A SIMPLE GRAPH TYPE FOR JULIA

ED SCHEINERMAN

1. Fundamentals

This is documentation for a `SimpleGraph` data type for Julia. The goal is to make working with graphs as painless as possible. The `SimpleGraph` data type is for simple graphs (undirected edges, no loops, no multiple edges). Vertices in a `SimpleGraph` are of a given Julia data type, which might be `Any`.

To begin, get the repository with this Julia command:

```
Pkg.clone("https://github.com/scheinerman/SimpleGraphs.jl.git")
```

This only has to be done once. To use the `SimpleGraph` type, give the command `using SimpleGraphs`.

The key constructor is `SimpleGraph` which creates a graph whose vertices may be any Julia type. Alternatively, `G=SimpleGraph{T}()` sets up `G` to be a graph in which all vertices must be of type `T`. Two special cases are built into this module: `IntGraph()` is a synonym for `SimpleGraph{Int}()` and `StringGraph()` is a synonym for `SimpleGraph{ASCIIString}`.

Vertices and edges are added to a graph with `add!` and deleted with `delete!`. Membership is checked with `has`.

```
julia> using SimpleGraphs

julia> G = IntGraph()
SimpleGraph{Int64} (0 vertices, 0 edges)

julia> add!(G,5)
true

julia> add!(G,1,2)
true

julia> G
SimpleGraph{Int64} (3 vertices, 1 edges)
```

Notice that adding an edge automatically adds its end points to the graph.

The number of vertices and edges can be queried with `NV` and `NE`. The vertex and edge sets are returned as arrays by `vlist(G)` and `elist(G)`.

```
julia> vlist(G)
3-element Array{Int64,1}:
 1
 2
 5

julia> elist(G)
1-element Array{(Int64,Int64),1}:
 (1,2)
```

Document version `2014:09:20:10:46`. Note: This documentation might not quite be in sync with the functionality found in the `SimpleGraphs` module. I will likely want to redo this at some point.

Use `deg(G,v)` for the degree of a vertex and `deg(G)` for the graph's degree sequence.

```
julia> deg(G,1)
1

julia> deg(G)
3-element Array{Int64,1}:
 1
 1
 0
```

The neighbors of a vertex can be sought with `neighbors(G,v)` or, alternatively, with `G[v]`. Edges can be queried with `G[v,w]`.

```
julia> G[1]
1-element Array{Int64,1}:
 2

julia> G[1,2]
true

julia> G[1,5]
false
```

The `Complete` method can be used to create complete graphs and complete bipartite graphs with `Int` vertices.

```
julia> Complete(5)
SimpleGraph{Int64} (5 vertices, 10 edges)

julia> Complete(3,3)
SimpleGraph{Int64} (6 vertices, 9 edges)
```

We also provide `Cycle(n)` and `Path(n)` to create the graphs C_n and P_n. An instance of a Erdős-Rényi random graph $G_{n,p}$ is returned by `RandomGraph(n,p)`.

The adjacency, Laplacian, and vertex-edge incidence matrices can be found with `adjaceny`, `Laplacian`, and `incidence`. It is important to note that the indexing of the rows/columns of these matrices might not correspond to the natural order of the underlying vertices (or edges).

The incidence matrix method takes an optional argument as to whether the matrix is signed (a 1 and a -1 in each column) or unsigned (only positive 1s). Also, it is a sparse matrix that can be converted to dense storage with `full`.

Finally, the `vertex_type` function can be used to query the data type of the vertices in the graph.

```
julia> G = StringGraph()
SimpleGraph{ASCIIString} (0 vertices, 0 edges)

julia> vertex_type(G)
ASCIIString (constructor with 1 method)
```

2. Look but don't touch

The `SimpleGraph` objects have three internal fields. It is not safe to change these directly, but there is no problem examining their values.

- `:V`
  This holds the vertex set of the graph. If `G` is a `SimpleGraph{T}` then `G.V` is of type `Set{T}`.
- `:E`
  This holds the edge set of the graph. If `G` is a `SimpleGraph{T}` then `G.E` is a `Set{(T,T)}`.

- `:Nflag`
  This boolean value indicates whether or not fast neighborhood lookup has been activated for this graph. See the discussion below and the description of the function `fastN!`.
- `:N`
  This is a `Dict` that keeps track of the neighborhood of each vertex. It is only active if the `Nflag` is set to `true`.

