Chapter 51

Graphs and their Representation

This chapter describes graphs and the techniques for representing them.

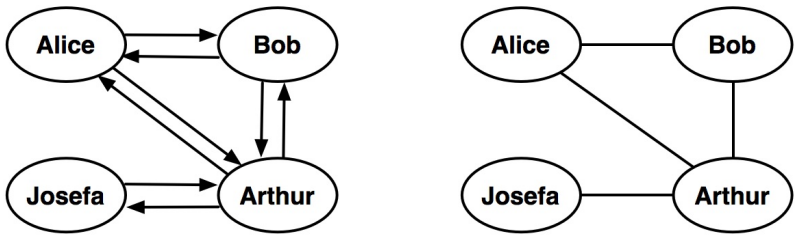
1 Graphs and Relations

Graphs (sometimes referred to as networks) are one of the most important abstractions in computer science. They are typically used to represent relationships between things from the most abstract to the most concrete, e.g., mathematical objects, people, events, and molecules. Here, what we mean by a "relationship" is essentially anything that we can represent abstractly by the mathematical notion of a relation. Recall that relation is defined as a subset of the Cartesian product of two sets.

Example 51.1 (Friends)**.** We can represent the friendship relation between a set of people P as a subset of the Cartesian product of the people, i.e., $G \subset P \times P$. If $P = \{Alice, Arthur, Bob, Josepha\}$ then a relation might be:

$$\{(Alice, Bob), (Alice, Arthur), (Bob, Alice), (Bob, Arthur),\\ (Arthur, Josefa), (Arthur, Bob), (Arthur, Alice), (Josefa, Arthur)\}$$

This relation can then be represented as a directed graph where each arc denotes a member of the relation or as an undirected graph where directed edge denotes a pair of the members of the relation of the form (a, b) and (b, a).



Background on Graphs. Chapter 6 presents a brief review of the graph terminology that we use in this book. The rest of this chapter and other chapters on graphs assumes familiarity with the basic graph terminology reviewed therein.

2 Applications of Graphs

Graphs are used in computer science to model many different kinds of data and phenomena. This section briefly mentions some of the many applications of graphs.

Social Network Graphs. In social network graphs, vertices are people and edges represent relationships among the people, such as who knows whom, who communicates with whom, who influences whom or others in social organizations. 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, etc.

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 widely used map applications to find the best routes between locations. They are also used for studying traffic patterns, traffic light timings, and many aspects of transportation.

Utility Graphs. In utility graphs, vertices are junctions and edges are conduits between junctions. Examples include the power grid carrying electricity throughout the world, the internet with junctions being routers, and the water supply network with the conduits being physical pipes. 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.

Document-Link Graphs. In document-link graphs, vertices are a set of documents, and edges are links between documents. The best 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.

Graphs in Compilers. Graphs are used extensively in compilers. Vertices can represent variables, instructions, or blocks of code, and the edges represent relationships among them. Often one of the first steps of a compiler is to turn the written syntax for the program into a graph, which is then manipulated. Such graphs can be used for type inference, for so called data flow analysis, register allocation and many other purposes.

Robot Motion Planning. Vertices represent the states a 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.

Neural Networks and Deep Learning. 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. Neural networks are also used for learning a variety of relationships from large data sets.

Protein-Protein Interactions Graphs. Vertices represent proteins and edges represent interactions between them; such interactions usually correspond to biological functions of proteins. 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.

Finite-Element Meshes. Vertices are cells in space, and edges represent neighboring cells. 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 (cells), and modeling the interaction of neighboring elements as a graph.

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.

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.

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.

Constraint Graphs. Vertices are items and edges represent constraints among them. 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.

Dependence Graphs. Vertices are tasks or jobs that need to be done, and edges are constraints specifying what tasks needs to be done before other tasks. The edges 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. These graphs should be acyclic.

3 Graphs Representations

We can represent graphs in many different ways. To choose an efficient and fast representation for a graph, it is important to know the kinds of operations that are needed by the algorithms that we expect to use on that graph. Common operations a graph $G = (V, E)$ include the following.

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

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

(3) Map a function over the (in or out) neighbors of a vertex $v \in V$.

