design of Sanskrit memory wheels[1], and by illusionists in mind-reading effects.

What ties these two topics together is that they are more naturally modelled as questions about Eulerian cycles in graphs rather than about Hamilton paths in graphs. One of our main aims in this chapter is to show that De Bruijn cycles always exist, and how to generate them quickly. Along the way we encounter other important combinatorial objects, such as necklaces, Lyndon words, and primitive polynomials over finite fields.

7.1 Eulerian Cycles

Certain combinatorial Gray code questions are more naturally posed as Eulerian cycle questions rather than as Hamiltonian cycle questions. Recall that an Eulerian cycle in a (multi)graph is a cycle that includes every edge exactly once. There is a simple characterization of Eulerian graphs, namely as given in Lemma 2.6: a connected (multi)graph is Eulerian if and only if every vertex has even degree.

Now back to the domino problem. Consider the complete graph K_7 with vertices labelled $0, 1, 2, 3, 4, 5, 6$ and with the self-loops $\{i, i\}$ added to each vertex i. There are 28 edges in this graph and each edge corresponds to a domino. An Eulerian cycle corresponds to a maximal domino game like that shown in Figure 7.1. Eulerian cycles clearly exist since each vertex has even degree 8.

7.1.1 Generating an Eulerian cycle

Let G be an directed Eulerian multigraph with n vertices. In this section we develop an algorithm that will generate an Eulerian cycle in in G. Along the way we will discover a nice formula for the number of Eulerian cycles and a CAT algorithm for generating all Eulerian cycles of G.

If G is a directed multigraph then $\overline{G}$ is the directed multigraph obtained by reversing the directions of the edges of G (i.e., for each edge (u, v) of G, there is a corresponding edge (v, u) in $\overline{G}$.

There is a simple algorithm for finding an Eulerian cycle in G given an in-tree rooted at some vertex r. We simply start at r and successively pick edges subject to the restriction that an edge of T is *not* used unless there is no other choice. A simple implementation of this rule is contained in Algorithm 7.1, and an example of its use is to be found in Figure 7.2.

At step (E1) we find a spanning out-tree T of $\overline{G}$ that is rooted at r. In general there may be several spanning out-trees rooted at r and it doesn't matter which one is used; depth-first search is a convenient and efficient way to find such a tree. The tree T is a spanning in-tree of G. We now modify (at line (E2)) the adjacency lists of G so that the unique edge $(u, v) \in T$ on the list for u is at the end of the list. We now say that the adjacency lists are *extreme* with respect to T. The remaining lines simply extract edges from the adjacency lists, destroying the lists in the process.

The running time of this algorithm is $O(m)$ where m is the number of edges in G. Since G is Eulerian, $m \geq n$. The depth-first search at line (E2) runs in time $O(n + m)$. The

[1]The nonsense Indian word yamátárájabhánasalagám is a mnemonic way of remembering the sequence (7.1), where an accented vowel represents a 1 and an unaccented vowel represents a 0. This word was used by medieval Indian poets and musicians as an aid in remembering all possible rythms. (There are 10 bits because the last two have been wrapped around.)

uvic01, ©Frank Ruskey, 1995–2001

(E1)	Use depth-first-search to find a spanning out-tree T of $\overline{G}$ rooted at r.
(E2)	Compute adjacency lists `adj` of G that are extreme with respect to T.
(E3)	$u := r$;
(E4)	**while** $adj[u] \neq$ **null do**
(E5)	$v := adj[u].vert$; $adj[u] := adj[u].next$;
(E6)	Output((u, v));
(E7)	$u := v$;

Algorithm 7.1: Algorithm to find an Eulerian cycle in a directed multigraph.

