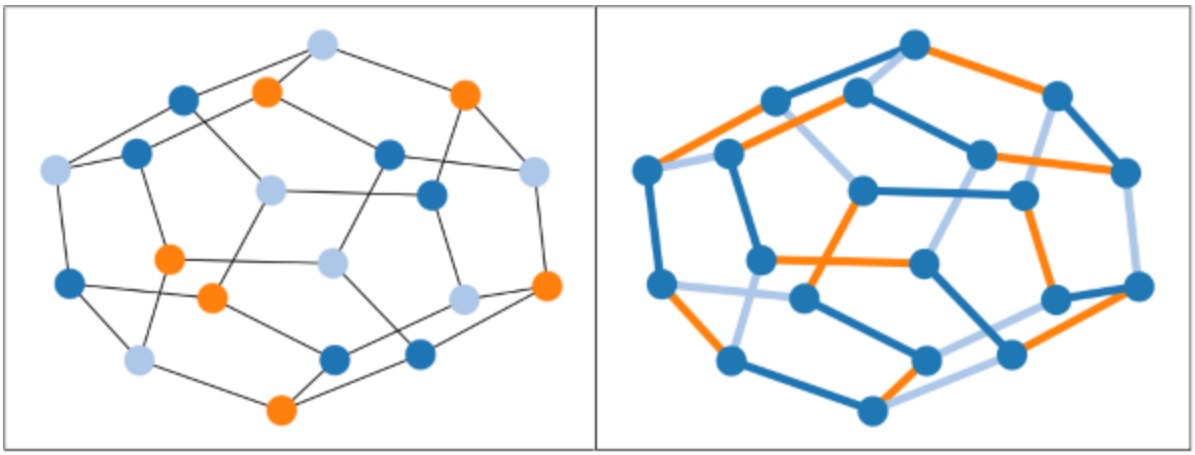
# GCol

*Release 2.2*

**R. Lewis**

**Feb 13, 2026**

# 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, face coloring, equitable coloring, weighted coloring, precoloring, list coloring, and maximum independent set identification. It also offers several tools for solution visualization.

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

CHAPTER

ONE

# GENERAL INFO

Welcome to the documentation of GCol – a Python library for graph coloring!

## 1.1 Quick Start

To install the GCol library, type the following at a command prompt:

```
python -m pip install gcol
```

or execute the following in a notebook:

```
!python -m pip install gcol
```

then restart the kernal. To start using this library, try executing the following code.

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

G = nx.dodecahedral_graph()
c = gcol.node_coloring(G)
print("Here is a node coloring of graph G:", c)
nx.draw_networkx(G, node_color=gcol.get_node_colors(G, c))
plt.show()
```

*This demonstration* gives further examples. You may also find it useful to consult the user guide of NetworkX, as GCol makes use of its data structures and functionality.

## 1.2 Publications

The algorithms and techniques used in this library come from the 2021 textbook of Lewis, R. (2021) A Guide to Graph Colouring: Algorithms and Applications, Springer Cham. (2nd Edition). In bibtex, this book is cited as:

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

A short description of this library is also published in the Journal of Open Source Software:

```
@article{10.21105/joss.07871,
  author = {Lewis, R. and Palmer, G.},
  title = {GCol: A High-Performance Python Library for Graph Colouring},
  journal = {Journal of Open Source Software},
  year = {2025},
  volume = {10},
  number = {108},
  pages = {7871},
  doi = {10.21105/joss.07871}
}
```

## 1.3 Support

The GCol repository is hosted on github. If you have any questions, please ask them on stackoverflow adding the tag `graph-coloring`. All documentation is listed on this website or, if you prefer, in this pdf document. If you have any suggestions for this library or notice any bugs, please contact the author using the contact details at www.rhydlewis.eu.

## 1.4 MIT License

CHAPTER

TWO

# AN INTRODUCTION TO THE GCOL LIBRARY

In this chapter, 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 -m pip install gcol`, or executing the following command in a notebook: `!python -m pip install gcol`, and then restarting the kernel.

Let us first review some basic terminology in graph theory.

- A graph is an object comprising a set of nodes that are linked by edges. They can be 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 $\chi(G)$. Identifying $\chi(G)$ is NP-hard.
- 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 to color 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 $G$. However, identifying $\chi'(G)$ is still NP-hard.
- A face coloring of a planar graph is an assignment of colors to one of its planar embeddings so that all pairs of adjacent faces have different colors. The aim is to use as few colors as possible. According to Euler's formula, an embedding of a planar graph with $n$ nodes and $m$ edges has exactly $m-n+2$ faces, including the external face. The smallest number of colors needed to color the faces of a planar embedding is known as its face chromatic number. Due to the Four Color Theorem, the face chromatic number is always less than or equal to four. The problem of determining a face four-coloring of a graph is polynomially solvable; however, determining whether or not the face chromatic number of a graph is three is NP-complete. Note that face colorings only exist in graphs that have a planar embedding. A graph has a planar embedding if and only if it is planar.
- 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 and face precoloring problem. The problem is NP-hard.
- In the list coloring problem, each node of the graph is associated with a list of *allowed colors*. The aim is to allocate colors to all nodes so that adjacent nodes have different colors, and each node has a color belonging to its list of allowed colors. Again, the same concepts apply for the edge list coloring problem and face list coloring problem. The problem is also NP-hard.

## 2.1 Getting Started

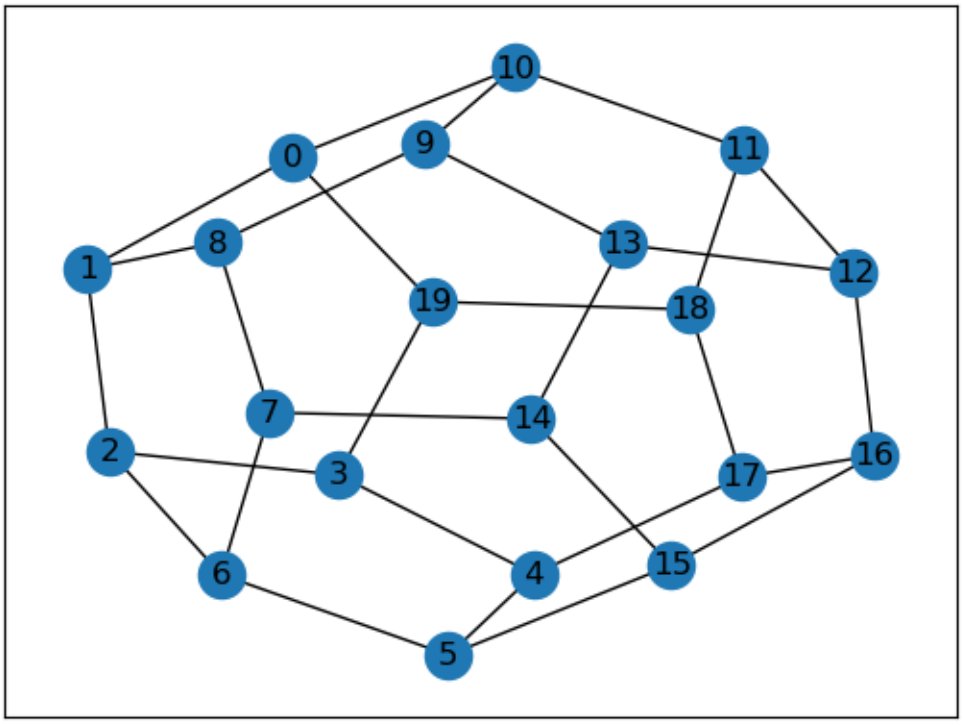
We start by importing the libraries we need. The `networkx` library is used to generate and visualize the graphs, the `matplotlib.pyplot` is used to render the drawings, and the `gcol` library contains all of our graph coloring routines.

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

## 2.2 Node Coloring and Visualization

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

```
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 do this. 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 pairs of adjacent nodes are always assigned to different colors, as required.

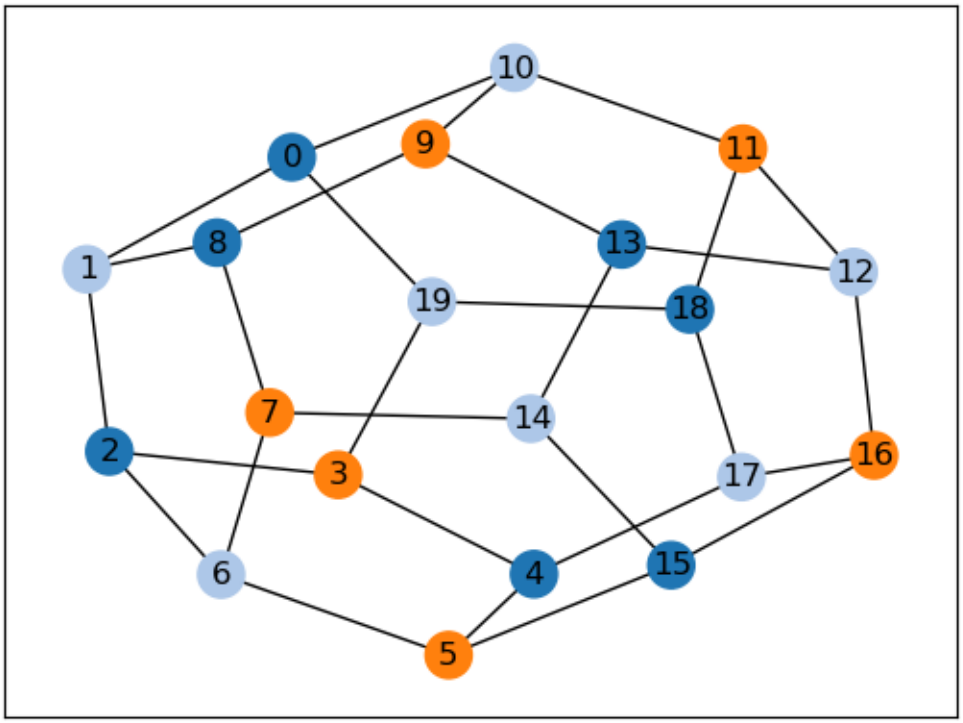
```
G = nx.dodecahedral_graph()
c = gcol.node_coloring(G)
print("Here is a node coloring of the above graph:", c)
print("The number of colors in this solution is:", max(c.values()) + 1)
print("Here is the same solution, expressed as a partition of the nodes:", gcol.
↪partition(c))
print("Here is a picture of the coloring:")
nx.draw_networkx(
    G,
    pos=nx.spring_layout(G, seed=1),
```

(continues on next page)

(continued from previous page)

```
    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 the coloring:
```




We can also write similar commands to determine the chromatic number, chromatic index, and face chromatic number 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))
print("The face chromatic number of this graph is:", gcol.face_chromatic_number(G))
```

```
The chromatic number of this graph is: 3
The chromatic index of this graph is: 3
The face chromatic number of this graph is: 4
```

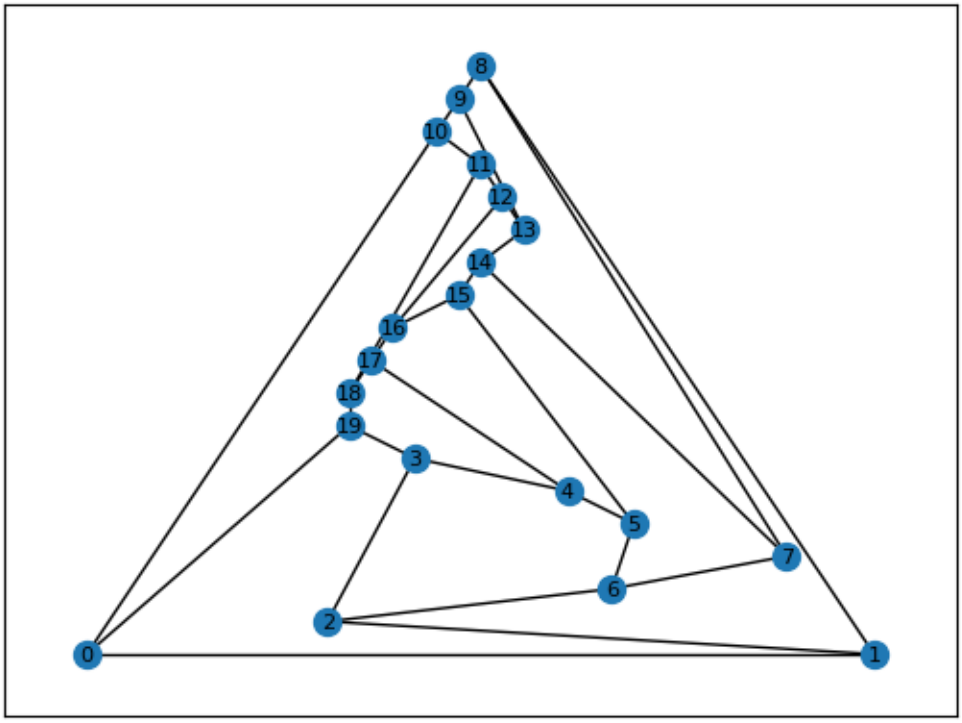
Above, we are allowed to use the `gcol.face_chromatic_number()` method, because the graph can be represented as a planar embedding. (If the graph does not have a planar embedding, an error will be raised.) Here is an example planar embedding. This is merely a positioning of the nodes so that no edges cross in the resultant visualization.

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

(continues on next page)

(continued from previous page)

```
    G,
    pos=nx.planar_layout(G),
    node_size=100,
    font_size=8
)
plt.show()
```

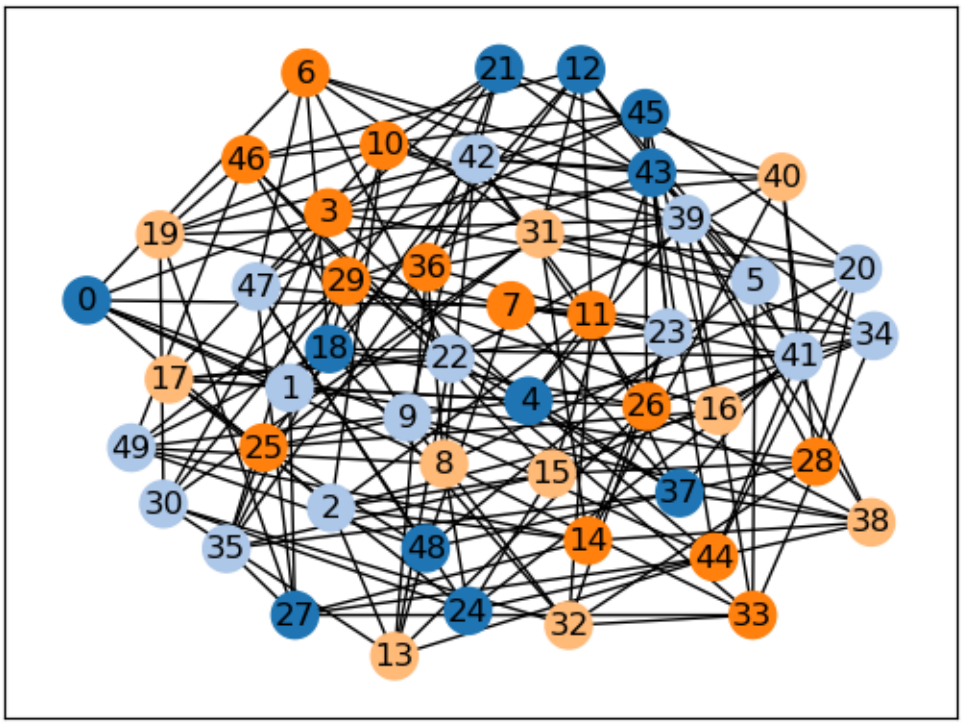



Observe that, although it looks different, this graph is equivalent to the graph shown in the previous figure.

We now show a node coloring of a slightly larger example, the so-called Hoffman-Singleton graph. This has 50 nodes, 175 edges and, as shown, a chromatic number of four.

```
G = nx.hoffman_singleton_graph()
c = gcol.node_coloring(G, opt_alg=1)
print("The number of colors in this solution is:", max(c.values()) + 1)
nx.draw_networkx(
    G,
    pos=nx.spring_layout(G, seed=1),
    node_color=gcol.get_node_colors(G, c)
)
plt.show()
```

```
The number of colors in this solution is: 4
```




To some, this visualization might seem quite cluttered; however, steps can be taken to remedy this, as we will see in the next chapter.

## 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 the solution produced is optimal.

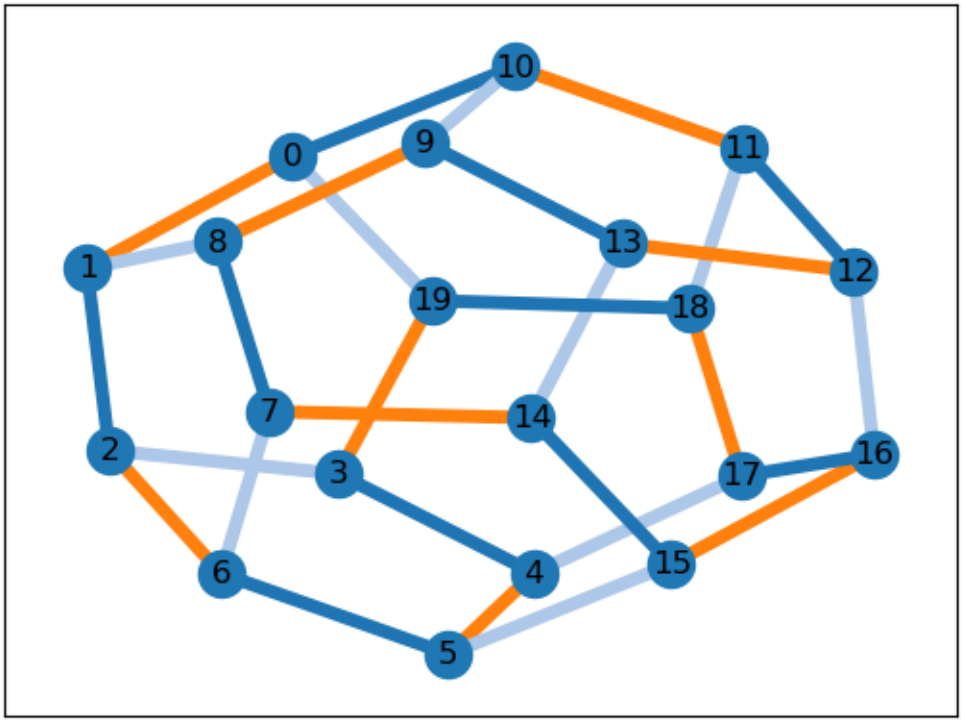
```
G = nx.dodecahedral_graph()
c = gcol.edge_coloring(G)
print("Here is the color of each edge:", c)
print("Here is the same solution, expressed as a partition of the edges:", gcol.
↪partition(c))
print("Maximum degree   =", max(G.degree(v) for v in G))
print("Number of colors =", max(c.values()) + 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}
```

(continues on next page)

(continued from previous page)

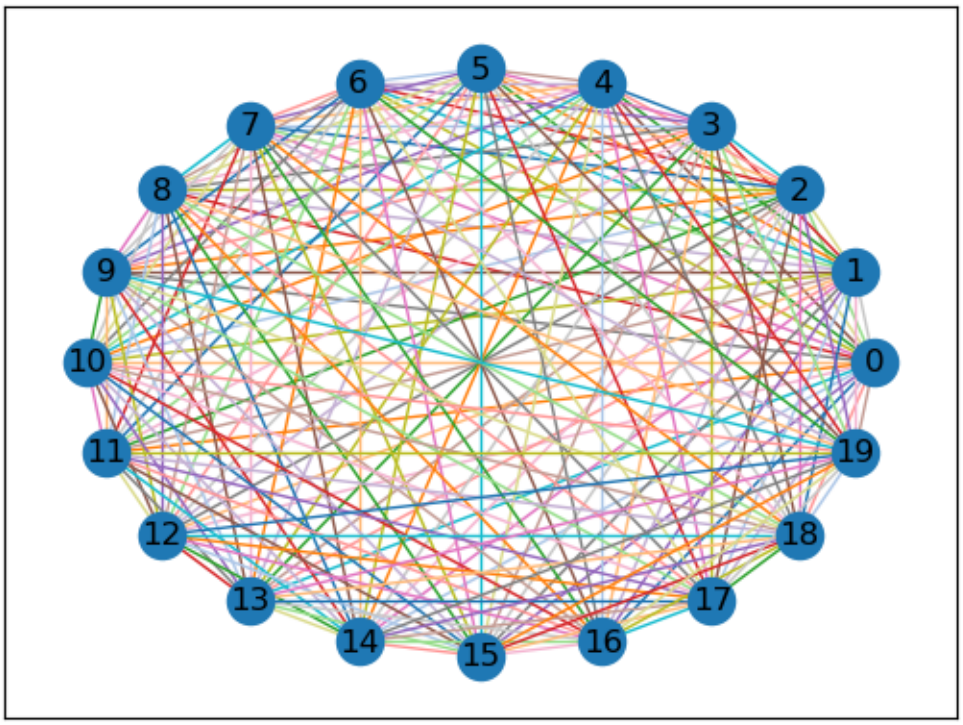
```
Here is the same solution, expressed as a partition of the edges: [[(0, 10), (1, 2), (11,
↪ 12), (14, 15), (16, 17), (18, 19), (3, 4), (5, 6), (7, 8), (9, 13)], [(0, 19), (1, 8),
↪ (11, 18), (12, 16), (13, 14), (2, 3), (4, 17), (5, 15), (6, 7), (9, 10)], [(0, 1),␣
↪(10, 11), (12, 13), (15, 16), (17,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




## 2.4 Face Coloring and Visualization

As noted earlier, a face coloring is an assignment of colors to each face in a planar embedding so that all pairs of adjacent faces have different colors. Since non-planar graphs do not have planar embeddings, the following routines are only suitable for planar graphs. Planar graphs featuring bridges are also disallowed, as these give faces that are adjacent to themselves. In the `gcol` library, face colorings of an embedding are determined by node-coloring its dual graph.

Below, we generate a two-dimensional grid graph `G`. In the dictionary `pos`, we also specify suitable $(x, y)$ coordinates for each node. These positions define the planar embedding. After determining a face coloring `c`, it is drawn to the screen using the `gcol.draw_face_coloring()` method. By default, the external face is not colored in the first visualization; however, we can change this by setting `external=True`, as shown in the second figure. The second figure also shows how the original graph can be added to the image to show the boundaries of each face.

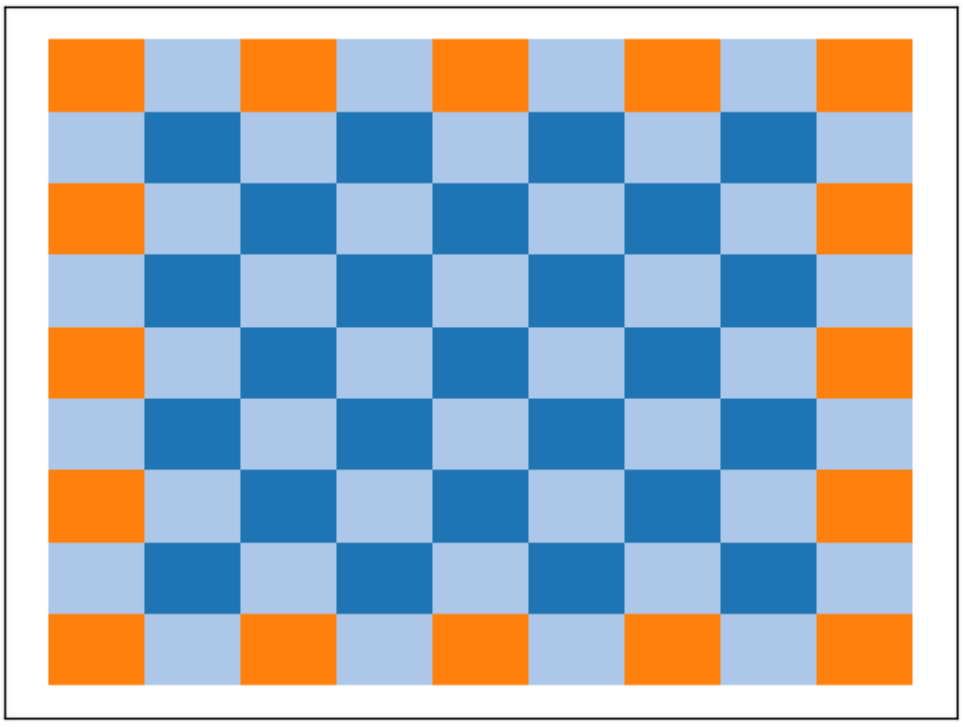
Here, the regular structure of grid graphs has allowed us to easily generate positions for each node. In some cases, however, a method of doing this may not be obvious. The third example therefore demonstrates how positions for each node can be determined using the NetworkX command `nx.planar_layout()`. This arranges the nodes in a mountain-like shape.

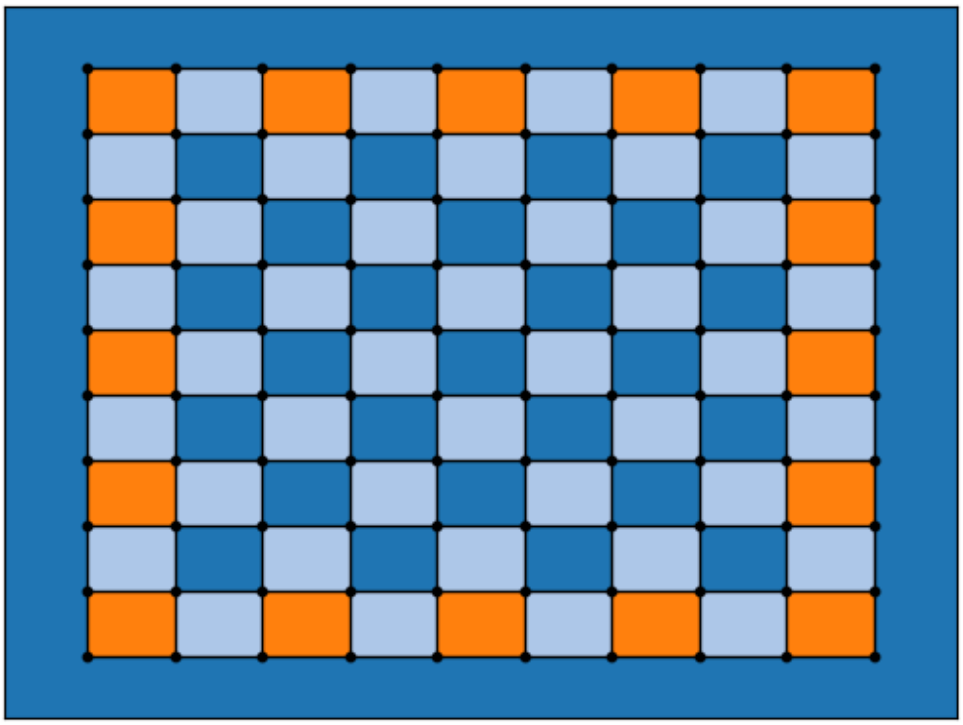
```
G = nx.grid_2d_graph(10, 10)
pos = dict((u, u) for u in G.nodes())
c = gcol.face_coloring(G, pos)

gcol.draw_face_coloring(c, pos)

gcol.draw_face_coloring(c, pos, external=True)
nx.draw_networkx(
    G,
    pos=pos,
```

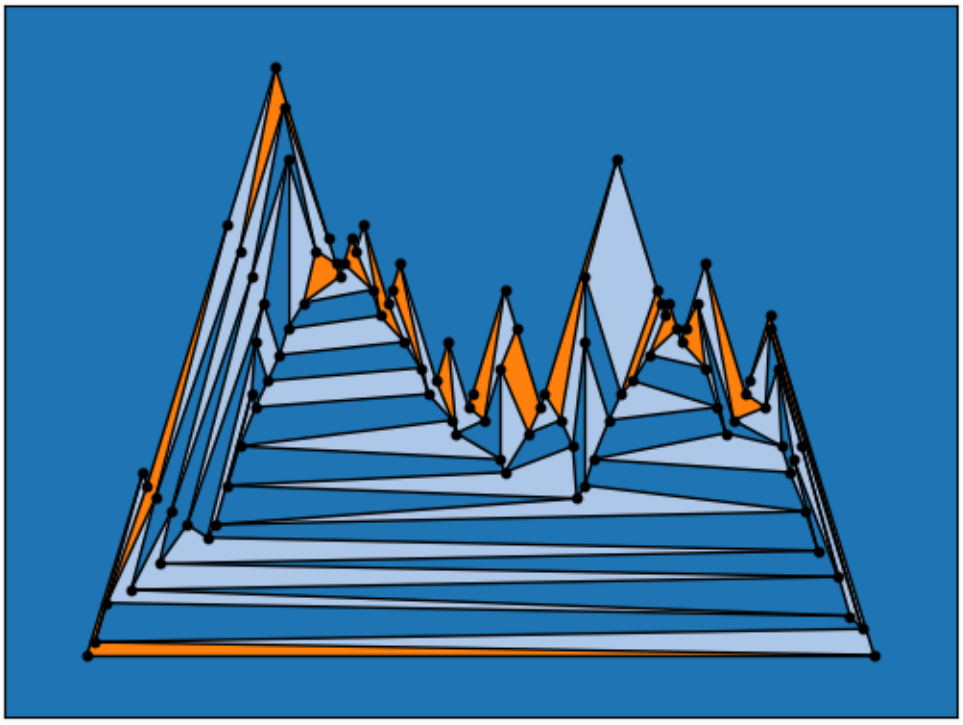
(continues on next page)

(continued from previous page)

```
    node_color='k',
    node_size=10,
    with_labels=False
)
plt.show()

pos = nx.planar_layout(G)
c = gcol.face_coloring(G, pos)
gcol.draw_face_coloring(c, pos, external=True)
nx.draw_networkx(
    G,
    pos=pos,
    node_color='k',
    node_size=10,
    with_labels=False
)
plt.show()
print("Face chromatic number =", gcol.face_chromatic_number(G))
```



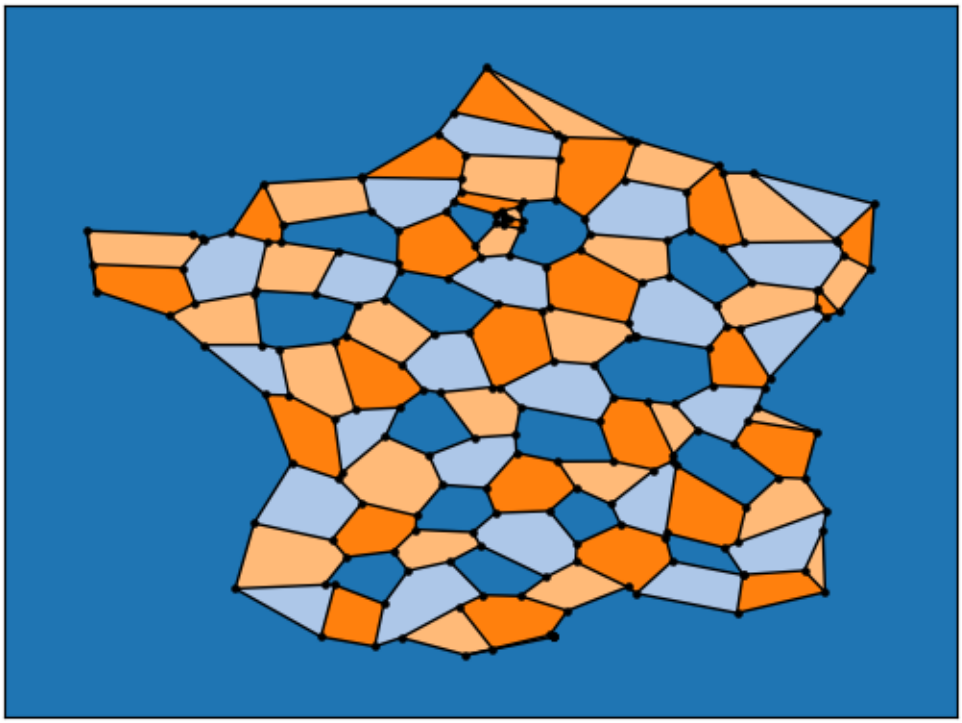




```
Face chromatic number = 3
```

The next example shows how we can use these routines to four-color the departments of France using the data from the file france.txt. Here, each node is assigned the attribute `pos`, so the command `nx.get_node_attributes()` is used to form the positions dictionary.

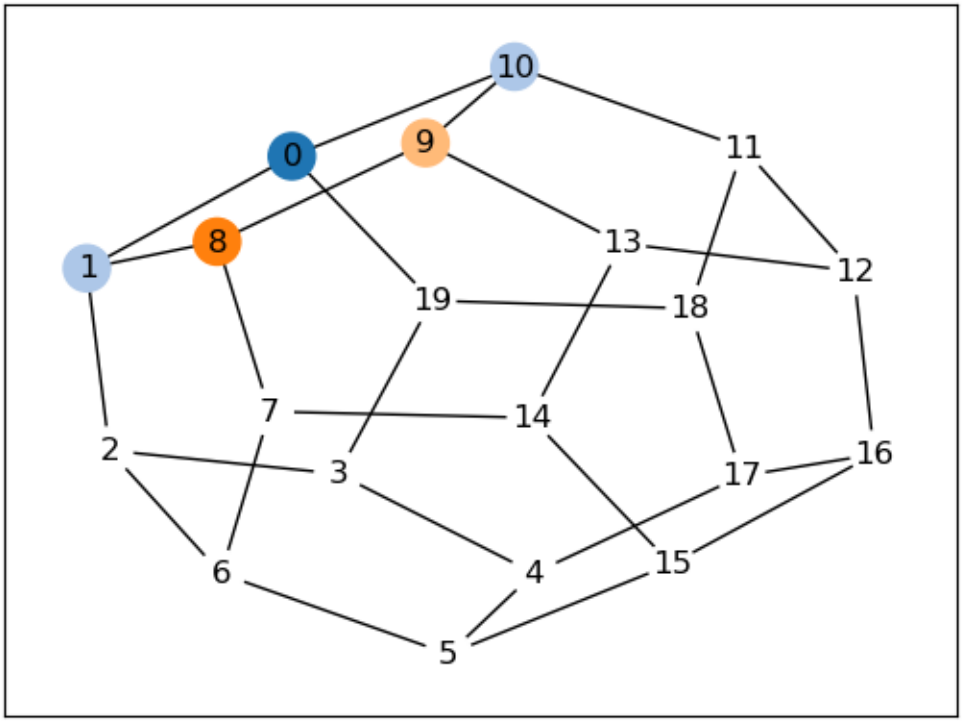
```
G = nx.Graph()
with open("france.txt", "r") as f:
    f.readline()
    n = int(f.readline())
    for i in range(n):
        L = f.readline().split(" ")
        G.add_node(i, pos=(float(L[0]),float(L[1])))
        for j in range(2, len(L)):
            G.add_edge(i, int(L[j]))

pos = nx.get_node_attributes(G, "pos")
c = gcol.face_coloring(G, pos, opt_alg=1)
gcol.draw_face_coloring(c, pos, external=True)
nx.draw_networkx(
    G,
    pos=pos,
    node_color='k',
    node_size=5,
    with_labels=False
)
plt.show()
```

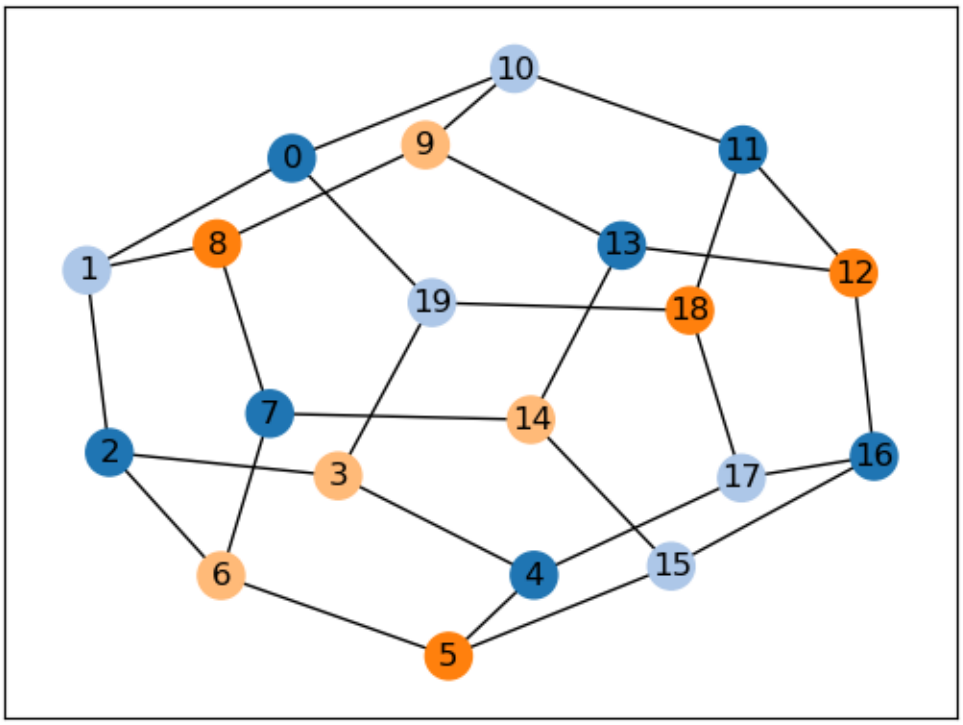


## 2.5 Precoloring

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 following example, 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.

