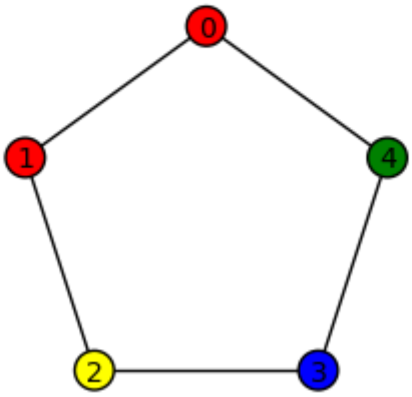
graph_dynamics_examples

```
attach("/Users/raichev/graph_dynamics/graph_dynamics.py")
# attach "/Users/arai021/graph_dynamics/graph_dynamics.py"

# For a list of Sage's graph generators, see
http://wiki.sagemath.org/graph_generators.
```

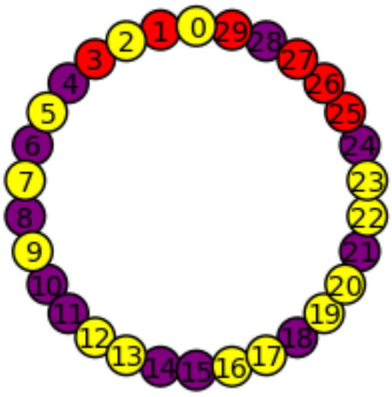
```
# Example: Using color()

G = graphs.CycleGraph(5)
coloring = color(G, ['red', 'red', 'yellow', 'blue', 'green'])
print(coloring)
G.show(vertex_colors=invert_dict(coloring), figsize=3)
```

```
{0: 'red', 1: 'red', 2: 'yellow', 3: 'blue', 4: 'green'}
```



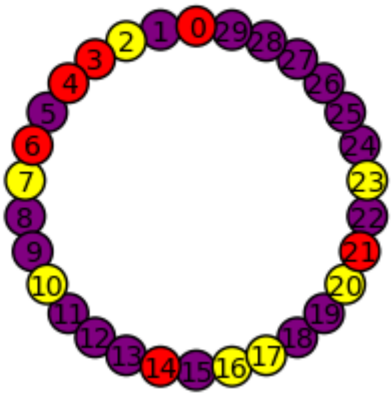

```
# Example: Using color_randomly()

G = graphs.CycleGraph(30)
coloring = color_randomly(G, ['red', 'yellow', 'purple'])
print(coloring)
G.show(vertex_colors=invert_dict(coloring), figsize=3)
```

```
{0: 'yellow', 1: 'red', 2: 'yellow', 3: 'red', 4: 'purple', 5
'yellow', 6: 'purple', 7: 'yellow', 8: 'purple', 9: 'yellow',
'purple', 11: 'purple', 12: 'yellow', 13: 'yellow', 14: 'purp
15: 'purple', 16: 'yellow', 17: 'yellow', 18: 'purple', 19:
'yellow', 20: 'yellow', 21: 'purple', 22: 'yellow', 23: 'yell
24: 'purple', 25: 'red', 26: 'red', 27: 'red', 28: 'purple',
'red'}
```




```
# Example: Using color_count()

G = graphs.CycleGraph(30)
coloring = color_randomly(G, ['red', 'yellow', 'purple'])
print(coloring)
G.show(vertex_colors=invert_dict(coloring), figsize=3)
print(color_count(coloring))
```

