Graphs.jl Documentation

Release 0.3

Dahua Lin

September 11, 2015

Contents

Graphs.jl is a Julia package that provides graph types and algorithms. The design of this package is inspired by the Boost Graph Library (*e.g.* using standardized generic interfaces), while taking advantage of Julia's language features (*e.g.* multiple dispatch).

CHAPTER 1

Contents

1.1 Main Features

An important aspect of *Graphs.jl* is the generic abstraction of graph concepts expressed via standardized interfaces, which allows access to a graph's structure while hiding the implementation details. This encourages reuse of data structures and algorithms. In particular, one can write generic graph algorithms that can be applied to different graph types as long as they implement the required interface.

In addition to the generic abstraction, there are other important features:

- **A variety of graph types tailored to different purposes**
  - generic adjacency list
  - generic incidence list
  - a simple graph type with compact and efficient representation
  - an extended graph type that supports labels and attributes
- **A collection of graph algorithms:**
  - graph traversal with visitor support: BFS, DFS
  - cycle detection
  - connected components
  - topological sorting
  - shortest paths: Dijkstra, Floyd-Warshall
  - minimum spanning trees: Prim, Kruskal
  - more algorithms are being implemented
- Matrix-based characterization: adjacency matrix, weight matrix, Laplacian matrix
- All data structures and algorithms are implemented in *pure Julia*, and thus they are portable.
- We paid special attention to the runtime performance. Many of the algorithms are very efficient. For example, a benchmark shows that it takes about *15 milliseconds* to run the Dijkstra's algorithm over a graph with *10 thousand* vertices and *1 million* edges on a macbook pro.

1.2 Generic Interfaces

In *Graphs.jl*, the graph concepts are abstracted into a set of generic interfaces. Below is a detailed specification. We *strongly* recommend reading through this specification before you write a graph type to ensure that your graph type conforms to the interface system.

1.2.1 Basic interface

All graph types should be declared as a sub-type of `AbstractGraph{V,E}`, where `V` is the vertex type, and `E` is the edge type.

The following two functions are provided for graphs of all types.

`vertex_type`(*g*)
: returns the vertex type of a graph, *i.e.*, `V`.

`edge_type`(*g*)
: returns the edge type of a graph, *i.e.*, `E`.

Note: The two basic functions above have been implemented over `AbstractGraph` and one need not implement them for specific graph types.

In addition, the following method needs to be implemented for each graph type:

`is_directed`(*g*)
: returns whether `g` is a directed graph.

1.2.2 Vertex List interface

`num_vertices`(*g*)
: returns the number of vertices contained in `g`.

`vertices`(*g*)
: returns an iterable view/container of all vertices.

1.2.3 Edge List interface

`num_edges`(*g*)
: returns the number of edges contained in `g`.

`edges`(*g*)
: returns an iterable view/container of all edges.

`source`(*e*, *g*)
: returns the source vertex of an edge `e` in graph `g`.

`target`(*e*, *g*)
: returns the target vertex of an edge `e` in graph `g`.

1.2.4 Vertex Map interface

`vertex_index`(*v*, *g*)
: returns the index of a vertex `v` in graph `g`. Each vertex must have a unique index between 1 and `num_vertices`.

1.2.5 Edge Map interface

edge_index(*e*, *g*)
returns the index of an edge `e` in graph `g`. Each edge must have a unique index between 1 and `num_edges`.

1.2.6 Adjacency List interface

out_degree(*v*, *g*)
returns the number of outgoing neighbors from vertex `v` in graph `g`.

out_neighbors(*v*, *g*)
returns an iterable view/container of all outgoing neighbors of vertex `v` in graph `g`.

The following example prints all vertices of a graph as well as its neighbors

```
for u in vertices(g)
    print("$u: ")
    for v in out_neighbors(u, g)
        println("$v ")
    end
    println()
end
```

1.2.7 Incidence List interface

out_degree(*v*, *g*)
returns the number of outgoing edges from vertex `v` in graph `g`.

out_edges(*v*, *g*)
returns an iterable view/container of outgoing edges from vertex `v` in graph `g`.

source(*e*, *g*)
returns the source vertex of an edge `e` in graph `g`.

target(*e*, *g*)
returns the target vertex of an edge `e` in graph `g`.

The following example prints all vertices of a graph as well as its incidence edges

```
for u in vertices(g)
    print("$u: ")
    for e in out_edges(u, g)
        v = target(e, g)
        println("($u -- $v) ")
    end
    println()
end
```

1.2.8 Bidirectional Incidence List interface

This interface refines the `Incidence List` and requires the implementation of two additional methods:

in_degree(*v*, *g*)
returns the number of incoming edges to vertex `v` in graph `g`.

in_edges(*v*, *g*)
returns an iterable view/container of the incoming edges to vertex `v` in graph `g`.

1.2.9 Interface declaration and verification

It is important to note that a specific graph type can implement multiple interfaces. If a method is required to be implemented for two interfaces (*e.g.*, `out_degree` in both adjacency list an incidence list), this method need only be implemented once.

