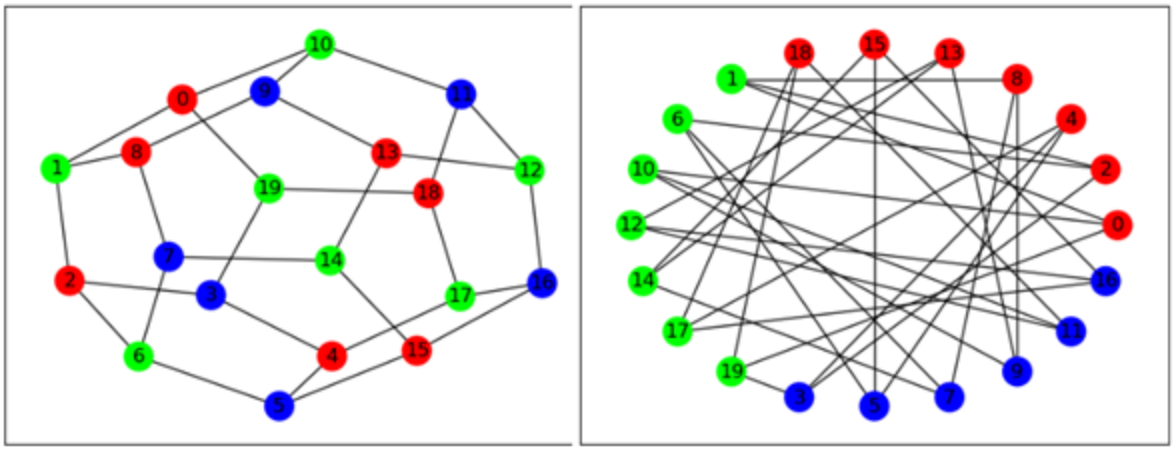
GCol

Release 0.0.4

R. Lewis

Nov 29, 2024

CONTENTS:



GCol is an open-source Python library for graph coloring that is built on top of the NetworkX package. It provides easy-to-use, high-performance algorithms for node coloring, edge coloring, equitable coloring, weighted coloring, precoloring, and maximum independent set indentification. It also offers several tools for solution visualization.

In general, graph coloring problems are NP-hard. This package therefore offers both exponential-time exact algorithms and polynomial-time heuristic algorithms.

CHAPTER

ONE

GENERAL INFO

GCol currently requires Python 3.7 or above. It also requires an istallation of NetworkX (ideally version 3.4 or above). To install the GCol library from the Python Package Index (PyPi), run the following command from a command prompt:

```
$ python pip install gcol
```

The algorithms and techniques used in this library are based on the 2021 textbook by Lewis, R. M. R. (2021) A Guide to Graph Colouring: Algorithms and Applications, Springer Cham. (2nd Edition). In bibtex, this book can be cited as:

```
@book{10.1007/978-3-030-81054-2,
  author = {Lewis, R. M. R.},
  title = {A Guide to Graph Colouring: Algorithms and Applications},
  year = {2021},
  isbn = {978-3-030-81056-6},
  publisher = {Springer Cham},
  edition = {2nd}
}
```

To start using this library, you might find it helpful to look at this *demonstration*, which gives step-by-step instructions and sample code. You may also want to consult the user guide of NetworkX, because GCol makes use of its data structures and functionality.

1.1 Support

The GCol repository is hosted on github. If you have any questions, please ask it on this dedicated question page on StackOverflow.

1.2 MIT License

OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

CHAPTER

TWO

AN INTRODUCTION TO THE GCOL LIBRARY

In this section, we demonstrate the functionality of the `gcol` library's routines using both textual and graphical output. If you have not done so already, the `gcol` library should first be installed by typing the following at the command line: `python pip install gcol`, and then restarting the kernal.

Let us first review of some basic terminology in graph theory.

- A graph is an object comprising a set of nodes that are linked by edges. They can visualized in diagram form, as shown below. Graphs are sometimes known as *networks*; nodes are sometimes called *vertices*.
- A node coloring of a graph is an assignment of colors to nodes so that all pairs of adjacent nodes have different colors. The aim is to use as few colors as possible. The smallest number of colors needed to color the nodes of a graph G is known as the graph's chromatic number, denoted by $\chi(G)$.
- An edge coloring of a graph is an assignment of colors to edges so that all pairs of adjacent edges have different colors (a pair of edges is considered adjacent if and only if they share a common endpoint). The aim is to use as few colors as possible. The smallest number of colors needed for coloring the edges of a graph G is known as the graph's chromatic index, denoted by $\chi'(G)$. According to Vizing's theorem, $\chi'(G)$ is either $\Delta(G)$ or $\Delta(G)+1$, where $\Delta(G)$ is the maximum degree in G.
- In the node precoloring problem, some of the nodes have already been assigned colors. The aim is to allocate colors to the remaining nodes so that we get a full coloring that uses a minimum number of colors. The same concepts apply for the edge precoloring problem.

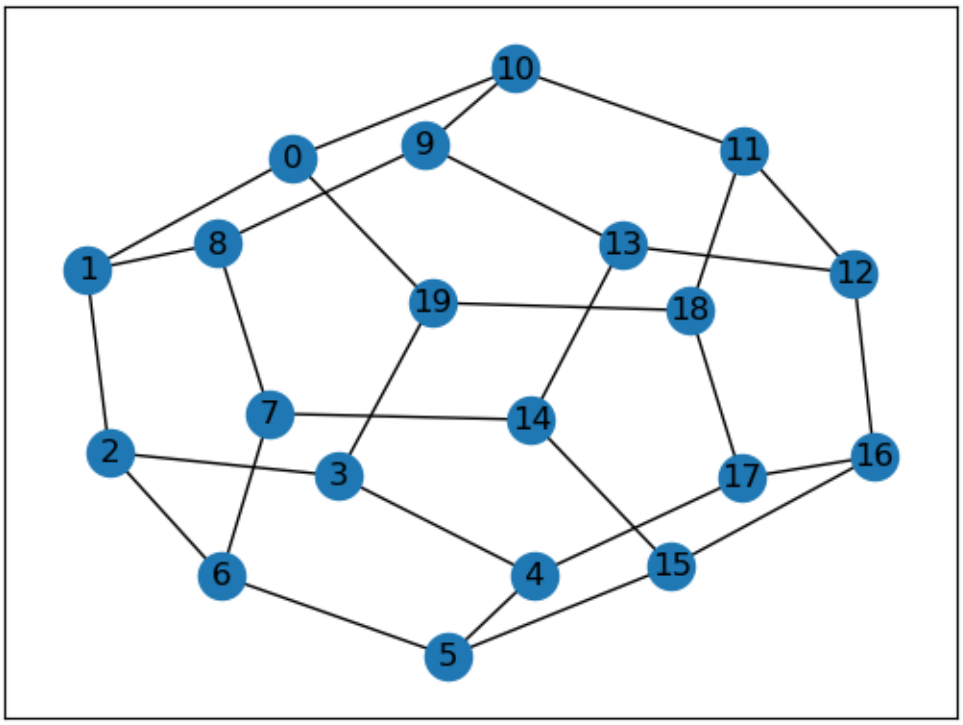
Note that all the above problems are NP-hard.

2.1 Getting Started

In this first example, we start by importing the libraries we need. The `networkx` library is used to generate and visualize the graphs, the `matplotlib.pyplot` is used to render the drawings, and the `gcol` library contains all of our graph coloring routines.

Having imported the relevant libraries, the following code generates a dodecahedron graph and draws it to the screen.

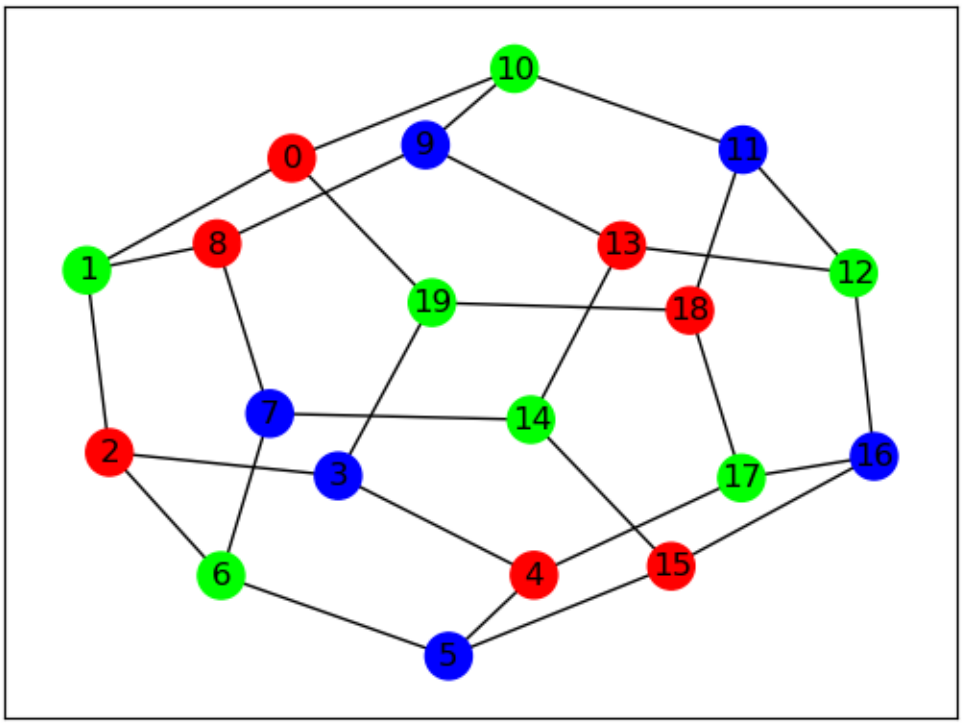
```
import networkx as nx              # popular python library for the graph theory
import matplotlib.pyplot as plt    # popular library for visualization
import gcol                        # the gcol library

G = nx.dodecahedral_graph()
nx.draw_networkx(G, pos=nx.spring_layout(G, seed=1))
plt.show()
```



Now that we have defined a graph, we can easily color it using `gcol`'s routines. The example below shows how to use the `gcol.node_coloring()` routine to color the nodes of `G`. Some information about this coloring is then written to the screen, along with a visualization. The colors of the nodes are held in the dictionary `c`, using the integers `0,1,2, ...` as color labels. We can also write the coloring as a partition, which groups all nodes of the same color. Note that adjacent nodes are always painted with different colors, as required.

```
c = gcol.node_coloring(G)

print("Here is a node coloring of the above graph:")
print(c)
print("The number of colors in this solution is:", max(c.values()) + 1)

print("Here is the same solution, expressed as a partition of the nodes:")
print(gcol.partition(c))

print("Here is a picture of this coloring:")
nx.draw_networkx(G,
                 pos=nx.spring_layout(G, seed=1),
                 node_color=gcol.get_node_colors(G, c))
plt.show()
```

```
Here is a node coloring of the above graph:
{0: 0, 1: 1, 19: 1, 10: 1, 2: 0, 3: 2, 8: 0, 9: 2, 18: 0, 11: 2, 6: 1, 7: 2, 4: 0, 5: 2,␣
↪13: 0, 12: 1, 14: 1, 15: 0, 16: 2, 17: 1}
The number of colors in this solution is: 3
Here is the same solution, expressed as a partition of the nodes:
[[0, 2, 4, 8, 13, 15, 18], [1, 6, 10, 12, 14, 17, 19], [3, 5, 7, 9, 11, 16]]
Here is a picture of this coloring:
```



We can also write similar commands to determine the chromatic number and chromatic index of this graph.

```
print("The chromatic number of this graph is:", gcol.chromatic_number(G))
print("The chromatic index of this graph is:", gcol.chromatic_index(G))
```

```
The chromatic number of this graph is: 3
The chromatic index of this graph is: 3
```

2.2 Node Coloring and Visualization

The previous example shows a node 3-coloring of the 20-node dodecahedron graph. The positions of the nodes in the visualization have been determined using the `nx.spring_layout()` routine from `networkx`; however, we can also choose to position the nodes based on their colors.

The first example below uses the routine `gcol.coloring_layout()` in combination with `nx.draw_networkx()` to position the nodes in a ring so that those of the same color are next to each other. Similarly, the second example uses the routine `gcol.multipartite_layout()` to put nodes of the same color into columns.

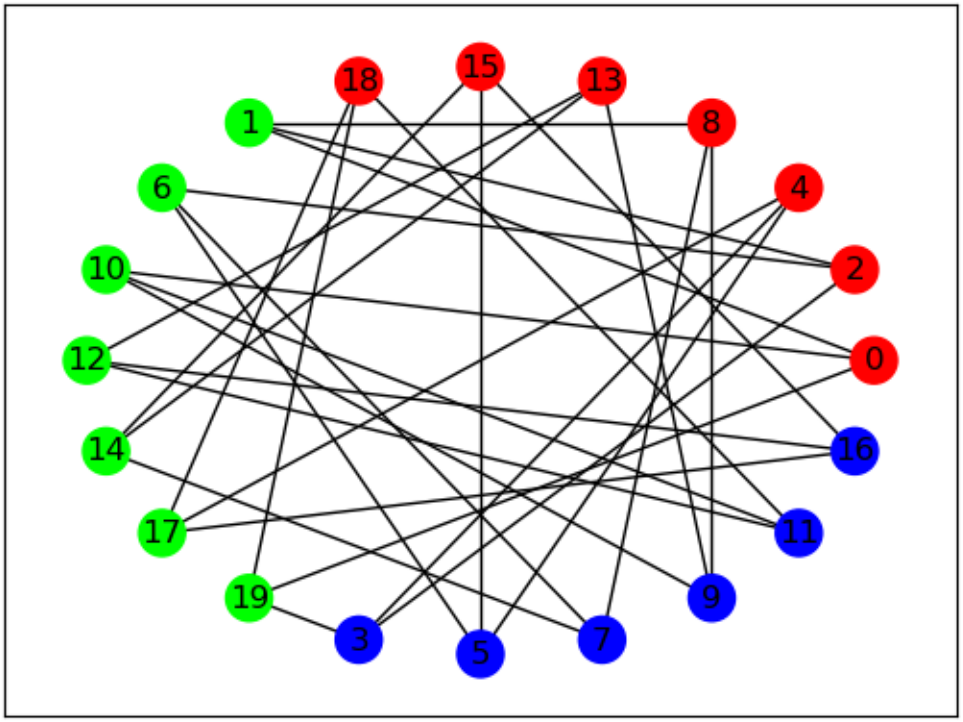
Note that, despite looking superficially different, the solutions shown are the same as the previous example.

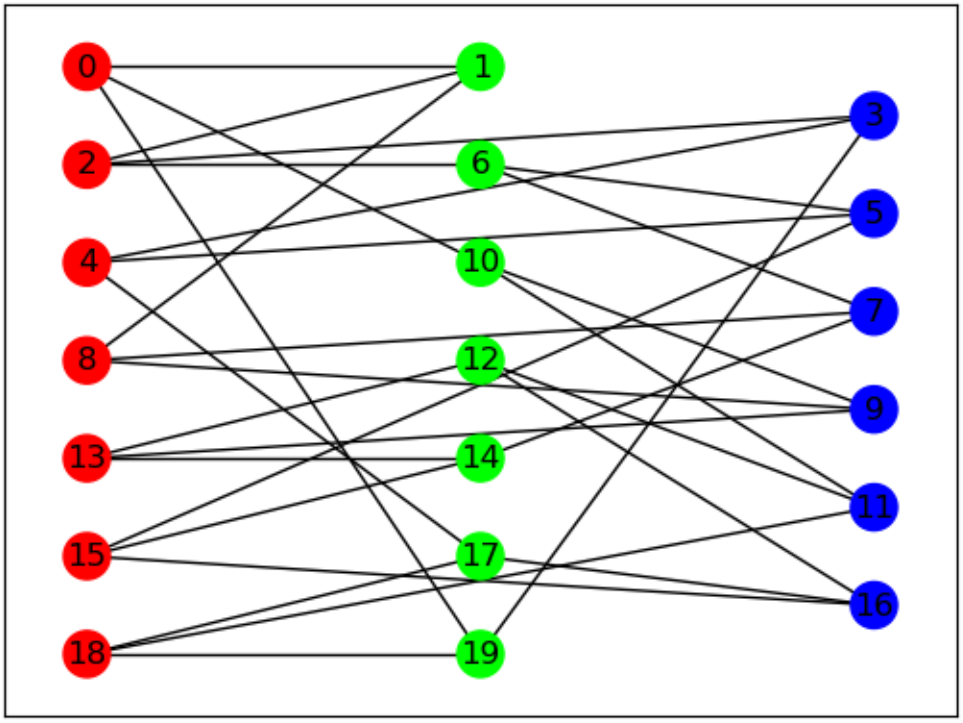
```
G = nx.dodecahedral_graph()
nx.draw_networkx(G,
                 pos=gcol.coloring_layout(G, c),
                 node_color=gcol.get_node_colors(G, c))
plt.show()

nx.draw_networkx(G,
                 pos=gcol.multipartite_layout(G, c),
                 node_color=gcol.get_node_colors(G, c))
```

(continues on next page)

(continued from previous page)

```
plt.show()
```

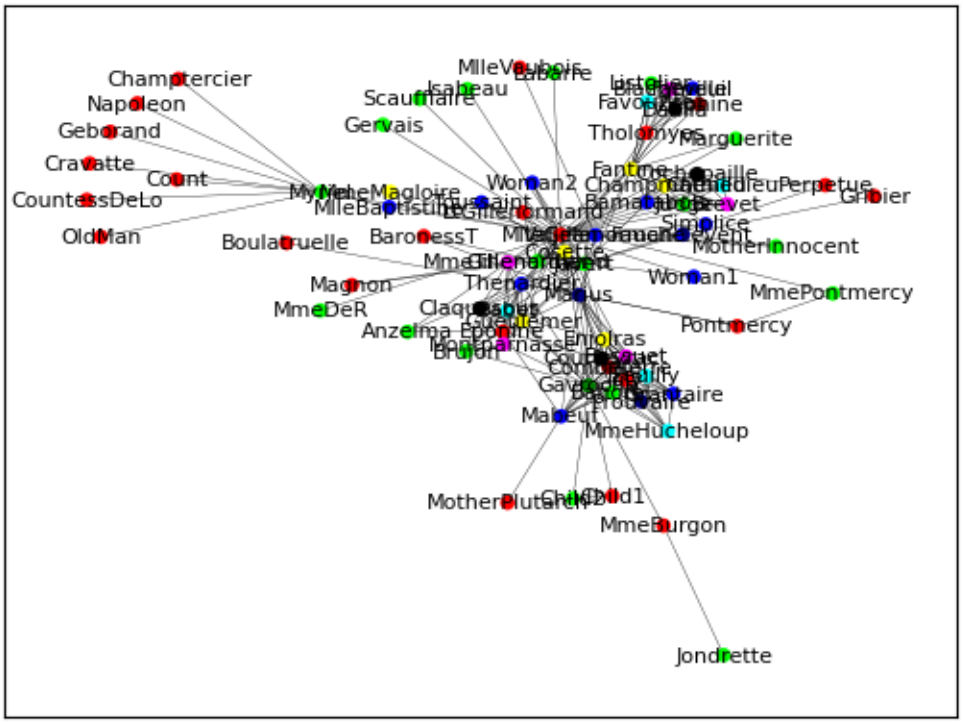




We will now do something similar with a larger graph. In the following, the nodes of the graph `G` represent the different characters in the play Les Miserables. Edges between nodes then indicate pairs of characters that appear in the same scenes together.

```
G = nx.les_miserables_graph()
c = gcol.node_coloring(G, opt_alg=1)

print("Number of colors =", max(c.values()) + 1)
nx.draw_networkx(G,
                 pos=nx.spring_layout(G, seed=1),
                 node_color=gcol.get_node_colors(G, c),
                 node_size=20,
                 font_size=8,
                 width=0.25)
plt.show()
```

```
Number of colors = 10
```



Note in the above that we have used the option `opt_alg=1` meaning that an exact algorithm has been used to produce the optimal solution. The output tells us that the nodes of `G` can be colored using a minimum of ten colors. In this case, it means that it is possible to partition the characters of Les Miserables into ten groups (but not fewer) so that the characters in each group never appear together.

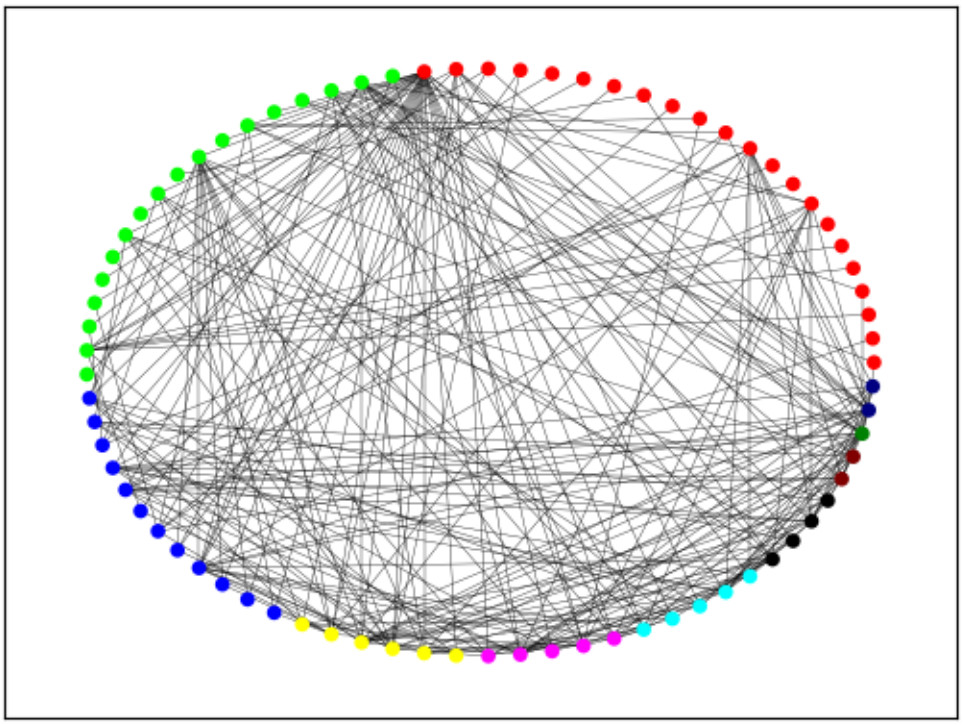
The visualization of the above graph appears rather cluttered, however, so we might choose to position the nodes according to color and remove the node's labels. This can be done using the following commands, which show the same solution.

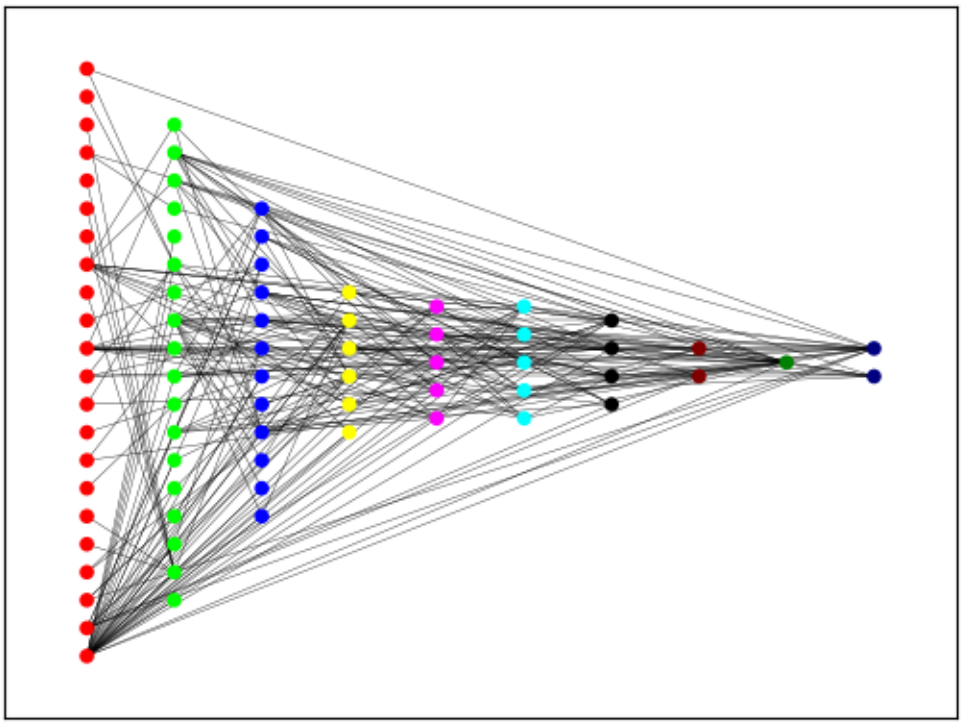
```
nx.draw_networkx(G,
                 pos=gcol.coloring_layout(G, c),
                 node_color=gcol.get_node_colors(G, c),
                 node_size=20,
                 with_labels=False,
                 width=0.25)
```

(continues on next page)

(continued from previous page)

```
plt.show()

nx.draw_networkx(G,
                 pos=gcol.multipartite_layout(G, c),
                 node_color=gcol.get_node_colors(G, c),
                 node_size=20,
                 with_labels=False,
                 width=0.25)
plt.show()
```





The following code carries out a similar sequence of operations for a random $G(n, p)$ graph. These graphs are generated at random by taking n nodes and then adding an edge between each pair of nodes at random with probability p. In this case we use $n = 100$ and $p = 0.05$. We also make use of local search by setting `opt_alg=2` and `it_limit=10000` in the `gcol.node_coloring()` routine.

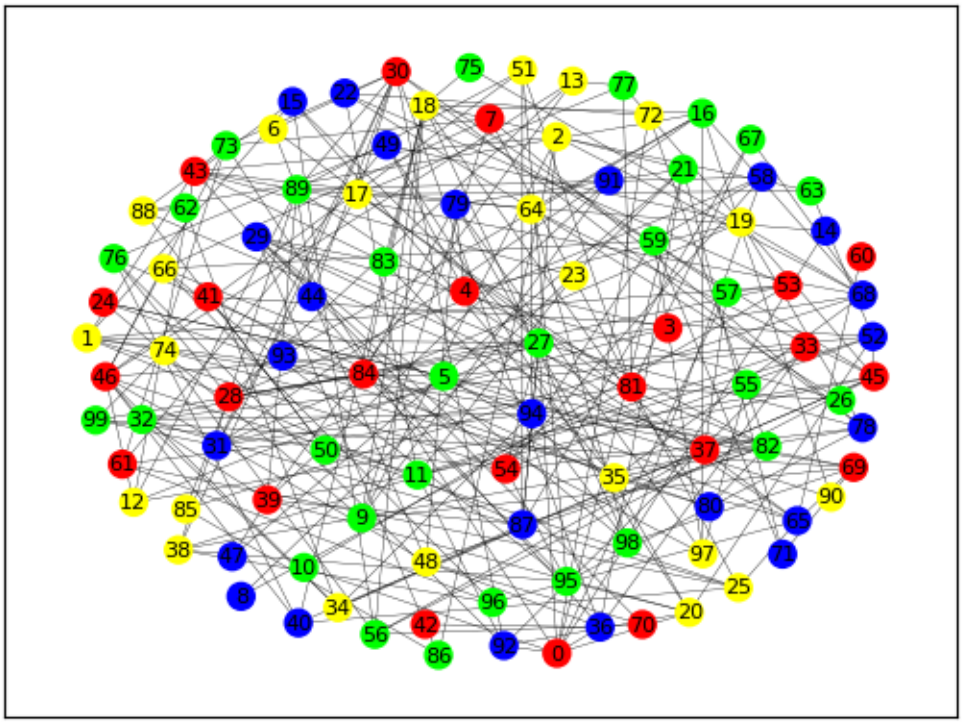
```
G = nx.gnp_random_graph(100, 0.05, seed=1)
c = gcol.node_coloring(G, strategy="random", opt_alg=2, it_limit=10000)

print("Number of colors =", max(c.values()) + 1)

nx.draw_networkx(G,
                 pos=nx.arf_layout(G),
                 node_color=gcol.get_node_colors(G, c),
                 node_size=100,
                 font_size=8,
                 width=0.25)
plt.show()

nx.draw_networkx(G,
                 pos=gcol.coloring_layout(G, c),
                 node_color=gcol.get_node_colors(G, c),
                 node_size=20,
                 with_labels=False,
                 width=0.25)
plt.show()

nx.draw_networkx(G,
                 pos=gcol.multipartite_layout(G, c),
                 node_color=gcol.get_node_colors(G, c),
```

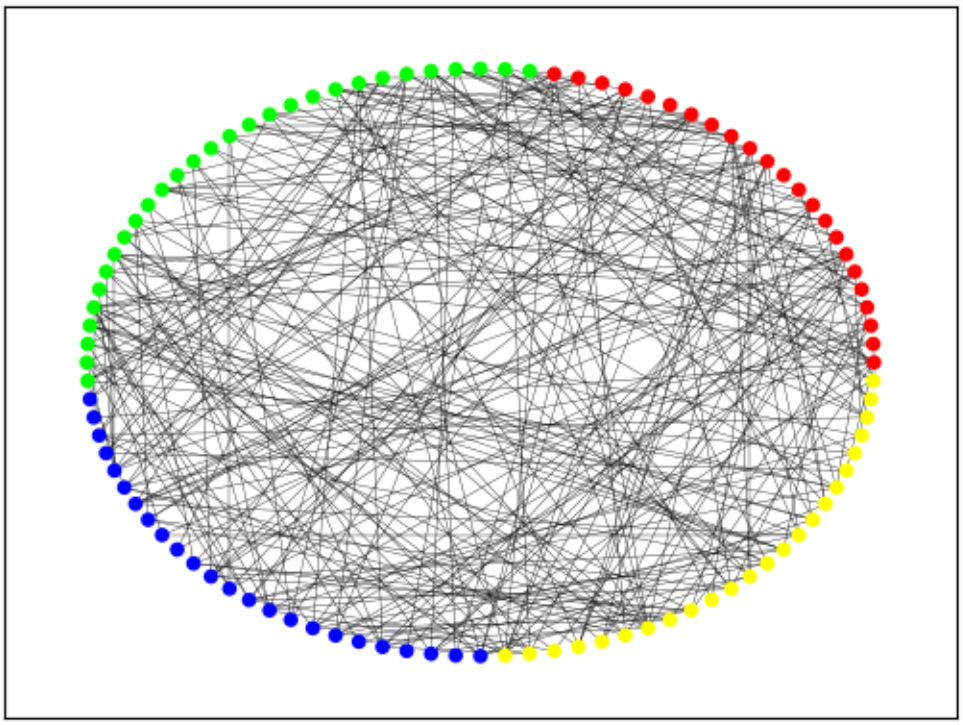
(continues on next page)

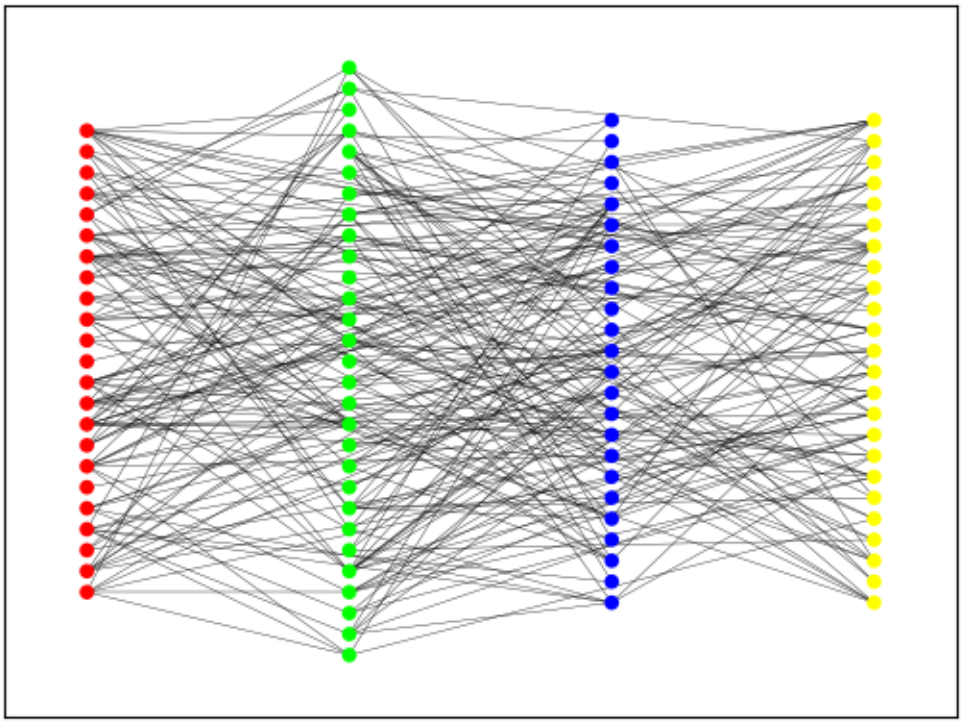
(continued from previous page)

```
                node_size=20,
                with_labels=False,
                width=0.25)
plt.show()
```

```
Number of colors = 4
```





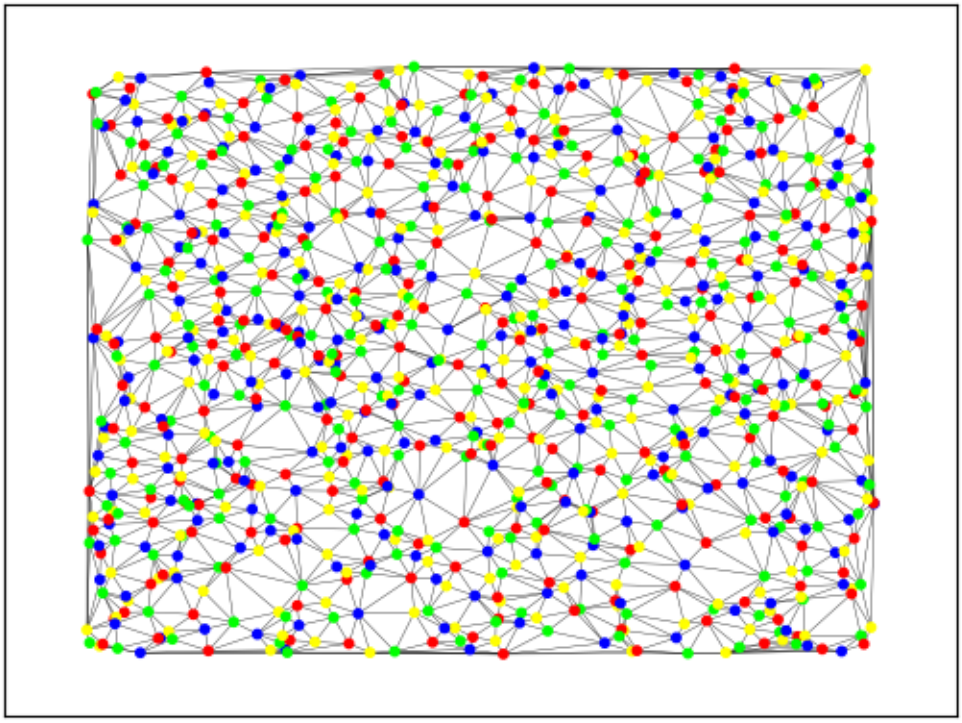


The final example in this section gives a bespoke routine for producing a dense planar graph. In the production of this graph, each node is given an (x, y) coordinate in the unit square. These coordinates are then used when drawing the graph. In the following code, we generate a 1000-node graph and color it using local search.