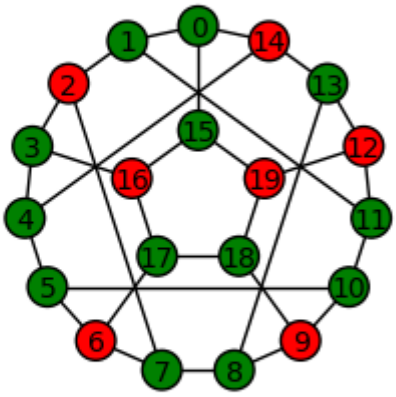
```
{0: 'red', 1: 'purple', 2: 'yellow', 3: 'red', 4: 'red', 5:
'purple', 6: 'red', 7: 'yellow', 8: 'purple', 9: 'purple', 1(
'yellow', 11: 'purple', 12: 'purple', 13: 'purple', 14: 'red'
'purple', 16: 'yellow', 17: 'yellow', 18: 'purple', 19: 'purp
20: 'yellow', 21: 'red', 22: 'purple', 23: 'yellow', 24: 'pur
25: 'purple', 26: 'purple', 27: 'purple', 28: 'purple', 29:
'purple'}
```



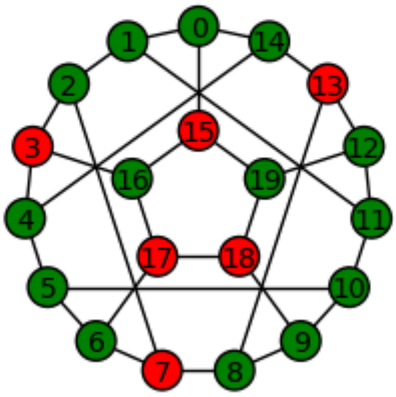

```
Counter({'purple': 17, 'yellow': 7, 'red': 6})
```

```
# Example: Iterating the majority rule

G = graphs.FlowerSnark()
color_palette = ['green', 'red']
ur = majority_rule
ur_kwargs = {}
initial_coloring = color_randomly(G, color_palette)
s, stabilized = iterate(ur, ur_kwargs, G, initial_coloring)
print('Stabilized?\n    %s' % stabilized)
print(s)
show_colorings(G, s, vertex_labels=True)
```

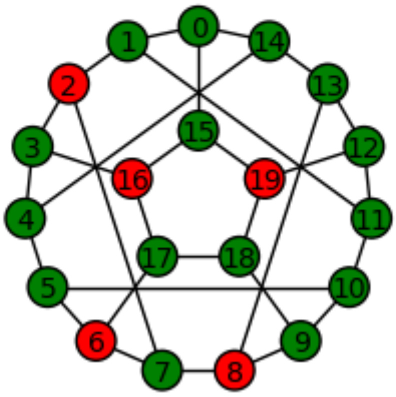
```
Stabilized?
    False
[{0: 'green', 1: 'green', 2: 'red', 3: 'green', 4: 'green', 5
'green', 6: 'red', 7: 'green', 8: 'green', 9: 'red', 10: 'gre
11: 'green', 12: 'red', 13: 'green', 14: 'red', 15: 'green',
'red', 17: 'green', 18: 'green', 19: 'red'}, {0: 'green', 1:
'green', 2: 'green', 3: 'red', 4: 'green', 5: 'green', 6: 'gr
7: 'red', 8: 'green', 9: 'green', 10: 'green', 11: 'green', 1
'green', 13: 'red', 14: 'green', 15: 'red', 16: 'green', 17:
18: 'red', 19: 'green'}, {0: 'green', 1: 'green', 2: 'red', 3
'green', 4: 'green', 5: 'green', 6: 'red', 7: 'green', 8: 're
'green', 10: 'green', 11: 'green', 12: 'green', 13: 'green',
'green', 15: 'green', 16: 'red', 17: 'green', 18: 'green', 19
'red'}, {0: 'green', 1: 'green', 2: 'green', 3: 'red', 4: 'gr
5: 'green', 6: 'green', 7: 'red', 8: 'green', 9: 'green', 10:
'green', 11: 'green', 12: 'green', 13: 'green', 14: 'green',
'red', 16: 'green', 17: 'red', 18: 'green', 19: 'green'}, {0:
'green', 1: 'green', 2: 'red', 3: 'green', 4: 'green', 5: 'gr
6: 'red', 7: 'green', 8: 'green', 9: 'green', 10: 'green', 11
'green', 12: 'green', 13: 'green', 14: 'green', 15: 'green',
'red', 17: 'green', 18: 'green', 19: 'green'}, {0: 'green', 1
'green', 2: 'green', 3: 'red', 4: 'green', 5: 'green', 6: 'gr
7: 'red', 8: 'green', 9: 'green', 10: 'green', 11: 'green', 1
'green', 13: 'green', 14: 'green', 15: 'green', 16: 'green',
'red', 18: 'green', 19: 'green'}, {0: 'green', 1: 'green', 2:
3: 'green', 4: 'green', 5: 'green', 6: 'red', 7: 'green', 8:
'green', 9: 'green', 10: 'green', 11: 'green', 12: 'green', 1
'green', 14: 'green', 15: 'green', 16: 'red', 17: 'green', 18
'green', 19: 'green'}, {0: 'green', 1: 'green', 2: 'green', 3
'red', 4: 'green', 5: 'green', 6: 'green', 7: 'red', 8: 'gree
'green', 10: 'green', 11: 'green', 12: 'green', 13: 'green',
'green', 15: 'green', 16: 'green', 17: 'red', 18: 'green', 19
'green'}, {0: 'green', 1: 'green', 2: 'red', 3: 'green', 4: '
5: 'green', 6: 'red', 7: 'green', 8: 'green', 9: 'green', 10:
'green', 11: 'green', 12: 'green', 13: 'green', 14: 'green',
'green', 16: 'red', 17: 'green', 18: 'green', 19: 'green'}, {
'green', 1: 'green', 2: 'green', 3: 'red', 4: 'green', 5: 'gr
6: 'green', 7: 'red', 8: 'green', 9: 'green', 10: 'green', 11
'green', 12: 'green', 13: 'green', 14: 'green', 15: 'green',
'green', 17: 'red', 18: 'green', 19: 'green'}, {0: 'green', 1
'green', 2: 'red', 3: 'green', 4: 'green', 5: 'green', 6: 're
'green', 8: 'green', 9: 'green', 10: 'green', 11: 'green', 12
'green', 13: 'green', 14: 'green', 15: 'green', 16: 'red', 17
'green', 18: 'green', 19: 'green'}]
Step 0
```