Julia does not provide a builtin mechanism for interface declaration. To declare that a specific graph type implements certain interfaces, one can use the macro `@graph_implements`. For example, to declare that a graph type `G` implements vertex list and adjacency list, one can write:

```
@graph_implements G vertex_list adjacency_list
```

This statement instantiates the following methods:

```
implements_vertex_list(::G) = true
implements_adjacency_list(::G) = true
```

The following is a list of supported interface names

- `vertex_list`
- `edge_list`
- `vertex_map`
- `edge_map`
- `adjacency_list`
- `incidence_list`
- `bidirectional_adjacency_list`
- `bidirectional_incidence_list`

In a function that implements a generic graph algorithm, one can use the macro `@graph_requires` to verify whether the input graph implements the required interfaces. A typical graph algorithm function may look like follows

```
function myfunc(g::AbstractGraph, params)
    @graph_requires g vertex_list adjacency_list
    ...
end
```

Here, the `@graph_requires` statement checks whether the graph `g` implements the interfaces for `vertex_list` and `adjacency_list`, and throws exceptions if `g` does not satisfy the requirements.

1.3 Vertices and Edges

1.3.1 Vertex Types

A vertex can be of any Julia type. For example, it can be an integer, a character, or a string.

This package provides two specific vertex types: `KeyVertex` and `ExVertex`. The definition of `KeyVertex` is:

```
immutable KeyVertex{K}
    index::Int
    key::K
end
```

Here, each vertex has a unique index and a key value of a user-chosen type (*e.g.* a string).

The definition of `ExVertex` is:

```
type ExVertex
    index::Int
    label::UTF8String
    attributes::Dict{UTF8String,Any}
end
```

The `ExVertex` type allows one to attach a label as well as other attributes to a vertex. The constructor of this type takes an index and a label string as arguments. The following code shows how one can create an instance of `ExVertex` and attach a price to it.

```
v = ExVertex(1, "vertex-a")
v.attributes["price"] = 100.0
```

The `ExVertex` type implements a `vertex_index` function, as

vertex_index(*v*)
: returns the index of the vertex `v`.

`SimpleGraph` is a special case where the vertices are of type `Int` and store both their index and identity. In all other graphs, `Int` vertices are unordered indices.

1.3.2 Edge Types

This package provides two edge types: `Edge` and `ExEdge`. The former is a basic edge type that simply encapsulates the source and target vertices of an edge, while the latter allows one to specify attributes.

The definition of `Edge` is given by

```
immutable Edge{V}
    index::Int
    source::V
    target::V
end

typealias IEdge Edge{Int}
```

The definition of `ExEdge` is given by

```
type ExEdge{V}
    index::Int
    source::V
    target::V
    attributes::Dict{UTF8String,Any}
end
```

`ExEdge` has two constructors, one takes `index`, `source`, and `target` as arguments, while the other use all four fields.

One can either construct an edge directly using the constructors, or use the `add_edge` methods for graphs, which can automatically assign an index to a new edge.

Both edge types implement the following methods:

edge_index(*e*)
: returns the index of the edge `e`.

source(*e*)
returns the source vertex of the edge `e`.

target(*e*)
returns the target vertex of the edge `e`.

A custom edge type `E{V}` which is constructible by `E(index::Int, s::V, t::V)` and implements the above methods is usable in the `VectorIncidenceList` parametric type. Construct such a list with `inclist(V,E{V})`, where E and V are your vertex and edge types. See test/inclist.jl for an example.

1.3.3 Edge Properties

Many algorithms use a property of an edge such as length, weight, flow, etc. as input. As the algorithms do not mandate any structure for the edge types, these edge properties can be passed through to the algorithm by an `EdgePropertyInspector`. An `EdgePropertyInspector` when passed to the `edge_property` method along with an edge and a graph, will return that property of an edge.

All edge property inspectors should be declared as a subtype of `AbstractEdgePropertyInspector{T}` where `T` is the type of the edge property. The edge propery inspector should respond to the following methods.

Three edge property inspectors are provided `ConstantEdgePropertyInspector`, `VectorEdgePropertyInspector` and `AttributeEdgePropertyInspector`.

`ConstantEdgePropertyInspector(c)` constructs an edge property inspector that returns the constant `c` for each edge.

`VectorEdgePropertyInspector(v)` constructs an edge property inspector that returns `v[edge_index(e, g)]`. It requires that `g` implement the `edge_map` interface.

`AttributeEdgePropertyInspector(name)` constructs an edge property inspector that returns the named attribute from an `ExEdge`. `AttributeEdgePropertyInspector` requires that the graph implements the `edge_map` interface.

1.4 Graph Types

This package provides several graph types that implement different subsets of interfaces. In particular, it has four graph types:

- `GenericEdgeList`
- `GenericAdjacencyList`
- `GenericIncidenceList`
- `GenericGraph`

All of these types are parametric. One can easily derive customized graph types using different type parameters.

1.4.1 Edge List

`GenericEdgeList` implements the edge list representation of a graph, where a list of all edges is maintained by each graph.

