Graphs, Explained

Guava's `common.graph` is a library for modeling graph-structured data, that is, entities and the relationships between them. Examples include webpages and hyperlinks; scientists and the papers that they write; airports and the routes between them; and people and their family ties (family trees). Its purpose is to provide a common and extensible language for working with such data.

Definitions

A graph consists of a set of **nodes** (also called vertices) and a set of **edges** (also called links, or arcs); each edge connects nodes to each other. The nodes incident to an edge are called its **endpoints**.

(While we introduce an interface called `Graph` below, we will use "graph" (lower case "g") as a general term referring to this type of data structure. When we want to refer to a specific type in this library, we capitalize it.)

An edge is **directed** if it has a defined start (its **source**) and end (its **target**, also called its destination). Otherwise, it is **undirected**. Directed edges are suitable for modeling asymmetric relations ("descended from", "links to", "authored by"), while undirected edges are suitable for modeling symmetric relations ("coauthored a paper with", "distance between", "sibling of").

A graph is directed if each of its edges are directed, and undirected if each of its edges are undirected. (`common.graph` does not support graphs that have both directed and undirected edges.)

Given this example:

```
graph.addEdge(nodeU, nodeV, edgeUV);
```

- `nodeU` and `nodeV` are mutually **adjacent**
- `edgeUV` is **incident** to `nodeU` and to `nodeV` (and vice versa)

If `graph` is directed, then:

- `nodeU` is a **predecessor** of `nodeV`
- `nodeV` is a **successor** of `nodeU`
- `edgeUV` is an **outgoing** edge (or out-edge) of `nodeU`
- `edgeUV` is an **incoming** edge (or in-edge) of `nodeV`
- `nodeU` is a **source** of `edgeUV`
- `nodeV` is a **target** of `edgeUV`

If `graph` is undirected, then:

- `nodeU` is a predecessor and a successor of `nodeV`
- `nodeV` is a predecessor and a successor of `nodeU`
- `edgeUV` is both an incoming and an outgoing edge of `nodeU`
- `edgeUV` is both an incoming and an outgoing edge of `nodeV`

All of these relationships are with respect to `graph`.

A **self-loop** is an edge that connects a node to itself; equivalently, it is an edge whose endpoints are the same node. If a self-loop is directed, it is both an outgoing and incoming edge of its incident node, and its incident node is both a source and a target of the self-loop edge.

Two edges are **parallel** if they connect the same nodes in the same order (if any), and **antiparallel** if they connect the same nodes in the opposite order. (Undirected edges cannot be antiparallel.)

Given this example:

```
directedGraph.addEdge(nodeU, nodeV, edgeUV_a);
directedGraph.addEdge(nodeU, nodeV, edgeUV_b);
directedGraph.addEdge(nodeV, nodeU, edgeVU);

undirectedGraph.addEdge(nodeU, nodeV, edgeUV_a);
undirectedGraph.addEdge(nodeU, nodeV, edgeUV_b);
undirectedGraph.addEdge(nodeV, nodeU, edgeVU);
```

In `directedGraph`, `edgeUV_a` and `edgeUV_b` are mutually parallel, and each is antiparallel with `edgeVU`.

In `undirectedGraph`, each of `edgeUV_a`, `edgeUV_b`, and `edgeVU` is mutually parallel with the other two.

Capabilities

`common.graph` is focused on providing interfaces and classes to support working with graphs. It does not provide functionality such as I/O or visualization support, and it has a very limited selection of utilities. See the FAQ for more on this topic.

As a whole, `common.graph` supports graphs of the following varieties:

- directed graphs
- undirected graphs
- nodes and/or edges with associated values (weights, labels, etc.)
- graphs that do/don't allow self-loops
- graphs that do/don't allow parallel edges (graphs with parallel edges are sometimes called multigraphs)
- graphs whose nodes/edges are insertion-ordered, sorted, or unordered

The kinds of graphs supported by a particular `common.graph` type are specified in its Javadoc. The kinds of graphs supported by the built-in implementations of each graph type are specified in the Javadoc for its associated `Builder` type. Specific *implementations* of the types in this library (especially third-party implementations) are not required to support all of these varieties, and may support others in addition.

The library is agnostic as to the choice of underlying data structures: relationships can be stored as matrices, adjacency lists, adjacency maps, etc. depending on what use cases the implementor wants to optimize for.