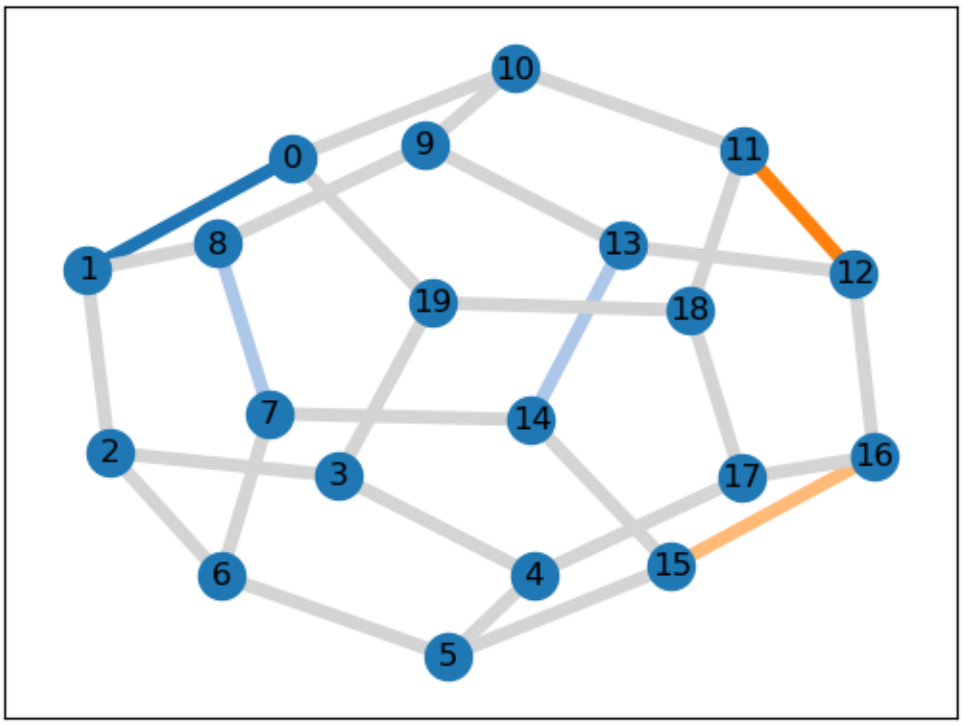
```
G = nx.dodecahedral_graph()
P = {0:0, 1:1, 8:2, 9:3, 10:1}
nx.draw_networkx(
    G,
    pos=nx.spring_layout(G, seed=1),
    node_color=gcol.get_node_colors(G, P)
)
plt.show()

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()
```

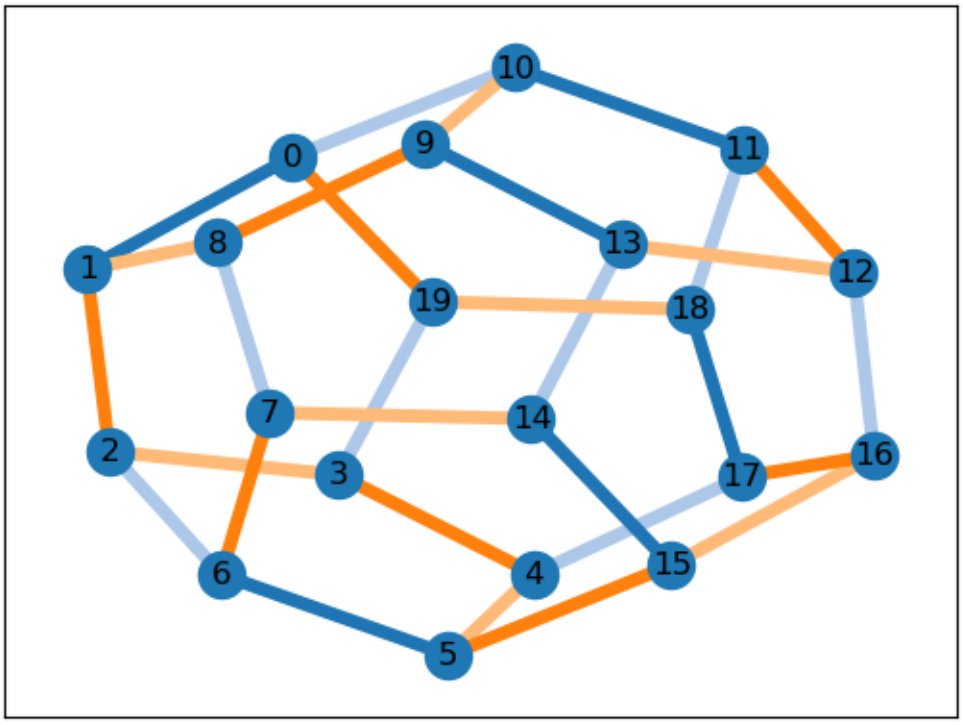





A similar process can also be followed for edge precoloring, which the following example demonstrates.

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

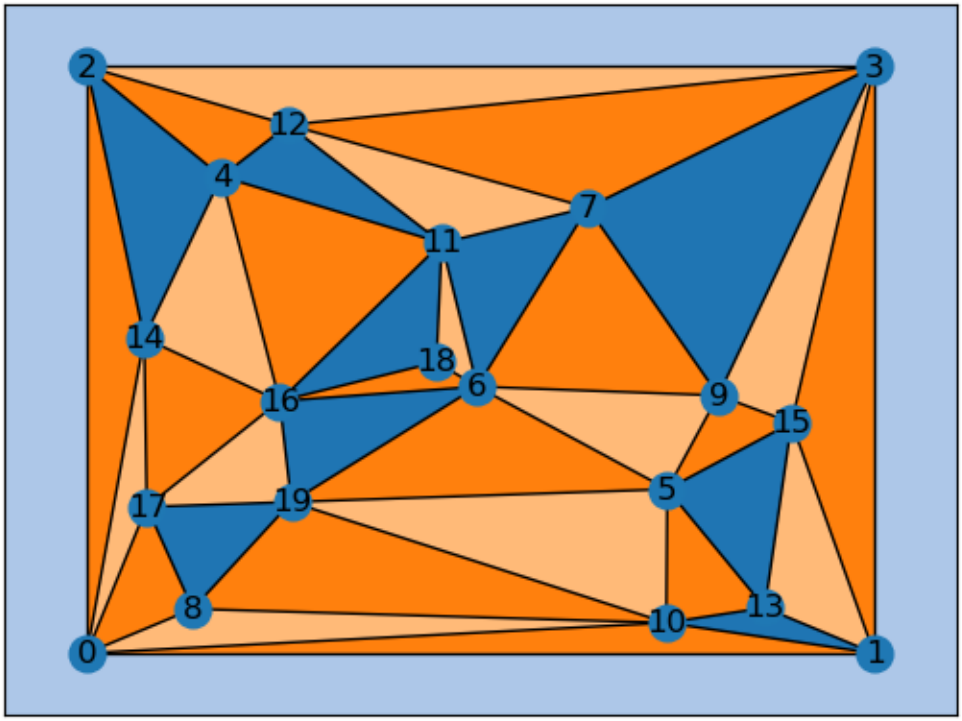



We can also precolor the faces of a planar embedding. To do this we follow the same process as the previous examples, though precolored faces are now identified by their sequences of surrounding nodes. As the example below demonstrates, each internal face in a planar embedding is characterized by the series of nodes that surround it *in a counterclockwise direction*. Similarly, the one external face is identified by the series of nodes traveling in *a clockwise direction*.

The following example creates a 20-node planar graph. It then precolors the external face `(0, 2, 3, 1)` with color 1, the internal face `(6, 9, 7)` with color 2, and the internal face `(16, 17, 19)` with color 3.

```
def make_planar_graph(n, seed=None):
    # Function for making a dense planar graph by placing nodes randomly
    # into the unit square, including corners
    assert n >= 4, "n parameter must be at least 4"
    import random
    from scipy.spatial import Delaunay
    random.seed(seed)
    P = [(0,0), (1,0), (0,1), (1, 1)]
    for i in range(4, n):
        P.append((random.uniform(0.05,0.95), random.uniform(0.05,0.95)))
    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(20, seed=1)
pos = nx.get_node_attributes(G, "pos")
P = {(0, 2, 3, 1): 1, (6, 9, 7): 2, (16, 17, 19): 3}
c = gcol.face_precoloring(G, pos, P, opt_alg=1)
gcol.draw_face_coloring(c, pos, True)
nx.draw_networkx(G, pos=pos, node_size=180, with_labels=True)
plt.show()
```



## 2.6 Solving Sudoku

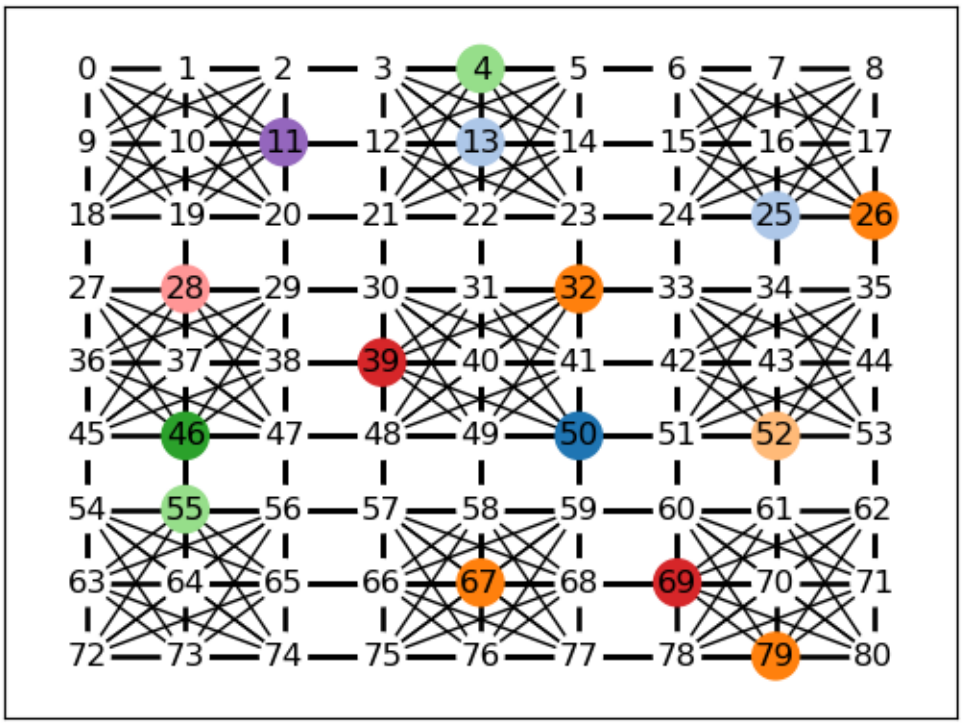
Node precoloring can also be used to solve sudoku puzzles. The objective in sudoku is to fill a $d^2 \times d^2$ grid with digits so that each column, each row, and each of the $d \times d$ boxes contain all digits from 0 to $d^2 - 1$. The puzzle comes with some of the cells filled. The player then needs 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.

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

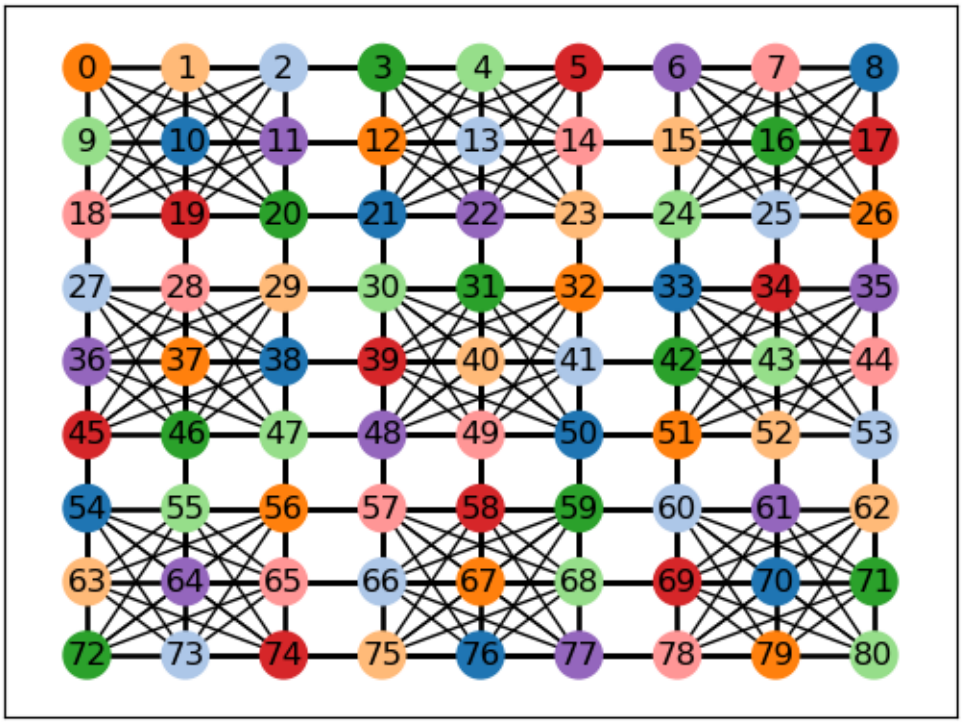
Sudoku puzzles can be solved by first forming a sudoku graph, which uses a node for each cell in the grid. Edges in this graph occur between all pairs of nodes in the same column, row or box. Finally, we use the filled cells in the puzzle to precolor the correct nodes. The puzzle is then solved by coloring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

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
```



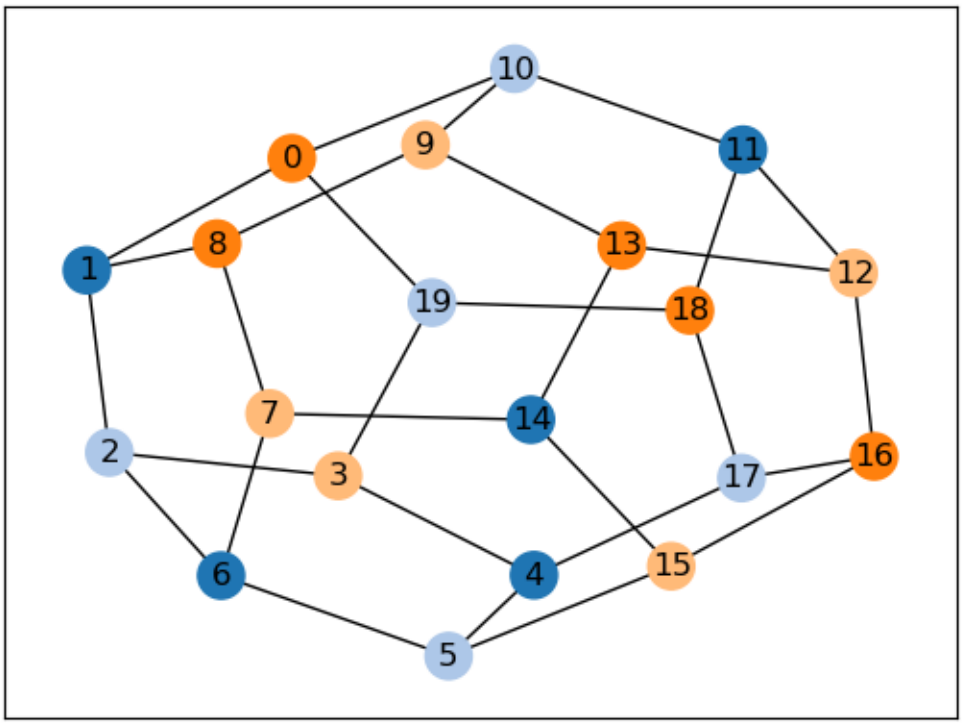
## 2.7 List Coloring

In the node list coloring problem, each node has a list of *allowed colors*. The aim is to color each node with one of its allowed colors while ensuring that adjacent nodes always have different colors.

In the following example, the dictionary `A` defines the allowed colors for each node in the graph `G`. (Node `0`, for example, can only be assigned to colors 1 or 2.). This particular problem instance is solvable, as the output demonstrates.

```
import random
random.seed(1)
G = nx.dodecahedral_graph()
A = {v: random.sample([0, 1, 2 ,3], 2) for v in G}
c = gcol.node_list_colouring(G, allowed_cols=A, opt_alg=1)
print("Allowed colors A =", A)
print("Solution c=", c)
nx.draw_networkx(
    G,
    pos=nx.spring_layout(G, seed=1),
    node_color=gcol.get_node_colors(G, c)
)
```

```
Allowed colors A = {0: [1, 2], 1: [0, 1], 2: [0, 1], 3: [3, 1], 4: [3, 0], 5: [0, 1], 6:␣
↪[0, 1], 7: [3, 2], 8: [0, 2], 9: [3, 1], 10: [1, 2], 11: [0, 1], 12: [0, 3], 13: [0,␣
↪2], 14: [0, 1], 15: [1, 3], 16: [0, 2], 17: [1, 3], 18: [3, 2], 19: [1, 3]}
Solution c= {1: 0, 2: 1, 8: 2, 3: 3, 6: 0, 7: 3, 4: 0, 19: 1, 5: 1, 15: 3, 0: 2, 10: 1,␣
↪11: 0, 12: 3, 9: 3, 14: 0, 13: 2, 17: 1, 16: 2, 18: 2}
```




List coloring can also be performed on the edges and faces of a graph. If the defined problem instance cannot be solved (that is, the lists of allowed colors are too restrictive), then a `ValueError` is raised.

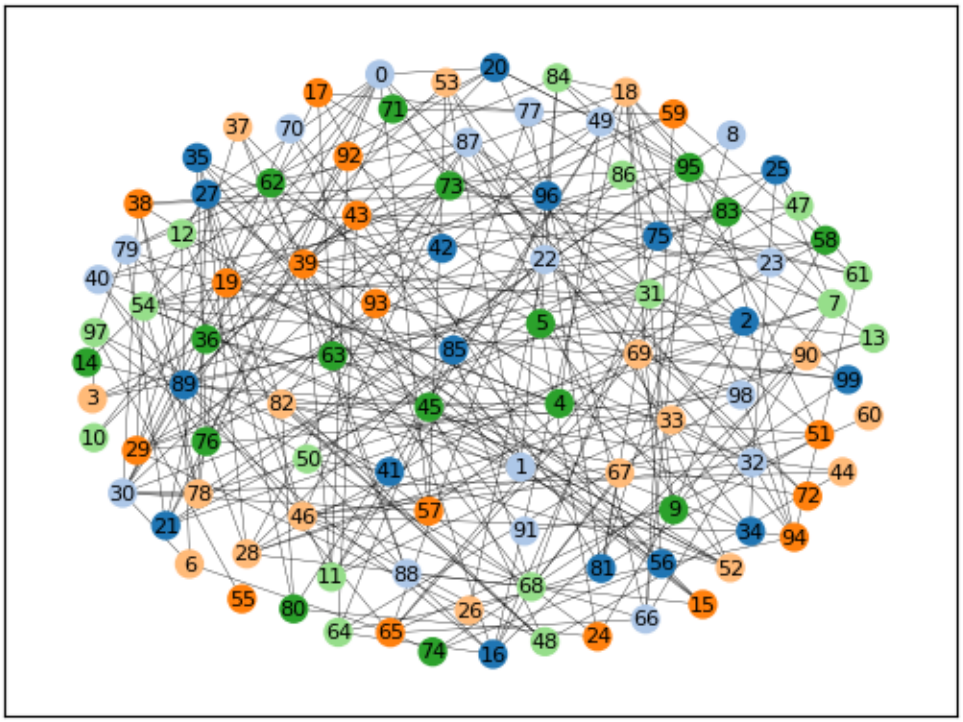
## 2.8 $k$-Coloring

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.

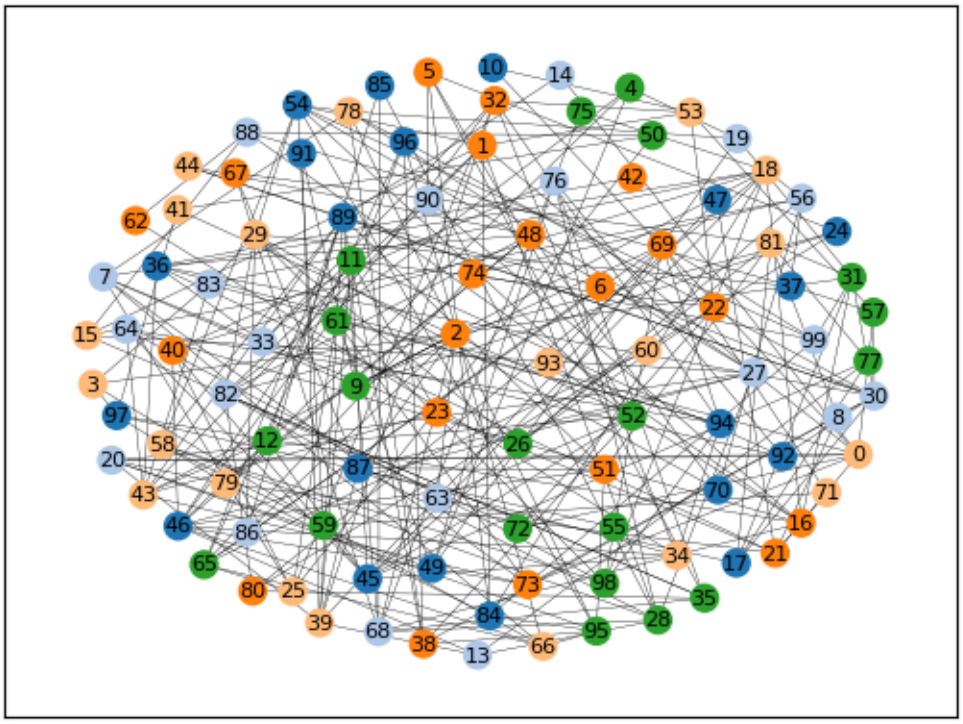
In the following example, we make use of `gcol.node_k_coloring()` method to produce node $k$-colorings of a random $G(n, p)$ graph. These graphs are generated by taking $n$ nodes and then adding an edge between each pair of nodes at random with probability $p$. Here we use $n = 100$ and $p = 0.05$ with $k = 6$, 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)

for k in [6, 5, 4]:
    c = gcol.node_k_coloring(G, k, opt_alg=2, it_limit=1000)
    print("Here is a node", k, "-coloring of G:")
    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()
```

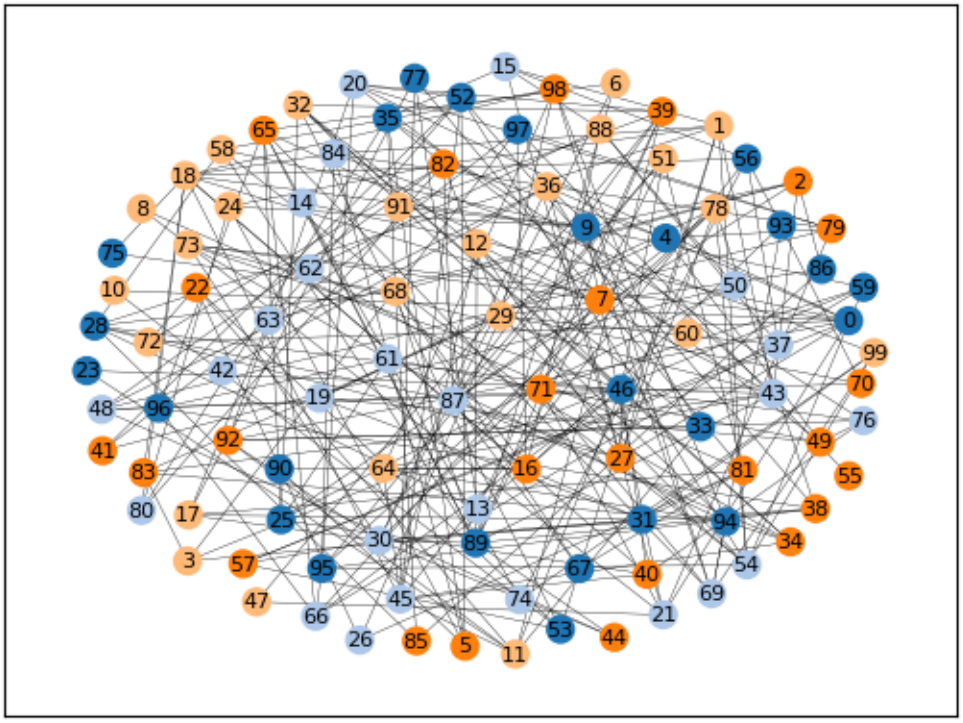
```
Here is a node 6 -coloring of G:
```



```
Here is a node 5 -coloring of G:
```



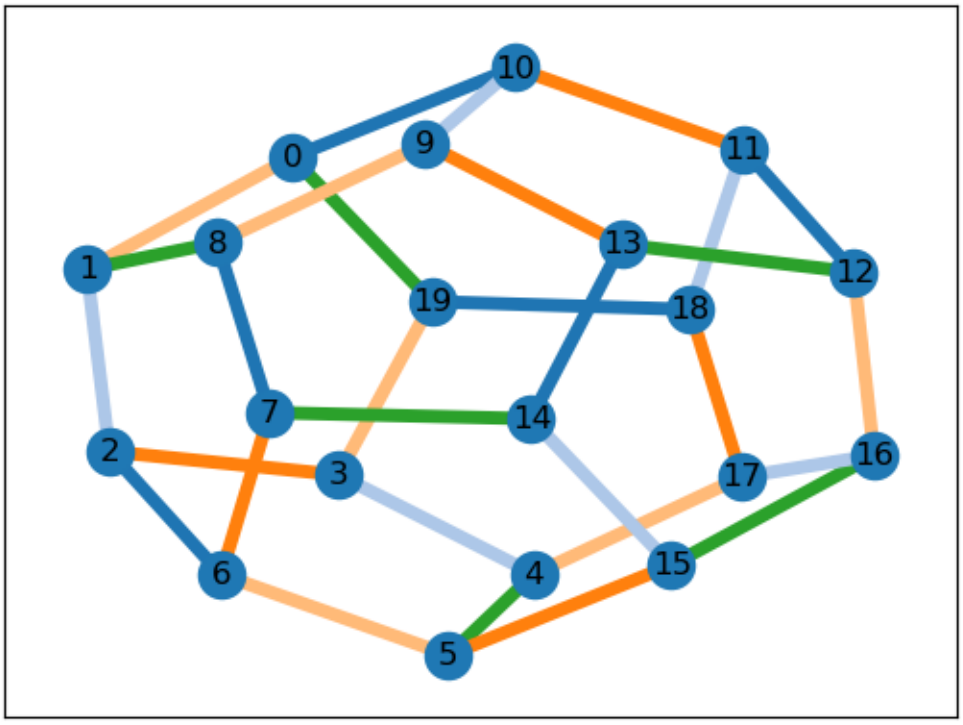

```
Here is a node 4 -coloring of G:
```



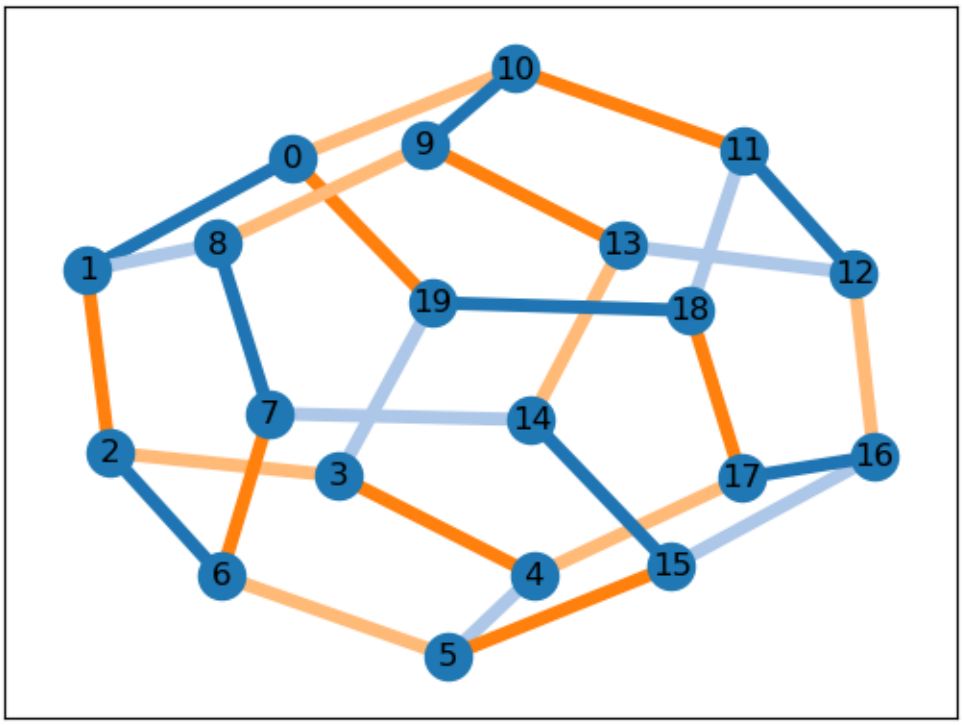

The following shows a similar process for edge $k$-coloring.

```
G = nx.dodecahedral_graph()

for k in [5, 4, 3]:
    c = gcol.edge_k_coloring(G, k)
    print("Here is an edge", k, "-coloring of G:")
    nx.draw_networkx(
        G,
        pos=nx.spring_layout(G, seed=1),
        edge_color=gcol.get_edge_colors(G, c),
        width=5
    )
    plt.show()
```

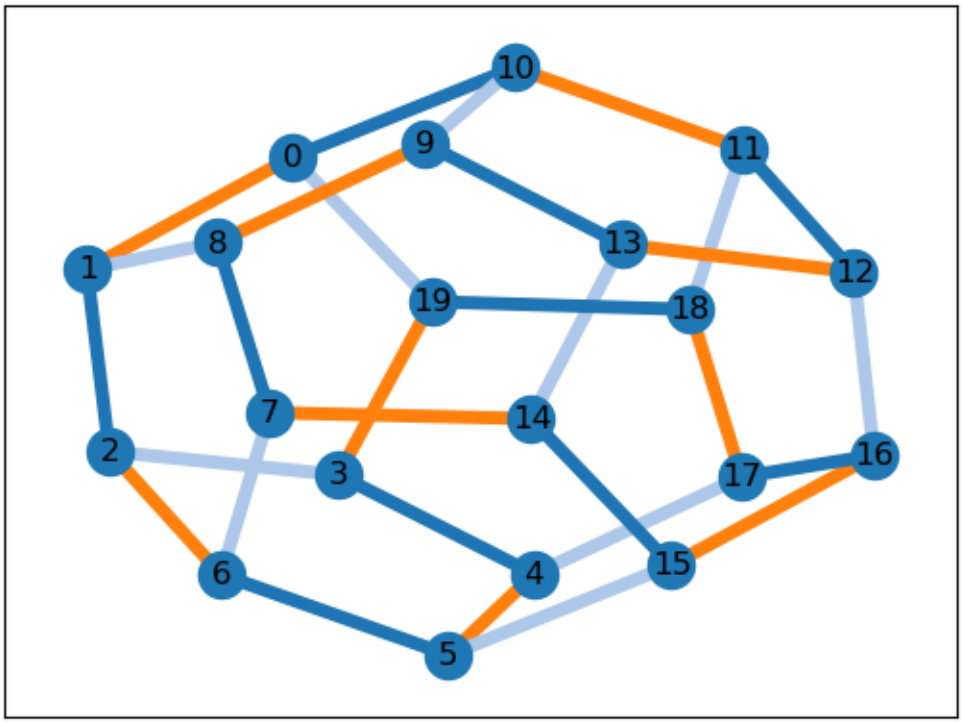
```
Here is an edge 5 -coloring of G:
```




```
Here is an edge 4 -coloring of G:
```




```
Here is an edge 3 -coloring of G:
```



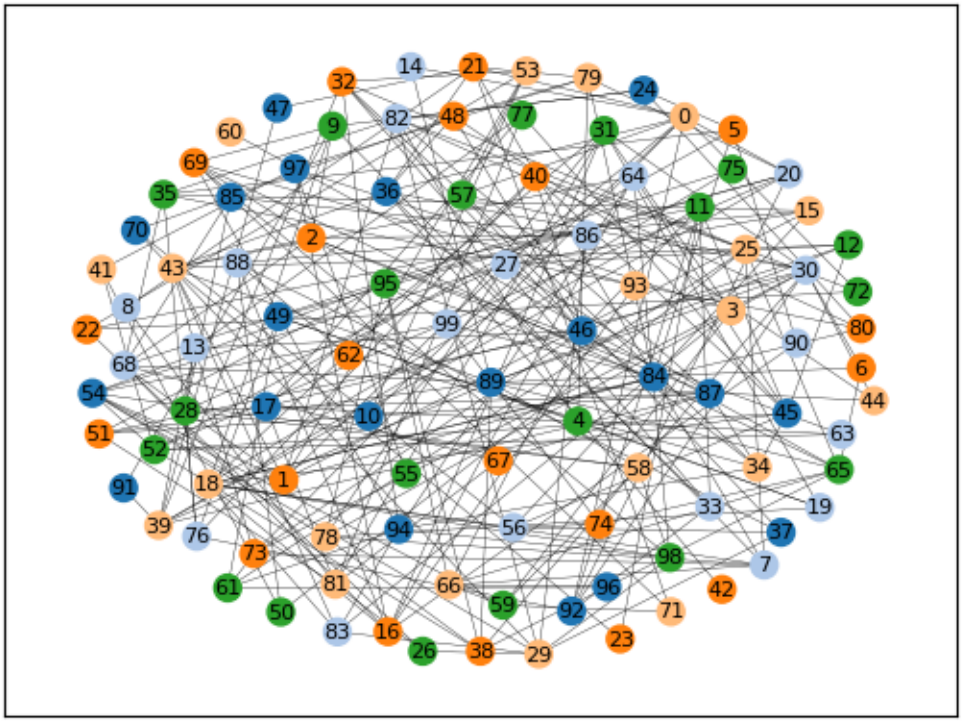
### 2.8.1 Equitable $k$-coloring

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)
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")
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 an equitable node-5-coloring of G,
Largest color class has 20 nodes
Smallest color class has 20 nodes
```

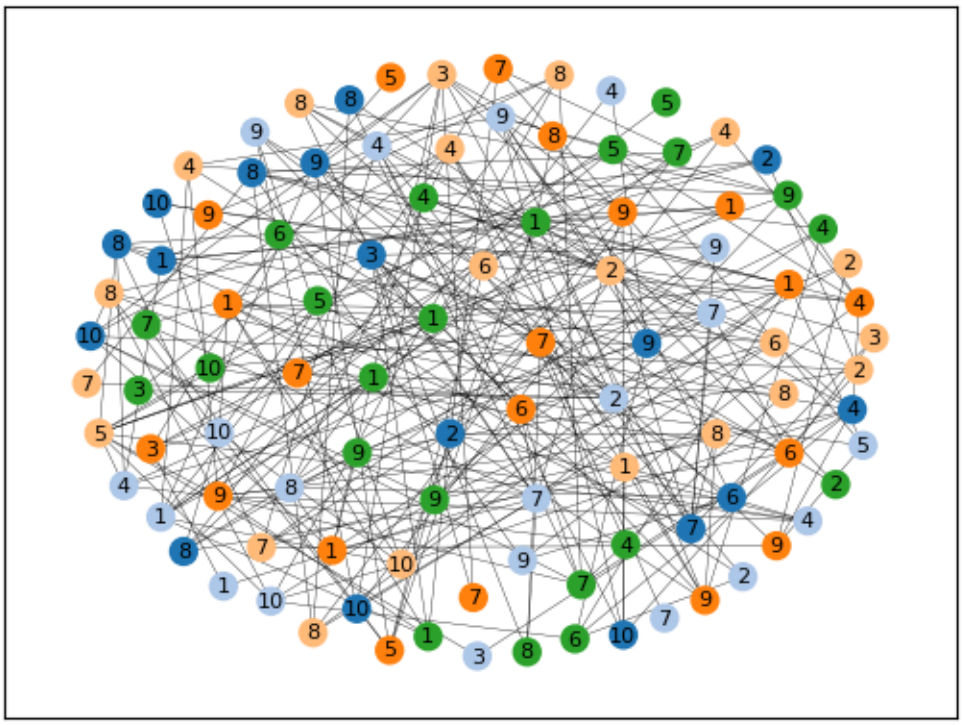



The following example also determines an equitable node 5-coloring for a random $G(100, 0.05)$ graph. However, in

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)

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

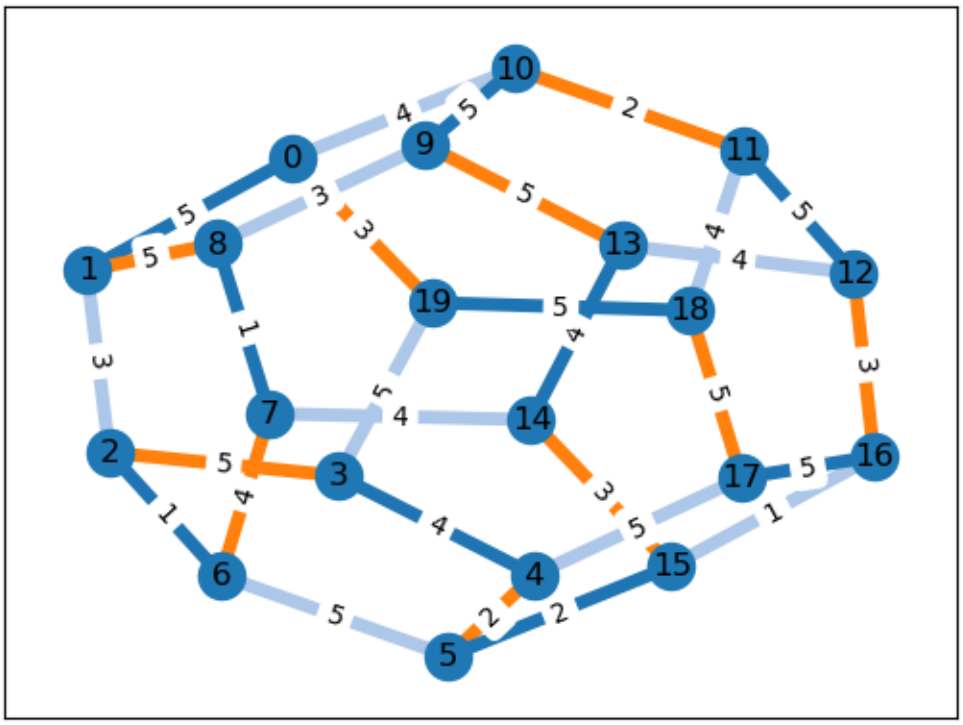
```
Here is an equitable node 5-coloring of the node-weighted graph G:
Weight of color class 0 = 115 [1, 2, 2, 3, 4, 6, 7, 8, 8, 8, 8, 9, 9, 10, 10, 10, 10]
Weight of color class 1 = 115 [1, 1, 2, 2, 3, 4, 4, 4, 4, 5, 7, 7, 7, 8, 9, 9, 9, 9, 10,␣
↪10]
Weight of color class 2 = 114 [1, 1, 1, 1, 3, 4, 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 using an edge-weighted graph.

