1

Graph theory concepts and definitions used in image processing and analysis

Olivier Lézoray

Université de Caen Basse-Normandie
GREYC UMR CNRS 6072
6 Bvd. Maréchal Juin
F-14050 CAEN, FRANCE
Email: olivier.lezoray@unicaen.fr

Leo Grady

Department of Image Analytics and Informatics
Siemens Corporate Research
755 College Rd.
Princeton, NJ 08540, USA
Email: Leo.Grady@siemens.com

CONTENTS

1.1 Introduction

Graphs are structures that have a long history in mathematics and have been applied in almost every scientific and engineering field (see [1] for mathematical history, and [2, 3, 4] for popular accounts). Many excellent primers on the mathematics of graph theory may be found in the literature [5, 6, 7, 8, 9, 10], and we encourage the reader to learn the basic mathematics of graph theory from these sources. We have two goals for this chapter: First, we make this work self-contained by reviewing the basic concepts and notations for graph theory that are used throughout this book. Second, we connect these concepts to image processing and analysis from a conceptual level and discuss implementation details.

1.2 Basic Graph Theory

Intuitively, a graph represents a set of elements and a set of pairwise relationships between those elements. The elements are called **nodes** or **vertices**, and the relationships are called **edges**. Formally, a **graph** $\mathcal{G}$ is defined by the sets $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ in which $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$. We may denote the i-th vertex as $v_i \in \mathcal{V}$, and the i-th edge as $e_i \in \mathcal{E}$. Since each edge is a subset of two vertices, we may also write $e_{ij} = \{v_i, v_j\}$. In this book we do not consider graphs with self-loops, meaning that $e_{ij} \in \mathcal{E}$ implies that $i \neq j$. We also do not consider graphs for which there are multiple edges of the same orientation connecting the same node pair. However, for specific graph encodings, self-loops and multiple edges can be useful (see Chapter 11).

Every graph may be viewed as *weighted*. Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, vertex weighting is a function $\hat{w} : \mathcal{V} \to \mathbb{R}$, and edge weighting is a function $w : \mathcal{E} \to \mathbb{R}$. To simplify the notation, we will use w to refer to both vertex and edge weighting. The weight of a vertex is denoted by $w(v_i)$ or w_i, and the weight of an edge incident to two vertices is denoted by $w(v_i, v_j)$ or w_{ij}. If $w_i = 1,\ \forall v_i \in \mathcal{V}$ and $w_{ij} = 1,\ \forall e_{ij} \in \mathcal{E}$, then we may consider the graph to be *unweighted*. If not otherwise specified, all node and edge weights are considered to be equal to unity. We treat $w_{ij} = 0$ as equivalent to $e_{ij} \notin \mathcal{E}$. Intuitively, an edge weight of zero is equivalent to meaning that the edge is not a member of the edge set.

Each edge is considered to be **oriented** and some edges additionally **directed**. An *orientation* of an edge means that each edge $e_{ij} \in \mathcal{E}$ contains an ordering of the vertices, v_i and v_j. An edge e_{ij} is *directed* if $w_{ij} \neq w_{ji}$. A graph for which none of the edges are directed is called an **undirected graph**, and a graph in which at least one edge is directed is called a **directed graph** or **digraph**. A directed edge is represented by the notation $e_{i \to j}$. A directed graph is more general than an undirected

graph because it does not require that $w_{ij} = w_{ji}$ for each edge. Consequently, all algorithms for directed graphs may also be applied to undirected graphs, but the converse may or may not be true. Therefore, in this chapter we use digraphs to illustrate the most general concepts.

The edge orientation of an undirected graph provides a reference to determine the *sign* of a flow through that edge. This concept was developed early in the circuit theory literature to describe the direction of current flow through a resistive branch (an edge). For example, a current flow through e_{ij} from node v_i to v_j is considered positive, while a current flow from v_j to v_i is considered negative. In this sense, flow through a directed edge is usually constrained to be strictly positive.

Drawing a graph is typically done by depicting each node with a circle and each edge with a line connecting circles representing the two nodes. The edge orientation or direction is usually represented by an arrow. Edge e_{ij} is drawn with an arrow pointing from node v_i to node v_j.

We consider two functions $s, t : \mathcal{E} \to \mathcal{V}$. Function s is called the **source function**, and function t is called the **target function**. Given an edge $e_{ij} = (v_i, v_j) \in \mathcal{E}$, we say that $s(e_{ij}) = v_i$ is the **origin** or **source** of e_{ij}, and $t(e_{ij}) = v_j$ is the **endpoint** or **target** of e_{ij}. Given any edge, $e_k \in \mathcal{E}$, the vertices $s(e_k)$ and $t(e_k)$ are called the **boundaries** of e_k, and the expression $t(e_k) - s(e_k)$ is called the boundary of e_k.

Given these preliminaries, we may now list a series of basic definitions:

1. *Adjacent*: Two nodes v_i and v_j are called **adjacent** if $\exists e_{ij} \in \mathcal{E}$ or $\exists e_{ji} \in \mathcal{E}$, which is denoted by $v_i \sim v_j$. Two edges e_{ij} and e_{sk} are adjacent if they share a common vertex, that is, if $i = s$, $i = k$, $j = s$, or $j = k$.

2. *Incident*: An edge e_{ij} is **incident** to nodes v_i and v_j (and each node is incident to the edge).

3. *Isolated*: A node v_i is called **isolated** if $w_{ij} = w_{ji} = 0,\ \forall v_j \in \mathcal{V}$. Intuitively, a node is isolated if it is not connected to the graph by an edge (of nonzero weight).

4. *Degree*: The **outer degree** of node v_i, $\deg^-(v_i)$, is equal to $\deg^-(v_i) = \sum_{e_{ij} \in \mathcal{E}} w_{ij}$. The **inner degree** of node v_i, $\deg^+(v_i)$, is equal to $\deg^+(v_i) = \sum_{e_{ji} \in \mathcal{E}} w_{ji}$. Note that, in an undirected graph, $\deg^-(v_i) = \deg^+(v_i),\ \forall v_i \in \mathcal{V}$. Consequently, we may simply use $d(v_i)$ or d_i to denote the degree of a vertex in an undirected graph. An isolated node v_i has zero degree.

5. *Complement*: A graph $\overline{\mathcal{G}} = (\mathcal{V}, \overline{\mathcal{E}})$ is called the **complement** to graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ if $\overline{\mathcal{E}} = \mathcal{V} \times \mathcal{V} - \mathcal{E}$. Therefore, the complement graph has all the same nodes as the original, but each node pair in the complement is connected iff the node pair was not connected in the original graph.

6. *Regular*: An undirected graph is called **regular** if $d_i = k,\ \forall v_i \in \mathcal{V}$ for some constant k.

7. *Complete (fully connected)*: An undirected graph is called **complete** or **fully connected** if each node is connected to every other node by an edge, that is when $\mathcal{E} = \mathcal{V} \times \mathcal{V}$. A complete graph of n vertices is denoted K_n. Figure 1.1 gives an example of K_5.

8. *Partial graph*: A graph $\mathcal{G}' = (\mathcal{V}, \mathcal{E}')$ is called a **partial graph** [11] of $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ if $\mathcal{E}' \subseteq \mathcal{E}$.

9. *Subgraph*: A graph $\mathcal{G}' = (\mathcal{V}', \mathcal{E}')$ is called a **subgraph** of $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ if $\mathcal{V}' \subseteq \mathcal{V}$, and $\mathcal{E}' = \{e_{ij} \in \mathcal{E} | v_i \in \mathcal{V}' \text{ and } v_j \in \mathcal{V}'\}$.

10. *clique*: A **clique** is defined as a fully connected subset of the vertex set (see Figure 1.2).

11. *Bipartite*: A graph is called **bipartite** if $\mathcal{V}$ can be partitioned into two subsets $\mathcal{V}_1 \subset \mathcal{V}$ and $\mathcal{V}_2 \subset \mathcal{V}$, where $\mathcal{V}_1 \cap \mathcal{V}_2 = \emptyset$ and $\mathcal{V}_1 \cup \mathcal{V}_2 = \mathcal{V}$, such that $\mathcal{E} \subseteq \mathcal{V}_1 \times \mathcal{V}_2$. If $|\mathcal{V}| = m$ and $|\mathcal{V}| = n$, and $\mathcal{E} = \mathcal{V}_1 \times \mathcal{V}_2$, then $\mathcal{G}$ is called a **complete bipartite** graph and is denoted by $K_{m,n}$. Figure 1.2 gives an example of $K_{3,5}$.