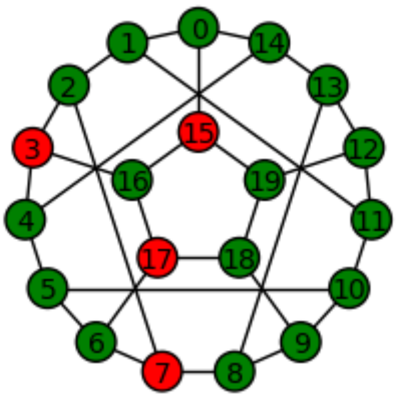
Step 1



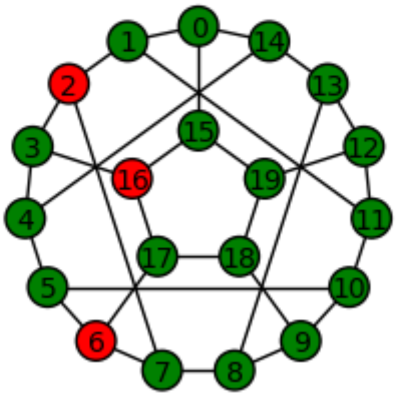
Step 2



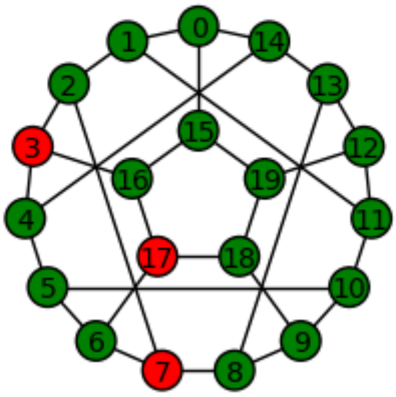
Step 3



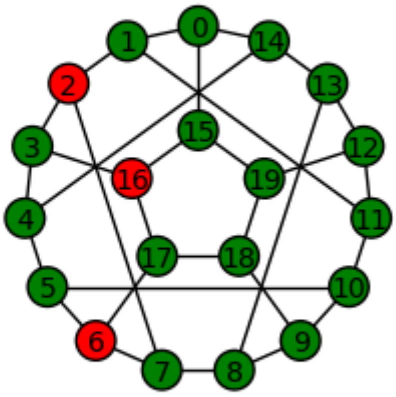
Step 4



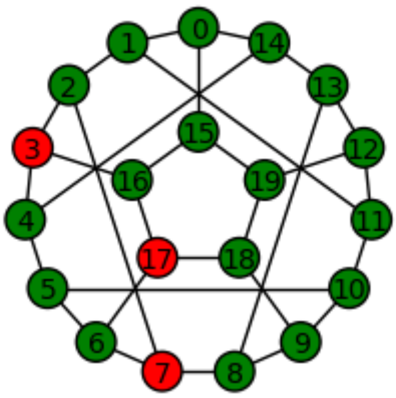
Step 5



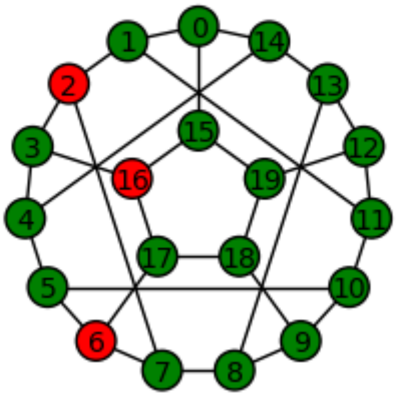
Step 6



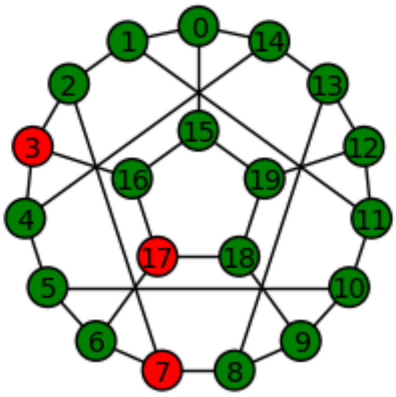
Step 7



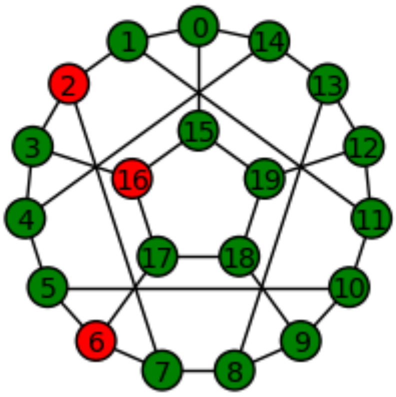
Step 8



Step 9



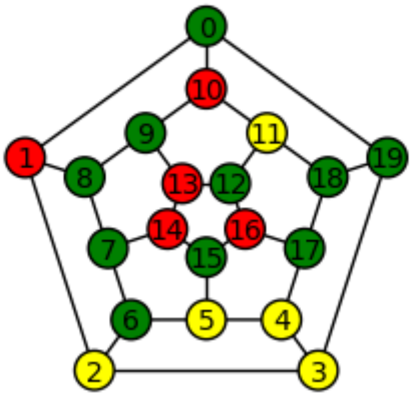
Step 10



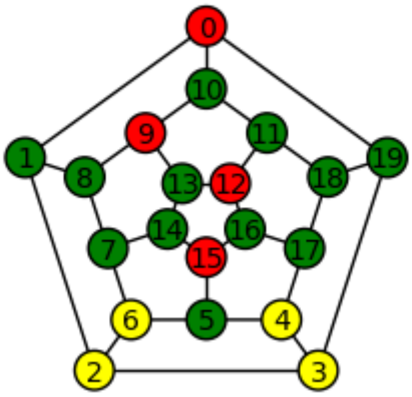

```
# Example: Iterating the plurality rule.

G = graphs.DodecahedralGraph()
ur = plurality_rule
ur_kwargs = {}
color_palette = ['green', 'red', 'yellow']
initial_coloring = color_randomly(G, color_palette)

s, stabilized = iterate(ur, ur_kwargs, G, initial_coloring)
print('Stabilized?\n    %s' % stabilized)
