Maggie Johnson
CS109

Graph Theory: The Basics

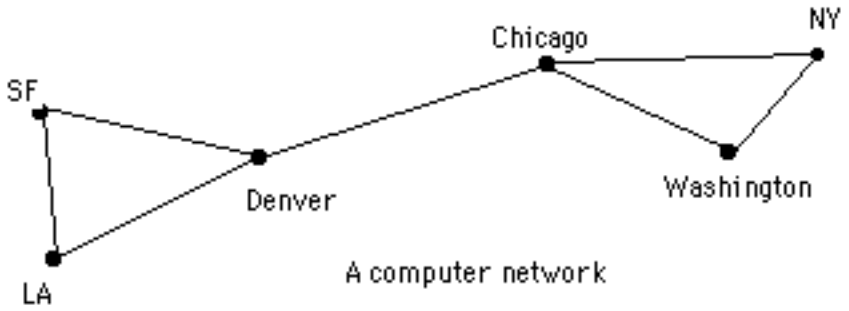
Key Topics

* Introduction
* Graph Lingo
* Some Special Graphs
* Applications of Special Graphs
* Graph Isomorphism
* Graph Traversals

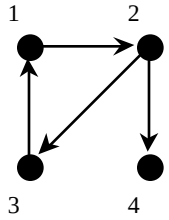
A graph is a convenient representation that we encountered earlier this quarter when we discussed relations. A graph is characterized by a many-to-many relationship between its elements. Some pictures of graphs:

A **simple graph** G = (V,E) consists of V, a nonempty set of vertices or nodes (the dots); and E, a (possibly empty) set of unordered pairs of distinct elements of V called edges (the lines).

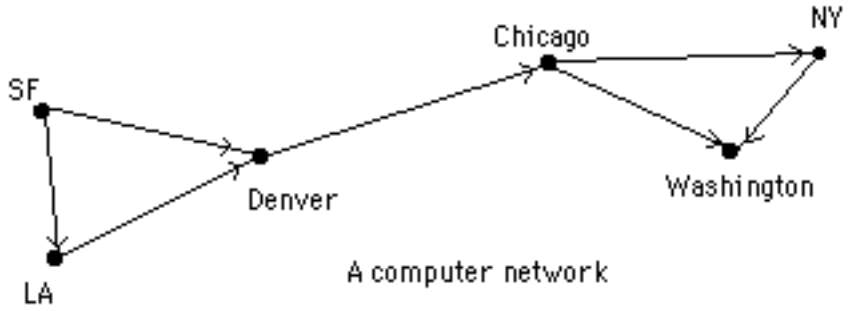
Simple graphs are often used to represent network models (models with many-many links between elements) where the only required information is the location of the connections:



A **directed graph** (**digraph**) G = (V,E) consists of V, a nonempty set of vertices or nodes; and E, a set of ordered pairs of distinct elements of V called edges.



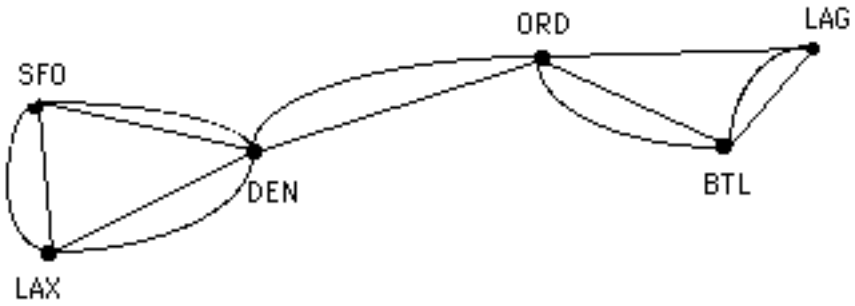
Directed graphs are also used to model networks where the direction of the connections is important.



A computer network

A **multigraph** G = (V,E) consists of V, a nonempty set of vertices; and E, a set of unordered pairs of distinct elements of V called edges. Multiple edges between two vertices are allowed in a multigraph.

Multigraphs are used to model situations where there is more than one connection between two vertices. For example, a flight map may have several flights between two cities:



A **directed multigraph** G = (V,E) consists of V, a nonempty set of vertices; and E, a set of ordered pairs of distinct elements of V called edges. Multiple edges between two vertices are allowed in a multigraph.

