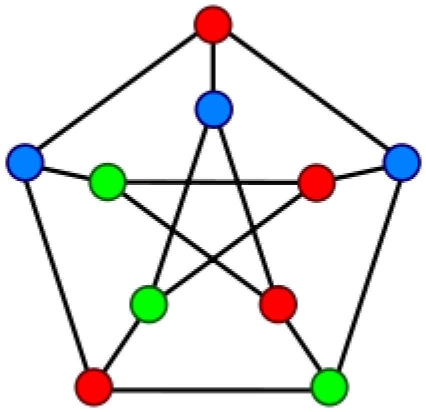
Graph Algorithms

Graphs are one of the unifying themes of computer science – an abstract representation which describes the organization of transportation systems, electrical circuits, human interactions, and telecommunication networks. That so many different structures can be modeled using a single formalism is a source of great power to the educated programmer.

In this tutorial, we focus on problems which require only an elementary knowledge of graph algorithms, specifically the appropriate use of graph data structures. Later on we will present graph traversal algorithms and problems relying on more advanced graph algorithms that find minimum spanning trees, shortest paths, and network flows.



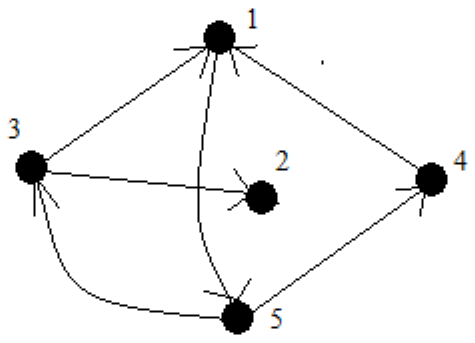
Flavor of Graphs

A graph G = (V,E) is defined by a set of vertices V , and a set of edges E consisting of ordered or unordered pairs of vertices from V . In modeling a road network, the vertices may represent the cities or junctions, certain pairs of which are connected by roads/edges. In analyzing the source code of a computer program, the vertices may represent lines of code, with an edge connecting lines x and y if y can be the next statement executed after x. In analyzing human interactions, the vertices typically represent people, with edges connecting pairs of related souls.

There are several fundamental properties of graphs which impact the choice of data structures used to represent them and algorithms available to analyze them. The first step in any graph problem is determining which flavor of graph you are dealing with -

• ***Undirected vs. Directed*** **:** A graph G = (V,E) is undirected if edge (x, y) ∈ E implies that (y, x) is also in E. If not, we say that the graph is directed. Road networks between cities are typically undirected, since any large road has lanes going in both directions. Street networks within cities are almost always directed, because there are typically at least a few one-way streets lurking about.

Program flow graphs are typically directed, because the execution flows from one line into the next and changes direction only at branches. Most graphs of graph-theoretic interest are undirected.



• ***Weighted vs. Unweighted*** **:** In weighted graphs, each edge (or vertex) of G is assigned a numerical value, or weight. Typical application-specific edge weights for road networks might be the distance, travel time, or maximum capacity between x and y. In unweighted graphs, there is no cost distinction between various edges and vertices.

The difference between weighted and unweighted graphs becomes particularly apparent in finding the shortest path between two vertices. For unweighted graphs, the shortest path must have the fewest number of edges, and can be found using the breadth-first search algorithm. Shortest paths in weighted graphs requires more sophisticated algorithms.

• ***Cyclic vs. Acyclic*** **:** An acyclic graph does not contain any cycles. Trees are connected acyclic undirected graphs. Trees are the simplest interesting graphs, and inherently recursive structures since cutting any edge leaves two smaller trees.

Directed acyclic graphs are called DAGs. They arise naturally in scheduling problems, where a directed edge (x, y) indicates that x must occur before y. An operation called topological sorting orders the vertices of a DAG so as to respect these precedence constraints. Topological sorting is typically the first step of any algorithm on a DAG.

• ***Simple vs. Non-simple*** **:** Certain types of edges complicate the task of working with graphs. A self-loop is an edge (x, x) involving only one vertex. An edge (x, y) is a multi-edge if it occurs more than once in the graph. Both of these structures require special care in implementing graph algorithms. Hence any graph which avoids them is called simple.

• ***Embedded vs. Topological*** **:** A graph is embedded if the vertices and edges have been assigned geometric positions. Thus any drawing of a graph is an embedding, which may or may not have algorithmic significance.

Occasionally, the structure of a graph is completely defined by the geometry of its embedding. For

example, if we are given a collection of points in the plane, and seek the minimum cost tour visiting all of them (i.e., the traveling salesman problem), the underlying topology is the complete graph connecting each pair of vertices. The weights are typically defined by the Euclidean distance between each pair of points.
Another example of topology from geometry arises in grids of points. Many problems on an n × m grid involve walking between neighboring points, so the edges are implicitly defined from the geometry.

• ***Implicit vs. Explicit*** **:** Many graphs are not explicitly constructed and then traversed, but built as we use them. A good example is in backtrack search. The vertices of this implicit search graph are the states of the search vector, while edges link pairs of states which can be directly generated from each other. It is often easier to work with an implicit graph than explicitly constructing it before analysis.

• ***Labeled vs. Unlabeled*** **:** In labeled graphs, each vertex is assigned a unique name or identifier to distinguish it from all other vertices. In unlabeled graphs, no such distinctions have been made.
Most graphs arising in applications are naturally and meaningfully labeled, such as city names in a transportation network. A common problem arising on graphs is that of isomorphism testing, determining whether the topological structure of two graphs are in fact identical if we ignore any labels. Such problems are typically solved using backtracking, by trying to assign each vertex in each graph a label such that the structures are identical.

Source:

http://www.learnalgorithms.in/#