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

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

(6) Insert or delete an isolated vertex.

(7) Insert or delete an edge.

Different representations do better on some operations and worse on others. Cost can also depend on the ***density*** of the graph, i.e. the relationship of the number of vertices and number of edges.

To enable high-level, mathematical reasoning about algorithms, we represent graphs by using the abstract data types such as sequences , sets , and tables . This approach enables specifying the algorithms at a high level and then selecting the lowest cost implementation for each algorithm.

Assumptions.

- In the rest of the chapter, we focus on directed graphs. To represent undirected graphs, we can simply keep each edge in both directions. In some cases, it suffices to keep an edge in just one direction.
- For the following discussion, consider a graph $G = (V, E)$ with n vertices and m edges.
- Throughout we assume that we only delete isolated vertices. If a vertex is incident on edges, then this means that we first have to delete the edges before deleting the vertex.

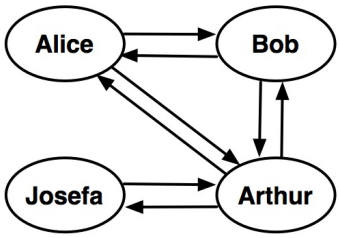
3.1 Edge Sets

Perhaps the simplest representation of a graph is based on its definition. Assuming we have a universe of possible vertices $\mathcal{V}$ (e.g., the integers, or character strings), we can represent directed graphs in that universe as:

$$G = (\mathcal{V}\ set,\ (\mathcal{V} \times \mathcal{V})\ set).$$

The $(\mathcal{V}\ set)$ is the set of vertices and the $((\mathcal{V} \times \mathcal{V})\ set)$ is the set of directed edges. The sets could be represented with lists, arrays, trees, or hash tables.

Example 51.2. Using the edge-set representation, the directed graph



can be prepresented as:

$$\begin{array}{rcl} \mathcal{V} & = & string \\ V & = & \{\text{Alice}, \text{Arthur}, \text{Bob}, \text{Josefa}\} : \mathcal{V}\ set \\ E & = & \{(\text{Alice}, \text{Bob}), (\text{Alice}, \text{Arthur}), (\text{Josefa}, \text{Arthur}), (\text{Bob}, \text{Arthur}), \\ & & (\text{Arthur}, \text{Josefa}), (\text{Arthur}, \text{Bob}), (\text{Arthur}, \text{Alice}), (\text{Bob}, \text{Alice})\} \\ & & : (\mathcal{V} \times \mathcal{V})\ set \end{array}$$

Consider the tree-based cost specification for sets. Using edge sets for a graph with m edges, we can determine if a directed edge (u, v) is in the graph with $O(\lg m) = O(\lg n)$ work using a *find*, and *insert* or *delete* an edge (u, v) in the same work.

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 of v requires filtering the edges whose first element matches v:

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

For m edges this requires $\Theta(m)$ work and $\Theta(\lg n)$ span, which is not work efficient.

Exercise 51.1. Prove that for a graph with n vertices and m edges, $O(\lg m) = O(\lg n)$.

Solution. For any graph, we have $m \leq n^2$ and therefore $O(\lg m) = O(\lg n)$.

Cost Specification 51.1 (Edge Sets for Graphs)**.** For a graph represented as $G = (\mathcal{V}\ set, (\mathcal{V} \times \mathcal{V})\ set)$ and assuming a tree-based cost model for sets, we have the following costs for

common graph operations.

	Work	**Span**
Map a function over all vertices $v \in V$	$\Theta(n)$	$\Theta(\lg n)$
Map a function over all edges $(u, v) \in E$	$\Theta(m)$	$\Theta(\lg n)$
Map a function over neighbors of a vertex	$\Theta(m)$	$\Theta(\lg n)$
Find the degree of a vertex	$\Theta(m)$	$\Theta(\lg n)$
Is edge $(u, v) \in E$	$\Theta(\lg n)$	$\Theta(\lg n)$
Insert or delete a vertex	$\Theta(\lg n)$	$\Theta(\lg n)$
Insert or delete an edge	$\Theta(\lg n)$	$\Theta(\lg n)$

This assumes the function being mapped has constant work and span. For vertex deletion, we assume that the vertex is isolated (has no incident edges).