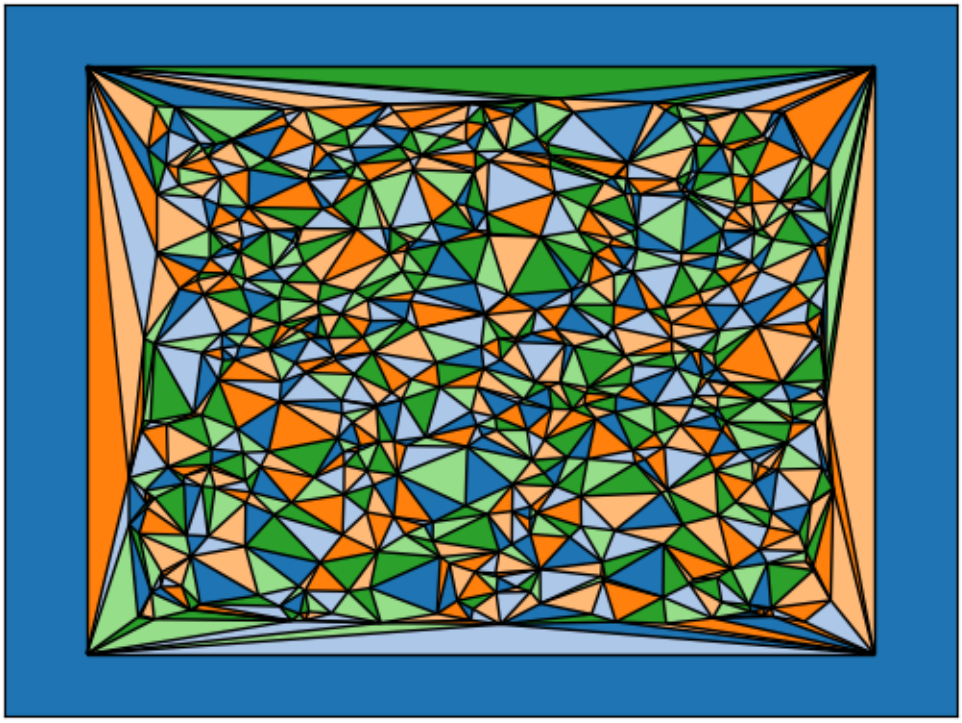
```
G = nx.dodecahedral_graph()
for u, v in G.edges():
    G.add_edge(u, v, edgeweight=random.randint(1,5))

c = gcol.equitable_edge_k_coloring(G, 3, weight="edgeweight", opt_alg=2, it_limit=1000)
print("Here is an equitable edge-3-coloring of the edge-weighted graph G,")
P = gcol.partition(c)
for j in range(len(P)):
    Wj = sorted([G.edges[e]["edgeweight"] for e in P[j]])
    print("Weight of color class", j, "=", sum(Wj), Wj)

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()
```

```
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]
```




The next example shows an equitable face coloring using 6 colors.

```
G = make_planar_graph(500, seed=1)
pos = nx.get_node_attributes(G, "pos")
c = gcol.equitable_face_k_coloring(G, pos, k=6)
gcol.draw_face_coloring(c, pos, True)
nx.draw_networkx(G, pos=pos, node_size=0, with_labels=False)
plt.show()
```



### 2.8.2 Minimum Cost $k$-Coloring

Sometimes we will be 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.

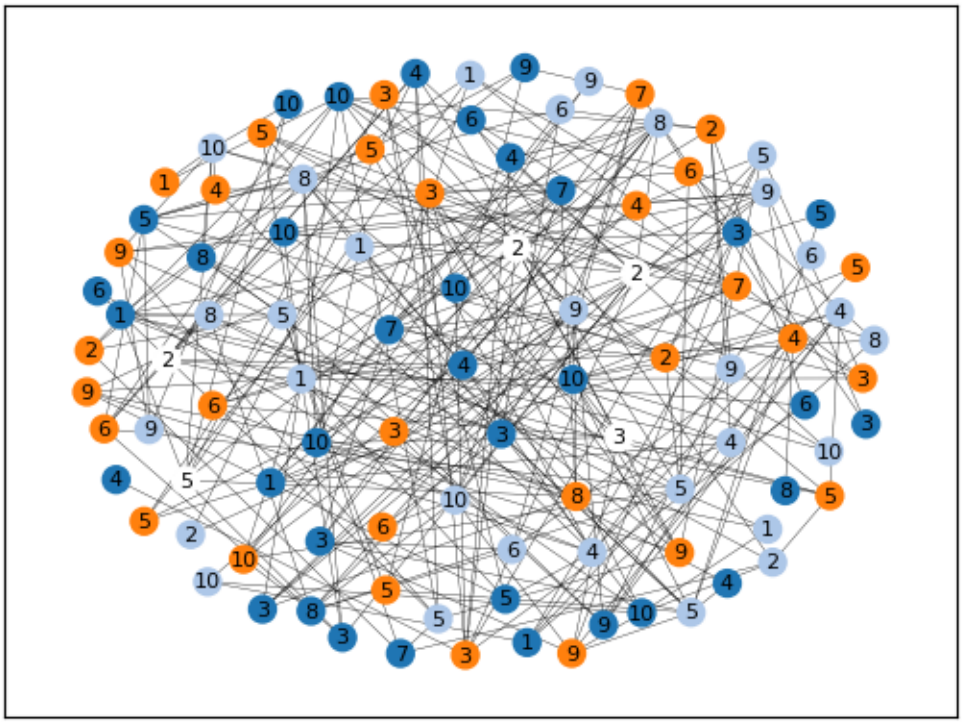
The following example creates a node-weighted random graph and then produces a node 3-coloring solution using the routine `gcol.min_cost_k_coloring()`. This solution has five uncolored nodes (shown in white) with a total weight of 14. In the visualization, node labels refer to node weights.

```
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)
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
```

(continues on next page)

(continued from previous page)

```
)
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 [2, 2, 2, 3, 5] giving a total cost = 14
```

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.

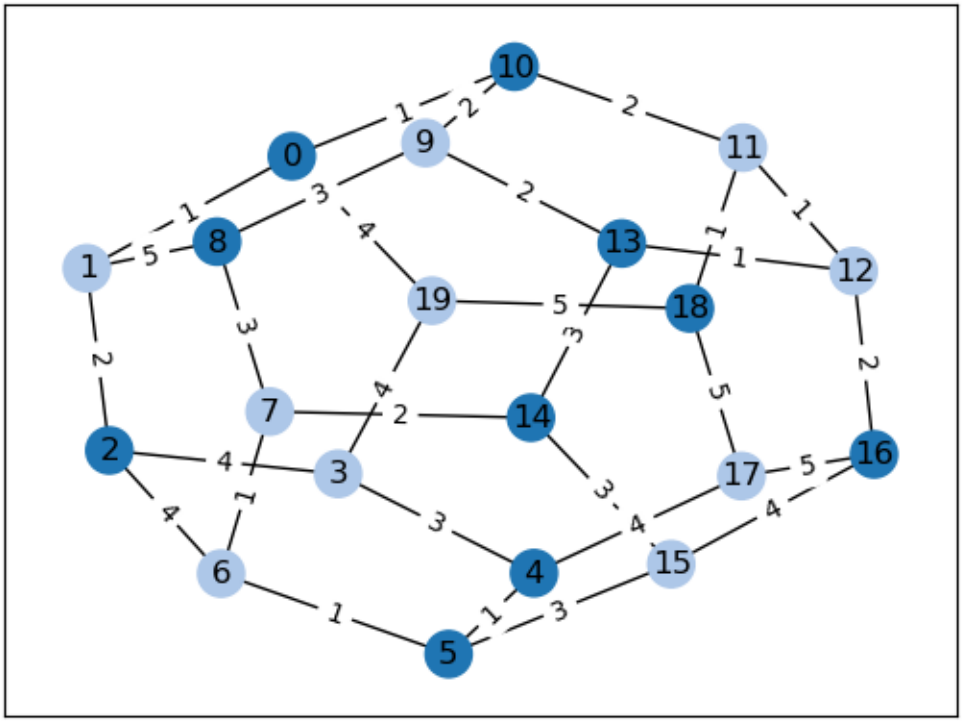
The following example creates a small edge-weighted graph and then produces a node 2-coloring using the routine `gcol.min_cost_k_coloring()`. Six of the edges are causing a clash, giving a total weight of 11.

```
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
)
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')
```

(continues on next page)

(continued from previous page)

```
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 [(0, 10), (3, 19), (4, 5), (6, 7), (11, 12), (13,
↪ 14)] giving a total cost of 11
```

## 2.9 Kempe Chains and $s$-Chains

Given a node coloring of a graph, a Kempe chain is a connected component in the graph induced by nodes of two different colors, $i$ and $j$. Given a proper node coloring (that is, a coloring where no adjacent nodes have the same color), interchanging the colors of the nodes in a Kempe chain creates a new, proper coloring.

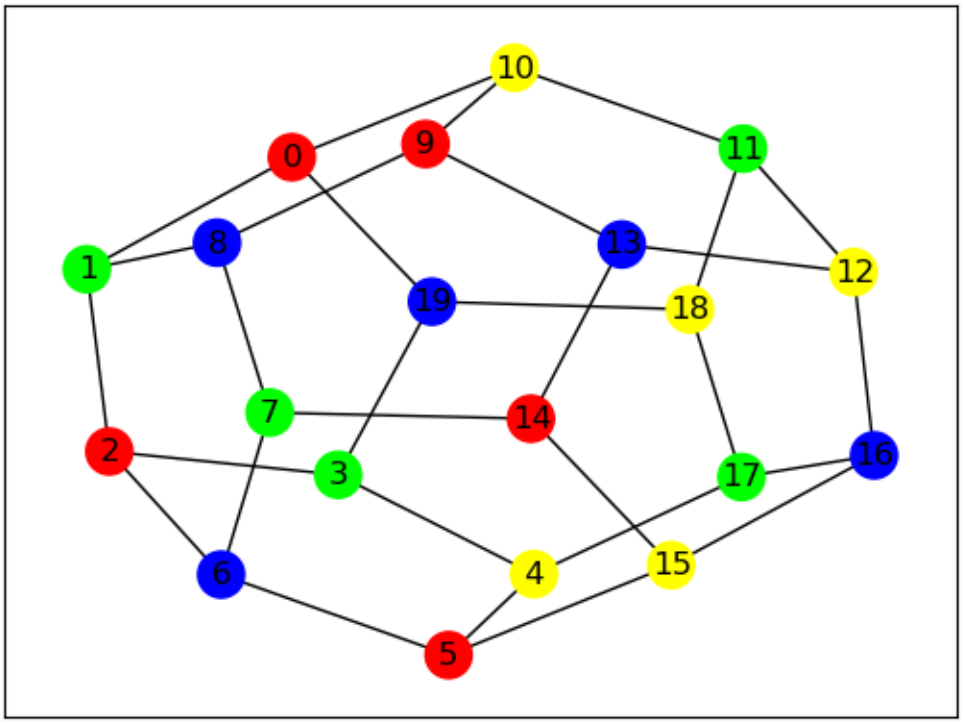
The following example takes a coloring `c` of a graph `G` and determines a red-yellow Kempe chain from node 5 (which is red). The nodes in the resultant Kempe chain are put in the set `C`. A Kempe chain interchange is then performed, which swaps the colors of all nodes in `C`, leading to the second solution shown.

```
G = nx.dodecahedral_graph()
c = gcol.node_k_coloring(G, 4)
nx.draw_networkx(
    G,
    pos=nx.spring_layout(G, seed=1),
    node_color=gcol.get_node_colors(G, c, gcol.colorful)
)
```

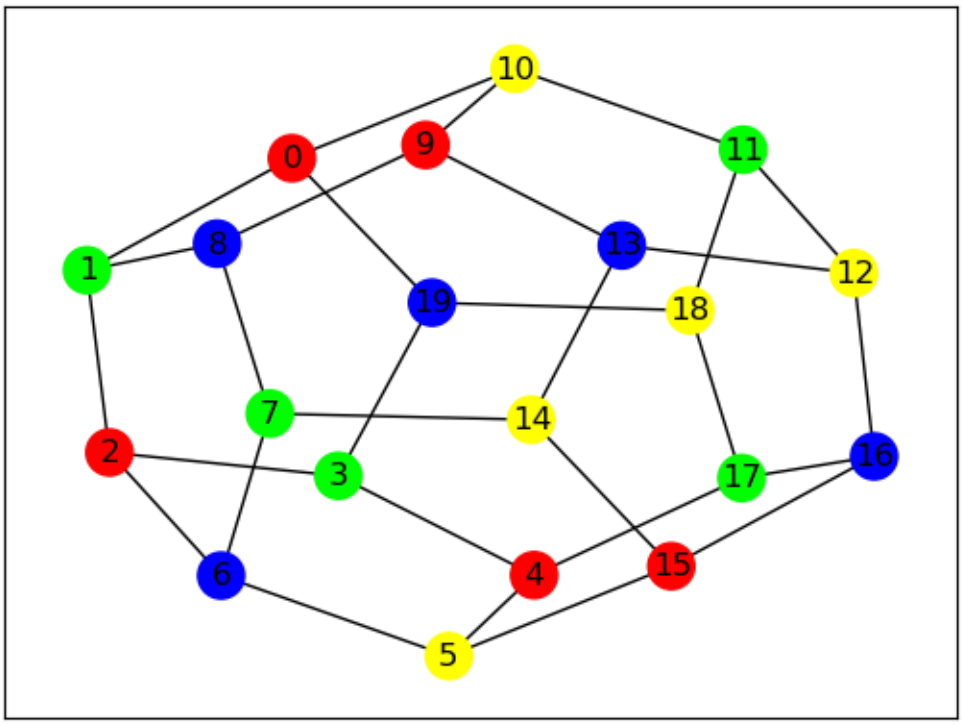
(continues on next page)

(continued from previous page)

```
plt.show()

i, j = 0, 3  # red, yellow
C = gcol.kempe_chain(G, c, 5, i, j)
print("Above, the red-yellow Kempe chain from node 5 has nodes:", C)
print("Interchanging the colors in this chain gives:")
for u in C:
    if c[u] == i:
        c[u] = j
    else:
        c[u] = i
nx.draw_networkx(
    G,
    pos=nx.spring_layout(G, seed=1),
    node_color=gcol.get_node_colors(G, c, gcol.colorful)
)
plt.show()
```




```
Above, the red-yellow Kempe chain from node 5 has nodes: {4, 5, 14, 15}
Interchanging the colors in this chain gives:
```




An $s$-chain is a generalization of a Kempe chain that allows more than two colors. Given a proper node coloring of a graph $G = (V, E)$, an $s$-chain is defined by a prescribed node $v \in V$ and sequence of unique colors $j_0, j_1, \ldots, j_{s-1}$, where the current color of $v$ is $j_0$. The result is the set of nodes that are reachable from $v$ in the digraph $G' = (V', A)$ in which

- $V' = \{u \ : \ u \in V \ \wedge \ c(u) \in \{j_0, j_1, \ldots, j_{s-1}\}\}$, and
- $A = \{(u, w) \ : \ \{u, w\} \in E \ \wedge \ c(u) = j_i \ \wedge \ c(w) = j_{(i+1) \bmod s}\}$,

where $c(u)$ gives the color of node $u$. In a proper coloring, interchanging the colors of all nodes in an $s$-chain via the following mapping

- $j_i \leftarrow j_{(i+1) \bmod s}$

results in a new proper coloring.

The following code shows an example of this. Here, an $s$-chain is generated in the coloring using node 4 and colors 0, 1, and 3 (red, green, and yellow respectively). The described color mapping is then used to interchange the colors on this $s$-chain, giving the coloring shown below.

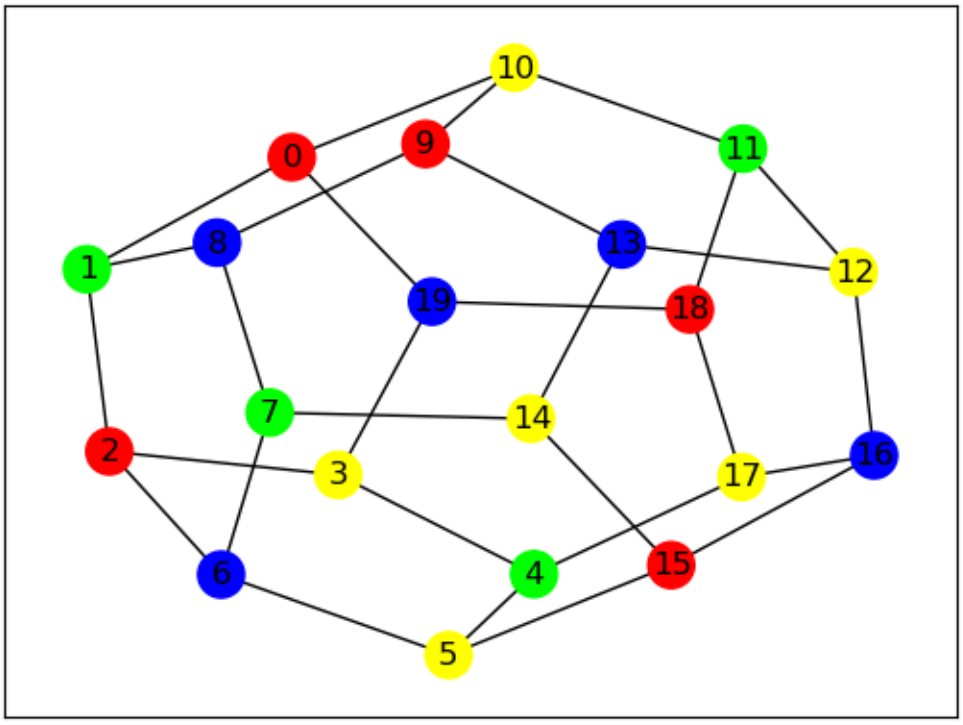
```
L = [0, 1, 3]  # red, green, yellow
C = gcol.s_chain(G, c, 4, L)
print("The s-chain for node 4 with color sequence", L, "is", C)

color_map = {L[j]: L[(j+1) % len(L)] for j in range(len(L))}
for u in C:
    c[u] = color_map[c[u]]

print("Interchanging the colors on this chain using", color_map, "gives:")
nx.draw_networkx(
    G,
    pos=nx.spring_layout(G, seed=1),
```

(continues on next page)

(continued from previous page)

```
    node_color=gcol.get_node_colors(G, c, gcol.colorful)
)
plt.show()
```

```
The s-chain for node 4 with color sequence [0, 1, 3] is {17, 18, 3, 4}
Interchanging the colors on this chain using {0: 1, 1: 3, 3: 0} gives:
```




## 2.10 Independent Sets, Cliques and Coverings

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

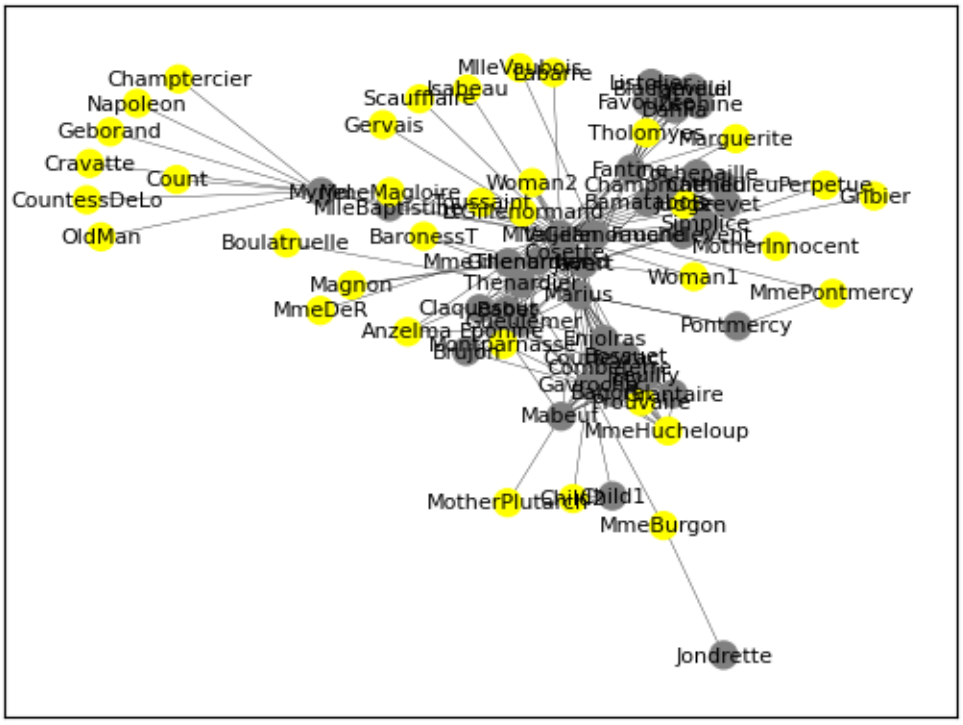
- The *maximum independent set problem* involves determining the largest subset of nodes in a graph $G$ such that none of the nodes in this set are neighboring. The size of the largest independent set in $G$ is known as the *independence number*, denoted by $\alpha(G)$.
- The *minimum node cover problem* 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 by the `gcol.max_independent_set()` method. Here, we use a graph in which nodes represent 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()
S = gcol.max_independent_set(G, it_limit=10000)
print("In Les Miserables, there is 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,
    width=0.25
)
plt.show()
```

```
In Les Miserables, there is 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.

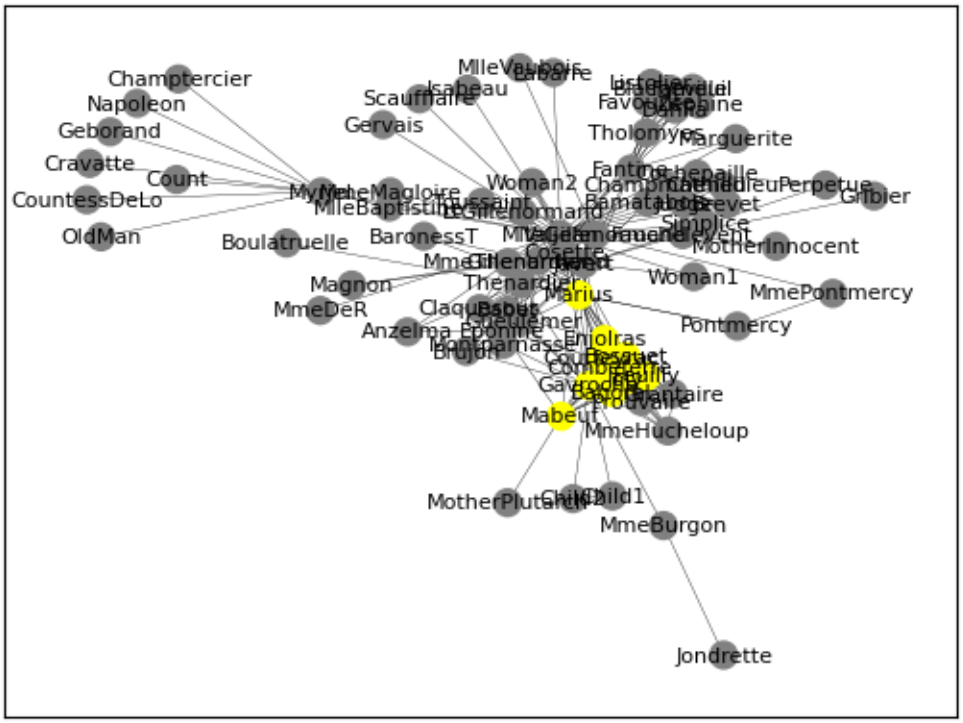
```
S = gcol.max_independent_set(nx.complement(G), it_limit=10000)
print("In the set of", len(G), "Les Miserables characters, there is a subset of",
      len(S), "characters who form a clique. These are", S)
nx.draw_networkx(
    G,
    nx.spring_layout(G, seed=1),
    node_color=gcol.get_set_colors(G, S),
```

(continues on next page)

(continued from previous page)

```
    node_size=100,
    font_size=8,
    width=0.25
)
plt.show()
```

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



CHAPTER

# THREE

# NODE PLACEMENT AND COLOR PALETTES

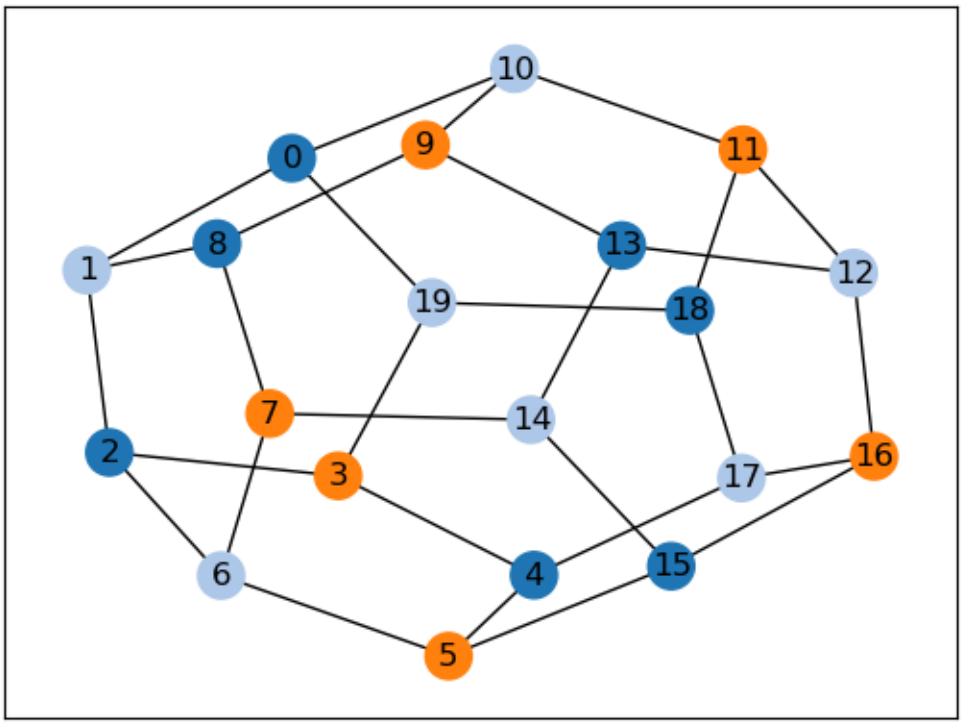
In this chapter, we demonstrate ways in which graph coloring solutions can be visualized using tools from the `networkx` and `gcol` libraries. Further information on the former can be found in its online documentation.

## 3.1 Node Placement

Node placement is an important aspect of network visualization because it shapes how structure and meaning are perceived. Good layouts can reveal patterns such as communities and hierarchies, making relationships easier to observe. Poor placement, by contrast, can obscure connectivity and suggest structure that is not actually present. Effective node placement schemes therefore seek to reduce visual clutter, highlight relevant features, and align the geometry of the drawing with the underlying topology of the network.

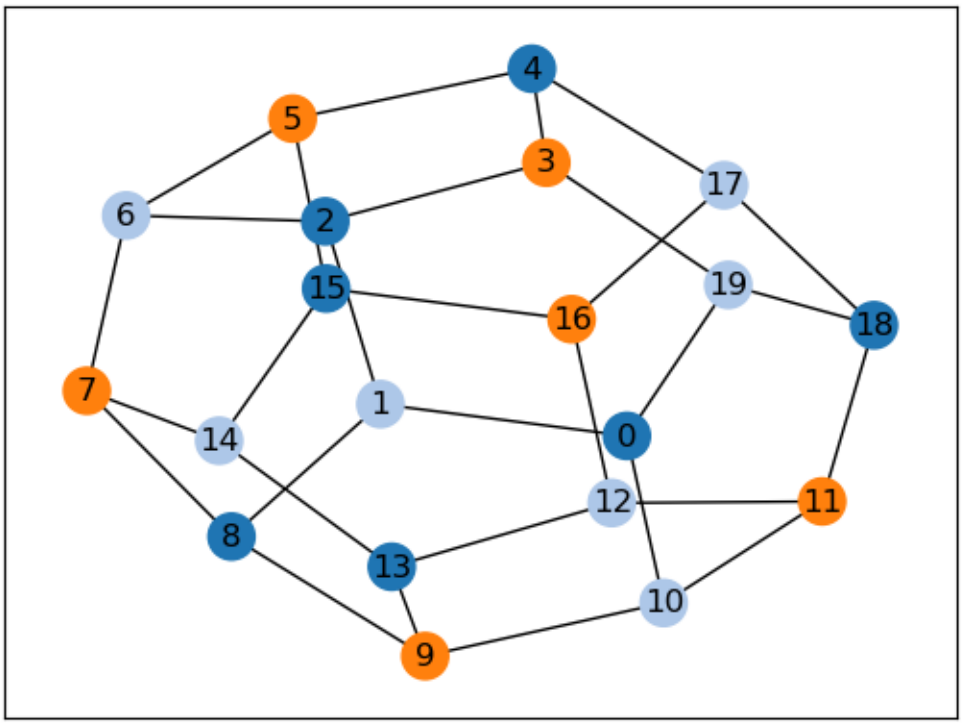
The following code generates and shows a coloring of a dodecahedron graph. This was also seen in the previous chapter. In the figure, the node positions are determined using the method `nx.spring_layout()`, which is provided by `networkx`. This is an example of a force-directed method, which models nodes as mutually repelling elements and edges as springs. The method iteratively adjusts the node' positions to minimize an energy function, balancing the attracting forces of edges and the repulsive forces from nodes. The aim is to create an aesthetically pleasing layout where groups of related nodes are close, unrelated nodes are separated, and few edges intersect.