Finally, a pseudograph is an undirected multigraph with loops:

To summarize then, pseudographs are the most general type of graphs since they may contain loops and multiple edges. Multigraphs come next allowing multiple edges. Simple graphs allow neither.

Graph Lingo

>>> Undirected Graphs:

1) Two vertices u and v in an undirected graph G are called *adjacent* (or neighbors) if {u,v} is an edge in G (note {} denotes unordered pair). If e is the edge connecting u and v, then e is said to be *incident with* vertices u and v, or e *connects* endpoints u and v.

2) The *degree* of a vertex in an undirected graph is the number of edges incident with it (a loop contributes twice to the degree of its vertex). The degree of vertex v is denoted deg(v). The degree of LAX in the multigraph above is 4; the degree of DEN is 6.

3) A vertex of degree 0 is called *isolated*. A vertex is *pendant* if it has degree 1.

What do we get when we add the degrees of all the vertices of a graph? Each edge must contribute 2 to the total sum because each edge is incident with exactly two (possibly equal) vertices. This gives us the famous Handshaking Theorem:

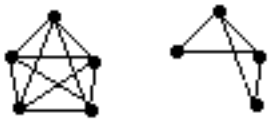
4) Handshaking Theorem: The sum of the degrees of the vertices of any undirected graph is twice the number of edges.

This theorem shows that the sum of the degrees of an undirected graph is always even. This simple fact has many consequences:

5) An undirected graph has an even number of vertices of odd degree.

> Proof: Let V1 be the set of vertices of even degree, and V2 be the set of vertices of odd degree in a graph G. So, we know that (sum of degrees of vertices of V1 + sum of degrees of vertices of V2) = 2 * the number of edges.
>
> We know that all the vertices of V1 are of even degree, so we know that the first term of the equality is also even. We also know that the sum of both terms must be even because the sum is equal to 2e. Therefore, the second term in the sum must also be even. Since all the terms in this sum are odd, there must be an even number of such terms. So, there are an even number of vertices of odd degree.

6) A *subgraph* of a graph G = (V,E) is a graph H = (W,F) where W is a subset of V and F is a subset of E.



7) The *union* of two simple graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is the simple graph with the vertex set $V_1 \cup V_2$ and edge set $E_1 \cup E_2$. The union of G_1 and G_2 is denoted $G_1 \cup G_2$.

>>> Directed Graphs:

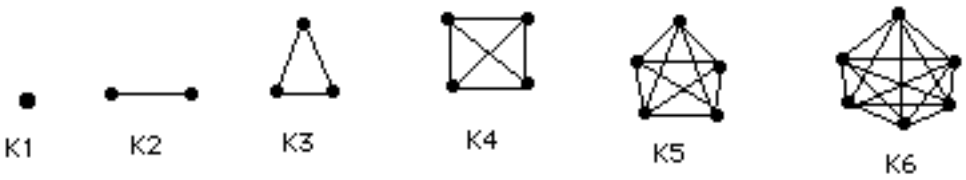
1) When (u,v) is an edge in a directed graph G, u is said to be adjacent to v and v is adjacent from u. The vertex u is the initial vertex of (u,v) and v is the terminal or end vertex of (u,v). The initial and terminal vertex of a loop are the same.

2) The in-degree of vertex v denoted by $\deg^-(v)$ is the number of edges with v as their terminal vertex (the number of edges coming into v). The out-degree of vertex v denoted by $\deg^+(v)$ is the number of edges with v as their initial vertex (the number of edges going out of v). A loop contributes one each to the in-degree and out-degree of its vertex. Vertex 2 of the small directed graph on page 1 has $\deg^+(2) = 2$ and $\deg^-(2) = 1$.

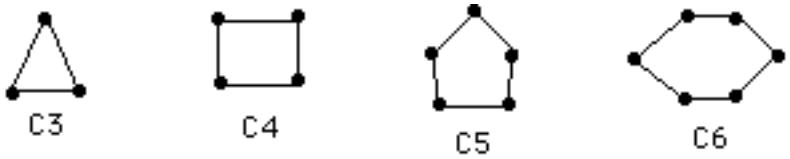
3) Since each edge has an initial vertex and a terminal vertex, the sum of the in-degrees and out-degrees of all the vertices in a graph are the same. Both of these sums equal the number of edges in the graph.