show_colorings(G, s, vertex_labels=True)
```

```
Stabilized?
    True
```

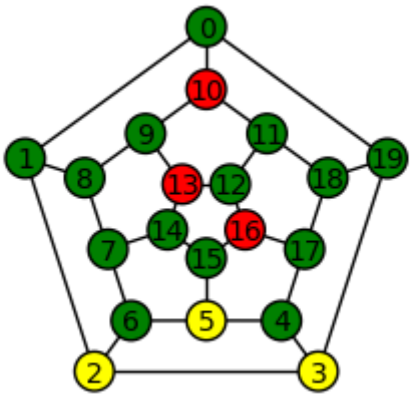
```
Step 0
```



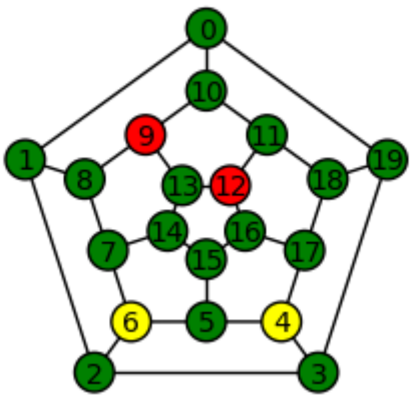

```
Step 1
```



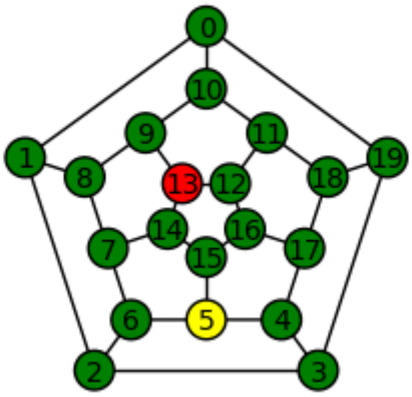

```
Step 2
```



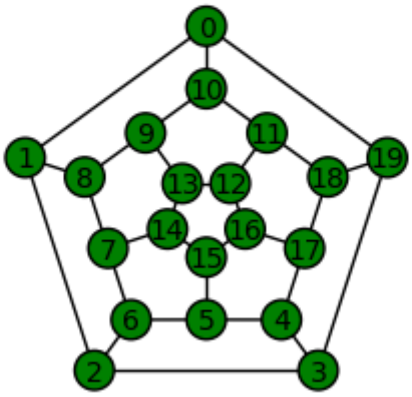

```
Step 3
```



```
Step 4
```

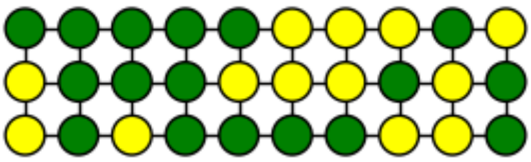



```
Step 5
```

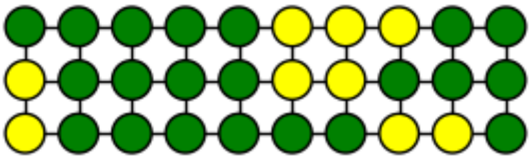



```
# Example: Iterating the GSL2 rule.

G = graphs.Grid2dGraph(3, 10)
color_palette = ['green', 'yellow']
ur = gsl2_rule
ur_kwargs = {'color_palette': color_palette, 'T': 0.7}
initial_coloring = color_randomly(G, color_palette)

s, stabilized = iterate(ur, ur_kwargs, G, initial_coloring)
print('Stabilized?\n    %s' % stabilized)
show_colorings(G, s)
```

```
Stabilized?
    True
Step 0
```