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



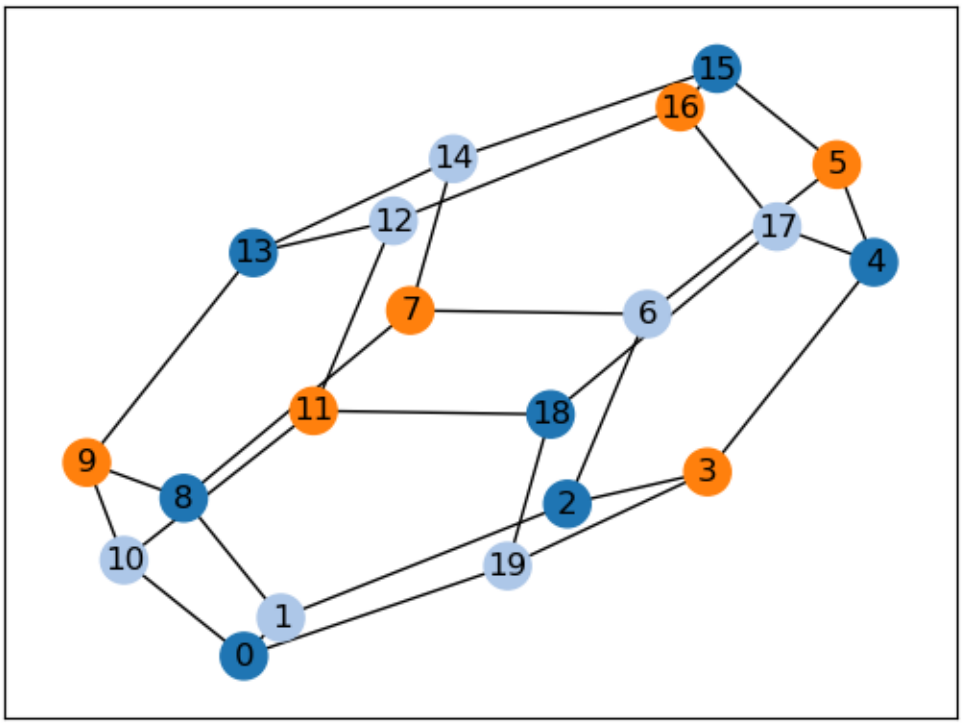

Different layouts can be determined with the `nx.spring_layout()` method by simply changing the seed parameter. Note that, although the look different, these colorings are equivalent.

```
nx.draw_networkx(
    G,
    pos=nx.spring_layout(G, seed=2),
    node_color=gcol.get_node_colors(G, c)
)
plt.show()
```



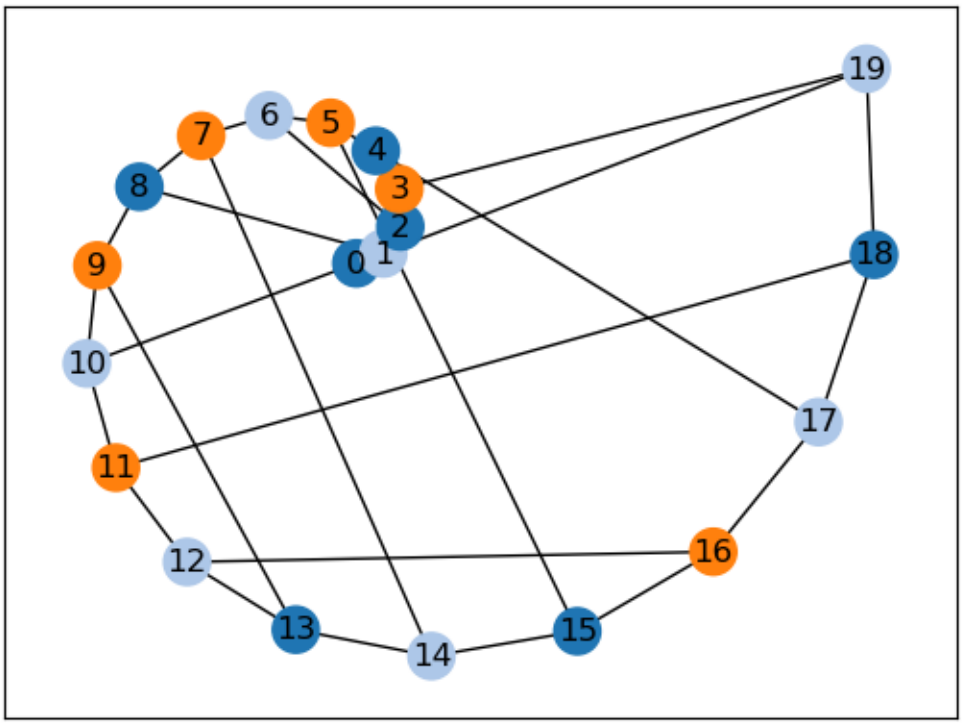

Other layout methods are also available in `networkx`, such as `nx.spectral_layout()`,

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



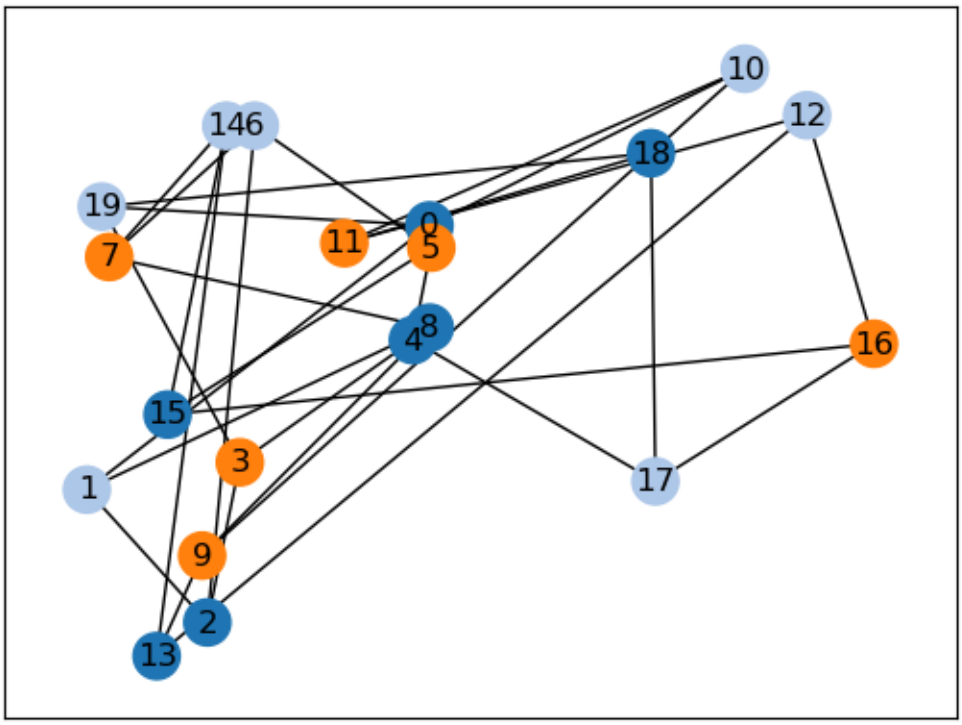

and `nx.sprial_layout()`.

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



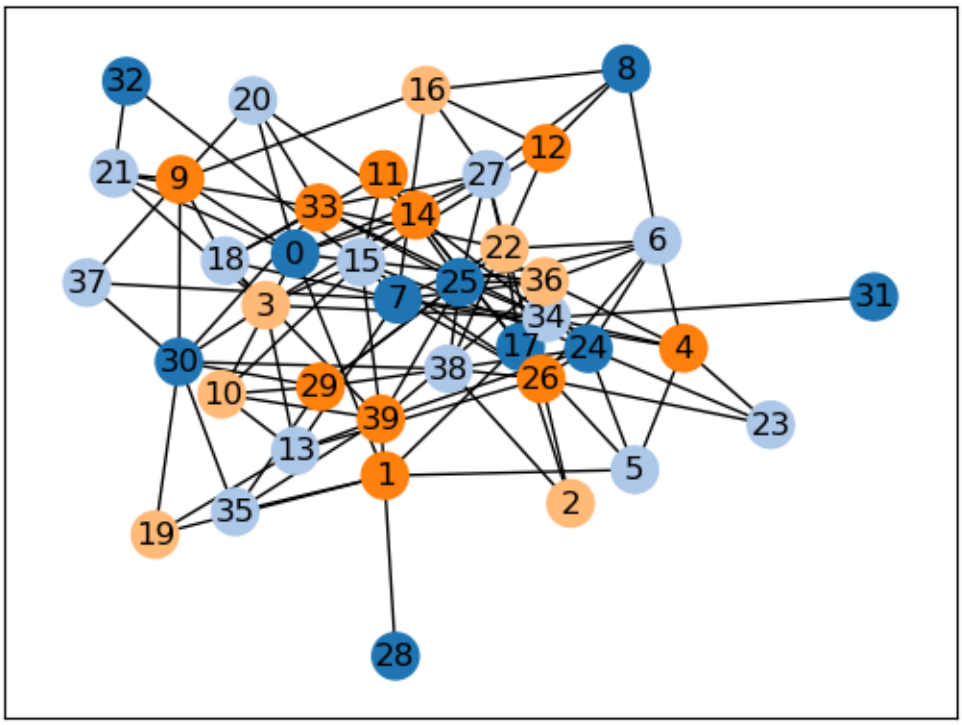

We can also simply place the nodes in random positions, though this may lead to unclear visualizations.

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



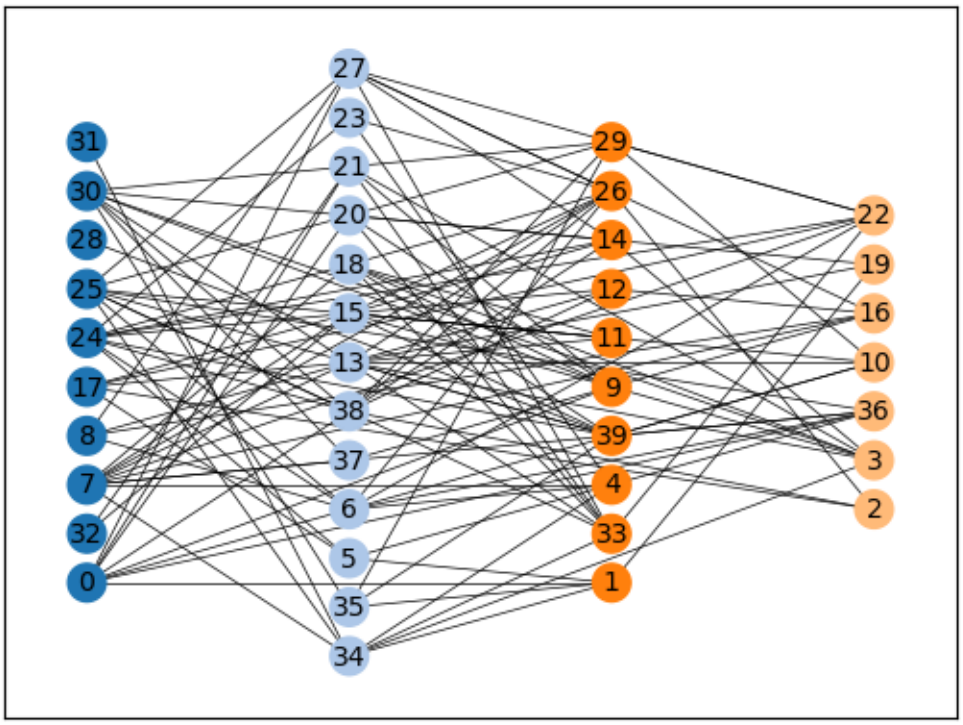

The visualization of graphs becomes more difficult for higher numbers of nodes and edges, and when the graph features few obvious structural patterns. The following example illustrates this using a randomly generated Erdos-Renyi graph $G(50, 0.15)$.

```
G = nx.gnp_random_graph(40, 0.15, seed=1)
c = gcol.node_coloring(G)
nx.draw_networkx(
    G,
    pos=nx.spring_layout(G, seed=1),
    node_color=gcol.get_node_colors(G, c)
)
plt.show()
```




In the above, the `nx.spring_layout()` method has given a rather cluttered layout. Also, the colors of the nodes have not been considered when determining their positions. Consequently, two alternatives are offered by the `gcol` library. The first of these, `gcol.multipartite_layout()` puts the nodes of each color into columns. The same solution as above is now shown below.

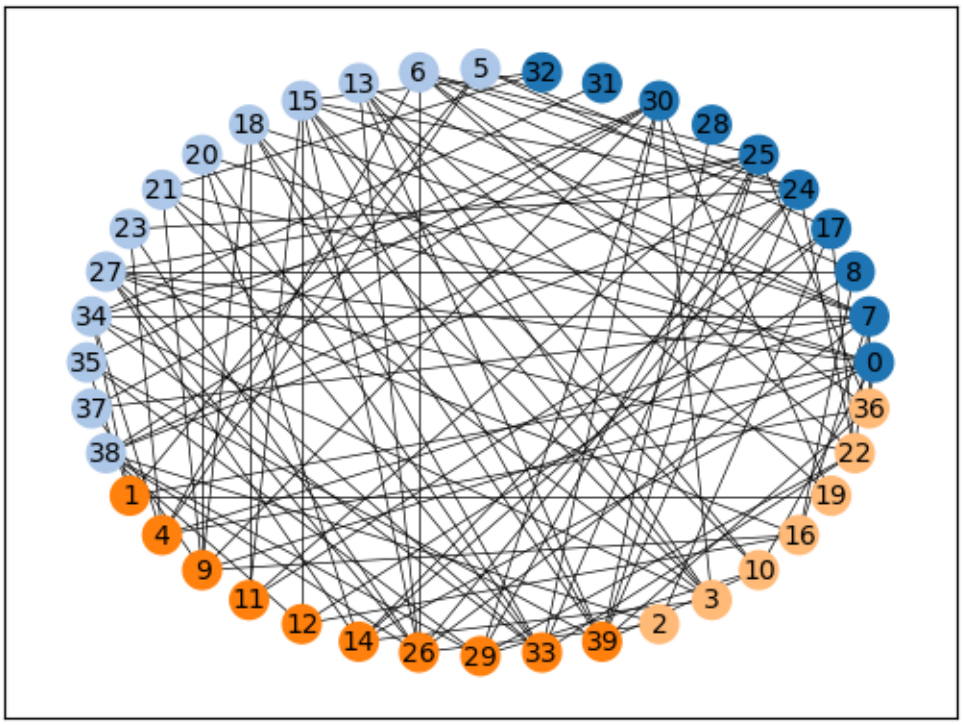
```
nx.draw_networkx(
    G,
    pos=gcol.multipartite_layout(G, c),
    node_color=gcol.get_node_colors(G, c),
    width=0.5,
    node_size=200,
    font_size=10
)
plt.show()
```




Note that we cannot have vertical edges when using this method of visualization. It is also clear that each column of nodes corresponds to an independent set.

The second option is the `gcol.coloring_layout()` method. This places all nodes on the circumference of a circle, with nodes of the same color in adjacent positions.

```
nx.draw_networkx(
    G,
    pos=gcol.coloring_layout(G, c),
    node_color=gcol.get_node_colors(G, c),
    width=0.5,
    node_size=200,
    font_size=10
)
plt.show()
```




## 3.2 Color Palettes

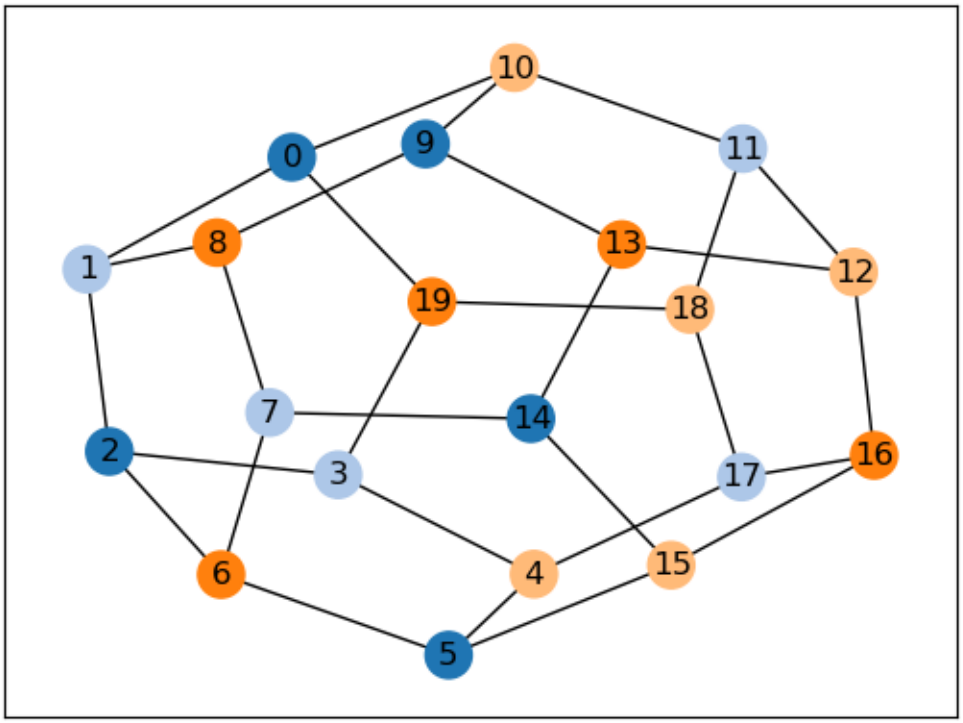
So far, the colors used to display the above solutions have been taken from the in-built default color palette `gcol.tableau`, which maps the integers $0, 1, 2, \ldots$ to RGB triplets. This palette is a collection of 21 colors, provided by Tableau, that are intended to be aesthetically pleasing and easy on the eye. However, other options are available in the `gcol` library: `gcol.colorful` gives a collection of 57 bright colors that are chosen to contrast each other as much as possible; `gcol.colorblind` gives eleven colors (also provided by Tableau) that are intended to be easily distinguishable by those with colorblindness. A demonstration of these palettes is now given.

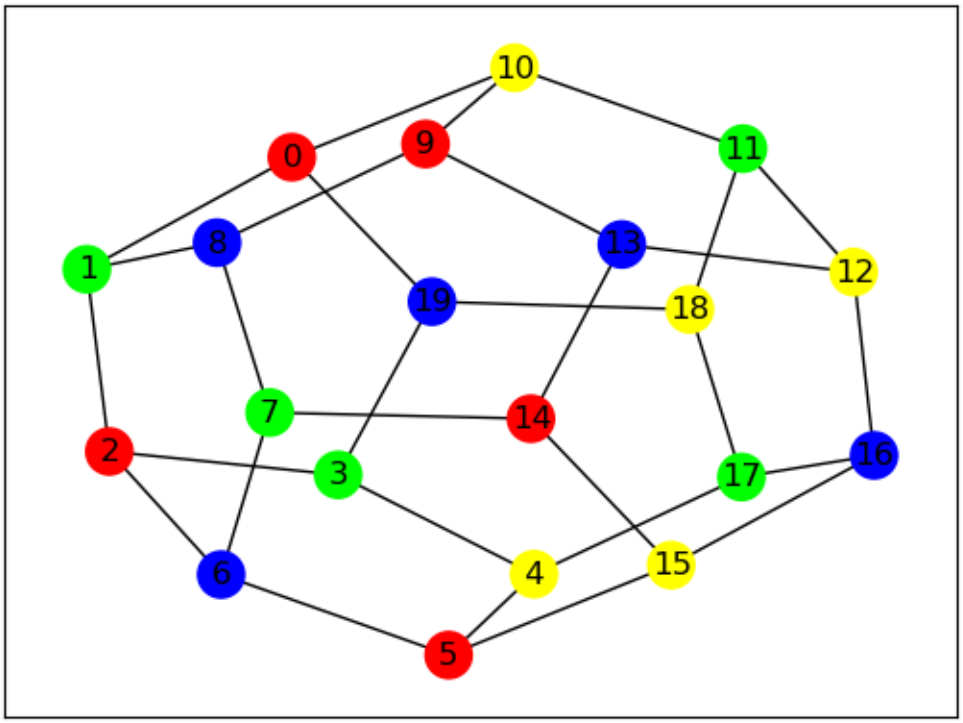
```
G = nx.dodecahedral_graph()
c = gcol.node_k_coloring(G, 4)
nx.draw_networkx(
    G,
    pos=nx.spring_layout(G, seed=1),
    node_color=gcol.get_node_colors(G, c, palette=gcol.tableau)
)
plt.show()

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

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

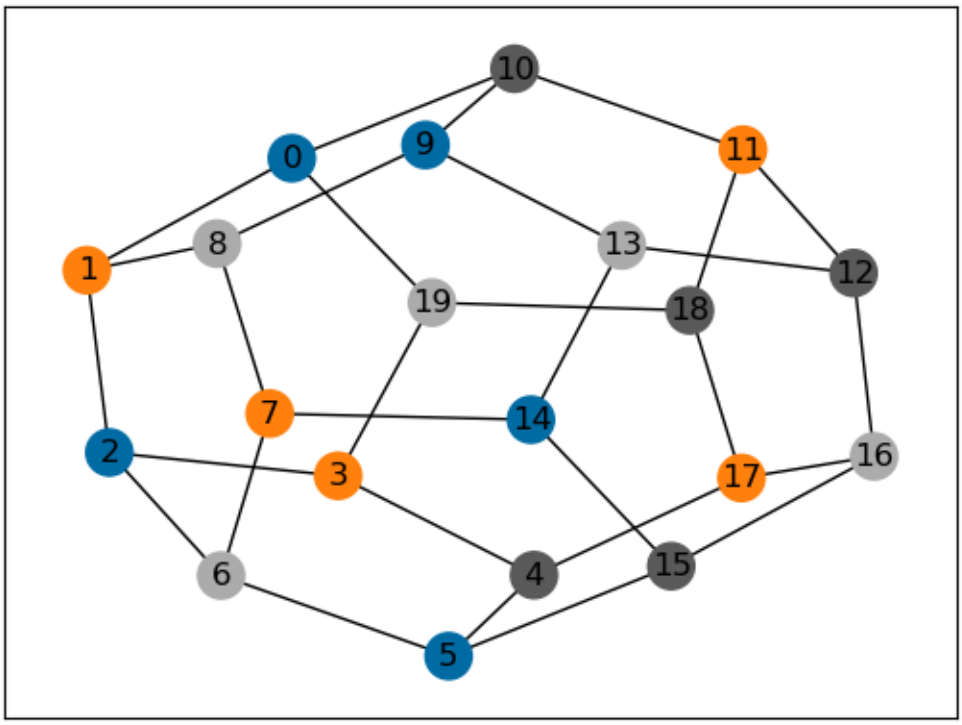
(continues on next page)

(continued from previous page)

```
    node_color=gcol.get_node_colors(G, c, palette=gcol.colorblind)
)
plt.show()
```






The following shows the colors that are available in each palette. These are identified by the integers, starting from 0.

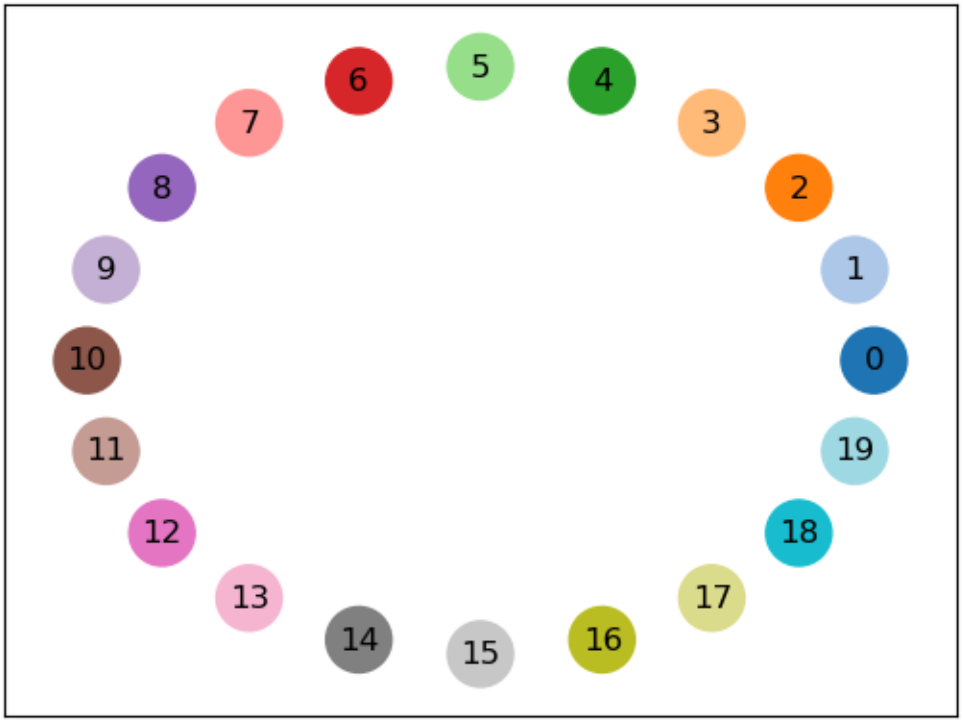
```
G = nx.complete_graph(20)
c = gcol.node_coloring(G)
nx.draw_networkx(
    G,
    node_color=gcol.get_node_colors(G, c, gcol.tableau),
    pos=gcol.coloring_layout(G, c),
    node_size=600,
    width=0.00
)
print("The (default) gcol.tableau palette (20 colors):")
plt.show()

G = nx.complete_graph(56)
c = gcol.node_coloring(G)
nx.draw_networkx(
    G,
    node_color=gcol.get_node_colors(G, c, gcol.colorful),
    pos=gcol.coloring_layout(G, c),
    node_size=150,
    font_size=10,
    width=0.00
)
print("The gcol.colorful palette (56 colors):")
plt.show()

G = nx.complete_graph(10)
c = gcol.node_coloring(G)
```

(continues on next page)

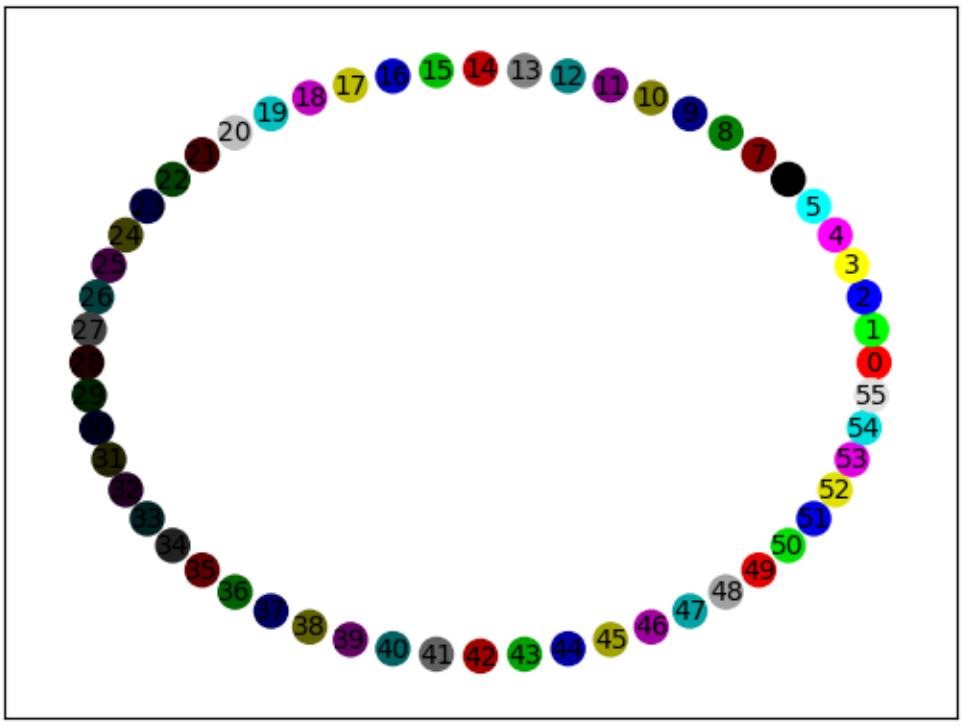
(continued from previous page)

```
nx.draw_networkx(
    G,
    node_color=gcol.get_node_colors(G, c, gcol.colorblind),
    pos=gcol.coloring_layout(G, c),
    node_size=800,
    width=0.00
)
print("The gcol.colorblind palette (10 colors):")
plt.show()
```

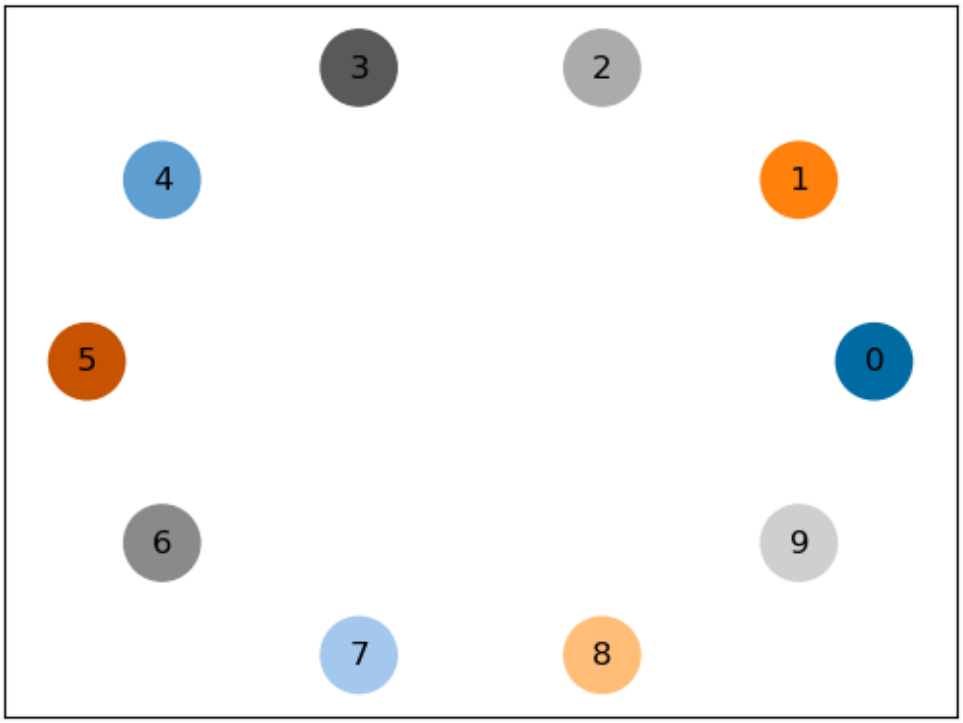
```
The (default) gcol.tableau palette (20 colors):
```




```
The gcol.colorful palette (56 colors):
```



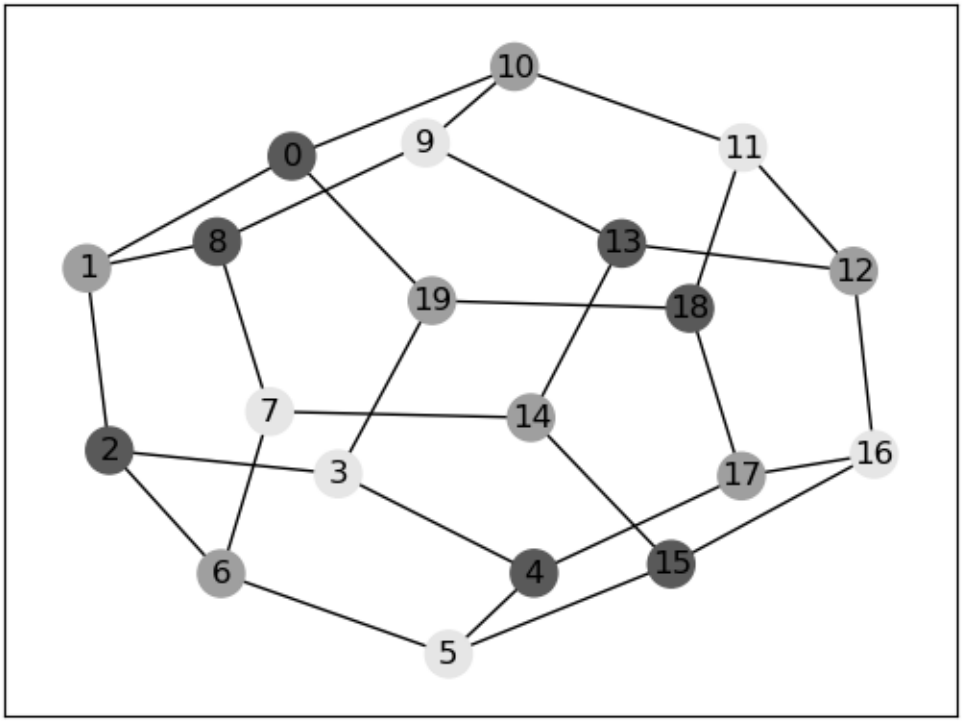
```
The gcol.colorblind palette (10 colors):
```




User-defined palettes can also be created. The following demonstrates how to create a greyscale palette based on the number of colors $k$ in the current solution `c`.

```
def grayscale(k):
    minVal, maxVal, palette = 0.35, 0.9, {}
    step = (maxVal - minVal) / (k - 1)
    for i in range(k):
        x = minVal + step * i
        palette[i] = (x, x, x)
    palette[-1] = (1.0, 1.0, 1.0)
    return palette

G = nx.dodecahedral_graph()
c = gcol.node_coloring(G)
k = max(c.values()) + 1
print("Custom greyscale palette based on three colors:")
nx.draw_networkx(
    G,
    pos=nx.spring_layout(G, seed=1),
    node_color=gcol.get_node_colors(G, c, palette=grayscale(k))
)
plt.show()
```

```
Custom greyscale palette based on three colors:
```



CHAPTER

FOUR

# CASE STUDY: EXAM TIMETABLING

In this chapter, we consider a practical case study concerning the use of graph coloring methods in the production of exam timetables at a university. This will help to showcase the various tools available in the `gcol` library.

## 4.1 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 sitting 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.

## 4.2 An Initial Solution

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.

## 4.3 Scheduling Large Exams First

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 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)
```

(continues on next page)

(continued from previous page)

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

(continues on next page)

(continued from previous page)

```
Timeslot 14 : [48, 132]
```

## 4.4 Balancing Exams

Despite the previous approach 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 443.

```
#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
```

(continues on next page)

(continued from previous page)

```
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
```

## 4.5 Limiting Timeslots

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)
P = gcol.partition(c)
U = list(u 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 unscheduled exams are", U)
print("They have the following sizes", [G.nodes[u]["weight"] for u in U])
```

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

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

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

FIVE

# PERFORMANCE ANALYSIS

In this chapter 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.

Here, algorithms are evaluated by looking at solution quality and run times. Details on 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* unlabeled $n$-node graphs. This allows us to make general statistical statements about performance across the set of all $n$-node graphs, although 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, where a wider set of trials is conducted.

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`, aggregated in a `pivot` table, and summarized in charts. 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.

We start by importing the libraries we need.

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

## 5.1 Differing Node Coloring Strategies

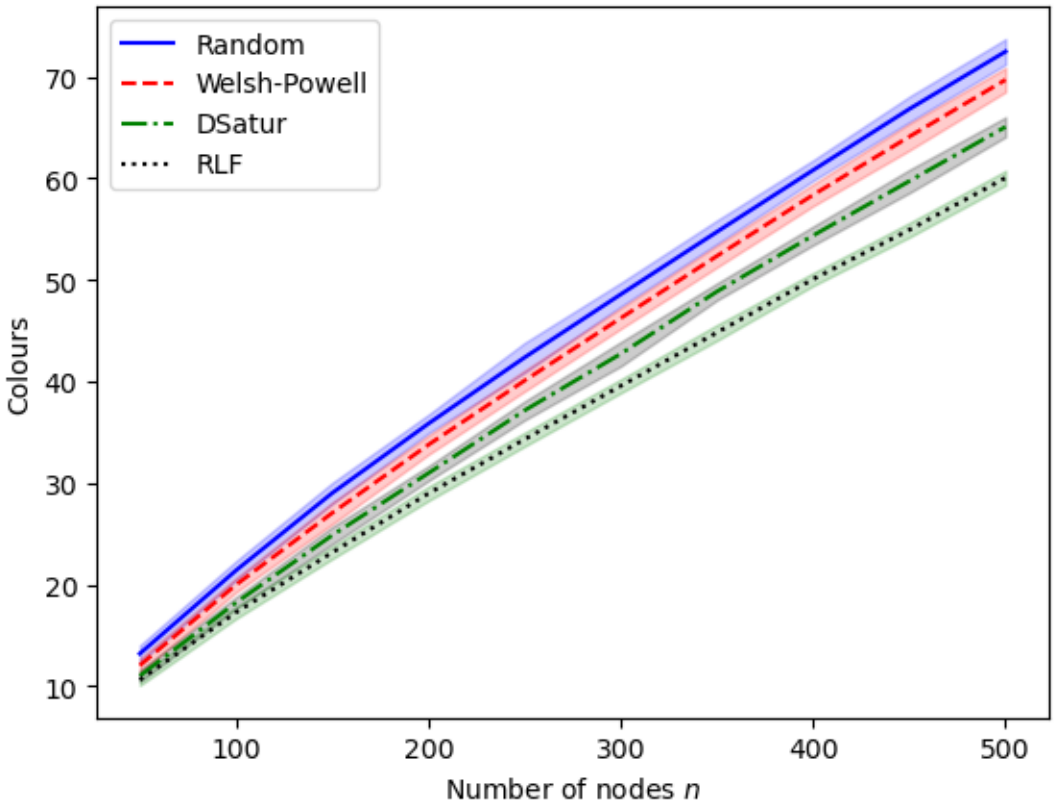
In our first experiment, we compare the different constructive strategies available for node coloring in the `gcol` library (namely `'random'`, `'welsh_powell'`, `'dsatur'`, and `'rlf'`) using $G(n, 0.5)$ graphs with values of $n$ between $50$ and $500$. The results are shown in the charts below. Further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

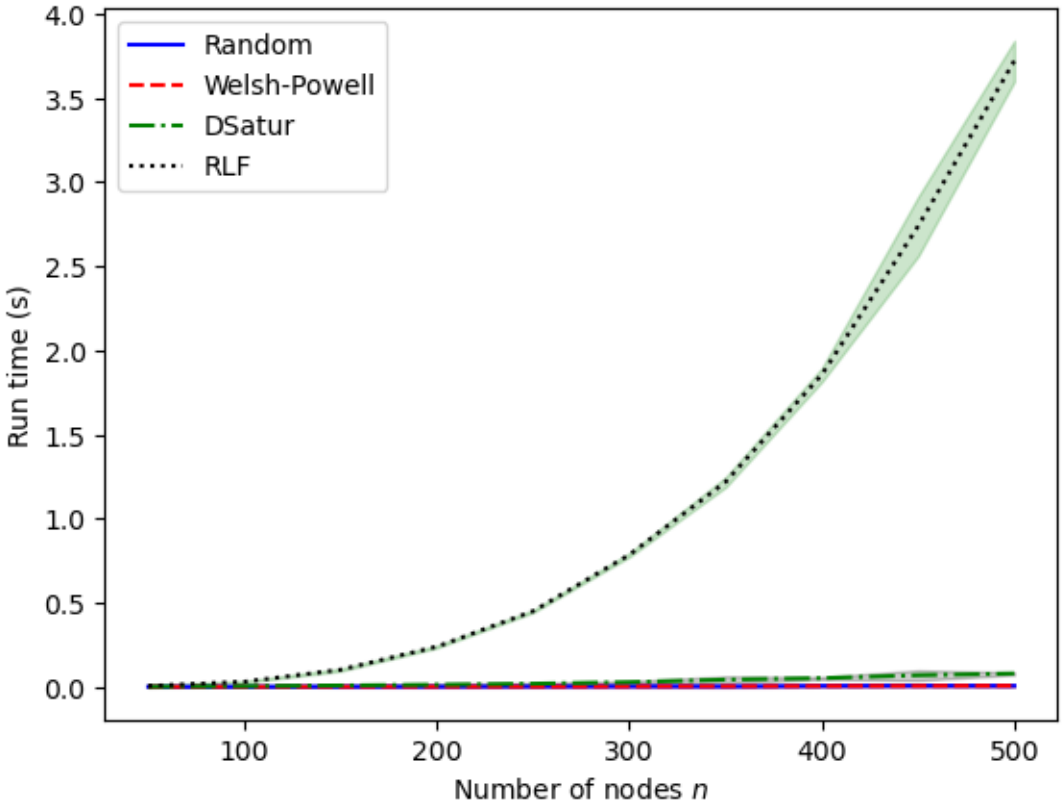
```
#Carry out the trials and put the results into a list
results = []
nVals = range(50,501,50)
for n in nVals:
    for seed in range(50):
```

(continues on next page)

(continued from previous page)

```
        G = nx.gnp_random_graph(n, 0.5, seed)
        for strategy in ["random", "welsh_powell", "dsatur", "rlf"]:
            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 and make a pivot table
df = pd.DataFrame(results, columns=["n", "seed", "strategy", "cols", "time"])
pivot = df.pivot_table(columns='strategy', aggfunc=['mean','std'], values=['cols','time
↪'], index='n')

# Now use the pivot table 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
4.0
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 other options. A good compromise seems to be struck by the `dsatur` strategy, which features comparatively good solution quality and run times.