Note that the nodes of all planar graphs can be colored using at most four colors. This is due to the famous Four Color Theorem.

```
def make_planar_graph(n, seed=None):
    #Function for making a dense planar graph by placing nodes randomly into the unit␣
↪square
    import random
    from scipy.spatial import Delaunay
    random.seed(seed)
    P = [(random.random(), random.random()) for i in range(n)]
    T = Delaunay(P).simplices.copy()
    G = nx.Graph()
    for v in range(n):
        G.add_node(v, pos=(P[v][0], P[v][1]))
    for x, y, z in T:
        G.add_edges_from([(x, y), (x, z), (y, z)])
    return G

G = make_planar_graph(1000, seed=1)
c = gcol.node_coloring(G, opt_alg=2, it_limit=10000)
print("Number of colors =", max(c.values()) + 1)
nx.draw_networkx(G,
                 pos=nx.get_node_attributes(G, "pos"),
                 with_labels=False,
                 node_size=10,
                 node_color=gcol.get_node_colors(G, c),
                 width=0.25)
plt.show()
```

```
Number of colors = 4
```

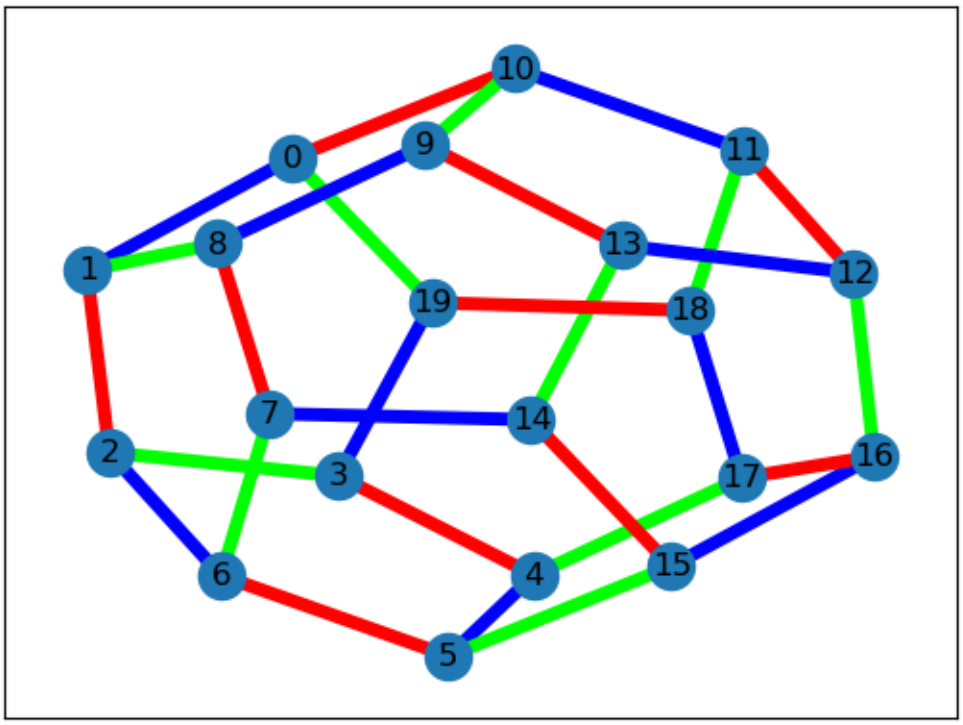


2.3 Edge Coloring and Visualization

The following example shows how we can use the `gcol` library to color the edges of a graph. As we have discussed, in edge coloring the maximum degree $\Delta(G)$ in the graph G gives a lower bound on the chromatic index $\chi'(G)$. Since $\Delta(G) = 3$ and an edge-3-coloring has been determined, we can conclude that this is an optimal solution.

```
G = nx.dodecahedral_graph()
c = gcol.edge_coloring(G)
print("Here is the color of each edge:")
print(c)

print("Here is the same solution, expressed as a partition of the edges:")
print(gcol.partition(c))

print("Maximum degree   =", max(G.degree(v) for v in G))
print("Number of colors =", max(c.values()) + 1)

nx.draw_networkx(G,
                 pos=nx.spring_layout(G, seed=1),
                 edge_color=gcol.get_edge_colors(G, c),
                 width=5)
```

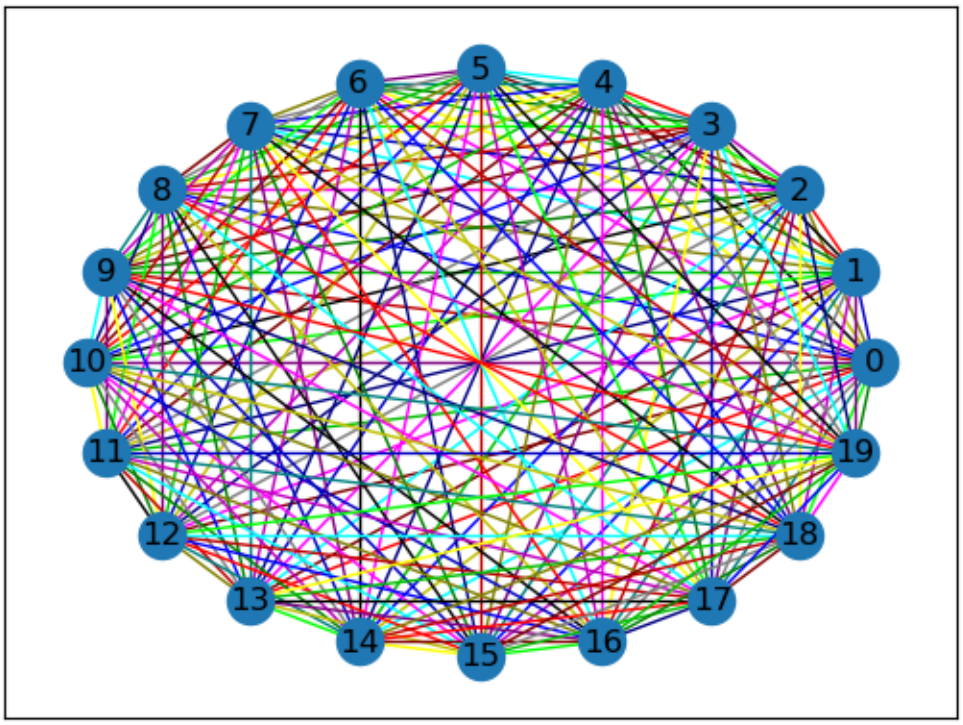
```
Here is the color of each edge:
{(11, 12): 0, (11, 18): 1, (10, 11): 2, (12, 16): 1, (12, 13): 2, (18, 19): 0, (17, 18):␣
↪2, (16, 17): 0, (4, 17): 1, (15, 16): 2, (0, 10): 0, (9, 10): 1, (9, 13): 0, (8, 9): 2,
↪ (13, 14): 1, (14, 15): 0, (5, 15): 1, (7, 14): 2, (0, 19): 1, (3, 19): 2, (0, 1): 2,␣
↪(3, 4): 0, (2, 3): 1, (1, 2): 0, (2, 6): 2, (5, 6): 0, (4, 5): 2, (1, 8): 1, (6, 7): 1,
↪ (7, 8): 0}
Here is the same solution, expressed as a partition of the edges:
[[(0, 10), (1, 2), (11, 12), (14, 15), (16, 17), (18, 19), (3, 4), (5, 6), (7, 8), (9,␣
↪13)], [(0, 19), (1, 8), (11, 18), (12, 16), (13, 14), (2, 3), (4, 17), (5, 15), (6, 7),
↪ (9, 10)], [(0, 1), (10, 11), (12, 13), (15, 16), (17, 18), (2, 6), (3, 19), (4, 5),␣
↪(7, 14), (8, 9)]]
Maximum degree   = 3
Number of colors = 3
```



Here is another example using a complete graph. Edge coloring in complete graphs has applications in sports league scheduling.

```
G = nx.complete_graph(20)
c = gcol.edge_coloring(G, opt_alg=1)
print("Maximum degree   =", max(G.degree(v) for v in G))
print("Number of colors =", max(c.values()) + 1)
nx.draw_networkx(G,
                 pos=nx.circular_layout(G),
                 edge_color=gcol.get_edge_colors(G, c))
```

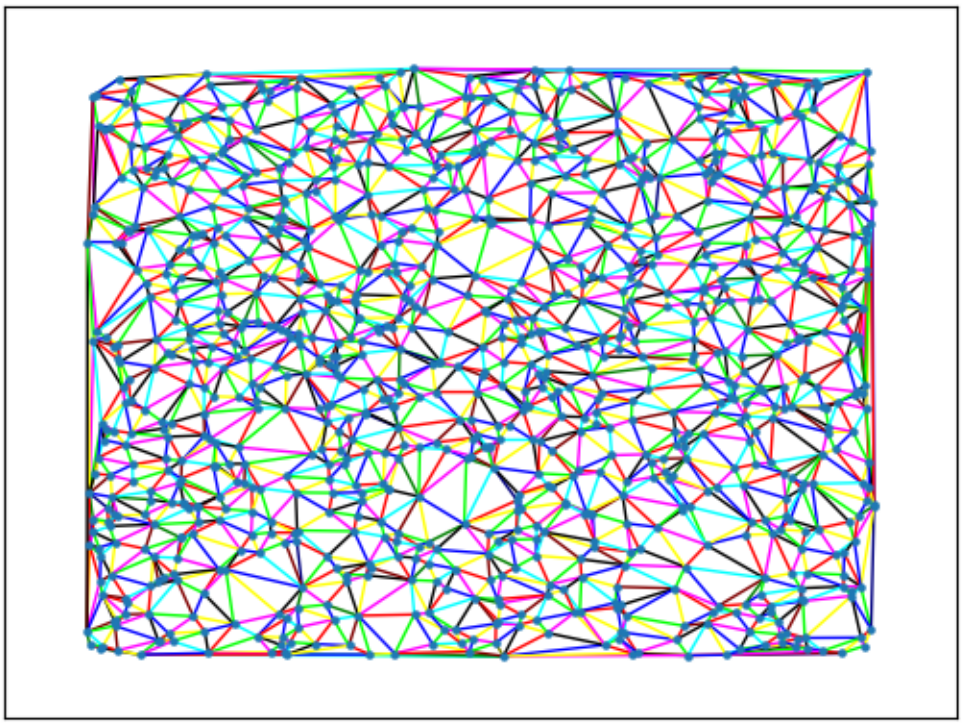
```
Maximum degree   = 19
Number of colors = 19
```



Here is another example using the same 1000-node planar graph from earlier.

```
G = make_planar_graph(1000, seed=1)
c = gcol.edge_coloring(G, opt_alg=2, it_limit=1000)
print("Maximum degree   =", max(G.degree(v) for v in G))
print("Number of colors =", max(c.values()) + 1)
nx.draw_networkx(G,
                 pos=nx.get_node_attributes(G, "pos"),
                 with_labels=False, node_size=5,
                 edge_color=gcol.get_edge_colors(G, c))
```

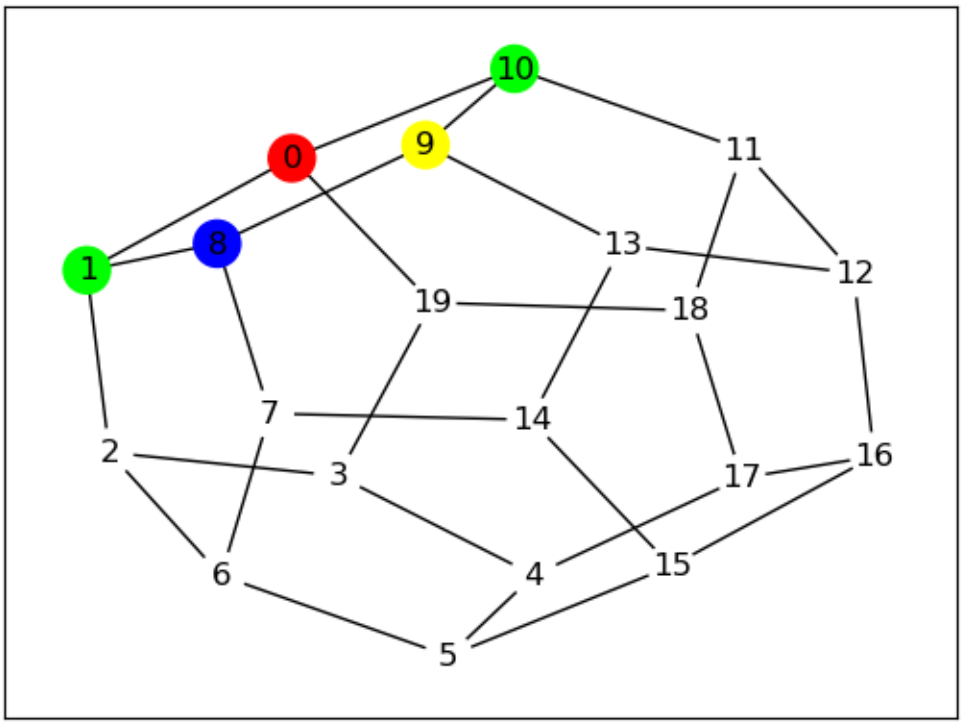
```
Maximum degree   = 11
Number of colors = 11
```

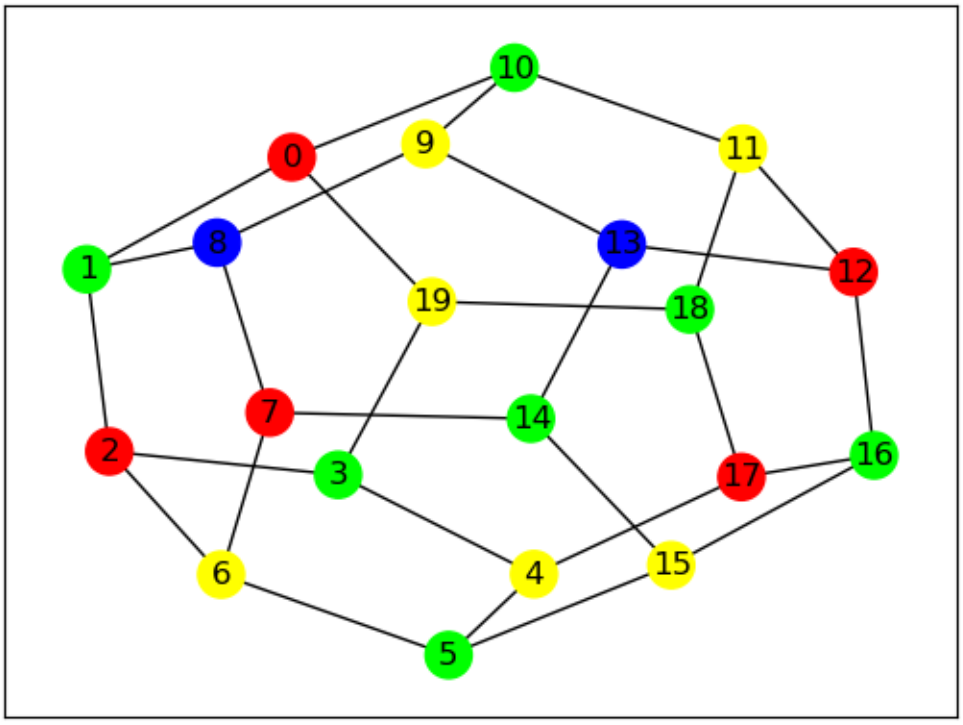


2.4 Precoloring

As mentioned earlier, in the node precoloring problem some of the nodes have already been assigned colors. The aim is to assign colors to the remaining nodes so that we get a full coloring that uses a minimum number of colors. In the example below, the dictionary P is used to assign nodes 0, 1, 8, 9 and 10 to colors 0, 1, 2, 3, and 1, respectively. This partial coloring is then shown, together with a corresponding full coloring.

```
G = nx.dodecahedral_graph()
P = {0:0, 1:1, 8:2, 9:3, 10:1}
nx.draw_networkx(G,
                 pos=nx.spring_layout(G, seed=1),
                 node_color=gcol.get_node_colors(G, P))
plt.show()

c = gcol.node_precoloring(G, P, strategy="random", opt_alg=2, it_limit=100)
nx.draw_networkx(G,
                 pos=nx.spring_layout(G, seed=1),
                 node_color=gcol.get_node_colors(G, c))
plt.show()
```





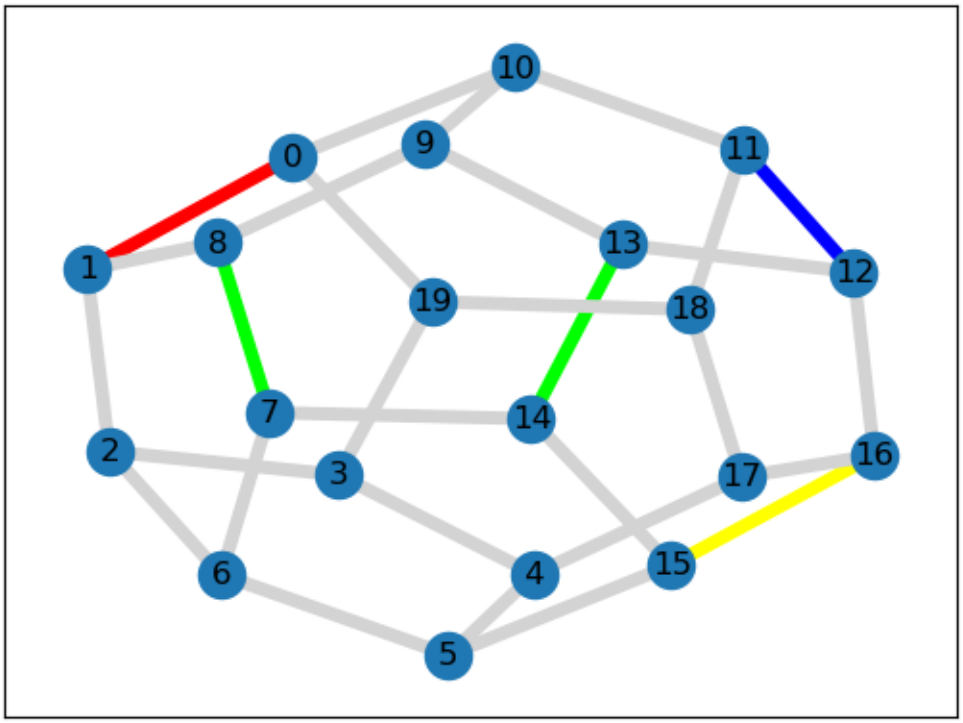
A similar process can also be followed for edge precoloring, which the following demonstrates. Note that, when defining edges in the dictionary `P` the endpoints must be given in the order used by NetworkX. For example, in the example below, using `(1,0):0` in `P` instead of `(0,1):0` will raise a `ValueError`.

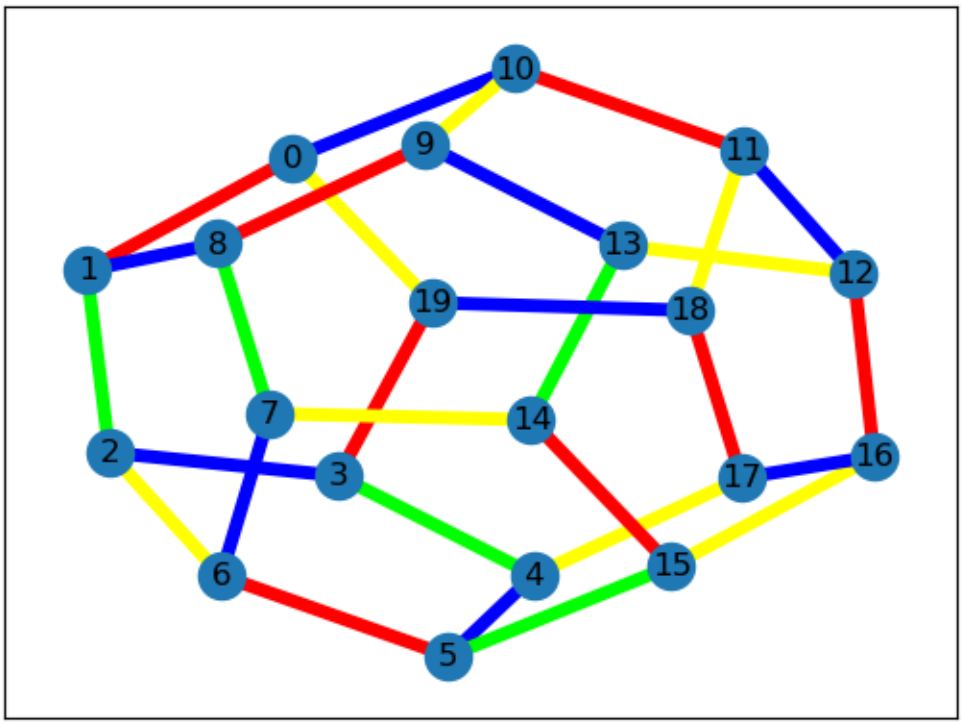
```
G = nx.dodecahedral_graph()
```

(continues on next page)

(continued from previous page)

```
P = {(0, 1): 0, (7, 8): 1, (13, 14): 1, (11, 12): 2, (15, 16): 3}
nx.draw_networkx(G,
                 pos=nx.spring_layout(G, seed=1),
                 edge_color=gcol.get_edge_colors(G, P),
                 width=5)
plt.show()

c = gcol.edge_precoloring(G, P, strategy="random", opt_alg=2, it_limit=100)
nx.draw_networkx(G,
                 pos=nx.spring_layout(G, seed=1),
                 edge_color=gcol.get_edge_colors(G, c),
                 width=5)
plt.show()
```





Node precoloring can also be used to solve sudoku puzzles. The objective in sudoku is to fill a $d^2 \times d^2$ grid with digits so that each column, each row, and each of the $d \times d$ boxes contains all of the digits from 0 to $d^2 - 1$. The puzzle comes with some of the cells filled. The player then needs to fill the remaining cells while satisfying the above constraints. Here is an example puzzle using $d = 3$ and the digits $0, 1, \ldots, 8$. Blank cells are marked by dots.

$$\begin{pmatrix} . & . & . & . & 5 & . & . & . & . \\ . & . & 8 & . & 1 & . & . & . & . \\ . & . & . & . & . & . & . & 1 & 2 \\ . & 7 & . & . & . & 2 & . & . & . \\ . & . & . & 6 & . & . & . & . & . \\ . & 4 & . & . & . & 0 & . & 3 & . \\ . & 5 & . & . & . & . & . & . & . \\ . & . & . & . & 2 & . & 6 & . & . \\ . & . & . & . & . & . & . & 2 & . \end{pmatrix}$$

Sudoku puzzles can be solved by first forming a sudoku graph, which uses a node for each cell in the grid. Edges in this graph occur between all pairs of nodes in the same column, row or box. Finally, we use the filled cells in the puzzle to precolor the correct nodes. The puzzle is then solved by coloring the remaining nodes using d^2 colors. The following code shows how to solve the above puzzle

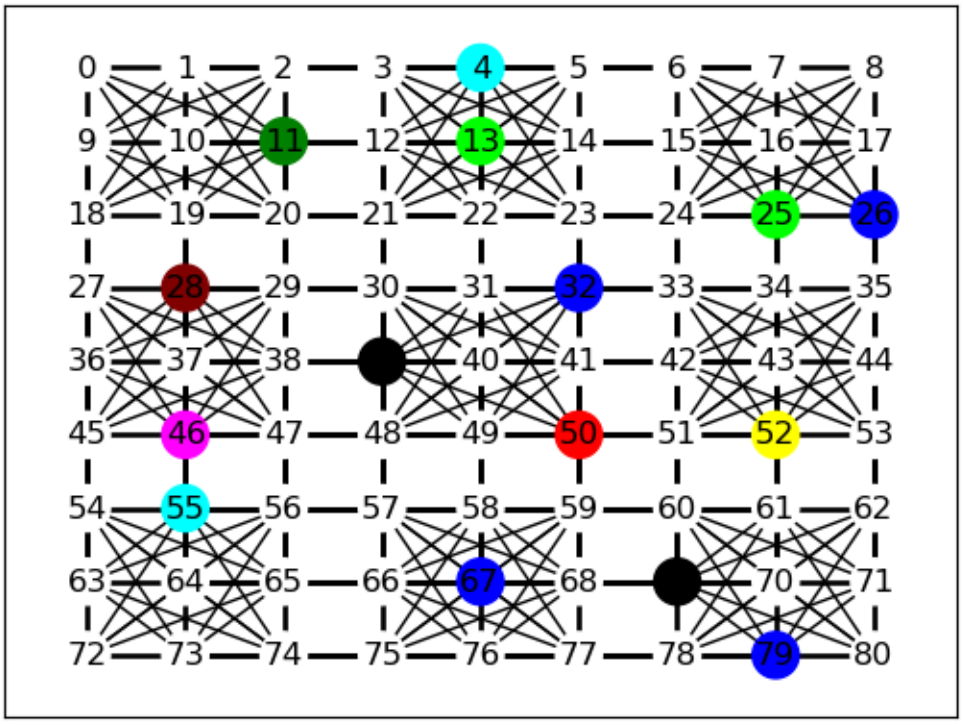
```
# Function for laying out the nodes of a (d**2 x d**2)-node sudoku puzzle
def sudoku_layout(G, d):
    pos = {}
    u = 0
    for i in range(d**2):
        for j in range(d**2):
            pos[u]=(j,-i)
            u += 1
    return pos
```

(continues on next page)

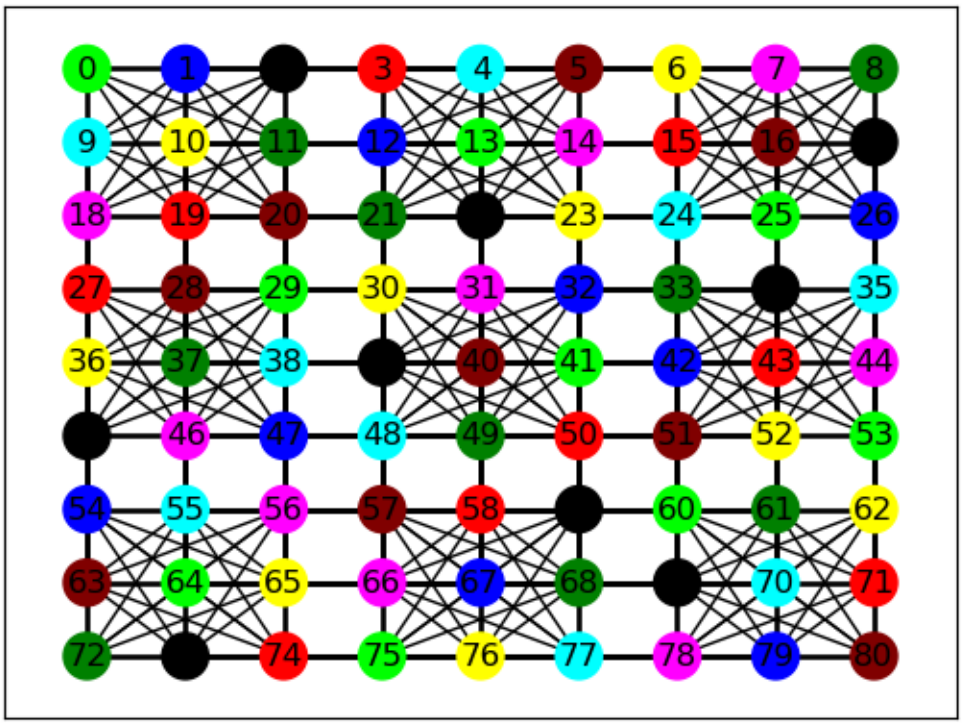
(continued from previous page)

```
G = nx.sudoku_graph(3)
P = {4:5, 11:8, 13:1, 25:1, 26:2, 28:7, 32:2, 39:6, 46:4, 50:0, 52:3, 55:5, 67:2, 69:6,␣
↪79:2}
print("Here is the sudoku puzzle from above")
nx.draw_networkx(G,
                 pos=sudoku_layout(G, 3),
                 node_color=gcol.get_node_colors(G, P))
plt.show()

c = gcol.node_precoloring(G, P, opt_alg=1)
print("Here is its solution. Number of colors =", max(c.values()) + 1)
nx.draw_networkx(G,
                 pos=sudoku_layout(G, 3),
                 node_color=gcol.get_node_colors(G, c))
plt.show()
```

```
Here is the sudoku puzzle from above
```




```
Here is its solution. Number of colors = 9
```



2.5 k-Coloring

The k-coloring problem is a version of the graph coloring problem where the desired number of colors k is stated beforehand by the user. For node coloring, if $k < \chi(G)$, then no solution is possible; for edge coloring, if $k < \chi'(G)$, then no solution is possible. Several variants of the k-coloring problem can be formulated, including equitable coloring and weighted graph coloring, using both weighted and unweighted graphs. Examples are considered below.