This design is redundant. One can test if two vertices are adjacent by looking for the edge in `E`. One can determine the neighbors of a given vertex by iterating over the edge set `E`. However, this approach is slow. By providing the extra `N` data structure, these operations are very fast.

By calling `fastN!(G,false)` the redudant neighborhood structure `N` is deleted. All functions will still work, but perhaps more slowly. Calling `fastN!(G,true)` rebuilds the `N` structure.

We recommend using the default structure unless the graph is so large that it consumes too much memory.

3. List of all functions

Creators. These are functions that create new graphs.

- `SimpleGraph`
  Use `G=SimpleGraph()` to create a graph whose vertices can be of `Any` type. To create a graph with vertices of a particular type `T` use `G=SimpleGraph{T}()`.
- `IntGraph`
  This a synonym for `SimpleGraph{Int}`. Use `G=IntGraph()` to create a vertex whose vertices are integers (type `Int`).
  Use `IntGraph(n)` to create a `SimpleGraph{Int}` graph with vertex set $\{1,2,\ldots,n\}$ and no edges.
- `StringGraph`
  This is a synonym for `SimpleGraph{ASCIIString}`. Use `G=StringGraph()` to create a graph whose vertices are character strings.
  One can also call `G=StringGraph(filename)` to read in a graph from a file. The file must have the following format:
  - Each line should contain one or two tokens (words) that do not contain any whitespace.
  - If a line contains one token, that token is added to the graph as a vertex.
  - If a line contains two tokens, an edge is added with those tokens as end points. If those two tokens are the same, no edge is created (since we do not allow loops) but a vertex is added (if not already in the graph).
  - If there are three or more tokens on a line, only the first two are read and the rest are ignored.
  - If the line is blank, it is ignored.
  - If the line begins with a `#`, the entire line is ignored. (This does put a mild limitation on the names of vertices.)
- `Complete`
  The `Complete` function can be used to create a complete graph K_n, a complete bipartite graph $K_{n,m}$, or a complete multipartite graph $K(n_1,n_2,\ldots,n_p)$.
  - Use `Complete(n)` to create a complete graph K_n.
  - Use `Complete(n,m)` to create a complete bipartite graph $K_{n,m}$.
  - Use `Complete([n1,n2,...,np])` to create a complete multipartite graph $K(n_1,n_2,\ldots,n_p)$.

  Note the last version requires that the part sizes be in an array. In this way we distinguish `Complete(n)` and `Complete([n])`. The first makes K_n and the second an edgeless graph with n vertices, i.e., $\overline{K_n}$. However, `Complete(n,m)` and `Complete([n,m])` build exactly the same graphs.

```
julia> G = Complete(4,5)
SimpleGraph{Int64} (9 vertices, 20 edges)

julia> H = Complete([4,5])
```

```
SimpleGraph{Int64} (9 vertices, 20 edges)

julia> G == H
true
```

- `Cube`

  Create the cube graph. The 2^n vertices of `Cube(n)` are `ASCIIString` objects. For example:

```
julia> G = Cube(3)
SimpleGraph{ASCIIString} (8 vertices, 12 edges)

julia> vlist(G)
8-element Array{ASCIIString,1}:
 "000"
 "001"
 "010"
 "011"
 "100"
 "101"
 "110"
 "111"

julia> elist(G)
12-element Array{(ASCIIString,ASCIIString),1}:
 ("000","001")
 ("000","010")
 ("000","100")
 ("001","011")
 ("001","101")
 ("010","011")
 ("010","110")
 ("011","111")
 ("100","101")
 ("100","110")
 ("101","111")
 ("110","111")
```

- `Path`

  Use `Path(n)` to create a path graph with n vertices.