## 5.2 Optimization Output

The following code demonstrates how the `verbose` parameter can be used to produce run-time output for the various optimization algorithms. This allows us to monitor algorithm performance during execution.

```
G = nx.gnp_random_graph(50, 0.5)
c = gcol.node_coloring(G, strategy="welsh_powell", opt_alg=1, verbose=1)
```

```
Running backtracking algorithm:
    Found solution with 13 colors. Total backtracking iterations = 0
    Found solution with 12 colors. Total backtracking iterations = 52
    Found solution with 11 colors. Total backtracking iterations = 1293
    Found solution with 10 colors. Total backtracking iterations = 28972977
Ending backtracking at iteration 34324898 - optimal solution achieved.
```

In the above example, the initial solution has used 13 colors. The backtracking algorithm (`opt_alg=1`) has been used to reduce the number of colors, eventually finding an optimal solution. We can do similar things with the other optimization algorithms while controlling their number of iterations:

```
G = nx.gnp_random_graph(50, 0.5)
c = gcol.node_coloring(G, strategy="welsh_powell", opt_alg=2, it_limit=10000, verbose=1)
```

```
Running local search algorithm:
    Found solution with 12 colors. Total local search iterations = 0 / 10000
    Found solution with 11 colors. Total local search iterations = 12 / 10000
    Found solution with 10 colors. Total local search iterations = 241 / 10000
Ending local search. Iteration limit of 10000 has been reached.
```

In some cases, we can also increase the amount of output by using a larger value with `verbose`:

```
G = nx.gnp_random_graph(50, 0.5)
c = gcol.node_coloring(G, strategy="welsh_powell", opt_alg=3, it_limit=10000, verbose=2)
```

```
Running local search algorithm:
    Found solution with 12 colors. Total local search iterations = 0 / 10000
    Running PartialCol algorithm using 11 colors
        Solution with 11 colors and cost 5 found by PartialCol at iteration 0
        Solution with 11 colors and cost 4 found by PartialCol at iteration 1
        Solution with 11 colors and cost 3 found by PartialCol at iteration 2
        Solution with 11 colors and cost 2 found by PartialCol at iteration 3
        Solution with 11 colors and cost 1 found by PartialCol at iteration 4
        Solution with 11 colors and cost 0 found by PartialCol at iteration 6
    Ending PartialCol
    Found solution with 11 colors. Total local search iterations = 6 / 10000
    Running PartialCol algorithm using 10 colors
        Solution with 10 colors and cost 5 found by PartialCol at iteration 0
```

(continues on next page)

(continued from previous page)

```
        Solution with 10 colors and cost 4 found by PartialCol at iteration 1
        Solution with 10 colors and cost 3 found by PartialCol at iteration 22
        Solution with 10 colors and cost 2 found by PartialCol at iteration 32
        Solution with 10 colors and cost 1 found by PartialCol at iteration 155
        Solution with 10 colors and cost 0 found by PartialCol at iteration 896
    Ending PartialCol
    Found solution with 10 colors. Total local search iterations = 902 / 10000
    Running PartialCol algorithm using 9 colors
        Solution with 9 colors and cost 5 found by PartialCol at iteration 0
        Solution with 9 colors and cost 4 found by PartialCol at iteration 6
        Solution with 9 colors and cost 3 found by PartialCol at iteration 46
        Solution with 9 colors and cost 2 found by PartialCol at iteration 1498
    Ending PartialCol
Ending local search. Iteration limit of 10000 has been reached.
```

## 5.3 Comparison to NetworkX

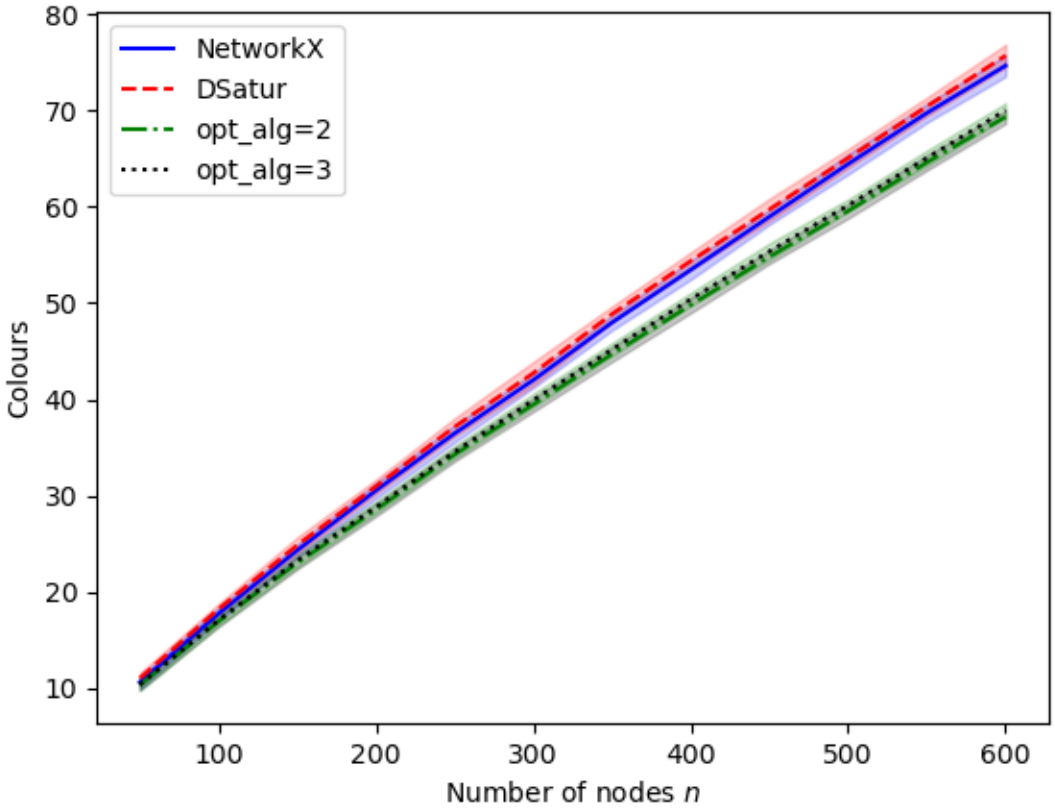
The next set of experiments compares the performance of `gcol`'s local search routines and NetworkX's interchange coloring routine. As a benchmark, we also include `gcol`'s `dsatur` option from earlier, which is also used to produce the initial solutions for the local search algorithms. For comparative purposes, two 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.

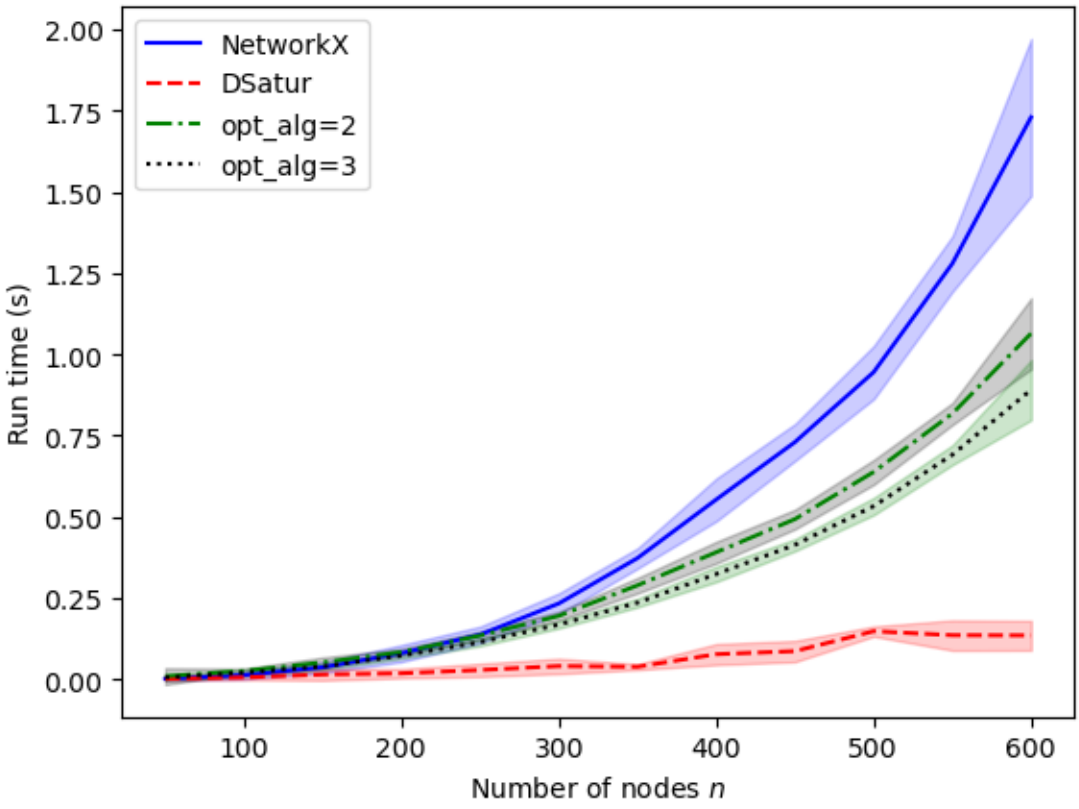
```
#Carry out the trials and put the results into a list
results = []
nVals = range(50,601,50)
for n in nVals:
    for seed in range(50):
        G = nx.gnp_random_graph(n, 0.5, seed)
        start = time.time()
        c = nx.greedy_color(G, "largest_first", interchange=True)
        results.append([n, 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')

# 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")]
```

(continues on next page)

(continued from previous page)

```
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 $n$")
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()
```


NetworkX
DSatur
opt_alg=2
opt_alg=3
Colours
Number of nodes $n$
80
70
60
50
40
30
20
10
100
200
300
400
500
600


NetworkX
DSatur
opt_alg=2
opt_alg=3
Run time (s)
Number of nodes $n$
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
100
200
300
400
500
600

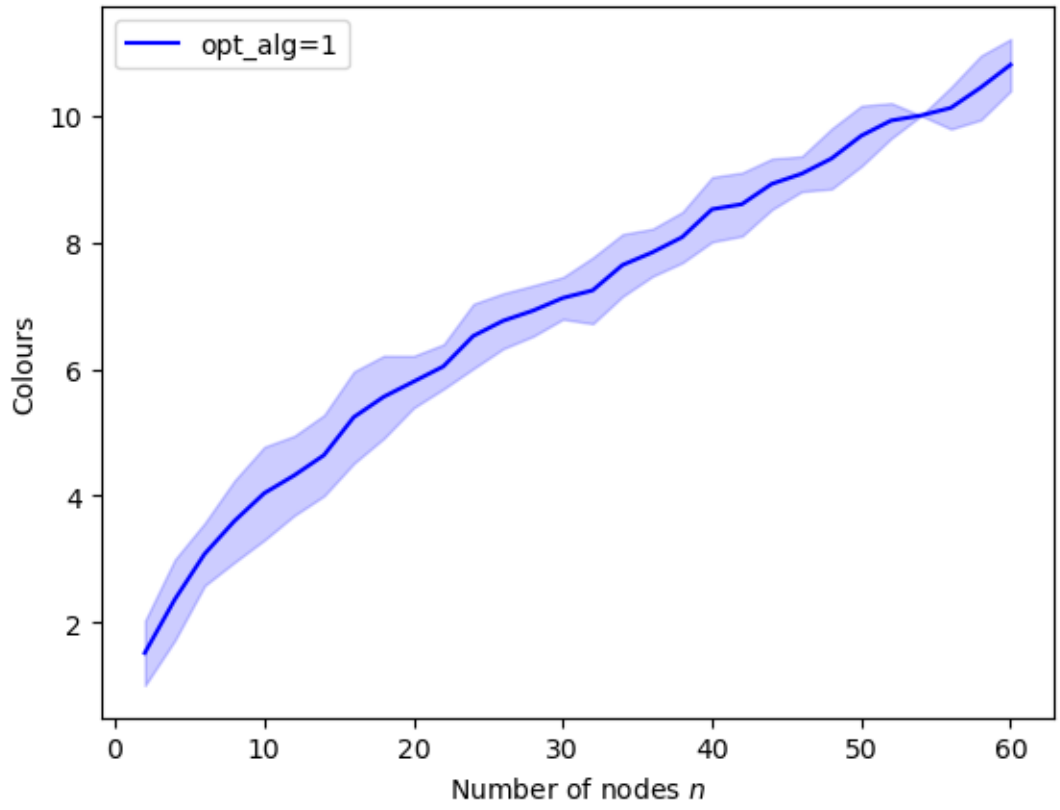
It is clear from the above results that the local search algorithms make significant improvements to the solutions provided by the `dsatur` strategy, albeit with additional time requirements. The solutions and run times of these local search algorithms are also superior to NetworkX's node coloring routines.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

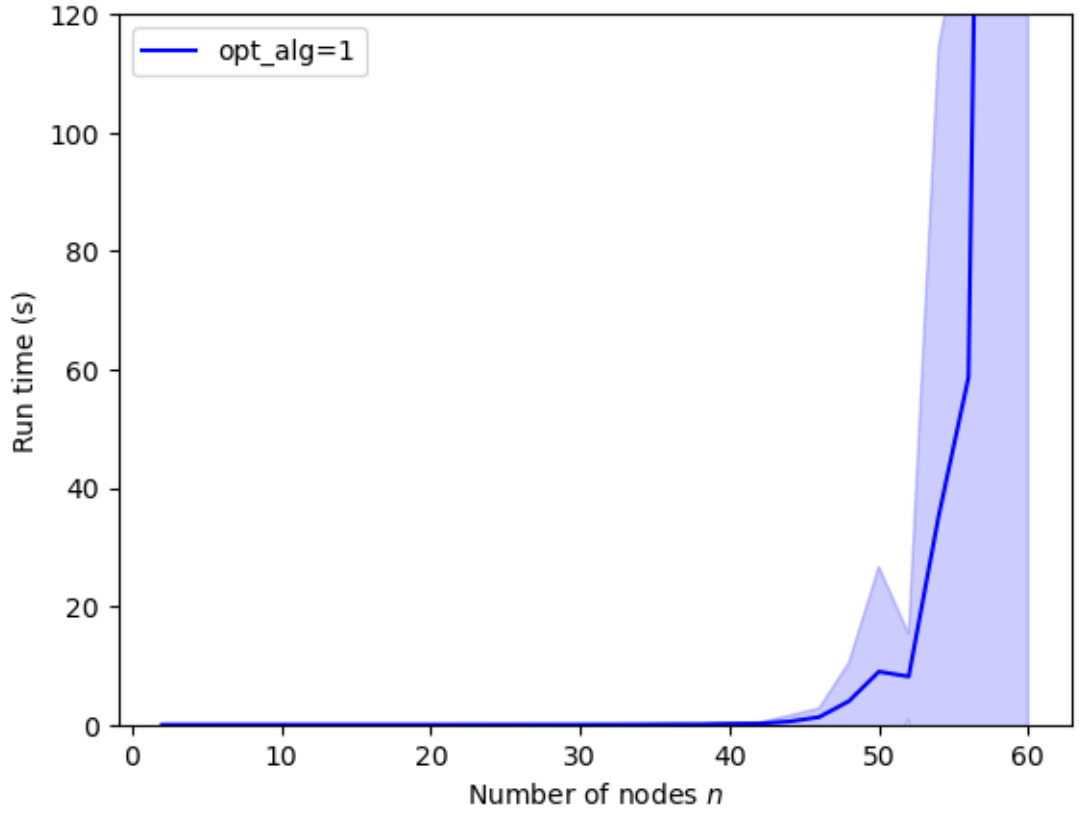
## 5.4 Exact Algorithm Performance

In addition to local search, the `gcol` library features an exact, exponential-time algorithm for node coloring, based on backtracking. This algorithm is invoked by setting `opt_alg=1` (the parameter `it_limit` is redundant here). 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, it 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(2,61,2)
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 dataframe from this list and make a pivot
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, 120))
plt.ylabel("Run time (s)")
plt.legend()
plt.show()
```


opt_alg=1
Colours
Number of nodes $n$
0
10
20
30
40
50
60
2
4
6
8
10


opt_alg=1
Run time (s)
Number of nodes $n$
0
10
20
30
40
50
60
0
20
40
60
80
100
120

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 demonstrates the disadvantages of using this exponential-time algorithm: once $n$ is increased beyond a moderately small value (approximately 55 here), run times become unpredictable and often very long. Note, however, that the specific $n$-values that give these long run times can vary considerably depending on the topology of the graph. For example, planar graphs and scale-free graphs with several hundred nodes are often solved very quickly by the backtracking algorithm. These sorts of results will usually need to be confirmed empirically.

## 5.5 Local Search Comparison

In addition to the above exact algorithm, the `gcol` library features a choice of four high-performance optimization heuristics, based on local search (more specifically, tabu search).

- `opt_alg=2` makes use of the TabuCol algorithm
- `opt_alg=3` makes use of the PartialCol algorithm
- `opt_alg=4` uses a hybrid evolutionary algorithm (HEA) in conjunction with TabuCol
- `opt_alg=5` uses a hybrid evolutionary algorithm (HEA) in conjunction with PartialCol

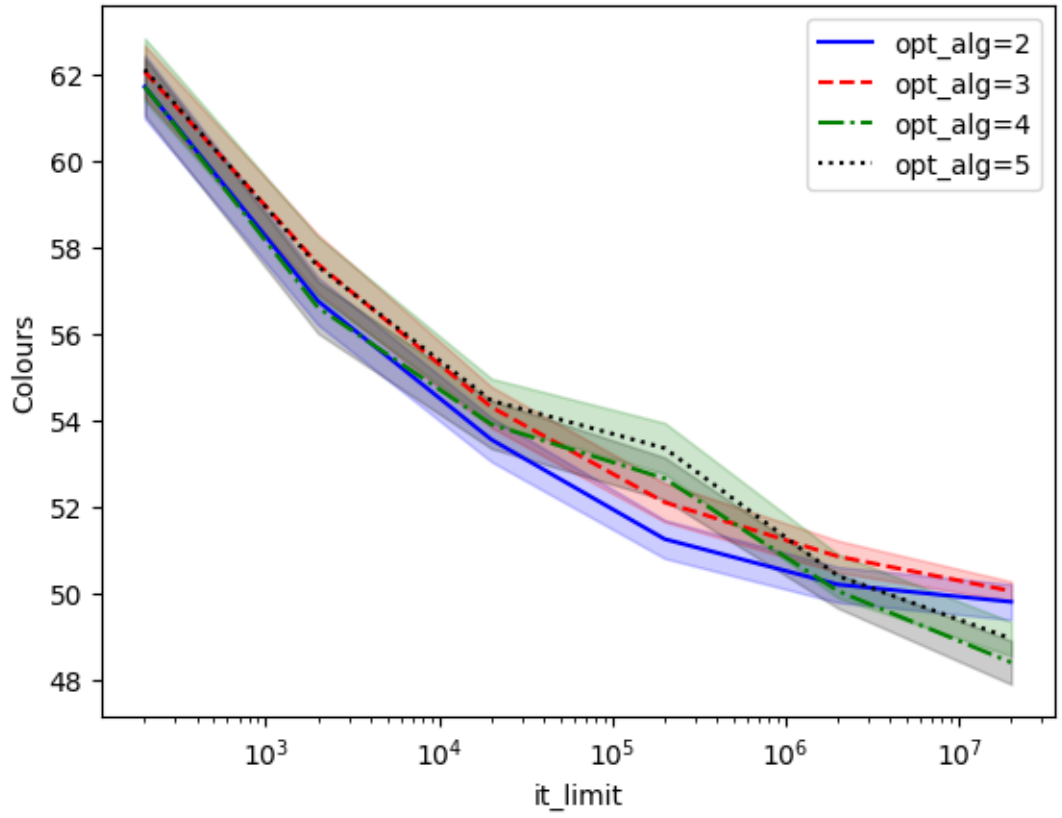
These are among the best-known algorithms for graph coloring. Further information on these can be found in the library's documentation and in this book. Executing these heuristics with higher iteration limits usually leads to better solutions (that is, solutions using fewer colors). To illustrate, the following code runs the four optimization algorithms with differing iteration limits on a set of twenty $G(500, 0.5)$ graphs.

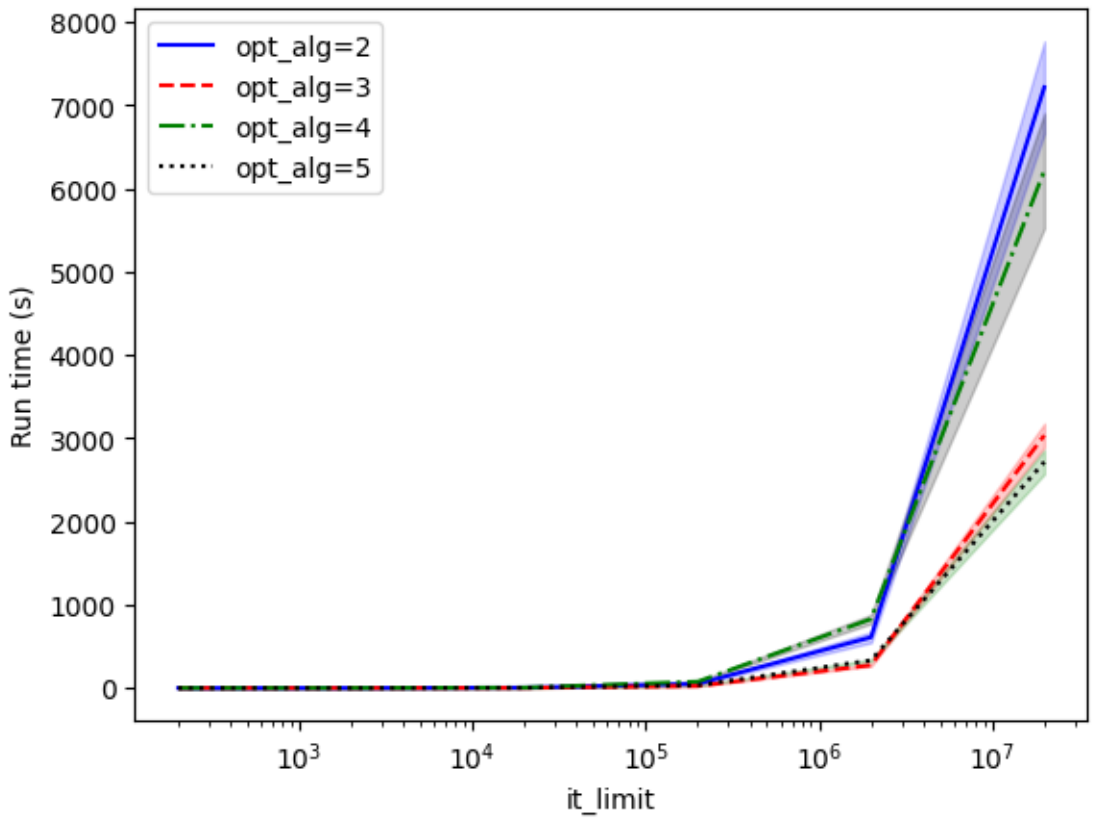
```
results = []
it_limits = [200, 2000, 20000, 200000, 2000000, 20000000]
for seed in range(20):
    G = nx.gnp_random_graph(500, 0.5, seed)
    for opt_alg in [2, 3, 4, 5]:
        for it_limit in it_limits:
            start = time.time()
            c = gcol.node_colouring(G, opt_alg=opt_alg, it_limit=it_limit)
            results.append([seed, opt_alg, it_limit, max(c.values()) + 1, time.time()-
↪start])

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

# Use the pivot table to make charts as before
mean1, SD1 = pivot[("mean","cols",2)], pivot[("std","cols",2)]
mean2, SD2 = pivot[("mean","cols",3)], pivot[("std","cols",3)]
mean3, SD3 = pivot[("mean","cols",4)], pivot[("std","cols",4)]
mean4, SD4 = pivot[("mean","cols",5)], pivot[("std","cols",5)]
plt.plot(it_limits, mean1, linestyle='-', linewidth=1.5, color="b", label='opt_alg=2')
plt.fill_between(it_limits, mean1-SD1, mean1+SD1, color='b', alpha=0.2)
plt.plot(it_limits, mean2, linestyle='--', linewidth=1.5, color="r", label='opt_alg=3')
plt.fill_between(it_limits, mean2-SD2, mean2+SD2, color='r', alpha=0.2)
plt.plot(it_limits, mean3, linestyle='-.', linewidth=1.5, color="g", label='opt_alg=4')
plt.fill_between(it_limits, mean3-SD3, mean3+SD3, color='k', alpha=0.2)
plt.plot(it_limits, mean4, linestyle=':', linewidth=1.5, color="black", label='opt_alg=5
↪')
```

(continues on next page)

(continued from previous page)

```
plt.fill_between(it_limits, mean4-SD4, mean4+SD4, color='g', alpha=0.2)
plt.xlabel("it_limit")
plt.ylabel("Colours")
plt.legend()
plt.xscale('log')
plt.show()

mean1, SD1 = pivot[("mean","time",2)], pivot[("std","time",2)]
mean2, SD2 = pivot[("mean","time",3)], pivot[("std","time",3)]
mean3, SD3 = pivot[("mean","time",4)], pivot[("std","time",4)]
mean4, SD4 = pivot[("mean","time",5)], pivot[("std","time",5)]
plt.plot(it_limits, mean1, linestyle='-', linewidth=1.5, color="b", label='opt_alg=2')
plt.fill_between(it_limits, mean1-SD1, mean1+SD1, color='b', alpha=0.2)
plt.plot(it_limits, mean2, linestyle='--', linewidth=1.5, color="r", label='opt_alg=3')
plt.fill_between(it_limits, mean2-SD2, mean2+SD2, color='r', alpha=0.2)
plt.plot(it_limits, mean3, linestyle='-.', linewidth=1.5, color="g", label='opt_alg=4')
plt.fill_between(it_limits, mean3-SD3, mean3+SD3, color='k', alpha=0.2)
plt.plot(it_limits, mean4, linestyle=':', linewidth=1.5, color="black", label='opt_alg=5
↪')
plt.fill_between(it_limits, mean4-SD4, mean4+SD4, color='g', alpha=0.2)
plt.xlabel("it_limit")
plt.ylabel("Run time (s)")
plt.legend()
plt.xscale('log')
plt.show()
```






Note that the charts above use log scales on their horizontal axes. The first chart shows how using an increased iteration limit can result in solutions that have fewer colors. For very high limits, the hybrid evolutionary algorithms are clearly more favorable. The second chart shows how the iteration limits affect run times with these graphs.

## 5.6 Equitable Coloring

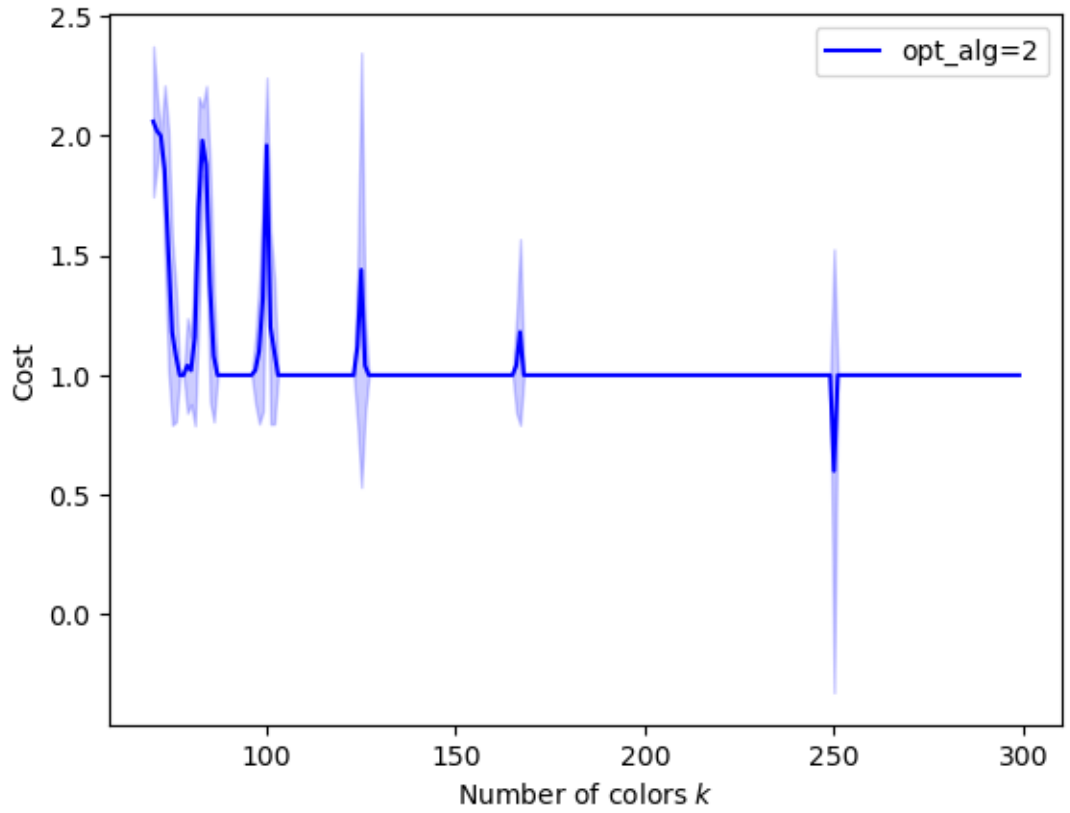
In the (unweighted) equitable node-coloring problem, we are interested in coloring the nodes with a user-defined number of colors $k$ so that (a) adjacent nodes have different colors, and (b) the number of nodes in each color is as equal as possible. The following trials run the `gcol.equitable_node_k_coloring()` 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.

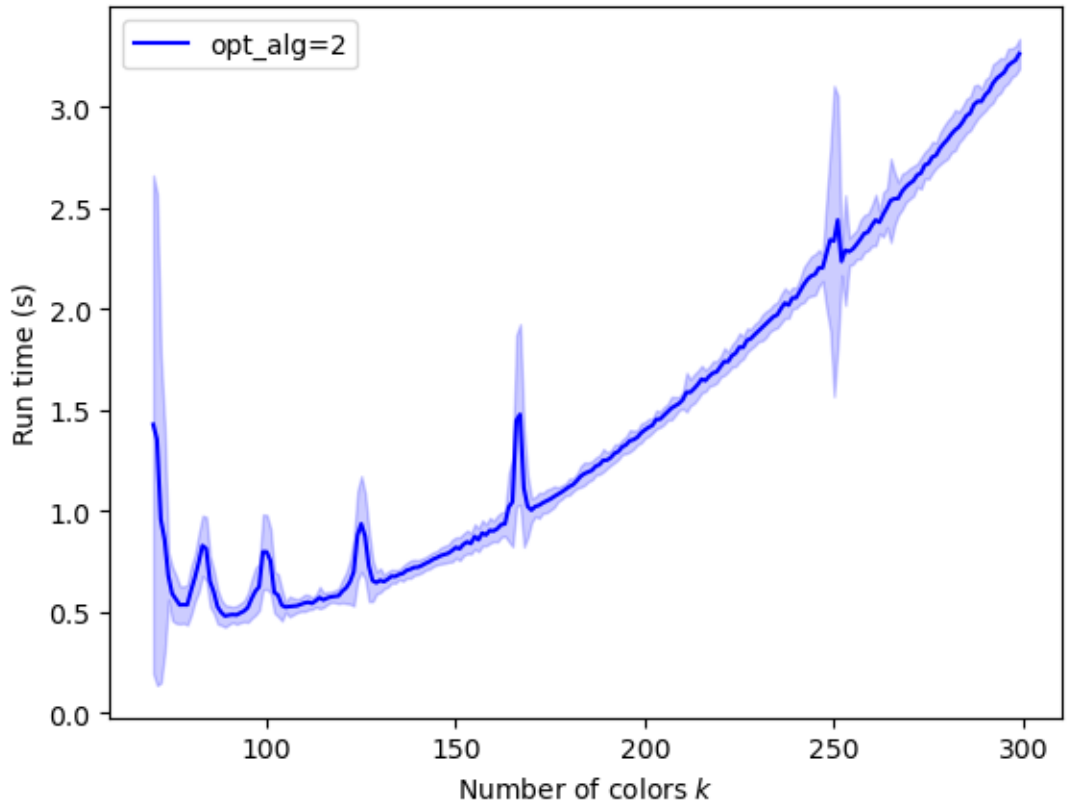
```
results = []
n = 500
kVals = range(70, 300, 1)
for seed in range(50):
    G = nx.gnp_random_graph(n, 0.5, seed)
    for k in kVals:
        start = time.time()
        c = gcol.equitable_node_k_coloring(G, k, opt_alg=2, it_limit=len(G))
        P = gcol.partition(c)
        cost = max(len(j) for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

(continues on next page)

(continued from previous page)

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

# 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 achieved, 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 features an exact equitable node $k$-coloring routine, but this can only be used for values of $k \geq \Delta(G) + 1$, 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.