In this first example, we make use of `gcol.node_k_coloring()` method to produce k-colorings of a random $G(1000, 0.05)$ graph for $k = 6$, 5, and 4. (For values of $k < 4$, solutions are not possible and a `ValueError` will be returned)

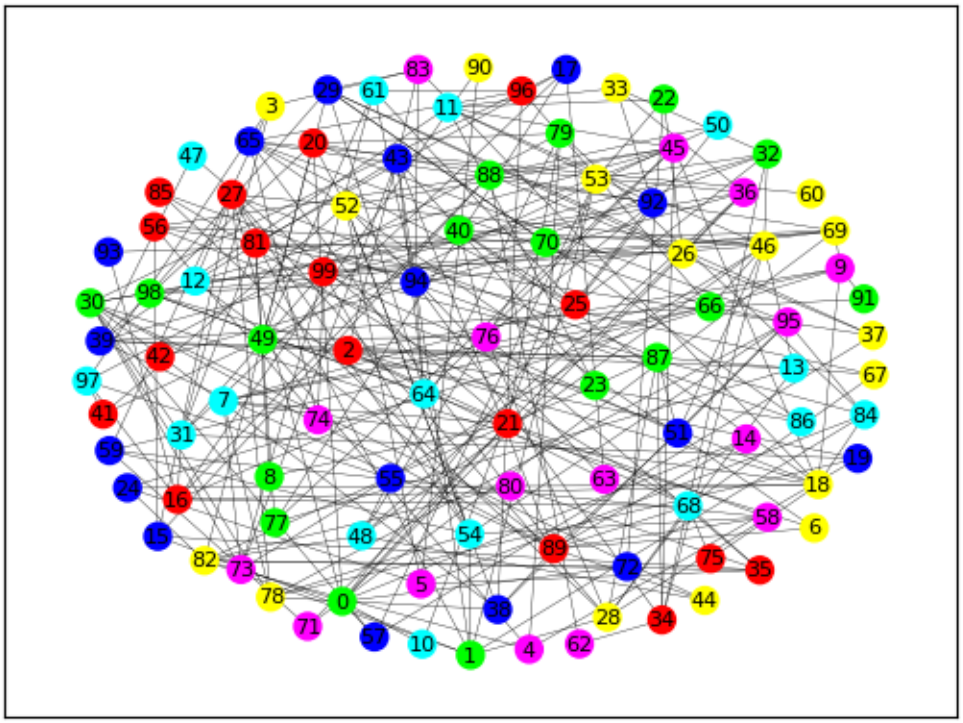
```
G = nx.gnp_random_graph(100, 0.05, seed=1)

print("Here is a node 6-coloring of G, ")
c = gcol.node_k_coloring(G, 6, opt_alg=2, it_limit=1000)
nx.draw_networkx(G,
                 pos=nx.arf_layout(G),
                 node_color=gcol.get_node_colors(G, c),
                 node_size=100,
                 font_size=8,
                 width=0.25)
plt.show()

print("here is a node 5-coloring of G,")
c = gcol.node_k_coloring(G, 5, opt_alg=2, it_limit=1000)
nx.draw_networkx(G,
                 pos=nx.arf_layout(G),
                 node_color=gcol.get_node_colors(G, c),
                 node_size=100,
```

(continues on next page)

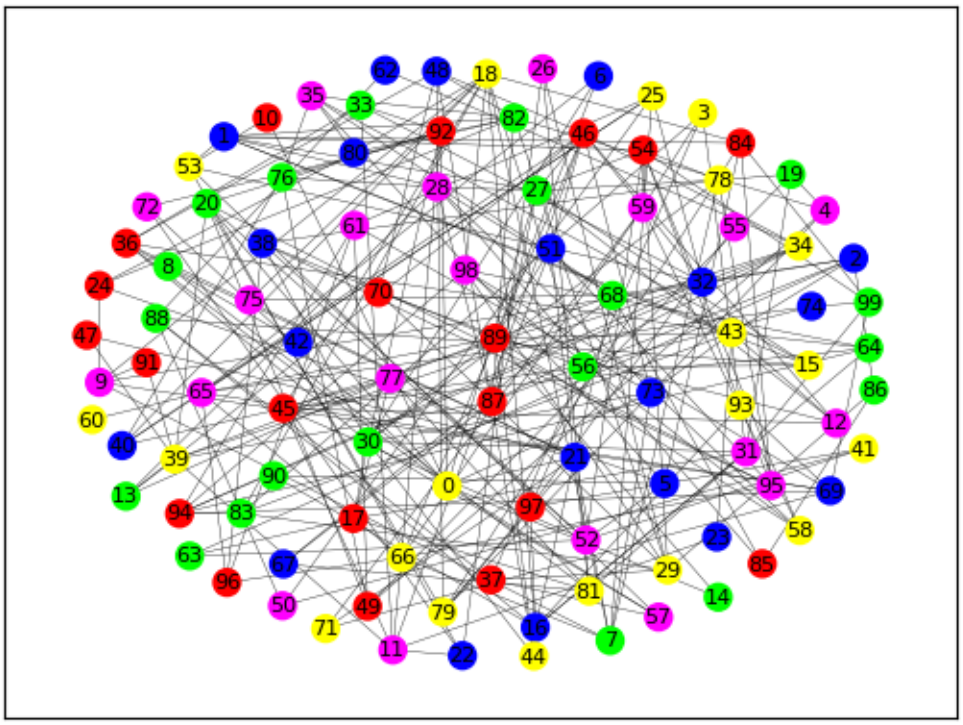
(continued from previous page)

```
                  font_size=8,
                  width=0.25)
plt.show()

print("and here is a node 4-coloring of G.")
c = gcol.node_k_coloring(G, 4, opt_alg=2, it_limit=1000)
nx.draw_networkx(G,
                  pos=nx.arf_layout(G),
                  node_color=gcol.get_node_colors(G, c),
                  node_size=100,
                  font_size=8,
                  width=0.25)
plt.show()
```

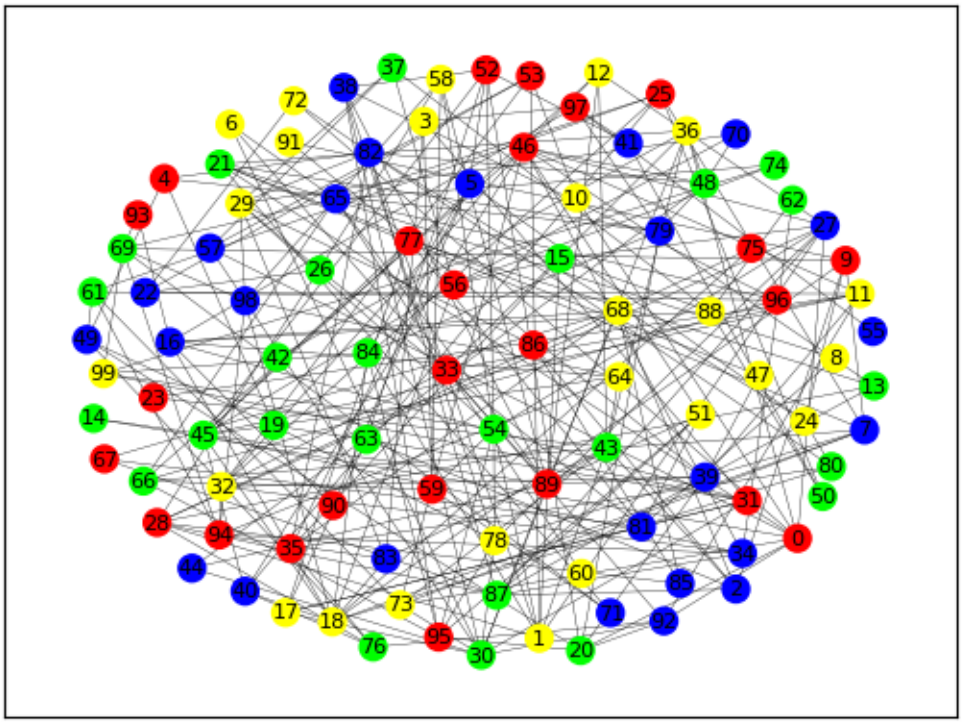
```
Here is a node 6-coloring of G,
```



```
here is a node 5-coloring of G,
```



```
and here is a node 4-coloring of G.
```



The following shows a similar process for edge k-coloring.

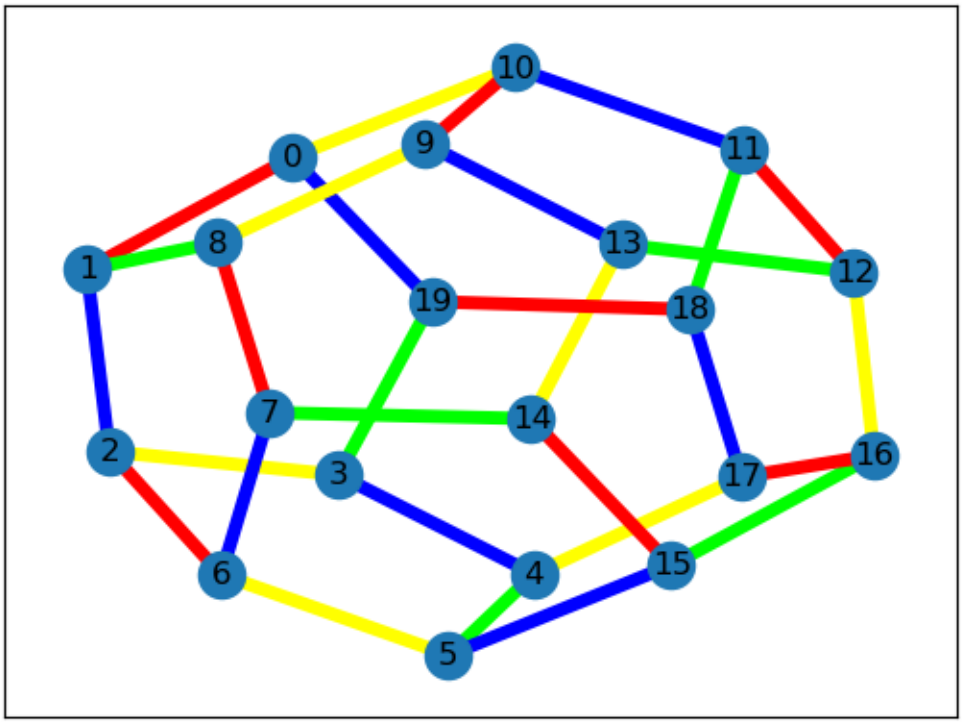
```
G = nx.dodecahedral_graph()
```

(continues on next page)

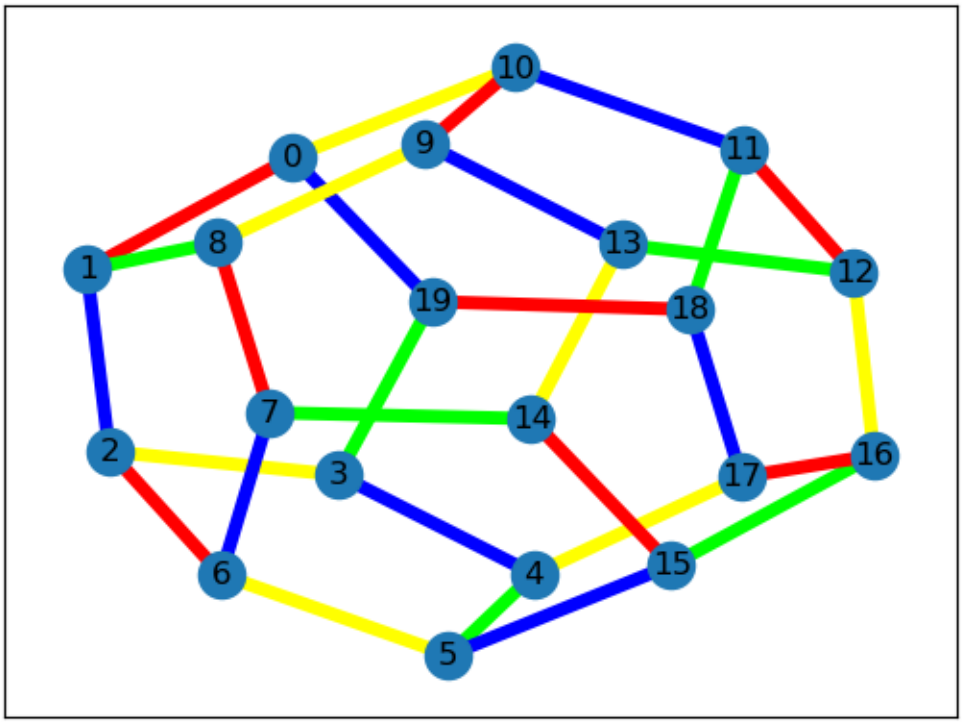
(continued from previous page)

```
print("Here is an edge 5-coloring of G,")
c = gcol.edge_k_coloring(G, 4)
nx.draw_networkx(G,
                 pos=nx.spring_layout(G, seed=1),
                 edge_color=gcol.get_edge_colors(G, c),
                 width=5)
plt.show()

print("here is an edge 4-coloring of G,")
c = gcol.edge_k_coloring(G, 4)
nx.draw_networkx(G,
                 pos=nx.spring_layout(G, seed=1),
                 edge_color=gcol.get_edge_colors(G, c),
                 width=5)
plt.show()

print("and here is an edge 3-coloring of G.")
c = gcol.edge_k_coloring(G, 3)
nx.draw_networkx(G,
                 pos=nx.spring_layout(G, seed=1),
                 edge_color=gcol.get_edge_colors(G, c),
                 width=5)
plt.show()
```

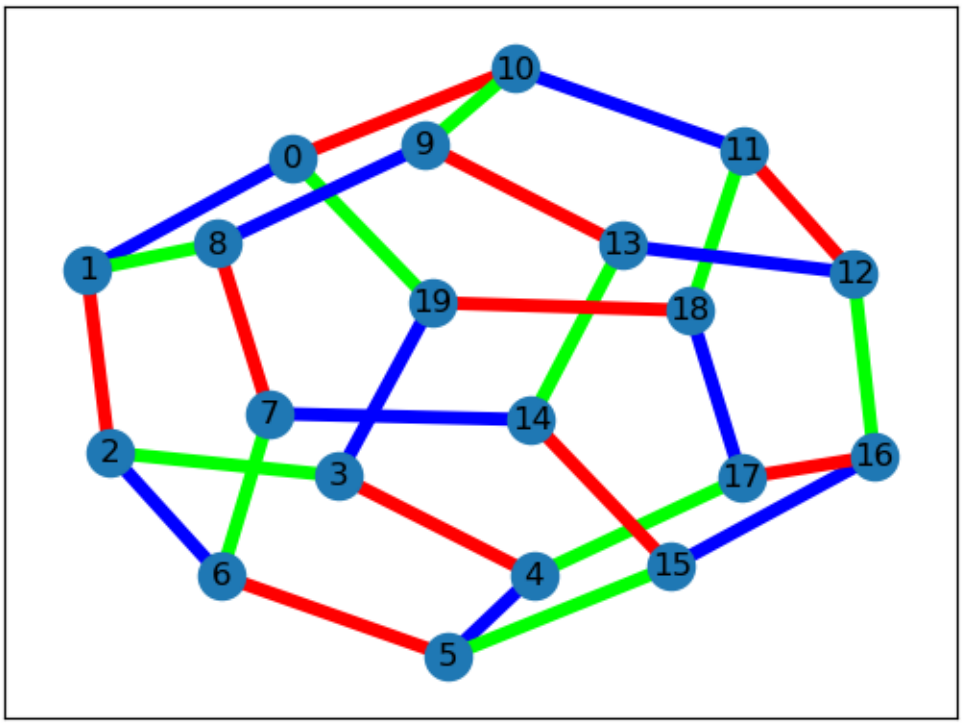
```
Here is an edge 5-coloring of G,
```



```
here is an edge 4-coloring of G,
```




```
and here is an edge 3-coloring of G.
```



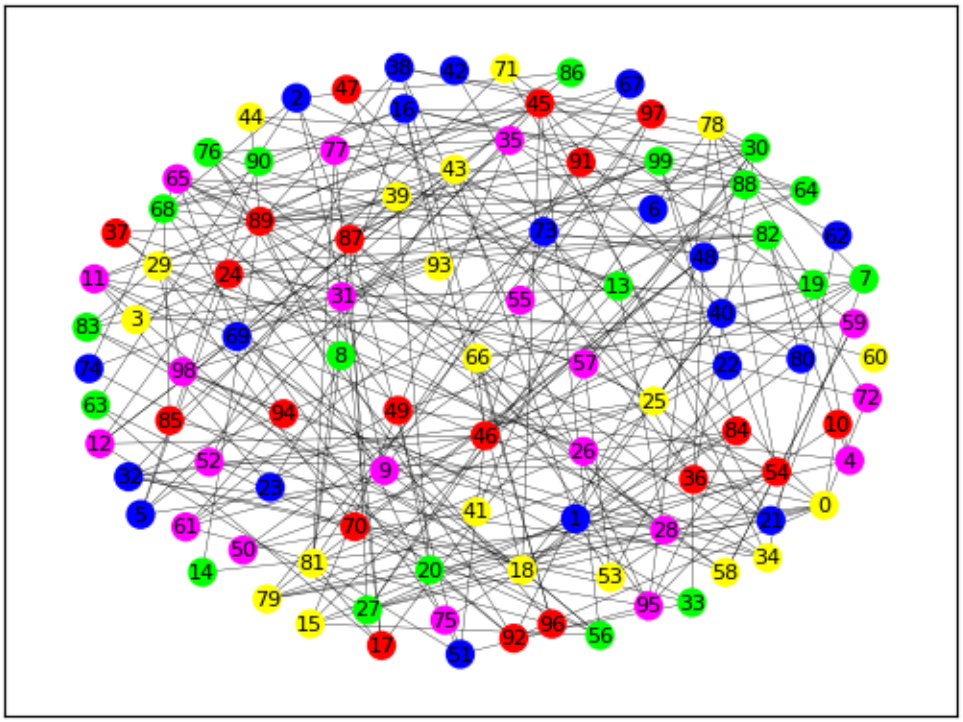
2.5.1 Equitable k-coloring

In the equitable node k-coloring problem we are seeking an assignment of colors to nodes so that no two adjacent nodes have the same color, and the number of nodes per-color is as uniform as possible. We can also choose to define positive weights on the nodes, in which case we are seeking a proper coloring in which the sum of the node weights in each color is as uniform as possible.

The following example determines an equitable node 5-coloring for a random $G(100, 0.05)$ graph.

```
G = nx.gnp_random_graph(100, 0.05, seed=1)

print("Here is an equitable node-5-coloring of G,")
c = gcol.equitable_node_k_coloring(G, 5, opt_alg=2, it_limit=1000)
nx.draw_networkx(G,
                 pos=nx.arf_layout(G),
                 node_color=gcol.get_node_colors(G, c),
                 node_size=100,
                 font_size=8,
                 width=0.25)
plt.show()
P = gcol.partition(c)
print("Largest color class has", max(len(j) for j in P), "nodes")
print("Smallest color class has", min(len(j) for j in P), "nodes")
```

```
Here is an equitable node-5-coloring of G,
```



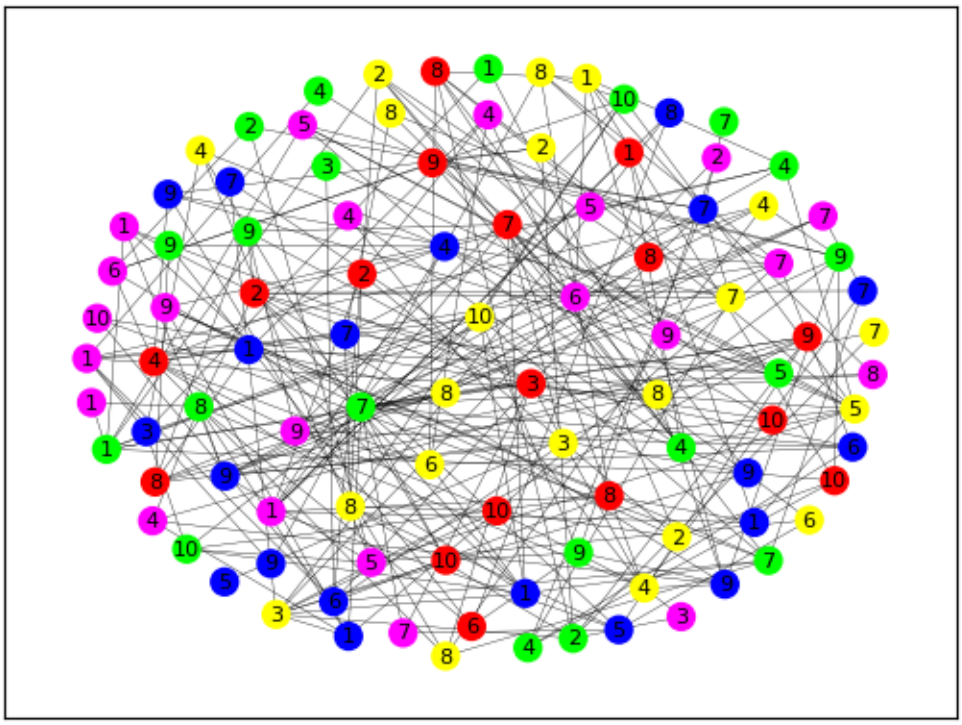
```
Largest color class has 20 nodes
Smallest color class has 20 nodes
```

The following example also determines an equitable node 5-coloring for a random $G(100, 0.05)$ graph. However, in this case, all nodes have been assigned weights randomly chosen from the set $\{1, 2, \ldots, 10\}$. The figure displays the

weight of each node and the text gives the total weight of each color class.

```
import random
random.seed(1)

H = nx.gnp_random_graph(100, 0.05, seed=1)
G = nx.Graph()
for u in H:
    G.add_node(u, weight=random.randint(1,10))
for u,v in H.edges():
    G.add_edge(u, v)

print("Here is an equitable node 5-coloring of the node-weighted graph G,")
c = gcol.equitable_node_k_coloring(G, 5, weight="weight", opt_alg=2, it_limit=1000)

labels = {u: G.nodes[u]['weight'] for u in G.nodes}
nx.draw_networkx(G,
                 pos=nx.arf_layout(G),
                 node_color=gcol.get_node_colors(G, c),
                 node_size=100,
                 font_size=8,
                 width=0.25,
                 labels=labels)
plt.show()
P = gcol.partition(c)
for j in range(len(P)):
    Wj = sorted([G.nodes[v]["weight"] for v in P[j]])
    print("Weight of color class", j, "=", sum(Wj), Wj)
```

```
Here is an equitable node 5-coloring of the node-weighted graph G,
```



```
Weight of color class 0 = 115 [1, 2, 2, 3, 4, 6, 7, 8, 8, 8, 8, 9, 9, 10, 10, 10, 10]
Weight of color class 1 = 115 [1, 1, 2, 2, 3, 4, 4, 4, 4, 5, 7, 7, 7, 8, 9, 9, 9, 9, 10,␣
↪10]
Weight of color class 2 = 114 [1, 1, 1, 1, 3, 4, 5, 5, 6, 6, 7, 7, 7, 7, 8, 9, 9, 9, 9,␣
↪9]
Weight of color class 3 = 114 [1, 2, 2, 2, 3, 3, 4, 4, 4, 5, 6, 6, 7, 7, 8, 8, 8, 8, 8,␣
↪8, 10]
Weight of color class 4 = 114 [1, 1, 1, 1, 2, 3, 4, 4, 4, 5, 5, 5, 6, 6, 7, 7, 7, 8, 9,␣
↪9, 9, 10]
```

The same process can also be followed to produce equitable edge k-colorings, as the following demonstrates. This uses an edge-weighted graph as indicated.

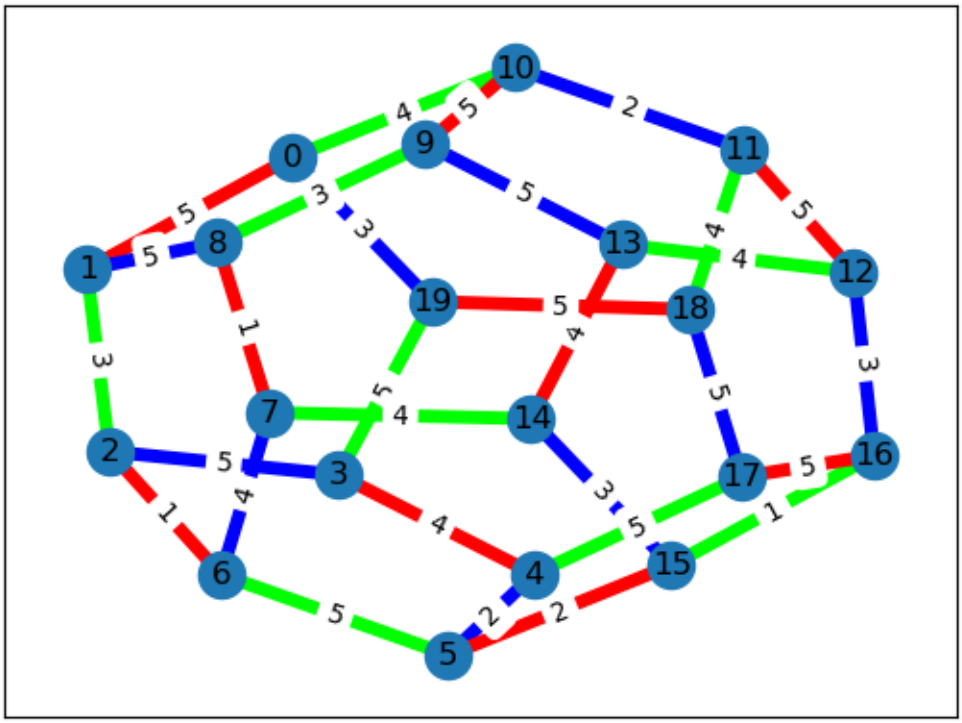
```
G = nx.dodecahedral_graph()
for u, v in G.edges():
    G.add_edge(u, v, edgeweight=random.randint(1,5))

print("Here is an equitable edge-3-coloring of the edge-weighted graph G,")
c = gcol.equitable_edge_k_coloring(G, 3, weight="edgeweight", opt_alg=2, it_limit=1000)

pos = nx.spring_layout(G, seed=1)
nx.draw_networkx(G, pos=pos, edge_color=gcol.get_edge_colors(G, c), width=5)
labels = nx.get_edge_attributes(G,'edgeweight')
nx.draw_networkx_edge_labels(G, pos=pos, edge_labels=labels)
plt.show()

P = gcol.partition(c)
for j in range(len(P)):
    Wj = sorted([G.edges[e]["edgeweight"] for e in P[j]])
```

(continues on next page)

(continued from previous page)

```
    print("Weight of color class", j, "=", sum(Wj), Wj)
```

```
Here is an equitable edge-3-coloring of the edge-weighted graph G,
```




```
Weight of color class 0 = 37 [1, 1, 2, 4, 4, 5, 5, 5, 5, 5]
Weight of color class 1 = 38 [1, 3, 3, 4, 4, 4, 4, 5, 5, 5]
Weight of color class 2 = 37 [2, 2, 3, 3, 3, 4, 5, 5, 5, 5]
```

2.5.2 Minimum Cost k-Coloring

Sometimes we are seeking a node k-coloring but are willing to allow some nodes to remain uncolored. This is particularly useful when using a value for k that is less than the graph's chromatic number $\chi(G)$. In such cases, we are seeking to minimize the number of uncolored nodes, while ensuring that adjacent colored nodes never have the same color. We might also choose to add positive weights to the nodes, in which case we will seek to minimize the sum of the weights of the uncolored nodes.

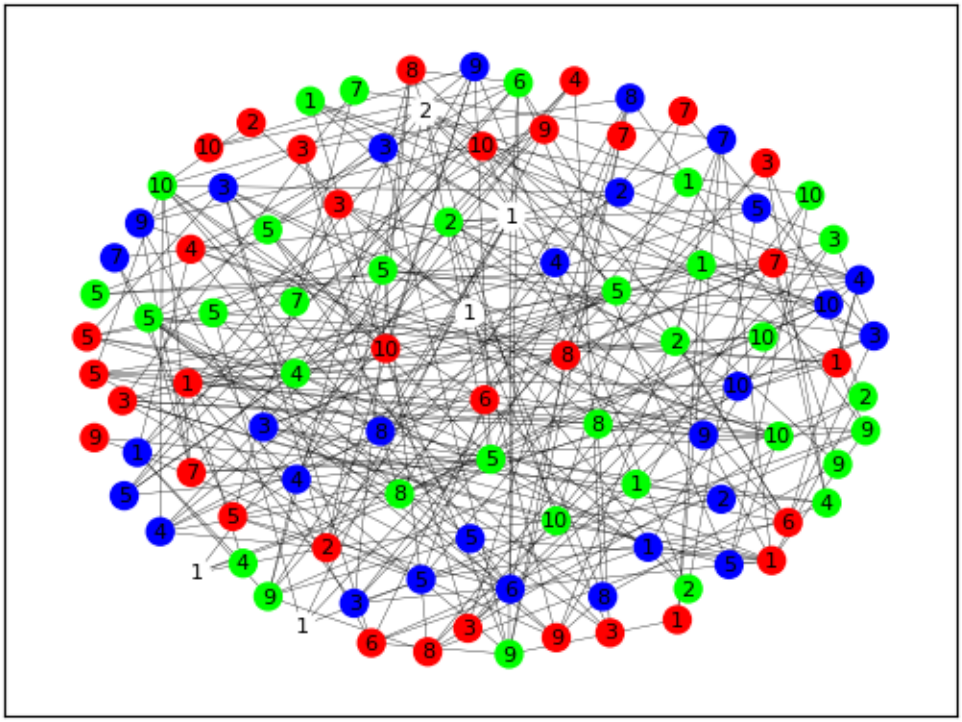
The following example creates a node-weighted random graph and then produces a node 3-coloring solution using the routine `gcol.min_cost_k_coloring()`. This solution has five uncolored nodes with a total weight of six.

```
H = nx.gnp_random_graph(100, 0.05, seed=1)
G = nx.Graph()
for u in H:
    G.add_node(u, weight=random.randint(1,10))
for u,v in H.edges():
    G.add_edge(u, v)
labels = {u: G.nodes[u]['weight'] for u in G}

c = gcol.min_cost_k_coloring(G, 3, weight="weight", weights_at="nodes", it_limit=1000)
```

(continues on next page)

(continued from previous page)

```
nx.draw_networkx(G,
                 pos=nx.arf_layout(G),
                 node_color=gcol.get_node_colors(G, c),
                 node_size=100,
                 font_size=8,
                 width=0.25,
                 labels=labels)
plt.show()

U = list(G.nodes[u]["weight"] for u in c if c[u] <= -1)
print("Uncolored nodes have weights", sorted(U), "giving a total cost =", sum(U))
```



```
Uncolored nodes have weights [1, 1, 1, 1, 2] giving a total cost = 6
```

In a similar fashion, we may prefer a solution in which all nodes are assigned to colors, but are willing to allow some clashes in a solution (a clash occurs when the endpoints of an edge have the same color). The aim is to now k-color all nodes while minimizing the number of clashes. Again, we might also choose to add positive weights to the edges, in which case we will seek to minimize the sum of the weights of the clashing edges.