Some Special Graphs

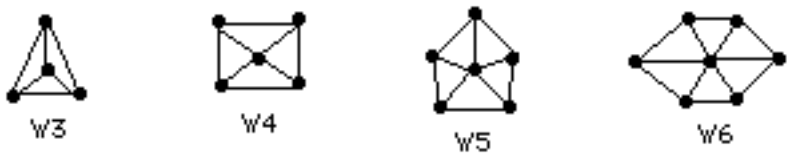
1) Complete Graphs: A complete graph on n vertices (denoted K_n) is the simple graph that contains exactly one edge between each pair of distinct vertices. The graphs K_n for n = 1..6 are displayed below:



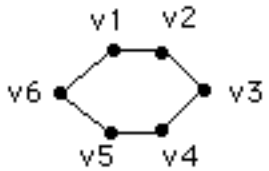
2) Cycles: The cycle C_n, n >= 3, consists of n vertices $v_1, v_2, \ldots v_n$ and edges $\{v_1, v_2\}$, $\{v_2, v_3\}, \ldots \{v_{n-1}, v_n\}$, and $\{v_n, v_1\}$. The cycles C_3, C_4, C_5 and C_6 are displayed below:



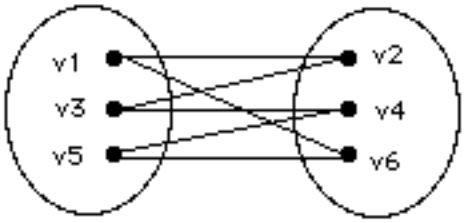
3) Wheels: We obtain the wheel W_n when we add an additional vertex to the inside of cycle C_n, for n >= 3, and connect this new vertex to each of the n vertices of C_n, by new edges. The wheels W_3, W_4, W_5 and W_6 are given below:



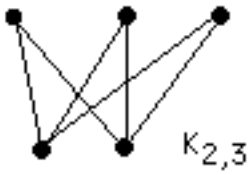
4) Bipartite Graphs: a simple graph G is called bipartite if its vertex set V can be partitioned into two disjoint nonempty sets V_1 and V_2 such that every edge in the graph connects a vertex in V_1 to a vertex in V_2, and no edge connects two vertices in V_1 or V_2.



C6



5) Complete Bipartite Graphs: A bipartite graph $K_{m,n}$ is complete if its two vertex sets have m and n vertices respectively, and every vertex in V_1 connects to every vertex in V_2..



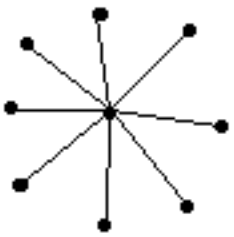
Some Applications of Special Graphs

Local Area Network Topologies: All the various computers and printers (as well as other peripherals) can be connected using a local area network. This makes it possible for printers and hard drives (as well as those other peripherals) to be shared by everyone on the network. There are three standard methods for connecting the separate nodes of a local area network - these are called topologies.

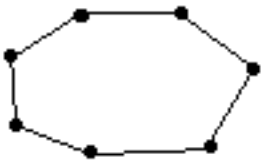
1) A *star* topology is where all the devices are connected to one central control device that handles all communications. This can be represented as a complete bipartite graph $K_{1,n}$ as shown in (A) below. All messages are sent via the central control device.

2) A *ring* topology is where each device is connected to exactly two other devices. These can be modeled using a cycle C_n as shown in (B) below. Messages are sent from device to device around the cycle until the intended device receives it.

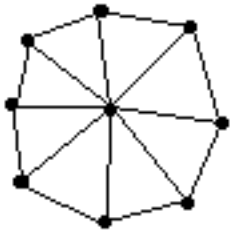
3) The problem with both a star and ring topology is the possibility of total network failure. With a star topology, the entire network is dependent on the central control device. If that goes down, so does the network. With a ring topology, a failure of any device on the network causes the whole network to fail. A third topology handles these risks with redundancy, by combining a star and ring topology. These *hybrid* networks can be modeled using a wheel W_n as shown in (C) below.