12. *Graph isomorphism*: Let $\mathcal{G}_1 = (\mathcal{V}_1, \mathcal{E}_1)$ and $\mathcal{G}_2 = (\mathcal{V}_2, \mathcal{E}_2)$ be two undirected graphs. A bijection $f : \mathcal{V}_1 \to \mathcal{V}_2$ from $\mathcal{G}_1$ to $\mathcal{G}_2$ is called a **graph isomorphism** if $(v_i, v_j) \in \mathcal{E}_1$ implies that $(f(v_i), f(v_j)) \in \mathcal{E}_2$. Figure 1.3 shows an example of an isomorphism between two graphs.

13. *Higher-order graphs (hypergraph)*: A graph $G = (\mathcal{V}, \mathcal{E}, \mathcal{F})$ is considered to be a **higher-order graph** or **hypergraph** if each element of $\mathcal{F}$, $f_i \in \mathcal{F}$ is defined as a set of nodes for which each $|f_i| > 2$. Each element of a higher-order set is called a **hyperedge**, and each hyperedge may also be weighted. A k**-uniform hypergraph** is one for which all hyperedges have size k. Therefore, a 3-uniform hypergraph would be a collection of node triplets.

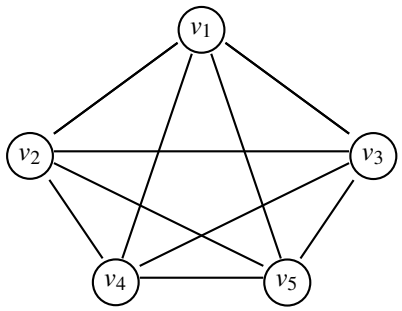


FIGURE 1.1
An example of complete graph K_5.

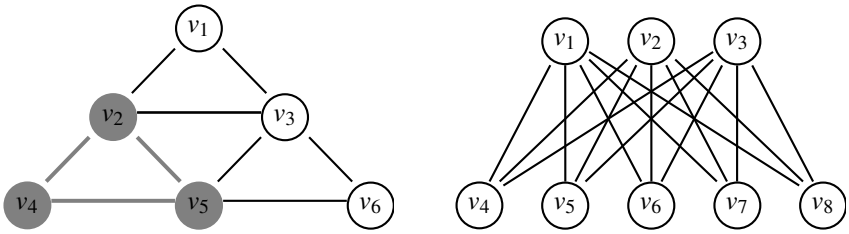


FIGURE 1.2
Left: A clique in a graph (grey). Right: The complete bipartite graph $K_{3,5}$ with $\mathcal{V}_1 = \{v_1, v_2, v_3\}$ and $\mathcal{V}_2 = \{v_4, v_5, v_6, v_7, v_8\}$.

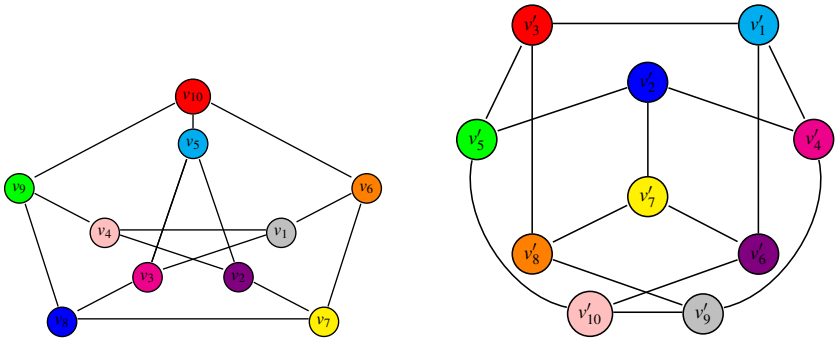


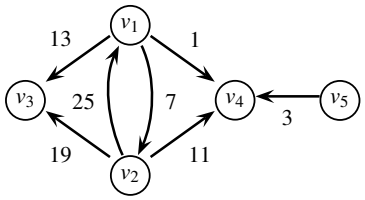
FIGURE 1.3
Example of two isomorphic graphs under the mapping $v_{10} \to v'_3$, $v_9 \to v'_5$, $v_8 \to v'_2$, $v_7 \to v'_7$, $v_6 \to v'_8$, $v_5 \to v'_1$, $v_4 \to v'_{10}$, $v_3 \to v'_4$, $v_2 \to v'_6$, $v_1 \to v'_9$.

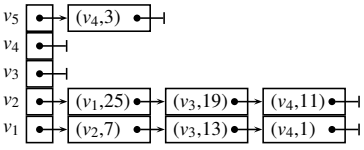
1.3 Graph Representation

Several computer representations of graphs can be considered. The data structures used to represent graphs can have a significant influence on the size of problems that can be performed on a computer and the speed with which they can be solved. It is therefore important to know the different representations of graphs. To illustrate them, we will use the graph depicted in Figure 1.4.

1.3.1 Matrix Representations

A graph may be represented by one of several different common matrices. A matrix representation may provide efficient storage (since most graphs used in image processing are sparse), but each matrix representation may also be viewed as an *operator* that acts on functions associated with the nodes or edges of a graph.





	v_1	v_2	v_3	v_4	v_5
v_1	0	7	13	1	0
v_2	25	0	19	11	0
v_3	0	0	0	0	0
v_4	0	0	0	0	0
v_5	0	0	0	3	0

	e_{12}	e_{13}	e_{14}	e_{21}	e_{23}	e_{24}	e_{54}
v_1	+1	+1	+1	-1	0	0	0
v_2	-1	0	0	+1	+1	+1	0
v_3	0	-1	0	0	-1	0	0
v_4	0	0	-1	0	0	-1	-1
v_5	0	0	0	0	0	0	+1

FIGURE 1.4
From top-left to bottom-right: a weighted directed graph, its adjacency list, its adjacency matrix, and its (transposed) incidence matrix representations.

The first matrix representation of a graph is given by the **incidence matrix**. The incidence matrix for $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is an $|\mathcal{E}| \times |\mathcal{V}|$ matrix $\mathbf{A}$ where

$$\mathbf{A}_{ij} = \begin{cases} -1 \text{ if } s(e_i) = v_j, \\ +1 \text{ if } t(e_i) = v_j, \\ \quad 0 \text{ otherwise.} \end{cases} \tag{1.1}$$

The incidence matrix has the unique property of the matrices in this section that it preserves the orientation information of each edge but not the edge weight. The adjoint (transpose) of the incidence matrix also represents the *boundary operator* for a graph in the sense that multiplying this matrix with a signed indicator vector of edges in a path will return the endpoints of the path. Furthermore, the incidence matrix defines the exterior derivative for functions associated with the nodes of the graph. As such, this matrix plays a central role in *discrete calculus* (see [12] for more details).

The **constitutive matrix** of graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is an $|\mathcal{E}| \times |\mathcal{E}|$ matrix $\mathbf{C}$ where

$$\mathbf{C}_{ij} = \begin{cases} w(e_i) \text{ if } i = j, \\ \quad 0 \text{ otherwise.} \end{cases} \tag{1.2}$$

The **adjacency matrix** representation of graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a $|\mathcal{V}| \times |\mathcal{V}|$ matrix $\mathbf{W}$ where

$$\mathbf{W}_{ij} = \begin{cases} w_{ij} \text{ if } e_{ij} \in \mathcal{E}, \\ \quad 0 \text{ otherwise.} \end{cases} \tag{1.3}$$

For undirected graphs the matrix $\mathbf{W}$ is symmetric.

The **Laplacian matrix** of an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a $|\mathcal{V}| \times |\mathcal{V}|$ matrix $\mathbf{GL}$ where

$$\mathbf{L}_{ij} = \begin{cases} d_i \text{ if } i = j, \\ -w_{ij} \text{ if } e_{ij} \in \mathcal{E}, \\ 0 \text{ otherwise,} \end{cases} \tag{1.4}$$

and $\mathbf{G}$ is a diagonal matrix with $\mathbf{G}_{ii} = w(v_i)$. In the context of the Laplacian matrix, $w(v_i) = 1$ or $w(v_i) = \frac{1}{d_i}$ are most often adopted (see [12] for a longer discussion on this point).

If $\mathbf{G} = \mathbf{I}$, then for an undirected graph these matrices are related to each other by the formula

$$\mathbf{A}^T\mathbf{C}\mathbf{A} = \mathbf{W} - \mathbf{D} = \mathbf{L}, \tag{1.5}$$

where $\mathbf{D}$ is a diagonal matrix of node degrees with $\mathbf{D}_{ii} = d_i$.

1.3.2 Adjacency Lists

The advantage of adjacency lists over matrix representations is less memory usage. Indeed, a full incidence matrix requires $O(|\mathcal{V}| \times |\mathcal{E}|)$ memory, and a full adjacency matrix requires $O(|\mathcal{V}|^2)$ memory. However, sparse graphs can take advantage of sparse matrix representations for a much more efficient storage.