  Also, given a list of vertices `verts`, then `Path(verts)` creates a path graph with edges `(verts[k],verts[k+1])` when `k=1:n-1`. It's the user's responsibility that there be no repeated entries in `verts`.
- `Grid`

  Use `Grid(n,m)` to create an $n \times m$ grid graph.
- `Cycle`

  Use `Cycle(n)` to create a cycle graph with n vertices. It is required that $n \geq 3$.
- `Wheel`

  Use `Wheel(n)` to create the wheel graph with n vertices. That is, a graph composed of an $(n-1)$-cycle with one additional vertex adjacent to every vertex on the cycle. This requires $n \geq 4$.
- `BuckyBall`

  Create the Buckyball graph with 30 vertices and 90 edges.
- `RandomGraph`

  Use `RandomGraph(n,p)` to create an Erdős-Rényi random graph with n vertices with edge probability p. If the argument `p` is omitted, it is assumed $p = \frac{1}{2}$.
- `RandomTree`

Use `RandomTree(n)` to create a random tree on vertex set `1:n`. All n^{n-2} trees are equally likely.

This works by creating an $n-2$-long sequence of random values, each in the range `1:n`. It then converts that Prüfer code to a tree using `code_to_tree`. This latter function is exposed for use by the user. It takes as input an array of integers and returns a tree assuming, that is, that the input is valid. No checks are done on the input so user beware.

- `Kneser`

  Use `Kneser(n,k)` to create the Kneser graph with those parameters (with $0 \le k \le n$). This is a graph with $\binom{n}{k}$ vertices that are the k-element subsets of $\{1, 2, \dots, n\}$. Two vertices u and v are adjacent iff $u \cap v = \emptyset$.

  Part of this implementation is a function `subsets(A,k)` where `A` is a `Set` and `k` is an `Int`. This creates the set of all k-element subsets of A.
- `Petersen`

  Use `Peteren()` to create Petersen's graph. This is created by calculating `Kneser(5,2)`.

  To remap the vertex names to $\{1, 2, \dots, 10\}$ use `relabel(Petersen())`.

Graph operations. These are operations that create new graphs from old.

- `line_graph`

  Use `line_graph(G)` to create the line graph of G. Note that if `G` has vertex type `T`, then this creates a graph with vertex type `(T,T)`.
- `complement` and `complement!`

  Use `complement(G)` to create the graph $\overline{G}$. The original graph is not changed and the vertex type of the new graph is that same the vertex type of `G`.

  We can use `G'` in lieu of `complement(G)`.

  Use `complement!(G)` to complement a graph in place (i.e., replace `G` with its own complement).
- `copy`

  Use `copy(G)` to create an independent copy of `G`.
- `induce`

  Use `induce(G,A)` to create the induced subgraph of `G` on vertex set `A`.
- `spanning_forest`

  Given a graph, this creates a maximal, acyclic, spanning subgraph. If the original graph is connected, this produces a spanning tree.
- `cartesian`

  Use `cartesian(G,H)` to compute the Cartesian product $G \times H$ of G and H. For example, to create a grid graph, do this: `cartesian(Path(n), Path(m))`.

  Note that `G*H` is equivalent to `cartesian(G,H)`.
- `relabel`

  Create a new graph, isomorphic to the old graph, in which the vertices have been renamed. Use `relabel(G,label)` where `G` is a simple graph and `labels` is a dictionary mapping vertices in `G` to new names. Trouble ensues if two vertices are mapped to the same label (we don't check).

  Note that if the vertex type of `G` is `S`, then `label` must be of type `Dict{S,T}`. The new graph produced with have type `SimpleGraph{T}`.

  Calling this with one argument, `relabel(G)`, will produce a relabeled version of `G` using consecutive integers starting with 1.
- `disjoint_union`

  The disjoint union of two graphs is formed by taking nonoverlapping copies of the two graphs and merging them into a single graph (with no additional edges). In Julia, we do this by appending the intger 1 or 2 to the vertex names. Thus, if a vertex of the first graph is `"alpha"`, then in the disjoint union its name will be `("alpha",1)`.

Use `disjoint_union(G,H)` to form the disjoint union. If the vertex types of the two graphs are both `T`, then the vertex type of the result is type `(T,Int)`. But if the two graphs have different vertex types, then the result has vertex type `Any`.

We append a 1 or a 2 to vertex names to ensure that we have two independent copies of the graphs. If the user knows that the two graphs have no vertices in common, then `union` might be a preferrable choice.

- `join`

  The join of two graphs is formed by taking nonoverlapping copies of the two graphs and then adding all possible edges between the two copies. To ensure the two copies of the given graphs are on distinct vertex sets, we append a 1 or a 2 to the vertex names (see the description for `disjoint_union`).