## 5.7 Independent Set Comparison

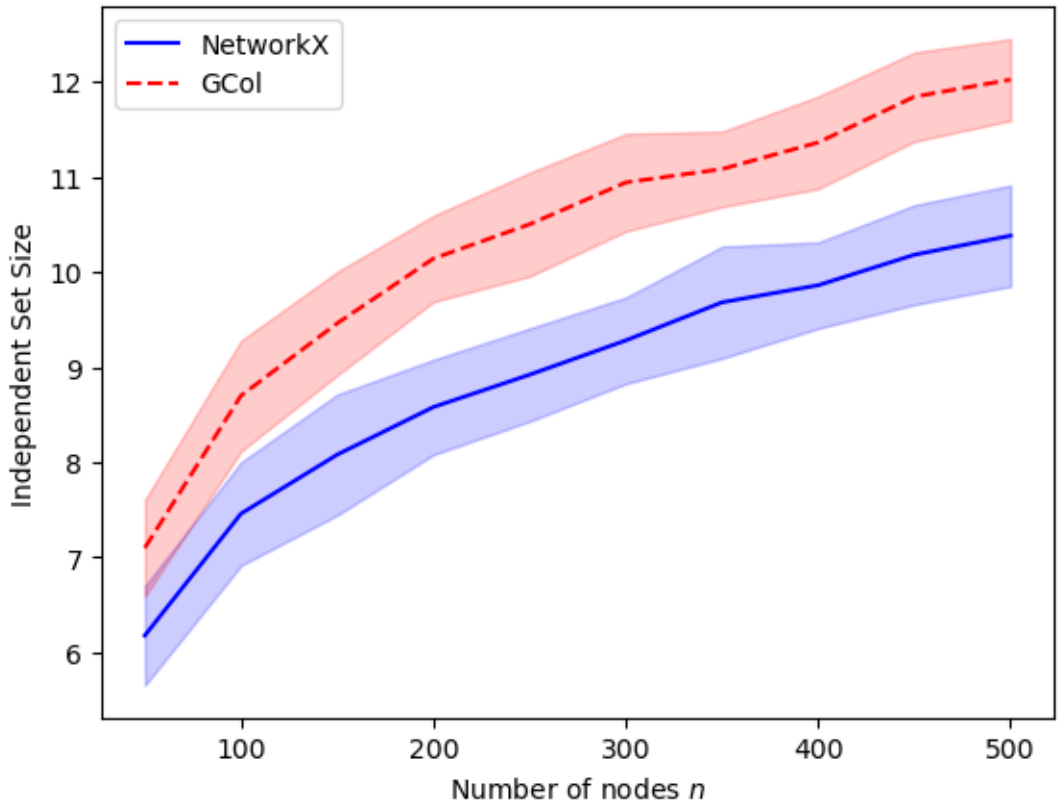
Our final set or trials looks at the performance the `gcol.max_independent_set()` routine and compares it to the approximation algorithm included in NetworkX for the same problem. As before, we use an iteration limit of $n$ for the former.

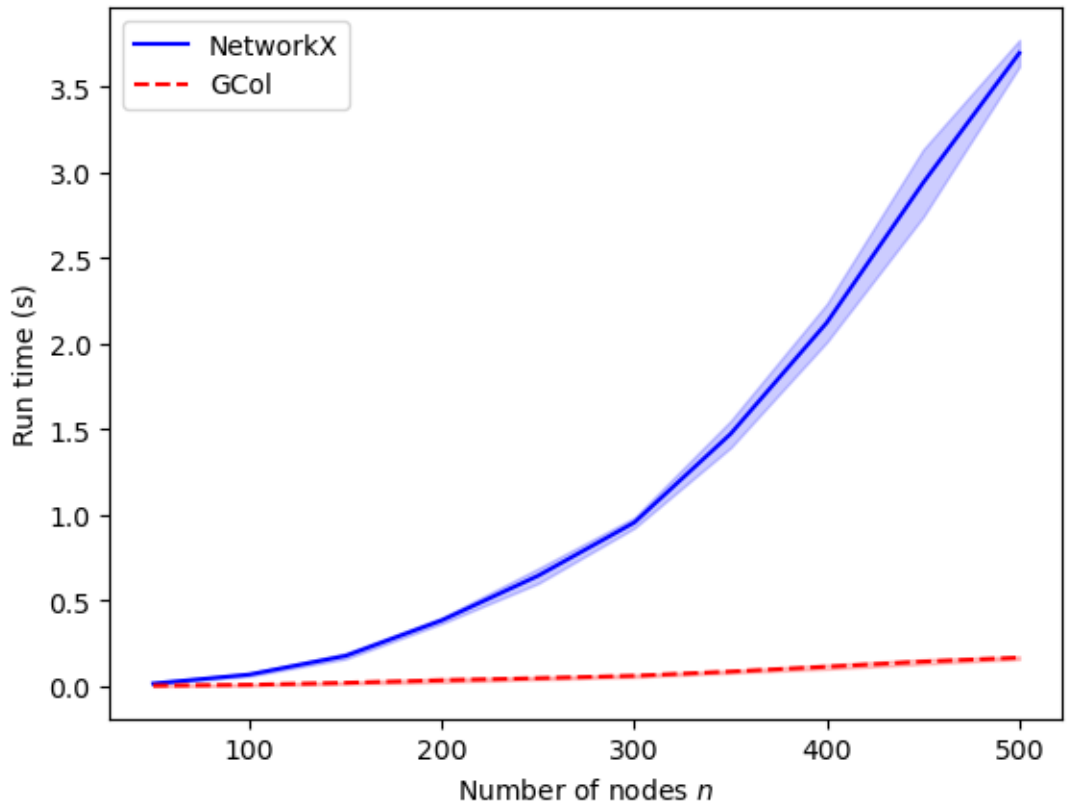
```
#Carry out the trials and put the results into a list
results = []
nVals = range(50,501,50)
for n in nVals:
    for seed in range(50):
```

(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')

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

CHAPTER

SIX

# GALLERY

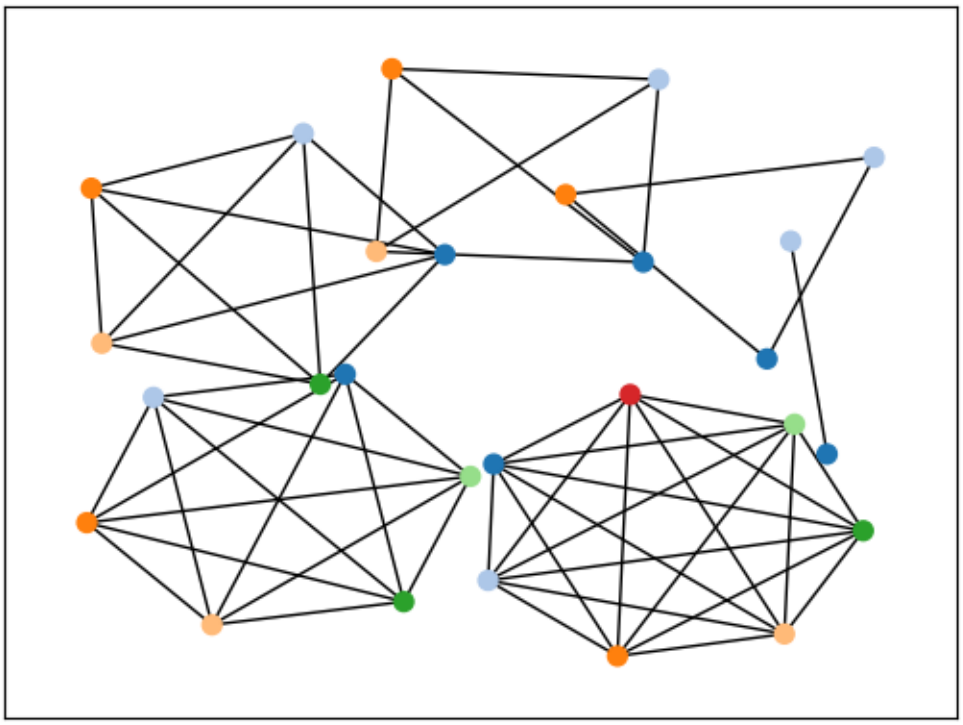
This chapter contains visualizations generated using the `gcol` library in conjunction with `networkx`. Each example is accompanied by the code used to generate it. We start by importing the necessary libraries.

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

## 6.1 Complete Graphs

A complete graph with $n$ nodes has a chromatic number of $n$. The following example demonstrates this simple result using complete graphs of up to seven nodes. The option `nx.kamada_kawai_layout()` is used to position the nodes in a pleasing manner.

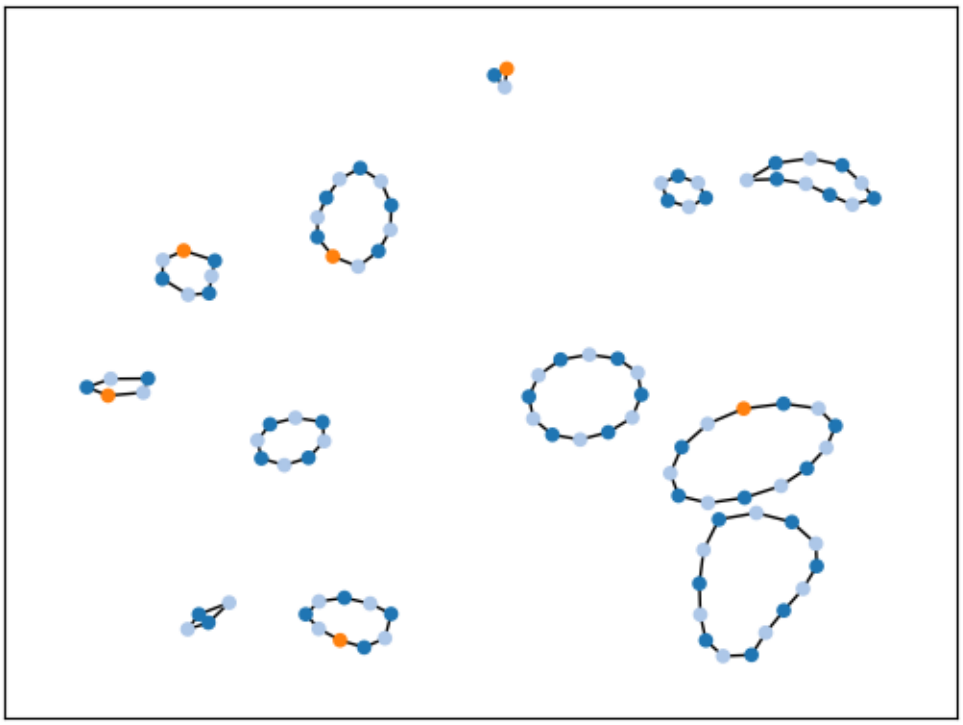
```
def make_complete(n, s):
    H = nx.Graph()
    for i in range(n-1):
        for j in range(i+1, n):
            H.add_edge(s+i, s+j)
    return H

G = nx.Graph()
for n in range(2, 8):
    s = n*(n-1)//2
    H = make_complete(n, s)
    G = nx.union(G, H)
c = gcol.node_coloring(G)
nx.draw_networkx(
    G,
    pos=nx.kamada_kawai_layout(G),
    node_color=gcol.get_node_colors(G, c),
    with_labels=False,
    node_size=50
)
plt.show()
```



## 6.2 Cycle Graphs

The following example considers cycle graphs with $n \geq 3$ nodes. When $n$ is even, cycle graphs are bipartite and therefore have a chromatic number of two; otherwise, the chromatic number is three. The following demonstrates this result in a similar manner to the previous example.

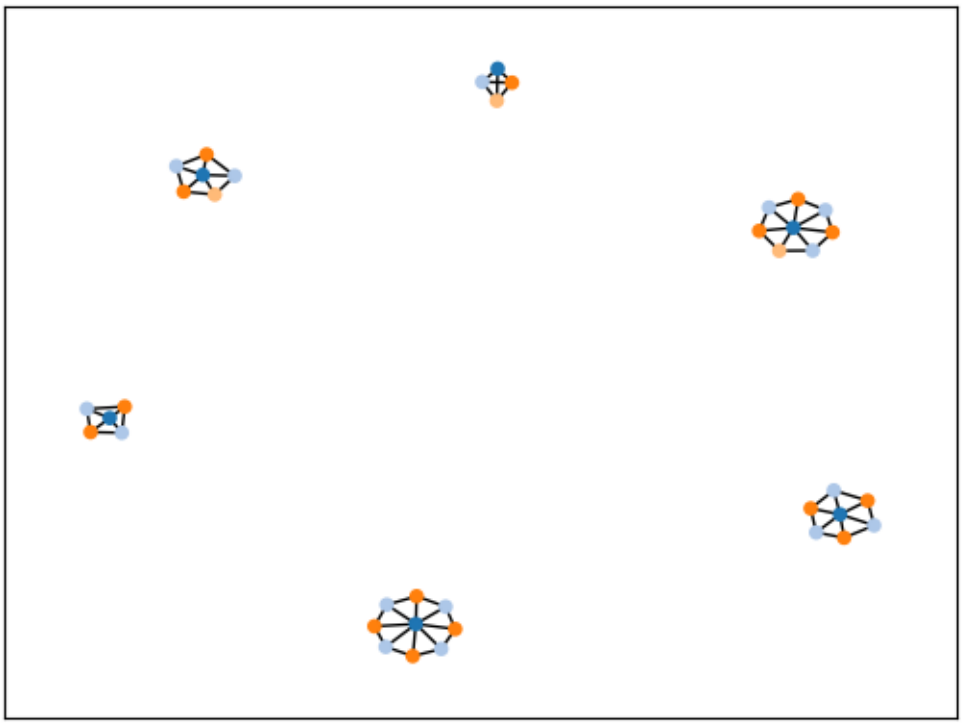
```
def make_cycle(n, s):
    H  = nx.Graph()
    for i in range(n-1):
        H.add_edge(s+i, s+i+1)
    H.add_edge(s+n-1,s)
    return H

G = nx.Graph()
for n in range(3, 15):
    s = n*(n-1)//2
    H = make_cycle(n, s)
    G = nx.union(G, H)
c = gcol.node_coloring(G)
nx.draw_networkx(
    G,
    pos=nx.spring_layout(G, seed=3),
    node_color=gcol.get_node_colors(G, c),
    with_labels=False,
    node_size=20
)
plt.show()
```



## 6.3 Wheel Graphs

The next example considers wheel graphs with $n \geq 4$ nodes. Cases with odd numbers of nodes have a chromatic number of three; otherwise, the chromatic number is four. The following demonstrates this.

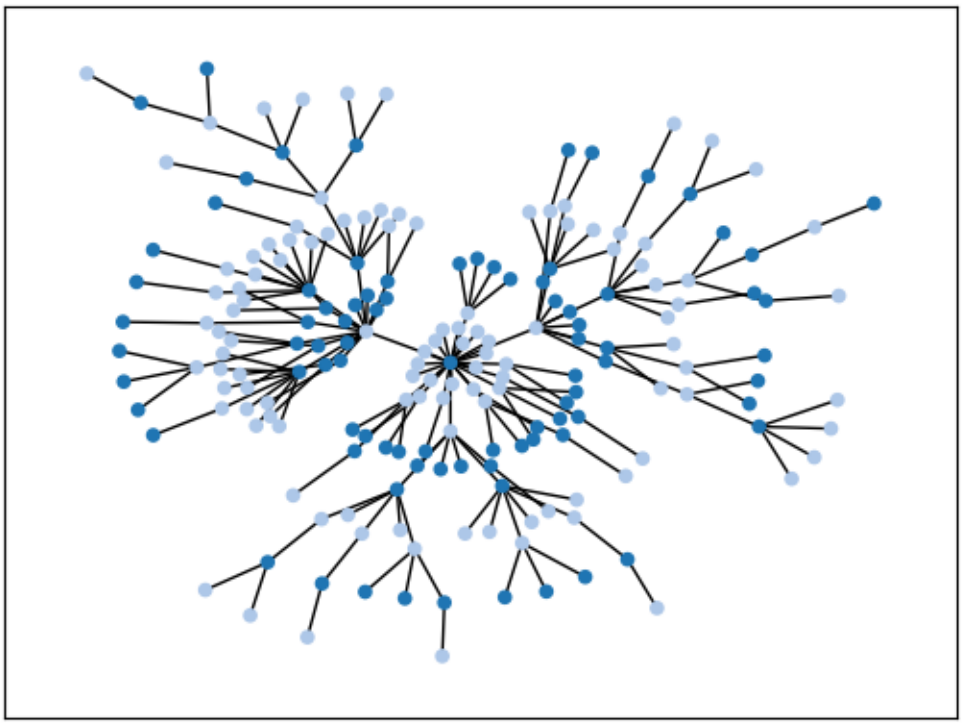
```
def make_wheel(n, s):
    H  = make_cycle(n-1, s)
    for i in range(n-1):
        H.add_edge(s+n-1, s+i)
    return H

G = nx.Graph()
for n in range(4, 10):
    s = n*(n-1)//2
    H = make_wheel(n, s)
    G = nx.union(G, H)
c = gcol.node_coloring(G)
nx.draw_networkx(
    G,
    pos=nx.spring_layout(G, seed=3),
    node_color=gcol.get_node_colors(G, c),
    with_labels=False,
    node_size=20
)
plt.show()
```



## 6.4 Trees

Trees are connected graphs that contain no cycles. Consequently, they are bipartite and have a chromatic number of two. The following code generates a tree using the NetworkX method `nx.barabasi_albert_graph()`. A node two-coloring of this tree is then generated.

```
G = nx.barabasi_albert_graph(200, 1)
c = gcol.node_coloring(G)
nx.draw_networkx(
    G,
    pos=nx.kamada_kawai_layout(G),
    node_color=gcol.get_node_colors(G, c),
    with_labels=False,
    node_size=20
)
plt.show()
```



## 6.5 Planar Graphs

The following code shows visualizations of a selection of planar graphs from the House of Graphs website. The names of the files used below refer to the graphs' ID numbers on the website. The files can also be found here. For each graph, we show a node coloring and face coloring.

The first two graphs considered below are Eulerian. Consequently, their face chromatic numbers are two, as illustrated.

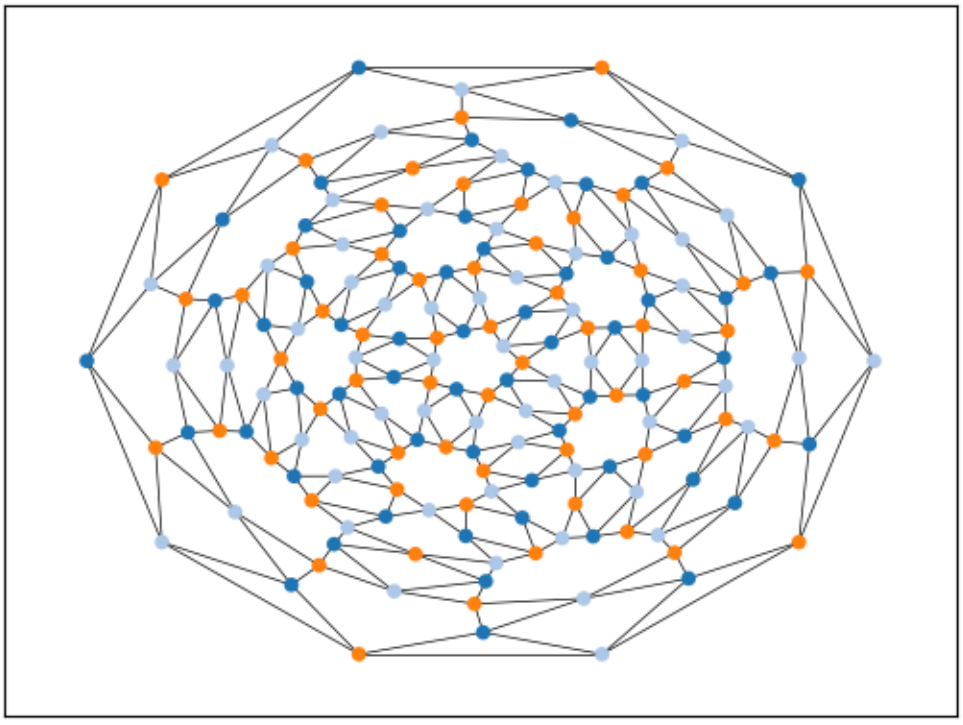
```
def graphFromFile(filename):
    G = nx.Graph()
    with open(filename, 'r') as f:
        f.readline()
        n = int(f.readline())
        for i in range(n):
            L = f.readline().split(" ")
            G.add_node(i, pos=(float(L[0]),float(L[1])))
            for j in range(2, len(L)):
                G.add_edge(i, int(L[j]))
    return G

files = ['HoG-51392.txt',
         'HoG-1317.txt',
         'HoG-1347.txt',
         'HoG-1122.txt']

for file in files:
    G = graphFromFile(file)
    pos = nx.get_node_attributes(G, "pos")
    c = gcol.node_coloring(G)
```

(continues on next page)

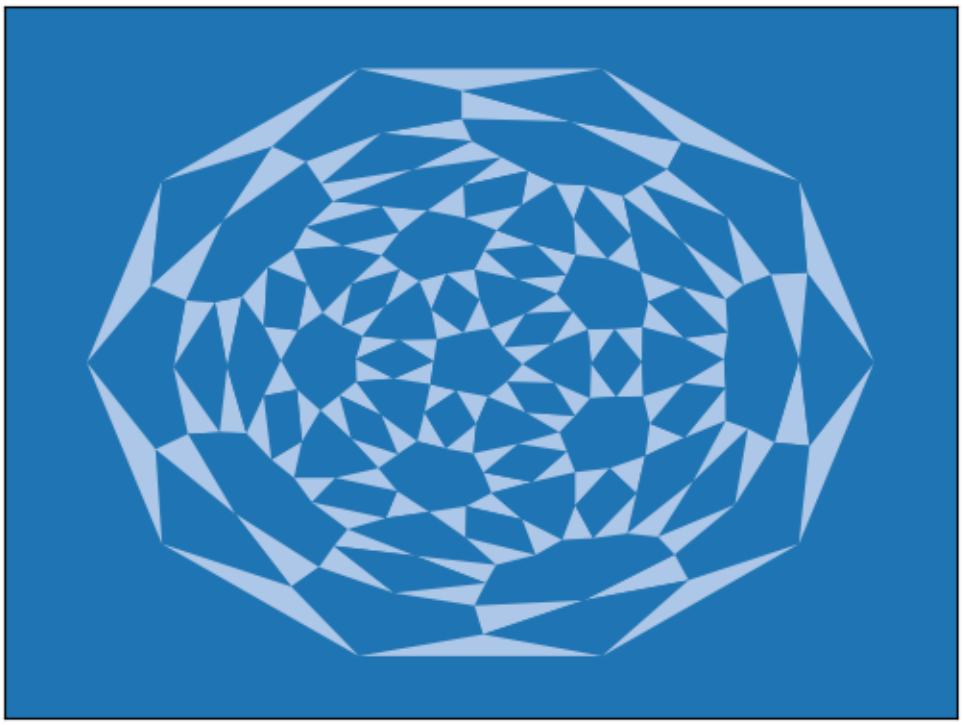
(continued from previous page)

```
    print("Colors =", max(c.values()) + 1)
    nx.draw_networkx(
        G,
        pos=pos,
        node_color=gcol.get_node_colors(G, c),
        with_labels=False,
        width=0.5,
        node_size=20
    )
    plt.show()
    c = gcol.face_coloring(G, pos)
    print("Colors =", max(c.values()) + 1)
    gcol.draw_face_coloring(c, pos, external=True)
    nx.draw_networkx(
        G,
        pos=pos,
        node_color='k',
        node_size=0,
        width=0,
        with_labels=False
    )
    plt.show()
```

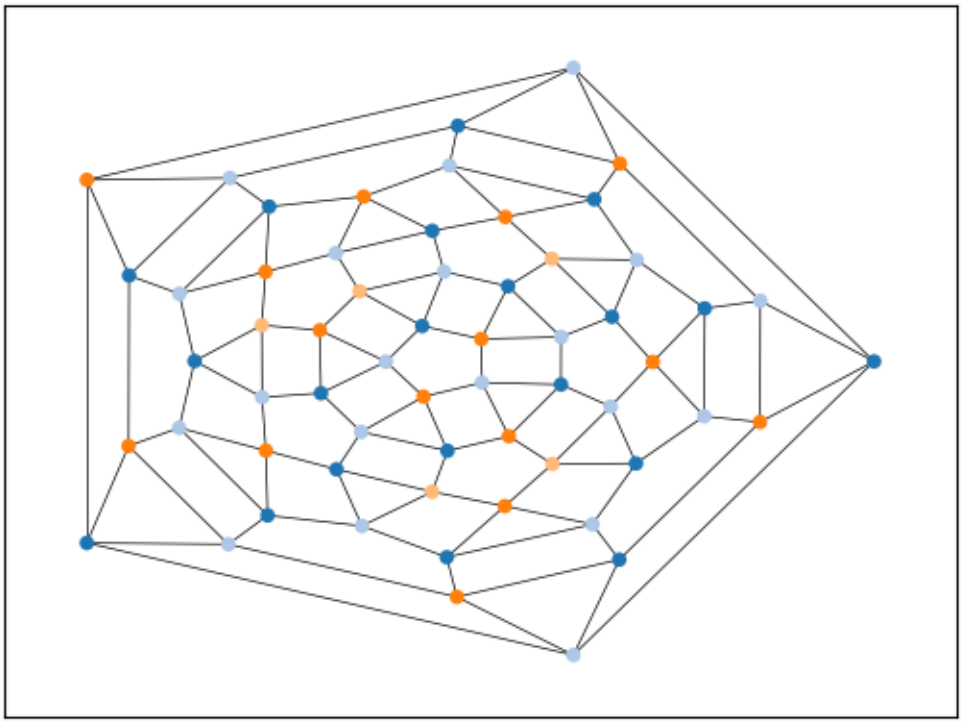
```
Colors = 3
```



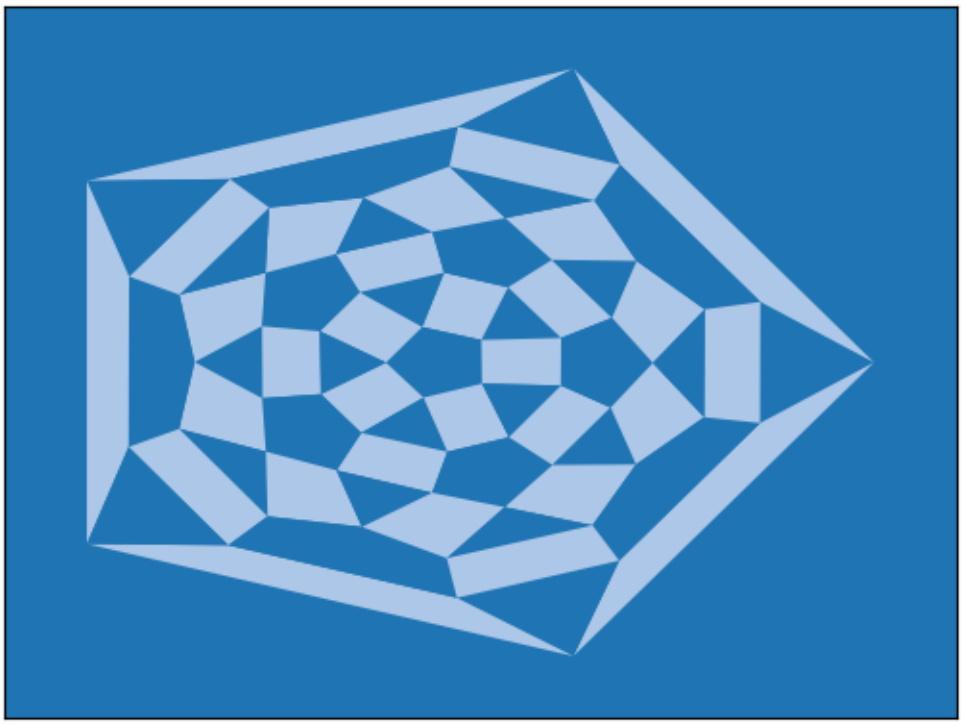
```
Colors = 2
```



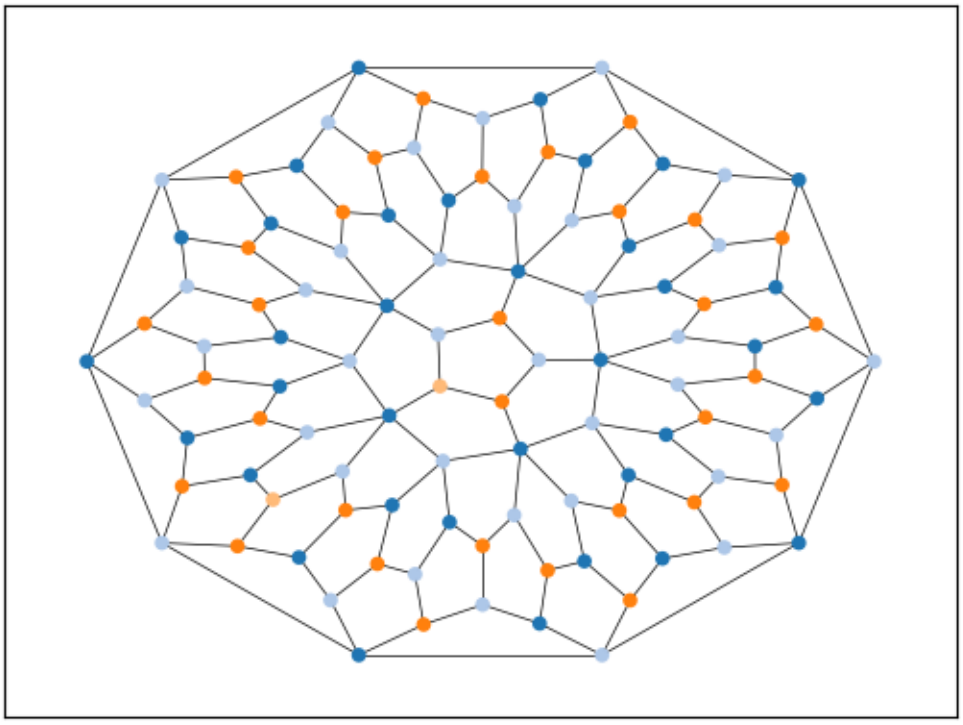
```
Colors = 4
```



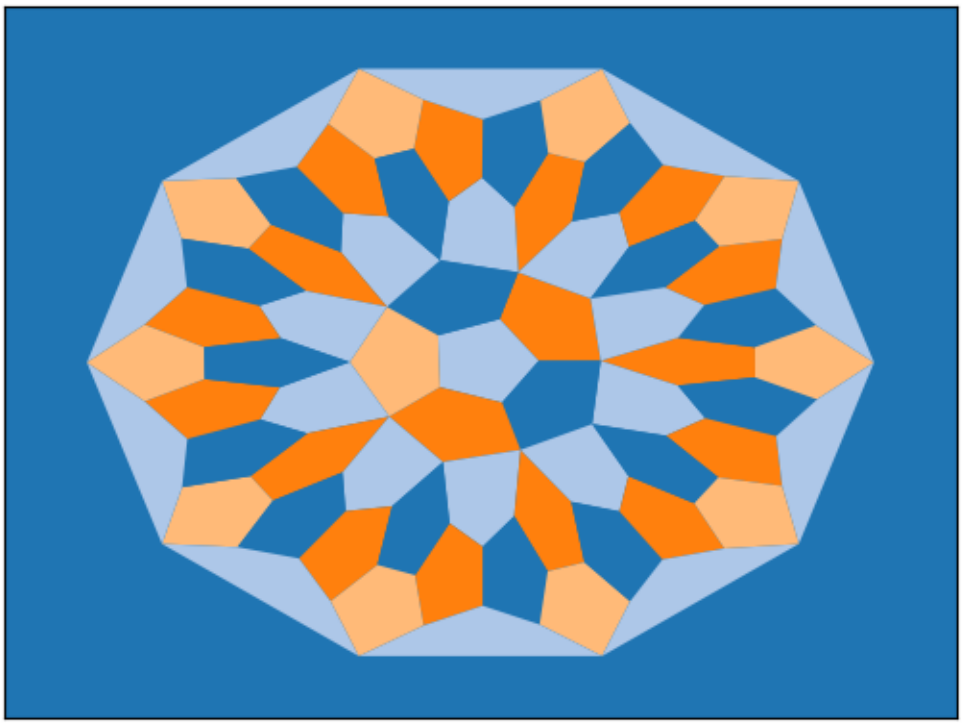
```
Colors = 2
```



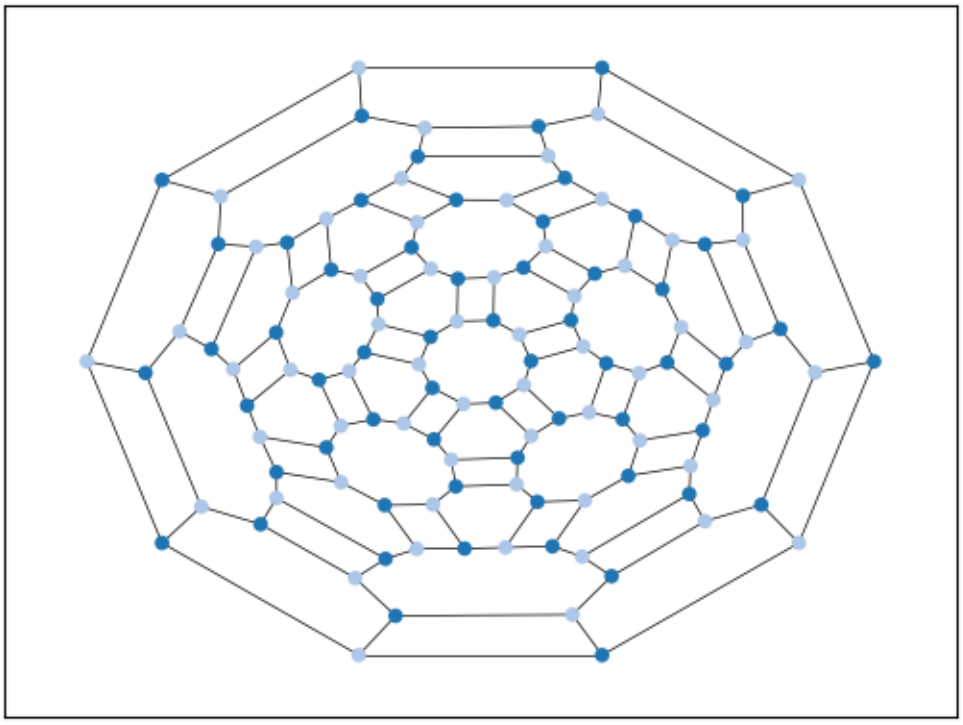
```
Colors = 4
```



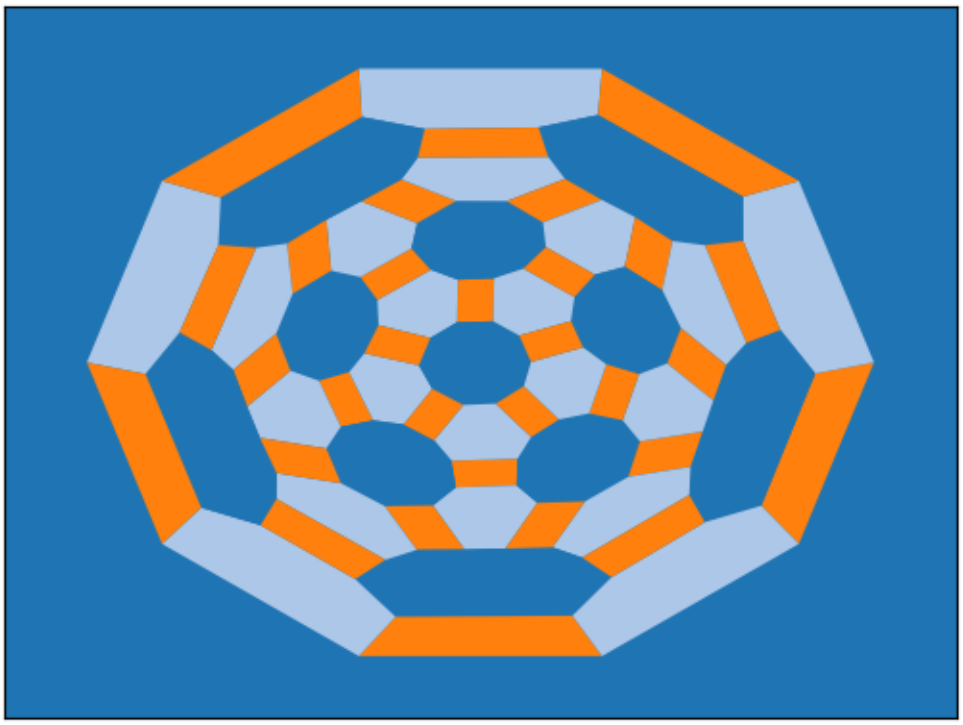
```
Colors = 4
```



```
Colors = 2
```



```
Colors = 3
```



The following code contains a function for randomly generating a planar graph. Specifically, it randomly places $n$ nodes into the unit square and then forms a Delaunay triangulation among these. In the images below, planar graphs with $n \in \{100, 250, 500, 1000\}$ nodes are generated in turn, and their faces are then colored.