An adjacency list representation of a graph is an array L of $|\mathcal{V}|$ linked lists (one for each vertex of $\mathcal{V}$). For each vertex v_i, there is a pointer L_i to a linked list containing all vertices v_j adjacent to v_i. For weighted graphs, the linked list contains both the target vertex and the edge weight. With this representation, iterating through the set of edges is $O(|\mathcal{V}| + |\mathcal{E}|)$ whereas it is $O(|\mathcal{V}|^2)$ for adjacency matrices. However, checking if an edge $e_k \in \mathcal{E}$ is an $O(|\mathcal{V}|)$ operation with adjacency lists whereas it is an $O(1)$ operation with adjacency matrices.

1.4 Paths, Trees, and Connectivity

Graphs model relationships between nodes. These relationships make it possible to define whether two nodes are *connected* by a series of pairwise relationships. This concept of connectivity, and the related ideas of paths and trees, appear in some form throughout this work. Therefore, we now review the concepts related to connectivity and the basic algorithms used to probe connectivity relationships.

1.4.1 Walks and Paths

Given a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and two vertices $v_i, v_j \in \mathcal{V}$, a **walk** (also called a chain) from v_i to v_j is a sequence $\pi(v_1, v_n) = (v_1, e_1, v_2, \ldots, v_{n-1}, e_{n-1}, v_n)$, where $n \geq 1$, $v_i \in \mathcal{V}$, and $e_j \in \mathcal{E}$:

$$v_1 = v_i, v_j = v_n, \text{ and } \{s(e_i), t(e_i)\} = \{v_i, v_{i+1}\}, 1 \leq i \leq n\ . \tag{1.6}$$

The length of the walk is denoted as $|\pi| = n$. A walk may contain a vertex or edge more than once. If $v_i = v_j$, the walk is called a **closed walk**; otherwise it is an **open walk**. A walk is called a **trail** if every edge is traversed only once. A closed trail is called a **circuit**. A circuit is called a **cycle** if all nodes are distinct. An open walk is a **path** if every vertex is distinct. A **subpath** is any sequential subset of a path. Figure

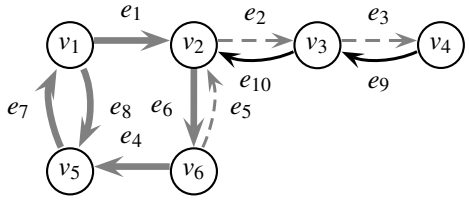


FIGURE 1.5
Illustration of a walk $\pi(v_1, v_1)$ (bolded grey), trail, and path $\pi(v_6, v_4)$ (dashed grey).

1.5 is used to illustrate the concepts of walk, trail, and path. In Figure 1.5, bolded grey arrows provide a walk $\pi(v_1, v_1) = (v_1, e_8, v_5, e_7, v_1, e_1, v_2, e_6, v_6, e_4, v_5, e_7, v_1)$ that is a closed walk, but is neither a trail (e_7 is traversed twice) nor a cycle. Dashed grey arrows provide a walk $\pi(v_6, v_4) = (v_6, e_5, v_2, e_2, v_3, e_3, v_4)$ that is an open walk, a trail, and a path. Black arrows are arrows not involved in the walks $\pi(v_1, v_1)$ and $\pi(v_6, v_4)$.

1.4.2 Connected Graphs

Two nodes are called **connected** if $\exists\pi(v_i, v_j)$ or $\exists\pi(v_j, v_i)$. A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is called a **connected graph** iff $\forall v_i, v_j \in \mathcal{V},\ \exists\pi(v_i, v_j)$, or $\exists\pi(v_j, v_i)$. Therefore, the relation

$$v_i\ R_w\ v_j = \begin{cases} v_i = v_j, \\ \exists\pi(v_i, v_j) \text{ or } \exists\pi(v_j, v_i), \end{cases} \tag{1.7}$$

is an equivalence relation. The induced equivalence classes by this relation form a partition of $\mathcal{V}$ into $\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_p$ subsets. Subgraphs $\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_p$ induced by $\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_p$ are called the *connected components* of the graph $\mathcal{G}$. Each connected component is a connected graph. The connected components of a graph $\mathcal{G}$ are the set of largest subgraphs of $\mathcal{G}$ that are each connected.

Two nodes are **strongly connected** if $\exists\pi(v_i, v_j)$ and $\exists\pi(v_j, v_i)$. If two nodes are

connected but not strongly connected, then the nodes are called **weakly connected**. Note that, in an undirected graph, all connected nodes are strongly connected. The graph is strongly connected iff $\forall v_i, v_j \in \mathcal{V},\ \exists \pi(v_i, v_j)$, and $\exists \pi(v_j, v_i)$. Therefore, the relation

$$v_i \; R_s \; v_j = \begin{cases} v_i = v_j \\ \exists \pi(v_i, v_j) \text{ and } \exists \pi(v_j, v_i), \end{cases} \tag{1.8}$$

is an equivalence relation. The subgraphs induced by the obtained partition of $\mathcal{G}$ are the *strongly connected components* of $\mathcal{G}$. Figure 1.6 illustrates the concepts of strongly and weakly connected components.

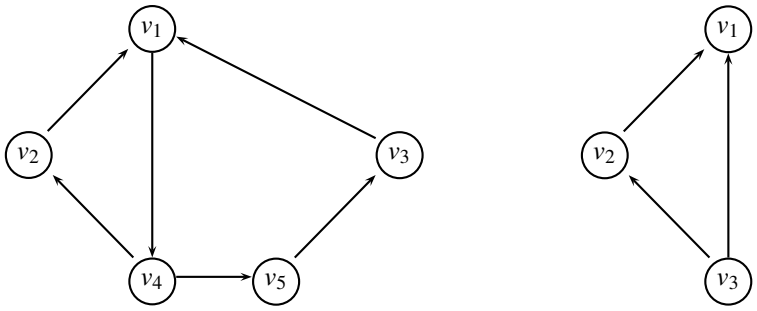


FIGURE 1.6
Left: A strongly connected component. Right: A weakly connected component.

1.4.3 Shortest Paths

The problems of routing in graphs (in particular the search for a shortest path) are among the oldest and most common problems in graph theory. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a weighted graph where a weight function $w : \mathcal{E} \rightarrow \mathbb{R}$ associates a real value (i.e., a weight also called a length in this context) to each edge.[1] In this section we consider only graphs for which all edge weights are nonnegative. Let $l(\pi(v_i, v_j))$ denote the total weight (or length) of a walk:

$$l(\pi(v_i, v_j)) = \sum_{e_k \in \pi(v_i, v_j)} w(e_k) \; . \tag{1.9}$$

[1]Edge weights may be used to represent either *affinities* or *distances* between nodes. An affinity weight of zero is viewed as equivalent to a disconnection (removal of the edge from $\mathcal{E}$), and a distance weight of ∞ is viewed as equivalent to a disconnection. An affinity weight a may be converted to a distance weight b via $b = \frac{1}{a}$. A much longer discussion on this relationship may be found in [12]. In this work, affinity weights and distance weights use the same notation, with the distinction being made by context. All weights in this section are considered to be distance weights.

We will also use the convention that the length of the walk equals ∞ if v_i and v_j are not connected and $l(\pi(v_i, v_i)) = 0$. The minimum length between two vertices is

$$l^*(v_i, v_j) = \arg\min_{\pi(v_i, v_j)} l(\pi(v_i, v_j)). \tag{1.10}$$

A walk satisfying the minimum in (1.10) is a path called a **shortest path**. The shortest path between two nodes may not be unique. Different algorithms can be used to compute l^* if one wants to find a shortest path from one vertex to all the others or between all pairs of vertices. We will restrict ourselves to the first kind of problem.

To compute the shortest path from one vertex to all the others, the most common algorithm is from Dijkstra [13]. Dijkstra's algorithm solves the problem for every pair v_i, v_j, where v_i is a fixed starting point and $v_j \in \mathcal{V}$. Dijkstra's algorithm exploits the property that the path between any two nodes contained in a shortest path is also a shortest path. Specifically, let $\pi(v_i, v_j)$ be a shortest path from v_i to v_j in a weighted connected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with positive weights ($w : \mathcal{E} \to \mathbb{R}^+$) and let v_k be a vertex in $\pi(v_i, v_j)$. Then, the subpath $\pi(v_i, v_k) \subset \pi(v_i, v_j)$ is a shortest path from v_i to v_k. Dijkstra's algorithm for computing shortest paths with v_i as a source is provided in Figure 1.7 with an example on a given graph. In computer vision, shortest paths have been used for example, for interactive image segmentation [14, 15].