`common.graph` does not (at this time) include *explicit* support for the following graph variants, although they can be modeled using the existing types:

- trees, forests
- graphs with elements of the same kind (nodes or edges) that have different types (for example: bipartite/k-partite graphs, multimodal graphs)
- hypergraphs

`common.graph` does not allow graphs with both directed and undirected edges.

The `Graphs` class provides some basic utilities (for example, copying and comparing graphs).

Graph Types

There are three top-level graph interfaces, that are distinguished by their representation of edges: `Graph`, `ValueGraph`, and `Network`. These are sibling types, i.e., none is a subtype of any of the others.

Each of these "top-level" interfaces extends `SuccessorsFunction` and `PredecessorsFunction`. These interfaces are meant to be used as the type of a parameter to graph algorithms (such as breadth first traversal) that only need a way of accessing the successors/predecessors of a node in a graph. This is especially useful in cases where the owner of a graph already has a representation that works for them and doesn't particularly want to serialize their representation into a `common.graph` type just to run one graph algorithm.

Graph

`Graph` is the simplest and most fundamental graph type. It defines the low-level operators for dealing with node-to-node relationships, such as `successors(node)`, `adjacentNodes(node)`, and `inDegree(node)`. Its nodes are first-class unique objects; you can think of them as analogous to `Map` keys into the `Graph` internal data structures.

The edges of a `Graph` are completely anonymous; they are defined only in terms of their endpoints.

Example use case: `Graph<Airport>`, whose edges connect the airports between which one can take a direct flight.

ValueGraph

`ValueGraph` has all the node-related methods that `Graph` does, but adds a couple of methods that retrieve a value for a specified edge.

The edges of a `ValueGraph` each have an associated user-specified value. These values need not be unique (as nodes are); the relationship between a `ValueGraph` and a `Graph` is analogous to that between a `Map` and a `Set`; a `Graph`'s edges are a set of pairs of endpoints, and a `ValueGraph`'s edges are a map from pairs of endpoints to values.)

`ValueGraph` provides an `asGraph()` method which returns a `Graph` view of the `ValueGraph`. This allows methods which operate on `Graph` instances to function for `ValueGraph` instances as well.

Example use case: `ValueGraph<Airport, Integer>`, whose edges values represent the time required to travel between the two `Airports` that the edge connects.

Network

`Network` has all the node-related methods that `Graph` does, but adds methods that work with edges and node-to-edge relationships, such as `outEdges(node)`, `incidentNodes(edge)`, and `edgesConnecting(nodeU, nodeV)`.

The edges of a `Network` are first-class (unique) objects, just as nodes are in all graph types. The uniqueness constraint for edges allows `Network` to natively support parallel edges, as well as the methods relating to edges and node-to-edge relationships.

`Network` provides an `asGraph()` method which returns a `Graph` view of the `Network`. This allows methods which operate on `Graph` instances to function for `Network` instances as well.

Example use case: `Network<Airport, Flight>`, in which the edges represent the specific flights that one can take to get from one airport to another.

Choosing the right graph type

The essential distinction between the three graph types is in their representation of edges.

`Graph` has edges which are anonymous connections between nodes, with no identity or properties of their own. You should use `Graph` if each pair of nodes is connected by at most one edge, and you don't need to associate any information with edges.

`ValueGraph` has edges which have values (e.g., edge weights or labels) that may or may not be unique. You should use `ValueGraph` if each pair of nodes is connected by at most one edge, and you need to associate information with edges that may be the same for different edges (for example, edge weights).

`Network` has edges which are first-class unique objects, just as nodes are. You should use `Network` if your edge objects are unique, and you want to be able to issue queries that reference them. (Note that this uniqueness allows `Network` to support parallel edges.)

Building graph instances

The implementation classes that `common.graph` provides are not public, by design. This reduces the number of public types that users need to know about, and makes it easier to navigate the various capabilities that the built-implementations provide, without overwhelming users that just want to create a graph.