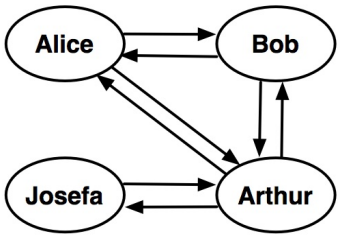
Exercise 51.2. What is the cost of deleting a vertex with out-degree d?

3.2 Adjacency Tables

Definition 51.2 (Adjacency Table Representation)**.** The ***adjacency-table*** representation of a graph consists of a table mapping every vertex to the set of its out-neighbors and can be defined as

$$G = (\mathcal{V} \times (\mathcal{V}\ set))\ table.$$

Example 51.3. Using the adjacency-table representation, the directed graph



can be prepresented as:

$$\{ \tag{51.1}$$
$$\text{Alice} \mapsto \{\text{Arthur}, \text{Bob}\}, \tag{51.2}$$
$$\text{Bob} \mapsto \{\text{Alice}, \text{Arthur}\}, \tag{51.3}$$
$$\text{Arthur} \mapsto \{\text{Alice}, \text{Josefa}\}, \tag{51.4}$$
$$\text{Josefa} \mapsto \{\text{Arthur}\} \tag{51.5}$$
$$\} \tag{51.6}$$

The adjacency-table representation supports efficient access to the out neighbors of a vertex by using a table lookup. Assuming the tree-based cost model for tables, this requires $\Theta(\lg n)$ work and span.

We can check if a directed edge (u, v) is in the graph by first obtaining the adjacency set for u, and then using a *find* operation to determine if v is in the set of neighbors. Using a tree-based cost model, this requires $\Theta(\lg n)$ work and span.

Inserting an edge, or deleting an edge requires $\Theta(\lg n)$ work and span. The cost of finding, inserting or deleting an edge is therefore the same as with edge sets.

Note that after we find the out-neighbor set of a vertex, we can apply a constant work function over the neighbors in $\Theta(d_G(v))$ work and $\Theta(\lg d_G(v))$ span.

Cost Specification 51.3 (Adjacency Tables)**.** For a graph represented as $G = (\mathcal{V} \times (\mathcal{V}\ set))\ table$ and assuming a tree-based cost model for sets and tables, we have that:

Operation	**Work**	**Span**
Map a function over all vertices $v \in V$	$\Theta(n)$	$\Theta(\lg n)$
Map a function over all edges $(u, v) \in E$	$\Theta(m)$	$\Theta(\lg n)$
Map a function over neighbors of a vertex	$\Theta(\lg n + d_g(v))$	$\Theta(\lg n)$
Find the degree of a vertex	$\Theta(\lg n)$	$\Theta(\lg n)$
Is edge $(u, v) \in E$	$\Theta(\lg n)$	$\Theta(\lg n)$
Insert or delete a vertex	$\Theta(\lg n)$	$\Theta(\lg n)$
Insert or delete an edge	$\Theta(\lg n)$	$\Theta(\lg n)$

This assumes the function being mapped uses constant work and span.

Note. The adjacency-table representation is more efficient than the edge-set only for operations that involve operating locally on individual vertices and their out-edges.

3.3 Adjacency Sequences

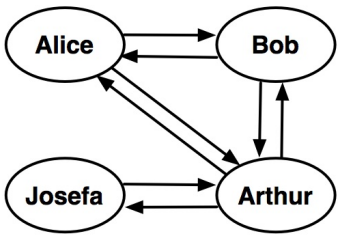
Definition 51.4 (Adjacency Sequences for Enumerable Graphs)**.** For enumerable graphs $G = (V, E)$, where $V = \{0 \ldots (n-1)\}$, we can use sequences to improve the efficiency of the adjacency table representation. Sequences can be used for both the outer table and inner set. The type of a graph in this representation is thus

$$G = (int\ seq)\ seq.$$

Here the length of the outer sequence in n, and the length of each inner sequences equals the degree of the corresponding vertex.

This representation allows for fast random access, requiring only $\Theta(1)$ work to access the i^{th} vertex.