Here is the definition of `GenericEdgeList`:

```
type EdgeList{V,E,VList,EList} <: AbstractGraph{V, E}
```

It has four type parameters:

- `V`: the vertex type
- `E`: the edge type
- `VList`: the type of the vertex list
- `EList`: the type of the edge list

The package defines the following aliases for convenience:

```
typealias SimpleEdgeList{E} GenericEdgeList{Int,E,UnitRange{Int},Vector{E}}
typealias EdgeList{V,E} GenericEdgeList{V,E,Vector{V},Vector{E}}
```

`GenericEdgeList` implements the following interfaces

- `vertex_list`
- `vertex_map`
- `edge_list`
- `edge_map`

Specifically, it implements the following methods:

`is_directed`(*g*)
returns whether `g` is a directed graph.

`num_vertices`(*g*)
returns the number of vertices contained in `g`.

`vertices`(*g*)
returns an iterable view/container of all vertices.

`num_edges`(*g*)
returns the number of edges contained in `g`.

`vertex_index`(*v*, *g*)
returns the index of a vertex `v` in graph `g`

`edges`(*g*)
returns the list of all edges

`edge_index`(*e*, *g*)
returns the index of `e` in graph `g`.

In addition, it implements following methods for construction:

`simple_edgelist`(*nv*, *edges*[, *is_directed=true*])
constructs a simple edge list with `nv` vertices and the given list of edges.

`edgelist`(*vs*, *edges*[, *is_directed=true*])
constructs an edge list given lists of vertices and edges.

1.4.2 Adjacency List

`GenericAdjacencyList` implements the adjacency list representation of a graph, where each vertex maintains a list of neighbors (*i.e.* adjacent vertices).

Here is the definition of `GenericAdjacencyList`:

```
type GenericAdjacencyList{V, VList, AdjList} <: AbstractGraph{V, Edge{V}}
```

It has three type parameters:

- `V`: the vertex type
- `VList`: the type of vertex list
- `AdjList`: the type of the adjacency list. Let `a` be an instance of `AdjList`, and `i` be the index of a vertex, then `a[i]` must be an iterable container of the neighbors.

The package defines following aliases for convenience:

```
typealias SimpleAdjacencyList GenericAdjacencyList{Int, UnitRange{Int}, Vector{Vector{Int}}}
typealias AdjacencyList{V} GenericAdjacencyList{V, Vector{V}, Vector{Vector{V}}}
```

`GenericAdjacencyList` implements the following interfaces

- `vertex_list`
- `vertex_map`
- `adjacency_list`

Specifically, it implements the following methods:

`is_directed`(*g*)
returns whether `g` is a directed graph.

`num_vertices`(*g*)
returns the number of vertices contained in `g`.

`vertices`(*g*)
returns an iterable view/container of all vertices.

`num_edges`(*g*)
returns the number of edges contained in `g`.

`vertex_index`(*v*, *g*)
returns the index of a vertex `v` in graph `g`

`out_degree`(*v*, *g*)
returns the number of outgoing neighbors from vertex `v` in graph `g`.

`out_neighbors`(*v*, *g*)
returns an iterable view/container of all outgoing neighbors of vertex `v` in graph `g`.

In addition, it implements following methods for construction:

`simple_adjlist`(*nv*[, *is_directed=true*])
constructs a simple adjacency list with `nv` vertices and no edges (initially).

`adjlist`(*V*[, *is_directed=true*])
constructs an empty adjacency list of vertex type `V`.

`adjlist`(*vs*[, *is_directed=true*])
constructs an adjacency list with a vector of vertices given by `vs`.

`add_vertex!(g, v)`
adds a vertex `v`. This function applies only to graph of type `AdjacencyList`. It returns the added vertex.

If the vertex type is `KeyVertex{K}`, then the second argument here can be the key value, and the function will constructs a vertex and assigns an index.

`add_edge!(g, u, v)`
adds an edge between u and v, such that `v` becomes an outgoing neighbor of `u`. If `g` is undirected, then `u` is also added to the neighbor list of `v`.

1.4.3 Incidence List

`GenericIncidenceList` implements the incidence list representation of a graph, where each vertex maintains a list of outgoing edges.

Here is the definition of `GenericIncidenceList`:

```
type GenericIncidenceList{V, E, VList, IncList} <: AbstractGraph{V, E}
```

It has four type parameters:

- `V`: the vertex type
- `E`: the edge type
- `VList`: the type of vertex list
- `IncList`: the type of incidence list. Let `a` be such a list, then `a[i]` should be an iterable container of edges.

The package defines following aliases for convenience:

```
typealias SimpleIncidenceList GenericIncidenceList{Int, IEdge, UnitRange{Int}, Vector{Vector{IEdge}}}
typealias IncidenceList{V,E} GenericIncidenceList{V, E, Vector{V}, Vector{Vector{E}}}
```

`GenericIncidenceList` implements the following interfaces:

- `vertex_list`
- `vertex_map`
- `edge_map`
- `adjacency_list`
- `incidence_list`

Specially, it implements the following methods:

`is_directed`(*g*)
: returns whether `g` is a directed graph.

`num_vertices`(*g*)
: returns the number of vertices contained in `g`.

`vertices`(*g*)
: returns an iterable view/container of all vertices.

`num_edges`(*g*)
: returns the number of edges contained in `g`.

`vertex_index`(*v*, *g*)
: returns the index of a vertex `v` in graph `g`

`edge_index`(*e*, *g*)
: returns the index of an edge `e` in graph `g`.

`source`(*e*, *g*)
: returns the source vertex of an edge `e` in graph `g`.

