1
Paths in Graphs

1.1 Introduction to Graph Theory

A *graph* $G(V,E)$ is a structure consisting of a set of *vertices* $V=\{v_1,v_2,\ldots\}$ and a set of *edges* $E=\{e_1,e_2,\ldots\}$; each edge e has two *endpoints*, which are vertices, and they are not necessarily distinct.

Unless otherwise stated, both V and E are assumed to be finite. In this case we say that G is finite.

For example, consider the graph in Figure 1.1. Here, $V=\{v_1,v_2,v_3,v_4,v_5\}$, $E=\{e_1,e_2,e_3,e_4,e_5\}$. The endpoints of e_2 are v_1 and v_2. Alternatively, we say that e_2 is *incident* on v_1 and v_2. The edges e_4 and e_5 have the same endpoints and are therefore called *parallel*. Both endpoints of e_1 are the same – v_1; such an edge is called a *self-loop*.

The *degree* of a vertex v, $d(v)$, is the number of times v is used as an endpoint. Clearly, a self-loop uses its endpoint twice. Thus, in our example, $d(v_1)=4$, $d(v_2)=3$. Also, a vertex whose degree is zero is called *isolated*. In our example, v_3 is isolated since $d(v_3)=0$.

Lemma 1.1 *In a finite graph the number of vertices of odd degree is even.*

Proof: Let $|V|$ and $|E|$ be the number of vertices and edges, respectively. It is easy to see that

$$\sum_{i=1}^{|V|} d(v_i) = 2\cdot|E|,$$

since each edge contributes two to the left-hand side: one to the degree of each of its two endpoints if they are distinct; and two to the degree of its endpoint if the edge is a self-loop. For the left-hand side to sum up to an even number, the number of odd terms must be even. ■

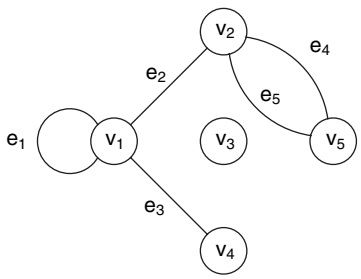

Figure 1.1: Example of a graph.

The notation $u \overset{e}{\text{---}} v$ means that the edge e is incident on vertices u and v. In this case we also say that e *connects* vertices u and v, or that vertices u and v are *adjacent*.

A *path*, P, is a sequence of vertices and edges, interweaved in the following way: P starts with a vertex, say v_0, followed by an edge e_1 incident to v_0, followed by the other endpoint v_1 of e_1, and so on. We write

$$P: \ v_0 \overset{e_1}{\text{---}} v_1 \overset{e_2}{\text{---}} v_2 \cdots$$

If P is finite, it ends with a vertex, say v_l. We call v_0 the *start*-vertex of P and v_l the *end*-vertex of P. The number of edge appearances in P, l, is called the *length* of P. If $l = 0$, then P is said to be *empty*, but it has a start-vertex and an end-vertex, which are identical. (We shall not use the term "path" unless a start-vertex exists.)

In a path, edges and vertices may appear more than once, unless otherwise stated. If no vertex appears more than once, and therefore no edge can appear more than once, the path is called *simple*.

A *circuit*, C, is a finite path in which the start and end vertices are identical. However, an empty path is not considered a circuit. By definition, the start and end vertices of a circuit are the same, and if there is no other repetition of a vertex, the circuit is called *simple*. However, a circuit of length two, $a \overset{e}{\text{---}} b \overset{e}{\text{---}} a$, where the same edge, e, appears twice, is not considered simple. (For a longer circuit, it is superfluous to state that if it is simple, then no edge appears more than once.) A self-loop is a simple circuit of length one.

If for every two vertices u and v of a graph G, there is a (finite) path that starts in u and ends in v, then G is said to be *connected*.

A *digraph* or *directed graph* $G(V, E)$ is defined similarly to a graph, except that the pair of endpoints of every edge is now ordered. If the ordered pair of

endpoints of a (directed) edge e is (u,v), we write

$$u \xrightarrow{e} v.$$

The vertex u is called the *start-vertex* of e; and the vertex, v, the *end-vertex* of e. The edge e is said to be directed from u to v. Edges with the same start-vertex and the same end-vertex are called *parallel*. If $u \neq v$, $u \xrightarrow{e_1} v$ and $v \xrightarrow{e_2} u$, then e_1 and e_2 are *antiparallel*. An edge $u \longrightarrow u$ is called a *self-loop*.