Example 51.4. We can relabel the directed graph



by assigning the labels $0, 1, 2, 3$ to Alice, Arthur, Bob, Josefa respectively. We can represent the resulting enumerable graph with the following adjacency sequence:

$$\langle \tag{51.7}$$
$$\quad \langle 1, 2 \rangle, \tag{51.8}$$
$$\quad \langle 0, 2, 3 \rangle, \tag{51.9}$$
$$\quad \langle 0, 1 \rangle, \tag{51.10}$$
$$\quad \langle 1 \rangle \tag{51.11}$$
$$\rangle. \tag{51.12}$$

Exercise 51.3. What is the cost of finding the out-neighbors of a vertex?

Solution. It is $\Theta(1)$ work and span.

Exercise 51.4. Why does the cost of mapping over all edges require $\Omega(n)$ work?

Solution. Because any algorithm that maps over all the edges must touch each and every vertex.

Cost Specification 51.5 (Adjacency Sequence)**.** Consider a graph with vertices $V = \{0, \ldots, n-1\}$, represented as $G = (int\ seq)\ seq$. Assuming an (persistent) array-sequence cost model, the cost of the key graph operations are as follows.

Operation	**Work**	**Span**
Map a function over all vertices $v \in V$	$\Theta(n)$	$\Theta(1)$
Map a function over all edges $(u, v) \in E$	$\Theta(n+m)$	$\Theta(1)$
Map a function over neighbors of a vertex	$\Theta(d_g(v))$	$\Theta(1)$
Find the degree of a vertex	$\Theta(1)$	$\Theta(1)$
Is edge $(u, v) \in E$	$\Theta(d_g(u))$	$\Theta(\lg d_g(u))$
Insert or delete a vertex	$\Theta(n)$	$\Theta(1)$
Insert or delete an edge	$\Theta(n)$	$\Theta(1)$

For mapping over vertices and edges, we assume that the function being mapped has constant work and span. For vertex deletions, we assume that all edges incident on the vertex has been removed and (the vertex is isolated).

Using ephemeral array sequences can improve work efficiency of deletion operations. A vertex can be marked deleted in $O(1)$ work. An edge can likewise be found and marked deleted in $\Theta(\lg d_g(n))$ work. Insertions can also be done more efficiently by an "array doubling technique", which yields amortized constant work per update. Recall, however, that ephemeral operations can be tricky to use in parallel algorithms.

Exercise 51.5. Give a constant-span algorithm for deleting an edge from a graph.

Hint (Finding an Element via Injection). Give a linear-work, constant-span algorithm for finding the position of an element in a sequence in constant span using sequence inject .

Hint (Deleting an Element). Give a linear-work, constant-span algorithm for deleting an element at a specified position in a sequence.

Adjacency List Representation. In the adjacency sequence representation, we can represent the inner sequences (the out-neighbor sequence of each vertex) by using arrays or lists. If we use lists, then the resulting representation is the same as the classic ***adjacency list*** representation of graphs. This is a traditional representation used in sequential algorithms. It is not well suited for parallel algorithms since traversing the adjacency list of a vertex will take span proportional to its degree.

Mixed Adjacency Sequences and Tables. It is possible to mix adjacency tables and adjacency sequences by having either the inner sets or the outer table be a sequence, but not both. Using sequences for the inner sets has the advantage that it defines an ordering over the edges of a vertex. This can be helpful in some algorithms.

3.4 Adjacency Matrices

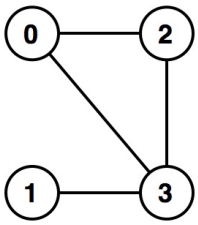
For enumerable graphs that are dense (i.e., m is not much smaller than n^2), representing the graph as a boolean matrix can make sense. For a graph with n vertices such a matrix has n rows and n columns and contains a `true` (or 1) in location (i, j) (i-th row and j-th column) if and only if $(i, j) \in E$. Otherwise it contains a `false` (or 0). For undirected graphs the matrix is symmetric and contains `false` (or 0) along the diagonal since undirected graphs have no self edges. For directed graphs the `true`s (1s) can be in arbitrary positions.