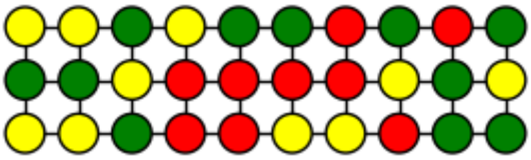


```
Step 1
```

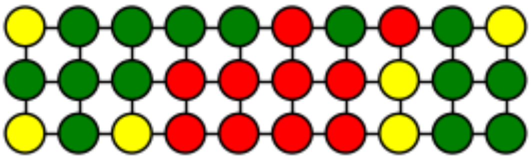


```
# Example: Iterating the GSL3 rule.

G = graphs.Grid2dGraph(3, 10)
color_palette = ['green', 'red', 'yellow']
ur = gsl3_rule
ur_kwargs = {'color_palette': color_palette, 'T': 0.6}
initial_coloring = color_randomly(G, color_palette)

s, stabilized = iterate(ur, ur_kwargs, G, initial_coloring)
print('Stabilized?\n    %s' % stabilized)
show_colorings(G, s)
```

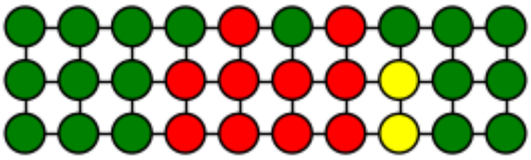
```
Stabilized?
    False
Step 0
```



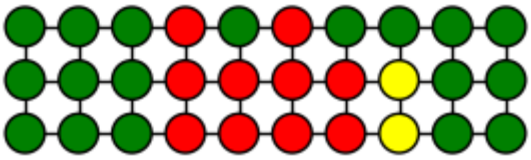
Step 1



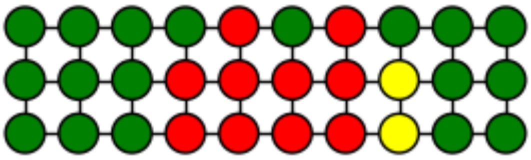
Step 2



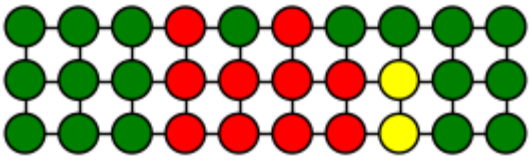
Step 3



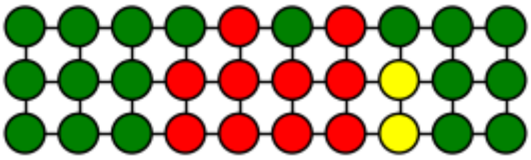
Step 4



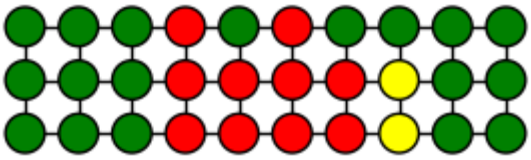
Step 5



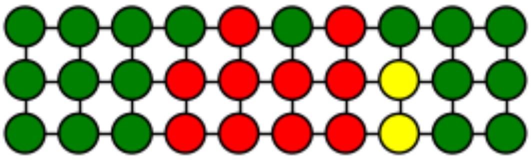
Step 6



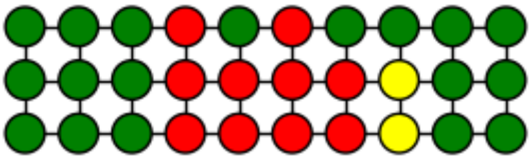
Step 7



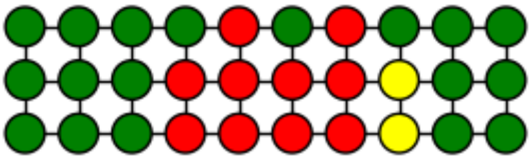
Step 8



Step 9

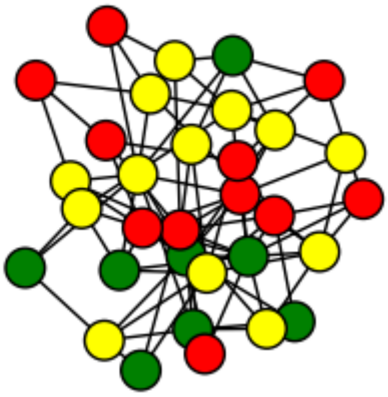


Step 10

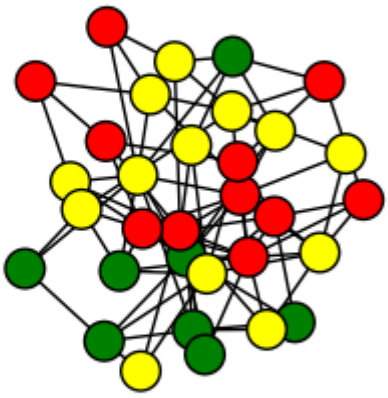


```
# Example: Iterating the GSL3 rule on a random graph

gg = graphs.RandomBarabasiAlbert
print(gg)
G = gg(32, 3)
color_palette = ['green', 'red', 'yellow']
ur = gsl3_rule
ur_kwargs = {'color_palette': color_palette, 'T': 0.6}
initial_coloring = color_randomly(G, color_palette)

s, stabilized = iterate(ur, ur_kwargs, G, initial_coloring)
print('Stabilized?\n    %s' % stabilized)
show_colorings(G, s)
```

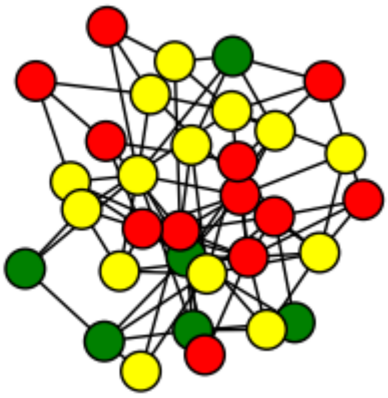
```
<function RandomBarabasiAlbert at 0x10e2cb938>
Stabilized?
    True
Step 0
```



```
Step 1
```



```
Step 2
```



```
graphs.Grid2dGraph??
```

File: /Applications/sage/local/lib/python2.7/site-packages/sage/graphs/generators/basic.py

Source Code (starting at line 1003):

```
def Grid2dGraph(n1, n2):
    r"""