The *out-degree* $d_{out}(v)$ of vertex v is the number of (directed) edges having v as their start-vertex; *in-degree* $d_{in}(v)$ is similarly defined. Clearly, for every finite digraph $G(V,E)$,

$$\sum_{v \in V} d_{in}(v) = \sum_{v \in V} d_{out}(v).$$

A *directed path* is similar to a path in an undirected graph; if the sequence of edges is $e_1, e_2, \cdots$ then for every $i \geqslant 1$, the end-vertex of e_i is the start-vertex of e_{i+1}. The directed path is *simple* if no vertex appears on it more than once. A finite directed path C is a *directed circuit* if the start-vertex and end-vertex of C are the same. If C consists of one edge, it is a *self-loop*. As stated, the start and end vertices of C are identical, but if there is no other repetition of a vertex, C is *simple*. A digraph is said to be *strongly connected* if, for every ordered pair of vertices (u,v) there is a directed path which starts at u and ends in v.

1.2 Computer Representation of Graphs

To understand the time and space complexities of graph algorithms one needs to know how graphs are represented in the computers memory. In this section two of the most common methods of graph representation are briefly described.

Let us consider graphs and digraphs that have no parallel edges. For such graphs, the specification of the two endpoints is sufficient to specify the edge; for digraphs, the specification of the start-vertex and the end-vertex is sufficient. Thus, we can represent such a graph or digraph of n vertices by an $n \times n$ matrix M, where $M_{ij} = 1$ if there is an edge connecting vertex v_i to v_j, and $M_{ij} = 0$, if not. Clearly, in the case of (undirected) graphs, $M_{ij} = 1$ implies that $M_{ji} = 1$; or in other words, M is symmetric. But in the case of digraphs, any $n \times n$ matrix of zeros and ones is possible. This matrix is called the *adjacency matrix*.

Given the adjacency matrix M of a graph, one can compute $d(v_i)$ by counting the number of ones in the i-th row, except that a one on the main diagonal represents a self-loop and contributes two to the count. For a digraph, the number

of ones in the i-th row is equal to $d_{out}(v_i)$, and the number of ones in the j-th column is equal to $d_{in}(v_j)$.

The adjacency matrix is not an efficient representation of the graph if the graph is *sparse*; namely, the number of edges is significantly smaller than n^2. In these cases, it is more efficient to use the *incidence lists* representation, described later. We use this representation, which also allows parallel edges, in this book unless stated otherwise.

A *vertex array* is used. For each vertex v, it lists v's incident edges and a pointer indicating the *current* edge. The *incidence list* may simply be an array or may be a linked list. Initially, the pointer points to the first edge on the list. Also, we use an *edge array*, which tells us for each edge its two endpoints (or start-vertex and end-vertex, in the case of a digraph).

Assume we want an algorithm TRACE(s, P), such that given a finite graph $G(V, E)$ and a start-vertex $s \in V$ traces a maximal path P that starts at s and does not use any edge more than once. Note that by "maximal" we do not mean that the resulting path, P, will be the longest possible; we only mean that P cannot be extended, that is, there are no unused incident edges at the end-vertex.

We can trace a path starting at s by taking the first edge e_1 on the incidence list of s, marking e_1 as "used" in the edge array, and looking up its other endpoint v_1 (which is s if e_1 is a self-loop). Next, use the vertex array to find the pointer to the current edge on the list of v_1. Scan the incidence list of v_1 for the first unused edge, take it, and so on. If the scanning hits the last edge and it is used, TRACE(s, P) halts. A PASCAL-like description of TRACE(s, P) is presented in Algorithm 1.1. Here is a list of the data structures it uses:

(i) A *vertex table* such that, for every $v \in V$, it includes the following:
  – A list of the edges incident on v, which ends with *NIL*
  – A pointer $N(v)$ to the current item in this list. Initially, $N(v)$ points to the first edge on the list (or to *NIL*, if the list is empty).

(ii) An *edge table* such that every $e \in E$ consists of the following:
  – The two endpoints of e
  – A flag that indicates whether e is *used* or *unused*. Initially, all edges are *unused*.

(iii) An array (or linked list) P of edges that is initially empty, and when TRACE(s, P) halts, will contain the resulting path.