`target`(*e*, *g*)
: returns the target vertex of an edge `e` in graph `g`.

`out_degree`(*v*, *g*)
: returns the number of outgoing neighbors from vertex `v` in graph `g`.

out_edges(*v*, *g*)
returns the number of outgoing edges from vertex `v` in graph `g`.

out_neighbors(*v*, *g*)
returns an iterable view/container of all outgoing neighbors of vertex `v` in graph `g`.

Note: `out_neighbors` here is implemented based on `out_edges` via a proxy type. Therefore, it may be less efficient than the counterpart for `GenericAdjacencyList`.

In addition, it implements following methods for construction:

simple_inclist(*nv*[, *is_directed=true*])
constructs a simple incidence list with `nv` vertices and no edges (initially).

inclist(*V*[, *is_directed=true*])
constructs an empty incidence list of vertex type `V`. The edge type is `Edge{V}`.

inclist(*vs*[, *is_directed=true*])
constructs an incidence list with a list of vertices `vs`. The edge type is `Edge{V}`.

inclist(*V*, *E*[, *is_directed=true*])
constructs an empty incidence list of vertex type `V`. The edge type is `E`.

inclist(*vs*, *E*[, *is_directed=true*])
constructs an incidence list with a list of vertices `vs`. The edge type is `E`.

add_vertex!(g, x)
adds a vertex. Here, `x` can be of a vertex type, or can be made into a vertex using `make_vertex(g, x)`.

add_edge!(g, e)
adds an edge `e` to the graph.

add_edge!(g, u, v)
adds an edge between `u` and `v`. This applies when `make_edge(g, u, v)` is defined for the input types.

1.4.4 Graph

`GenericGraph` provides a complete interface by integrating edge list, bidirectional adjacency list, and bidirectional incidence list into one type. The definition is given by

```
type GenericGraph{V,E,VList,EList,IncList} <: AbstractGraph{V,E}
```

It has six type parameters:

- `V`: the vertex type
- `E`: the edge type
- `VList`: the type of vertex list
- `EList`: the type of edge list
- `IncList`: the type of incidence list

It also defines `SimpleGraph` as follows

```
typealias SimpleGraph GenericGraph{Int,IEdge,UnitRange{Int},Vector{IEdge},Vector{Vector{IEdge}}}
```

and a more full-fledged type `Graph` as follows

```
typealias Graph{V,E} GenericGraph{V,E,Vector{V},Vector{E},Vector{Vector{E}}}
```

`GenericGraph` implements the following interfaces:

- `vertex_list`
- `edge_list`
- `vertex_map`
- `edge_map`
- `adjacency_list`
- `incidence_list`
- `bidirectional_adjacency_list`
- `bidirectional_incidence_list`

Specifically, it implements the following methods:

is_directed(*g*)
returns whether `g` is a directed graph.

num_vertices(*g*)
returns the number of vertices contained in `g`.

vertices(*g*)
returns an iterable view/container of all vertices.

num_edges(*g*)
returns the number of edges contained in `g`.

edges(*g*)
returns an iterable view/container of all edges.

vertex_index(*v*, *g*)
returns the index of a vertex `v` in graph `g`

edge_index(*e*, *g*)
returns the index of a vertex `e` in graph `g`.

source(*e*, *g*)
returns the source vertex of an edge `e` in graph `g`.

target(*e*, *g*)
returns the target vertex of an edge `e` in graph `g`.

out_degree(*v*, *g*)
returns the number of outgoing neighbors from vertex `v` in graph `g`.

out_edges(*v*, *g*)
returns the number of outgoing edges from vertex `v` in graph `g`.

out_neighbors(*v*, *g*)
returns an iterable view/container of all outgoing neighbors of vertex `v` in graph `g`.

in_degree(*v*, *g*)
returns the number of incoming neighbors to vertex `v` in graph `g`.

in_edges(*v*, *g*)
returns the number of incoming edges to vertex `v` in graph `g`.

in_neighbors(*v*, *g*)
returns an iterable view/container of all incoming neighbors to vertex `v` in graph `g`.

In addition, it also implements the following methods for construction:

simple_graph (*nv*[, *is_directed=true*])
constructs an instance of `SimpleGraph` with `nv` vertices and no edges (initially).

graph (*vertices*, *edges*[, *is_directed=true*])
constructs an instance of `Graph` with given vertices and edges.

add_vertex!(g, x)
adds a vertex. Here, `x` can be of a vertex type, or can be made into a vertex using `make_vertex(g, x)`.

add_edge!(g, e)
adds an edge `e` to the graph.

add_edge!(g, u, v)
adds an edge between `u` and `v`. This applies when `make_edge(g, u, v)` is defined for the input types.

1.5 Graph Algorithms

`Graphs.jl` implements a collection of classic graph algorithms:

- graph traversal with visitor support: BFS, DFS, MAS
- cycle detection
- connected components
- topological sorting
- shortest paths: Dijkstra, Floyd-Warshall, A*
- minimum spanning trees: Prim, Kruskal
- flow: Minimum Cut
- random graph generation
- more algorithms are being implemented

1.5.1 Graph Traversal

Graph traversal refers to a process that traverses vertices of a graph following certain order (starting from user-input sources). This package implements three traversal schemes: *breadth-first*, *depth-first*, and *Maximum-Adjacency*.