A matrix can be represented as a sequence of sequences of booleans (or zeros and ones), for which the type of the representation is:

$$G = (bool\ seq)\ seq$$

For a graph with n vertices the outer sequence and all the inner sequences have equal length n.

Example 51.5. The graph:



has the adjacency matrix:

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

which can be represented as the nested sequence:

$$\langle\ \langle 0,0,1,1\rangle, \tag{51.13}$$
$$\langle 0,0,0,1\rangle, \tag{51.14}$$
$$\langle 1,0,0,1\rangle, \tag{51.15}$$
$$\langle 1,1,1,0\rangle\ \rangle. \tag{51.16}$$

Cost Specification 51.6 (Adjacency Matrix)**.** For a graph represented as an adjacency matrix with $V = \{0, \ldots, n-1\}$ and $G = (bool\ seq)\ seq$, and assuming the an array-sequence cost model, we have that:

Operation	**Work**	**Span**
Map a function over all vertices $v \in V$	$\Theta(n)$	$\Theta(1)$
Map a function over all edges $(u,v) \in E$	$\Theta(n^2)$	$\Theta(1)$
Map a function over neighbors of a vertex	$\Theta(n)$	$\Theta(1)$
Find the degree of a vertex	$\Theta(n)$	$\Theta(\lg n)$
Is edge $(u,v) \in E$	$\Theta(1)$	$\Theta(1)$
Insert or delete a vertex	$\Theta(n^2)$	$\Theta(1)$
Insert or delete an edge	$\Theta(n)$	$\Theta(1)$

These costs assume the function being mapped uses constant work and span.

As with other representations, using ephemeral sequences can improve the efficiency of update operations (insert/delete a vertex or an edge) to amortized constant time.

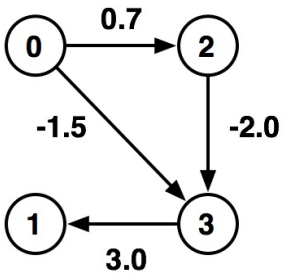
Exercise 51.6. Give a constant-span algorithm for computing the complement of a graph.

Solution. Map over the vertices a function that for each out-edge complements the boolean.

3.5 Representing Weighted Graphs

Many applications of graphs require associating values with the edges of a graph resulting in weighted or labeled graphs . This section presents several techniques for representing such graphs.

Example 51.6. An example directed weighted graph.



Label Table. This chapter covered three different representations of graphs: edge sets, adjacency tables, and adjacency sequences. We can extend each of these representations to support edge-weights by representing the function from edges to weights as a separate table. This ***weight table*** maps each edge to its value and allows finding weight of an edge $e = (u, v)$ by using a table lookup.

Example 51.7. For the weighted graph shown above the weight table is:

$$W = \{(0,2) \mapsto 0.7,\ (0,3) \mapsto -1.5, (2,3) \mapsto -2.0, (3,1) \mapsto 3.0\}\,.$$

Weight tables work uniformly with all graph representations, and they are elegant, because they separate edge weights from the structural information. However, keeping a separate weight table creates redundancy, wasting space and possibly requiring additional work to lookup the weights. We can eliminate the redundancy by storing the weights directly with the edge.

For example, when using the edge-set representation for graphs, we can keep the weight along with the edge in the edge sets. Similarly, when using the adjacency-table representation, we can replace each set of neighbors with a set consisting of the neighbor and the weight of the edge from the vertex to the neighbor. Finally, we can extend an adjacency sequences by creating a sequence of neighbor-weight pairs for each out edge of a vertex.

Example 51.8. For the weighted graph shown above the adjacency table representation is

$$G = \{0 \mapsto \{2 \mapsto 0.7, 3 \mapsto -1.5\}\,, 2 \mapsto \{3 \mapsto -2.0\}\,, 3 \mapsto \{1 \mapsto 3.0\}\}\,,$$

and the adjacency sequence representation is

$$G = \langle\, \langle\, (2, 0.7),\ (3, -1.5)\, \rangle\,, \langle\, \rangle\,, \langle\, (3, -2.0)\, \rangle\,, \langle\, (1, 3.0)\, \rangle\, \rangle\,.$$