$\mathcal{S} = \mathcal{V} \setminus v_i, t(v_i) = 0$

$$t(v_j) = \begin{cases} w_{ij} \text{ if } e_{ij} \in \mathcal{E} \\ +\infty \text{ otherwise} \end{cases}$$

while $\mathcal{S} \neq \emptyset$ **do**
  Select v_j such that $t(v_j) = \min_{v_k \in \mathcal{S}} t(v_k)$
  $\mathcal{S} = \mathcal{S} \setminus v_j$
  for all $v_k \in \mathcal{S}$ and $e_{jk} \in \mathcal{E}$ **do**
    $t(v_k) \leftarrow \min(t(v_k), t(v_j) + l(e_{jk}))$
  end for
end while
$\forall v_j, t(v_j)$ provides the value of $l^*(v_i, v_j)$

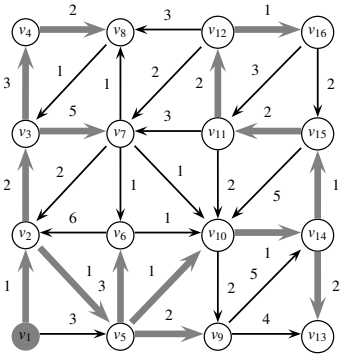


FIGURE 1.7
Left: Dijkstra's algorithm for computing shortest path from a source vertex v_i to all vertices in a graph. Right: Illustration of Dijkstra's algorithm. The source is vertex v_1. The shortest path from v_1 to other vertices is shown with bolded grey arrows.

1.4.4 Trees and Minimum Spanning Trees

A **tree** is an undirected connected graph without cycles (acyclic). An unconnected tree without cycles is called a forest (each connected component of a forest is a tree). More precisely, if $\mathcal{G}$ is a graph with $|\mathcal{V}| = n$ vertices and if $\mathcal{G}$ is a tree, the following properties are equivalent:

- $\mathcal{G}$ is connected without cycles,
- $\mathcal{G}$ is connected and has $n-1$ edges,
- $\mathcal{G}$ has no cycles and has $n-1$ edges,
- $\mathcal{G}$ has no cycles and is maximal for this property (i.e., adding any edge creates a cycle),
- $\mathcal{G}$ is connected and is minimal for this property (i.e., removing any edge makes the graph disconnected),
- There exists a unique path between any two vertices of $\mathcal{G}$.

A partial graph $\mathcal{G}' = (\mathcal{V}, \mathcal{E}')$ of a connected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is called a **spanning tree** of $\mathcal{G}$ if $\mathcal{G}'$ is a tree. There is at least one spanning tree for any connected graph.

We may define the weight (or cost) of any tree $\mathcal{T} = (\mathcal{V}, \mathcal{E}_\mathcal{T})$ by

$$c(\mathcal{T}) = \sum_{e_k \in \mathcal{E}_\mathcal{T}} w(e_k) \, . \tag{1.11}$$

A spanning tree of the graph that minimizes the cost (compared to all spanning trees of the graph) is called a **minimal spanning tree**. The minimal spanning tree of a graph may not be unique. The problem of finding a minimal spanning tree of a graph appears in many applications (e.g., in phone network design). There are several algorithms for finding minimum spanning trees, and we present the one proposed by Prim. The principle of Prim's algorithm is to progressively build a tree. The algorithm starts from the edge of minimum weight and adds iteratively to the tree the edge of minimum weight among all the possible edges that maintain a partial graph that is acyclic. More precisely, for a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, one progressively constructs from a vertex v_1 a subset $\mathcal{V}'$ with $\{v_1\} \subseteq \mathcal{V}' \subseteq \mathcal{V}$ and a subset $\mathcal{E}' \subseteq \mathcal{E}$ such that the partial graph $\mathcal{G}' = (\mathcal{V}, \mathcal{E}')$ is a minimum spanning tree of $\mathcal{G}$. To do this, at each step, one selects the edge e_{ij} of minimum weight in the set $\{e_{kl} \mid e_{kl} \in \mathcal{E},\ v_k \in \mathcal{V}',\ l \in \mathcal{V} \setminus \mathcal{V}'\}$. Subset $\mathcal{V}'$ and $\mathcal{E}'$ are then augmented with vertex v_j and edge e_{ij}: $\mathcal{V}' \leftarrow \mathcal{V}' \cup \{v_j\}$ and $\mathcal{E}' \leftarrow \mathcal{E}' \cup \{e_{ij}\}$. The algorithm terminates when $\mathcal{V}' = \mathcal{V}$. Prim's algorithm for computing minimum spanning trees is provided in Figure 1.8 along with an example on a given graph. In computer vision, minimum spanning trees have been used, for example, for image segmentation [16] and image hierarchical representation [17].

1.4.5 Maximum Flows and Minimum Cuts

A transport network is a weighted directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ such that

- There exists exactly one vertex v_1 with no predecessor called the **source** (and denoted by v_s), that is, $\deg^+(v_1) = 0$,

$\mathcal{V}' = \{v_1\}, \mathcal{E}' = \emptyset$
for all $v_j \in \mathcal{V} \backslash \mathcal{V}'$ **do**
  $L(v_j) = l(e_{1j})$
end for
while $\mathcal{V}' \neq \mathcal{V}$ **do**
  Choose $v_i \in \mathcal{V} \backslash \mathcal{V}'$ such that $L(v_i)$ is minimum
  Let e_k be the associated edge
  $\mathcal{V}' \leftarrow \mathcal{V}' \cup \{v_i\}$
  $\mathcal{E}' \leftarrow \mathcal{E}' \cup \{e_k\}$
  for all $v_j \in \mathcal{V} \backslash \mathcal{V}'$ **do**
    if $l(e_{ij}) < L(v_j)$ **then**
      $L(v_j) = l(e_{ij})$
    end if
  end for
end while

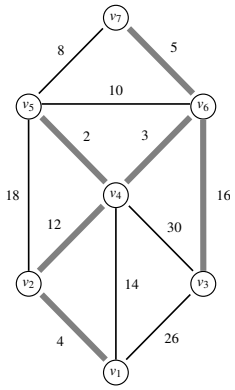


FIGURE 1.8
Left: Prim's algorithm for computing minimum spanning trees in a graph starting from node v_1. Note that, any node may be selected as v_1, and the algorithm will generate a minimal spanning tree. However, when a graph contains multiple minimal spanning trees the selection of v_1 may determine which minimal spanning tree is determined by the algorithm. Right: Illustration of Prim's algorithm. The minimum spanning tree is shown by the grey bolded edges.

- There exists exactly one vertex v_n with no successor called the **sink** (and denoted by v_t), that is, $\deg^-(v_n) = 0$,
- There exists at least one path from v_s to v_t,
- The weight $w(e_{ij})$ of the directed edge e_{ij} is called the **capacity** and it is a nonnegative real number, that is, we have a mapping $w : \mathcal{E} \rightarrow \mathbb{R}^+$.

We denote by $\mathcal{A}^+(v_i)$ the set of inward directed edges from vertex v_i:

$$\mathcal{A}^+(v_i) = \{e_{ji} \in \mathcal{E}\} . \tag{1.12}$$

Similarly, we denote by $\mathcal{A}^-(v_i)$ the set of outward directed edges from vertex v_i:

$$\mathcal{A}^-(v_i) = \{e_{ij} \in \mathcal{E}\} . \tag{1.13}$$

A function $\varphi : \mathcal{E} \rightarrow \mathbb{R}^+$ is called a **flow** iff

- Each vertex $v_i \notin \{v_s, v_t\}$ satisfies the conservation condition (also called Kirchhoff's Current Law):

$$\sum_{e_{ji}} \varphi(e_{ji}) - \sum_{e_{ij}} \varphi(e_{ij}) = 0, \tag{1.14}$$

  which may also be written in matrix form as

$$\mathbf{A}^T \boldsymbol{\varphi} = \mathbf{p}, \tag{1.15}$$

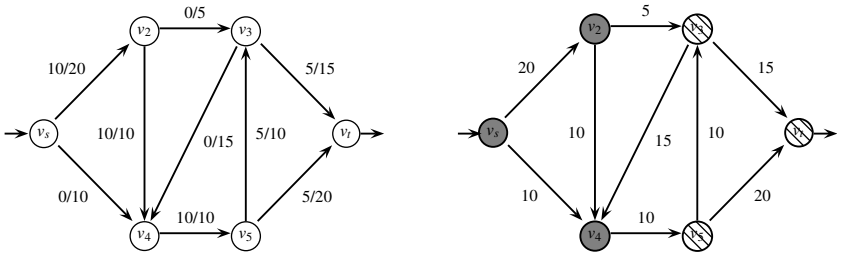


FIGURE 1.9
Left: an (s,t)-flow with value 10. Each edge is labeled with its flow/capacity. Right: An (s,t)-cut with capacity 30, $\mathcal{V}_1 = \{v_s, v_2, v_4\}$ (grey) and $\mathcal{V}_2 = \{v_3, v_5, v_t\}$ (striped).

where $p_i = 0$ for any $v_i \notin \{v_s, v_t\}$. Intuitively, this law states that the flow entering each node must also leave that node (i.e., conservation of flow).