During traveral, each vertex maintains a status (also called *color*), which is an integer value defined as below:

- `color = 0`: the vertex has not been encountered (*i.e.* discovered)
- `color = 1`: the vertex has been discovered and remains open
- `color = 2`: the vertex has been closed (*i.e.* all its neighbors have been examined)

traverse_graph (*graph*, *alg*, *source*, *visitor*[, *colormap*])

Parameters

- **graph** – The input graph, which must implement `vertex_map` and `adjacency_list`.
- **alg** – The algorithm of traveral, which can be either `BreadthFirst()` or `DepthFirst()`.
- **source** – The source vertex (or vertices). The traversal starts from here.
- **visitor** – The visitor which performs certain operations along the traversal.

- **colormap** – An integer vector that indicates the status of each vertex. If this is input by the user, the status will be written to the input vector, otherwise an internal color vector will be created.

Here, `visitor` must be an instance of a sub-type of `AbstractGraphVisitor`. A specific graph visitor type can choose to implement some or all of the following methods.

`discover_vertex!(visitor, v)`
invoked when a vertex `v` is encountered for the first time. This function should return whether to continue traversal.

`open_vertex!(visitor, v)`
invoked when a vertex `v` is about to examine `v`'s neighbors.

`examine_neighbor!(visitor, u, v, color, ecolor)`
invoked when a neighbor/out-going edge is examined. Here `color` is the status of `v`, and `ecolor` is the status of the outgoing edge. Edge statuses are currently only considered by depth-first search.

`close_vertex!(visitor, v)`
invoked when all neighbors of `v` has been examined.

If a method of these is not implemented, it will automatically fallback to no-op. The package provides some pre-defined visitor types:

- `TrivialGraphVisitor`: all methods are no-op.
- `VertexListVisitor`: it has a field `vertices`, which is a vector comprised of vertices in the order of being discovered.
- `LogGraphVisitor`: it prints message to show the progress of the traversal.

Many graph algorithms can be implemented based on graph traversal through certain visitors or by using the colormap in certain ways. For example, in this package, topological sorting, connected components, and cycle detection are all implemented using `traverse_graph` with specifically designed visitors.

1.5.2 Cycle detection

In graph theory, a cycle is defined to be a path that starts from some vertex `v` and ends up at `v`.

`test_cyclic_by_dfs`(*g*)
Tests whether a graph contains a cycle through depth-first search. It returns `true` when it finds a cycle, otherwise `false`. Here, `g` must implement `vertex_list`, `vertex_map`, and `adjacency_list`.

1.5.3 Connected components

In graph theory, a connected component (in an undirected graph) refers to a subset of vertices such that there exists a path between any pair of them.

`connected_components`(*g*)
Returns a vector of components, where each component is represented by a vector of vertices. Here, `g` must be an undirected graph, and implement `vertex_list`, `vertex_map`, and `adjacency_list`.

1.5.4 Cliques

In graph theory, a clique in an undirected graph is a subset of its vertices such that every two vertices in the subset are connected by an edge. A maximal clique is the largest clique containing a given node.

maximal_cliques(*g*)
Returns a vector of maximal cliques, where each maximal clique is represented by a vector of vertices. Here, `g` must be an undirected graph, and implement `vertex_list` and `adjacency_list`.

1.5.5 Topological Sorting

Topological sorting of an acyclic directed graph is a linear ordering of vertices, such that for each directed edge `(u, v)`, `u` always comes before `v` in the ordering.

topological_sort_by_dfs(*g*)
Returns a topological sorting of the vertices in `g` in the form of a vector of vertices. Here, `g` may be directed or undirected, and implement `vertex_list`, `vertex_map`, and `adjacency_list`.

1.5.6 Shortest Paths

This package implements three classic algorithms for finding shortest paths: *Dijkstra's algorithm*, the *Floyd-Warshall algorithm*, and the *A* algorithm*. We plan to implement the *Bellman-Ford algorithm* and *Johnson's algorithm* in the near future.

Dijkstra's Algorithm

dijkstra_shortest_paths(*graph*, *edge_dists*, *source*[, *visitor*])
Performs Dijkstra's algorithm to find shortest paths to all vertices from input sources.

Parameters

- **graph** – The input graph
- **edge_dists** – The vector of edge distances or an edge property inspector.
- **source** – The source vertex (or vertices)
- **visitor** – An visitor instance

Returns An instance of `DijkstraStates` that encapsulates the results.

Here, `graph` can be directed or undirected. It must implement `vertex_map`, `edge_map` and `incidence_list`. *edge_dists* is optional; if not specified, default distances of *1* are used for each edge.