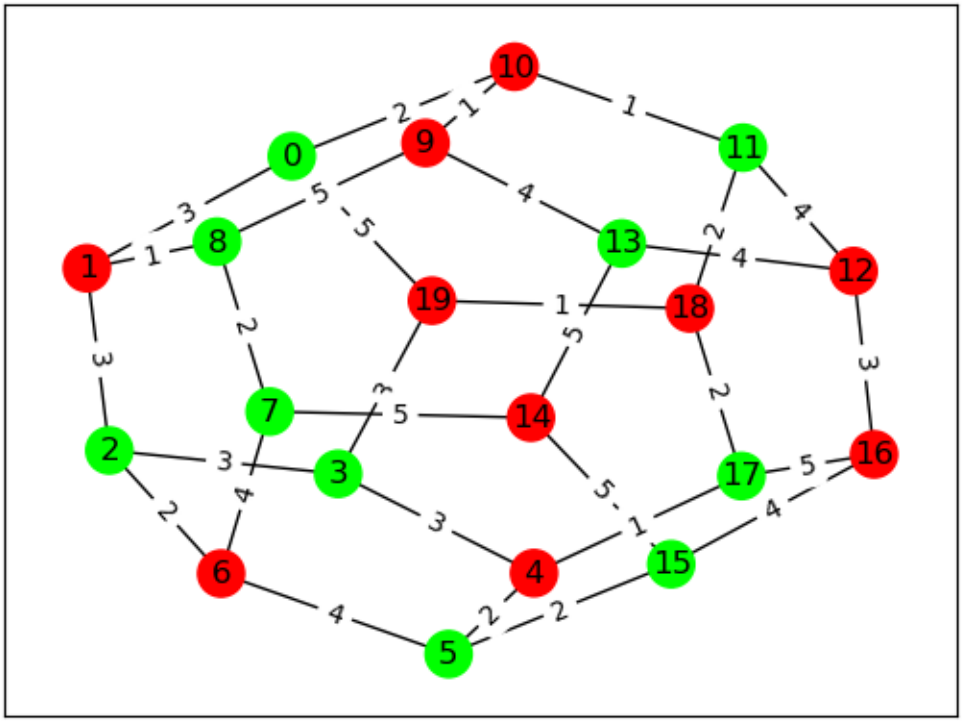
The following example creates a small edge-weighted graph and then produces a node 2-coloring using the routine `gcol.min_cost_k_coloring()`. Six of the edges are causing a clash, giving a total weight of 12.

```
G = nx.dodecahedral_graph()
for u, v in G.edges():
    G.add_edge(u, v, edgeweight=random.randint(1,5))

c = gcol.min_cost_k_coloring(G, 2, weight="edgeweight", weights_at="edges", it_
↪limit=1000)
```

(continues on next page)

(continued from previous page)

```
pos = nx.spring_layout(G, seed=1)
nx.draw_networkx(G, pos=pos, node_color=gcol.get_node_colors(G, c))
labels = nx.get_edge_attributes(G,'edgeweight')
nx.draw_networkx_edge_labels(G, pos=pos, edge_labels=labels)
plt.show()

C = list( (u,v) for (u, v) in G.edges() if c[u]==c[v])
print("The following edges are causing clashes", C, "giving a total cost of",␣
↪sum(G[u][v]["edgeweight"] for (u, v) in C))
```




```
The following edges are causing clashes [(2, 3), (5, 15), (7, 8), (9, 10), (12, 16), (18,
↪ 19)] giving a total cost of 12
```

2.6 Kempe Chains

Given a node coloring of a graph, a Kempe chain is a connected set of nodes that alternate in color. Equivalently, it is a maximal connected subgraph that contains nodes of at most two colors. Interchanging the colors of the nodes in a Kempe chain creates a new coloring that uses the same number of colors, or one fewer color.

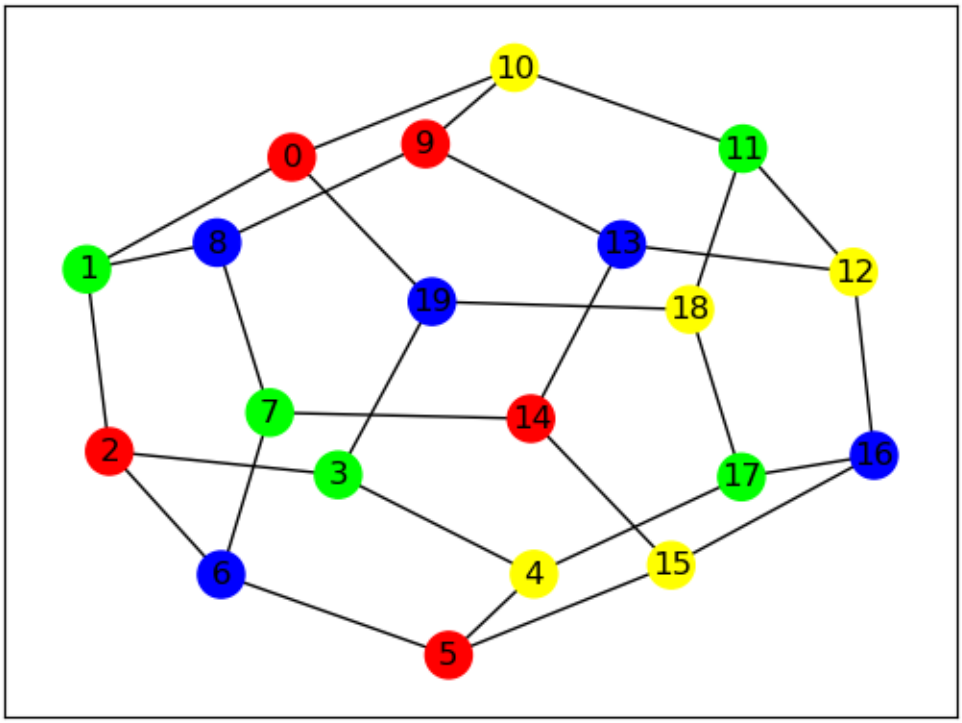
The following example takes a coloring `c` of a graph `G` and determines a Kempe using node 18 (which is yellow) and color 1 (green). The resultant Kempe chain is therefore the connected component of yellow and green nodes that contains node 18. The nodes in this chain are stored in the set `K`. A Kempe chain interchange is then performed, which swaps the colors of the nodes in `K`, leading to the second solution below.

```
G = nx.dodecahedral_graph()
c = gcol.node_k_coloring(G, 4)
nx.draw_networkx(G,
```

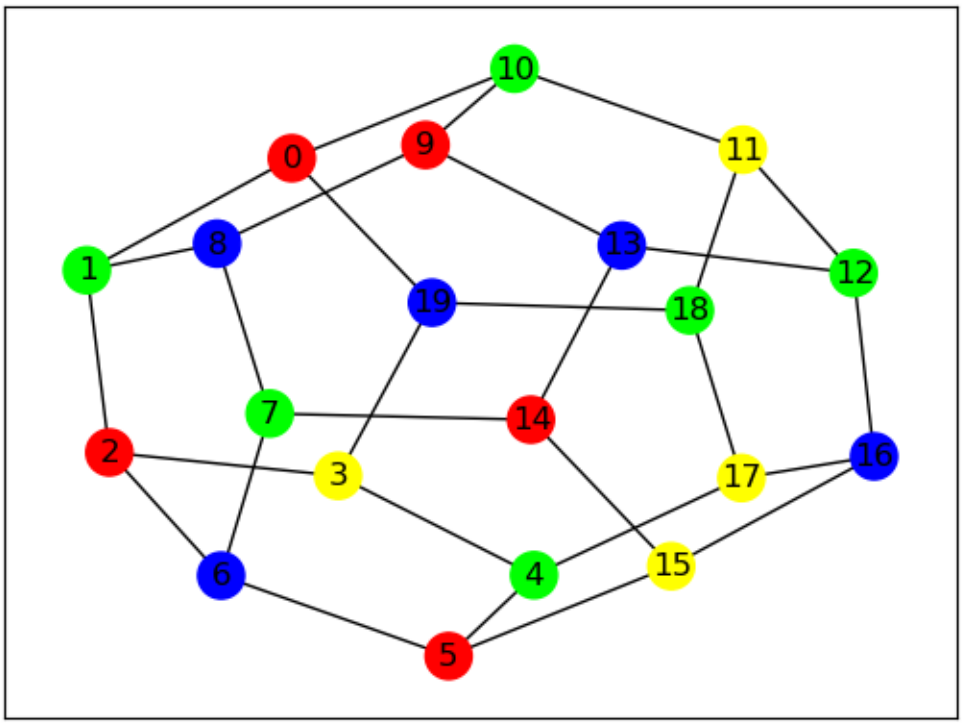
(continues on next page)

(continued from previous page)

```
                pos=nx.spring_layout(G, seed=1),
                node_color=gcol.get_node_colors(G, c))
plt.show()

K = gcol.kempe_chain(G, c, 18, 1)
print("Kempe Chain built from node-18 and color 1 =", K)

#do a Kempe chain interchange
col1 = c[18]
col2 = 1
for v in K:
    if c[v] == col1:
        c[v] = col2
    else:
        c[v] = col1

print("Interchanging the colors of these nodes gives:")
nx.draw_networkx(G,
                pos=nx.spring_layout(G, seed=1),
                node_color=gcol.get_node_colors(G, c))
plt.show()
```




```
Kempe Chain built from node-18 and color 1 = {3, 4, 10, 11, 12, 17, 18}
Interchanging the colors of these nodes gives:
```



2.7 Independent Sets, Cliques and Coverings

In this final section we show how the algorithms of the `gcol` library can be used to find (possibly approximate) solutions to the following three NP-hard optimization problems.

- The *maximum independent set problem* involves determining the largest subset of nodes in a graph G such that none of the nodes in this set are neighboring. The size of the largest independent set in G is known as the *independence number*, denoted by $\alpha(G)$.
- The *minimum node cover problem* involves determining the smallest subset of nodes in G such that every edge in the graph has at least one endpoint from this set.
- The *maximum clique problem* involves determining the largest subset of nodes in G such that every pair of nodes in this set is adjacent. The size of the largest clique in a graph G is known as the *clique number*, denoted by $\omega(G)$.

We can also define weights on the nodes, if desired. If these cases the aims are to now maximize (or minimize) the sum of the weights of the selected nodes.

The following example demonstrates how a large independent set of nodes can be determined in the Les Miserables graph using the `gcol.max_independent_set()` method.

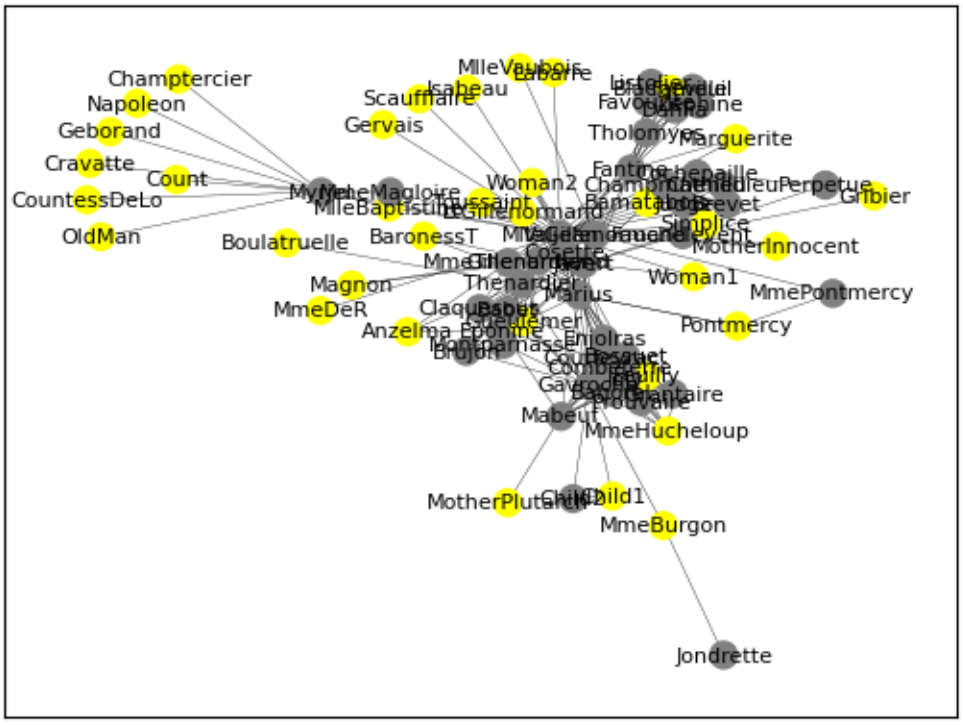
```
G = nx.les_miserables_graph()
S = gcol.max_independent_set(G, it_limit=10000)
print("In the set of", len(G), "Les Miserables characters, there's a subset of", len(S),
↪"characters who never meet.")
nx.draw_networkx(G,
                 nx.spring_layout(G, seed=1),
                 node_color=gcol.get_set_colors(G, S),
                 node_size=100,
                 font_size=8,
```

(continues on next page)

(continued from previous page)

```
                  width=0.25)
plt.show()
```

```
In the set of 77 Les Miserables characters, there's a subset of 35 characters who never␣
↪meet.
```



In the above, the members of the independent set, whose size we have tried to maximize, are shown in yellow. The set of grey nodes, whose size has been minimized, gives us a node covering. Hence, we have determined solutions to both problems.

Large cliques can also be found by using the `gcol.max_independent_set()` routine on the graph's complement. A demonstration of this is shown below.

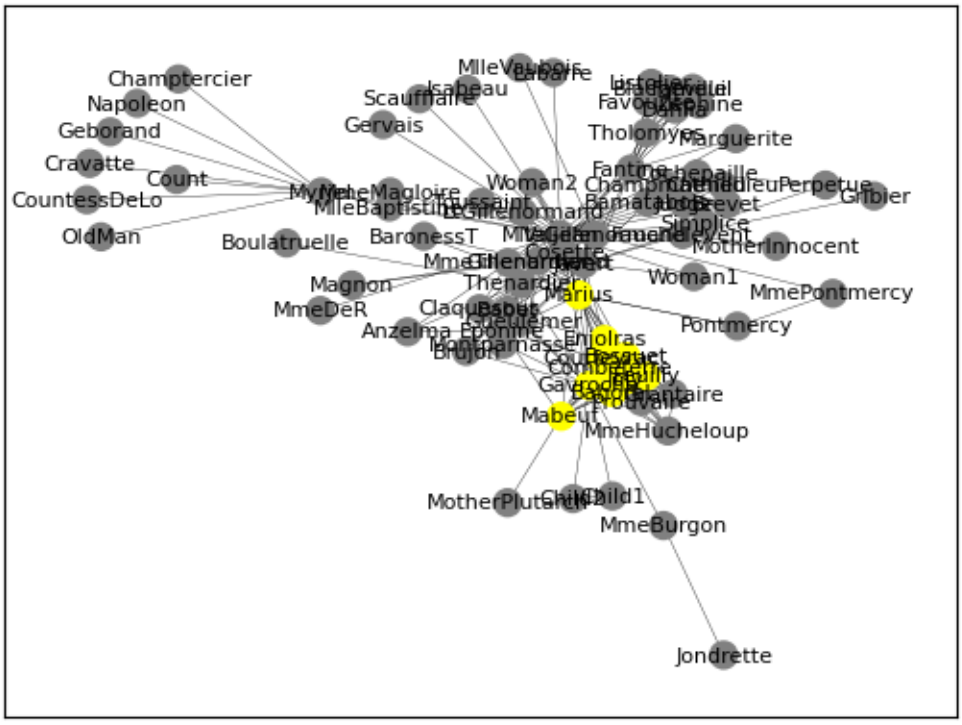
```
S = gcol.max_independent_set(nx.complement(G), it_limit=10000)
print("In the set of", len(G), "Les Miserables characters, there's a subset of", len(S),
↪"characters who form a clique. These are", S)
nx.draw_networkx(G,
                 nx.spring_layout(G, seed=1),
                 node_color=gcol.get_set_colors(G, S),
                 node_size=100,
                 font_size=8,
                 width=0.25)
plt.show()
nx.draw_networkx(G,
                 nx.arf_layout(G),
                 node_color=gcol.get_set_colors(G, S),
                 node_size=100,
                 font_size=8,
```

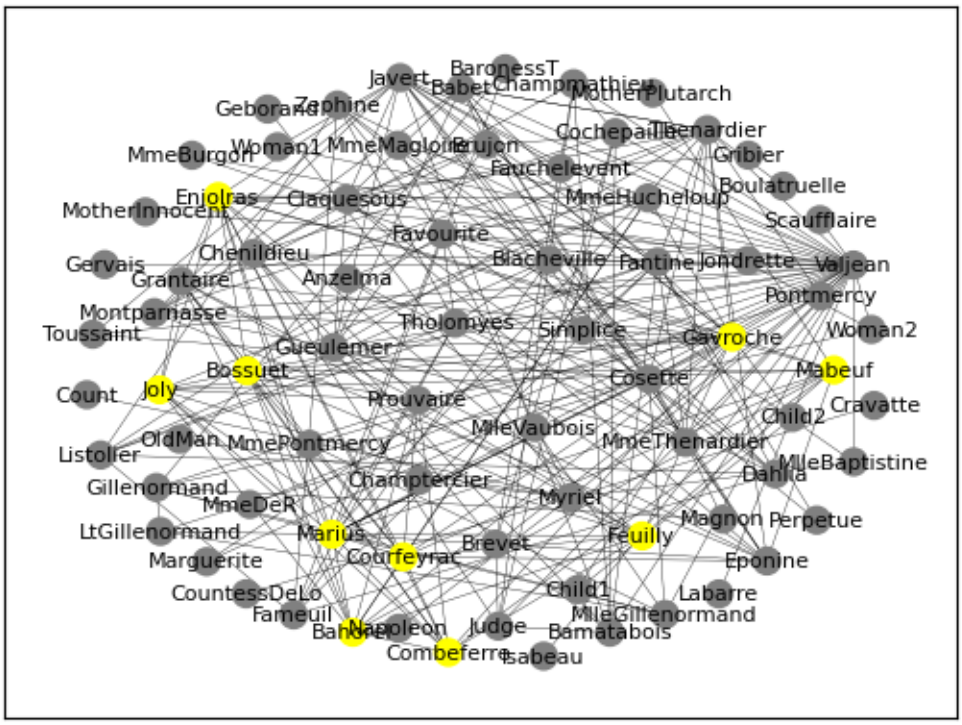
(continues on next page)

(continued from previous page)

```
                width=0.25)
plt.show()
```

```
In the set of 77 Les Miserables characters, there's a subset of 10 characters who form a␣
↪clique. These are ['Combeferre', 'Feuilly', 'Mabeuf', 'Bahorel', 'Joly', 'Courfeyrac',
↪'Bossuet', 'Enjolras', 'Marius', 'Gavroche']
```




BaronessT
Javert
Babet
Champmathieu
MotherPlutarch
Geborand
Zephine
Cochepaille
Thenardier
Woman1
MmeMagloire
Brujon
MmeBurgon
Gribier
Fauchelevent
Boulatruelle
Enjolras
Claquesous
MmeHucheloup
MotherInnocent
Scaufflaire
Favourite
Chenildieu
Blacheville
Fantine
Jondrette
Valjean
Gervais
Grantaire
Anzelma
Pontmercy
Montparnasse
Tholomyes
Simplice
Gavroche
Woman2
Toussaint
Gueulemer
Bossuet
Cosette
Mabeuf
Count
Joly
Prouvaire
Cravatte
Child2
MlleVaubois
OldMan
MmePontmercy
MmeThenardier
Listolier
MlleBaptistine
Dahlia
Gillenormand
Champtercier
Myriel
MmeDeR
Magnon
Perpetue
LtGillenormand
Marius
Brevet
Feuilly
Marguerite
Courfeyrac
Eponine
CountessDeLo
Child1
Labarre
Fameuil
MlleGillenormand
Napoleon
Judge
Bamatabois
Combeferre
Isabeau

CHAPTER

THREE

CASE STUDY: EXAM TIMETABLING

In this section, we consider a practical case study concerning the use of graph coloring methods in the production of exam timetables at a university.

3.1 Background

Each year, a college needs to produce an exam timetable for its students. The aim is to assign each exam to a "timeslot" while trying to avoid "clashes". (A clash occurs when there are one or more students who need to attend a pair of exams, but these exams have been assigned to the same timeslot and therefore occur at the same time.)

The problem is specified using an $n \times n$ symmetrical matrix X, where n is the number of exams to schedule. An element X_{ij} in this matrix gives the number of students who need to sit both exams i and j. Also, an element X_{ii} gives the total number of students who need to sit exam i. For example, in the following matrix:

$$\begin{pmatrix} 9 & 0 & 0 & 1 & 0 & 0 & 6 & 0 \\ 0 & 4 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 9 & 0 & 4 & 0 & 0 & 0 \\ 1 & 0 & 0 & 3 & 0 & 0 & 0 & 1 \\ 0 & 0 & 4 & 0 & 8 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 & 0 & 4 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 & 9 & 2 \\ 0 & 1 & 0 & 1 & 3 & 0 & 2 & 8 \end{pmatrix}$$

- There are $n = 8$ exams (labelled 0 to 7),
- There is one student who needs to sit exams 0 and 3 (because $X_{0,3} = 1$),
- There are six students who need to sit exams 0 and 7 (because $X_{0,7} = 6$),
- Nine students are sitting exam 0 (because $X_{0,0} = 9$), and so on.

In this matrix, zeros indicate that no students need to sit the two corresponding exams. This means that the pair of exams can be assigned to the same timeslot, and no students will be inconvenienced.

A college administrator has been put in charge of creating this year's timetable, which involves $n = 139$ exams. The full details of this problem are given to her in the file timetable.txt, which contains an $n \times n$ symmetrical matrix as described above. She quickly realizes that this problem is too large to solve by hand but can be handled using graph coloring techniques, using nodes for exams and colors for timeslots.

To start, our administrator reads the file timetable.txt into a matrix X and creates an n-node graph where each node corresponds to an exam. Nodes in this graph are then made adjacent whenever the corresponding pair of exams has a common student. That is, nodes v_i and v_j are made adjacent if and only if $X_{ij} > 0$ and $i \neq j$. Having done this, the administrator determines the chromatic number of this graph. This corresponds to the minimum number of timeslots that are needed to construct a clash-free timetable.

```
import networkx as nx
import gcol
import matplotlib.pyplot as plt

#Read in the text file and store in matrix X
X = []
with open('timetable.txt','r') as f:
    n = int(f.readline())
    for i in range(n):
        line = f.readline().split(",")
        line = [int(x) for x in line]
        X.append(line)

#Construct the graph G
G = nx.Graph()
G.add_nodes_from([i for i in range(n) if X[i][i] > 0])
for i in range(n-1):
    for j in range(i+1, n):
        if X[i][j] > 0:
            G.add_edge(i, j)
print("Constructed a", G)
print("Minimum number of timeslots needed for a clash free timetable =", gcol.chromatic_
↪number(G))
```

```
Constructed a Graph with 139 nodes and 1381 edges
Minimum number of timeslots needed for a clash free timetable = 13
```

The results reveal that clash-free timetables are only possible when 13 or more timeslots are used to schedule these exams.

In previous years, the college has constructed the timetable by taking the largest 15 exams and assigning each one to a different timeslot. The remaining exams are then added to the timetable, creating new timeslots where needed. The administrator thinks this could be a way forward and therefore uses `gcol`'s precoloring routines to emulate this process.

```
#Get a list of exams, sorted by size, and assign the first 15 to different colors
sizeExam = [(X[i][i], i) for i in range(n)]
sizeExam.sort(reverse=True)
print("Here are the sizes of each exam")
print(sizeExam)

P = {}
for i in range(15):
    print("Exam", sizeExam[i][1], "has", sizeExam[i][0], "students. Assigning to timeslot
↪", i)
    P[sizeExam[i][1]] = i

c = gcol.node_precoloring(G, P, opt_alg=1)
P = gcol.partition(c)
print("Here are the exams assigned to each timeslot")
for i in range(len(P)):
    print("Timeslot", i, ":", P[i])
```

```
Here are the sizes of each exam
[(236, 71), (208, 137), (208, 134), (208, 96), (208, 70), (208, 2), (127, 107), (121, 5),
↪ (119, 109), (118, 105), (117, 73), (110, 106), (101, 68), (90, 28), (89, 132), (89,␣
↪104), (87, 103), (84, 3), (77, 135), (74, 133), (73, 102), (71, 131), (67, 48), (61,␣
↪138), (39, 26), (35, 129), (35, 97), (34, 128), (34, 127), (34, 93), (34, 92), (34,␣
↪66), (34, 65), (34, 46), (34, 45), (34, 25), (34, 24), (33, 69), (32, 126), (32, 121),␣
↪(32, 94), (32, 91), (32, 86), (32, 64), (32, 59), (32, 44), (32, 39), (32, 23), (32,␣
↪18), (31, 99), (30, 111), (30, 98), (30, 76), (30, 49), (30, 29), (30, 8), (29, 120),␣
↪(29, 113), (29, 100), (29, 85), (29, 78), (29, 58), (29, 51), (29, 38), (29, 31), (29,␣
↪17), (29, 10), (28, 136), (28, 118), (28, 117), (28, 115), (28, 83), (28, 82), (28,␣
↪80), (28, 56), (28, 55), (28, 53), (28, 36), (28, 35), (28, 33), (28, 15), (28, 14),␣
↪(28, 12), (27, 110), (27, 101), (27, 74), (27, 6), (26, 112), (26, 77), (26, 50), (26,␣
↪30), (26, 9), (24, 116), (24, 81), (24, 54), (24, 34), (24, 13), (23, 1), (21, 95),␣
↪(20, 122), (20, 87), (20, 60), (20, 40), (20, 19), (19, 125), (19, 124), (19, 90), (19,
↪ 89), (19, 63), (19, 62), (19, 43), (19, 42), (19, 22), (19, 21), (13, 108), (12, 67),␣
↪(12, 0), (10, 123), (10, 88), (10, 61), (10, 41), (10, 20), (9, 114), (9, 79), (9, 52),
↪ (9, 32), (9, 11), (8, 7), (7, 75), (7, 72), (2, 4), (1, 47), (0, 130), (0, 119), (0,␣
↪84), (0, 57), (0, 37), (0, 27), (0, 16)]
Exam 71 has 236 students. Assigning to timeslot 0
Exam 137 has 208 students. Assigning to timeslot 1
Exam 134 has 208 students. Assigning to timeslot 2
Exam 96 has 208 students. Assigning to timeslot 3
Exam 70 has 208 students. Assigning to timeslot 4
Exam 2 has 208 students. Assigning to timeslot 5
Exam 107 has 127 students. Assigning to timeslot 6
Exam 5 has 121 students. Assigning to timeslot 7
Exam 109 has 119 students. Assigning to timeslot 8
Exam 105 has 118 students. Assigning to timeslot 9
Exam 73 has 117 students. Assigning to timeslot 10
Exam 106 has 110 students. Assigning to timeslot 11
Exam 68 has 101 students. Assigning to timeslot 12
Exam 28 has 90 students. Assigning to timeslot 13
Exam 132 has 89 students. Assigning to timeslot 14
Here are the exams assigned to each timeslot
Timeslot 0 : [47, 71, 76, 77, 78, 79, 80, 81, 82, 83, 131]
Timeslot 1 : [133, 135, 137]
Timeslot 2 : [120, 121, 122, 124, 125, 126, 127, 128, 134]
Timeslot 3 : [17, 18, 19, 21, 22, 23, 24, 25, 41, 94, 95, 96, 99, 119]
Timeslot 4 : [37, 70, 102, 103, 138]
Timeslot 5 : [2, 85, 86, 87, 89, 90, 91, 92, 93, 108, 110, 123]
Timeslot 6 : [0, 1, 104, 107]
Timeslot 7 : [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
Timeslot 8 : [109]
Timeslot 9 : [58, 59, 60, 62, 63, 64, 65, 66, 88, 105, 129]
Timeslot 10 : [20, 26, 49, 50, 51, 52, 53, 54, 55, 56, 57, 72, 73, 74, 75]
Timeslot 11 : [106]
Timeslot 12 : [38, 39, 40, 42, 43, 44, 45, 46, 61, 67, 68, 69]
Timeslot 13 : [27, 28, 48, 84, 97, 98, 100, 101, 111, 112, 113, 114, 115, 116, 117, 118]
Timeslot 14 : [29, 30, 31, 32, 33, 34, 35, 36, 130, 132, 136]
```

Despite this process working satisfactorily, the administrator decides that it is overly complex and abandons it. She also notices that the seven smallest exams in the dataset have no attending students, so she decides to remove them from the graph in future calculations. She is also worried that some timeslots might contain too many exams and that the

university will not have enough seats available in these cases. As a result, she decides to try and balance the number of students sitting exams in each timeslot.