To create an instance of one of the built-in implementations of a graph type, use the corresponding `Builder` class: `GraphBuilder`, `ValueGraphBuilder`, or `NetworkBuilder`. Examples:

```
// Creating mutable graphs
MutableGraph<Integer> graph = GraphBuilder.undirected().build();

MutableValueGraph<City, Distance> roads = ValueGraphBuilder.directed()
    .incidentEdgeOrder(ElementOrder.stable())
    .build();

MutableNetwork<Webpage, Link> webSnapshot = NetworkBuilder.directed()
    .allowsParallelEdges(true)
    .nodeOrder(ElementOrder.natural())
    .expectedNodeCount(100000)
    .expectedEdgeCount(1000000)
    .build();

// Creating an immutable graph
ImmutableGraph<Country> countryAdjacencyGraph =
    GraphBuilder.undirected()
        .<Country>immutable()
        .putEdge(FRANCE, GERMANY)
        .putEdge(FRANCE, BELGIUM)
        .putEdge(GERMANY, BELGIUM)
        .addNode(ICELAND)
        .build();
```

- You can get an instance of a graph `Builder` in one of two ways:
  - calling the static methods `directed()` or `undirected()`. Each Graph instance that the `Builder` provides will be directed or undirected.
  - calling the static method `from()`, which gives you a `Builder` based on an existing graph instance.
- After you've created your `Builder` instance, you can optionally specify other characteristics and capabilities.
- Building mutable graphs
  - You can call `build()` on the same `Builder` instance multiple times

    to build multiple graph instances with the same configuration.
  - You don't need to specify the element type(s) on the `Builder`; specifying them on the graph type itself is sufficient.
  - The `build()` method returns a `Mutable` subtype of the associated graph type, which provides mutation methods; more on this in "`Mutable` and `Immutable` graphs", below.
- Building immutable graphs
  - You can call `immmutable()` on the same `Builder` instance multiple times to create multiple `ImmutableGraph.Builder` instances with the same configuration.
  - You need to specify the element type(s) on the `immutable` call.

Builder constraints vs. optimization hints

The `Builder` types generally provide two types of options: constraints and optimization hints.

Constraints specify behaviors and properties that graphs created by a given `Builder` instance must satisfy, such as:

- whether the graph is directed
- whether this graph allows self-loops
- whether this graph's edges are sorted

and so forth.

Optimization hints may optionally be used by the implementation class to increase efficiency, for example, to determine the type or initial size of internal data structures. They are not guaranteed to have any effect.

Each graph type provides accessors corresponding to its `Builder`-specified constraints, but does not provide accessors for optimization hints.

`Mutable` and `Immutable` graphs

`Mutable*` types

Each graph type has a corresponding `Mutable*` subtype: `MutableGraph`, `MutableValueGraph`, and `MutableNetwork`. These subtypes define the mutation methods:

- methods for adding and removing nodes:
  - `addNode(node)` and `removeNode(node)`
- methods for adding and removing edges:
  - `MutableGraph`
    - `putEdge(nodeU, nodeV)`
    - `removeEdge(nodeU, nodeV)`
  - `MutableValueGraph`
    - `putEdgeValue(nodeU, nodeV, value)`

        * `removeEdge(nodeU, nodeV)`
    - `MutableNetwork`
        * `addEdge(nodeU, nodeV, edge)`
        * `removeEdge(edge)`

This is a departure from the way that the Java Collections Framework–and Guava's new collection types–have historically worked; each of those types includes signatures for (optional) mutation methods. We chose to break out the mutable methods into subtypes in part to encourage defensive programming: generally speaking, if your code only examines or traverses a graph and does not mutate it, its input should be specified as on `Graph`, `ValueGraph`, or `Network` rather than their mutable subtypes. On the other hand, if your code does need to mutate an object, it's helpful for your code to have to call attention to that fact by working with a type that labels itself "Mutable".

Since `Graph`, etc. are interfaces, even though they don't include mutation methods, providing an instance of this interface to a caller *does not guarantee* that it will not be mutated by the caller, as (if it is in fact a `Mutable*` subtype) the caller could cast it to that subtype. If you want to provide a contractual guarantee that a graph which is a method parameter or return value cannot be modified, you should use the `Immutable` implementations; more on this below.

`Immutable*` implementations

Each graph type also has a corresponding `Immutable` implementation. These classes are analogous to Guava's `ImmutableSet`, `ImmutableList`, `ImmutableMap`, etc.: once constructed, they cannot be modified, and they use efficient immutable data structures internally.

Unlike the other Guava `Immutable` types, however, these implementations do not have any method signatures for mutation methods, so they don't need to throw `UnsupportedOperationException` for attempted mutates.