The following is an example that shows how to use this function:

```
# construct a graph and the edge distance vector

g = simple_inclist(5)

inputs = [       # each element is (u, v, dist)
    (1, 2, 10.),
    (1, 3, 5.),
    (2, 3, 2.),
    (3, 2, 3.),
    (2, 4, 1.),
    (3, 5, 2.),
    (4, 5, 4.),
    (5, 4, 6.),
    (5, 1, 7.),
    (3, 4, 9.) ]
```

```
ne = length(inputs)
dists = zeros(ne)

for i = 1 : ne
    a = inputs[i]
    add_edge!(g, a[1], a[2])   # add edge
    dists[i] = a[3]             # set distance
end

r = dijkstra_shortest_paths(g, dists, 1)

@assert r.parents == [1, 3, 1, 2, 3]
@assert r.dists == [0., 8., 5., 9., 7.]
```

The result has several fields, among which the following are most useful:

- `parents[i]`: the parent vertex of the i-th vertex. The parent of each source vertex is itself.
- `hasparent[i]`: `true` if the i-th vertex has a parent, and `false` otherwise. When `hasparent[i] == false`, it means that the vertex at index `i` isn't reachable from any source. Note that `hasparent[i] == true` for all source vertices.
- `dists[i]`: the minimum distance from the i-th vertex to source.

The user can (optionally) provide a visitor that perform operations along with the algorithm. The visitor must be an instance of a sub type of `AbstractDijkstraVisitor`, which may implement part of all of the following methods.

`discover_vertex!(visitor, u, v, d)`
Invoked when a new vertex `v` is first discovered (from the parent `u`). `d` is the initial distance from `v` to source.

`include_vertex!(visitor, u, v, d)`
Invoked when the distance of a vertex is determined (at the point `v` is popped from the heap). This function should return whether to continue the procedure. One can use a visitor to terminate the algorithm earlier by letting this function return `false` under certain conditions.

`update_vertex!(visitor, u, v, d)`
Invoked when the distance to a vertex is updated (relaxed).

`close_vertex!(visitor, u, v, d)`
Invoked when a vertex is closed (all its neighbors have been examined).

`enumerate_paths`(*vertices*, *parent_indices*[, *dest*])
Returns an array of vectors (containing vertices), whose `i`-th element corresponds to the path from a source to vertex `dest[i]`. Empty vectors indicate vertices that are unreachable from the source. `dest` can be a subset of indices, or left unspecified (in which case, all the indices will be considered). If `dest` is a single index, then the result is just an array of vertices, corresponding to the path from a source to `dest`.

`enumerate_indices`(*parent_indices*[, *dest*])
Returns an array of indices corresponding to the vertices returned by *enumerate_paths(vertices, parent_indices[, dest])*

The following is an example that shows how to use this function:

```
julia> g4 = Graphs.inclist([4,5,6,7],is_directed=true)
julia> add_edge!(g4,4,5); add_edge!(g4,4,6); add_edge!(g4,5,6); add_edge!(g4,6,7)
julia> s4 = dijkstra_shortest_paths(g4,5)
julia> sps = enumerate_indices(s4.parent_indices) # dest: all indices
4-element Array{Array{Int64,1},1}:
 []
 [2]
```

```
 [2,3]
 [2,3,4]

julia> enumerate_indices(s4.parent_indices, [2,4]) # dest: subset of indices
2-element Array{Array{Int64,1},1}:
 [2]
 [2,3,4]

julia> enumerate_indices(s4.parent_indices, 4) # dest: single index
3-element Array{Int64,1}:
 2
 3
 4

julia> enumerate_paths(vertices(g4), s4.parent_indices) # dest: all vertices
4-element Array{Array{Int64,1},1}:
 []
 [5]
 [5,6]
 [5,6,7]

julia> enumerate_paths(vertices(g4), s4.parent_indices, [2,4]) # dest: subset of vertices
2-element Array{Array{Int64,1},1}:
 [5]
 [5,6,7]

julia> enumerate_paths(vertices(g4), s4.parent_indices, 4) # dest: single vertex
3-element Array{Int64,1}:
 5
 6
 7
```

Remark: `enumerate_paths` and `enumerate_indices` are applicable to the results from both `dijkstra_shortest_paths` and `bellman_ford_shortest_paths`.

Bellman Ford Algorithm

bellman_ford_shortest_paths(*graph*, *edge_dists*, *source*)

Performs Bellman Ford algorithm to find shortest paths to all vertices from input sources.

Parameters

- **graph** – The input graph
- **edge_dists** – The vector of edge distances or an edge property inspector.
- **source** – The source vertex (or vertices)

Returns An instance of `BellmanFordStates` that encapsulates the results.

Here, `graph` can be directed or undirected. Weights can be negative for a directed graph. It must implement `vertex_map`, `edge_map` and `incidence_list`. If there is a negative weight cycle an exception of `NegativeCycleError` is thrown.

The result has several fields, among which the following are most useful:

- `parents[i]`: the parent vertex of the i-th vertex. The parent of each source vertex is itself.
- `dists[i]`: the minimum distance from the i-th vertex to source.

has_negative_edge_cycle(*graph*, *edge_dists*)

Tests if the graph has a negative weight cycle.

Parameters

- **graph** – The input graph
- **edge_dists** – The vector of edge distances or an edge property inspector.

Returns `true` if there is a negative weight cycle, `false` otherwise.e

Floyd-Warshall's algorithm

floyd_warshall (*dists*)
Performs Floyd-Warshall algorithm to compute shortest path lengths between each pair of vertices.

Parameters **dists** – The edge distance matrix.