  Use `join(G,H)` to form the join. If the two graphs have the same vertex type `T`, then the result has vertex type `(T,Int)`. Otherwise, the resulting graph has vertex type `Any`.
- `union`

  Given graphs G and H, the union has vertex set $V(G) \cup V(H)$ and edge set $E(G) \cup E(H)$.
- `trim`

  Trimming a graph means to repeatedly remove vertices of a given degree d or less until either all vertices have been removed or the remaining vertices induce a subgraph all of whose vertices have degree greater than d.

  Use `trim(G,d)` to trim the graph, with `trim(G)` equivalent to `trim(G,0)`. The latter simply removes all isolated vertices.

Manipulators and inspectors. These are functions that are used to modify a graph and to inspect its structure.

- `isequal`

  Test two graphs to see if they are the same; that is, the graphs must have equal vertex and edge sets. They need not be the same object. While this can be invoked as `isequal(G,H)` it is more convenient to use `G==H`.

  Note: If the vertex type of either graph cannot be sorted by < then equality testing is slower (unless fast neighhborhood lookup is engaged).
- `add!`

  Use this to add vertices or edges to a graph. The syntax `add!(G,v)` adds a vertex and calling `add!(G,v,w)` adds an edge.

  These return `true` if the operation succeeded in adding a *new* vertex or edge.
- `delete!`

  Use this to delete vertices or edges from a graph. The syntax `delete!(G,v)` to delete a vertex (and all edges incident thereon) and `delete!(G,v,w)` to delete an edge.

  Returns `true` if successful. If the vertex or edge slated for removal was not in the graph, returns `false`.
- `contract!`

  Mutates a graph by contractin an edge. Calling `contract!(G,u,v)` adds all vertices in `v`'s neighborhood to `u`'s neighborhood and then deletes vertex `v`. Typically `(u,v)` is an edge of the graph but this is not necessary.

  This returns `true` is the operation is successful, but `false` if either `u` or `v` is not a vertex of the graph or if `u==v`.
- `has`

  Test for the presence of a vertex of edge. Use `has(G,v)` to test if `v` is a vertex of the graph and `has(G,v,w)` to test if the edge is present. Returns `true` if so and `false` if not.

  Note that `G[v,w]` is a synonym for `has(G,v,w)`.
- `vlist`

Use `vlist(G)` to get the vertex set of the graph as a one-dimensional array. If possible, the vertices are sorted in ascending order.

- `elist`

  Use `elist(G)` to get the edge set of the graph as a one-dimensional array of 2-tuples. If possible, the edges are sorted in ascending lexicographic order.
- `neighbors`

  Use `neighbors(G,v)` to get the set of neighbors of vertex `v` as a one-dimensional array.

  Note that `G[v]` is a synonym.
- `deg`

  Use `deg(G,v)` to get the degree of vertex `v` and `deg(G)` to get the degree sequence of G as a one-dimensional array (in decreasing order).
- `fastN!`

  This is explained in §2.

  Use this to switch on `fastN!(G,true)` or to switch off `fastN!(G,false)` rapid neighborhood lookup. If off, neighborhood lookup can be slow. If on, the data structure supporting the graph is roughly tripled in size.

  The difference is especially striking when looking for paths between vertices with `find_path`.
- `NV` and `NE`

  Use `NV(G)` to get the number of vertices and `NE(G)` to get the number of edges.
- `is_connected`

  Use `is_connected(G)` to determine if the graph is connected.
- `num_components`

  Use `num_components(G)` to determine the number of connected components in the graph.
- `components`

  The function `components(G)` determines the vertex sets of the connected components of the graph. The return value is a set of sets. That is, if the graph has vertex type `T`, then this function produces an object of type `Set{Set{T}}`.

  If what one wants is a *list* of *subgraphs* of a graph that are the connected components of the graph, do this:

```
[ induce(G,A) for A in components(G) ]
```

  Alternatively, if one wants a list of lists, do this:

```
[ collect(A) for A in components(G) ]
```

- `find_path`

  The function `find_path(G,u,v)` finds a shortest path from `u` to `v` if one exists. An empty array is returned if there is no such path. An error is raised if either vertex is absent from the graph.
- `dist` and `dist_matrix`

  These are used to find distances between vertices in a graph. The distance between vertices u and v is defined to be the number of edges in a shortest (u, v)-path. If there is no such path, one typically says that $d(u, v)$ is undefined of ∞. However since these functions report distances as `Int` values, we signal the absence of a (u, v) path by the value -1.