A

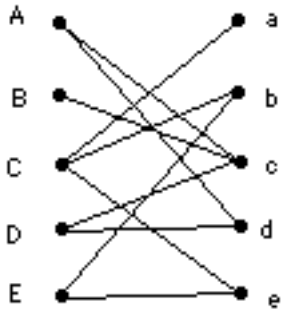


B



C

Job Search: Suppose we have five people A, B, C, D, E and five jobs a, b, c, d, e; various people are qualified for the five various jobs. The problem is to find a feasible one-to-one matching of people to jobs, or show that no such matching exists. We can represent this situation by a bipartite graph with vertices for each person and each job, and edges between indicating which people are qualified for which jobs.

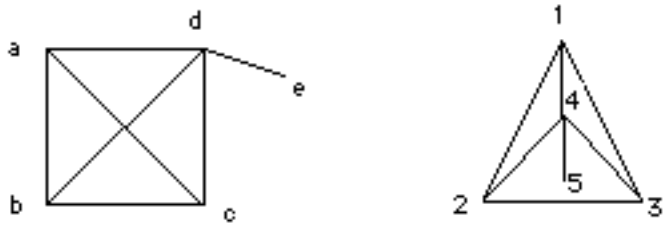


Does there exist a feasible matching of people to jobs in the above graph? _____

The answer can be found by considering people A, B, and D. These three people as a set are collectively qualified for only two jobs (c and d), so there is no feasible matching possible for these three people (an application of the our old favorite, the Pigeonhole Principle).

Graph Isomorphism

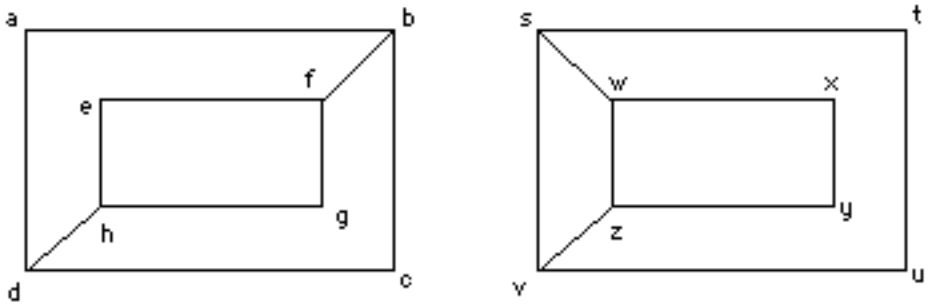
When are two graphs equivalent? Are the following two 5-vertex graphs equivalent? _____



First we have to define "equivalent". If we consider the underlying set of vertices and edges, we see that equivalent should mean that we can find a one-to-one correspondence between the vertices in the two graphs such that a pair of vertices are adjacent in one graph if and only if the corresponding pair of vertices are adjacent in the other graph. Such a user-defined one-to-one correspondence between vertices is called an **isomorphism**, and the two graphs are then called **isomorphic** (note: isos comes from the Greek for "equal" and morphe for "form"). The two graphs above are isomorphic if we map as follows: 1-a, 2-b, 3-c, 4-d, 5-e.

To be isomorphic, two graphs must have the same number of vertices and edges, and the degrees of each vertex must be the same. But that's not enough to show that two graphs are isomorphic. You must map the one-to-one correspondence to be sure.

Are the following graphs isomorphic? _____



Each graph has 8 vertices and 10 edges. They also both have four vertices of degree 2 and four of degree 3. It looks like they might be isomorphic, but they are not. Notice that deg(*a*) = 2. This vertex must correspond to either *t*, *u*, *x*, or *y* in the other graph since these are the only vertices of degree 2. However, each of these four vertices is adjacent to other vertices of degree 2 and degree 3. But vertex *a* is adjacent to two vertices of degree 3.

Note that one quick way of testing if two graphs are isomorphic is seeing if their matrix representation are the same. (We looked at matrix representations when studying relations). The matrices of the two isomorphic graphs given in the first example are shown below:

	a	b	c	d	e
a	0	1	1	1	0
b	1	0	1	1	0
c	1	1	0	1	0
d	1	1	1	0	1
e	0	0	0	0	1

	1	2	3	4	5
1	0	1	1	1	0
2	1	0	1	1	0
3	1	1	0	1	0
4	1	1	1	0	1
5	0	0	0	0	1

Recognizing isomorphisms is of practical importance. Researchers working with organic compounds build up large dictionaries of compounds that they have previously analyzed. When a new compound is found, they want to know if it is already in the dictionary. These dictionaries can have many compounds of the same molecular formula, but differ in their structure as represented in graphs (and possibly in other ways as well). One of the things that a researcher must do is test the new compound to see if its graph structure is isomorphic to any in the dictionary. If it is, then one of the main tests for equivalence is passed.