You create an instance of an `ImmutableGraph`, etc. in one of two ways:

Using `GraphBuilder`:

```
ImmutableGraph<Country> immutableGraph1 =
    GraphBuilder.undirected()
        .<Country>immutable()
        .putEdge(FRANCE, GERMANY)
        .putEdge(FRANCE, BELGIUM)
        .putEdge(GERMANY, BELGIUM)
        .addNode(ICELAND)
        .build();
```

Using `ImmutableGraph.copyOf()`:

```
ImmutableGraph<Integer> immutableGraph2 = ImmutableGraph.copyOf(otherGraph);
```

Immutable graphs are always guaranteed to provide a stable incident edge order. If the graph is populated using `GraphBuilder`, then the incident edge order will be insertion order where possible (see `ElementOrder.stable()` for more info). When using `copyOf`, then the incident edge order will be the order in which they are visited during the copy process.

Guarantees Each `Immutable*` type makes the following guarantees:

- **shallow immutability**: elements can never be added, removed or replaced (these classes do not implement the `Mutable*` interfaces)
- **deterministic iteration**: the iteration orders are always the same as those of the input graph
- **thread safety**: it is safe to access this graph concurrently from multiple threads
- **integrity**: this type cannot be subclassed outside this package (which would allow these guarantees to be violated)

Treat these classes as "interfaces", not implementations Each of the `Immutable*` classes is a type offering meaningful behavioral guarantees – not merely a specific implementation. You should treat them as interfaces in every important sense of the word.

Fields and method return values that store an `Immutable*` instance (such as `ImmutableGraph`) should be declared to be of the `Immutable*` type rather than the corresponding interface type (such as `Graph`). This communicates to callers all of the semantic guarantees listed above, which is almost always very useful information.

On the other hand, a parameter type of `ImmutableGraph` is generally a nuisance to callers. Instead, accept `Graph`.

Warning: as noted elsewhere, it is almost always a bad idea to modify an element (in a way that affects its `equals()` behavior) while it is contained in a collection. Undefined behavior and bugs will result. It's best to avoid using mutable objects as elements of an `Immutable*` instance at all, as users may expect your "immutable" object to be deeply immutable.

Graph elements (nodes and edges)

Elements must be useable as `Map` keys

The graph elements provided by the user should be thought of as keys into the internal data structures maintained by the graph implementations. Thus, the classes used to represent graph elements must have `equals()` and `hashCode()` implementations that have, or induce, the properties listed below.

Uniqueness If `A` and `B` satisfy `A.equals(B) == true` then at most one of the two may be an element of the graph.

Consistency between `hashCode()` and `equals()` `hashCode()` must be consistent with `equals()` as defined by `Object.hashCode()`.

Ordering consistency with `equals()` If the nodes are sorted (for example, via `GraphBuilder.orderNodes()`), the ordering must be consistent with `equals()`, as defined by `Comparator` and `Comparable`.

Non-recursiveness `hashCode` and `equals()` *must not* recursively reference other elements, as in this example:

```
// DON'T use a class like this as a graph element (or Map key/Set element)
public final class Node<T> {
  T value;
  Set<Node<T>> successors;

  public boolean equals(Object o) {
    Node<T> other = (Node<T>) o;
    return Objects.equals(value, other.value)
        && Objects.equals(successors, other.successors);
  }

  public int hashCode() {
    return Objects.hash(value, successors);
  }
}
```

Using such a class as a `common.graph` element type (e.g., `Graph<Node<T>>`) has these problems:

- **redundancy**: the implementations of `Graph` provided by the `common.graph` library already store these relationships
- **inefficiency**: adding/accessing such elements calls `equals()` (and possibly `hashCode()`), which require O(n) time
- **infeasibility**: if there are cycles in the graph, `equals()` and `hashCode()` may never terminate

Instead, just use the `T` value itself as the node type (assuming that the `T` values are themselves valid `Map` keys).

Elements and mutable state

If graph elements have mutable state:

- the mutable state must not be reflected in the `equals()/hashCode()` methods (this is discussed in the `Map` documentation in detail)
- don't construct multiple elements that are equal to each other and expect them to be interchangeable. In particular, when adding such elements to a graph, you should create them once and store the reference if you will

need to refer to those elements more than once during creation (rather than passing `new MyMutableNode(id)` to each `add*()` call).

If you need to store mutable per-element state, one option is to use immutable elements and store the mutable state in a separate data structure (e.g. an element-to-state map).

Elements must be non-null

The methods that add elements to graphs are contractually required to reject null elements.