Returns The matrix of shortest path lengths.

floyd_warshall!(dists)
Performs Floyd-Warshall algorithm inplace, updating an edge distance matrix into a matrix of shortest path lengths.

floyd_warshall!(dists, nexts)
Performs Floyd-Warshall algorithm inplace, and writes the next-hop matrix. When this function finishes, `nexts[i,j]` is the next hop of `i` along the shortest path from `i` to `j`. One can reconstruct the shortest path based on this matrix.

A*

shortest_path (*graph*, *dists*, *s*, *t*[, *heuristic*])
Find the shortest path between vertices `s` and `t` of `graph` using Hart, Nilsson and Raphael's A* algorithm.

Parameters

- **graph** – the input graph
- **dists** – the edge distance matrix or an edge property inspector
- **s** – the start vertex
- **t** – the end vertex
- **heuristic** – a function underestimating the distance from its input node to `t`.

Returns an array of edges representing the shortest path.

1.5.7 Minimum Spanning Trees

This package implements two algorithm to find a minimum spanning tree of a graph: *Prim's algorithm* and *Kruskal's algorithm*.

Prim's algorithm

Prim's algorithm finds a minimum spanning tree by growing from a root vertex, adding one edge at each iteration.

prim_minimum_spantree (*graph*, *eweights*, *root*)
Perform Prim's algorithm to find a minimum spanning tree.

Parameters

- **graph** – the input graph
- **eweights** – the edge weights (a vector or an edge property inspector)
- **root** – the root vertex

Returns `(re, rw)`, where `re` is a vector of edges that constitute the resultant tree, and `rw` is the vector of corresponding edge weights.

Kruskal's algorithm

Kruskal's algorithm finds a minimum spanning tree (or forest) by gradually uniting disjoint trees.

kruskal_minimum_spantree(*graph*, *eweights*[, *K=1*])

Parameters

- **graph** – the input graph
- **eweights** – the edge weights (a vector or an edge property inspector)
- **K** – the number of trees in the resultant forest. If `K = 1`, it ends up with a tree. This argument is optional. By default, it is set to `1`.

Returns `(re, rw)`, where `re` is a vector of edges that constitute the resultant tree, and `rw` is the vector of corresponding edge weights.

1.5.8 Flow

This package implements Simple Minimum Cut

Simple Minimum Cut

Stoer's simple minimum cut gets the minimum cut of an undirected graph.

min_cut(*graph*[, *eweights*])

Parameters

- **graph** – the input graph
- **eweights** – the edge weights (a vector or an edge property inspector). This argument is optional. If not given edges are weight "1"

Returns `(parity, bestcut)`, where `parity` is a vector of boolean values that determines the partition and `bestcut` is the weight of the cut that makes this partition.

1.5.9 Random Graphs

Erdős–Rényi graphs

The Erdős–Rényi model sets an edge between each pair of vertices with equal probability, independently of the other edges.

erdos_renyi_graph(*g, n, p[; has_self_loops=false]*)

Add edges between vertices 1:n of graph `g` randomly, adding each possible edge with probability `p` independently of all others.

Parameters

- **g** – the input graph
- **n** – the number of vertices between which to add edges
- **p** – the probability with which to add each edge
- **has_self_loops** – whether to consider edges `v -> v`.

Returns the graph `g`.

erdos_renyi_graph(*n*, *p*[, *has_self_loops=false*])
Convenience function to construct an `n`-vertex Erdős–Rényi graph as an incidence list.

Watts-Strogatz graphs

The Watts–Strogatz model is a random graph generation model that produces graphs with small-world properties, including short average path lengths and high clustering.

watts_strogatz_graph(*g*, *n*, *k*, *beta*)
Adjust the edges between vertices 1:n of the graph `g` in accordance with the Watts-Strogatz model.

Parameters

- **g** – the input graph
- **n** – the number of vertices between which to adjust edges
- **k** – the base degree of each vertex (n > k, k >= 2, k must be even.)
- **beta** – the probability of each edge being "rewired".

Returns the graph `g`.

watts_strogatz_graph(*n*, *k*, *beta*)
Convenience function to construct an `n`-vertex Watts-Strogatz graph as an incidence list.

1.6 Matrix Representation

Matrix representation of graphs are widely used in algebraic analysis of graphs. This package comprises functions that derive matrix representation of an input graph.

1.6.1 Adjacency Matrix

An *adjacency matrix* is defined as

$$A(u,v) = \begin{cases} 1 & (u,v) \in E \\ 0 & \text{otherwise} \end{cases}$$

adjacency_matrix(*graph*)
Constructs an adjacency matrix for a graph.

1.6.2 Weight Matrix

A *weight matrix* is defined as

$$W(u,v) = \begin{cases} w(e) & e = (u,v) \in E \\ 0 & \text{otherwise} \end{cases}$$

weight_matrix(*graph*, *eweights*)
Constructs a weight matrix from a graph and a vector of edge weights. Here, `g` must implement `edge_map` and (`edge_list` or `incidence_list`).

1.6.3 Distance Matrix

A *distance matrix* is defined as

$$W(u,v) = \begin{cases} 0 & u = v \\ w(e) & u \neq v \text{ and } \{u,v\} \in E \\ Inf & \text{otherwise} \end{cases}$$

distance_matrix(*graph*, *eweights*)
Constructs a distance matrix from a graph and a vector of edge weights. Here, `g` must implement `edge_map` and (`edge_list` or `incidence_list`).

