Graphs

Formulating a simple, precise specification of a computational problem is often a prerequisite to writing a computer program for solving the problem. Many computational problems are best stated in terms of graphs: A directed graph $G(V,E)$ consists of a finite set of vertices V and a set of (directed) edges or arcs E. An edge is an ordered pair of vertices (v,w) and is usually indicated by drawing a line between v and w, with an arrow pointing towards w. Stated in mathematical terms, a directed graph $G(V,E)$ is just a binary relation $E \subseteq V \times V$ on a finite set V. Undirected graphs may be regarded as special kinds of directed graphs, such that $(u,v) \in E \leftrightarrow (v,u) \in E$. Thus, since the directions of the edges are unimportant, an undirected graph $G(V,E)$ consists of a finite set of vertices V, and a set of edges E, each of which is an unordered pair of vertices $\{u,v\}$. As we have defined them, graphs are allowed to have self-loops; i.e. edges of the form (u,u) that go from a vertex to itself. Sometimes it is more convenient to disallow such self-loops.

Graphs are useful for modeling a diverse number of situations. For example, the vertices of a graph might represent cities, and edges might represent highways that connect them. In this case, the edges would be undirected. Alternatively, an edge might represent a flight from one city to another, and now edges would be directed (a certain airlines might have a non-stop flight from SFO to LAX, but no non-stop flight back from LAX to SFO).

Graphs can also be used to represent more abstract relationships. For example, the vertices of a graph might represent tasks, and the edges might represent precedence constraints: a directed edge from u to v says that task u must be completed before v can be started. An important problem in this context is scheduling: in what order should the tasks be scheduled so that all the precedence constraints are satisfied.

A **path** in a directed graph $G=(V,E)$ is a sequence of neighbor edges $(v_1,v_2),(v_2,v_3),...,(v_{n-2},v_{n-1}),(v_{n-1},v_n)$. In this case we say that there is a path between v_1 and v_n. A path in an undirected graph is defined similarly. A path is called **simple** if it has no repeating vertices.

A graph is called **connected** if there is a path between any two distinct vertices.

If $G=(V,E)$ is an undirected graph then the **degree** of vertex $v \in V$ is the number of edges incident to v. In notation, $degree(v) = |\{u \in V : \{v,u\} \in V\}|$. A vertex v whose degeee is 0 is called an **isolated vertex**.

Eulerian Paths and Tours

Euler invented graph theory trying to solve a problem that today is called The Seven Bridges of Königsberg.

An **Eulerian path** is a path in a graph that uses each edge exactly once (sometimes to emphasize that Eulerian paths are not simple, i.e. that they might go through a vertex multiple number of times, they are called Eulerian walks). An **Eulerian tour** is Eulerian path whose starting point is also an end point.

The next theorem gives necessary and sufficient conditions of a graph having Eulerian tour.

Theorem. An undirected multigraph $G=(V,E,F)$ has Eulerian tour if and only if

1)when isolated vertices of G are removed G is connected graph and

2)for every vertex $v \in V$, degree of v is even.

Let us prove "only if" part. So we know that a graph has Eulerian tour. That means that every vertex that has an edge is in the tour, therefore it is connected with all other vertices in the tour. That proofs condition 1). For each vertex on the tour, except the first one, walk leaves it just after entering, thus every time using two edges, therefore it uses even number of edges. Since Eulerian tour uses each edge exactly once, every vertex has even degree. Also the first vertex has even degree because walk leaves it in the beginning but returns at the end. That proves condition 2).

Our strategy to construct a Eulerian graph and thus to prove "if" part is to:

1) start walking from some vertices u, never repeating edges until we are stuck. By fact 1 below, we can only get stuck at u.

2) Pick some untraversed edge $\{u', v'\}$ with one end point u' on connected walk.

3) Repeat 1) starting from u' and splice in resulting closed walk.

The strategy relies on three simple facts that can be shown by an induction.

Fact 1. If a graph G has only even degree vertices then a walk starting from any vertex s that does not repeat edges can get stuck only at s.

Informal idea to prove this fact is that since every vertex has even number of edges, we can go out of it if we came in.

Fact 2. If a graph (except its isolated vertices) is connected and not all edges are traversed in a path P, then we can find a vertex that is common with path P and has unused edge.

Fact 3. Two closed paths with a common vertex can be combined in one closed path.

de Bruijn Graphs

de Bruijn sequence is a 2^n bit circular sequence such that every string of length n occurs as a contiguous substring of the sequence exactly once.

de Bruijn graph $G = (V, E)$ is used to generate such sequences. A set of vertices is a set of all $n-1$ bit strings, formally $V = \{0,1\}^{n-1}$. Each vertex $a_1a_2...a_{n-1}$ has two outgoing edges $(a_1a_2...a_{n-1}, a_2a_3...a_{n-1}0)$, $(a_1a_2...a_{n-1}, a_2a_3...a_{n-1}1)$, therefore also two incoming edges $(0a_1a_2...a_{n-2}, a_1a_2...a_{n-1})$, $(1a_1a_2...a_{n-2}, a_1a_2...a_{n-1})$.

de Bruijn sequence is generated by Eulerian tour in this graph. The theorem above can be modified to work for directed multigraphs, too. All we need to change is the second condition; it must say: "for every vertex v in V, indegree of v equals outdegree of v". Obviously, de Bruijn graph satisfies this and the other condition, therefore it has Eulerian tour T.

Starting from any vertex, walk along T, by adding a corresponding bit for every edge. It turns out to be de Bruijn sequence.