

(4) Return the degree of a vertex $v \in V$.

(5) Determine if the edge (u, v) is in E.

(6) Insert or delete vertices.

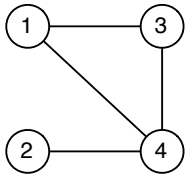


Figure 9.3: An undirected graph.

(7) Insert or delete edges.

Traditionally, there are four standard representations, all of which assume that vertices are numbered from $1, 2, \ldots, n$ (or $0, 1, \ldots, n-1$). As we consider different representations, we will illustrate how the graph shown in Figure 9.3 is represented using each one. In this chapter we will not worry about associating data or weights with the edges and leave that for Chapter 13.

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. For directed graphs the 1s can be in arbitrary positions.

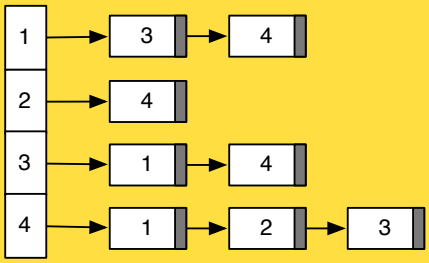
Example 9.10. *Using an adjacency matrix, the graph in Figure 9.3 is represented as follows.*

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

The main problem with adjacency matrices is their space demand of $\Theta(n^2)$. Graphs are often sparse, with far fewer edges than $\Theta(n^2)$.

Adjacency list. An array A of length n where each entry $A[i]$ contains a pointer to a linked list of all 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.

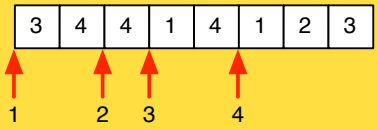
Example 9.11. *Using adjacency lists, the graph Figure 9.3 is represented as follows.*



Adjacency lists are not well suited for parallelism since the lists require that we traverse the neighbors of a vertex sequentially.

Adjacency array. Similar to an adjacency list, an adjacency array keeps the neighbors of all vertices, one after another, in an array `adj`; and separately, keeps an array of indices that tell us where in the `adj` array to look for the neighbors of each vertex.

Example 9.12. *Using an adjacency array, the graph Figure 9.3 is represented as follows.*



Edge list. A list of pairs $(i, j) \in E$.

Representing graphs, for parallel algorithms. In this book we take a more abstract view of the representation of graphs and instead of jumping right down to the low-level data structures such as arrays or linked-lists, we will consider representations based on sets and tables. We then view the standard representations as the special cases when using particular implementations of sets and tables. This makes it easier to apply parallel operations to the graphs. It also allows us to use arbitrary sets of vertices instead of requiring the vertices to labeled from $1, 2, \ldots, n$ (or $0, 1, \ldots, n-1$). The two representations we consider, edge sets and adjacency tables, are generalizations of edge lists and adjacency lists, respectively. Here we mostly consider directed graphs. To represent undirected graphs one can, for example, keep each edge in both directions, or in some cases just keep it in one direction.

Edge Sets. The simplest representation of a graph is based on its definition as a set of vertices V and a set of directed edges $A \subseteq V \times V$. If we use the set ADT, the keys for the edge set

are simply pairs of vertices. The representation is similar to the edge list representation, but it abstracts away from the particular data structure used for the set—the set could be implemented as a list, an array, a tree, or a hash table. Consider, for example, the tree-based cost specification for sets given in Chapter 7. For m edges this would allow us to determine if an arc (u, v) is in the graph with $O(\log m)$ work using a find, and allow us to insert or delete an arc (u, v) in the same work. We note that we will often use $O(\log n)$ instead of $O(\log m)$, where n is the number of vertices. This is OK to do since $m \leq n^2$, which means that $O(\log m)$ implies $O(\log n)$.

Although edge sets are efficient for finding, inserting, or deleting an edge, they are not efficient if we want to identify the neighbors of a vertex v. For example, finding the set of out edges requires a filter based on checking if the first element of each pair matches v:

$$\{(x, y) \in E \mid v = x\}$$

For m edges this requires $\Theta(m)$ work and $O(\log n)$ span, which is not efficient in terms of work. Indeed just about any representation of sets would require at least $O(m)$ work.

Adjacency Tables. To more efficiently access neighbors we will use adjacency tables, which are a generalization of adjacency lists and adjacency arrays. The *adjacency table* representation is a table that maps every vertex to the set of its (out) neighbors. This is simply an edge-set table.

Question 9.13. *What is the cost of accessing the neighbors of a vertex?*

In this representation, accessing the out neighbors of a vertex v is cheap since it just requires a lookup in the table. Assuming the tree cost model for tables, this can be done in $O(\log n)$ work and span.

Question 9.14. *Can you give an algorithm for finding an edge (u, v)?*

One can determine if a particular arc (u, v) is in the graph by first pulling out the adjacency set for u and then using a find to determine if v is in the set of neighbors. These both can be done in $O(\log n)$ work and span. Similarly inserting an arc, or deleting an arc can be done in $O(\log n)$ work and span. The cost of finding, inserting or deleting an edge is therefore the same as with edge sets. Note that in general, once the neighbor set has been pulled out, we can apply a constant work function over the neighbors in $O(d_G(v))$ work and $O(\log d_G(v))$ span.

Adjacency Sequences. A special case of adjacency tables are adjacency sequences. Recall that a sequence is a table with a domain taken from $\{0, \ldots, n-1\}$. However the cost model sequences allow for faster random access, requiring only $O(1)$ work to access the i^{th} element rather than $O(\log n)$. This allows, for example, accessing a vertex at less cost. This is traded off for the fact that certain operations, such as subselecting vertices, is more difficult.

Because of the reduced cost of access, we will sometimes use a sequence of sequence of integers (`(int seq) seq`) to represent a graph.

Costs. The cost of edge sets and adjacency tables is summarized with the following cost specification.

Cost Specification 9.15 (Graphs)**.** *The cost for various graph operations assuming a tree-based cost model for tables and sets and an array-based cost model for sequences. Assumes the function being mapped uses constant work and span. All costs are big-O.*

	edge set		***adj table***		***adj seq***	
	work	span	work	span	work	span
$(u, v) \stackrel{?}{\in} G$	$\log n$	$\log n$	$\log n$	$\log n$	n	$\log n$
map over edges	m	$\log n$	m	$\log n$	m	1
find neighbors	m	$\log n$	$\log n$	$\log n$	1	1
map over neighbors	$d_G(v)$	$\log n$	$d_G(v)$	$\log n$	$d_G(v)$	1
$d_G(v)$	m	$\log n$	$\log n$	$\log n$	1	1

.