Notice that in each application of the "while" loop of TRACE (lines 2–10 in Algorithm 1.1), either $N(v)$ is moved to the next item on the incidence list of v (line 4), or lines 6–10 are applied, but not both. The number of times line

```
Procedure TRACE(s, P)

1   v ← s
2   while N(v) points to an edge (and not to NIL) do
3       if N(v) points to a used edge do
4           change N(v) to point to the next item on the list
5       else do
6           e ← N(v)
7           change the flag of e to used
8           add e to the end of P
9           use the edge table to find the second endpoint of e, say u
10          v ← u
```

Algorithm 1.1: The TRACE algorithm.

4 is applied is clearly $O(|E|)$. The number of times lines 6–10 are applied is also $O(|E|)$, since the flag of an unused edge changes to used, and each of these lines takes time bounded by a constant to run. Thus, the time complexity of TRACE is $O(|E|)$.[1] (In fact, if the length of the resulting P is L then the time complexity is $O(l)$; this follows from the fact that each edge that joins P can "cause a waste" of computing time only twice: once when it joins P and, at most, once again by its appearance on the incidence list of the adjacent vertex.)

If one uses the adjacency matrix representation, in the worst case, the tracing algorithm takes time (and space) complexity $\Omega(|V|^2)$.[2] And if $|E| << |V|^2$, as is the case for *sparse* graphs, the complexity is reduced by using the incidence-list representation. Since in most applications, the graphs are sparse, we prefer to use the incidence-list representation.

Note that in our discussions of complexity, we assume that the word length of our computer is sufficient to store the names of our atomic components: vertices and edges. If one does not make this assumption, then one may have to allow $\Omega(\log(|E|+|V|))$ bits to represent the atomic components, and to multiply the complexities by this factor.

[1] $f(x)$ is $O(g(x))$ if there are two constants k_1 and k_2, such that for every x, $f(x) \leqslant k_1 \cdot g(x) + k_2$.
[2] $f(x)$ is $\Omega(g(x))$ if there are two constants k_3 and k_4, such that for every x, $f(x) \leqslant k_3 \cdot g(x) + k_4$.

1.3 Euler Graphs

An *Euler path* of a finite undirected graph $G(V, E)$ is a path such that every edge of G appears on it once. Therefore, the length of an Euler path is $|E|$. If G has an Euler path, then it is called an *Euler graph*.

Theorem 1.1 *A finite (undirected) connected graph is an Euler graph if and only if exactly two vertices are of odd degree or all vertices are of even degree. In the latter case, every Euler path of the graph is a circuit, and in the former case, none is.*

As an immediate conclusion of Theorem 1.1 we observe that none of the graphs in Figure 1.2 is an Euler graph because both have four vertices of odd degree. The graph shown in Figure 1.2(a) is the famous Königsberg bridge problem solved by Euler in 1736. The graph shown in Figure 1.2(b) is a common misleading puzzle of the type "draw without lifting your pen from the paper."

Proof: It is clear that if a graph has an Euler path that is not a circuit, then the start-vertex and the end-vertex of the path are of odd degree, while all the other vertices are of even degree. Also, if a graph has an Euler circuit, then all vertices are of even degree.

Assume now that G is a finite graph with exactly two vertices of odd degree, a and b. We now describe an algorithm (A), which will find an Euler path from a to b.

First, trace a maximal path P, as in the previous section, starting at vertex a. Since G is finite, the algorithm halts, producing a path. But as soon as the path emanates from a, one of the edges incident to a is used, and a's residual degree becomes even. Thus, every time a is reentered, there is an unused edge to leave

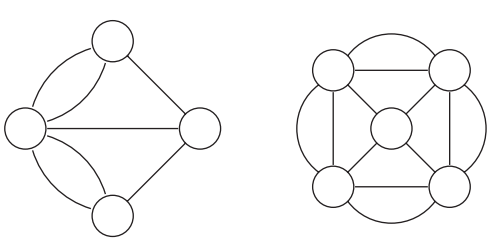

Figure 1.2: Non-Eulerian graphs.

a by. This proves that P cannot end in a. Similarly, if vertex $v \in V \setminus \{a, b\}$, then P cannot end in v. It follows that P ends in b.

If P contains all the edges of G, we are done. If not, we make the following observations:

- The residual degree of every vertex is even.
- There is an unused edge incident on some vertex v that is on P. To see that this must be so, let $u \overset{e}{\text{---}} w$ be an unused edge. If either u or w is on P, we are done. If not, since G is connected, there is a path Q from a to u. There must be unused edges on Q. Going from a on Q, the first unused edge we encounter fits the bill.