- For each edge e_{ij}, the capacity constraint $\varphi(e_{ij}) \leq w(e_{ij})$ is satisfied.

The value of a flow is $|\varphi| = \sum_{e_{ij} \in \mathcal{A}^-(v_s)} \varphi(e_{ij}) = \sum_{e_{ij} \in \mathcal{A}^+(v_t)} \varphi(e_{ij}) = p_s$. A flow φ^* is a **maximum flow** if its value is the largest possible, that is, $|\varphi^*| \geq |\varphi|$ for every other flow φ. Figure 1.9 shows an (s,t) flow with value 10.

An (s,t)-cut is a partition $P =< \mathcal{V}_1, \mathcal{V}_2 >$ of the vertices into subsets $\mathcal{V}_1$ and $\mathcal{V}_2$, such that, $\mathcal{V}_1 \cap \mathcal{V}_2 = \emptyset$, $\mathcal{V}_1 \cup \mathcal{V}_2 = \mathcal{V}$, $v_s \in \mathcal{V}_1$ and $v_t \in \mathcal{V}_2$. The capacity of a cut P is the sum of the capacities of the edges that start in $\mathcal{V}_1$ and end in $\mathcal{V}_2$:

$$C(P) = \sum_{v_i \in \mathcal{V}_1} \sum_{v_j \in \mathcal{V}_2} w(e_{ij}) \,, \tag{1.16}$$

with $w(e_{ij}) = 0$ if $e_{ij} \notin \mathcal{E}$. Figure 1.9 shows an (s,t)-cut with capacity 30. The **minimum cut** problem is to compute an (s,t)-cut for which the capacity is as low as possible. Intuitively, the minimum cut is the least expensive way to disrupt all flow from v_s to v_t. In fact, one can prove that, for any weighted directed graph, there is always a flow φ and a cut $(\mathcal{V}_1, \mathcal{V}_2)$ such that the value of the maximum flow is equal to the capacity of the minimum cut [18].

We call an edge e_{ij} **saturated** by a flow if $\varphi(e_{ij}) = w_{ij}$. A path from v_s to v_t is saturated if any one of its edges is saturated. The **residual capacity** of an edge $c_r : \mathcal{V} \times \mathcal{V} \to \mathbb{R}^+$ is

$$c_r(e_{ij}) = \begin{cases} w_{ij} - \varphi(e_{ij}) \text{ if } e_{ij} \in \mathcal{E}, \\ \varphi(e_{ij}) \text{ if } e_{ji} \in \mathcal{E}, \\ 0 \text{ otherwise.} \end{cases} \tag{1.17}$$

The residual capacity represents the flow quantity that can still go through the edge. We may define the **residual graph** as a partial graph $\mathcal{G}_r = (\mathcal{V}, \mathcal{E}_r)$ of the initial

$\varphi(e_{ij}) \leftarrow 0, \forall e_{ij} \in \mathcal{E}$
Construct the residual graph $\mathcal{G}_r = (\mathcal{V}, \mathcal{E}_r)$
while $\exists\, \pi(v_s, v_t)$ in $\mathcal{G}_r$ such that $c_r(\pi) > 0$ **do**
  for all $e_{ij} \in \pi$ **do**
    $\varphi(e_{ij}) \leftarrow \varphi(e_{ij}) + c_r(\pi)$
    $\varphi(e_{ji}) \leftarrow \varphi(e_{ji}) - c_r(\pi)$
  end for
end while

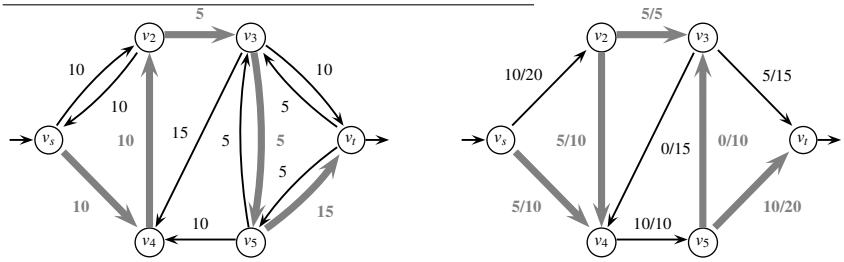


FIGURE 1.10
Top: Ford–Fulkerson algorithm for maximum flow. Bottom: An augmenting path π in $\mathcal{G}_r$ with $c_r(\pi) = 5$, and the associated augmented flow φ'.

graph where all the edges of zero residual capacity have been removed, that is, $\mathcal{E}_r = \{e_{ij} \in \mathcal{E} \mid c_r(e_{ij}) > 0\}$.

Given a path π in $\mathcal{G}_r$ from v_s to v_t, the residual capacity of an augmenting path is the minimum of all the residual capacities of its edges:

$$c_r(\pi) = \arg\min_{e_{ij} \in \pi} c_r(e_j). \tag{1.18}$$

Such a path is called an **augmenting path** if $c_r(\pi) > 0$. From this augmenting path we can define a new augmented flow function φ':

$$\varphi'(e_{ij}) = \begin{cases} \varphi(e_{ij}) + c_r(\pi) \text{ if } e_{ij} \in \pi, \\ \varphi(e_{ij}) - c_r(\pi) \text{ if } e_{ji} \in \pi, \\ \varphi(e_{ij}) \text{ otherwise.} \end{cases} \tag{1.19}$$

A flow is called a **maximum flow** if and only if the corresponding residual graph is disconnected (i.e., there is no augmenting path). There are several algorithms for finding maximum flows, but we present the classic algorithm proposed by Ford and Fulkerson [18]. The principle of the Ford–Fulkerson algorithm is to add flow along one path as long as there exists an augmenting path in the graph. Figure 1.10 provides the algorithm along with one step of generating an augmenting path. In computer vision, the use of maximum-flow/min-cut solutions to solve problems is typically referred to as *graph cuts*, following the publication by Boykov et al. [19]. Furthermore,

the Ford–Fulkerson algorithm was found to be inefficient for many computer vision problems, with the algorithm by Boykov and Kolmogorov generally preferred [20].

1.5 Graph Models in Image Processing and Analysis

In the previous sections we have presented some of the basic definitions and notations used for graph theory in this book. These definitions and notations may be found in other sources in the literature but were presented to make this work self-contained. However, there are many ways in which graph theory is applied in image processing and computer vision that are specialized to these domains. We now build on the previous definitions to provide some of the basics for how graph theory has been specialized for image processing and computer vision in order to provide a foundation for the subsequent chapters.

1.5.1 The Regular Lattice

Digital image processing operates on sampled data from an underlying continuous light field. Each sample is called a **pixel** in 2D or a **voxel** in 3D. The sampling is based on the notion of *lattice*, which can be viewed as a regular tiling of a space by a primitive cell. A d-dimensional lattice $\mathbb{L}^d$ is a subset of the d-dimensional Euclidean space $\mathbb{R}^d$ with

$$\mathbb{L}^d = \{\mathbf{x} \in \mathbb{R}^d \mid \mathbf{x} = \sum_{i=1}^{d} k_i \mathbf{u_i},\ k_i \in \mathbb{Z}\} = \mathbf{U}\mathbb{Z}^d\,, \tag{1.20}$$

where $\mathbf{u_1}, \ldots, \mathbf{u_d} \in \mathbb{R}^d$ form a basis of $\mathbb{R}^d$ and $\mathbf{U}$ is the matrix of column vectors. A lattice in a d-dimensional space is therefore a set of all possible vectors represented as integer weighted combinations of d linearly independent basis vectors. The lattice points are given by $\mathbf{Uk}$ with $\mathbf{k}^T = (k_1, \ldots, k_d) \in \mathbb{Z}^d$.

The unit cell of $\mathbb{L}^d$ with respect to the basis $\mathbf{u_1}, \ldots, \mathbf{u_d}$ is

$$C = \{\mathbf{x} \in \mathbb{R}^d \mid \mathbf{x} = \sum_{i=1}^{d} k_i \mathbf{u_i},\ k_i \in [0,1]\} = \mathbf{U}\cdot[0, \mathbf{u_1}]^d\,, \tag{1.21}$$

where $[0, \mathbf{u_1}]^d$ is the unit cube in $\mathbb{R}^d$. The volume of C is given by $|\det \mathbf{U}|$. The density of the lattice is given by $1/|\det \mathbf{U}|$, that is, the lattice sites per unit surface. The set $\{C + \mathbf{x} \mid \mathbf{x} \in \mathbb{R}^d\}$ of all lattice cells covers $\mathbb{R}^d$. The Voronoi cell of a lattice is called the unit cell (a cell of a lattice whose translations cover the whole space). The Voronoi cell encloses all points that are closer to the origin than to other lattice points.