A similar problem arises in designing efficient integrated circuits for an electrical network. If the design problem has already been solved for an isomorphic network, valuable savings in time and money are possible.

Graph Traversals

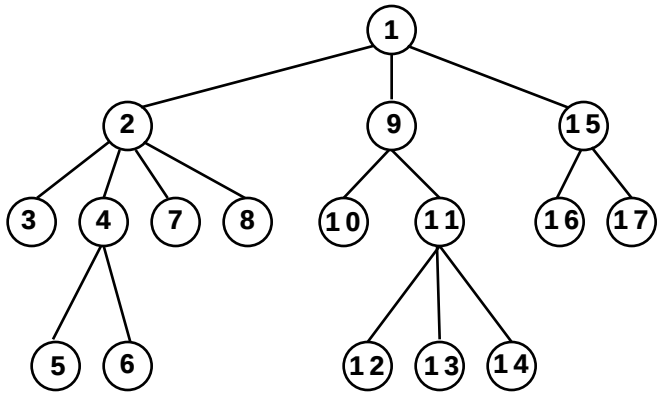
The problem of traversing a graph is fundamental to a large number of algorithms. We will consider the slightly easier problem of traversing a tree, and we will consider only two of the several strategies available: depth-first and breadth-first.

* Depth First Traversal:

A depth-first traversal from a given vertex v goes as deep into the graph as possible, and then backtracks to v and goes down another path. A depth first traversal requires a stack. Two pieces of information must be maintained in each record on the stack: 1) the node *n* currently being expanded, and 2) which of *n*'s children have yet to be visited. A sketch of an iterative algorithm using a stack is presented below.

Depth First Algorithm:

1) Push the pair <*root, root's children*> onto the stack
2) Let *t* = top of stack
3) If *t* is empty, then stop
4) Otherwise, print (or examine) *t*
5) If *t* has no unvisited children, then pop
6) else
   7) Let *c* = *t*'s first unvisited child
   8) Remove *c* from *t*'s list of unvisited children
   9) Push the pair <*c, c's children*> onto the stack
10) Goto 2.



Order nodes are visited during a depth first traversal

<1, {2,9,15}>
<2, {3,4,7,8}> <1, {9,15}>
<3, {}> <2, {4,7,8}> <1, {9, 15}>
<2, {4,7,8}> <1, {9,15}>
<4, {5,6}> <2, {7,8}> <1, {9,15}>
<5, {}> <4, {6}> <2, {7,8}> <1, {9,15}>
<4, {6}> <2, {7,8}> <1, {9, 15}>
<6, {}> <4, {}> <2, {7,8}> <1, {9,15}>
<4, {}> <2, {7,8}> <1, {9,15}>

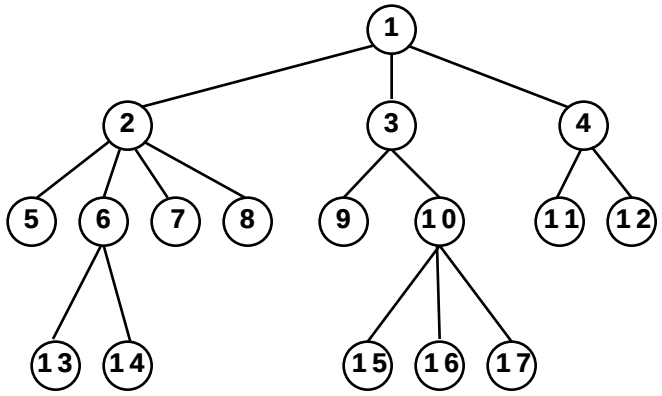
<2, {7,8}> <1, {9,15}>
<7, {}> <2, {8}> <1, {9,15}>
<2, {8}> <1, {9,15}>
<8, {}> <2, {}> <1, {9,15}>
<2, {}> <1, {9,15}>
<1, {9, 15}>
…

*** Breadth First Traversal:**

The breadth first algorithm visits the nodes of the tree in layers, as shown in the picture below. The key to breadth first searching is to use a queue to maintain a list of the subtrees that need to be explored.