Now, trace a maximal path P' in the residual graph, which consists of the set V of vertices and all edges of E that are not in P. Start P' at v. Since all vertices of the residual graph are of even degree, P' ends in v (and is therefore a circuit). Next, combine P and P' to form one path from a to b as follows: Follow P until it enters v. Now, incorporate P', and then follow the remainder of P.

Repeat, incorporating additional circuits into the present path as long as there are unused edges. Since the graph is finite, this process will terminate, yielding an Euler path.

If all vertices of the graph are of even degree, the first traced path can start at any vertex, and will be a circuit. The remainder of the algorithm is similar to to this process. ■

In the case of digraphs, a *directed Euler path* is a directed path in which every edge appears once. A *directed Euler circuit* is a directed Euler path for which the start and end vertices are identical. In addition, digraph is called *Euler* if it has a directed Euler path (or circuit).

The *underlying (undirected) graph* of a digraph is the graph resulting from the digraph if the direction of the edges is ignored. Thus, the underlying graph of the digraph in Figure 1.3(a) is shown in Figure 1.3(b).

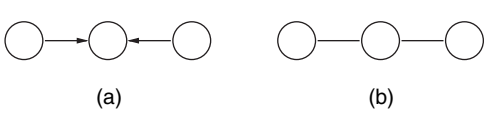

Figure 1.3: A digraph and its underlying graph.

Theorem 1.2 *A finite digraph is an Euler digraph if and only if its underlying graph is connected and one of the following two conditions holds:*

(i) There is one vertex a *such that* $d_{out}(a) = d_{in}(a)+1$*, and another vertex* b *such that* $d_{out}(b)+1 = d_{in}(b)$*, while for every other vertex* v*,* $d_{out}(v) = d_{in}(v)$.
(ii) For every vertex v*,* $d_{out}(v) = d_{in}(v)$.

In the former case, every directed Euler path starts at a *and ends in* b*. In the latter, every directed Euler path is a directed Euler circuit.*

The proof of Theorem 1.2 is along the same lines as the proof of Theorem 1.1, and is therefore not repeated here.

Let us make now a few comments about the complexity of the algorithm $\mathcal{A}$ for finding an Euler path, as described in the proof of Theorem 1.1. Our purpose is to show that the time complexity of the algorithm is $O(|E|)$.

Assume $G(V, E)$ is presented in the incidence list's data structure. The main path P and the detour P' will be represented by linked lists, where each item on the list represents an edge.

In the vertex table, we add for each vertex v the following two items:

(i) A flag that indicates whether v is already on the main path P or the detour P'. Initially, this flag is "unvisited."
(ii) For every visited vertex v, there is a pointer $E(v)$ to the location on the path of the edge through which v was first encountered. Initially, for every v, $E(v) =$*NIL*.

We shall also use a list L of visited vertices. Each vertex enters L once, when its flag is changed from "unvisited" to "visited."

$\mathcal{A}$ starts by running TRACE(a, P), updating the vertices' flags, and $E(v)$ for each newly visited vertex v. Next, the following loop is applied:

If L is empty, $\mathcal{A}$ halts. If not, take a vertex v from L, and remove v from L. Use TRACE(v, P') to produce P'. Look up edge $E(v)$, recording the location of the edge e it is linked to. Change this link to point to the first edge on P'. Now, let the last edge of P' point to e.

Note that when TRACE(v, P') terminates, v has no unused incident edges. This explains why we can remove v from L.

Now that P' has been incorporated into P, the loop is repeated.

It is not hard to see that both the time and space complexities of $\mathcal{A}$ are $O(|E|)$.

1.4 De Bruijn Sequences

Let $\Sigma = \{0, 1, ..., \sigma - 1\}$ be an alphabet of σ letters. Clearly, there are $L = \sigma^n$ different words of length n over Σ. A *de Bruijn sequence* is a (circular) sequence $a_0 a_1 \cdots a_{L-1}$ over Σ such that for every word w of length n over Σ there exists a (unique) $0 \leqslant j < L$ such that

$$a_j a_{j+1} \cdots a_{j+n-1} = w,$$

where the computation of the indexes is modulo L.