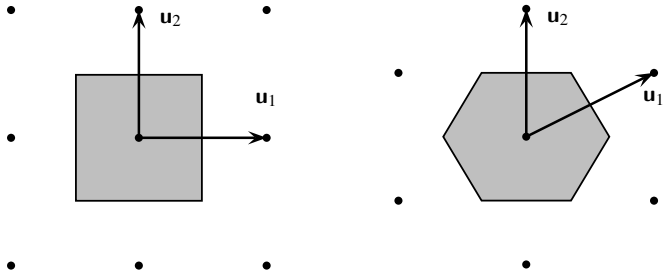


FIGURE 1.11
The rectangular (left) and hexagonal (right) lattices and their associated Voronoi cells.

The Voronoi cell boundaries are equidistant hyperplanes between surrounding lattice points. Two well-known 2D lattices are the rectangular $\mathbf{U_1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ and hexagonal $\mathbf{U_2} = \begin{bmatrix} \frac{\sqrt{3}}{2} & 0 \\ \frac{1}{2} & 1 \end{bmatrix}$ lattices. Figure 1.11 presents these two lattices. By far the most common lattice in image processing and computer vision is the rectangular sampling lattice derived from $\mathbf{U_1}$. In 3D image processing, the lattice sampling is typically given by

$$\mathbf{U_3} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & k \end{bmatrix}, \tag{1.22}$$

for some constant k in which the typical situation is that $k \geq 1$.

The most common way of using a graph to model image data is to identify every node with a pixel (or voxel) and to impose an edge structure to define local neighborhoods for each node. Since a lattice defines a regular arrangement of pixels it is possible to define a regular graph by using edges to connect pairs of nodes that fall within a fixed Euclidean distance of each other.

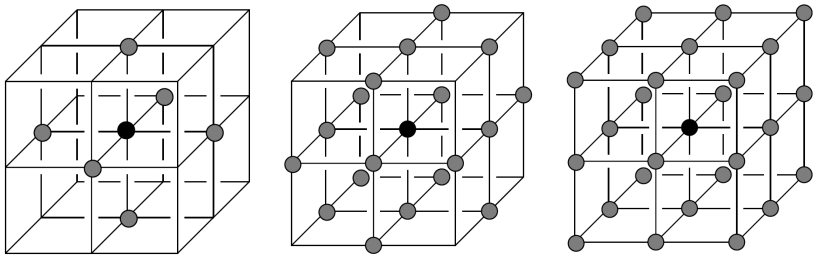
For example, in a rectangular lattice, two pixels $\mathbf{p} = (p_1, p_2)^T$ and $\mathbf{q} = (q_1, q_2)^T$ of $\mathbb{Z}^2$, are called 4-adjacent (or 4-connected) if they differ by at most one coordinate:

$$|p_1 - q_1| + |p_2 - q_2| = 1\,, \tag{1.23}$$

and 8-adjacent (or 8-connected) if they differ at most of 2 coordinates:

$$\max(|p_1 - q_1|, |p_2 - q_2|) = 1\,, \tag{1.24}$$

In the hexagonal lattice, only one adjacency relationship exists: 6-adjacency. Figure 1.11 presents adjacency relationships on the 2D rectangular and hexagonal lattices. Similarly, in $\mathbb{Z}^3$, we can say in a rectangular lattice that voxels are 6-adjacent,



(a) 6-adjacency. (b) 18-adjacency. (c) 26-adjacency.

FIGURE 1.12
Different adjacency structures in a 3D lattice.

18-adjacent, or 26-adjacent whether they differ at most of 1, 2, or 3 coordinates. Figure 1.12 presents adjacency relationships between 3D cells (voxels).

If one considers edges that connect pairs of nodes that are not directly (i.e., locally) spatially adjacent (e.g., 4- or 8-connected in $\mathbb{Z}^2$) [21], the corresponding edges are called nonlocal edges [22]. The level of nonlocality of an edge depends on how far the two points are in the underlying domain. These nonlocal edges are usually obtained from proximity graphs (see Section 1.5.3) on the pixel coordinates (e.g., ϵ-ball graphs) or on pixel features (e.g., k-nearest-neighbor graphs on patches).

1.5.2 Irregular tessellations

Image data is almost always sampled in a regular, rectangular lattice that is modeled with the graphs in the previous section. However, images are often simplified prior to processing by merging together regions of adjacent pixels using an untargeted image segmentation algorithm. The purpose of this simplification is typically to reduce processing time for further operations, but it can also be used to compress the image representation or to improve the effectiveness of subsequent operations. When this image simplification has been performed, a graph may be associated with the simplified image by identifying each merged region with one node and defining an edge relationship for two adjacency regions. Unfortunately, this graph is no longer guaranteed to be regular due to the image-dependent and space-varying merging of the underlying pixels. In this section we review common simplification methods that produce these irregular graphs.

One very well-known irregular image tessellation is the region quadtree tessellation [25]. A **region quadtree** is a hierarchical image representation that is derived from recursively subdividing the 2D space into four quadrants of equal size until every square contains one homogeneous region (based on appearance properties of every region of the underlying rectangular grid) or contains a maximum of one pixel.

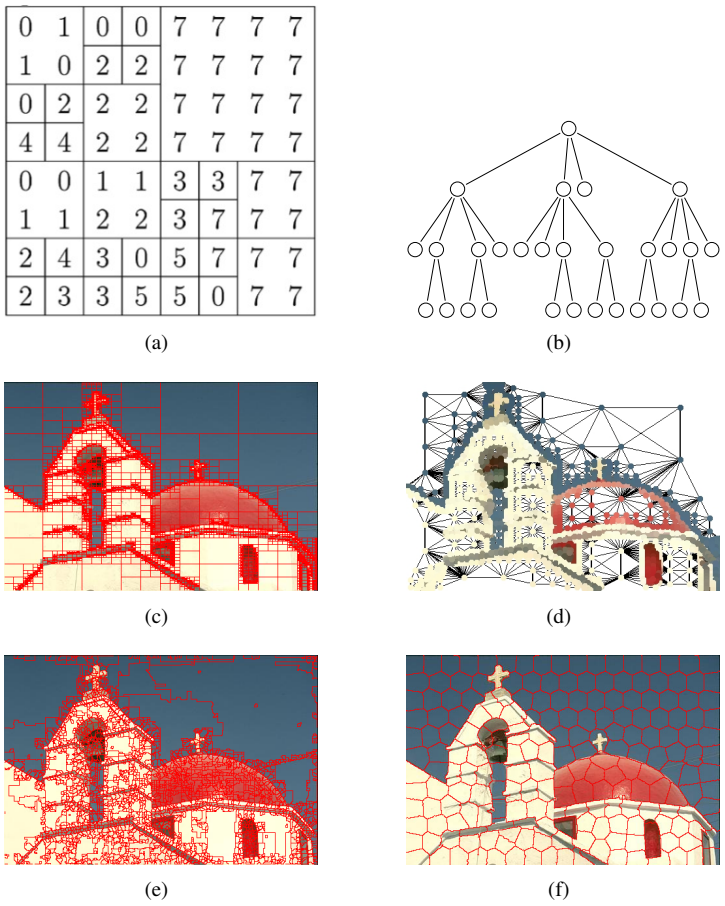


FIGURE 1.13
(a) An image with the quadtree tessellation, (b) the associated partition tree, (c) a real image with the quadtree tessellation, (d) the region adjacency graph associated to the quadtree partition, (e) and (f) two different irregular tessellations of an image using image-dependent superpixel segmentation methods: Watershed [23] and SLIC superpixels [24].

A region quadtree tesselation can be easily represented by a tree where each node of the tree corresponds to a square. The set of all nodes at a given depth provides a given level of decomposition of the 2D space. Figure 1.13 presents a simple example on an image (Figure 1.13(a)) with the associated tree (Figure 1.13(b)), as well as an example on a real image (Figure 1.13(c)). Quadtrees easily generalize to higher dimensions. For instance, in three dimensions, the 3D space is recursively subdivided into eight octants, and thus the tree is called an **octree**, where each node has eight children.