Library contracts and behaviors

This section discusses behaviors of the built-in implementations of the `common.graph` types.

Mutation

You can add an edge whose incident nodes have not previously been added to the graph. If they're not already present, they're silently added to the graph:

```
Graph<Integer> graph = GraphBuilder.directed().build();  // graph is empty
graph.putEdge(1, 2);  // this adds 1 and 2 as nodes of this graph, and puts
                      // an edge between them
if (graph.nodes().contains(1)) {  // evaluates to "true"
  ...
}
```

Graph `equals()` and graph equivalence

As of Guava 22, `common.graph`'s graph types each define `equals()` in a way that makes sense for the particular type:

- `Graph.equals()` defines two `Graph`s to be equal if they have the same node and edge sets (that is, each edge has the same endpoints and same direction in both graphs).
- `ValueGraph.equals()` defines two `ValueGraph`s to be equal if they have the same node and edge sets, and equal edges have equal values.
- `Network.equals()` defines two `Network`s to be equal if they have the same node and edge sets, and each edge object has connects the same nodes in the same direction (if any).

In addition, for each graph type, two graphs can be equal only if their edges have the same directedness (both graphs are directed or both are undirected).

Of course, `hashCode()` is defined consistently with `equals()` for each graph type.

If you want to compare two `Network`s or two `ValueGraph`s based only on connectivity, or to compare a `Network` or a `ValueGraph` to a `Graph`, you can use the `Graph` view that `Network` and `ValueGraph` provide:

```
Graph<Integer> graph1, graph2;
ValueGraph<Integer, Double> valueGraph1, valueGraph2;
Network<Integer, MyEdge> network1, network2;

// compare based on nodes and node relationships only
if (graph1.equals(graph2)) { ... }
if (valueGraph1.asGraph().equals(valueGraph2.asGraph())) { ... }
if (network1.asGraph().equals(graph1.asGraph())) { ... }

// compare based on nodes, node relationships, and edge values
if (valueGraph1.equals(valueGraph2)) { ... }

// compare based on nodes, node relationships, and edge identities
if (network1.equals(network2)) { ... }
```

Accessor methods

Accessors which return collections:

- may return views of the graph; modifications to the graph which affect a view (for example, calling `addNode(n)` or `removeNode(n)` while iterating through `nodes()`) are not supported and may result in throwing `ConcurrentModificationException`.
- will return empty collections if their inputs are valid but no elements satisfy the request (for example: `adjacentNodes(node)` will return an empty collection if `node` has no adjacent nodes).

Accessors will throw `IllegalArgumentException` if passed an element that is not in the graph.

While some Java Collection Framework methods such as `contains()` take `Object` parameters rather than the appropriate generic type specifier, as of Guava 22, the `common.graph` methods take the generic type specifiers to improve type safety.

Synchronization

It is up to each graph implementation to determine its own synchronization policy. By default, undefined behavior may result from the invocation of any method on a graph that is being mutated by another thread.

Generally speaking, the built-in mutable implementations provide no synchronization guarantees, but the `Immutable*` classes (by virtue of being immutable) are thread-safe.

Element objects

The node, edge, and value objects that you add to your graphs are irrelevant to the built-in implementations; they're just used as keys to internal data structures. This means that nodes/edges may be shared among graph instances.

By default, node and edge objects are insertion-ordered (that is, are visited by the `Iterator`s for `nodes()` and `edges()` in the order in which they were added to the graph, as with `LinkedHashSet`).

Notes for implementors

Storage models

`common.graph` supports multiple mechanisms for storing the topology of a graph, including:

- the graph implementation stores the topology (for example, by storing a `Map<N, Set<N>>` that maps nodes onto their adjacent nodes); this implies that the nodes are just keys, and can be shared among graphs
- the nodes store the topology (for example, by storing a `List<E>` of adjacent nodes); this (usually) implies that nodes are graph-specific
- a separate data repository (for example, a database) stores the topology

Note: `Multimap`s are not sufficient internal data structures for Graph implementations that support isolated nodes (nodes that have no incident edges), due to their restriction that a key either maps to at least one value, or is not present in the `Multimap`.