The most important case is that of $\sigma = 2$. Binary de Bruijn sequences are of great importance in coding theory and can be generated by shift registers. (See Golomb, 1967, on the subject.) In this section we discuss the existence of de Bruijn sequences for every σ and every n.

For that purpose let us define the *de Bruijn digraph* $G_{\sigma,n}(V, E)$ as follows:

(i) $V = \Sigma^{n-1}$; i.e., the set of all σ^{n-1} words of length $n-1$ over Σ.
(ii) $E = \Sigma^n$.
(iii) The directed edge $b_1 b_2 \cdots b_n$ starts at vertex $b_1 b_2 \cdots b_{n-1}$ and ends in vertex $b_2 b_3 \cdots b_n$.

Digraphs $G_{2,3}$ and $G_{2,4}$ are shown in Figure 1.4. $G_{3,2}$ is shown in Figure 1.5.

Observe that if $w_1, w_2 \in \Sigma^n$, then w_2 can follow w_1 in a de Bruijn sequence only if in $G_{\sigma,n}$ the edge w_2 starts at the vertex in which w_1 ends. It follows that there is a de Bruijn sequence for σ and n if and only if there is a directed Euler circuit in $G_{\sigma,n}$.

For example, consider the directed Euler circuit of $G_{2,3}$, which consists of the following sequence of directed edges:

$$000,\ 001,\ 011,\ 111,\ 110,\ 101,\ 010,\ 100.$$

The corresponding de Bruijn sequence, 00011101, follows by reading the first letter of each word (edge) in the circuit.

Theorem 1.3 *For every σ and n, $G_{\sigma,n}$ has a directed Euler circuit.*

Proof: To use Theorem 1.2 we have to show that the underlying undirected graph of $G_{\sigma,n}$ is connected and that for every vertex v, $d_{out}(v) = d_{in}(v)$.

Let us show that $G_{\sigma,n}$ is strongly connected. This implies that its underlying undirected graph is connected.

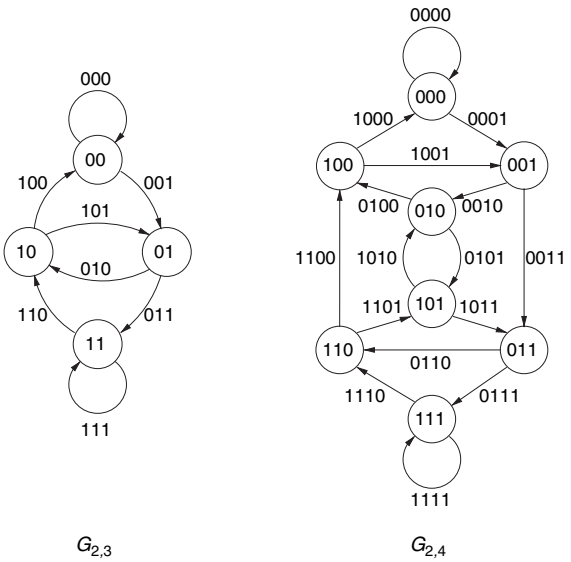

Figure 1.4: Examples of de Bruijn digraphs for $\sigma = 2$.

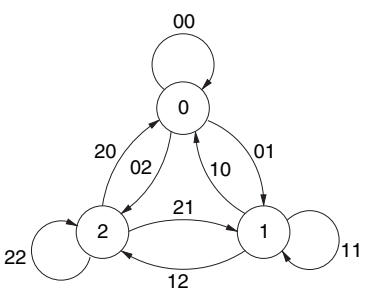

Figure 1.5: $G_{3,2}$

Let $b_1b_2\cdots b_{n-1}$ and $c_1c_2\cdots c_{n-1}$ be any two vertices. The directed path

$$b_1b_2\cdots b_{n-1}c_1,\ b_2b_3\cdots b_{n-1}c_1c_2,\ \ldots,b_{n-1}c_1\cdots c_{n-1}$$

is of length $n-1$, it starts at vertex $b_1b_2\cdots b_{n-1}$ and ends in vertex $c_1c_2\cdots c_{n-1}$, showing that $G_{\sigma,n}$ is strongly connected. (Observe that this directed path is not necessarily simple; it may use vertices and edges more than once.)

Now, observe that for each vertex $v = b_1b_2\cdots b_{n-1}$, every outgoing edge is of the form $b_1b_2\cdots b_{n-1}c$, where c can be any of the σ letters. Thus, $d_{out}(v) = \sigma$.