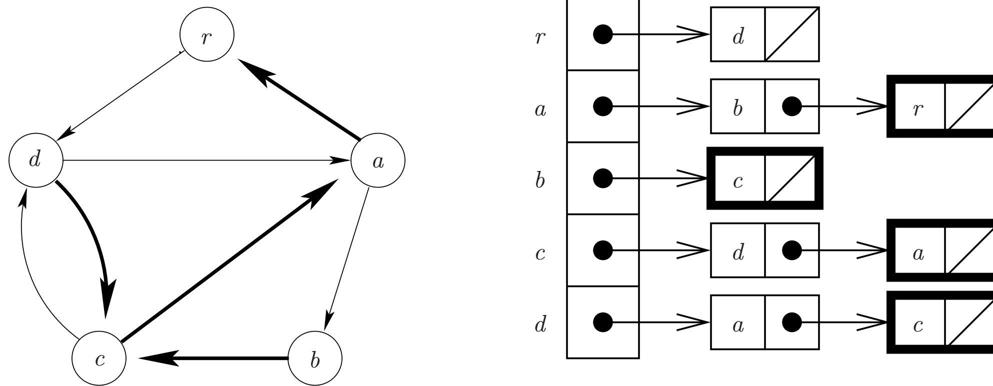

Figure 7.2: Example of finding an Euler cycle in a directed multigraph. (a) A directed multigraph G, with spanning in-tree found by depth-first search of $\overline{G}$ shown in bold, and (b) extreme adjacency lists with respect to T, with edges of T thicker. The cycle produced by this example is $r, d, a, b, c, d, c, a, r$.

uvic01, ©Frank Ruskey, 1995–2001

computation of $\overline{G}$ takes time $O(m)$ the remaining computation (line (E3-E6)) takes time $O(m)$ since the body of the while loop at line (E3) is executed exactly m times.

THEOREM 7.1 *Given a directed Eulerian multigraph G, Algorithm 7.1 outputs a list of edges along an Eulerian cycle of G.*

PROOF: Since G is Eulerian, the path P produced by the algorithm must end at r. Imagine that there is some edge (v, w) that is not in P. Since the algorithm terminated it must be the case that $v \neq r$. Clearly, any edge on the adjacency list for v that follows (v, w) must also not be in P. Thus, because the edge lists are extreme with respect to T, we may assume that (v, w) is in T. Since $G - P$ is balanced, there is an edge (u, v) also not in P, which again we can take to be in T. Continuing in this manner we obtain a path of edges in $(G - P) \cap T$ that terminates at r. But then, since $G - P$ is balanced, it must contain an edge (r, q), contrary to the terminating condition of the algorithm. □

How many Eulerian does a connected, balanced multigraph G have? In answering this question we regard an Eulerian cycle as being a *circular* list of edges; the edge that starts the list is immaterial. The answer is provided by our algorithm. Clearly, different in-trees T produce different cycles, as do different adjacency lists that are extreme with respect to T. A graph G has $\tau(G)$ different spanning in-trees rooted at a given vertex r and there are $(d^+(v) - 1)!$ ways of arranging the adjacency list of v so that it is extreme with respect to T. Thus it is plausible that the number of Eulerian cycles in G is

$$\tau(G) \prod_{v \in V} (d^-(v) - 1)!. \tag{7.2}$$

To prove (7.2) we must show that we can recover the adjacency lists and tree T from an Eulerian cycle C. Fix an edge (r, s) to be the first on the cycle. Define the adjacency list for vertex v to simply be the edges of the form (v, w) in the order that they are encountered on C. With these adjacency lists lines (E3-E8) will produce the cycle C. To finish the proof we need to show that the the collection of edges

$$S = \{(v, v') \in E \mid v \neq r \text{ and } (v, v') \text{ is the last occurence of } v \text{ on } C\}$$

is an in-tree rooted at r. The set S contains $n - 1$ edges; we must show that it forms no cycles. Assume to the contrary that such a cycle X exists and let (y, z) be the first of its edges that occur on C, and let (x, y) be the previous edge on X. Unless $y = r$, there is some edge (y, z') that follows (x, y) on C, in contradiction to the way (y, z) was chosen. But we cannot have $y = r$ either since then $(y, z) = (r, z)$ would be in S.

How fast can we generate all Eulerian cycles in a graph? We need to generate permutations of edges on adjacency lists. There are many CAT algorithms for generating permutations (e.g., Algorithms ???, ???, and ???). We also need to generate spanning trees of a graph. This topic is taken up in Chapter ???. There are CAT algorithms for generating spanning trees of undirected graphs, but what about spanning in-trees of directed graphs? **!!! Is this a simple reduction or a research problem ???**

7.1.2 An Eulerian Cycle in the Directed n-cube

Given the central role played by hypercubes in the previous chapters, it is fitting that the next problem is one that can be modeled on the n-cube. We must admit, however, that it is

uvic01, ©Frank Ruskey, 1995–2001