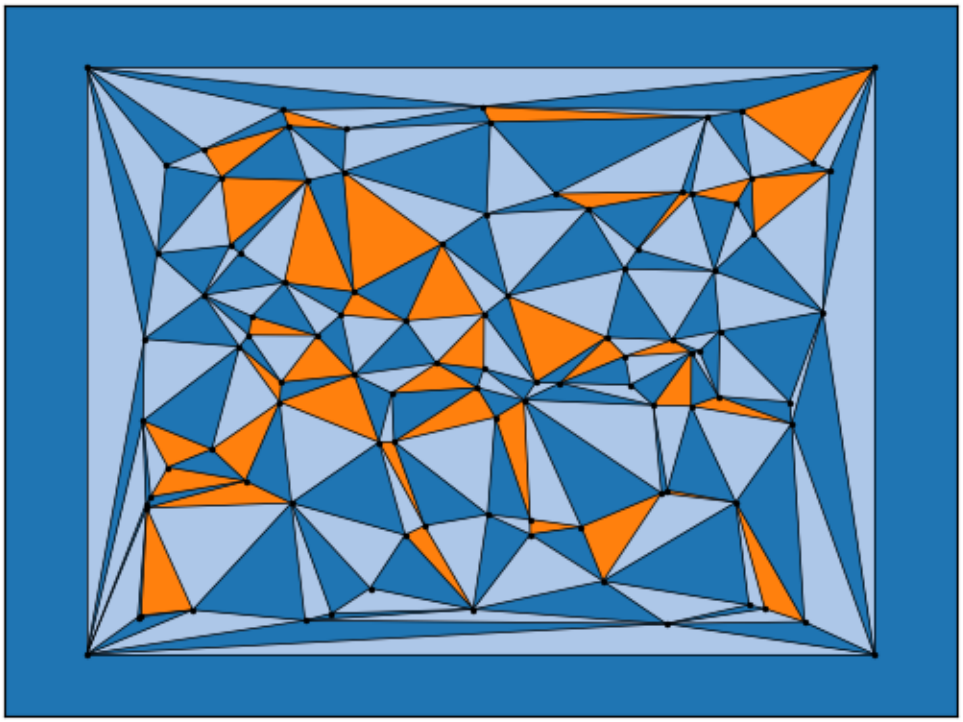
```
def make_planar_graph(n, seed=None):
    # Function for making a dense planar graph by placing nodes randomly
    # into the unit square, including corners
    assert n >= 4, "n parameter must be at least 4"
    import random
    from scipy.spatial import Delaunay
    random.seed(seed)
    P = [(0,0), (1,0), (0,1), (1, 1)]
    for i in range(4, n):
        P.append((random.uniform(0.05,0.95), random.uniform(0.05,0.95)))
    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

for n in [100, 250, 500, 1000]:
    G = make_planar_graph(n, seed=1)
    pos = nx.get_node_attributes(G, "pos")
    c = gcol.face_coloring(G, pos)
    gcol.draw_face_coloring(c, pos, external=True)
    print("Number of nodes  =", n)
    print("Number of edges  =", G.number_of_edges())
```

(continues on next page)

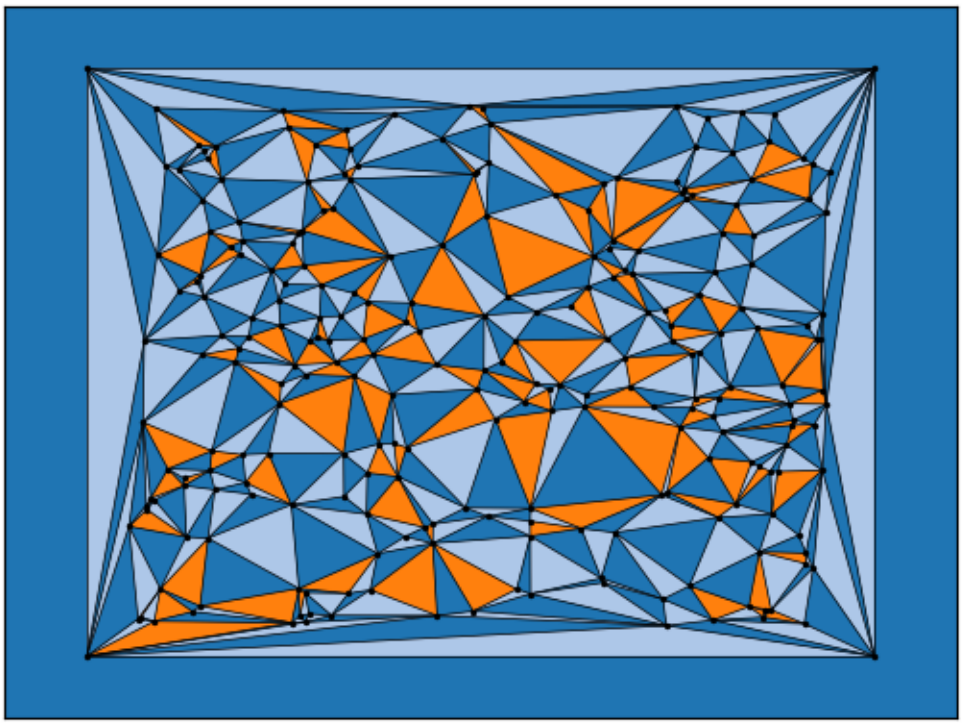
(continued from previous page)

```
    print("Number of faces  =", 2- n + G.number_of_edges())
    print("Number of colors =", max(c.values()) + 1)
    nx.draw_networkx(
        G,
        pos=pos,
        with_labels=False,
        node_size=2,
        node_color="black",
        width=0.5
    )
    plt.show()
```

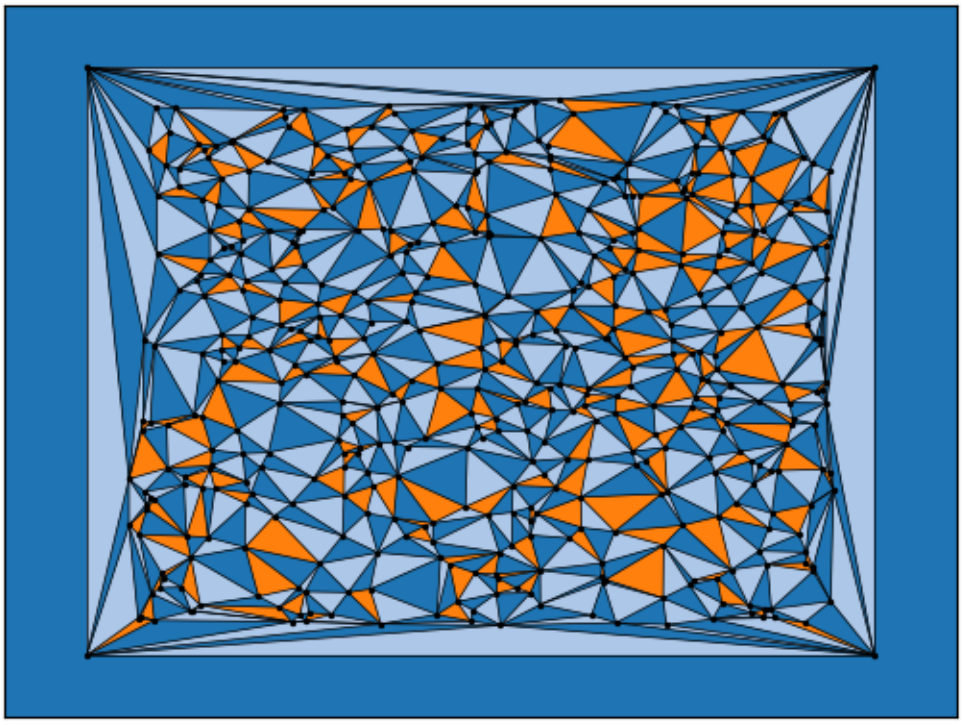
```
Number of nodes  = 100
Number of edges  = 293
Number of faces  = 195
Number of colors = 3
```



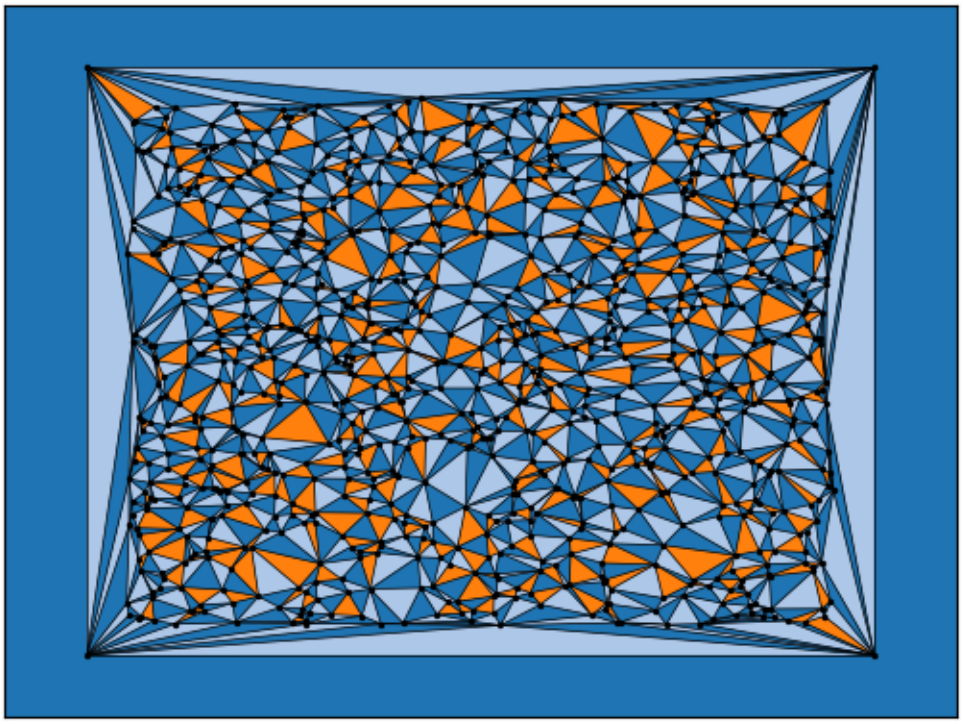
```
Number of nodes  = 250
Number of edges  = 743
Number of faces  = 495
Number of colors = 3
```



```
Number of nodes  = 500
Number of edges  = 1493
Number of faces  = 995
Number of colors = 3
```



```
Number of nodes  = 1000
Number of edges  = 2993
Number of faces  = 1995
Number of colors = 3
```



## 6.6 Interval Graphs

An interval graph is a graph that represents the intersections of intervals on the real line. Formally, each node $v_i$ corresponds to an interval $[a_i, b_i]$ (where $a_i, b_i \in \mathbb{R}$), and two nodes are connected by an edge if and only if their corresponding intervals overlap. Unlike general graphs, interval graphs can be optimally colored in polynomial time. This is done by simply sorting the intervals by their left endpoints and then using the corresponding node ordering with the greedy coloring algorithm.

The following code provides an example. Here, a sorted list `I` of intervals is first generated. These are then used to produce the interval graph `G` which is then optimally colored using the `greedy_color()` method. In the final figure, the corresponding intervals are shown. As required, overlapping intervals always have different colors and the minimum number of colors (six) is being used.

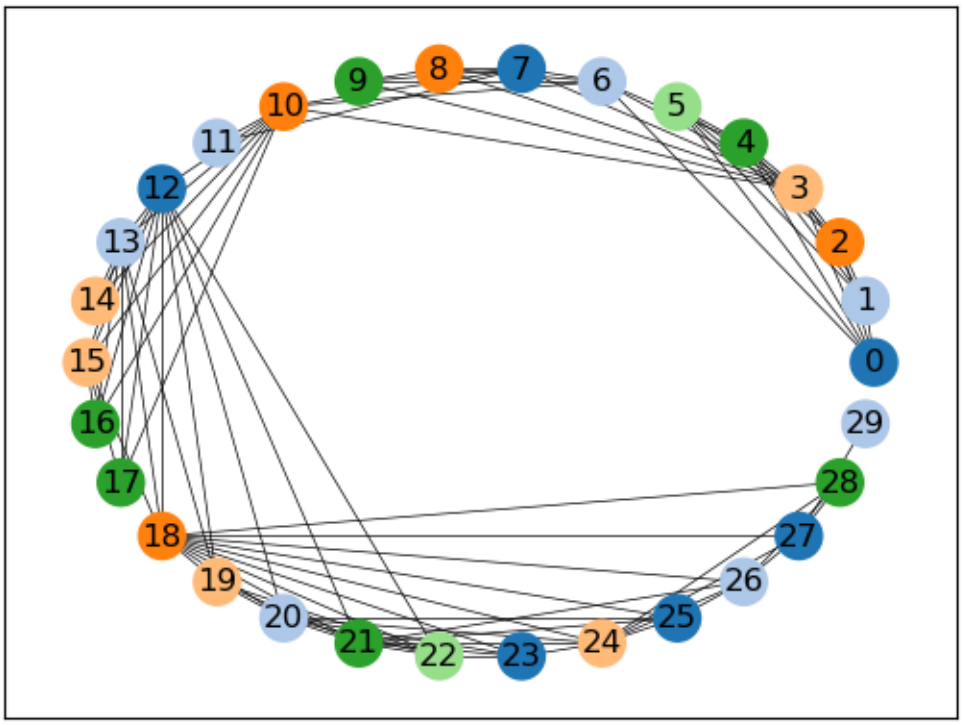
```
import random
import itertools

def greedy_color(G, L):
    # Greedily color the nodes in the order they appear in L
    c = {}
    for u in L:
        adjcols = {c[v] for v in G[u] if v in c}
        for j in itertools.count():
            if j not in adjcols:
```

(continues on next page)

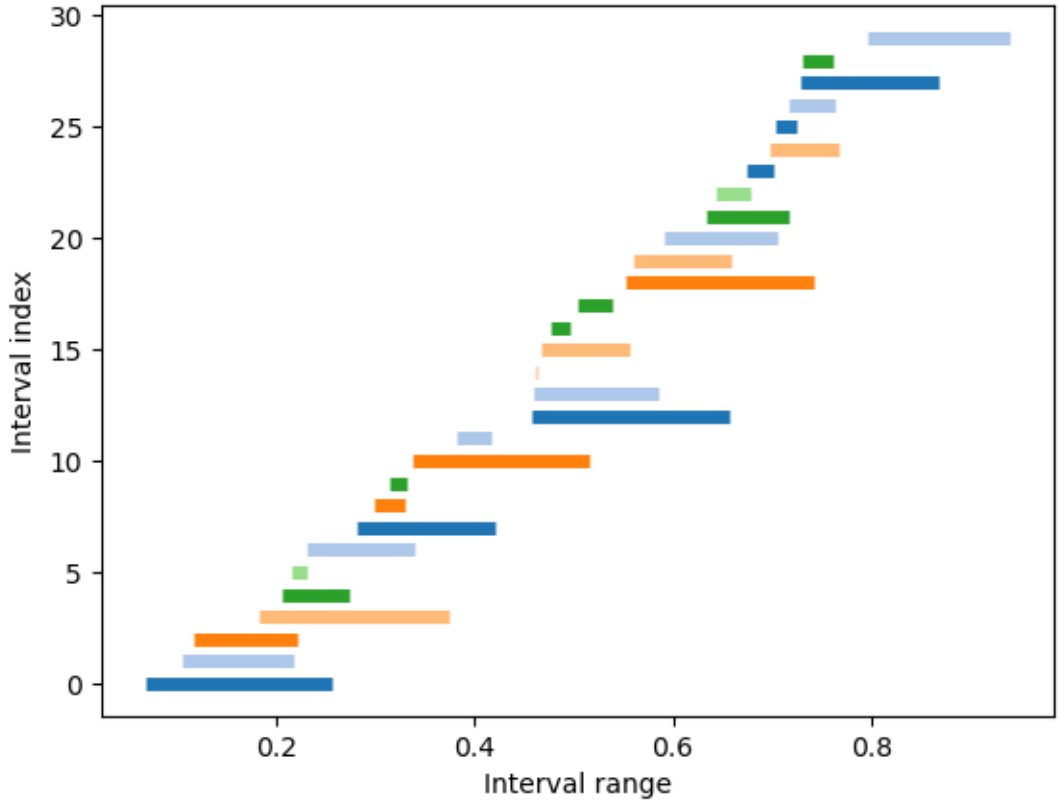
(continued from previous page)

```
                break
        c[u] = j
    return c

# Generate a list of random intervals sorted by starting value
random.seed(1)
n, I = 30, []
for i in range(n):
    x = random.uniform(0.2, 0.8)
    a, b = sorted([x, x + random.uniform(-0.2, 0.2)])
    I.append((round(a, 3), round(b, 3)))
I.sort(key=lambda x: x[0])
print("The generated intervals are as follows:", I)

print("This gives the following interval graph and coloring:")
G = nx.Graph()
for i in range(n):
    G.add_node(i)
for i in range(n-1):
    for j in range(i+1, n):
        if max(I[i][0], I[j][0]) < min(I[i][1], I[j][1]):
            G.add_edge(i, j)
c = greedy_color(G, [i for i in range(n)])
nx.draw_networkx(
    G,
    pos=nx.circular_layout(G),
    width=0.5,
    node_color=gcol.get_node_colors(G, c, gcol.tableau)
)
plt.show()

print("Here is the corresponding interval coloring:")
for i in range(n):
    plt.hlines(y=i, xmin=I[i][0], xmax=I[i][1], colors=gcol.tableau[c[i]], linewidths=5)
plt.xlabel("Interval range")
plt.ylabel("Interval index")
plt.show()
print("Number of colors =", max(c.values()) + 1)
```

```
The generated intervals are as follows: [(0.068, 0.256), (0.106, 0.217), (0.118, 0.221),␣
↪(0.182, 0.374), (0.206, 0.273), (0.215, 0.232), (0.232, 0.34), (0.281, 0.42), (0.299,␣
↪0.33), (0.315, 0.331), (0.337, 0.515), (0.382, 0.417), (0.458, 0.657), (0.46, 0.585),␣
↪(0.461, 0.463), (0.467, 0.556), (0.477, 0.497), (0.503, 0.539), (0.553, 0.741), (0.56,␣
↪0.658), (0.591, 0.706), (0.633, 0.717), (0.644, 0.678), (0.675, 0.701), (0.698, 0.766),
↪ (0.703, 0.725), (0.716, 0.763), (0.729, 0.868), (0.731, 0.762), (0.796, 0.94)]
This gives the following interval graph and coloring:
```




```
Here is the corresponding interval coloring:
```



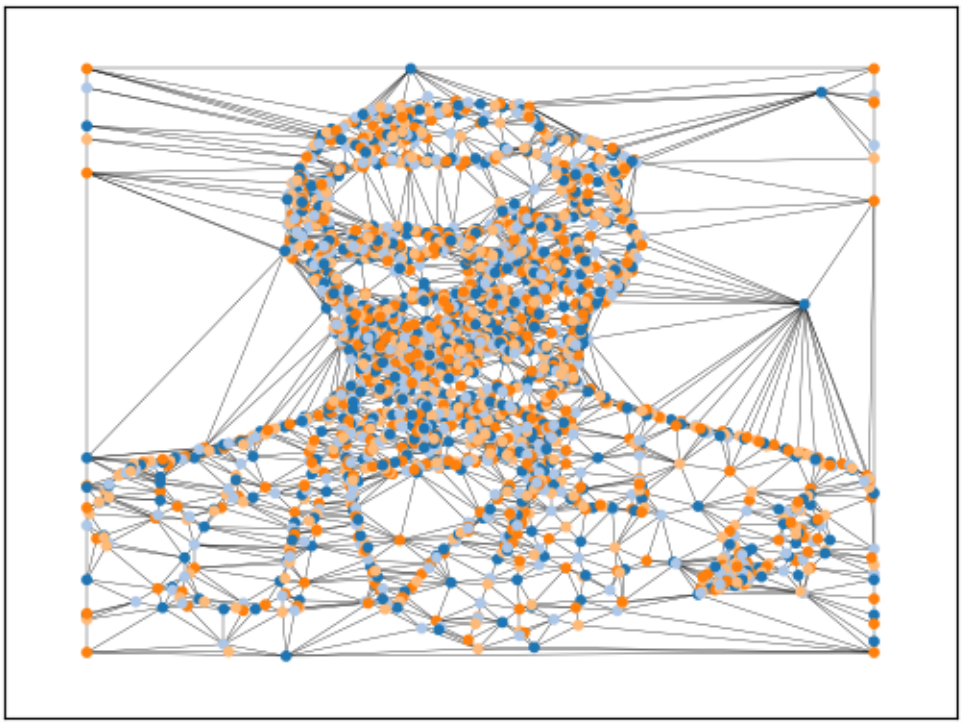
```
Number of colors = 6
```

## 6.7 Triangulations of Images

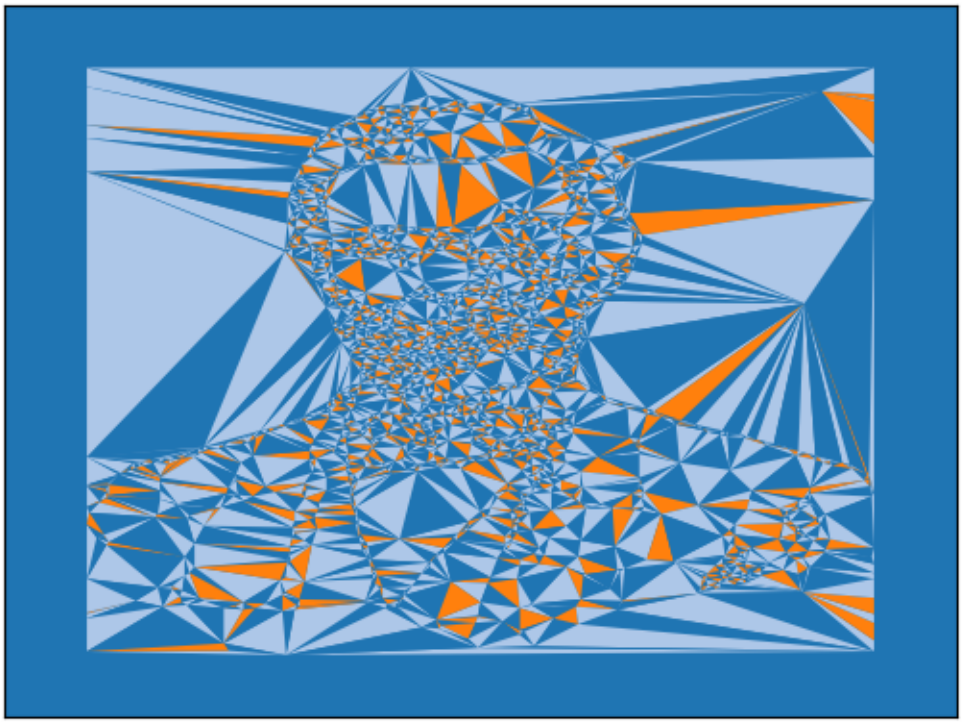
The next two examples consider Delaunay triangulations generated from images. These triangulations were generated using the tool at this website and correspond to planar embeddings. The first image, Lincoln.txt is a portrait of Abraham Lincoln; the second, flag.txt, is a picture of the flag of Wales.

```
files = ['Lincoln.txt', 'flag.txt']

for file in files:
    G = graphFromFile(file)
    pos = nx.get_node_attributes(G, "pos")
    c = gcol.node_coloring(G, opt_alg=3, it_limit=100000)
    print("Colors =", max(c.values()) + 1)
    nx.draw_networkx(
        G,
        pos=pos,
        node_color=gcol.get_node_colors(G, c),
        with_labels=False,
        width=0.25,
        node_size=10
    )
    plt.show()

    c = gcol.face_coloring(G, pos, opt_alg=3, it_limit=100000)
    print("Colors =", max(c.values()) + 1)
    gcol.draw_face_coloring(c, pos, external=True)
    nx.draw_networkx(
        G,
        pos=pos,
        node_color='k',
        node_size=0,
        width=0,
        with_labels=False
    )
    plt.show()
```

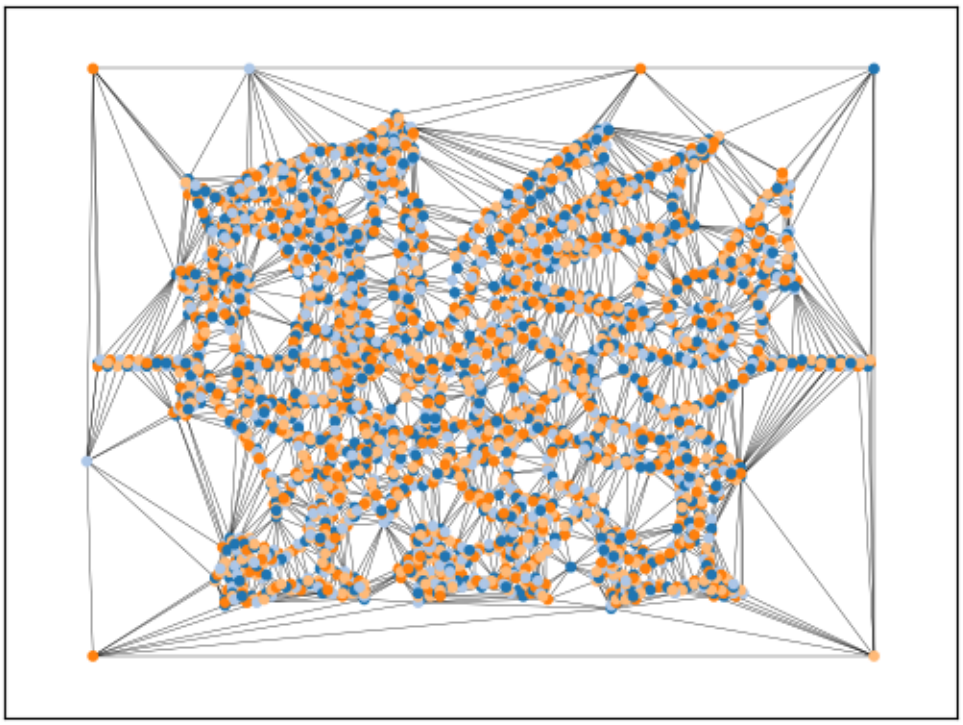
```
Colors = 4
```



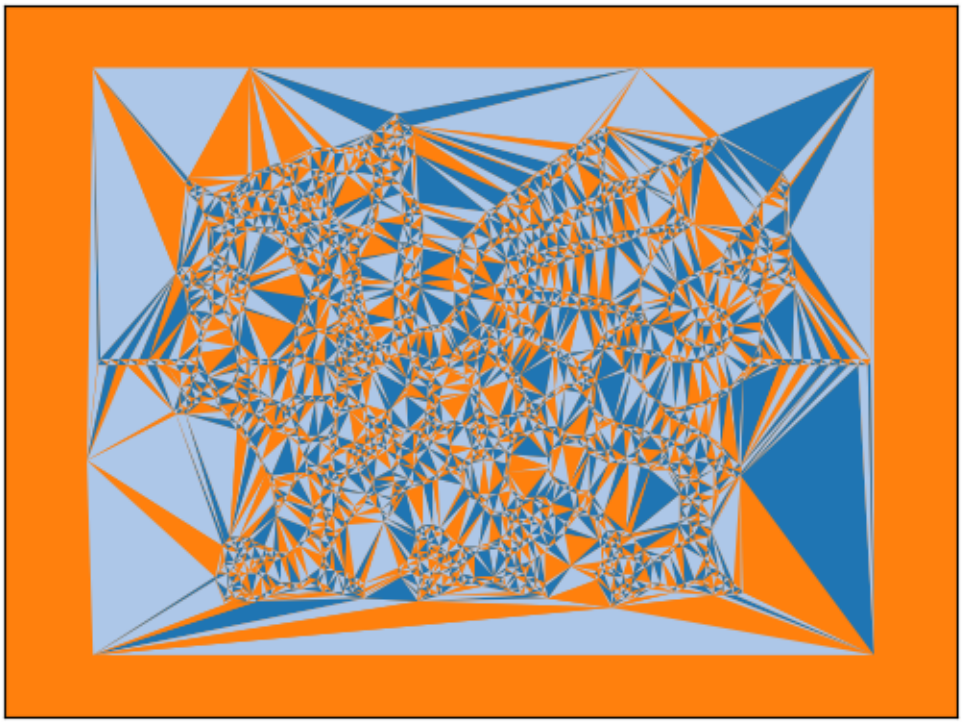
```
Colors = 3
```



```
Colors = 4
```



```
Colors = 3
```

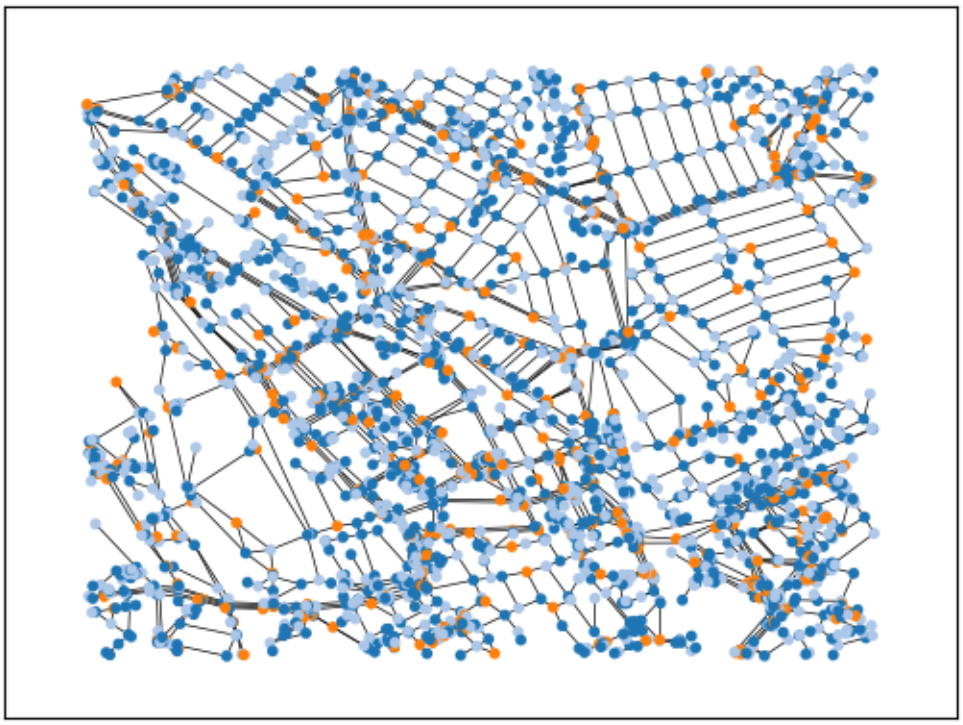


## 6.8 Coloring Street Maps

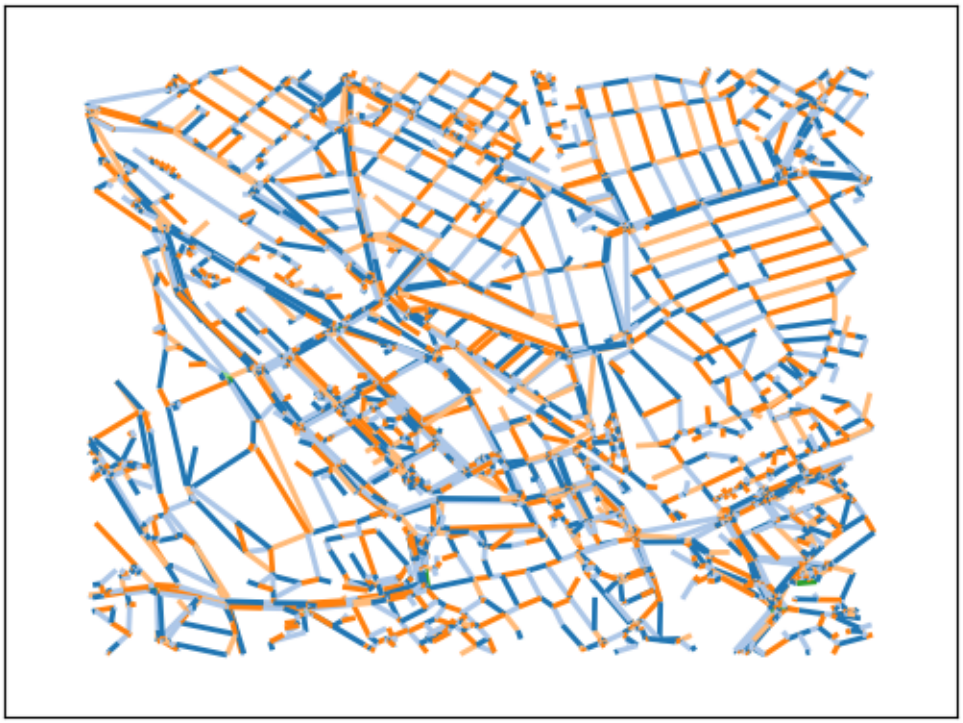
The following images show a node coloring and edge coloring of the street map of Cardiff, Wales. In these graphs, edges correspond to street segments, and nodes correspond to intersections and dead ends. The file used for these images, cardiffstreets.txt, was generated using the osmnx library. An edge coloring of a street map is useful in that any path in the graph can be described by just two things: the starting node, and a unique sequence of colors representing each successive edge along the path.

```
G = graphFromFile('cardiffstreets.txt')
pos = nx.get_node_attributes(G, "pos")
c = gcol.node_coloring(G)
print("Colors =", max(c.values()) + 1)
nx.draw_networkx(
    G,
    pos=pos,
    node_color=gcol.get_node_colors(G, c),
    with_labels=False,
    width=0.5,
    node_size=10
)
plt.show()

c = gcol.edge_coloring(G)
print("Colors =", max(c.values()) + 1)
nx.draw_networkx(
    G,
    pos=pos,
    edge_color=gcol.get_edge_colors(G, c),
    node_size=0,
    width=2,
    with_labels=False
)
plt.show()
```

```
Colors = 3
```



```
Colors = 6
```

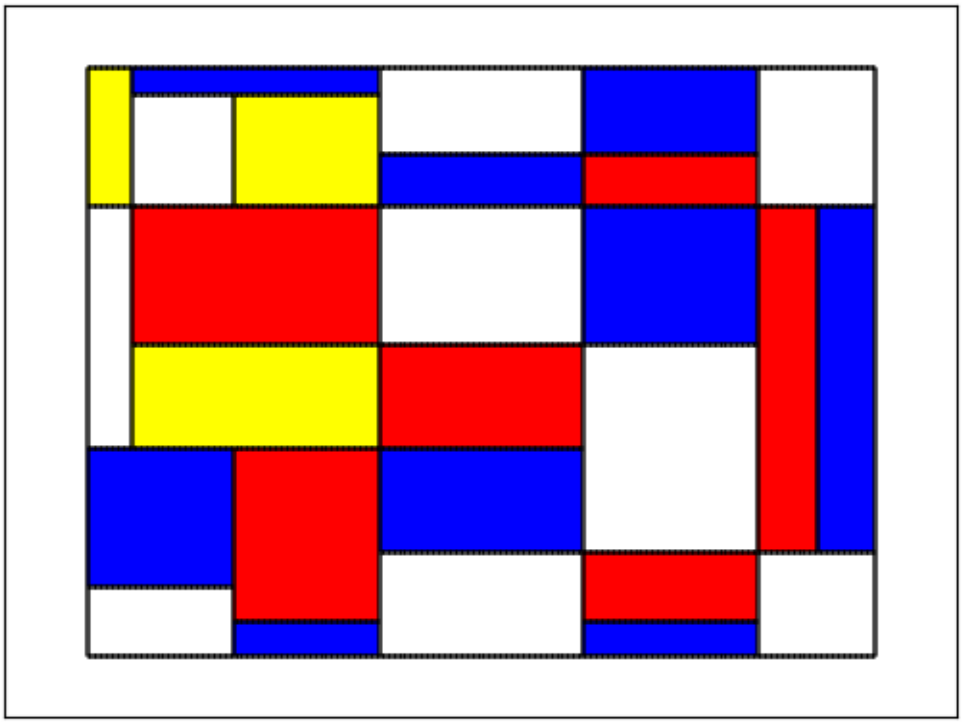


## 6.9 Mondrian Art

The following code generates a mock-up of Piet Mondrian's *Composition A* (1923). The planar embedding of this graph can be found here.

```
G = graphFromFile('mondrian.txt')
pos = nx.get_node_attributes(G, "pos")
c = gcol.face_coloring(G, pos)
for u in c:
    if c[u] == 1:
        c[u] = -1
print("Colors =", max(c.values()) + 1)
gcol.draw_face_coloring(c, pos, palette=gcol.colorful)
nx.draw_networkx(
    G,
    pos=pos,
    node_size=0,
    width=2,
    with_labels=False
)
plt.show()
```

```
Colors = 4
```



## 6.10 An April Fool's Joke

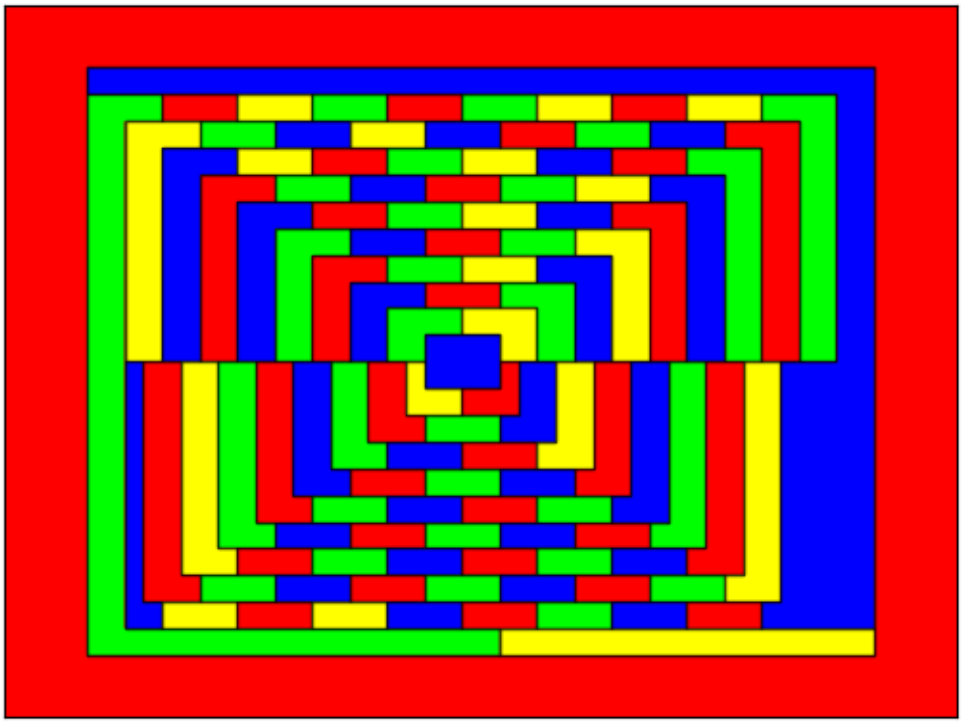
In 1975, Martin Gardner published an April Fool's article in *Scientific American* entitled "Mathematical Games: Six Sensational Discoveries that Somehow or Another have Escaped Public Attention". One of the many deliberately false

claims made in the article was that the faces of the following graph could not be four-colored, therefore disproving the four color theorem.