Accessor behavior

For accessors that return a collection, there are several options for the semantics, including:

1. Collection is an immutable copy (e.g. `ImmutableSet`): attempts to modify the collection in any way will throw an exception, and modifications to the graph will **not** be reflected in the collection.
2. Collection is an unmodifiable view (e.g. `Collections.unmodifiableSet()`): attempts to modify the collection in any way will throw an exception, and modifications to the graph will be reflected in the collection.
3. Collection is a mutable copy: it may be modified, but modifications to the collection will **not** be reflected in the graph, and vice versa.
4. Collection is a modifiable view: it may be modified, and modifications to the collection will be reflected in the graph, and vice versa.

(In theory one could return a collection which passes through writes in one direction but not the other (collection-to-graph or vice-versa), but this is basically never going to be useful or clear, so please don't. :))

(1) and (2) are generally preferred; as of this writing, the built-in implementations generally use (2).

(2) is a workable option, but may be confusing to users if they expect that modifications will affect the graph, or that modifications to the graph would be reflected in the set.

(3) is a hazardous design choice and should be used only with extreme caution, because keeping the internal data structures consistent can be tricky.

`Abstract*` classes

Each graph type has a corresponding `Abstract` class: `AbstractGraph`, etc.

Implementors of the graph interfaces should, if possible, extend the appropriate abstract class rather than implementing the interface directly. The abstract classes provide implementations of several key methods that can be tricky to do correctly, or for which it's helpful to have consistent implementations, such as:

- `*degree()`
- `toString()`
- `Graph.edges()`
- `Network.asGraph()`

Code examples

Is `node` in the graph?

```
graph.nodes().contains(node);
```

Is there an edge between nodes `u` and `v` (that are known to be in the graph)?

In the case where the graph is undirected, the ordering of the arguments `u` and `v` in the examples below is irrelevant.

```
// This is the preferred syntax since 23.0 for all graph types.
graphs.hasEdgeConnecting(u, v);

// These are equivalent (to each other and to the above expression).
graph.successors(u).contains(v);
graph.predecessors(v).contains(u);

// This is equivalent to the expressions above if the graph is undirected.
graph.adjacentNodes(u).contains(v);

// This works only for Networks.
!network.edgesConnecting(u, v).isEmpty();
```

```
// This works only if "network" has at most a single edge connecting u to v.
network.edgeConnecting(u, v).isPresent();  // Java 8 only
network.edgeConnectingOrNull(u, v) != null;

// These work only for ValueGraphs.
valueGraph.edgeValue(u, v).isPresent();  // Java 8 only
valueGraph.edgeValueOrDefault(u, v, null) != null;
```

Basic `Graph` example

```
ImmutableGraph<Integer> graph =
    GraphBuilder.directed()
        .<Integer>immutable()
        .addNode(1)
        .putEdge(2, 3) // also adds nodes 2 and 3 if not already present
        .putEdge(2, 3) // no effect; Graph does not support parallel edges
        .build();

Set<Integer> successorsOfTwo = graph.successors(2); // returns {3}
```

Basic `ValueGraph` example

```
MutableValueGraph<Integer, Double> weightedGraph = ValueGraphBuilder.directed().build();
weightedGraph.addNode(1);
weightedGraph.putEdgeValue(2, 3, 1.5);  // also adds nodes 2 and 3 if not already present
weightedGraph.putEdgeValue(3, 5, 1.5);  // edge values (like Map values) need not be unique
...
weightedGraph.putEdgeValue(2, 3, 2.0);  // updates the value for (2,3) to 2.0
```

Basic `Network` example

```
MutableNetwork<Integer, String> network = NetworkBuilder.directed().build();
network.addNode(1);
network.addEdge("2->3", 2, 3);  // also adds nodes 2 and 3 if not already present

Set<Integer> successorsOfTwo = network.successors(2);  // returns {3}
Set<String> outEdgesOfTwo = network.outEdges(2);   // returns {"2->3"}

network.addEdge("2->3 too", 2, 3);  // throws; Network disallows parallel edges
                                    // by default
network.addEdge("2->3", 2, 3);  // no effect; this edge is already present
                                // and connecting these nodes in this order

Set<String> inEdgesOfFour = network.inEdges(4); // throws; node not in graph
```

Traversing an undirected graph node-wise

```
// Return all nodes reachable by traversing 2 edges starting from "node"
// (ignoring edge direction and edge weights, if any, and not including "node").
Set<N> getTwoHopNeighbors(Graph<N> graph, N node) {
  Set<N> twoHopNeighbors = new HashSet<>();
  for (N neighbor : graph.adjacentNodes(node)) {
    twoHopNeighbors.addAll(graph.adjacentNodes(neighbor));
  }
  twoHopNeighbors.remove(node);
  return twoHopNeighbors;
}
```

