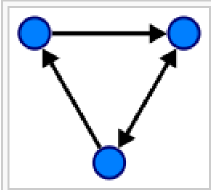
Graph (abstract data type)

From Wikipedia, the free encyclopedia
(Redirected from Graph (data structure))

This article **needs additional citations for verification**. Please help improve this article by adding citations to reliable sources. Unsourced material may be challenged and removed. *(October 2010)*


A graph with 3 nodes and 3 edges.

In computer science, a **graph** is an abstract data type that is meant to implement the graph and directed graph concepts from mathematics.

A graph data structure consists of a finite (and possibly mutable) set of **nodes** or **vertices**, together with a set of ordered pairs of these nodes (or, in some cases, a set of unordered pairs). These pairs are known as **edges** or **arcs**. As in mathematics, an edge (*x*,*y*) is said to **point** or **go from** *x* **to** *y*. The nodes may be part of the graph structure, or may be external entities represented by integer indices or references.

A graph data structure may also associate to each edge some **edge value**, such as a symbolic label or a numeric attribute (cost, capacity, length, etc.).

Contents [hide]

Algorithms [edit]

Graph algorithms are a significant field of interest within computer science. Typical higher-level operations associated with graphs are: finding a path between two nodes, like depth-first search and breadth-first search and finding the shortest path from one node to another, like Dijkstra's algorithm. A solution to finding the shortest path from each node to every other node also exists in the form of the Floyd–Warshall algorithm.

Operations [edit]

The basic operations provided by a graph data structure *G* usually include:

- `adjacent`(*G*, *x*, *y*): tests whether there is an edge from node *x* to node *y*.
- `neighbors`(*G*, *x*): lists all nodes *y* such that there is an edge from *x* to *y*.
- `add`(*G*, *x*, *y*): adds to *G* the edge from *x* to *y*, if it is not there.
- `delete`(*G*, *x*, *y*): removes the edge from *x* to *y*, if it is there.
- `get_node_value`(*G*, *x*): returns the value associated with the node *x*.
- `set_node_value`(*G*, *x*, *a*): sets the value associated with the node *x* to *a*.

Structures that associate values to the edges usually also provide:

- `get_edge_value`(*G*, *x*, *y*): returns the value associated to the edge (*x*,*y*).
- `set_edge_value`(*G*, *x*, *y*, *v*): sets the value associated to the edge (*x*,*y*) to *v*.

Representations [edit]

Different data structures for the representation of graphs are used in practice:

Adjacency list

Vertices are stored as records or objects, and every vertex stores a list of adjacent vertices. This data structure allows the storage of additional data on the vertices. Additional data can be stored if edges are also stored as objects, in which case each vertex stores its incident edges and each edge stores its incident vertices.

Adjacency matrix

A two-dimensional matrix, in which the rows represent source vertices and columns represent destination vertices. Data on edges and vertices must be stored externally. Only the cost for one edge can be stored between each pair of vertices.

Incidence matrix

A two-dimensional Boolean matrix, in which the rows represent the vertices and columns represent the edges. The entries indicate whether the vertex at a row is incident to the edge at a column.

The following table gives the time complexity cost of performing various operations on graphs, for each of these representations.[citation needed] In the matrix representations, the entries encode the cost of following an edge. The cost of edges that are not present are assumed to be ∞.

	Adjacency list	Adjacency matrix	Incidence matrix
Storage	$O(\|V\|+\|E\|)$	$O(\|V\|^2)$	$O(\|V\|\cdot\|E\|)$
Add vertex	$O(1)$	$O(\|V\|^2)$	$O(\|V\|\cdot\|E\|)$
Add edge	$O(1)$	$O(1)$	$O(\|V\|\cdot\|E\|)$
Remove vertex	$O(\|E\|)$	$O(\|V\|^2)$	$O(\|V\|\cdot\|E\|)$
Remove edge	$O(\|E\|)$	$O(1)$	$O(\|V\|\cdot\|E\|)$
Query: are vertices u, v adjacent? (Assuming that the storage positions for u, v are known)	$O(\|V\|)$	$O(1)$	$O(\|E\|)$
Remarks	When removing edges or vertices, need to find all vertices or edges	Slow for add/remove vertices, because matrix must be resized/copied	Slow to add or remove vertices and edges, because matrix must be resized/copied

Adjacency lists are generally preferred because they efficiently represent sparse graphs. An adjacency matrix is preferred if the graph is dense, that is the number of edges $|E|$ is close to the number of vertices squared, $|V|^2$, or if one must be able to quickly look up if there is an edge connecting two vertices.[1]

See also [edit]

- Graph traversal for graph walking strategies
- Graph database for graph (data structure) persistency
- Graph rewriting for rule based transformations of graphs (graph data structures)
- Graph drawing software for software, systems, and providers of systems for drawing graphs

References [edit]

1. ^ Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). Introduction to Algorithms (2nd ed.). MIT Press and McGraw–Hill. ISBN 0-262-53196-8.

External links [edit]

- Boost Graph Library: a powerful C++ graph library
- Networkx: a Python graph library
- Graphs Tutorial by Jebril FILALI

v · t · e	Data structures [hide]
Types	Collection · Container
Abstract	Associative array · Double-ended priority queue · Double-ended queue · List · Map · Multimap · Priority queue · Queue · Set (multiset) · Disjoint Sets · Stack
Arrays	Bit array · Circular buffer · Dynamic array · Hash table · Hashed array tree · Sparse array
Linked	Association list · Linked list · Skip list · Unrolled linked list · XOR linked list
Trees	B-tree · Binary search tree (AA · AVL · red-black · self-balancing · splay) · Heap (binary · binomial · Fibonacci) · R-tree (R* · R+ · Hilbert) · Trie (Hash tree)

Graphs	Binary decision diagram · Directed acyclic graph · Directed acyclic word graph
List of data structures	

Categories: Graph theory | Graph data structures | Abstract data types | Graphs | Hypergraphs