Sage Quick Reference: Graph Theory

Steven Rafael Turner

Sage Version 4.7

http://wiki.sagemath.org/quickref

Constructing

Adjacency Mapping:

`G=Graph([GF(13), lambda i,j: conditions on i,j])`

Input is a list whose first item are vertices and the other is some adjacency function: [list of vertices, function]

Adjacency Lists:

```
G=Graph({0:[1,2,3], 2:[4]})
G=Graph({0:{1:"x",2:"z",3:"a"}, 2:{5:"out"}})
```

x, z, a, and out are labels for edges and be used as weights.

Adjacency Matrix:

`A = numpy.array([[0,1,1],[1,0,1],[1,1,0]])`

Don't forget to import numpy for the NumPy matrix or ndarray.

`M = Matrix([(....), (....), . . . ])`

Edge List with or without labels:

`G = Graph([(1,3,"Label"),(3,8,"Or"),(5,2)])`

Incidence Matrix:

`M = Matrix(2, [-1,0,0,0,1, 1,-1,0,0,0])`

Graph6 Or Sparse6 string

```
G=':IgMoqoCUOqeb\n:I`EDOAEQ?PccSsge\N\n'
graphs_list.from_sparse6(G)
```

Above is a list of graphs using sparse6 strings.

NetworkX Graph

```
g = networkx.Graph({0:[1,2,3], 2:[4]})
DiGraph(g)
g_2 = networkx.MultiGraph({0:[1,2,3], 2:[4]})
Graph(g_2)
```

Don't forget to import networkx

Centrality Measures

```
G.centrality_betweenness(normalized=False)
G.centrality_closeness(v=1)
G.centrality_degree()
```

Graph Deletions and Additions

```
G.add_cycle([vertices])
G.add_edge(edge)
G.add_edges(iterable of edges)
G.add_path
G.add_vertex(Name of isolated vertex)
G.add_vertices(iterable of vertices)
G.delete_edge( v_1, v_2, 'label')
G.delete_edges(iterable of edges)
G.delete_multiedge(v_1, v_2)
G.delete_vertex(v_1)
G.delete_vertices(iterable of vertices)
G.merge_vertices([vertices])
```

Connectivity and Cuts

```
G.is_connected()
G.edge_connectivity()
G.edge_cut(source, sink
G.blocks_and_cut_vertices()
G.max_cut()
G.edge_disjoint_paths(v1,v2, method='LP')
```

This method can us LP (Linear Programming) or FF (Ford-Fulkerson)

```
vertex_disjoint_paths(v1,v2)
G.flow(1,2)
```

There are many options to this function please check the documentation.

Conversions

```
G.to_directed()
G.to_undirected()
G.sparse6_string()
G.graph6_string()
```

Products

```
G.strong_product(H)
G.tensor_product(H)
G.categorical_product(H)
```

Same as the tensor product.

```
G.disjunctive_product(H)
G.lexicographic_product(H)
G.cartesian_product(H)
```

Boolean Queries

```
G.is_tree()
G.is_forest()
G.is_gallai_tree()
G.is_interval()
G.is_regular()
G.is_chordal()
G.is_eulerian()
G.is_hamiltonian()
G.is_interval()
G.is_independent_set([vertices])
G.is_overfull()
G.is_regular(k)
```

Can test for being k-regular, by default k=None.

Common Invariants

```
G.diameter()
G.average_distance()
G.edge_disjoint_spanning_trees(k)
G.girth()
G.size()
G.order()
G.radius()
```

Graph Coloring

```
G.chromatic_polynomial()
G.chromatic_number(algorithm="DLX")
```

You can change DLX (dancing links) to CP (chromatic polynomial coefficients) or MILP (mixed integer linear program)

`G.coloring(algorithm="DLX")`

You can change DLX to MILP

`G.is_perfect(certificate=False)`

Planarity

```
G.is_planar()
G.is_circular_planar()
G.is_drawn_free_of_edge_crossings()
G.layout_planar(test=True, set_embedding=True
G.set_planar_positions()
```

Search and Shortest Path

```
list(G.depth_first_search([vertices], distance=4)
list(G.breadth_first_search([vertices])
dist,pred = graph.shortest_path_all_pairs(by_weig
```

Choice of algorithms: BFS or Floyd-Warshall-Python

```
G.shortest_path_length(v_1,v_2, by_weight=True
G.shortest_path_lengths(v_1)
G.shortest_path(v_1,v_2)
```

Spanning Trees

`G.steiner_tree(g.vertices()[:10])`

`G.spanning_trees_count()`
`G.edge_disjoint_spanning_trees(2, root vertex)`
`G.min_spanning_tree(weight_function=somefunction, algorithm='Kruskal',starting_vertex=3)`
Kruskal can be change to Prim_fringe, Prim_edge, or NetworkX

Linear Algebra

Matrices

`G.kirchhoff_matrix()`
`G.laplacian_matrix()`
Same as the kirchoff matrix
`G.weighted_adjacency_matrix()`
`G.adjacency_matrix()`
`G.incidence_matrix()`

Operations

`G.characteristic_polynomial()`
`G.cycle_basis()`
`G.spectrum()`
`G.eigenspaces(laplacian=True)`
`G.eigenvectors(laplacian=True)`

Automorphism and Isomorphism Related

`G.automorphism_group()`
`G.is_isomorphic(H)`
`G.is_vertex_transitive()`
`G.canonical_label()`
`G.minor(graph of minor to find)`

Generic Clustering

`G.cluster_transitivity()`
`G.cluster_triangles()`
`G.clustering_average()`
`G..clustering_coeff(nbunch=[0,1,2],weights=True)`

Clique Analysis

`G.is_clique([vertices])`
`G.cliques_vertex_clique_number(vertices=[(0, 1), (1, 2)],algorithm="networkx")`
networkx can be replaced with cliquer.
`G.cliques_number_of()`
`G.cliques_maximum()`
`G.cliques_maximal()`
`G.cliques_get_max_clique_graph()`
`G.cliques_get_clique_bipartite()`
`G.cliques_containing_vertex()`
`G.clique_number(algorithm="cliquer")`
cliquer can be replaced with networkx.
`G.clique_maximum()`
`G.clique_complex()`

Component Algorithms

`G.is_connected()`
`G.connected_component_containing_vertex(vertex)`
`G.connected_components_number()`
`G.connected_components_subgraphs()`
`G.strong_orientation()`
`G.strongly_connected_components()`
`G.strongly_connected_components_digraph()`
`G.strongly_connected_components_subgraphs()`
`G.strongly_connected_component_containing_vertex(vertex)`
`G.is_strongly_connected()`

NP Problems

`G.vertex_cover(algorithm='Cliquer')`
The algorithm can be changed to MILP (mixed integer linear program. Note that MILP requires packages GLPK or CBC.
`G.hamiltonian_cycle()`
`G.traveling_salesman_problem()`