    Returns a `2`-dimensional grid graph with `n_1n_2` nodes (`n_1` ro
    `n_2` columns).
```

```
    A 2d grid graph resembles a `2` dimensional grid. All inner nodes
    connected to their `4` neighbors. Outer (non-corner) nodes are
    connected to their `3` neighbors. Corner nodes are connected to th
    2 neighbors.

    This constructor depends on NetworkX numeric labels.

    PLOTTING: Upon construction, the position dictionary is filled to
    override the spring-layout algorithm. By convention, nodes are
    labelled in (row, column) pairs with `(0, 0)` in the top left corn
    Edges will always be horizontal and vertical - another advantage o
    filling the position dictionary.

    EXAMPLES: Construct and show a grid 2d graph Rows = `5`, Columns =

    ::

        sage: g = graphs.Grid2dGraph(5,7)
        sage: g.show() # long time

    TESTS:

    Senseless input::

        sage: graphs.Grid2dGraph(5,0)
Traceback (click to the left of this block for traceback)
...
```

```
# Example: Using get_stats() on a fixed graph

color_palette = ['green', 'red', 'yellow']
ur = gsl3_rule
urk = {'color_palette': color_palette}
gg = graphs.Grid2dGraph
ggk = {'n1': 3, 'n2': 10}
cf = color_randomly
cfk = {'color_palette': color_palette}

num_runs = 1000
num_stabilized, mean_steps, mean_initial, mean_final =
get_stats(ur, urk, gg, ggk, cf, cfk, num_runs=num_runs)

print(ur)
print(gg)
print(cf)
print('-'*40)
print('Number of runs: %s' % num_runs)
print('Number of runs that stabilized: %s' % num_stabilized)
print('Mean number of steps required to stabilize: %s' %
mean_steps)
print('Mean initial color counts: %s' % mean_initial)
print('Mean finial color counts: %s' % mean_final)
```

```
Traceback (click to the left of this block for traceback)
...
SyntaxError: invalid syntax
```

```
graphs.RandomBarabasiAlbert??
```