To do this, she creates a similar graph as above but specifies a weight for each node, which gives the size of the corresponding exam. She also ignores the seven empty exams. As shown, she is now able to produce a clash-free 13-timeslot solution in which the number of students per-timeslot ranges between 417 and 442.

```
#Construct the node-weighted graph G
G = nx.Graph()
for i in range(n):
    if X[i][i] > 0:
        G.add_node(i, weight=X[i][i])
for i in range(n-1):
    for j in range(i+1, n):
        if X[i][i] > 0 and X[j][j] > 0 and X[i][j] > 0:
            G.add_edge(i, j)

c = gcol.equitable_node_k_coloring(G, 13, weight="weight")
P = gcol.partition(c)
print("Here are the exams assigned to each timeslot")
for i in range(len(P)):
    print("Timeslot", i, ":", P[i])
for j in range(len(P)):
    Wj = [G.nodes[v]["weight"] for v in P[j]]
    print("Number of students in timeslot", j, "=", sum(Wj))
```

```
Here are the exams assigned to each timeslot
Timeslot 0 : [3, 72, 73, 74, 134]
Timeslot 1 : [0, 1, 8, 13, 14, 17, 18, 19, 21, 22, 23, 24, 25, 33, 41, 51, 56]
Timeslot 2 : [67, 68, 69, 104, 105, 132]
Timeslot 3 : [5, 6, 7, 96, 135]
Timeslot 4 : [20, 26, 70, 102, 103]
Timeslot 5 : [12, 32, 47, 75, 78, 106, 107, 111, 117, 118, 129]
Timeslot 6 : [2, 108, 109, 110, 138]
Timeslot 7 : [4, 28, 97, 98, 100, 101, 137]
Timeslot 8 : [9, 11, 15, 29, 35, 71, 133]
Timeslot 9 : [36, 38, 39, 40, 42, 43, 44, 45, 46, 53, 79, 82, 112, 116, 131]
Timeslot 10 : [10, 52, 58, 59, 60, 62, 63, 64, 65, 66, 76, 77, 80, 81, 83, 88, 136]
Timeslot 11 : [30, 31, 48, 54, 55, 61, 85, 86, 87, 89, 90, 91, 92, 93, 115]
Timeslot 12 : [34, 49, 50, 94, 95, 99, 113, 114, 120, 121, 122, 123, 124, 125, 126, 127,␣
↪128]
Number of students in timeslot 0 = 443
Number of students in timeslot 1 = 431
Number of students in timeslot 2 = 442
Number of students in timeslot 3 = 441
Number of students in timeslot 4 = 417
Number of students in timeslot 5 = 432
Number of students in timeslot 6 = 428
Number of students in timeslot 7 = 421
Number of students in timeslot 8 = 431
Number of students in timeslot 9 = 433
Number of students in timeslot 10 = 431
Number of students in timeslot 11 = 431
Number of students in timeslot 12 = 431
```

At this point, the administrator is feeling rather pleased with herself: she has produced a 13-timeslot solution, proved that this is the minimum number of timeslots needed, and found a nice balance of students per-timeslot. She is somewhat irritated, then, when the college manager tells her that there is ample seating capacity, but only twelve timeslots. The latter means that the timetable will either need to have some clashes or some unscheduled exams.

To investigate this, she first constructs a solution that seeks to minimize the total size of unscheduled exams. To do this, she uses the same node-weighted graph as above, but makes use of the `gcol.min_cost_k_coloring()` routine. This leads to the following solution:

```
c = gcol.min_cost_k_coloring(G, 12, weight="weight", weights_at="nodes", it_limit=10000)
U = list(G.nodes[u]["weight"] for u in c if c[u] <= -1)
print("Here are the exams assigned to each timeslot")
for i in range(len(P)):
    print("Timeslot", i, ":", P[i])
print("The", len(U), "unscheduled exams have the following sizes", sorted(U), ", which␣
↪totals", sum(U))
```

```
Here are the exams assigned to each timeslot
Timeslot 0 : [3, 72, 73, 74, 134]
Timeslot 1 : [0, 1, 8, 13, 14, 17, 18, 19, 21, 22, 23, 24, 25, 33, 41, 51, 56]
Timeslot 2 : [67, 68, 69, 104, 105, 132]
Timeslot 3 : [5, 6, 7, 96, 135]
Timeslot 4 : [20, 26, 70, 102, 103]
Timeslot 5 : [12, 32, 47, 75, 78, 106, 107, 111, 117, 118, 129]
Timeslot 6 : [2, 108, 109, 110, 138]
Timeslot 7 : [4, 28, 97, 98, 100, 101, 137]
Timeslot 8 : [9, 11, 15, 29, 35, 71, 133]
Timeslot 9 : [36, 38, 39, 40, 42, 43, 44, 45, 46, 53, 79, 82, 112, 116, 131]
Timeslot 10 : [10, 52, 58, 59, 60, 62, 63, 64, 65, 66, 76, 77, 80, 81, 83, 88, 136]
Timeslot 11 : [30, 31, 48, 54, 55, 61, 85, 86, 87, 89, 90, 91, 92, 93, 115]
Timeslot 12 : [34, 49, 50, 94, 95, 99, 113, 114, 120, 121, 122, 123, 124, 125, 126, 127,␣
↪128]
The 8 unscheduled exams have the following sizes [19, 19, 20, 29, 32, 32, 34, 34] ,␣
↪which totals 219
```

As shown, this gives a 12-timeslot solution but leaves 8 unscheduled exams.

She then tries to minimize the number of clashes instead by forming an edge-weighted graph in which each edge $\{v_i, v_j\}$ has a weight equal to X_{ij}.

```
#Construct the edge-weighted graph G
G = nx.Graph()
for i in range(n):
    if X[i][i] > 0:
        G.add_node(i)
for i in range(n-1):
    for j in range(i+1, n):
        if X[i][i] > 0 and X[j][j] > 0 and X[i][j] > 0:
            G.add_edge(i, j, weight=X[i][j])

c = gcol.min_cost_k_coloring(G, 12, weight="weight", weights_at="edges", it_limit=10000)
print("Here are the exams assigned to each timeslot")
for i in range(len(P)):
    print("Timeslot", i, ":", P[i])
```

(continues on next page)

(continued from previous page)

```
print("Here are the clashes in this timetable:")
for u, v in G.edges():
    if c[u] == c[v]:
        print("Exams", u , "and", v, "assigned to timeslot", c[v], "but have", X[u][v],
↪"common student(s)")
```

```
Here are the exams assigned to each timeslot
Timeslot 0 : [3, 72, 73, 74, 134]
Timeslot 1 : [0, 1, 8, 13, 14, 17, 18, 19, 21, 22, 23, 24, 25, 33, 41, 51, 56]
Timeslot 2 : [67, 68, 69, 104, 105, 132]
Timeslot 3 : [5, 6, 7, 96, 135]
Timeslot 4 : [20, 26, 70, 102, 103]
Timeslot 5 : [12, 32, 47, 75, 78, 106, 107, 111, 117, 118, 129]
Timeslot 6 : [2, 108, 109, 110, 138]
Timeslot 7 : [4, 28, 97, 98, 100, 101, 137]
Timeslot 8 : [9, 11, 15, 29, 35, 71, 133]
Timeslot 9 : [36, 38, 39, 40, 42, 43, 44, 45, 46, 53, 79, 82, 112, 116, 131]
Timeslot 10 : [10, 52, 58, 59, 60, 62, 63, 64, 65, 66, 76, 77, 80, 81, 83, 88, 136]
Timeslot 11 : [30, 31, 48, 54, 55, 61, 85, 86, 87, 89, 90, 91, 92, 93, 115]
Timeslot 12 : [34, 49, 50, 94, 95, 99, 113, 114, 120, 121, 122, 123, 124, 125, 126, 127,␣
↪128]
Here are the clashes in this timetable:
Exams 26 and 138 assigned to timeslot 4 but have 1 common student(s)
```

As shown, this leads to a 12-timeslot solution in which only one student is affected by a clash. She submits this solution to her manager who is so pleased, he gives her a promotion.

CHAPTER

FOUR

PERFORMANCE ANALYSIS

In this notebook we analyze the behavior of the various node coloring algorithms in the `gcol` library. Where appropriate, we also make comparisons to similar algorithms from the `networkx` library.

In this analysis, algorithms are evaluated by looking at solution quality and run times. Details on asymptotic algorithm complexity (in terms of big O notation) can be found in `gcol`'s documentation. Here, all tests are conducted on randomly generated Erdos-Renyi graphs, commonly denoted by $G(n, p)$. These graphs are constructed by taking n nodes and adding an edge between each node pair at random with probability p. The expected number of edges in a $G(n, p)$ graph is therefore $\binom{n}{2}p$, while the expected node degree is $(n-1)p$.

For these tests, we use differing values for n but keep p fixed at 0.5. This is due to a result of Nick Wormald, who has established that for $n \gtrapprox 30$, a set of randomly constructed $G(n, 0.5)$ graphs can be considered equivalent to a random sample from the population of *all* n-node graphs. Note, however, that although this allows us to make general statistical statements about performance, different observations may well be made when executing these algorithms on specifically chosen topologies, such as scale-free graphs and planar graphs. Examples of these differences are discussed in this book.

In the code below, each trial involves generating a set of $G(n, 0.5)$ graphs using a range of values of n. The results of the algorithms are then written to a Pandas dataframe `df`, and this data is summarized in charts and pivot tables. Lines in the charts give mean values, while the shaded areas indicate one standard deviation on either side of these means. All results below were found by executing the code on a 3.0 GHtz Windows 11 PC with 16 GB of RAM.

4.1 Differing Node Coloring Strategies

In our first experiment, we compare the different constructive strategies available for node coloring in the `gcol` library (namely `'random'`, `'welsh_powell'`, `'dsatur'`, and `'rlf'`) using $G(n, 0.5)$ graphs with values of n between 50 and 500. The results are shown in the pivot table and charts below. Further details on these algorithms can be found in `gcol`'s documentation.

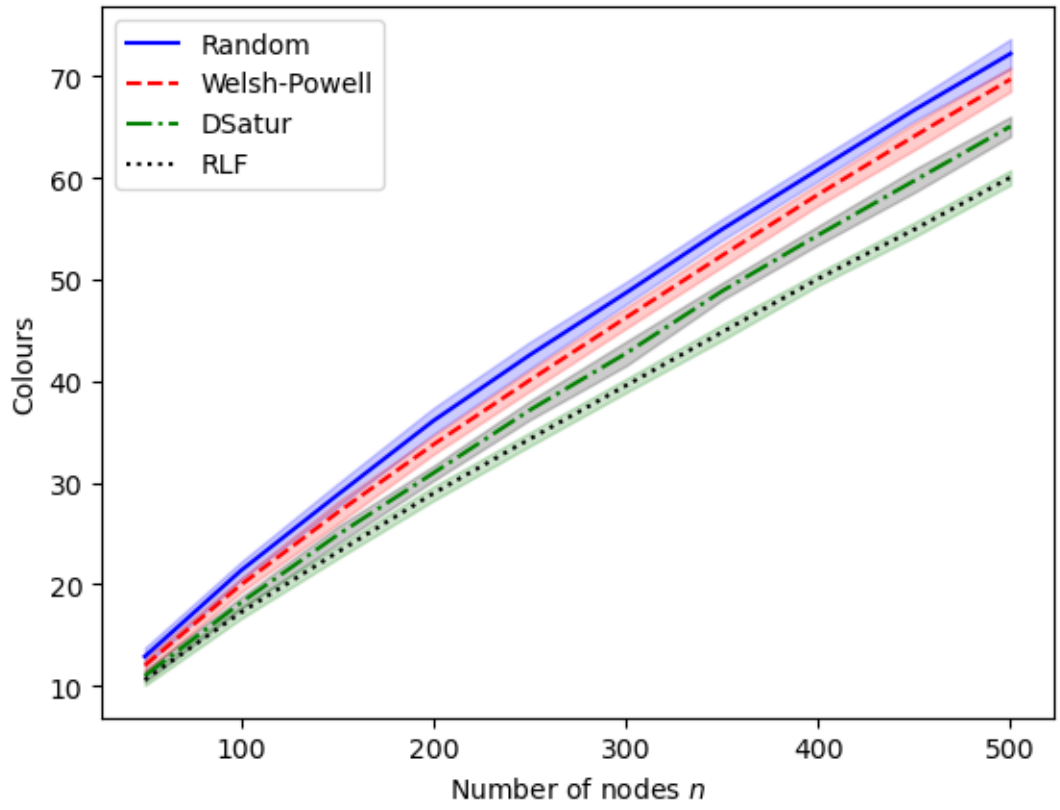
```
import pandas as pd
import networkx as nx
import gcol
import matplotlib.pyplot as plt
import time

#Carry out the trials and put the results into a list
results = []
nVals = range(50,501,50)
for n in nVals:
    for seed in range(50):
        G = nx.gnp_random_graph(n, 0.5, seed)
        for strategy in ["random", "welsh_powell", "dsatur", "rlf"]:
```

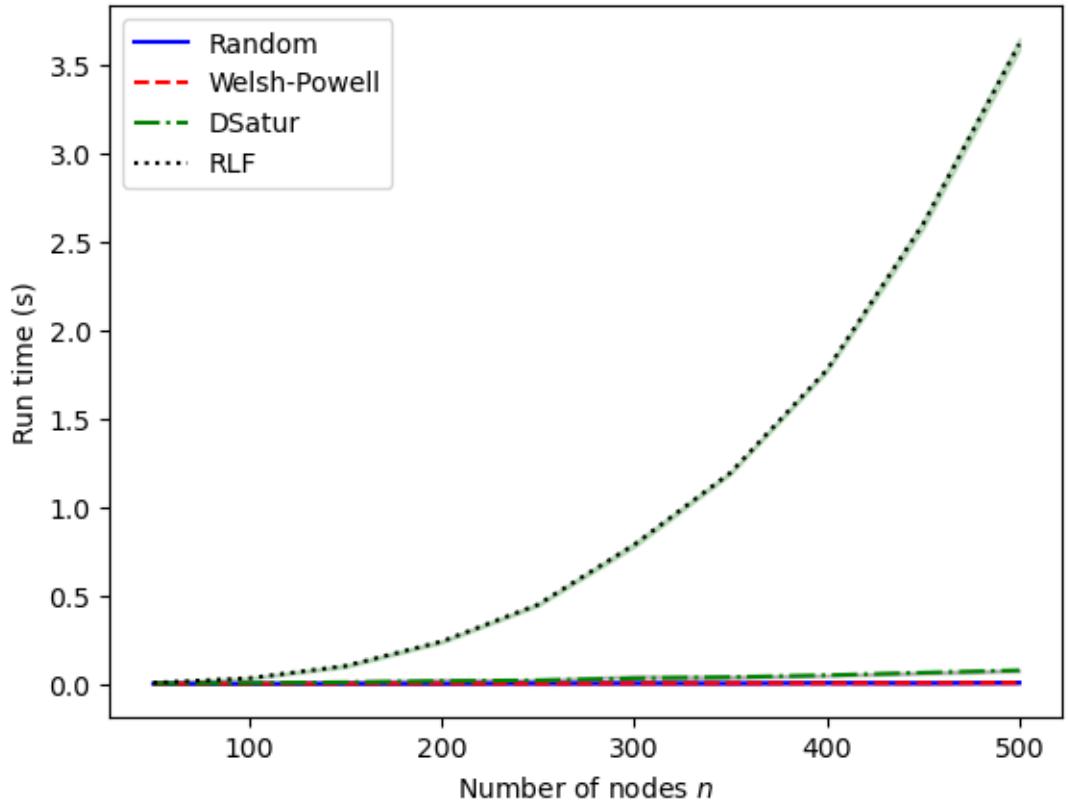
(continues on next page)

(continued from previous page)

```
            start = time.time()
            c = gcol.node_coloring(G, strategy)
            results.append([n, seed, strategy, max(c.values()) + 1, time.time()-start])

# Create a pandas dataframe from this list, make a pivot table, and display it
df = pd.DataFrame(results, columns=["n", "seed", "strategy", "cols", "time"])
pivot = df.pivot_table(columns='strategy', aggfunc=['mean','std'], values=['cols','time
↪'], index='n')
display(pivot)
```

```
# Now use the pivot table above to make a chart that compares mean solution quality
mean1, SD1 = pivot[("mean","cols","random")], pivot[("std","cols","random")]
mean2, SD2 = pivot[("mean","cols","welsh_powell")], pivot[("std","cols","welsh_powell")]
mean3, SD3 = pivot[("mean","cols","dsatur")], pivot[("std","cols","dsatur")]
mean4, SD4 = pivot[("mean","cols","rlf")], pivot[("std","cols","rlf")]
plt.plot(nVals, mean1, linestyle='-', linewidth=1.5, color="b", label='Random')
plt.fill_between(nVals, mean1-SD1, mean1+SD1, color='b', alpha=0.2)
plt.plot(nVals, mean2, linestyle='--', linewidth=1.5, color="r", label='Welsh-Powell')
plt.fill_between(nVals, mean2-SD2, mean2+SD2, color='r', alpha=0.2)
plt.plot(nVals, mean3, linestyle='-.', linewidth=1.5, color="g", label='DSatur')
plt.fill_between(nVals, mean3-SD3, mean3+SD3, color='k', alpha=0.2)
plt.plot(nVals, mean4, linestyle=':', linewidth=1.5, color="black", label='RLF')
plt.fill_between(nVals, mean4-SD4, mean4+SD4, color='g', alpha=0.2)
plt.xlabel("Number of nodes $n$")
plt.ylabel("Colours")
plt.legend()
plt.show()

# and do the same for mean run times
mean1, SD1 = pivot[("mean","time","random")], pivot[("std","time","random")]
mean2, SD2 = pivot[("mean","time","welsh_powell")], pivot[("std","time","welsh_powell")]
mean3, SD3 = pivot[("mean","time","dsatur")], pivot[("std","time","dsatur")]
mean4, SD4 = pivot[("mean","time","rlf")], pivot[("std","time","rlf")]
plt.plot(nVals, mean1, linestyle='-', linewidth=1.5, color="b", label='Random')
plt.fill_between(nVals, mean1-SD1, mean1+SD1, color='b', alpha=0.2)
plt.plot(nVals, mean2, linestyle='--', linewidth=1.5, color="r", label='Welsh-Powell')
plt.fill_between(nVals, mean2-SD2, mean2+SD2, color='r', alpha=0.2)
plt.plot(nVals, mean3, linestyle='-.', linewidth=1.5, color="g", label='DSatur')
plt.fill_between(nVals, mean3-SD3, mean3+SD3, color='k', alpha=0.2)
plt.plot(nVals, mean4, linestyle=':', linewidth=1.5, color="black", label='RLF')
plt.fill_between(nVals, mean4-SD4, mean4+SD4, color='g', alpha=0.2)
plt.xlabel("Number of nodes $n$")
plt.ylabel("Run time (s)")
plt.legend()
plt.show()
```


Random
Welsh-Powell
DSatur
RLF
Colours
Number of nodes n
10
20
30
40
50
60
70
100
200
300
400
500


Random
Welsh-Powell
DSatur
RLF
Run time (s)
Number of nodes n
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
100
200
300
400
500

The results above show that the `random` and `welsh-powell` strategies produce the poorest solutions overall (in terms of the number of colors they use) while the RLF algorithm produces the best. This gap also seems to widen for larger values of n. On the other hand, the RLF algorithm has less favorable run times, as shown in the second chart. This is to be expected because the RLF algorithm has a higher complexity than the others. A good compromise seems to be struck by the `dsatur` strategy, which features comparatively good solution quality and run times.

4.2 Comparison to NetworkX

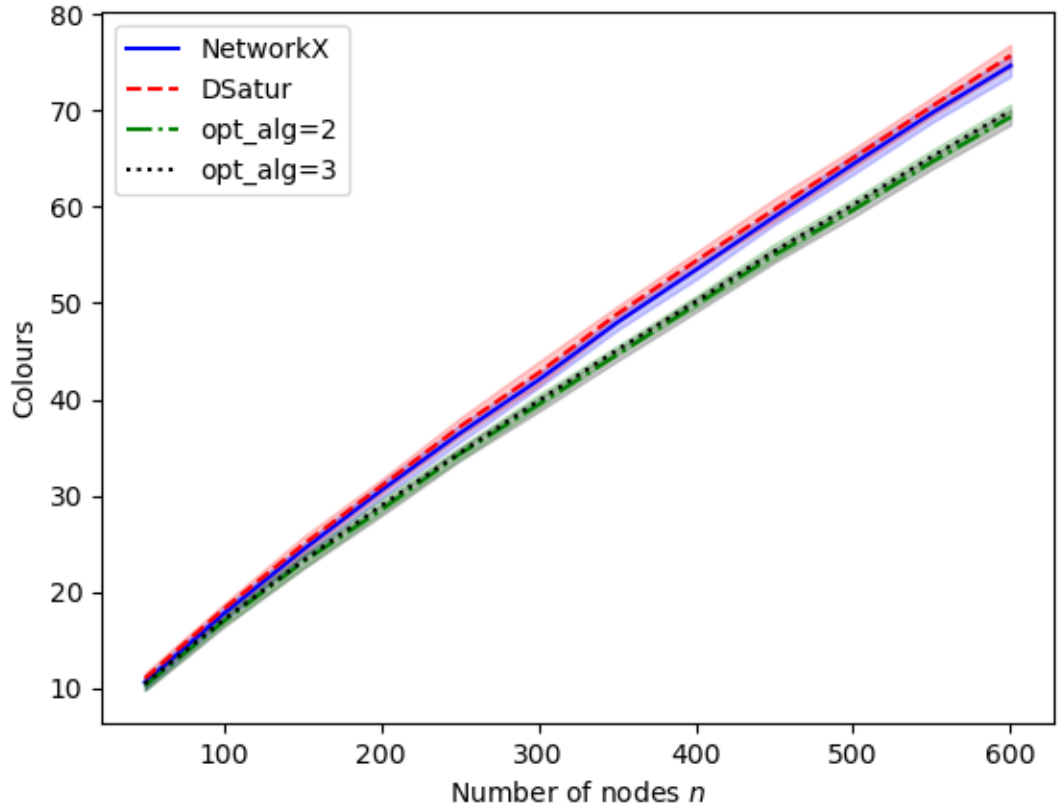
The next set of experiments compares the performance of `gcol`'s local search routines and NetworkX's interchange coloring routine. As a benchmark, we also include `gcol`'s `dsatur` option from earlier, which has also been used to produce the initial solutions for the local search algorithms. For comparative purposes, both of `gcol`'s local search algorithms (`opt_alg=2` and `opt_alg=3`) are used here, and we impose a fixed iteration limit of n. The results are collected and displayed in the same manner as the previous example.

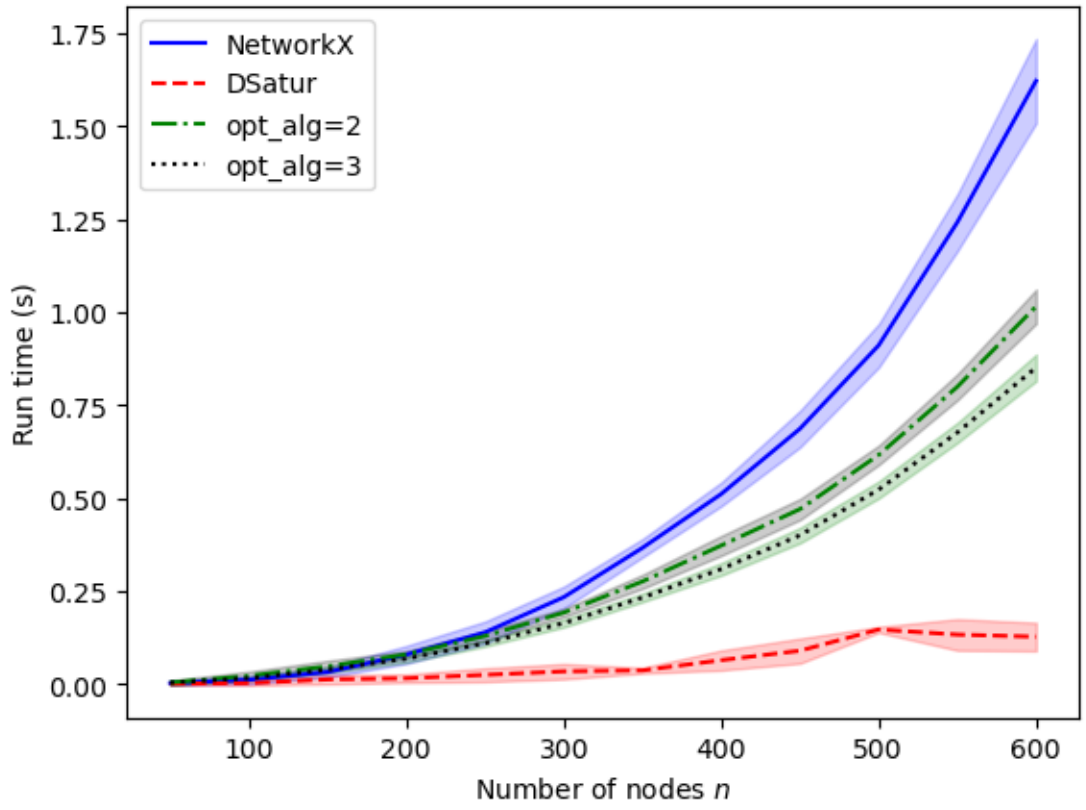
```
#Carry out the trials and put the results into a list
results = []
nVals = range(50,601,50)
for n in nVals:
    for seed in range(50):
        G = nx.gnp_random_graph(n, 0.5, seed)
        start = time.time()
        c = nx.greedy_color(G, "largest_first", interchange=True)
        results.append([n, seed, "networkx", max(c.values()) + 1, time.time()-start])
        start = time.time()
        c = gcol.node_coloring(G)
        results.append([n, seed, "dsatur", max(c.values()) + 1, time.time()-start])
        start = time.time()
        c = gcol.node_coloring(G, opt_alg=2, it_limit=len(G))
        results.append([n, seed, "opt_alg=2", max(c.values()) + 1, time.time()-start])
        start = time.time()
        c = gcol.node_coloring(G, opt_alg=3, it_limit=len(G))
        results.append([n, seed, "opt_alg=3", max(c.values()) + 1, time.time()-start])

# Create a pandas dataframe from this list and make a pivot table
df = pd.DataFrame(results, columns=["n", "seed", "alg", "cols", "time"])
pivot = df.pivot_table(columns='alg', aggfunc=['mean','std'], values=['cols','time'],␣
↪index='n')
display(pivot)

# Use the pivot table to make charts as before
mean1, SD1 = pivot[("mean","cols","networkx")], pivot[("std","cols","networkx")]
mean2, SD2 = pivot[("mean","cols","dsatur")], pivot[("std","cols","dsatur")]
mean3, SD3 = pivot[("mean","cols","opt_alg=2")], pivot[("std","cols","opt_alg=2")]
mean4, SD4 = pivot[("mean","cols","opt_alg=3")], pivot[("std","cols","opt_alg=3")]
plt.plot(nVals, mean1, linestyle='-', linewidth=1.5, color="b", label='NetworkX')
plt.fill_between(nVals, mean1-SD1, mean1+SD1, color='b', alpha=0.2)
plt.plot(nVals, mean2, linestyle='--', linewidth=1.5, color="r", label='DSatur')
plt.fill_between(nVals, mean2-SD2, mean2+SD2, color='r', alpha=0.2)
plt.plot(nVals, mean3, linestyle='-.', linewidth=1.5, color="g", label='opt_alg=2')
plt.fill_between(nVals, mean3-SD3, mean3+SD3, color='k', alpha=0.2)
plt.plot(nVals, mean4, linestyle=':', linewidth=1.5, color="black", label='opt_alg=3')
plt.fill_between(nVals, mean4-SD4, mean4+SD4, color='g', alpha=0.2)
plt.xlabel("Number of nodes $n$")
```

(continues on next page)

(continued from previous page)

```
plt.ylabel("Colours")
plt.legend()
plt.show()

mean1, SD1 = pivot[("mean","time","networkx")], pivot[("std","time","networkx")]
mean2, SD2 = pivot[("mean","time","dsatur")], pivot[("std","time","dsatur")]
mean3, SD3 = pivot[("mean","time","opt_alg=2")], pivot[("std","time","opt_alg=2")]
mean4, SD4 = pivot[("mean","time","opt_alg=3")], pivot[("std","time","opt_alg=3")]
plt.plot(nVals, mean1, linestyle='-', linewidth=1.5, color="b", label='NetworkX')
plt.fill_between(nVals, mean1-SD1, mean1+SD1, color='b', alpha=0.2)
plt.plot(nVals, mean2, linestyle='--', linewidth=1.5, color="r", label='DSatur')
plt.fill_between(nVals, mean2-SD2, mean2+SD2, color='r', alpha=0.2)
plt.plot(nVals, mean3, linestyle='-.', linewidth=1.5, color="g", label='opt_alg=2')
plt.fill_between(nVals, mean3-SD3, mean3+SD3, color='k', alpha=0.2)
plt.plot(nVals, mean4, linestyle=':', linewidth=1.5, color="black", label='opt_alg=3')
plt.fill_between(nVals, mean4-SD4, mean4+SD4, color='g', alpha=0.2)
plt.xlabel("Number of nodes $n$")
plt.ylabel("Run time (s)")
plt.legend()
plt.show()
```





It is clear from the above results that the local search algorithms make significant improvements to the solutions provided by the `dsatur` strategy, albeit with additional time requirements. The solutions and run times of these local search algorithms are also superior to NetworkX's node coloring routines. Note that further improvements in solution quality might also be found by increasing the iteration limit of the local search algorithms.

4.3 Exact Algorithm Performance

In addition to two local search heuristics, the `gcol` library also features an exact, exponential-time algorithm for node coloring, based on backtracking. This algorithm is invoked by setting `opt_alg=1`. At the start of this algorithm's execution, a large clique C is identified in G using the NetworkX function `nx.max_clique()`. The nodes of C are then permanently assigned to different colors. The main backtracking algorithm is then executed and only halts only when a solution using C colors has been identified, or when the algorithm has backtracked to the root of the search tree. In both cases the returned solution will be optimal (that is, will be using the minimum number of colors).

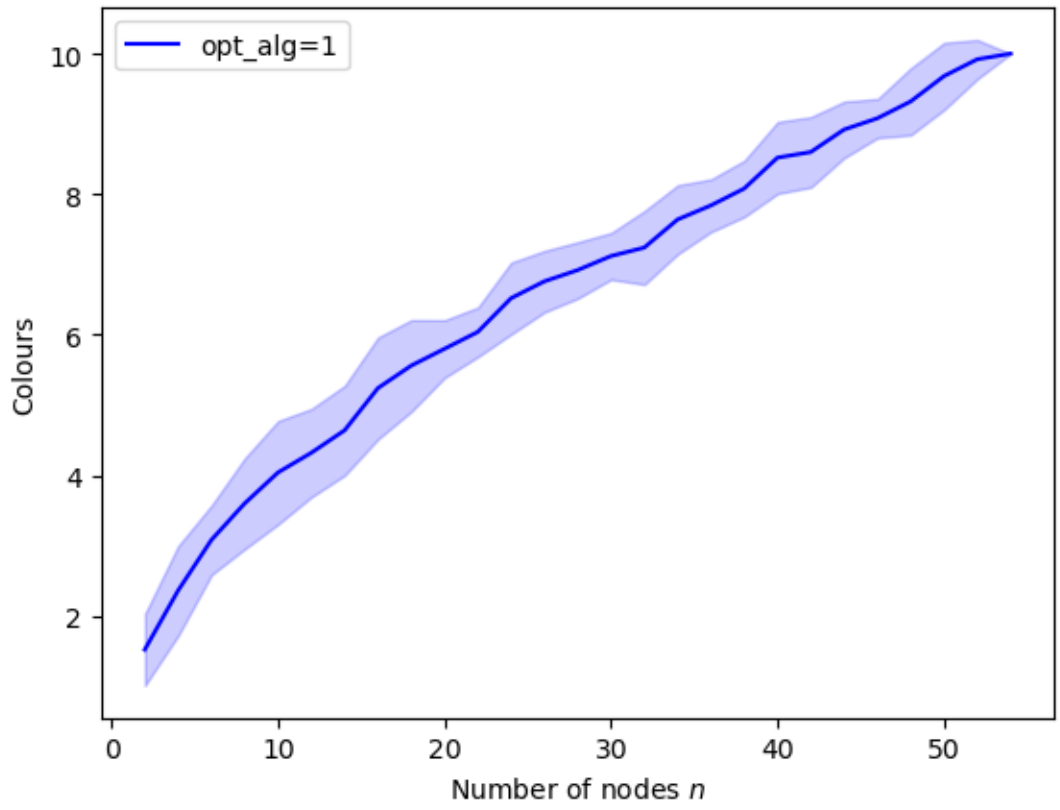
The following code evaluates the performance of this algorithm on $G(n, 0.5)$ graphs for a range of n-values.