Finally, any partition of the classical rectangular grid can be viewed as producing an irregular tessellation of an image. Therefore any segmentation of an image can be associated with a **region adjacency graph**. Figure 1.13(d) presents the region adjacency graph associated to the partition depicted in 1.13(c). Specifically, given a graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$ where each node is identified with a pixel, a partition into R connected regions $\mathcal{V}_1 \cup \mathcal{V}_2 \cup \ldots \cup \mathcal{V}_R = \mathcal{V}$, $\mathcal{V}_1 \cap \mathcal{V}_2 \cap \ldots \cap \mathcal{V}_R = \emptyset$ may be identified with a new graph $\tilde{\mathcal{G}} = \{\tilde{\mathcal{V}}, \tilde{\mathcal{E}}\}$ where each partition of $\mathcal{V}$ is identified with a node of $\tilde{\mathcal{V}}$, that is, $\mathcal{V}_i \in \tilde{\mathcal{V}}$. A common method for defining $\tilde{\mathcal{E}}$ is to let the edge weight between each new node equal the sum of the edge weights connecting each original node in the sets, that is, for edge $e_{ij} \in \tilde{\mathcal{E}}$

$$w_{ij} = \sum_{e_{ks}, v_k \in \mathcal{V}_i, v_s \in \mathcal{V}_j} w_{ks}. \tag{1.25}$$

Each contiguous $\mathcal{V}_i$ is called a **superpixel**. Superpixels are an increasingly popular trend in computer vision and image processing. This popularity is partly due to the success of graph-based algorithms, which can be used to operate efficiently on these irregular structures (as opposed to traditional image-processing algorithms which required that each element is organized on a grid). Indeed, graph-based models (e.g., Markov Random Fields or Conditional Random Fields) can provide dramatic speed increases when moving from pixel-based graphs to superpixel-based graphs due to the drastic reduction in the number of nodes. For many vision tasks, compact and highly uniform superpixels that respect image boundaries are desirable. Typical algorithms for generating superpixels are Normalized Cuts [21], Watersheds [23], Turbo Pixels [26], and simple linear iterative clustering superpixels [24], etc. Figure 1.13(e) and (f) presents some irregular tessellations based on superpixel algorithms.

1.5.3 Proximity Graphs for Unorganized Embedded Data

Sometimes the relevant features of an image are neither individual pixels nor a partition of the image. For example, the relevant features of an image might be the blood cells in a biomedical image, and it is these blood cells that we wish to identify with the nodes of our graph. We may assume that each node (feature) in a dD image is associated with a coordinate in d dimensions. However, although each node has a geometric representation, it is much less clear than before how to construct a meaningful edge set for a graph based on the proximity of features.

There are many ways to construct a proximity graph representation from a set of data points that are embedded in $\mathbb{R}^d$. Let us consider a set of data points $\{\mathbf{x_1}, \ldots, \mathbf{x_n}\} \in \mathbb{R}^d$. To each data point we associate a vertex of a proximity graph $\mathcal{G}$ to define a set of vertices $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$. Determining the edge set $\mathcal{E}$ of the proximity graph $\mathcal{G}$ requires defining the neighbors of each vertex v_i according to its embedding $\mathbf{x}_i$. A proximity graph is therefore a graph in which two vertices are connected by an edge iff the data points associated to the vertices satisfy particular geometric requirements. Such particular geometric requirements are usually based on a metric measuring the distance between two data points. A usual choice of metric is the Euclidean metric. We will denote as $\mathcal{D}(v_i, v_j) = \|\mathbf{x}_i - \mathbf{x}_j\|_2$ the Euclidean distance between vertices, and as $\mathcal{B}(v_i\ ;\ r) = \{\mathbf{x}_j \in \mathbb{R}^n \mid \mathcal{D}(v_i, v_j) \leq r\}$ the closed ball of radius r centered on $\mathbf{x}_i$. Classical proximity graphs are:

- The ϵ-ball graph: $v_i \sim v_j$ if $\mathbf{x}_j \in \mathcal{B}(v_i\ ;\ \epsilon)$.
- The k-nearest-neighbor graph (k-NNG): $v_i \sim v_j$ if the distance between $\mathbf{x}_i$ and $\mathbf{x}_i$ is among the k-th smallest distances from $\mathbf{x}_i$ to other data points. The k-NNG is a directed graph since one can have $\mathbf{x}_i$ among the k-nearest neighbors of $\mathbf{x}_j$ but *not* vice versa.
- The Euclidean Minimum Spanning Tree (EMST): This is a connected tree subgraph that contains all the vertices and has a minimum sum of edge weights (see Section 1.4.4). The weight of the edge between two vertices is the Euclidean distance between the corresponding data points.
- The symmetric k-nearest-neighbor graph (Sk-NNG): $v_i \sim v_j$ if $\mathbf{x}_i$ is among the k-nearest neighbors of y *or* vice versa.
- The mutual k-nearest-neighbor graph (Mk-NNG): $v_i \sim v_j$ if $\mathbf{x}_i$ is among the k-nearest neighbors of y *and* vice versa. All vertices in a mutual k-NN graph have a degree upper-bounded by k, which is not usually the case with standard k-NN graphs.
- The Relative Neighborhood Graph (RNG): $v_i \sim v_j$ iff there is no vertex in

$$\mathcal{B}\left(v_i\ ;\ \mathcal{D}(v_i, v_j)\right) \cap \mathcal{B}\left(v_j\ ;\ \mathcal{D}(v_i, v_j)\right)\ . \tag{1.26}$$

- The Gabriel Graph (GG): $v_i \sim v_j$ iff there is no vertex in

$$\mathcal{B}\left(\frac{v_i + v_j}{2}\ ;\ \frac{\mathcal{D}(v_i, v_j)}{2}\right)\ . \tag{1.27}$$

- The β-Skeleton Graph (β-SG): $v_i \sim v_j$ iff there is no vertex in

$$\mathcal{B}\left(\left(1 - \frac{\beta}{2}\right) v_i + \frac{\beta}{2} v_j\ ;\ \frac{\beta}{2}\mathcal{D}(v_i, v_j)\right) \cap \mathcal{B}\left(\left(1 - \frac{\beta}{2}\right) v_j + \frac{\beta}{2} v_i\ ;\ \frac{\beta}{2}\mathcal{D}(v_i, v_j)\right)\ . \tag{1.28}$$

  The Gabriel Graph is obtained with $\beta = 1$, and the Relative Neighborhood Graph with $\beta = 2$.

- The Delaunay Triangulation (DT): $v_i \sim v_j$ iff there is a closed ball $\mathcal{B}(\cdot\,;\,r)$ with v_i and v_j on its boundary and no other vertex v_k contained in it. The Delaunay Triangulation is the dual of the Voronoi irregular tessellation where each Voronoi cell is defined by the set $\{\mathbf{x} \in \mathbb{R}^n \mid \mathcal{D}(\mathbf{x}, v_k) \leq \mathcal{D}(\mathbf{x}, v_j) \text{ for all } v_j \neq v_k\}$. In such a graph, $\forall\, v_i,\ \deg(v_i) = 3$.
- The Complete Graph (CG): $v_i \sim v_j\ \forall (v_i, v_j)$, that is, it is a fully connected graph that contains an edge for each pair of vertices and $\mathcal{E} = \mathcal{V} \times \mathcal{V}$.

A surprising relationship between the edge sets of these graphs is that [27]

$$1 - \text{NNG} \subseteq \text{EMST} \subseteq \text{RNG} \subseteq \text{GG} \subseteq \text{DT} \subseteq \text{CG}. \tag{1.29}$$