File: /Applications/sage/local/lib/python2.7/site-packages/sage/graphs/generators/random.py

Source Code (starting at line 135):

```
def RandomBarabasiAlbert(n, m, seed=None):
    u"""
    Return a random graph created using the Barabasi-Albert preferenti
    attachment model.

    A graph with m vertices and no edges is initialized, and a graph o
    vertices is grown by attaching new vertices each with m edges that
    attached to existing vertices, preferentially with high degree.

    INPUT:

    - ``n`` - number of vertices in the graph

    - ``m`` - number of edges to attach from each new node

    - ``seed`` - for random number generator

    EXAMPLES:

    We show the edge list of a random graph on 6 nodes with m = 2.

    ::

        sage: graphs.RandomBarabasiAlbert(6,2).edges(labels=False)
        [(0, 2), (0, 3), (0, 4), (1, 2), (2, 3), (2, 4), (2, 5), (3, 5

    We plot a random graph on 12 nodes with m = 3.

    ::

        sage: ba = graphs.RandomBarabasiAlbert(12,3)
        sage: ba.show()  # long time

    We view many random graphs using a graphics array::

        sage: g = []
        sage: j = []
        sage: for i in range(1,10):
        ...     k = graphs.RandomBarabasiAlbert(i+3, 3)
        ...     g.append(k)
        ...
        sage: for i in range(3):
        ...     n = []
        ...     for m in range(3):
        ...         n.append(g[3*i + m].plot(vertex_size=50, vertex_lab
        ...     j.append(n)
        ...
        sage: G = sage.plot.graphics.GraphicsArray(j)
        sage: G.show()  # long time

    """
    if seed is None:
```

```
        seed = current_randstate().long_seed()
    import networkx
    return graph.Graph(networkx.barabasi_albert_graph(n,m,seed=seed))
```

```
# Example: Using get_stats() on a random graph

color_palette = ['green', 'red', 'yellow']
ur = gsl3_rule
urk = {'color_palette': color_palette}
gg = graphs.RandomBarabasiAlbert
ggk = {'n': 52, 'm': 3}
cf = color_randomly
cfk = {'color_palette': color_palette}

num_runs = 1000
num_stabilized, mean_steps, mean_initial, mean_final =
get_stats(ur, urk, gg, ggk, cf, cfk, num_runs=num_runs)

print(ur)
print(gg)
print(cf)
print('-'*20)
print('Number of runs: %s' % num_runs)
print('Number of runs that stabilized: %s' % num_stabilized)
print('Mean number of steps required to stabilize: %s' %
mean_steps)
print('Mean initial color counts: %s' % mean_initial)
print('Mean finial color counts: %s' % mean_final)
```

```
<function gsl3_rule at 0x10e221a28>
<function RandomBarabasiAlbert at 0x10e2cb938>
<function color_randomly at 0x10e221848>
--------------------
Number of runs: 1000
Number of runs that stabilized: 934
Mean number of steps required to stabilize: 5
Mean initial color counts: Counter({'green': 17, 'red': 17,
'yellow': 17})
Mean finial color counts: Counter({'red': 21, 'green': 19, 'y
11})
```