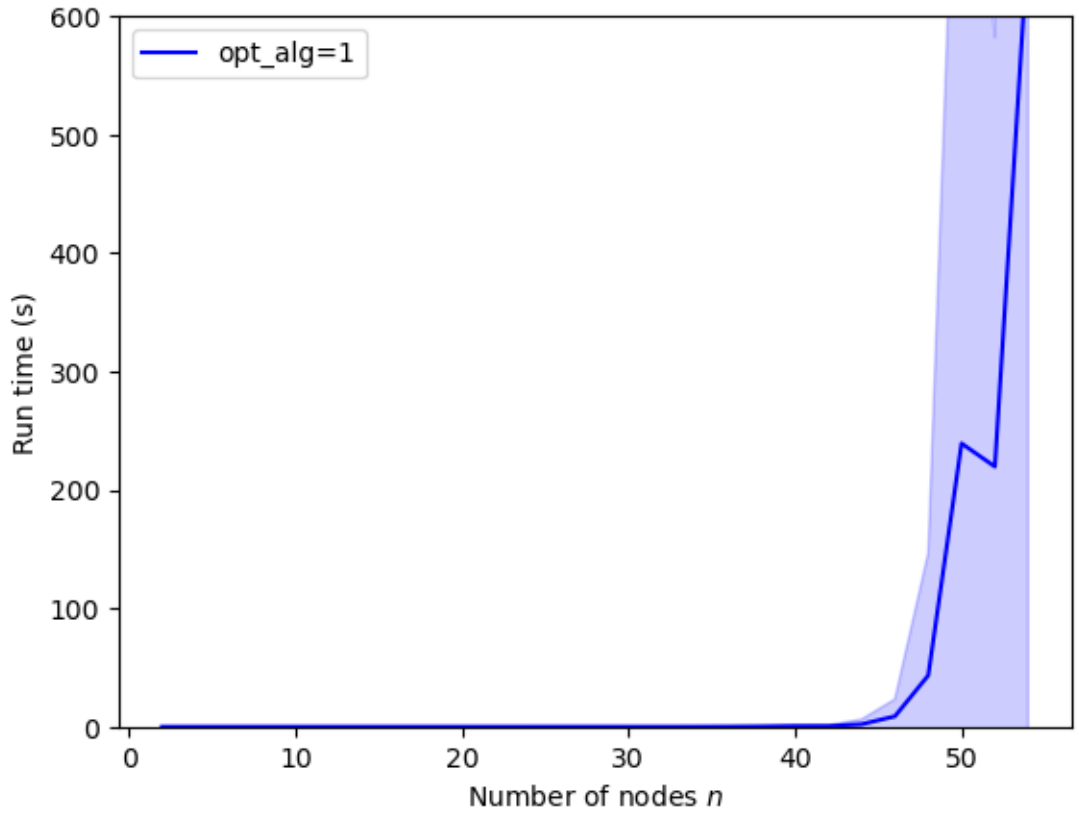
```
results = []
nVals = range(2,55,2)
for n in nVals:
    for seed in range(25):
        G = nx.gnp_random_graph(n, 0.5, seed)
        start = time.time()
        c = gcol.node_coloring(G, opt_alg=1)
        results.append([n, seed, "opt_alg=1", max(c.values()) + 1, time.time()-start])

# Create a pandas datafram from this list and make a pivot
```

(continues on next page)

(continued from previous page)

```
df = pd.DataFrame(results, columns=["n", "seed", "alg", "cols", "time"])
pivot = df.pivot_table(columns='alg', aggfunc=['mean','std'], values=['cols','time'],␣
↪index='n')

# Use the pivot table above to make the charts as before
mean1, SD1 = pivot[("mean","cols","opt_alg=1")], pivot[("std","cols","opt_alg=1")]
plt.plot(nVals, mean1, linestyle='-', linewidth=1.5, color="b", label='opt_alg=1')
plt.fill_between(nVals, mean1-SD1, mean1+SD1, color='b', alpha=0.2)
plt.xlabel("Number of nodes $n$")
plt.ylabel("Colours")
plt.legend()
plt.show()

mean1, SD1 = pivot[("mean","time","opt_alg=1")], pivot[("std","time","opt_alg=1")]
plt.plot(nVals, mean1, linestyle='-', linewidth=1.5, color="b", label='opt_alg=1')
plt.fill_between(nVals, mean1-SD1, mean1+SD1, color='b', alpha=0.2)
plt.xlabel("Number of nodes $n$")
plt.ylim((0, 600))
plt.ylabel("Run time (s)")
plt.legend()
plt.show()
```





The first chart above shows the chromatic numbers from a sample of $G(n, 0.5)$ graphs for an increasing number of nodes n. It can be seen that the chromatic number rises in a close-to-linear fashion in relation to n. The second figure also demonstrates the disadvantages of using an exponential-time algorithm: once n is increased beyond a moderately small value (approximately 50 here), run times become too high and/or unpredictable for practical use. Note, however, that the specific n-values that give in these infeasible run times can vary considerably depending on the topology of the graph. For example, planar graphs and scale-free graphs can often be solved very quickly for graphs with several hundred nodes. These sorts of results will usually need to be confirmed empirically.

4.4 Equitable Coloring

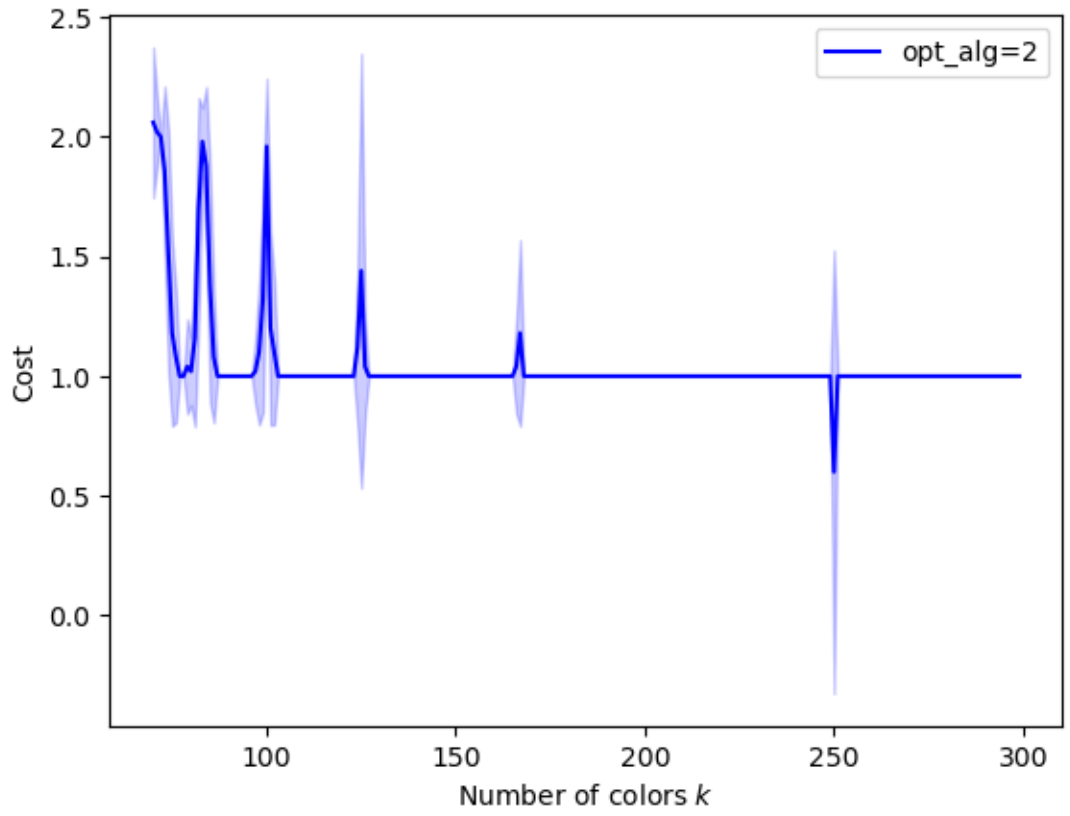
In the equitable node-coloring problem, we are interested in coloring the nodes with a user-defined number of colors k so that (a) adjacent nodes have different colors, and (b) the number of nodes in each color is as equal as possible. The following trials run the `gcol.equitable_node_k_coloring()` method on a sample of random $G(500, 0.5)$ graphs over a range of suitable k-values. The reported cost is simply the difference in size between the largest and smallest color classes in a solution. Hence, if k is a divisor of n, a cost of zero indicates an equitable k-coloring, else a cost of one indicates an equitable coloring.

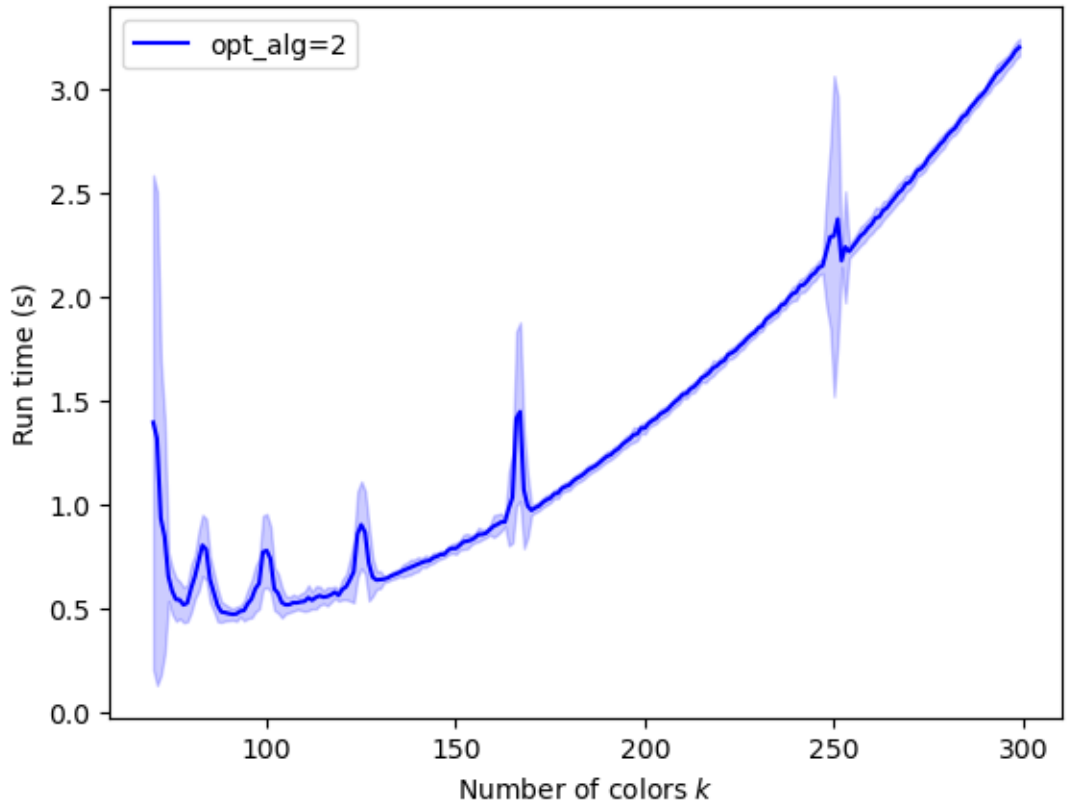
```
results = []
n = 500
kVals = range(70, 300, 1)
for seed in range(50):
    G = nx.gnp_random_graph(n, 0.5, seed)
    for k in kVals:
        start = time.time()
        c = gcol.equitable_node_k_coloring(G, k, opt_alg=2, it_limit=len(G))
        P = gcol.partition(c)
```

(continues on next page)

(continued from previous page)

```
        cost = max(len(j) for j in P) - min(len(j) for j in P)
        results.append([k, seed, "opt_alg=2", cost, time.time()-start])

# Create a pandas dataframe from this list and make a pivot table
df = pd.DataFrame(results, columns=["k", "seed", "alg", "cost", "time"])
pivot = df.pivot_table(columns='alg', aggfunc=['mean','std'], values=['cost','time'],␣
↪index='k')
display(pivot)

# Use the pivot table above to make charts as before
mean1, SD1 = pivot[("mean","cost","opt_alg=2")], pivot[("std","cost","opt_alg=2")]
plt.plot(kVals, mean1, linestyle='-', linewidth=1.5, color="b", label='opt_alg=2')
plt.fill_between(kVals, mean1-SD1, mean1+SD1, color='b', alpha=0.2)
plt.xlabel("Number of colors $k$")
plt.ylabel("Cost")
plt.legend()
plt.show()

mean1, SD1 = pivot[("mean","time","opt_alg=2")], pivot[("std","time","opt_alg=2")]
plt.plot(kVals, mean1, linestyle='-', linewidth=1.5, color="b", label='opt_alg=2')
plt.fill_between(kVals, mean1-SD1, mean1+SD1, color='b', alpha=0.2)
plt.xlabel("Number of colors $k$")
plt.ylabel("Run time (s)")
plt.legend()
plt.show()
```





The first chart above demonstrates that the `gcol.equitable_node_k_coloring()` method consistently achieves equitable node k-colorings. The exceptions occur for low values of k (which are close to the chromatic number) and when k is a divisor of n. In the former case, the low number of available colors restricts the choice of appropriate colors for each node, often leading to inequitable colorings. On the other hand, when k is a divisor of n, the algorithm is seeking a solution with a cost of zero, meaning that each color class must have *exactly* the same number of nodes. If this cannot be done, then a cost of at least two must be incurred.

The second chart above also indicates that runtimes of this routine increase slightly when k is a divisor of n. Run times also lengthen due to increases in k. The latter is due to the larger number of solutions that need to be evaluated in each iteration of the local search algorithm used with this routine. More details on this algorithm can be found in `gcol`'s documentation.

Finally, note that NetworkX also features an equitable node k-coloring routine, but this can only be used when $k \geq \Delta(G)$, where $\Delta(G)$ is the highest node degree in the graph. In the $G(500, 0.5)$ graphs considered here, the minimum valid value for k is approximately 280.

4.5 Independent Set Comparison

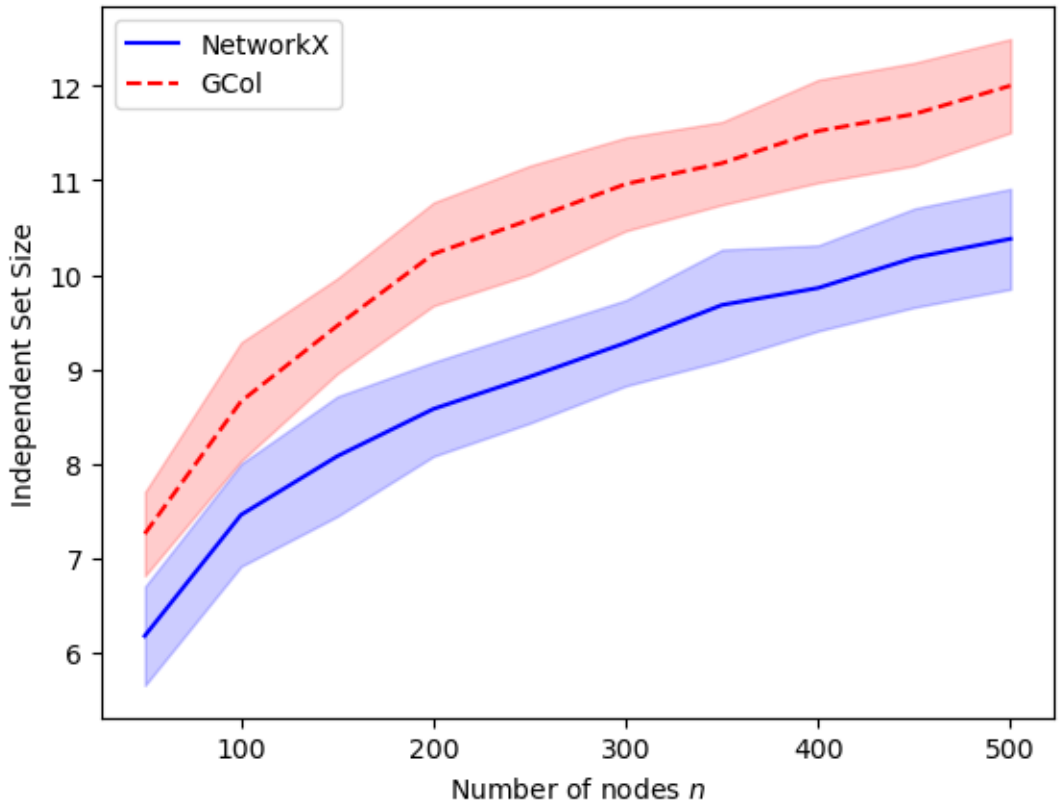
Our final set or trials looks at the performance the `gcol.max_independent_set()` routine and compares it to the approximation algorithm included in NetworkX for the same problem. As before, we use an iteration limit of n for the former.

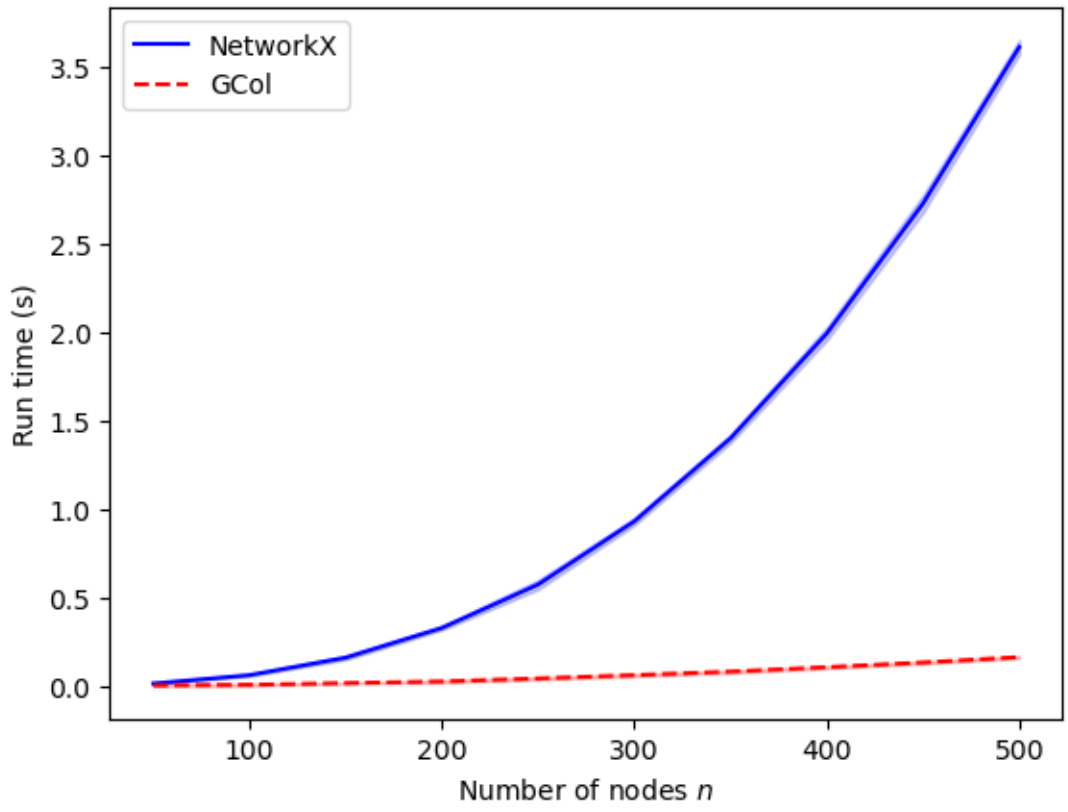
```
#Carry out the trials and put the results into a list
results = []
nVals = range(50,501,50)
for n in nVals:
    for seed in range(50):
```

(continues on next page)

(continued from previous page)

```
        G = nx.gnp_random_graph(n, 0.5, seed)
        start = time.time()
        S = gcol.max_independent_set(G, it_limit=len(G))
        results.append([n, seed, "gcol", len(S), time.time()-start])
        start = time.time()
        S = nx.approximation.maximum_independent_set(G)
        results.append([n, seed, "networkx", len(S), time.time()-start])

# Create a pandas dataframe from this list and make a pivot table
df = pd.DataFrame(results, columns=["n", "seed", "alg", "size", "time"])
pivot = df.pivot_table(columns='alg', aggfunc=['mean','std'], values=['size','time'],␣
↪index='n')
display(pivot)

# Create the charts as before
mean1, SD1 = pivot[("mean","size","networkx")], pivot[("std","size","networkx")]
mean2, SD2 = pivot[("mean","size","gcol")], pivot[("std","size","gcol")]
plt.plot(nVals, mean1, linestyle='-', linewidth=1.5, color="b", label='NetworkX')
plt.fill_between(nVals, mean1-SD1, mean1+SD1, color='b', alpha=0.2)
plt.plot(nVals, mean2, linestyle='--', linewidth=1.5, color="r", label='GCol')
plt.fill_between(nVals, mean2-SD2, mean2+SD2, color='r', alpha=0.2)
plt.xlabel("Number of nodes $n$")
plt.ylabel("Independent Set Size")
plt.legend()
plt.show()

mean1, SD1 = pivot[("mean","time","networkx")], pivot[("std","time","networkx")]
mean2, SD2 = pivot[("mean","time","gcol")], pivot[("std","time","gcol")]
plt.plot(nVals, mean1, linestyle='-', linewidth=1.5, color="b", label='NetworkX')
plt.fill_between(nVals, mean1-SD1, mean1+SD1, color='b', alpha=0.2)
plt.plot(nVals, mean2, linestyle='--', linewidth=1.5, color="r", label='GCol')
plt.fill_between(nVals, mean2-SD2, mean2+SD2, color='r', alpha=0.2)
plt.xlabel("Number of nodes $n$")
plt.ylabel("Run time (s)")
plt.legend()
plt.show()
```


NetworkX
GCol
Independent Set Size
12
11
10
9
8
7
6
100
200
300
400
500
Number of nodes n


NetworkX
GCol
Run time (s)
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
100
200
300
400
500
Number of nodes n

The results above show clearly that the `gcol.max_independent_set()` routine produces larger independent sets (and therefore better quality solutions) in less run time. As before, further improvements in solution quality (but longer run times) may also be found by increasing the `it_limit` parameter.

CHAPTER

FIVE

DOCUMENTATION

The following commands are available in the `gcol` graph coloring library. These are written in pure Python, which can be viewed by clicking the `source` links below.

`gcol.coloring.`**`chromatic_index`**`(`G`)`

Returns the chromatic index of the graph `G`.

The chromatic index of a graph G is the minimum number of colors needed to color the edges so that no two adjacent edges have the same color (a pair of edges is considered adjacent if and only if they share a common endpoint). The chromatic index is commonly denoted by $\chi'(G)$. Equivalently, $\chi'(G)$ is the minimum number of matchings needed to partition the edges of G. According to Vizing's theorem [1], $\chi'(G)$ is equal to either $\Delta(G)$ or $\Delta(G) + 1$, where $\Delta(G)$ is the maximum degree in G.

Determining the chromatic index of a graph is NP-hard. The approach used here is based on the backtracking algorithm of [2]. This is exact but operates in exponential time. It is therefore only suitable for graphs that are small, or that have topologies suited to its search strategies.

In this implementation, edge colorings of a graph G are determined by forming G's line graph $L(G)$ and then passing $L(G)$ to the *chromatic_number()* method.

Parameters

G
[NetworkX graph] The chromatic index for this graph will be calculated.

Returns

int
A nonnegative integer that gives the chromatic index of `G`.

Raises

NotImplementedError
If `G` is a directed graph or a multigraph.

See also

chromatic_number
node_coloring

Notes

The backtracking approach used here is an implementation of the exact algorithm described in [2]. It has exponential runtime and halts only when the chromatic index has been determined. Further details of this algorithm are given in the notes section of the *node_coloring()* method.

The above algorithm is described in detail in [2]. The c++ code used in [2] and [3] forms the basis of this library's Python implementations.

References

[1], [2], [3]

Examples

```
>>> import networkx as nx
>>> import gcol
>>>
>>> G = nx.dodecahedral_graph()
>>> chi = gcol.chromatic_index(G)
>>> print("Chromatic index is", chi)
Chromatic index is 3
```

gcol.coloring.**chromatic_number**(G)

Returns the chromatic number of the graph G.

The chromatic number of a graph G is the minimum number of colors needed to color the nodes so that no two adjacent nodes have the same color. It is commonly denoted by $\chi(G)$. Equivalently, $\chi(G)$ is the minimum number of independent sets needed to partition the nodes of G.

Determining the chromatic number is NP-hard. The approach used here is based on the backtracking algorithm of [1]. This is exact but operates in exponential time. It is therefore only suitable for graphs that are small, or that have topologies suited to its search strategies.

Parameters

G
[NetworkX graph] The chromatic number for this graph will be calculated.

Returns

int
A nonnegative integer that gives the chromatic number of G.

Raises

NotImplementedError
If G is a directed graph or a multigraph.

See also

chromatic_index
node_coloring

Notes

The backtracking approach used here is an implementation of the exact algorithm described in [1]. It has exponential runtime and halts only when the chromatic number has been determined. Further details of this algorithm are given in the notes section of the *node_coloring()* method.

The above algorithm is described in detail in [1]. The c++ code used in [1] and [2] forms the basis of this library's Python implementations.

References

[1], [2]

Examples

```
>>> import networkx as nx
>>> import gcol
>>>
>>> G = nx.dodecahedral_graph()
>>> chi = gcol.chromatic_number(G)
>>> print("Chromatic number is", chi)
Chromatic number is 3
```

gcol.coloring.**coloring_layout**(G, c)

Arranges the nodes of the graph in a circle so that nodes of the same color are next to each other. This method is designed to be used with the `pos` argument in the drawing functions of NetworkX (see example below).

Parameters

G
: [NetworkX graph] The graph we want to visualize.

c
: [dict] A dictionary with keys representing nodes and values representing their colors. Colors are identified by the integers $0, 1, 2, \ldots$. Nodes with negative values are ignored.

Returns

pos
: [dict] A dictionary of positions keyed by node.

See also

get_node_colors
multipartite_layout

Examples

```
>>> import networkx as nx
>>> import matplotlib.pyplot as plt
>>> import gcol
>>>
>>> G = nx.dodecahedral_graph()
>>> c = gcol.node_coloring(G)
>>> nx.draw_networkx(G, pos=gcol.coloring_layout(G, c), node_color=gcol.get_node_
```

(continues on next page)

(continued from previous page)

```
↪colors(G, c))
>>> plt.show()
```

gcol.coloring.**edge_coloring**(*G*, *strategy='dsatur'*, *opt_alg=None*, *it_limit=0*)

Returns a coloring of a graph's edges.

An edge coloring of a graph is an assignment of colors to edges so that adjacent edges have different colors (a pair of edges is considered adjacent if and only if they share a common endpoint). The aim is to use as few colors as possible. A set of edges assigned to the same color corresponds to a matching; hence the equivalent aim is to partition the graph's edges into a minimum number of matchings.

The smallest number of colors needed for coloring the edges of a graph G is known as the graph's chromatic index, denoted by $\chi'(G)$. Equivalently, $\chi'(G)$ is the minimum number of matchings needed to partition the nodes of a simple graph G. According to Vizing's theorem [1], $\chi'(G)$ is either $\Delta(G)$ or $\Delta(G)+1$, where $\Delta(G)$ is the maximum degree in G.

Determining an edge coloring that minimizes the number of colors is an NP-hard problem. This method therefore includes options for using an exponential-time exact algorithm (based on backtracking), or a choice of two polynomial-time heuristic algorithms (based on local search). The exact algorithm is generally only suitable for graphs that are small, or that have topologies suited to its search strategies. In all other cases, the local search algorithms are more appropriate.

In this implementation, edge colorings of a graph G are determined by forming G's line graph $L(G)$, and then passing $L(G)$ to the *node_coloring()* method. All parameters are therefore the same as the latter. (Note that, if a graph $G = (V, E)$ has n nodes and m edges, its line graph $L(G)$ will have m nodes and $\frac{1}{2}\sum_{v \in V} \deg(v)^2 - m$ edges.)

Parameters

G
[NetworkX graph] The edges of this graph will be colored.

strategy: string, optional (default='dsatur')
A string specifying the method used to generate the initial solution. It must be one of the following:

- `'random'` : Randomly orders $L(G)$'s nodes and then applies the greedy algorithm for graph node coloring [2].
- `'welsh-powell'` : Orders $L(G)$'s nodes by decreasing degree, then applies the greedy algorithm.
- `'dsatur'` : Uses the DSatur algorithm for graph node coloring on $L(G)$ [3].
- `'rlf'` : Uses the recursive largest first (RLF) algorithm for graph node coloring on $L(G)$ [4].

opt_alg
[None or int, optional (default=None)] An integer specifying the optimization method that will be used to try to reduce the number of colors. It must be one of the following

- `1` : An exact, exponential-time algorithm based on backtracking. The algorithm halts only when an optimal solution has been found.
- `2` : A local search algorithm that seeks to reduce the number of colors by temporarily allowing adjacent nodes in $L(G)$ to have the same color. Each iteration has a complexity $O(m + kn)$, where n is the number of nodes in $L(G)$, m is the number of edges in $L(G)$, and k is the number of colors in the current solution.

- `3` : A local search algorithm that seeks to reduce the number of colors by temporarily allowing nodes in $L(G)$ to be uncolored. Each iteration has a complexity $O(m + kn)$, as above.
- `None` : No optimization is performed.

it_limit
[int, optional (default=0)] Number of iterations of the local search procedure. Only applicable when using `opt_alg=2` or `opt_alg=3`.

Returns

dict
A dictionary with keys representing edges and values representing their colors. Colors are identified by the integers $0, 1, 2, \ldots$. The number of colors being used in a solution `c` is therefore `max(c.values()) + 1`.

Raises

NotImplementedError
If `G` is a directed graph or a multigraph.

ValueError
If `strategy` is not among the supported options. If `opt_alg` is not among the supported options. If `it_limit` is not a nonnegative integer.

See also

node_coloring
chromatic_index
edge_k_coloring

Notes

As mentioned, in this implementation, edge colorings of a graph G are determined by forming G's line graph $L(G)$ and then passing $L(G)$ to the *node_coloring()* method. All details are therefore the same as those in the latter, where they are documented more fully.

All the above algorithms and bounds are described in detail in [5]. The c++ code used in [5] and [6] forms the basis of this library's Python implementations.