Breadth First Algorithm:

1) Add the root of tree to queue
2) If the queue is empty, then stop
3) Otherwise, let *h* = head of the queue (remove h from the queue)
4) Print or examine *h*
5) Add pointers to the children of *h* to the tail of the queue
6) Goto 2



Order nodes are visited during a breadth first traversal

Node Visited	*Contents of Queue*
1	1
2	2 3 4
3	3 4 5 6 7 8
4	4 5 6 7 8 9 10
5	5 6 7 8 9 10 11 12
6	6 7 8 9 10 11 12
7	7 8 9 10 11 12 13 14
8	8 9 10 11 12 13 14

9	9 10 11 12 13 14
10	10 11 12 13 14
11	11 12 13 14 15 16 17
12	12 13 14 15 16 17
13	13 14 15 16 17
14	14 15 16 17
15	15 16 17
16	16
17	17
	poof!

Bibliography, and Additional Information

* For general information on graphs and graph algorithms, refer to the following textbooks. For more information on the basics presented above, refer to Aho, Ullman, pp. 435-442 & pp. 466-482; more information can be found in any discrete math textbook (Rosen or Epp).

A. Aho, J. Hopcroft, J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Reading, MA: Addison-Wesley, 1974.

A. Aho, J.D. Ullman, *Foundations of Computer Science*, New York: W.H. Freeman, 1992.

S. Epp, *Discrete Mathematics with Applications*, Belmont, CA: Wadsworth, 1990.

J.A. McHugh, *Algorithmic Graph Theory*, Englewood Cliffs, NJ: Prentice-Hall, 1990.

K. Rosen, *Discrete Mathematics and its Applications 2nd Ed.*, New York: McGraw-Hill, 1991.

R. Sedgewick, *Algorithms, 2nd Ed.*, Reading, MA: Addison-Wesley, 1988. (Note: the description of the stable marriage problem can be found here.)

A. Tucker, *Applied Combinatorics 2nd ed.*, New York: Wiley, 1985.

R.J. Wilson, *Introduction to Graph Theory, 3rd ed.*, Essex, England: Longman, 1985.

* For graph applications see:

G. Chartrand, *Graphs as Mathematical Models*, Boston: Prindle, Weber and Schmidt, 1977.

R.J. Wilson, L. Beineke, *Applications of Graph Theory,* London: Academic Press, 1979.

* The history of graph theory is covered in:

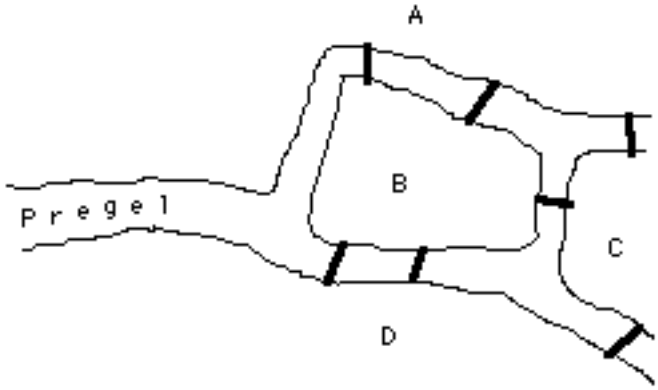
N. Biggs, E. Lloyd, R.J. Wilson, *Graph Theory 1736-1936*, Oxford: Clarendon, 1986.

Historical Notes

The subject of graph theory began in 1736 when the great mathematician Leonhard Euler published a paper that contained the solution to the following puzzle:

The town of Konigsberg in Prussia (now Kaliningrad in the USSR) is built at a point

where two branches of the Pregel River join. The town consists of an island and some land along the river banks. All land masses are connected by seven bridges:



Given this geographical situation, is it possible for a person to take a walk around town, starting and ending at the same location, and crossing each of the seven bridges exactly once?

This problem can be modeled and solved using graphs which represents the first time such a model was used. (We will solve this problem in a future class.) The term "graph" was first used by J. Sylvester in 1877, and the first book on graph theory, by D. Konig, did not appear until 1936.

Depth first search was first used to create efficient graph algorithms by Hopcroft and Tarjan in 1973 ("Efficient Algorithms for Graph Manipulations," *Communications of the ACM* 16:6, p. 372). Breadth first search was first introduced in the implementation of network optimization problems, e.g., minimal path problems and network flows.