  Use `dist(G,v,w)` to find the distance between the specified vertices in the graph.

  Use `dist(G,v)` to find the distances from vertex `v` to all vertices in the grpah. This is returned as a `Dict`.

  Use `dist(G)` to find all distances between all vertices in the graph. For example:

```
julia> G = Cycle(10)
SimpleGraph{Int64} (10 vertices, 10 edges)

julia> d = dist(G);

julia> d[(3,9)]
4
```

The function `dist_matrix` creates an $n \times n$-matrix whose i, j-entry is the distance between the i^{th} and j^{th} vertex of the graph where the order is produced by `vlist`.

- `diam`
  Compute the diameter of a graph. Note that `diam(G)` returns -1 if the graph is not connected.
- `is_cut_edge`
  Determine if a given edge is a cut edge. This can be called either as `is_cut_edge(G,u,v)` where `u` and `v` are vertices or as `is_cut_edge(G,e)` where `e` is an edge (i.e., a 2-tuple of vertices.
  If `(u,v)` is not an edge of the graph, an error is raised.
- `euler`
  This is used to find Eulerian trails and tours in a graph. Typical call is `euler(G,u,v)` to find an Eulerian trail starting at `u` and ending at `v`. The first element of that arrary is `u` and the last is `v`. If a trail is found, the length of the array is `NE(G)+1`. Otherwise, an empty array is returned.
  The graph may have isolated vertices, and these are ignored.
  The call `euler(G,u)` is shorthand for `euler(G,u,u)`. A simple call to `euler(G)` will attempt to find an Euler tour from some vertex in the graph.
  If the graph is edgeless, then `euler(G,u)` and `euler(G,u,u)` return the 1-element array `[u]`. Calling `euler(G)` will pick `u` for you. An empty array is returned if the graph has no vertices. (This is mildly unfortunate as an empty array indicates failure to find a trail for nonempty graphs.)
- `bipartition` and `two_color`
  Used to find a bipartition or two-coloring of a graph if the graph is bipartite; otherwise, return an error. The function `bipartition` returns a two-element `Set` containing the two parts (themselves `Set` objects). The function `two_color` returns a map (`Dict`) from the vertex set to the set $\{1,2\}$.

```
julia> G = Cycle(6)
SimpleGraph{Int64} (6 vertices, 6 edges)

julia> two_color(G)
[5=>1,4=>2,6=>2,2=>2,3=>1,1=>1]

julia> bipartition(G)
Set{Set{Int64}}(Set{Int64}(5,3,1),Set{Int64}(4,6,2))
```

- `greedy_color` and `random_greedy_color`
  This is a simple graph coloring function. Given an ordering of the vertices of the graph, `greedy_color` creates a proper, greedy coloring. If the ordering is not provided, then a degree-decreasing ordering is given. Use `greedy_color(G,seq)` where `seq` is a permutation of the vertex set (if you wish to specify the order) or simply `greedy_color(G)` in which case a degree-decreasing ordering is used.
  The second function performs multiple greedy colorings on random orderings of the vertex set. Use `random_greedy_color(G,nreps)` where `nreps` is the number of random orders generated.
  In all cases a `Dict` is returned that maps the vertex set to a range of the form `[1:k]`.

Graph matrices. These functions return standard matrices associated with graphs.

- `adjacency`
  Use `adjacency(G)` to return the adjacency matrix of the graph.
- `laplace`
  Use `laplace(G)` to return the Laplacian matrix of the graph.
- `incidence`
  Use `incidence(G)` to return the signed incidence matrix of the graph. This is equivalent to `incidence(G,true)`. Calling `incidence(G,false)` returns the unsigned incidence matrix.

Assignment of $+1$ and -1 in each column tries to put the $+1$ on the vertex that sorts lower than the vertex that gets a -1. If the vertices are not comparable by $<$ (less than), the assignment is unpredictable.

Note that `incidence` returns a sparse matrix. Use `full(incidence(G))` if a full-storage matrix is desired.

Converting. This is explained in §5.