References

[1], [2], [3], [4], [5], [6]

Examples

```
>>> import networkx as nx
>>> import gcol
>>>
>>> G = nx.dodecahedral_graph()
>>> c = gcol.edge_coloring(G)
>>> print("Coloring is", c)
Coloring is {(11, 12): 0, (11, 18): 1, (10, 11): 2, (12, 16): 1, (12, 13): 2, (18,␣
↪19): 0, (17, 18): 2, (16, 17): 0, (4, 17): 1, (15, 16): 2, (0, 10): 0, (9, 10): 1,
↪ (9, 13): 0, (8, 9): 2, (13, 14): 1, (14, 15): 0, (5, 15): 1, (7, 14): 2, (0,␣
```

(continues on next page)

(continued from previous page)

```
↪19): 1, (3, 19): 2, (0, 1): 2, (3, 4): 0, (2, 3): 1, (1, 2): 0, (2, 6): 2, (5,␣
↪6): 0, (4, 5): 2, (1, 8): 1, (6, 7): 1, (7, 8): 0}
>>>
>>> print("Number of colors =", max(c.values()) + 1)
Number of colors = 3
>>>
>>> c = gcol.edge_coloring(G, strategy="rlf", opt_alg=2, it_limit=1000)
>>> print("Coloring is", c)
Coloring is {(3, 4): 0, (17, 18): 0, (0, 19): 0, (10, 11): 0, (12, 16): 0, (5, 15):␣
↪0, (13, 14): 0, (8, 9): 0, (1, 2): 0, (6, 7): 0, (16, 17): 1, (4, 5): 1, (14,␣
↪15): 1, (2, 6): 1, (3, 19): 1, (11, 18): 1, (12, 13): 1, (9, 10): 1, (0, 1): 1,␣
↪(7, 8): 1, (18, 19): 2, (5, 6): 2, (4, 17): 2, (0, 10): 2, (9, 13): 2, (1, 8): 2,␣
↪(15, 16): 2, (11, 12): 2, (2, 3): 2, (7, 14): 2}
>>>
>>> print("Number of colors =", max(c.values()) + 1)
Number of colors = 3
```

gcol.coloring.**edge_k_coloring**(*G*, *k*, *opt_alg=None*, *it_limit=0*)

Attempts to color the edges of a graph `G` using `k` colors so that adjacent edges have different colors (a pair of edges is considered adjacent if and only if they share a common endpoint). A set of edges assigned to the same color corresponds to a matching; hence the equivalent aim is to partition the graph's edges into `k` matchings.

The smallest number of colors needed for coloring the edges of a graph G is known as the graph's chromatic index, denoted by $\chi'(G)$. Equivalently, $\chi'(G)$ is the minimum number of matchings needed to partition the nodes of a simple graph G. According to Vizing's theorem [1], $\chi'(G)$ is either $\Delta(G)$ or $\Delta(G)+1$, where $\Delta(G)$ is the maximum degree in G. The problem of determining an edge k-coloring is polynomially solvable for any $k > \Delta(G)$. Similarly, it is certain no edge k-coloring exists for $k < \Delta(G)$. For $k = \Delta(G)$, however, the problem is NP-hard.

This method therefore includes options for using an exact exponential-time algorithm (based on backtracking), or a choice of two polynomial-time heuristic algorithms (based on local search). The exact algorithm is generally only suitable for larger values of k, for graphs that are small, or graphs that have topologies suited to its search strategies. In all other cases, the local search algorithms are more appropriate.

This method follows the steps used by the *node_k_coloring()* method. That is, edge k-colorings of a graph G are determined by forming G's line graph $L(G)$ and then passing $L(G)$ to the *node_k_coloring()* method. All parameters are therefore the same as the latter. (Note that, if a graph $G = (V, E)$ has n nodes and m edges, its line graph $L(G)$ will have m nodes and $\frac{1}{2}\sum_{v \in V} \deg(v)^2 - m$ edges.)

If an edge k-coloring cannot be determined by the algorithm, a `ValueError` exception is raised. Otherwise, an edge k-coloring is returned.

Parameters

G
: [NetworkX graph] The edges of this graph will be colored.

k
: [int] The number of colors to use.

opt_alg
: [None or int, optional (default=None)] An integer specifying the optimization method that will be used to try to reduce the number of colors (if this is seen to be greater than k). It must be one of the following

- 1 : An exact, exponential-time algorithm based on backtracking. The algorithm halts only when the existence of an edge k-coloring has been proved or disproved.

- `2` : A local search algorithm that seeks to reduce the number of colors by temporarily allowing adjacent nodes in $L(G)$ to have the same color. Each iteration has a complexity $O(m + kn)$, where n is the number of nodes in $L(G)$, m is the number of edges in $L(G)$, and k is the number of colors in the current solution.
- `3` : A local search algorithm that seeks to reduce the number of colors by temporarily allowing nodes in $L(G)$ to be uncolored. Each iteration has a complexity $O(m + kn)$, as above.
- `None` : No optimization is performed.

it_limit
[int, optional (default=0)] Number of iterations of the local search procedure. Only applicable when using `opt_alg=2` or `opt_alg=3`.

Returns

dict
A dictionary with keys representing edges and values representing their colors. Colors are identified by the integers $0, 1, 2, \ldots, k-1$.

Raises

NotImplementedError
If `G` is a directed graph or a multigraph.

ValueError
If `opt_alg` is not among the supported options. If `it_limit` is not a nonnegative integer. If `k` is not a nonnegative integer. If a clique larger than `k` is observed in the line graph of G. If `k` is less than the maximum degree in `G`. If an edge k-coloring could not be determined.

See also

edge_coloring
equitable_edge_k_coloring
node_k_coloring

Notes

As mentioned, in this implementation, edge colorings of a graph G are determined by forming G's line graph $L(G)$ and then passing $L(G)$ to the *node_k_coloring()* method. All details are therefore the same as those in the latter. The routine halts immediately once an edge k-coloring has been achieved.

All the above algorithms and bounds are described in detail in [2]. The c++ code used in [2] and [3] forms the basis of this library's Python implementations.

References

[1], [2], [3]

Examples

```
>>> import networkx as nx
>>> import gcol
>>>
>>> G = nx.dodecahedral_graph()
```

(continues on next page)

(continued from previous page)

```
>>> c = gcol.edge_k_coloring(G, 4)
>>> print(c)
{(11, 12): 0, (11, 18): 1, (10, 11): 2, (12, 16): 3, (12, 13): 1, (18, 19): 0, (17,␣
↪18): 2, (16, 17): 0, (4, 17): 3, (15, 16): 1, (0, 10): 3, (9, 10): 0, (0, 19): 2,␣
↪(9, 13): 2, (3, 19): 1, (0, 1): 0, (8, 9): 3, (13, 14): 3, (3, 4): 2, (1, 8): 1,␣
↪(14, 15): 0, (4, 5): 1, (2, 3): 3, (1, 2): 2, (7, 8): 0, (5, 15): 2, (7, 14): 1,␣
↪(2, 6): 0, (5, 6): 3, (6, 7): 2}
>>>
>>> c = gcol.edge_k_coloring(G, 3)
>>> print(c)
{(11, 12): 0, (11, 18): 1, (10, 11): 2, (12, 16): 1, (12, 13): 2, (18, 19): 0, (17,␣
↪18): 2, (16, 17): 0, (4, 17): 1, (15, 16): 2, (0, 10): 0, (9, 10): 1, (9, 13): 0,␣
↪(8, 9): 2, (13, 14): 1, (14, 15): 0, (5, 15): 1, (7, 14): 2, (0, 19): 1, (3, 19):␣
↪2, (0, 1): 2, (3, 4): 0, (2, 3): 1, (1, 2): 0, (2, 6): 2, (5, 6): 0, (4, 5): 2,␣
↪(1, 8): 1, (6, 7): 1, (7, 8): 0}
```

gcol.coloring.**edge_precoloring**(*G*, *precol=None*, *strategy='dsatur'*, *opt_alg=None*, *it_limit=0*)

Returns a coloring of a graph's edges in which some of the edges have been precolored.

An edge coloring of a graph is an assignment of colors to edges so that adjacent edges have different colors (a pair of edges is considered adjacent if and only if they share a common endpoint). The aim is to use as few colors as possible. A set of edges assigned to the same color corresponds to a matching; hence the equivalent aim is to partition the graph's edges into a minimum number of matchings.

In the edge precoloring problem, some of the edges have already been assigned colors. The aim is to allocate colors to the remaining edges so that we get a full edge coloring that uses a minimum number of colors.

The edge precoloring problem is NP-hard. This method therefore includes options for using an exponential-time exact algorithm (based on backtracking), or a choice of two polynomial-time heuristic algorithms (based on local search). The exact algorithm is generally only suitable for graphs that are small, or that have topologies suited to its search strategies. In all other cases, the local search algorithms are more appropriate.

In this implementation, edge colorings of a graph G are determined by forming G's line graph $L(G)$ and then passing $L(G)$ to the *node_precoloring()* method. All parameters are therefore the same as the latter. (Note that, if a graph $G = (V, E)$ has n nodes and m edges, its line graph $L(G)$ will have m nodes and $\frac{1}{2}\sum_{v \in V} \deg(v)^2 - m$ edges.)

Parameters

G
: [NetworkX graph] The edges of this graph will be colored.

precol
: [None or dict, optional (default=None)] A dictionary that specifies the colors of any precolored edges.

strategy
: [string, optional (default='dsatur')] A string specifying the method used to generate the initial solution. It must be one of the following:

- `'random'` : Randomly orders $L(G)$'s nodes and then applies the greedy algorithm for graph node coloring [1].
- `'welsh-powell'` : Orders $L(G)$'s nodes by decreasing degree, then applies the greedy algorithm.
- `'dsatur'` : Uses the DSatur algorithm for graph node coloring on $L(G)$ [2].

- `'rlf'` : Uses the recursive largest first (RLF) algorithm for graph node coloring on $L(G)$ [3].

opt_alg
[None or int, optional (default=None)] An integer specifying the optimization method that will be used to try to reduce the number of colors. It must be one of the following

- `1` : An exact, exponential-time algorithm based on backtracking. The algorithm halts only when an optimal solution has been found.
- `2` : A local search algorithm that seeks to reduce the number of colors by temporarily allowing adjacent nodes in $L(G)$ to have the same color. Each iteration has a complexity $O(m + kn)$, where n is the number of nodes in $L(G)$, m is the number of edges, and k is the number of colors in the current solution.
- `3` : A local search algorithm that seeks to reduce the number of colors by temporarily allowing nodes in $L(G)$ to be uncolored. Each iteration has a complexity $O(m + kn)$, as above.
- `None` : No optimization is performed.

it_limit
[int, optional (default=0)] Number of iterations of the local search procedure. Only applicable when using `opt_alg=2` or `opt_alg=3`.

Returns

dict
A dictionary with keys representing edges and values representing their colors. Colors are identified by the integers $0, 1, 2, \ldots$. The number of colors being used in a solution `c` is therefore `max(c.values()) + 1`. If `precol[(u,v)]==j` then `c[(u,v)]==j`.

Raises

NotImplementedError
If `G` is a directed graph or a multigraph.

ValueError
If `strategy` is not among the supported options. If `opt_alg` is not among the supported options. If `it_limit` is not a nonnegative integer. If `precol` contains an edge that is not in `G`. If `precol` contains a non-integer color label. If `precol` contains a pair of adjacent edges assigned to the same color. If `precol` uses an integer color label j, but there exists a color label $0 \leq i < j$ that is not being used.

See also

edge_coloring
node_precoloring
node_coloring

Notes

As mentioned, in this implementation, edge colorings of a graph G are determined by forming G's line graph $L(G)$ and then passing $L(G)$ to the *node_precoloring()* method. All details are therefore the same as those in the latter, where they are documented.

All the above algorithms and bounds are described in detail in [4]. The c++ code used in [4] and [5] forms the basis of this library's Python implementations.

References

[1], [2], [3], [4], [5]

Examples

```
>>> import networkx as nx
>>> import gcol
>>>
>>> G = nx.dodecahedral_graph()
>>> p = {(0, 1):0, (8, 9): 1, (10, 11): 2, (11, 12): 3}
>>> c = gcol.edge_precoloring(G, precol=p)
>>> print("Coloring is",c)
Coloring is {(0, 1): 0, (8, 9): 1, (10, 11): 2, (11, 12): 3, (1, 8): 2, (0, 10): 1,␣
↪(9, 10): 0, (11, 18): 0, (7, 8): 0, (1, 2): 1, (0, 19): 2, (9, 13): 2, (18, 19):␣
↪1, (12, 13): 0, (17, 18): 2, (3, 19): 0, (12, 16): 1, (13, 14): 1, (2, 3): 2, (16,
↪ 17): 0, (7, 14): 2, (3, 4): 1, (4, 17): 3, (2, 6): 0, (15, 16): 2, (14, 15): 0,␣
↪(6, 7): 1, (4, 5): 0, (5, 15): 1, (5, 6): 2}
>>>
>>> print("Number of colors =", max(c.values()) + 1)
Number of colors = 4
```

gcol.coloring.**equitable_edge_k_coloring**(*G*, *k*, *weight=None*, *opt_alg=None*, *it_limit=0*)

Attempts to color the edges of a graph using `k` colors so that (a) adjacent edges have different colors, and (b) the weight of each color class is equal. (A pair of edges is considered adjacent if and only if they share a common endpoint.) If `weight=None`, the weight of a color class is the number of edges assigned to that color; otherwise, it is the sum of the weights of the edges assigned to that color.

Equivalently, this routine seeks to partition the graph's edges into k matchings so that the weight of each matching is equal.

This method first follows the steps used by the *edge_k_coloring()* method to try and find an edge k-coloring. That is, edge colorings of a graph G are determined by forming G's line graph $L(G)$ and then passing $L(G)$ to the *node_k_coloring()* method. All parameters are therefore the same as the latter. (Note that, if a graph $G = (V, E)$ has n nodes and m edges, its line graph $L(G)$ will have m nodes and $\frac{1}{2}\sum_{v \in V} \deg(v)^2 - m$ edges.)

If an edge k-coloring cannot be determined by the algorithm, a `ValueError` exception is raised. Otherwise, once an edge k-coloring has been formed, the algorithm uses a bespoke local search operator to reduce the variance in weights across the k colors. In solutions returned by this method, adjacent edges always receive different colors; however, the coloring is not guaranteed to be equitable, even if an equitable edge k-coloring exists.

Parameters

G
: [NetworkX graph] The edges of this graph will be colored.

k
: [int] The number of colors to use.

weight
: [None or string, optional (default=None)] If `None`, every edge is assumed to have a weight of 1. If a string, this should correspond to a defined edge attribute. Edge weights must be positive.

opt_alg
: [None or int, optional (default=None)] An integer specifying the optimization method that will be used to try to reduce the number of colors (if this is seen to be greater than `k`). It must be one of the following

- `1` : An exact, exponential-time algorithm based on backtracking. The algorithm halts only when the existence of an edge k-coloring has been proved or disproved.
- `2` : A local search algorithm that seeks to reduce the number of colors by temporarily allowing adjacent nodes in $L(G)$ to have the same color. Each iteration has a complexity $O(m + kn)$, where n is the number of nodes in $L(G)$, m is the number of edges in $L(G)$, and k is the number of colors in the current solution.
- `3` : A local search algorithm that seeks to reduce the number of colors by temporarily allowing nodes in $L(G)$ to be uncolored. Each iteration has a complexity $O(m + kn)$, as above.
- None : No optimization is performed.

it_limit
[int, optional (default=0)] Number of iterations of the local search procedure. Only applicable when using `opt_alg=2` or `opt_alg=3`.

Returns

dict
A dictionary with keys representing edges and values representing their colors. Colors are identified by the integers $0, 1, 2, \ldots, k - 1$.

Raises

NotImplementedError
If `G` is a directed graph or a multigraph.

ValueError
If `opt_alg` is not among the supported options. If `it_limit` is not a nonnegative integer. If `k` is not a nonnegative integer. If a clique larger than `k` is observed in the line graph of G. If `k` is less than the maximum degree in `G`. If an edge k-coloring could not be determined. If an edge with a non-positive weight is specified.

KeyError
If an edge does not have the attribute defined by `weight`

See also

edge_k_coloring
node_k_coloring
equitable_node_k_coloring
kempe_chain

Notes

As mentioned, in this implementation edge colorings of a graph G are determined by forming G's line graph $L(G)$ and then following the same steps as the *node_k_coloring()* method to try and find a node k-coloring of $L(G)$; however, it also takes edge weights into account if needed. If an edge k-coloring is achieved, a bespoke local search operator (based on steepest descent) is then used to try to reduce the variance in weights across the k color classes. This follows the same steps as the *equitable_node_k_coloring()* method, using $L(G)$. Further details on this optimization method can be found in Chapter 7 of [2], or in [3].

All the above algorithms are described in detail in [2]. The c++ code used in [2] and [4] forms the basis of this library's Python implementations.

References

[1], [2], [3], [4]

Examples

```
>>> import networkx as nx
>>> import gcol
>>>
>>> G = nx.dodecahedral_graph()
>>> c = gcol.equitable_edge_k_coloring(G, 4)
>>> P = gcol.partition(c)
>>> print(P)
[[(11, 12), (18, 19), (16, 17), (9, 10), (0, 1), (14, 15), (7, 8), (2, 6)], [(11,␣
↪18), (12, 13), (15, 16), (3, 19), (1, 8), (4, 5), (7, 14)], [(10, 11), (17, 18),␣
↪(0, 19), (9, 13), (3, 4), (1, 2), (5, 15), (6, 7)], [(12, 16), (4, 17), (0, 10),␣
↪(8, 9), (13, 14), (2, 3), (5, 6)]]
>>> print("Size of smallest color class =", min(len(j) for j in P))
Size of smallest color class = 7
>>> print("Size of biggest color class =", max(len(j) for j in P))
Size of biggest color class = 8
>>>
>>> #Now add some (arbitrary) weights to the edges
>>> for e in G.edges():
>>>     G.add_edge(e[0], e[1], weight = abs(e[0]-e[1]))
>>> c = gcol.equitable_edge_k_coloring(G, 5, weight="weight")
>>> P = gcol.partition(c)
>>> print(P)
[[(11, 12), (18, 19), (4, 17), (13, 14), (1, 8), (2, 3)], [(11, 18), (9, 13), (0,␣
↪1), (3, 4), (5, 15), (7, 8)], [(10, 11), (17, 18), (15, 16), (0, 19), (1, 2), (6,␣
↪7)], [(12, 16), (9, 10), (3, 19), (14, 15), (5, 6)], [(12, 13), (16, 17), (0, 10),
↪ (8, 9), (4, 5), (7, 14), (2, 6)]]
>>> print("Weight of lightest color class =", min(sum(G[u][v]["weight"] for u, v in␣
↪j) for j in P))
Weight of lightest color class = 23
>>> print("Weight of heaviest color class =", max(sum(G[u][v]["weight"] for u, v in␣
↪j) for j in P))
Weight of heaviest color class = 25
```

gcol.coloring.**equitable_node_k_coloring**(*G*, *k*, *weight=None*, *opt_alg=None*, *it_limit=0*)

Attempts to color the nodes of a graph using `k` colors so that (a) all adjacent nodes have different colors, and (b) the weight of each color class is equal. If `weight=None`, the weight of a color class is the number of nodes assigned to that color; otherwise, it is the sum of the weights of the nodes assigned to that color.

Equivalently, this routine seeks to partition the graph's nodes into `k` independent sets so that the weight of each independent set is equal.

Determining an equitable node k-coloring is NP-hard. This method first follows the steps used by the *node_k_coloring()* method to try and find a node k-coloring. If this is achieved, the algorithm then uses a bespoke local search operator to reduce the variance in weights across the k colors.

If a node k-coloring cannot be determined by the algorithm, a `ValueError` exception is raised. Otherwise, a node k-coloring is returned in which the variance in weights across the k color classes has been minimized. In solutions returned by this method, neighboring nodes always receive different colors; however, the coloring is not guaranteed to be equitable, even if an equitable node k-coloring exists.

Parameters

G
[NetworkX graph] The nodes of this graph will be colored.

k
[int] The number of colors to use.

weight
[None or string, optional (default=None)] If `None`, every node is assumed to have a weight of `1`. If string, this should correspond to a defined node attribute. Node weights must be positive.

opt_alg
[int, optional (default=None)] An integer specifying the optimization method that will be used to try to reduce the number of colors (if this is seen to be greater than k). It must be one of the following

- `1` : An exact, exponential-time algorithm based on backtracking. The algorithm halts only when the existence of a node k-coloring has been proved or disproved.
- `2` : A local search algorithm that seeks to reduce the number of colors by temporarily allowing adjacent nodes to have the same color. Each iteration has a complexity $O(m + kn)$, where n is the number of nodes in the modified graph, m is the number of edges, and k is the number of colors in the current solution.
- `3` : A local search algorithm that seeks to reduce the number of colors by temporarily allowing nodes to be uncolored. Each iteration has a complexity $O(m + kn)$, as above.
- `None` : No optimization is performed.

it_limit
[int, optional (default=0)] Number of iterations of the local search procedure. Only applicable when using `opt_alg=2` or `opt_alg=3`.

Returns

dict
A dictionary with keys representing nodes and values representing their colors. Colors are identified by the integers $0, 1, 2, \ldots, k-1$.

Raises

NotImplementedError
If `G` is a directed graph or a multigraph.

ValueError
If `opt_alg` is not among the supported options. If `it_limit` is not a nonnegative integer. If `k` is not a nonnegative integer. If a clique larger than `k` is observed in the graph. If a node k-coloring could not be determined. If a node with a non-positive weight is specified.

KeyError
If a node does not have the attribute defined by `weight`

See also

node_k_coloring
equitable_edge_k_coloring
kempe_chain

Notes

This method first follows the same steps as the *node_k_coloring()* method to try and find a node k-coloring; however, it also takes node weights into account if needed. If a node k-coloring is achieved, a bespoke local search operator (based on steepest descent) is then used to try to reduce the variance in weights across the k color classes. This process involves evaluating each Kempe-chain interchange in the current solution [1] and performing the interchange that results in the largest reduction in variance. This process repeats until there are no interchanges that reduce the variance. Each iteration of this local search process takes $O(n^2)$ time. Further details on this optimization method can be found in Chapter 7 of [2], or in [3].

All the above algorithms are described in detail in [2]. The c++ code used in [2] and [4] forms the basis of this library's Python implementations.

References

[1], [2], [3], [4]

Examples

```
>>> import networkx as nx
>>> import gcol
>>>
>>> G = nx.dodecahedral_graph()
>>> c = gcol.equitable_node_k_coloring(G, 4)
>>> P = gcol.partition(c)
>>> print(P)
[[0, 2, 9, 5, 14], [1, 3, 11, 7, 17], [19, 8, 6, 13, 16], [10, 18, 4, 12, 15]]
>>> print("Size of smallest color class =", min(len(j) for j in P))
Size of smallest color class = 5
>>> print("Size of biggest color class =", max(len(j) for j in P))
Size of biggest color class = 5
>>>
>>> #Now do similar with a node-weighted graph
>>> G = nx.Graph()
>>> G.add_node(0, weight=20)
>>> G.add_node(1, weight=9)
>>> G.add_node(2, weight=25)
>>> G.add_node(3, weight=10)
>>> G.add_edges_from([(0,2), (1,2), (3, 2)])
>>> c = gcol.equitable_node_k_coloring(G, 3, weight="weight")
>>> P = gcol.partition(c)
>>> print(P)
[[2], [0], [1, 3]]
>>>
>>> print("Weight of lightest color class =", min(sum(G.nodes[v]['weight'] for v in␣
↪j) for j in P))
Weight of lightest color class = 19
>>>
>>> print("Weight of heaviest color class =", max(sum(G.nodes[v]['weight'] for v in␣
↪j) for j in P))
Weight of heaviest color class = 25
```

gcol.coloring.**get_edge_colors**(G, c)

Generates an RGB color for each edge in the graph G based on its color label in c. This method is designed to

be used with the `edge_color` argument in the drawing functions of NetworkX (see example below). If an edge is marked as uncolored (i.e., assigned a negative value, or not present in `c`), it is painted light grey.

Parameters

G
[NetworkX graph] The graph we want to visualize.

c
[dict] A dictionary with keys representing edges and values representing their colors. Colors are identified by the integers $0, 1, 2, \ldots$.

Returns

list
A list specifying the RGB color that each edge should be painted with.

Raises

ValueError
If `c` uses more than 56 colors.

See also

get_set_colors
get_node_colors

Notes

Color 0 is set to red; color 1, green; and color, 2 blue. Beyond this, a sequence of RGB values are generated for each integer, aiming to keep the presented colors as distict as possible.

Examples

```
>>> import networkx as nx
>>> import matplotlib.pyplot as plt
>>> import gcol
>>>
>>> G = nx.dodecahedral_graph()
>>> c = gcol.edge_coloring(G)
>>> nx.draw_networkx(G, pos=nx.spring_layout(G), edge_color=gcol.get_edge_colors(G,␣
↪c))
>>> plt.show()
```

gcol.coloring.**get_node_colors**(G, c)

Generates an RGB color for each node in the graph `G` based on its color label in `c`. This method is designed to be used with the `node_color` argument in the drawing functions of NetworkX (see example below). If a node is marked as uncolored (i.e., assigned a negative value, or not present in `c`), it is painted white.

Parameters

G
[NetworkX graph] The graph we want to visualize.

c
[dict] A dictionary with keys representing nodes and values representing their colors. Colors are identified by the integers $0, 1, 2, \ldots$.

Returns

list
A list specifying the RGB color that each node should be painted with.

Raises

ValueError
If `c` uses more than 56 colors.

See also

get_set_colors
get_edge_colors

Notes

Color 0 is set to red; color 1, green; and color, 2 blue. Beyond this, a sequence of RGB values are generated for each integer, aiming to keep the presented colors as distict as possible.

Examples

```
>>> import networkx as nx
>>> import matplotlib.pyplot as plt
>>> import gcol
>>>
>>> G = nx.dodecahedral_graph()
>>> c = gcol.node_coloring(G)
>>> nx.draw_networkx(G, pos=nx.spring_layout(G), node_color=gcol.get_node_colors(G,␣
↪c))
>>> plt.show()
```

`gcol.coloring.`**`get_set_colors`**(*G*, *S*, *S_color='yellow'*, *other_color='grey'*)

Generates an RGB color for each node in the graph based on whether it is a member of the set `S`. By default, nodes in `S` are painted yellow and all others are painted grey. This method is designed to be used with the `node_color` argument in the drawing functions of NetworkX (see example below).

Parameters

G
[NetworkX graph] The graph we want to visualize.

S
[list or set] A subset of `G`'s nodes.

S_color
[color, optional (default='yellow')] Desired color of the nodes in `S`. Other options include `'blue'`, `'cyan'`, `'green'`, `'black'`, `'magenta'`, `'red'`, `'white'`, and `'yellow'`.

other_color
[color, optional (default='grey')] Desired color of the nodes not in `S`.

Returns

list
A list specifying the RGB color that each node should be painted with.

See also

get_node_colors
get_edge_colors

Examples

```
>>> import networkx as nx
>>> import matplotlib.pyplot as plt
>>> import gcol
>>>
>>> G = nx.dodecahedral_graph()
>>> S = gcol.max_independent_set(G, it_limit=1000)
>>> nx.draw_networkx(G, pos=nx.spring_layout(G), node_color=gcol.get_set_colors(G,␣
↪S))
>>> plt.show()
```

gcol.coloring.**kempe_chain**(G, c, s, j)

Given a node coloring `c` of a graph `G`, this method returns the set of nodes in the Kempe chain generated from a source node `s` using the color `j`.

A Kempe chain is a connected set of nodes in a graph that alternates between two colors [1]. Equivalently, it is a maximal connected subgraph that contains nodes of at most two colors. Interchanging the colors of the nodes in a Kempe chain creates a new coloring that uses the same number of colors or one fewer color. Two k-colorings of a graph are considered *Kempe equivalent* if one can be obtained from the other through a series of Kempe chain interchanges [2]. It is also known that, if k is larger than the degeneracy of a graph, then all k-colorings of this graph are Kempe equivalent [2].

Parameters

G
[NetworkX graph] The graph that we want to compute a Kempe chain for.

c
[dict] A node coloring of `G`. Pairs of adjacent nodes cannot be allocated to the same color. Any uncolored nodes `u` should have `c[u]==-1`.

s
[node] The source node that we will generate the Kempe chain from.

j
[int] The second color to use. The first color is that of `s`.

Returns

set
The set of nodes, reachable from `s`, that alternate between colors `i` and `j` (where `c[s]==i`).

Raises

NotImplementedError
If `G` is a directed graph or a multigraph.

ValueError
If `c` contains a pair of adjacent nodes assigned to the same color. If `c` contains an invalid color label or no color label for `s`. If `G` contains a node that is not in `c`.

See also

equitable_node_k_coloring

Notes

This method uses an extension of breadth-first search and operates in $O(n + m)$ time. If `c[s]==j`, then the Kempe chain contains node `s` only.

References

[1], [2]

Examples

```
>>> import networkx as nx
>>> import gcol
>>>
>>> G = nx.dodecahedral_graph()
>>> c = gcol.node_coloring(G)
>>> S = gcol.kempe_chain(G, c, 0, 1)
>>> print("Example Kempe chain =", S)
Example Kempe chain = {0, 1, 2, 4, 6, 8, 10, 17, 18, 19}
```

gcol.coloring.**max_independent_set**(*G*, *weight=None*, *it_limit=0*)

Attempts to identify the largest independent set of nodes in a graph. Nodes can also be allocated weights if desired.

The maximum independent set in a graph G is the largest subset of nodes in which none are adjacent. The size of the largest independent in a graph G is known as the independence number of G and is often denoted by $\alpha(G)$. Similarly, the maximum-weighted independent set in G is the subset of mutually nonadjacent nodes whose weight-total is maximized.

The problem of determining a maximum(-weighted) independent set of nodes is NP-hard. Consequently, this method makes use of a polynomial-time heuristic based on local search. It will always return an independent set but offers no guarantees on whether this is the optimal solution. The algorithm halts once the iteration limit has been reached.

Note that the similar problem of determining the maximum(-weighted) independent set of edges is equivalent to finding a maximum(-weighted) matching in a graph. This is a polynomially solvable problem and can be solved by the Blossom algorithm.

Parameters

G
: [NetworkX graph] An independent set of nodes in this graph will be returned.

weight
: [None or string, optional (default=None)] If `None`, every node is assumed to have a weight of 1. If a string, this should correspond to a defined node attribute. All node weights must be positive.

it_limit
: [int, optional (default=0)] Number of iterations of the local search procedure. Each iteration has a complexity $O(m + n)$, where n is the number of nodes and m is the number of edges.

Returns

list
A list containing the nodes belonging to the independent set.

Raises

NotImplementedError
If `G` is a directed graph or a multigraph.

ValueError
If `it_limit` is not a nonnegative integer. If a node with a non-positive weight is specified.

KeyError
If a node does not have the attribute defined by `weight`.

See also

node_k_coloring
node_coloring

Notes

This method uses the PartialCol algorithm for node k-coloring using $k = 1$. The set of nodes assigned to this color corresponds to the independent set. PartialCol is based on tabu search. Here, each iteration of PartialCol has complexity $O(n + m)$. It also occupies $O(n + m)$ of memory space.

The above algorithm is described in detail in [1]. The c++ code used in [1] and [2] forms the basis of this library's Python implementations.

References

[1], [2]

Examples

```
>>> import networkx as nx
>>> import gcol
>>>
>>> G = nx.dodecahedral_graph()
>>> S = gcol.max_independent_set(G, it_limit=1000)
>>> print("Independent set =", S)
Independent set = [19, 10, 2, 8, 5, 12, 14, 17]
>>>
>>> # Do similar with a node-weighted graph
>>> G = nx.Graph()
>>> G.add_node(0, weight=20)
>>> G.add_node(1, weight=9)
>>> G.add_node(2, weight=25)
>>> G.add_node(3, weight=10)
>>> G.add_edges_from([(0,2), (1,2), (3, 2)])
>>> S = gcol.max_independent_set(G, weight="weight", it_limit=1000)
>>> print("Independent set =", S)
Independent set = [0, 1, 3]
```

gcol.coloring.**min_cost_k_coloring**(*G*, *k*, *weight=None*, *weights_at='nodes'*, *it_limit=0*)

Colors the nodes of the graph using `k` colors so that a cost function is minimized. Equivalently, this routine partitions a graph's nodes while attempting to minimize a specific cost function.