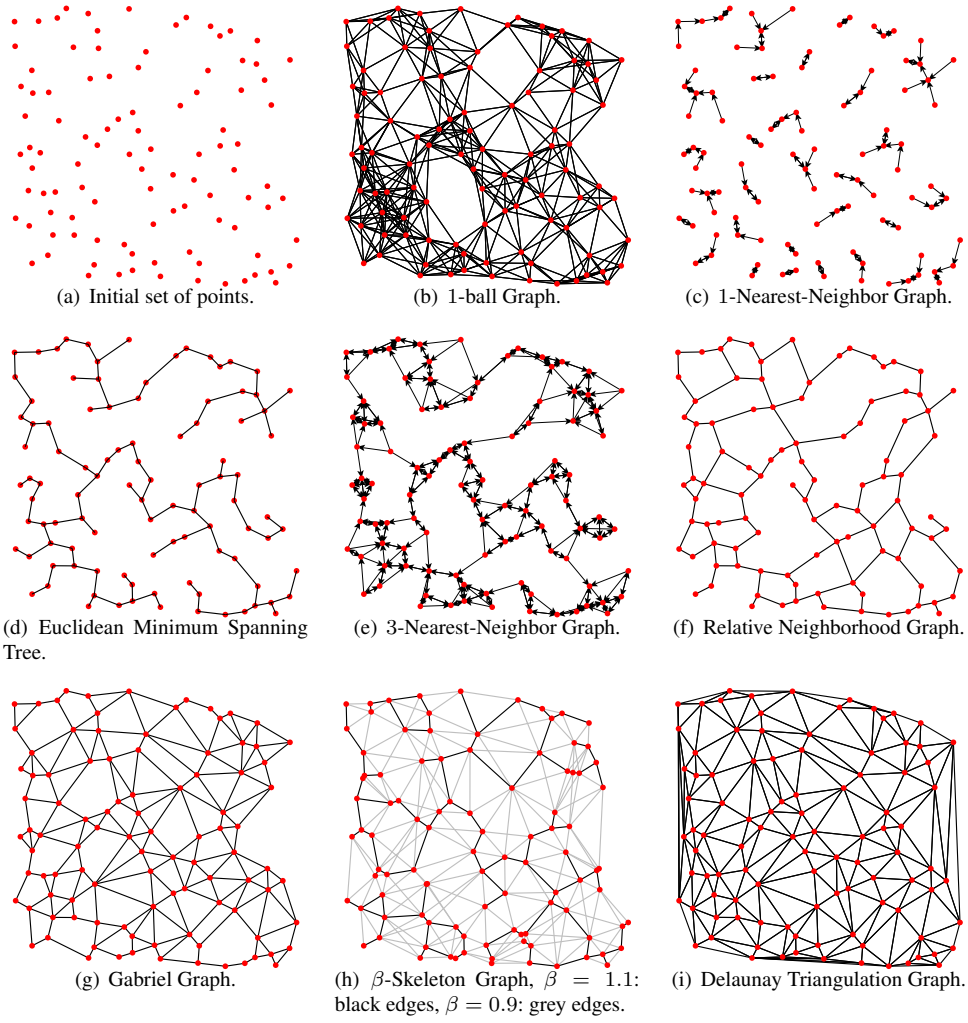
Figure 1.14 presents examples of some of these proximity graphs. All these graphs are extensively used in different image analysis tasks. These methods have been used in Scientific Computing to obtain triangular irregular grids (triangular meshes) adapted for the finite element method [28] for applications in physical modeling used in industries such as automotive, aeronautical, etc. Proximity graphs have also been used in Computational and Discrete Geometry [29] for the analysis of point sets in $\mathbb{R}^2$ (e.g., for 2D shapes) or $\mathbb{R}^3$ (e.g., for 3D meshes). They are also the basis of many classification and model reduction methods [30] that operate on feature vectors in $\mathbb{R}^d$, for example, Spectral Clustering [31], Manifold Learning [32], object matching, etc. Figure 1.15 presents some proximity graph examples on real-world data.

1.6 Conclusion

Graphs are tools that make it possible to model pairwise relationships between data elements and have found wide application in science and engineering. Computer vision and image processing have a long history of using graph models, but these models have been increasingly dominant representations in the recent literature. Graphs may be used to model spatial relationships between nearby or distant pixels, between image regions, between features, or as models of objects and parts. The subsequent chapters of this book are written by leading researchers who will provide the reader with a detailed understanding of how graph theory is being successfully applied to solve a wide range of problems in computer vision and image processing.

Bibliography

[1] N. Biggs, E. Lloyd, and R. Wilson, *Graph Theory, 1736–1936*. Oxford University Press, New-York, 1986.

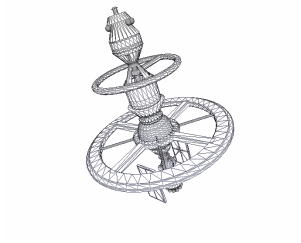


(a) Initial set of points. (b) 1-ball Graph. (c) 1-Nearest-Neighbor Graph.

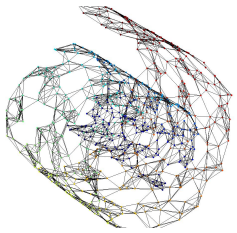
(d) Euclidean Minimum Spanning Tree. (e) 3-Nearest-Neighbor Graph. (f) Relative Neighborhood Graph.

(g) Gabriel Graph. (h) β-Skeleton Graph, $\beta = 1.1$: black edges, $\beta = 0.9$: grey edges. (i) Delaunay Triangulation Graph.

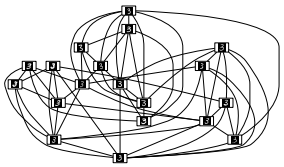
FIGURE 1.14
Examples of proximity graphs from a set of 100 points in $\mathbb{Z}^2$.



(a) 3D Mesh of a satellite obtained by Delaunay Triangulation.



(b) Nearest Neighbor graph of points in $\mathbb{R}^3$.



(c) Nearest Neighbor graph of an image database (with each image treated as a vector of $\mathbb{R}^{16}$).

FIGURE 1.15
Some proximity graph examples on real-world data.

[2] A.-L. Barabasi, *Linked: How Everything Is Connected to Everything Else and What It Means*. Plume, New-York, 2003.

[3] D. J. Watts, *Six Degrees: The Science of a Connected Age*. W. W. Norton, 2004.

[4] N. A. Christakis and J. H. Fowler, *Connected: The Surprising Power of Our Social Networks and How They Shape Our Lives*. Little, Brown and Company, 2009.

[5] F. Harary, *Graph Theory*. Addison-Wesley, 1994.

[6] A. Gibbons, *Algorithmic Graph Theory*. Cambridge University Press, 1989.

[7] N. Biggs, *Algebraic Graph Theory*, 2nd ed. Cambridge University Press, 1994.

[8] R. Diestel, *Graph Theory*, 4th ed., ser. Graduate Texts in Mathematics. Springer-Verlag, 2010, vol. 173.

[9] J. L. Gross and J. Yellen, *Handbook of Graph Theory*, ser. Discrete Mathematics and Its Applications. CRC Press, 2003, vol. 25.

[10] J. Bondy and U. Murty, *Graph Theory*, 3rd ed., ser. Graduate Texts in Mathematics. Springer-Verlag, 2008, vol. 244.

[11] J. Gallier, *Discrete Mathematics*, 1st ed., ser. Universitext. Springer-Verlag, 2011.

[12] L. Grady and J. R. Polimeni, *Discrete Calculus: Applied Analysis on Graphs for Computational Science*. Springer, 2010.

[13] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[14] E. N. Mortensen and W. A. Barrett, "Intelligent scissors for image composition," in *SIGGRAPH*, 1995, pp. 191–198.

[15] P. Felzenszwalb and R. Zabih, "Dynamic programming and graph algorithms in computer vision," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 33, no. 4, pp. 721–740, 2011.

[16] P. F. Felzenszwalb and D. P. Huttenlocher, "Efficient graph-based image segmentation," *Int. J. Computer Vision*, vol. 59, no. 2, pp. 167–181, 2004.

[17] L. Najman, "On the equivalence between hierarchical segmentations and ultrametric watersheds," *J. of Math. Imaging Vision*, vol. 40, no. 3, pp. 231–247, 2011.

[18] L. Ford Jr and D. Fulkerson, "A suggested computation for maximal multi-commodity network flows," *Manage. Sci.*, pp. 97–101, 1958.

[19] Y. Boykov, O. Veksler, and R. Zabih, "Fast approximate energy minimization via graph cuts," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 23, no. 11, pp. 1222–1239, 2001.

[20] Y. Boykov and V. Kolmogorov, "An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 26, no. 9, pp. 1124–1137, 2004.

[21] J. Shi and J. Malik, "Normalized cuts and image segmentation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 22, no. 8, pp. 888–905, 2000.

[22] A. Elmoataz, O. Lezoray, and S. Bougleux, "Nonlocal discrete regularization on weighted graphs: A framework for image and manifold processing," *IEEE Trans. Image Process.*, vol. 17, no. 7, pp. 1047–1060, 2008.

[23] L. Vincent and P. Soille, "Watersheds in digital spaces: An efficient algorithm based on immersion simulations," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 13, no. 6, pp. 583–598, 1991.

[24] A. Radhakrishna, "Finding Objects of Interest in Images using Saliency and Superpixels," Ph.D. dissertation, 2011.

[25] H. Samet, "The quadtree and related hierarchical data structures," *ACM Comput. Surv.*, vol. 16, no. 2, pp. 187–260, 1984.

[26] A. Levinshtein, A. Stere, K. N. Kutulakos, D. J. Fleet, S. J. Dickinson, and K. Siddiqi, "Turbopixels: Fast superpixels using geometric flows," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 31, no. 12, pp. 2290–2297, 2009.

[27] G. T. Toussaint, "The relative neighbourhood graph of a finite planar set," *Pattern Recogn.*, vol. 12, no. 4, pp. 261–268, 1980.

[28] O. Zienkiewicz, R. Taylor, and J. Zhu, *The Finite Element Method: Its Basis and Fundamentals*, 6th ed. Elsevier, 2005.

[29] J. E. Goodman and J. O'Rourke, *Handbook of Discrete and Computational Geometry*, 2nd ed. CRC Press LLC, 2004.

[30] M. A. Carreira-Perpinan and R. S. Zemel, "Proximity graphs for clustering and manifold learning," in *NIPS*, 2004.

[31] U. von Luxburg, "A tutorial on spectral clustering," *Stat. Comp.*, vol. 17, no. 4, pp. 395–416, 2007.

[32] J. A. Lee and M. Verleysen, *Nonlinear Dimensionality Reduction*. Springer, 2007.