Traversing a directed graph edge-wise

```
// Update the shortest-path weighted distances of the successors to "node"
// in a directed Network (inner loop of Dijkstra's algorithm)
// given a known distance for {@code node} stored in a {@code Map<N, Double>},
// and a {@code Function<E, Double>} for retrieving a weight for an edge.
void updateDistancesFrom(Network<N, E> network, N node) {
  double nodeDistance = distances.get(node);
  for (E outEdge : network.outEdges(node)) {
    N target = network.target(outEdge);
    double targetDistance = nodeDistance + edgeWeights.apply(outEdge);
    if (targetDistance < distances.getOrDefault(target, Double.MAX_VALUE)) {
      distances.put(target, targetDistance);
    }
  }
}
```

FAQ

Why did Guava introduce `common.graph`?

The same arguments apply to graphs as to many other things that Guava does:

- code reuse/interoperability/unification of paradigms: lots of things relate to graph processing
- efficiency: how much code is using inefficient graph representations? too much space (e.g. matrix representations)?
- correctness: how much code is doing graph analysis wrong?
- promotion of use of graph as ADT: how many people would be working with graphs if it were easy?
- simplicity: code which deals with graphs is easier to understand if it's explicitly using that metaphor.

What kinds of graphs does `common.graph` support?

This is answered in the "Capabilities" section, above.

`common.graph` doesn't have feature/algorithm X, can you add it?

Maybe. You can email us at `guava-discuss@googlegroups.com` or open an issue on GitHub.

Our philosophy is that something should only be part of Guava if (a) it fits in with Guava's core mission and (b) there is good reason to expect that it will be reasonably widely used.

`common.graph` will probably never have capabilities like visualization and I/O; those would be projects unto themselves and don't fit well with Guava's mission.

Capabilities like traversal, filtering, or transformation are better fits, and thus more likely to be included, although ultimately we expect that other graph libraries will provide most capabilities.

Does it support very large graphs (i.e., MapReduce scale)?

Not at this time. Graphs in the low millions of nodes should be workable, but you should think of this library as analogous to the Java Collections Framework types (`Map`, `List`, `Set`, and so on).

How can I define the order of `successors(node)`?

Setting `incidentEdgeOrder()` to `ElementOrder.stable()` in the graph builder makes sure that `successors(node)` returns the successors of `node` in the order that the edges were inserted. This is also true for most other methods that relate to the incident edges of a node, such as (such as `incidentEdges(node)`).

Why should I use it instead of something else?

tl;dr: you should use whatever works for you, but please let us know what you need if this library doesn't support it!

The main competitors to this library (for Java) are: JUNG and JGraphT.

`JUNG` was co-created by Joshua O'Madadhain (the `common.graph` lead) in 2003, and he still maintains it. JUNG is fairly mature and full-featured and is widely used, but has a lot of cruft and inefficiencies. Now that `common.graph` has been released externally, he is working on a new version of `JUNG` which uses `common.graph` for its data model.

`JGraphT` is another third-party Java graph library that's been around for a while. We're not as familiar with it, so we can't comment on it in detail, but it has at least some things in common with `JUNG`. This library also includes a number of adapter classes to adapt `common.graph` graphs into `JGraphT` graphs.

Rolling your own solution is sometimes the right answer if you have very specific requirements. But just as you wouldn't normally implement your own hash table in Java (instead of using `HashMap` or `ImmutableMap`), you should consider using `common.graph` (or, if necessary, another existing graph library) for all the reasons listed above.

Major Contributors

`common.graph` has been a team effort, and we've had help from a number of people both inside and outside Google, but these are the people that have had the greatest impact.

- **Omar Darwish** did a lot of the early implementations, and set the standard for the test coverage.
- **James Sexton** has been the single most prolific contributor to the project and has had a significant influence on its direction and its designs. He's responsible for some of the key features, and for the efficiency of the implementations that we provide.
- **Joshua O'Madadhain** started the `common.graph` project after reflecting on the strengths and weaknesses of JUNG, which he also helped to create. He leads the project, and has reviewed or written virtually every aspect of the design and the code.
- **Jens Nyman** contributed many of the more recent additions such as `Traverser` and immutable graph builders. He also has a major influence on the future direction of the project.