This routine will always produce a k-coloring. However, this solution may include some clashes (that is, instances of adjacent nodes having the same color), or uncolored nodes. The aim is to minimize the number (or total weight) of these occurrences.

Determining a minimum cost solution to these problems is NP-hard. This routine employs polynomial-time heuristic algorithms based on local search.

Parameters

G
[NetworkX graph] The nodes of this graph will be colored.

k
[int] The number of colors to use.

weight
[None or string, optional (default=None)] If `None`, every node and edge is assumed to have a weight of `1`. If string, this should correspond to a defined node or edge attribute. All node and edge weights must be positive.

weights_at
[string, optional (default='nodes')] A string that must be one of the following:

- `'nodes'` : Here, nodes can be left uncolored in a solution. If `weight=None`, the method seeks a k-coloring in which the number of uncolored nodes is minimized; otherwise, the method seeks a k-coloring that minimizes the sum of the weights of the uncolored nodes. Clashes are not permitted in a solution. The algorithm halts when a zero-cost solution has been determined (this corresponds to a full, proper node k-coloring), or when the iteration limit is reached.
- `'edges'` : Here, clashes are permitted in a solution. If `weight=None`, the method seeks a k-coloring in which the number of clashes is minimized; otherwise, the method seeks a coloring that minimizes the sum of the weights of edges involved in a clash. Uncolored nodes are not permitted in a solution. The algorithm halts when a zero-cost solution has been determined (this corresponds to a full, proper node k-coloring), or when the iteration limit is reached.

it_limit
[int, optional (default=0)] Number of iterations of the local search procedure. Each iteration has a complexity $O(m + kn)$, where n is the number of nodes, m is the number of edges, and k is the number of colors.

Returns

dict
A dictionary with keys representing nodes and values representing their colors. Colors are identified by the integers $0, 1, 2, \ldots, k-1$. Uncolored nodes are given a value of `-1`.

Raises

NotImplementedError
If `G` is a directed graph or a multigraph.

ValueError
If `weights_at` is not among the supported options. If `it_limit` is not a nonnegative integer. If `k` is not a nonnegative integer. If a node/edge with a non-positive weight is specified.

KeyError
If `weights_at=='nodes'` and a node does not have the attribute defined by `weight`. If `weights_at=='edges'` and an edge does not have the attribute defined by `weight`.

See also

node_k_coloring

Notes

If `weights_at='edges'`, the TabuCol algorithm is used. This algorithm is based on tabu search and operates using k colors, allowing clashes to occur. The aim is to alter the color assignments so that the number of clashes (or the total weight of all clashing edges) is minimized. Each iteration of TabuCol has complexity $O(nk + m)$. The process also uses $O(nk + m)$ memory.

If `weights_at='nodes'`, the PartialCol algorithm is used. This algorithm is also based on tabu search and operates using k colors, allowing some nodes to be left uncolored. The aim is to make alterations to the color assignments so that the number of uncolored nodes (or the total weight of the uncolored nodes) is minimized. As with TabuCol, each iteration of PartialCol has complexity $O(nk + m)$. This process also uses $O(nk + m)$ memory.

All the above algorithms are described in detail in [1]. The c++ code used in [1] and [2] forms the basis of this library's Python implementations.

References

[1], [2]

Examples

```
>>> import networkx as nx
>>> import gcol
>>>
>>> # Unweighted graph
>>> G = nx.dodecahedral_graph()
>>> c = gcol.min_cost_k_coloring(G, 2, weights_at="nodes", it_limit=1000)
>>> P = gcol.partition(c)
>>> print(P)
[[0, 2, 8, 18, 4, 13, 15], [1, 19, 10, 6, 12, 14, 17]]
>>> for u in G:
>>>     if c[u] == -1:
>>>         print("Node", u, "is not colored")
Node 3 is not colored
Node 5 is not colored
Node 7 is not colored
Node 9 is not colored
Node 11 is not colored
Node 16 is not colored
>>>
>>> # Edge-weighted graph (arbitrary weights)
>>> for e in G.edges():
>>>     G.add_edge(e[0], e[1], weight = abs(e[0]-e[1]))
>>> c = gcol.min_cost_k_coloring(G, 2, weights_at="edges", it_limit=1000)
```

(continues on next page)

(continued from previous page)

```
>>> P = gcol.partition(c)
>>> print(P)
[[0, 2, 8, 18, 11, 7, 4, 13, 15, 16], [1, 19, 10, 3, 9, 6, 5, 12, 14, 17]]
>>> for u, v in G.edges():
>>>     if c[u] == c[v]:
>>>         print("Edge", u, v, "( cost =", G[u][v]["weight"], ") is clashing ")
Edge 3 19 ( cost = 16 ) is clashing
Edge 5 6 ( cost = 1 ) is clashing
Edge 7 8 ( cost = 1 ) is clashing
Edge 9 10 ( cost = 1 ) is clashing
Edge 11 18 ( cost = 7 ) is clashing
Edge 15 16 ( cost = 1 ) is clashing
```

gcol.coloring.**multipartite_layout**(*G*, *c*)

Arranges the nodes of the graph into columns so that those of the same color are in the same column. This method is designed to be used with the `pos` argument in the drawing functions of NetworkX (see example below).

Parameters

G
[NetworkX graph] The graph we want to visualize.

c
[dict] A dictionary with keys representing nodes and values representing their colors. Colors are identified by the integers $0, 1, 2, \ldots$. Nodes with negative color labels are ignored.

Returns

pos
[dict] A dictionary of positions keyed by node.

See also

get_node_colors
coloring_layout

Examples

```
>>> import networkx as nx
>>> import matplotlib.pyplot as plt
>>> import gcol
>>>
>>> G = nx.dodecahedral_graph()
>>> c = gcol.node_coloring(G)
>>> nx.draw_networkx(G, pos=gcol.multipartite_layout(G, c), node_color=gcol.get_
↪node_colors(G, c))
>>> plt.show()
```

gcol.coloring.**node_coloring**(*G*, *strategy='dsatur'*, *opt_alg=None*, *it_limit=0*)

Returns a coloring of a graph's nodes.

A node coloring of a graph is an assignment of colors to nodes so that adjacent nodes have different colors. The aim is to use as few colors as possible. A set of nodes assigned to the same color represents an independent set; hence the equivalent aim is to partition the graph's nodes into a minimum number of independent sets.

The smallest number of colors needed to color the nodes of a graph G is known as the graph's chromatic number, denoted by $\chi(G)$. Equivalently, $\chi(G)$ is the minimum number of independent sets needed to partition the nodes of G.

Determining a node coloring that minimizes the number of colors is an NP-hard problem. This method therefore includes options for using an exact exponential-time algorithm (based on backtracking), or a choice of two polynomial-time heuristic algorithms (based on local search). The exact algorithm is generally only suitable for graphs that are small, or that have topologies suited to its search strategies. In all other cases, the local search algorithms are more appropriate.

Parameters

G
[NetworkX graph] The nodes of this graph will be colored.

strategy
[string, optional (default='dsatur')] A string specifying the method used to generate an initial solution. It must be one of the following:

- `'random'` : Randomly orders the graph's nodes and then applies the greedy algorithm for graph node coloring [1].
- `'welsh-powell'` : Orders the graph's nodes by decreasing degree, then applies the greedy algorithm.
- `'dsatur'` : Uses the DSatur algorithm for graph node coloring [2].
- `'rlf'` : Uses the recursive largest first (RLF) algorithm for graph node coloring [3].

opt_alg
[None or int, optional (default=None)] An integer specifying the optimization method that will be used to try to reduce the number of colors. It must be one of the following

- `1` : An exact, exponential-time algorithm based on backtracking. The algorithm halts only when an optimal solution has been found.
- `2` : A local search algorithm that seeks to reduce the number of colors by temporarily allowing adjacent nodes to have the same color. Each iteration has a complexity $O(m + kn)$, where n is the number of nodes in the graph, m is the number of edges, and k is the number of colors in the current solution.
- 3 : A local search algorithm that seeks to reduce the number of colors by temporarily allowing nodes to be uncolored. Each iteration has a complexity $O(m + kn)$, as above.
- None : No optimization is performed.

it_limit
[int, optional (default=0)] Number of iterations of the local search procedure. Only applicable when using `opt_alg=2` or `opt_alg=3`.

Returns

dict
A dictionary with keys representing nodes and values representing their colors. Colors are identified by the integers $0, 1, 2, \ldots$. The number of colors being used in a solution `c` is therefore `max(c.values()) + 1`.

Raises

NotImplementedError
If `G` is a directed graph or a multigraph.

ValueError
If `strategy` is not among the supported options. If `opt_alg` is not among the supported options. If `it_limit` is not a nonnegative integer.

See also

chromatic_number
node_k_coloring
edge_coloring

Notes

Given a graph $G = (V, E)$ with n nodes and m edges, the greedy algorithm for node coloring operates in $O(n + m)$ time.

The `random` strategy operates by first randomly permuting the nodes (an $O(n)$ operation) before applying the greedy algorithm. It is guaranteed to produce a solution with $k \leq \Delta(G) + 1$ colors, where $\Delta(G)$ is the highest node degree in the graph G.

The `welsh-powell` strategy operates by sorting the nodes by decreasing degree (an $O(n \lg n)$ operation), and then applies the greedy algorithm. Its overall complexity is therefore $O(n \lg n + m)$. Assuming that the nodes are labelled $v_1, v_2, \ldots, v_n$ so that $\deg(v_1) \geq \deg(v_2) \geq \ldots \geq \deg(v_n)$, this method is guaranteed to produce a solution with $k \leq \max_{i=1,\ldots,n} \min(\deg(v_i) + 1, i)$ colors. This bound is an improvement on $\Delta(G) + 1$.

The `dsatur` and `rlf` strategies are exact for bipartite, cycle, and wheel graphs (that is, solutions with the minimum number of colors are guaranteed). The implementation of `dsatur` uses a priority queue and has a complexity of $O(n \lg n + m \lg m)$. The `rlf` implementation has a complexity of $O(nm)$. In general, the `rlf` strategy yields the best solutions of the four strategies, though it is computationally more expensive. If expense is an issue, then `dsatur` is a cheaper alternative that also offers high-quality solutions in most cases. See [2], [3], and [4] for further information.

If an optimization algorithm is used, further efforts are made to reduce the number of colors. The backtracking approach (`opt_alg=1`) is an implementation of the exact algorithm described in [4]. It has exponential runtime and halts only when an optimum solution has been found. At the start of execution, a large clique $C \subseteq V$ is identified using the NetworkX function `max_clique(G)` and the nodes of C are each assigned to a different color. The main backtracking algorithm is then executed and only halts only when a solution using $|C|$ colors has been identified, or when the algorithm has backtracked to the root of the search tree. In both cases the returned solution will be optimal (that is, will be using $\chi(G)$ colors).

If local search is used (`opt_alg=2` or `opt_alg=3`), the algorithm removes a color class and uses the chosen local search routine to seek a proper coloring using the remaining colors. If this is successful, the process repeats. The algorithm is executed until a solution using $|C|$ colors has been identified (as above), or until the iteration limit is reached. Fewer colors (but longer run times) occur with larger iteration limits.

If `opt_alg=2`, the TabuCol algorithm is used. This algorithm is based on tabu search and operates by fixing the number of colors but allowing clashes to occur (a clash is the occurrence of two adjacent nodes having the same color). The aim is to alter the color assignments so that the number of clashes is reduced to zero. Each iteration of TabuCol has complexity $O(nk + m)$, where k is the number of colors currently being used. The process also uses $O(nk + m)$ memory.

If `opt_alg=3`, the PartialCol algorithm is used. This algorithm is also based on tabu search and operates by fixing the number of colors but allowing some nodes to be left uncolored. The aim is to make alterations to the color assignments so that no uncolored nodes remain. As with TabuCol, each iteration of PartialCol has complexity $O(nk + m)$ and uses $O(nk + m)$ memory.

All the above algorithms and bounds are described in detail in [4]. The c++ code used in [4] and [5] forms the basis of this library's Python implementations.

References

[1], [2], [3], [4], [5]

Examples

```
>>> import networkx as nx
>>> import gcol
>>>
>>> G = nx.dodecahedral_graph()
>>> c = gcol.node_coloring(G)
>>> print("Coloring is", c)
Coloring is {0: 0, 1: 1, 19: 1, 10: 1, 2: 0, 3: 2, 8: 0, 9: 2, 18: 0, 11: 2, 6: 1,␣
↪7: 2, 4: 0, 5: 2, 13: 0, 12: 1, 14: 1, 15: 0, 16: 2, 17: 1}
>>> print("Number of colors =", max(c.values()) + 1)
Number of colors = 3
>>>
>>> print("Partition view =", gcol.partition(c))
Partition view = [[0, 2, 8, 18, 4, 13, 15], [1, 19, 10, 6, 12, 14, 17], [3, 9, 11,␣
↪7, 5, 16]]
>>>
>>> # Example with a larger graph and different parameters
>>> G = nx.gnp_random_graph(50, 0.2, seed=1)
>>> c = gcol.node_coloring(G, strategy="dsatur", opt_alg=2, it_limit=1000)
>>> print("Coloring is", c)
Coloring is {18: 0, 31: 2, 2: 4, 20: 1, 10: 3, 46: 0, 49: 1, 29: 3, 37: 2, 9: 1, 7:␣
↪2, 33: 0, 21: 4, 26: 2, 5: 4, 16: 0, 41: 1, 39: 0, 13: 3, 14: 4, 17: 3, 28: 0,␣
↪35: 1, 42: 4, 4: 4, 11: 3, 3: 2, 48: 3, 40: 3, 0: 0, 30: 0, 6: 2, 8: 3, 25: 1,␣
↪34: 0, 44: 3, 24: 1, 1: 4, 47: 4, 15: 1, 23: 4, 32: 4, 45: 0, 22: 1, 43: 4, 36: 2,
↪ 19: 3, 12: 3, 38: 1, 27: 2}
>>>
>>> print("Number of colors =", max(c.values()) + 1)
Number of colors = 5
```

gcol.coloring.**node_k_coloring**(*G*, *k*, *opt_alg=None*, *it_limit=0*)

Attempts to color the nodes of a graph using k colors so that adjacent nodes have different colors. A set of nodes assigned to the same color corresponds to an independent set; hence the equivalent aim is to partition the graph's nodes into k independent sets.

Determining whether a node k-coloring exists for G is NP-complete. This method therefore includes options for using an exact exponential-time algorithm (based on backtracking), or a choice of two polynomial-time heuristic algorithms (based on local search). The exact algorithm is generally only suitable for larger values of k, for graphs that are small, or graphs that have topologies suited to its search strategies. In all other cases, the local search algorithms are more appropriate.

If a node k-coloring cannot be determined by the algorithm, a `ValueError` exception is raised. Otherwise, a node k-coloring is returned.

Parameters

G

[NetworkX graph] The nodes of this graph will be colored.

k
[int] The number of colors to use.

opt_alg
[None or int, optional (default=None)] An integer specifying the optimization method that will be used to try to reduce the number of colors (if this is seen to be greater than k). It must be one of the following

- `1` : An exact, exponential-time algorithm based on backtracking. The algorithm halts only when the existence of a node k-coloring has been proved or disproved.
- `2` : A local search algorithm that seeks to reduce the number of colors by temporarily allowing adjacent nodes to have the same color. Each iteration has a complexity $O(m + kn)$, where n is the number of nodes in the graph, m is the number of edges, and k is the number of colors in the current solution.
- `3` : A local search algorithm that seeks to reduce the number of colors by temporarily allowing nodes to be uncolored. Each iteration has a complexity $O(m + kn)$, as above.
- None : No optimization is performed.

it_limit
[int, optional (default=0)] Number of iterations of the local search procedure. Only applicable when using `opt_alg=2` or `opt_alg=3`.

Returns

dict
A dictionary with keys representing edges and values representing their colors. Colors are identified by the integers $0, 1, 2, \ldots, k-1$.

Raises

NotImplementedError
If `G` is a directed graph or a multigraph.

ValueError
If `opt_alg` is not among the supported options. If `it_limit` is not a nonnegative integer. If `k` is not a nonnegative integer. If a clique larger than `k` is observed in the graph. If a node k-coloring could not be determined.

See also

node_coloring
equitable_node_k_coloring
edge_k_coloring

Notes

This method begins by coloring the nodes in the order determined by the DSatur algorithm [1]. During this process, each node is assigned to the feasible color class j (where $0 \leq j \leq k$) with the fewest nodes. This encourages an equitable spread of nodes across the k colors. This process has a complexity of $O((n \lg n) + (nk) + (m \lg m)$. If a node k-coloring cannot be achieved in this way, further optimization is carried out, if desired. These optimization routines are the same as those used by the *node_coloring()* method. They also halt immediately once a node k-coloring has been achieved.

All the above algorithms are described in detail in [2]. The c++ code used in [2] and [3] forms the basis of this library's Python implementations.

References

[1], [2], [3]

Examples

```
>>> import networkx as nx
>>> import gcol
>>>
>>> G = nx.dodecahedral_graph()
>>> c = gcol.node_k_coloring(G, 4)
>>> print(c)
{0: 0, 1: 1, 19: 2, 10: 3, 2: 0, 3: 1, 8: 2, 9: 0, 18: 3, 11: 1, 6: 2, 4: 3, 5: 0,␣
↪7: 1, 13: 2, 12: 3, 14: 0, 17: 1, 16: 2, 15: 3}
>>>
>>> c = gcol.node_k_coloring(G, 3)
>>> print(c)
{0: 0, 1: 1, 19: 2, 10: 1, 2: 0, 3: 1, 8: 2, 9: 0, 18: 0, 11: 2, 6: 1, 7: 0, 4: 2,␣
↪5: 0, 17: 1, 13: 2, 14: 1, 15: 2, 16: 0, 12: 1}
```

gcol.coloring.**node_precoloring**(*G*, *precol=None*, *strategy='dsatur'*, *opt_alg=None*, *it_limit=0*)

Returns a coloring of a graph's nodes in which some of the nodes have been precolored.

A node coloring of a graph is an assignment of colors to nodes so that adjacent nodes have different colors. The aim is to use as few colors as possible. A set of nodes assigned to the same color corresponds to an independent set; hence the equivalent aim is to partition the graph's nodes into a minimum number of independent sets.

In the node precoloring problem, some of the nodes have already been assigned colors. The aim is to allocate colors to the remaining nodes so that we get a full, proper node coloring that uses a minimum number of colors. The node precoloring problem can be used to model the Latin square completion problem and Sudoku puzzles [1].

The node precoloring problem is NP-hard. This method therefore includes options for using an exponential-time exact algorithm (based on backtracking), or a choice of two polynomial-time heuristic algorithms (based on local search). The exact algorithm is generally only suitable for graphs that are small, or that have topologies suited to its search strategies. In all other cases, the local search algorithms are more appropriate.

In this implementation, solutions are found by taking all nodes pre-allocated to the same color j and merging them into a single super-node. Edges are then added between all pairs of super-nodes, and the modified graph is passed to the *node_coloring()* method. All parameters are therefore the same as the latter. This modification process is described in more detail in Chapter 6 of [1].

Parameters

G
: [NetworkX graph] The nodes of this graph will be colored.

precol
: [None or dict, optional (default=None)] A dictionary that specifies the (integer) colors of any precolored nodes.

strategy
: [string, optional (default='dsatur')] A string specifying the method used to generate the initial solution. It must be one of the following:

  - `'random'` : Randomly orders the modified graph's nodes and then applies the greedy algorithm for graph node coloring [2].

- `'welsh-powell'` : Orders the modified graphs nodes by decreasing degree, then applies the greedy algorithm.
- `'dsatur'` : Uses the DSatur algorithm for graph node coloring on the modified graph [3].
- `'rlf'` : Uses the recursive largest first (RLF) algorithm for graph node coloring on the modified graph [4].

opt_alg
[None or int, optional (default=None)] An integer specifying the optimization method that will be used to try to reduce the number of colors. It must be one of the following

- `1` : An exact, exponential-time algorithm based on backtracking. The algorithm halts only when an optimal solution has been found.
- `2` : A local search algorithm that seeks to reduce the number of colors by temporarily allowing adjacent nodes to have the same color. Each iteration has a complexity $O(m + kn)$, where n is the number of nodes in the modified graph, m is the number of edges, and k is the number of colors in the current solution.
- `3` : A local search algorithm that seeks to reduce the number of colors by temporarily allowing nodes to be uncolored. Each iteration has a complexity $O(m + kn)$, as above.
- `None` : No optimization is performed.

it_limit
[int, optional (default=0)] Number of iterations of the local search procedure. Only applicable when using `opt_alg=2` or `opt_alg=3`.

Returns

dict
A dictionary with keys representing nodes and values representing their colors. Colors are identified by the integers $0, 1, 2, \ldots$. The number of colors being used in a solution `c` is therefore `max(c.values()) + 1`. If `precol[v]==j` then `c[v]==j`.

Raises

NotImplementedError
If `G` is a directed graph or a multigraph.

ValueError
If `strategy` is not among the supported options. If `opt_alg` is not among the supported options. If `it_limit` is not a nonnegative integer. If `G` contains a node with the name `'super'`. If `precol` contains a node that is not in `G`. If `precol` contains a non-integer color label. If `precol` contains a pair of adjacent nodes assigned the same color. If `precol` uses an integer color label j, but there exists a color label $0 \leq i < j$ that is not being used.

See also

node_coloring
edge_precoloring

Notes

As mentioned, in this implementation, solutions are formed by passing a modified version of the graph to *node_coloring()* method. All details are therefore the same as those in the latter, where they are documented.

All the above algorithms and bounds are described in detail in [1]. The c++ code used in [1] and [5] forms the basis of this library's Python implementations.

References

[1], [2], [3], [4], [5]

Examples

```
>>> import networkx as nx
>>> import gcol
>>>
>>> G = nx.dodecahedral_graph()
>>> p = {0:1, 8:0, 9:1}
>>> c = gcol.node_precoloring(G, precol=p)
>>> print("Coloring is", c)
Coloring is {0: 1, 9: 1, 1: 2, 8: 0, 19: 2, 13: 2, 2: 1, 3: 0, 7: 1, 14: 0, 18: 1,␣
↪12: 1, 6: 2, 4: 1, 5: 0, 15: 1, 11: 2, 10: 0, 17: 2, 16: 0}
>>>
>>> p = {i:i for i in range(5)}
>>> c = gcol.node_precoloring(G, precol=p, strategy="dsatur", opt_alg=2, it_
↪limit=1000)
>>> print(c)
{0: 0, 4: 4, 1: 1, 2: 2, 3: 3, 19: 4, 10: 4, 5: 0, 6: 4, 17: 0, 18: 1, 11: 0, 8: 0,␣
↪7: 1, 9: 1, 15: 4, 14: 0, 16: 1, 13: 4, 12: 2}
```

gcol.coloring.**partition**(*c*)

Converts a coloring into its equivalent partition-based representation. Negative color labels (signifying uncolored nodes/edges) are ignored.

Parameters

c

[dict] A dictionary with keys representing nodes or edges and values representing their colors. Colors are identified by the integers $0, 1, 2, \ldots$.

Returns

list

A list in which each element is a list containing the nodes/edges assigned to a particular color.

See also

equitable_node_k_coloring
equitable_edge_k_coloring

Notes

If all nodes in a color class are named by numerical values, the nodes are sorted in ascending order. Otherwise, the nodes of each color class are sorted by their string equivalents.

Examples

```
>>> import networkx as nx
>>> import gcol
>>>
>>> G = nx.dodecahedral_graph()
>>> c = gcol.node_coloring(G)
>>> print(gcol.partition(c))
[[0, 2, 8, 18, 4, 13, 15], [1, 19, 10, 6, 12, 14, 17], [3, 9, 11, 7, 5, 16]]
>>>
>>> c = gcol.edge_coloring(G)
>>> print(gcol.partition(c))
[[(11, 12), (18, 19), (16, 17), (0, 10), (9, 13), (14, 15), (3, 4), (1, 2), (5, 6),␣
↪(7, 8)], [(11, 18), (12, 16), (4, 17), (9, 10), (13, 14), (5, 15), (0, 19), (2,␣
↪3), (1, 8), (6, 7)], [(10, 11), (12, 13), (17, 18), (15, 16), (8, 9), (7, 14), (3,
↪ 19), (0, 1), (2, 6), (4, 5)]]
```

- genindex
- modindex
- search

BIBLIOGRAPHY

[1] Wikipedia: Vizing's Theorem <https://en.wikipedia.org/wiki/Vizing%27s_theorem>

[2] Lewis, R. (2021) A Guide to Graph Colouring: Algorithms and Applications (second ed.). Springer. ISBN: 978-3-030-81053-5. <https://link.springer.com/book/10.1007/978-3-030-81054-2>.

[3] Lewis, R: Graph Colouring Algorithm User Guide <https://rhydlewis.eu/gcol/>

[1] Lewis, R. (2021) A Guide to Graph Colouring: Algorithms and Applications (second ed.). Springer. ISBN: 978-3-030-81053-5. <https://link.springer.com/book/10.1007/978-3-030-81054-2>.

[2] Lewis, R: Graph Colouring Algorithm User Guide <https://rhydlewis.eu/gcol/>

[1] Wikipedia: Vizing's Theorem <https://en.wikipedia.org/wiki/Vizing%27s_theorem>

[2] Wikipedia: Greedy Coloring <https://en.wikipedia.org/wiki/Greedy_coloring>

[3] Wikipedia: DSatur <https://en.wikipedia.org/wiki/DSatur>

[4] Wikipedia: Recursive largest first (RLF) algorithm <https://en.wikipedia.org/wiki/Recursive_largest_first_algorithm>

[5] Lewis, R. (2021) A Guide to Graph Colouring: Algorithms and Applications (second ed.). Springer. ISBN: 978-3-030-81053-5. <https://link.springer.com/book/10.1007/978-3-030-81054-2>.

[6] Lewis, R: Graph Colouring Algorithm User Guide <https://rhydlewis.eu/gcol/>

[1] Wikipedia: Vizing's Theorem <https://en.wikipedia.org/wiki/Vizing%27s_theorem>

[2] Lewis, R. (2021) A Guide to Graph Colouring: Algorithms and Applications (second ed.). Springer. ISBN: 978-3-030-81053-5. <https://link.springer.com/book/10.1007/978-3-030-81054-2>.

[3] Lewis, R: Graph Colouring Algorithm User Guide <https://rhydlewis.eu/gcol/>

[1] Wikipedia: Greedy Coloring <https://en.wikipedia.org/wiki/Greedy_coloring>

[2] Wikipedia: DSatur <https://en.wikipedia.org/wiki/DSatur>

[3] Wikipedia: Recursive largest first (RLF) algorithm <https://en.wikipedia.org/wiki/Recursive_largest_first_algorithm>

[4] Lewis, R. (2021) A Guide to Graph Colouring: Algorithms and Applications (second ed.). Springer. ISBN: 978-3-030-81053-5. <https://link.springer.com/book/10.1007/978-3-030-81054-2>.

[5] Lewis, R: Graph Colouring Algorithm User Guide <https://rhydlewis.eu/gcol/>

[1] Wikipedia: Vizing's Theorem <https://en.wikipedia.org/wiki/Vizing%27s_theorem>

[2] Lewis, R. (2021) A Guide to Graph Colouring: Algorithms and Applications (second ed.). Springer. ISBN: 978-3-030-81053-5. <https://link.springer.com/book/10.1007/978-3-030-81054-2>.

[3] Lewis, R. and F. Carroll (2016) 'Creating Seating Plans: A Practical Application'. Journal of the Operational Research Society, vol. 67(11), pp. 1353-1362.

[4] Lewis, R: Graph Colouring Algorithm User Guide <https://rhydlewis.eu/gcol/>

[1] Wikipedia: Kempe Chain <https://en.wikipedia.org/wiki/Kempe_chain>

[2] Lewis, R. (2021) A Guide to Graph Colouring: Algorithms and Applications (second ed.). Springer. ISBN: 978-3-030-81053-5. <https://link.springer.com/book/10.1007/978-3-030-81054-2>.

[3] Lewis, R. and F. Carroll (2016) 'Creating Seating Plans: A Practical Application'. Journal of the Operational Research Society, vol. 67(11), pp. 1353-1362.

[4] Lewis, R: Graph Colouring Algorithm User Guide <https://rhydlewis.eu/gcol/>

[1] Wikipedia: Kempe Chain <https://en.wikipedia.org/wiki/Kempe_chain>

[2] Cranston, D. (2024) Graph Coloring Methods <https://graphcoloringmethods.com/>

[1] Lewis, R. (2021) A Guide to Graph Colouring: Algorithms and Applications (second ed.). Springer. ISBN: 978-3-030-81053-5. <https://link.springer.com/book/10.1007/978-3-030-81054-2>.

[2] Lewis, R: Graph Colouring Algorithm User Guide <https://rhydlewis.eu/gcol/>

[1] Lewis, R. (2021) A Guide to Graph Colouring: Algorithms and Applications (second ed.). Springer. ISBN: 978-3-030-81053-5. <https://link.springer.com/book/10.1007/978-3-030-81054-2>.

[2] Lewis, R: Graph Colouring Algorithm User Guide <https://rhydlewis.eu/gcol/>

[1] Wikipedia: Greedy Coloring <https://en.wikipedia.org/wiki/Greedy_coloring>

[2] Wikipedia: DSatur <https://en.wikipedia.org/wiki/DSatur>

[3] Wikipedia: Recursive largest first (RLF) algorithm <https://en.wikipedia.org/wiki/Recursive_largest_first_algorithm>

[4] Lewis, R. (2021) A Guide to Graph Colouring: Algorithms and Applications (second ed.). Springer. ISBN: 978-3-030-81053-5. <https://link.springer.com/book/10.1007/978-3-030-81054-2>.

[5] Lewis, R: Graph Colouring Algorithm User Guide <https://rhydlewis.eu/gcol/>

[1] Wikipedia: DSatur <https://en.wikipedia.org/wiki/DSatur>

[2] Lewis, R. (2021) A Guide to Graph Colouring: Algorithms and Applications (second ed.). Springer. ISBN: 978-3-030-81053-5. <https://link.springer.com/book/10.1007/978-3-030-81054-2>.

[3] Lewis, R: Graph Colouring Algorithm User Guide <https://rhydlewis.eu/gcol/>

[1] Lewis, R. (2021) A Guide to Graph Colouring: Algorithms and Applications (second ed.). Springer. ISBN: 978-3-030-81053-5. <https://link.springer.com/book/10.1007/978-3-030-81054-2>.

[2] Wikipedia: Greedy Coloring <https://en.wikipedia.org/wiki/Greedy_coloring>

[3] Wikipedia: DSatur <https://en.wikipedia.org/wiki/DSatur>

[4] Wikipedia: Recursive largest first (RLF) algorithm <https://en.wikipedia.org/wiki/Recursive_largest_first_algorithm>

[5] Lewis, R: Graph Colouring Algorithm User Guide <https://rhydlewis.eu/gcol/>

PYTHON MODULE INDEX