1.6.4 Laplacian Matrix

Laplacian matrix is significant in algebraic graph theory. The eigenvalues of a Laplacian matrix characterizes important properties of a graph. For an undirected graph, it is defined as:

$$L(u,v) = \begin{cases} deg(u) & u = v \\ -1 & u \neq v \text{ and } \{u,v\} \in E \\ 0 & \text{otherwise} \end{cases}$$

laplacian_matrix(*graph*)
Constructs a Laplacian matrix over an undirected graph.

For graphs with weighted edges, we have

laplacian_matrix(*graph*, *eweights*)
Constructs a weighted Laplacian matrix from an undirected graph with a vector of edge weights.

1.7 Graph Generators

`Graphs.jl` implements a collection of classic graph generators, each of which returns a `simple_graph`:

static_complete_graph(*n*[, *is_directed=true*])
Creates a (default directed) complete graph with `n` vertices. A complete graph has edges connecting each pair of vertices.

simple_star_graph(*n*[, *is_directed=true*])
Creates a (default directed) star graph with `n` vertices. A star graph has a central vertex with edges to each other vertex.

simple_path_graph(**n**[, **is_directed=true**])
Creates a (default directed) path graph with `n` vertices. A path graph connects each successive vertex by a single edge.

simple_wheel_graph(*n*[, *is_directed=true*])
Creates a (default directed) wheel graph with `n` vertices. A wheel graph is a star graph with the outer vertices connected via a closed path graph.

simple_diamond_graph()
A diamond graph.

simple_bull_graph()
A bull graph.

simple_chvatal_graph()
A Chvátal graph.

simple_cubical_graph()
A Platonic cubical graph.

simple_desargues_graph()
A Desargues graph.

simple_dodecahedral_graph()
A Platonic dodecahedral graph.

simple_frucht_graph()
A Frucht graph.

simple_heawood_graph()
A Heawood graph.

simple_house_graph()
A graph mimicing the classic outline of a house.

simple_house_x_graph()
A house graph, with two edges crossing the bottom square.

simple_icosahedral_graph()
A Platonic icosahedral graph.

simple_krackhardt_kite_graph()
A Krackhardt-Kite social network.

moebius_kantor_graph()
A Möbius-Kantor graph.

simple_octahedral_graph()
A Platonic octahedral graph.

simple_pappus_graph()
A Pappus graph.

simple_petersen_graph()
A Petersen graph.

simple_sedgewick_maze_graph()
A simple maze graph used in Sedgewick's *Algorithms in C++: Graph Algorithms (3rd ed.)*

simple_tetrahedral_graph()
A Platonic tetrahedral graph.

simple_truncated_cube_graph()
A skeleton of the truncated cube graph.

simple_truncated_tetrahedron_graph()
A skeleton of the truncated tetrahedron graph.

simple_tutte_graph()
A Tutte graph.

1.8 Examples

To help you to get started with Graphs.jl, here is a simple example:

```
julia> using Graphs

julia> g = simple_graph(3)
Directed Graph (3 vertices, 0 edges)

julia> add_edge!(g, 1, 2)
edge [1]: 1 -- 2

julia> add_edge!(g, 3, 2)
edge [2]: 3 -- 2

julia> add_edge!(g, 3, 1)
edge [3]: 3 -- 1

julia> plot(g)
```

We can also generate simple graphs with graph generators. For example:

```
julia> g2 = simple_cubical_graph()
Undirected Graph (8 vertices, 12 edges)
```

and check the number of vertices and edges with:

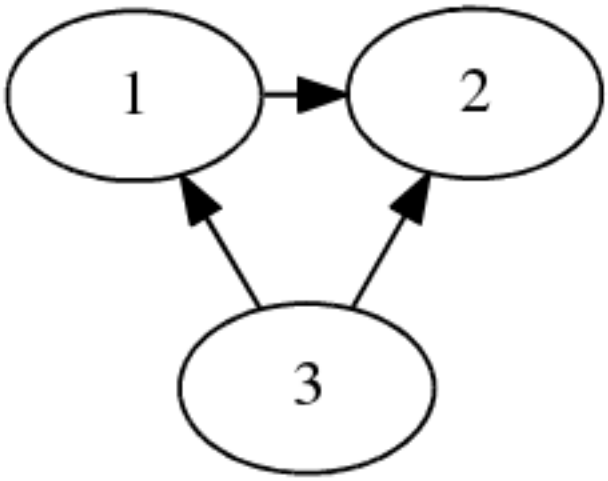
```
julia> num_vertices(g2)
8

julia> num_edges(g2)
12
```

We can use an adjacency matrix to represent `g2`:

```
julia> adjacency_matrix(g2)
8x8 Array{Bool,2}:
false   true  false   true   true  false  false  false
true  false   true  false  false  false  false   true
false   true  false   true  false  false   true  false
true  false   true  false  false   true  false  false
true  false  false  false  false   true  false   true
false  false  false   true   true  false   true  false
false  false   true  false  false   true  false   true
false   true  false  false   true  false   true  false
```

graphviz:



Indices and tables

- genindex
- search

T

V

W