- `convert_simple`

  Use `convert_simple(G)` to create a Julia `Graphs.simple_graph` version of a graph, together with dictionaries to translate between one vertex set and the other.

4. Errors and gotchas

Errors raised. The functions in the `SimpleGraphs` module generally do not raise errors. Function such as `add!` and `delete!` return `false` if the graph is not changed by the requested modification.

However, there are some instances in which an error might be raised.

- An error is raised if one attempts to add a vertex of a type that is incompatible with the vertex type of the graph. Here's an example:

```
julia> G = StringGraph()
SimpleGraph{ASCIIString} (0 vertices, 0 edges)

julia> add!(G,4)
ERROR: no method convert(Type{ASCIIString},Int64)
```

- An error is raised if one attempts to find the neighborhood or the degree of a vertex that is not in the graph. Here's an example:

```
julia> G = Complete(5)
SimpleGraph{Int64} (5 vertices, 10 edges)

julia> G[6]
ERROR: Graph does not contain requested vertex
```

- An error is raised if one attempts to create a cycle with fewer than three vertices.

Of course, code can be wrapped in a `try`-`catch` block to handle these possibilities gracefully.

Please specify the vertex type. Although `G=SimpleGraph()` allows `Any` type vertex, we recommend specifying the type of vertex desired. Here's why.

Because we do not allow loops, the code for `add!(G,v,w)` first checks if `v==w` and if so, does not add the edge and returns the value `false`. So far so good.

Internally, the vertices of the graph are held in a Julia `Set` container. Now `Set` either does or does not contain a given object; the object cannot be in the `Set` twice. Where this gets us into a bit of trouble is that the integer value `1` and the floating point value `1.0` are different objects and therefore may cohabit the same `Set`. Here's an illustration:

```
julia> A = Set()
Set{Any}()

julia> push!(A,1)
Set{Any}(1)

julia> push!(A,1.0)
Set{Any}(1.0,1)
```

The implication of this is that the vertex set of a `SimpleGraph` might contain both the integer `1` and the floating point number `1.0`. However, we cannot add an edge between these two vertices because the test `1==1.0` returns `true`.

Here's how this plays out:

```
julia> G = SimpleGraph()
SimpleGraph{Any} (0 vertices, 0 edges)

julia> add!(G,1)
true

julia> has(G,1.0)
false

julia> add!(G,1.0)
true

julia> add!(G,1,1.0)
false
```

In principle, we could fix this problem by using a more liberal filter in `add!(G,v,w)` that allows the addition of an edge with `v==w` provided `typeof(v)` and `typeof(w)` are different. But that would not entirely solve the problem.

Consider this example:

```
julia> G = SimpleGraph()
SimpleGraph{Any} (0 vertices, 0 edges)

julia> add!(G,1)
true

julia> add!(G,BigInt(1))
true

julia> vlist(G)
2-element Array{Any,1}:
 1
 1
```

It appears that the number 1 is in the vertex set twice!

The preferred solution is to avoid using `G=SimpleGraph()` and, instead, to use `G=SimpleGraph{T}()` where `T` is the data type of the vertices. The ready-to-use `IntGraph` and `StringGraph` are handy for these popular vertex types.

```
julia> G = IntGraph()
SimpleGraph{Int64} (0 vertices, 0 edges)

julia> add!(G,1)
true

julia> add!(G,BigInt(1))
ERROR: 1 is not a valid key for type Int64

julia> add!(G,1.0)
ERROR: 1.0 is not a valid key for type Int64
```

Unsortable vertex types. Edges in a `SimpleGraph` are held as a tuple. If the end points of the edge can be compared using the < operator, then the smaller end point comes first in the pair. Otherwise, the order is arbitrary. In the latter case, graph equalty checking is slower.

In general, it's best to specify vertex types for graphs, preferring `SimpleGraph{T}()` for some type `T` that supports < comparison.

5. Convert to Graphs.jl

The `Graphs` module defined in `Graphs.jl` is another tool for dealing with graphs. We provide the function `convert_simple` to convert a graph from a `SimpleGraph` to a `simple_graph` type graph from the `Graphs` module distributed with Julia.

A `simple_graph`'s vertex set is always of the form `1:n`, so the output of `convert_simple` provides two dictionaries for mapping from the vertex set of the `SimpleGraph` to the vertex set of the `simple_graph`, and back again.