Of course, like all planar embeddings, a four coloring *is* possible in this case, as we now demonstrate.

```
G = graphFromFile('mcgregorgraph.txt')
pos = nx.get_node_attributes(G, "pos")
c = gcol.face_coloring(G, pos, opt_alg=1)
gcol.draw_face_coloring(
    c, pos, external=True, palette=gcol.colorful
)
print("Colors =", max(c.values()) + 1)
nx.draw_networkx(
    G,
    pos=pos,
    node_size=0,
    width=1,
    with_labels=False
)
plt.show()
```

```
Colors = 4
```



CHAPTER

SEVEN

# DOCUMENTATION

The following methods are available in the `gcol` graph coloring library. All methods containing the word `color` in their name can also be invoked using the alternative spelling `colour`.

## 7.1 Edge Coloring

Edge coloring functions.

`gcol.edge_coloring.`**`chromatic_index`**`(`$G$`)`

Return 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.

If `G` contains any self-loops.

**See also**

*`gcol.node_coloring.chromatic_number()`*
*`gcol.node_coloring.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.edge_coloring.**edge_coloring**(*G*, *strategy='dsatur'*, *opt_alg=None*, *it_limit=0*, *verbose=0*)

Return 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 four 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.
- `4` : A hybrid evolutionary algorithm (HEA) that evolves a small population of solutions. During execution, when each new solution is created, the local search method used in Option `2` above is applied for a fixed number of iterations. Each iteration of this HEA therefore has a complexity of $O(m + kn)$, as above.
- `5` : A hybrid evolutionary algorithm is applied (as above), using the local search method from Option `3`.
- `None` : No optimization is performed.

Further details of these algorithms are given in the notes section of the `node_coloring()` method.

**it_limit**
[int, optional (default=0)] Number of iterations of the local search procedure. Not applicable when using `opt_alg=1`.

**verbose**
[int, optional (default=0)] If set to a positive value, information is output during the optimization process. The higher the value, the more information.

**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.

If `G` contains any self-loops.

**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 `verbose` is not a nonnegative integer.

**See also**

*chromatic_index*
*edge_k_coloring*
*gcol.node_coloring.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_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, ..., (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, ..., (7, 14): 2}
>>>
>>> print("Number of colors =", max(c.values()) + 1)
Number of colors = 3
```

`gcol.edge_coloring.`**`edge_k_coloring`**(*G*, *k*, *opt_alg=None*, *it_limit=0*, *verbose=0*)

Attempt to color the edges of a graph `G` using `k` colors.

This is done 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 four 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.
- `4` : A hybrid evolutionary algorithm (HEA) that evolves a small population of solutions. During execution, when each new solution is created, the local search method used in Option `2` above is applied for a fixed number of iterations. Each iteration of this HEA therefore has a complexity of $O(m + kn)$, as above.
- `5` : A hybrid evolutionary algorithm is applied (as above), using the local search method from Option `3`.
- `None` : No optimization is performed.

Further details of these algorithms are given in the notes section of the `node_coloring()` method.

**it_limit**
[int, optional (default=0)] Number of iterations of the local search procedure. Not applicable when using `opt_alg=1`.

**verbose**
[int, optional (default=0)] If set to a positive value, information is output during the optimization process. The higher the value, the more information.

**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.

If `G` contains any self-loops.

**ValueError**
If `opt_alg` is not among the supported options.

If `it_limit` is not a nonnegative integer.

If `verbose` 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*
*gcol.node_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()
>>> c = gcol.edge_k_coloring(G, 4)
>>> print(c)
{(11, 12): 0, (11, 18): 1, (10, 11): 2, ..., (6, 7): 2}
>>>
>>> c = gcol.edge_k_coloring(G, 3)
```

(continues on next page)

(continued from previous page)

```
>>> print(c)
{(11, 12): 0, (11, 18): 1, (10, 11): 2, ..., (7, 8): 0}
```

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

Return a solution to the edge list coloring problem on `G`.

In the edge list coloring problem, each edge $\{u, v\}$ is associated with a list of allowed colors $L(\{u, v\})$. A solution is an assignment of colors to all edges so that adjacent edges have different colors, and the color of each edge $\{u, v\}$ belongs to the list $L(\{u, v\})$.

The list coloring problem is NP-hard. This method therefore includes options for using an exponential-time exact algorithm (based on backtracking), or a choice of four 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, colors are defined using labels belonging to $\{0, 1, 2, \ldots\}$. Assuming $k$ is the largest color label appearing across all lists, the aim is to find a solution using at most $k$ colors. If this cannot be achieved, an exception is raised (see below).

Here, edge colorings of a graph $G$ are determined by forming $G$'s line graph $L(G)$ and then passing $L(G)$ to the `node_list_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.

**allowed_cols**
: [None or dict, optional (default=None)] A dictionary keyed by the edges of `G`. `allowed_cols[(u, v)]` should be a list of (nonnegative integer) colors that give the permissible colors for edge `(u, v)`. If None, an edge coloring with the minimum number of colors is returned.

**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.

  - `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.
- `4` : A hybrid evolutionary algorithm (HEA) that evolves a small population of solutions. During execution, when each new solution is created, the local search method used in Option `2` above is applied for a fixed number of iterations. Each iteration of this HEA therefore has a complexity of $O(m + kn)$, as above.
- `5` : A hybrid evolutionary algorithm is applied (as above), using the local search method from Option `3`.
- `None` : No optimization is performed.

Further details of these algorithms are given in the notes section of the `node_coloring()` method.

**it_limit**
[int, optional (default=0)] Number of iterations of the local search procedure. Not applicable when using `opt_alg=1`.

**verbose**
[int, optional (default=0)] If set to a positive value, information is output during the optimization process. The higher the value, the more information.

**Returns**

**dict**
A dictionary with keys representing edges and values representing their colors. Colors are identified by the integers $0, 1, 2, \ldots$. If `c[(u,v)]==j` then `j` is an element of `allowed_cols[(u, v)]`.

**Raises**

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

If `G` contains any self-loops.

**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 `verbose` is not a nonnegative integer.

If `allowed_cols` has an edge not in G, or is missing an entry for an edge in `G`.

If `allowed_cols` contains a color label that is not a nonnegative integer.

If `allowed_cols` contains a list that is empty.

If `allowed_cols` contains a pair of adjacent edges that have the same single allowed color.

If `allowed_cols` contains entries for the same edges $(u, v)$ and $(v, u)$.

If an edge list coloring could not be determined (the lists of allowed colors are too restrictive).

**TypeError**
If `allowed_cols` is not a dict.

**See also**

*gcol.node_coloring.node_coloring()*
*gcol.edge_coloring.edge_precoloring()*
*gcol.node_coloring.node_list_coloring()*

### Notes

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.cycle_graph(4)
>>> E = {(0, 1): [0, 1], (1, 2): [1, 2], (2, 3): [1], (3, 0): [0, 1]}
>>> c = gcol.edge_list_coloring(G, E)
>>> print("Coloring is", c)
Coloring is {(0, 3): 0, (2, 3): 1, (0, 1): 1, (1, 2): 0}
```

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

Return a coloring of a graph's edges where some edges are precolored.

An edge coloring of a graph is an assignment of colors to edges so that adjacent edges have different colors. In the edge precoloring problem, some of the edges have already been assigned colors. Colors are defined using labels belonging to the set $\{0, 1, 2, \ldots\}$. Assuming $k$ is the largest color label used in the precoloring, the aim is to color all remaining edges using $l$ colors, where $l$ is minimized and $k \leq l$.

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 four 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.
- `4` : A hybrid evolutionary algorithm (HEA) that evolves a small population of solutions. During execution, when each new solution is created, the local search method used in Option `2` above is applied for a fixed number of iterations. Each iteration of this HEA therefore has a complexity of $O(m + kn)$, as above.
- `5` : A hybrid evolutionary algorithm is applied (as above), using the local search method from Option `3`.
- `None` : No optimization is performed.

Further details of these algorithms are given in the notes section of the `node_coloring()` method.

**it_limit**
[int, optional (default=0)] Number of iterations of the local search procedure. Not applicable when using `opt_alg=1`.

**verbose**
[int, optional (default=0)] If set to a positive value, information is output during the optimization process. The higher the value, the more information.

**Returns**

**dict**
A dictionary with keys representing edges and values representing their colors. Colors are identified by the integers $0, 1, 2, \ldots$. If `precol[(u,v)]==j` then `c[(u,v)]==j`.

**Raises**

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

If `G` contains any self-loops.

**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 `verbose` is not a nonnegative integer.

If `precol` contains an edge that is not in `G`.

If `precol` contains a color label that is not a nonnegative integer.

If `precol` contains a pair of adjacent edges assigned to the same color.

If `precol` contains entries for the same edges (u, v) and (v, u).

**TypeError**

If `precol` is not a dict.

**See also**

*edge_coloring*
*gcol.node_coloring.node_precoloring()*
*gcol.node_coloring.node_coloring()*

### Notes

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, ..., (5, 6): 2}
```

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

Attempt to color the edges of a graph using `k` colors.

This is done 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 standard deviation 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.
  - `4` : A hybrid evolutionary algorithm (HEA) that evolves a small population of solutions. During execution, when each new solution is created, the local search method used in Option `2` above is applied for a fixed number of iterations. Each iteration of this HEA therefore has a complexity of $O(m + kn)$, as above.
  - `5` : A hybrid evolutionary algorithm is applied (as above), using the local search method from Option `3`.
  - `None` : No optimization is performed.

  Further details of these algorithms are given in the notes section of the `node_coloring()` method.

**it_limit**
: [int, optional (default=0)] Number of iterations of the local search procedure. Not applicable when using `opt_alg=1`.

**verbose**
[int, optional (default=0)] If set to a positive value, information is output during the optimization process. The higher the value, the more information.

**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.

If `G` contains any self-loops.

**ValueError**
If `opt_alg` is not among the supported options.

If `it_limit` is not a nonnegative integer.

If `verbose` 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*
*gcol.node_coloring.node_k_coloring()*
*gcol.node_coloring.equitable_node_k_coloring()*
*gcol.node_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 standard deviation 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), ..., (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), ..., (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
```

## 7.2 Face Coloring

Face coloring functions.

gcol.face_coloring.**dual_graph**(*G*, *pos*)

Return the dual graph of the specified planar embedding of G.

The dual graph $G^*$ of a planar graph $G$ has a node for each face in the supplied embedding of $G$, including the external face. Each edge in $G^*$ corresponds to a pair of faces in $G$ that share a boundary. According to Euler's formula, $G$ has $f = m-n+2$ faces, where $n$ is its number of nodes and $m$ is the number of edges. Consequently, $G^*$ has $f$ nodes.

The graph G must be planar. The embedding is defined by pos, which should give valid $(x, y)$ coordinates for all nodes in G so that none of its edges (expressed as straight lines) cross. G should also be connected and have no bridges.

**Parameters**

**G**
[NetworkX graph] A bridge-free, connected planar graph.

**pos**
[dict] A dictionary of positions keyed by node. All positions should be $(x, y)$ coordinates, and none of the edges in the resultant embedding should be crossing.

**Returns**

**NetworkX graph**
The dual graph in which nodes are labelled using integers from 0 upwards. Node 0 corresponds to the single external face of `G`; the remainder correspond to the internal faces.

**list**
The $i$th element in the list is a tuple giving the sequence of nodes that surround the $i$th face in `G`. This face corresponds to node $i$ in the returned dual graph. The first element in this list defines the external face; the remainder define the internal faces. For internal faces, the nodes are listed in a counterclockwise order. For the external face, the nodes are listed in a clockwise order.

**Raises**

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

If `G` contains any self-loops.

If `G` is not connected or is a singleton.

**TypeError**
If `pos` is not a dictionary.

**ValueError**
If `pos` has missing or invalid entries.

If `pos` does not specify a valid planar embedding of `G`.

If `G` is not planar or contains bridges.

**See also**

*face_chromatic_number*
*face_coloring*

### Notes

The structure of a dual graph $G^*$ depends on the coordinates of the nodes in $G$. That is, different planar embeddings of $G$ can lead to different dual graphs.

If the graph `G` contains no articulation points, all faces will be cycles. If `G` does contain any articulation points, these can occur more than once on the boundary of a face; hence, some faces may be circuits.

At present, this method does not allow self-loops, disconnected graphs (including faces with holes), or multi-edges. Multi-edges can be simulated by using paths of degree-two nodes with suitable coordinates.

### Examples

```
>>> import gcol
>>> import networkx as nx
>>>
>>> G = nx.dodecahedral_graph()
>>> pos = nx.planar_layout(G)
>>> H, faces = gcol.dual_graph(G, pos)
>>> print(H)
Graph with 12 nodes and 30 edges
>>> print(faces)
[(0, 10, 9, 8, 1), ..., (13, 12, 16, 15, 14)]
```

gcol.face_coloring.**equitable_face_k_coloring**(*G*, *pos*, *k*, *opt_alg=None*, *it_limit=0*, *verbose=0*)

Attempt to color the faces of a planar graph `G` using `k` colors.

This is done so that (a) adjacent faces have different colors, and (b) the number of faces with each color is equal. (A pair of faces is adjacent if and only if they share a bordering edge.) The graph `G` must be planar, and `pos` should give valid $(x, y)$ coordinates for all nodes.

This method first follows the steps used by the *face_k_coloring()* method. All parameters are therefore the same as the latter. If a face $k$-coloring cannot be determined by the algorithm, a `ValueError` exception is raised. Otherwise, once a face $k$-coloring has been formed, the algorithm uses a bespoke local search operator to reduce the standard deviation in the number of faces in each color class.

In solutions returned by this method, adjacent faces always receive different colors; however, the coloring is not guaranteed to be equitable, even if an equitable face $k$-coloring exists.

According to the four color theorem [1], face colorings never require more than four colors. In this implementation, face colorings of a graph $G$ are determined by forming $G$'s dual graph.

**Parameters**

**G**
[NetworkX graph] A bridge-free, connected planar graph. Its faces will be colored.

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

**pos**
[dict] A dictionary of positions keyed by node. All positions should be $(x, y)$ coordinates, and none of the edges in the resultant embedding should be crossing.

**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 the dual to have the same color. Each iteration has a complexity $O(m + kn)$, where $n$ is the number of nodes in the dual, $m$ is the number of edges in the dual, 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 the dual to be uncolored. Each iteration has a complexity $O(m + kn)$, as above.

- `4` : A hybrid evolutionary algorithm (HEA) that evolves a small population of solutions. During execution, when each new solution is created, the local search method used in Option `2` above is applied for a fixed number of iterations. Each iteration of this HEA therefore has a complexity of $O(m + kn)$, as above.
- `5` : A hybrid evolutionary algorithm is applied (as above), using the local search method from Option `3`.
- `None` : No optimization is performed.

Further details of these algorithms are given in the notes section of the `node_coloring()` method.

**it_limit**
[int, optional (default=0)] Number of iterations of the local search procedure. Not applicable when using `opt_alg=1`.

**verbose**
[int, optional (default=0)] If set to a positive value, information is output during the optimization process. The higher the value, the more information.

**Returns**

**dict**
A dictionary with keys representing faces 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`. Each face (key) is a tuple that gives the sequence of nodes that surround it. The first element in `c` defines the external face; the remainder defines the internal faces. For internal faces, the nodes are listed in a counterclockwise order; for the external face, the nodes are listed in a clockwise order.

**Raises**

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

If `G` contains any self-loops.

If `G` is not connected or is a singleton.

**TypeError**
If `pos` is not a dictionary.

**ValueError**
If `opt_alg` is not among the supported options.

If `it_limit` is not a nonnegative integer.

If `verbose` is not a nonnegative integer.

If `pos` has missing or invalid entries.

If `pos` does not specify a valid planar embedding of `G`.

If `G` is not planar or contains bridges.

If `k` is not a nonnegative integer.

If a clique larger than `k` is observed in the dual graph of $G$.

If a face $k$-coloring could not be determined.

**See also**

*face_coloring*
*gcol.node_coloring.equitable_node_k_coloring()*

### Notes

As mentioned, in this implementation face colorings of a graph $G$ are determined by forming its dual and then passing this to the `node_k_coloring()` method. If a face $k$-coloring is achieved, a bespoke local search operator (based on steepest descent) is then used to try to reduce the standard deviation in sizes across the $k$ color classes. This follows the same steps as the `equitable_node_k_coloring()` method, using the dual of `G`. Further details on this optimization method can be found in Chapter 7 of [2].

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 gcol
>>> import networkx as nx
>>>
>>> G = nx.dodecahedral_graph()
>>> pos = nx.planar_layout(G)
>>> c = gcol.equitable_face_k_coloring(G, pos, 5)
>>> print(c)
{(1, 0, 10, 9, 8): 0, (10, 0, 19, 18, 11): 2, ..., (13, 12, 16, 15, 14): 0}
```

gcol.face_coloring.**face_chromatic_number**($G$)

Return the face chromatic number of the planar graph `G`.

The face chromatic number of a planar graph $G$ is the minimum number of colors needed to color its faces so that no two adjacent faces have the same color (a pair of faces is considered adjacent if and only if they share a common bordering edge). According to the four color theorem [1], it will never exceed four.

The approach used here is based on the backtracking algorithm of [2]. This is exact but operates in exponential time in the worst case. In this implementation, the solution is found for $G$ by determining a planar embedding, forming the dual graph, and then passing this to the `chromatic_number()` method.

**Parameters**

**G**
[NetworkX graph] The face chromatic number for this graph will be calculated. It must be bridge-free, connected, and planar.

**Returns**

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

**Raises**

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

If `G` contains any self-loops.

If `G` is not connected or is a singleton.

**ValueError**
If `G` is not planar or has bridges.

**See also**

*face_coloring*
*gcol.node_coloring.chromatic_number()*

### 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 face 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 [2]. The c++ code used in [2] and [3] forms the basis of this library's Python implementations.

### References

[1], [2], [3]

### Examples

```
>>> import gcol
>>> import networkx as nx
>>>
>>> G = nx.dodecahedral_graph()
>>> print(gcol.face_chromatic_number(G))
4
```

`gcol.face_coloring.`**face_coloring**(*G*, *pos*, *strategy='dsatur'*, *opt_alg=None*, *it_limit=0*, *verbose=0*)

Return a coloring of a planar graph's faces.

A face coloring is an assignment of colors to the faces of a graph's planar embedding so that adjacent faces have different colors (a pair of faces is adjacent if and only if they share a bordering edge). The aim is to use as few colors as possible. Face colorings are only possible in planar embeddings; hence the graph `G` must be planar, and `pos` should give valid $(x, y)$ coordinates for all nodes.

The smallest number of colors needed to face color a graph is known as its face chromatic number. According to the four color theorem [1], face colorings never require more than four colors. In this implementation, face colorings of a graph $G$ are determined by forming $G$'s dual graph, and then coloring the dual's nodes using the `node_coloring()` method. All parameters are therefore the same as the latter. If $G$ has $n$ nodes and $m$ edges, its embedding has exactly $m - n + 2$ faces, including the external face.

**Parameters**

**G**
[NetworkX graph] A bridge-free, connected planar graph. Its faces will be colored.

**pos**
[dict] A dictionary of positions keyed by node. All positions should be $(x, y)$ coordinates, and none of the edges in the resultant embedding should be crossing.

**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 dual's nodes and then applies the greedy algorithm for graph node coloring [2].
- `'welsh-powell'` : Orders the dual's nodes by decreasing degree, then applies the greedy algorithm.
- `'dsatur'` : Uses the DSatur algorithm for graph node coloring on the dual [3].
- `'rlf'` : Uses the recursive largest first (RLF) algorithm for graph node coloring on the dual [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 the dual to have the same color. Each iteration has a complexity $O(m + kn)$, where $n$ is the number of nodes in the dual, $m$ is the number of edges in the dual, 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 the dual to be uncolored. Each iteration has a complexity $O(m + kn)$, as above.
- `4` : A hybrid evolutionary algorithm (HEA) that evolves a small population of solutions. During execution, when each new solution is created, the local search method used in Option `2` above is applied for a fixed number of iterations. Each iteration of this HEA therefore has a complexity of $O(m + kn)$, as above.
- `5` : A hybrid evolutionary algorithm is applied (as above), using the local search method from Option `3`.
- `None` : No optimization is performed.

Further details of these algorithms are given in the notes section of the `node_coloring()` method.

**it_limit**
[int, optional (default=0)] Number of iterations of the local search procedure. Not applicable when using `opt_alg=1`.

**verbose**
[int, optional (default=0)] If set to a positive value, information is output during the optimization process. The higher the value, the more information.

**Returns**

**dict**
A dictionary with keys representing faces 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`. Each face (key) is a tuple that gives the sequence of nodes that surround it. The first element in `c` defines the external face; the remainder defines the internal faces. For internal faces, the nodes are listed in a counterclockwise order; for the external face, the nodes are listed in a clockwise order.

**Raises**

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

If `G` contains any self-loops.

If `G` is not connected or is a singleton.

**TypeError**
If `pos` is not a dictionary.

**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 `verbose` is not a nonnegative integer.

If `pos` has missing or invalid entries.

If `pos` does not specify a valid planar embedding of `G`.

If `G` is not planar or contains bridges.

**See also**

*face_chromatic_number*
*gcol.node_coloring.node_coloring()*

### Notes

As mentioned, in this implementation face colorings of a graph $G$ are determined by forming its dual and then passing this to the `node_coloring()` method. All details are therefore the same as those in the latter method, 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 gcol
>>> import networkx as nx
>>>
>>> G = nx.dodecahedral_graph()
>>> pos = nx.planar_layout(G)
>>> c = gcol.face_coloring(G, pos)
>>> print(c)
{(1, 0, 10, 9, 8): 0, (10, 0, 19, 18, 11): 2, ..., (13, 12, 16, 15, 14): 2}
```

gcol.face_coloring.**face_k_coloring**(*G*, *pos*, *k*, *opt_alg=None*, *it_limit=0*, *verbose=0*)

Attempt to color the faces of a planar graph `G` using `k` colors.

This is done so that adjacent faces have different colors (a pair of faces is adjacent if and only if they share a bordering edge). The graph `G` must be planar, and `pos` should give valid $(x, y)$ coordinates for all nodes.

According to the four color theorem [1], face colorings never require more than four colors. In this implementation, face colorings of a graph $G$ are determined by forming $G$'s dual graph, and then coloring the dual's nodes using the `node_k_coloring()` method. All parameters are therefore the same as the latter. If $G$ has $n$ nodes and $m$ edges, its embedding has exactly $m - n + 2$ faces, including the external face.

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

**Parameters**

**G**
[NetworkX graph] A bridge-free, connected planar graph. Its faces will be colored.

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

**pos**
[dict] A dictionary of positions keyed by node. All positions should be $(x, y)$ coordinates, and none of the edges in the resultant embedding should be crossing.

**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 the dual to have the same color. Each iteration has a complexity $O(m + kn)$, where $n$ is the number of nodes in the dual, $m$ is the number of edges in the dual, 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 the dual to be uncolored. Each iteration has a complexity $O(m + kn)$, as above.
- `4` : A hybrid evolutionary algorithm (HEA) that evolves a small population of solutions. During execution, when each new solution is created, the local search method used in Option `2` above is applied for a fixed number of iterations. Each iteration of this HEA therefore has a complexity of $O(m + kn)$, as above.
- `5` : A hybrid evolutionary algorithm is applied (as above), using the local search method from Option `3`.
- `None` : No optimization is performed.

Further details of these algorithms are given in the notes section of the `node_coloring()` method.

**it_limit**
[int, optional (default=0)] Number of iterations of the local search procedure. Not applicable when using `opt_alg=1`.

**verbose**
[int, optional (default=0)] If set to a positive value, information is output during the optimization process. The higher the value, the more information.

**Returns**

**dict**

A dictionary with keys representing faces 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`. Each face (key) is a tuple that gives the sequence of nodes that surround it. he first element in `c` defines the external face; the remainder defines the internal faces. For internal faces, the nodes are listed in a counterclockwise order; for the external face, the nodes are listed in a clockwise order.

**Raises**

**NotImplementedError**

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

If `G` contains any self-loops.

If `G` is not connected or is a singleton.

**TypeError**

If `pos` is not a dictionary.

**ValueError**

If `opt_alg` is not among the supported options.

If `it_limit` is not a nonnegative integer.

If `verbose` is not a nonnegative integer.

If `pos` has missing or invalid entries.

If `pos` does not specify a valid planar embedding of `G`.

If `G` is not planar or contains bridges.

If `k` is not a nonnegative integer.

If a clique larger than `k` is observed in the dual graph of $G$.

If a face $k$-coloring could not be determined.

**See also**

*face_coloring*
*gcol.node_coloring.node_k_coloring()*

## Notes

As mentioned, in this implementation face colorings of a graph $G$ are determined by forming its dual and then passing this to the `node_k_coloring()` method. All details are therefore the same as those in the latter method, where they are documented more fully. The routine halts immediately once a face $k$-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 gcol
>>> import networkx as nx
>>>
>>> G = nx.dodecahedral_graph()
>>> pos = nx.planar_layout(G)
>>> c = gcol.face_k_coloring(G, pos, 5)
>>> print(c)
{(1, 0, 10, 9, 8): 0, (10, 0, 19, 18, 11): 2, ..., (13, 12, 16, 15, 14): 0}
```

gcol.face_coloring.**face_list_coloring**(*G*, *pos*, *allowed_cols=None*, *strategy='dsatur'*, *opt_alg=None*, *it_limit=0*, *verbose=0*)

Return a solution to the face list coloring problem on `G`.

A face coloring is an assignment of colors to the faces of a graph's planar embedding so that adjacent faces have different colors. In the face list coloring problem, each face $f$ is associated with a list of allowed colors $L(f)$. A solution is an assignment of colors to all faces so that adjacent faces have different colors, and the color of each face $f$ belongs to the list $L(f)$.

Face colorings are only possible in planar embeddings; hence the graph `G` must be planar, and `pos` should give valid $(x, y)$ coordinates for all nodes.

Here, colors are defined using labels belonging to $\{0, 1, 2, \ldots\}$. Assuming $k$ is the largest color label appearing across all lists, the aim is to find a solution using at most $k$ colors. If this cannot be done, an exception is raised (see below).

In this implementation, face colorings of a graph $G$ are determined by forming $G$'s dual graph, and then passing the dual to the `node_list_coloring()` method. All parameters are therefore the same as the latter.

**Parameters**

**G**
: [NetworkX graph] A bridge-free, connected planar graph. Its faces will be colored.

**pos**
: [dict] A dictionary of positions keyed by node. All positions should be $(x, y)$ coordinates, and none of the edges in the resultant embedding should be crossing.

**allowed_cols**
: [None or dict, optional (default=None)] A dictionary, keyed by faces, that specifies the allowed colors of each face. Faces are identified using a tuple of nodes. Each internal face is characterized by the series of nodes that surround it in a counterclockwise direction; the one external face is identified by the series of nodes, traveling in a clockwise direction. Each value in `allowed_cols` should be a list of (nonnegative integer) colors that give the permissible colors for the corresponding face. If None, a face coloring with the minimum number of colors is returned.

**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 dual's nodes and then applies the greedy algorithm for graph node coloring [1].
  - `'welsh-powell'` : Orders the dual's nodes by decreasing degree, then applies the greedy algorithm.
  - `'dsatur'` : Uses the DSatur algorithm for graph node coloring on the dual [2].

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

**opt_alg**
[None or int, optional (default=None)] An integer specifying the optimization method that will be used.

- `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 the dual to have the same color. Each iteration has a complexity $O(m + kn)$, where $n$ is the number of nodes in the dual, $m$ is the number of edges in the dual, 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 the dual to be uncolored. Each iteration has a complexity $O(m + kn)$, as above.
- `4` : A hybrid evolutionary algorithm (HEA) that evolves a small population of solutions. During execution, when each new solution is created, the local search method used in Option `2` above is applied for a fixed number of iterations. Each iteration of this HEA therefore has a complexity of $O(m + kn)$, as above.
- `5` : A hybrid evolutionary algorithm is applied (as above), using the local search method from Option `3`.
- `None` : No optimization is performed.

Further details of these algorithms are given in the notes section of the `node_coloring()` method.

**it_limit**
[int, optional (default=0)] Number of iterations of the local search procedure. Not applicable when using `opt_alg=1`.

**verbose**
[int, optional (default=0)] If set to a positive value, information is output during the optimization process. The higher the value, the more information.

**Returns**

**dict**
A dictionary with keys representing faces and values representing their colors. Colors are identified by the integers $0, 1, 2, \ldots$. Each face (key) is a tuple that gives the sequence of nodes that surround it. The first element in `c` defines the external face; the remainder define the internal faces. For internal faces, the nodes are listed in a counterclockwise order; for the external face, the nodes are listed in a clockwise order. Keys in the dict are rotated so that the node with minimal label appears first.

**Raises**

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

If `G` contains any self-loops.

**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 `verbose` is not a nonnegative integer.

If `allowed_cols` has a face not in the calculated embedding of `G`.

If `allowed_cols` contains a color label that is not a nonnegative integer.

If `allowed_cols` contains a pair of adjacent faces that have the same single allowed color.

If a face list k-coloring could not be determined (the lists of allowed colors are too restrictive).

**TypeError**
If `allowed_cols` is not a dict.

**See also**

*gcol.node_coloring.node_coloring()*
*gcol.node_coloring.node_list_coloring()*
*face_coloring*

### Notes

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.cycle_graph(4)
>>> G.add_edge(0, 2)
>>> pos = {0: (0, 0), 1: (1, 0), 2: (1, 1), 3: (0, 1)}
>>> F = {(0, 3, 2, 1): [0, 1], (0, 2, 3): [0, 2], (2, 0, 1): [0]}
>>> c = gcol.face_list_coloring(G, pos, F)
>>> print(c)
{(0, 3, 2, 1): 1, (0, 2, 3): 2, (0, 1, 2): 0}
```

gcol.face_coloring.**face_precoloring**(*G*, *pos*, *precol=None*, *strategy='dsatur'*, *opt_alg=None*, *it_limit=0*, *verbose=0*)

Give a face coloring of a planar graph where some faces are precolored.

A face coloring is an assignment of colors to the faces of a graph's planar embedding so that adjacent faces have different colors. In the face precoloring problem, some of the faces have already been assigned colors. Colors are defined using labels belonging to the set $\{0, 1, 2, \ldots\}$. Assuming $k$ is the largest color label used in the precoloring, the aim is to color all remaining faces using $l$ colors, where $l$ is minimized and $k \leq l$.

Face colorings are only possible in planar embeddings; hence the graph `G` must be planar, and `pos` should give valid $(x, y)$ coordinates for all nodes.

In this implementation, face colorings of a graph $G$ are determined by forming $G$'s dual graph, and then passing the dual to the `node_precoloring()` method. All parameters are therefore the same as the latter.

**Parameters**

**G**
[NetworkX graph] A bridge-free, connected planar graph. Its faces will be colored.

**pos**
[dict] A dictionary of positions keyed by node. All positions should be $(x, y)$ coordinates, and none of the edges in the resultant embedding should be crossing.

**precol**
[None or dict, optional (default=None)] A dictionary, keyed by faces, that specifies the colors of the precolored faces. Faces are identified using a tuple of nodes. Each internal face is characterized by the series of nodes that surround it in a counterclockwise direction; the one external face is identified by the series of nodes, traveling in a clockwise direction.

**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 dual's nodes and then applies the greedy algorithm for graph node coloring [1].
- `'welsh-powell'` : Orders the dual's nodes by decreasing degree, then applies the greedy algorithm.
- `'dsatur'` : Uses the DSatur algorithm for graph node coloring on the dual [2].
- `'rlf'` : Uses the recursive largest first (RLF) algorithm for graph node coloring on the dual [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 the dual to have the same color. Each iteration has a complexity $O(m + kn)$, where $n$ is the number of nodes in the dual, $m$ is the number of edges in the dual, 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 the dual to be uncolored. Each iteration has a complexity $O(m + kn)$, as above.
- `4` : A hybrid evolutionary algorithm (HEA) that evolves a small population of solutions. During execution, when each new solution is created, the local search method used in Option `2` above is applied for a fixed number of iterations. Each iteration of this HEA therefore has a complexity of $O(m + kn)$, as above.
- `5` : A hybrid evolutionary algorithm is applied (as above), using the local search method from Option `3`.
- `None` : No optimization is performed.

Further details of these algorithms are given in the notes section of the `node_coloring()` method.

**it_limit**
[int, optional (default=0)] Number of iterations of the local search procedure. Not applicable when using `opt_alg=1`.

**verbose**
[int, optional (default=0)] If set to a positive value, information is output during the optimization process. The higher the value, the more information.

**Returns**

**dict**
A dictionary with keys representing faces and values representing their colors. Colors are identified by the integers $0, 1, 2, \ldots$. Each face (key) is a tuple that gives the sequence of nodes that surround it. The first element in `c` defines the external face; the remainder define the internal faces. For internal faces, the nodes are listed in a counterclockwise order; for the external face, the nodes are listed in a clockwise order.

**Raises**

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

If `G` contains any self-loops.

If `G` is not connected or is a singleton.

**TypeError**
If `pos` is not a dictionary.

**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 `verbose` is not a nonnegative integer.

If `precol` contains a face that is not in the embedding of `G`.

If `precol` contains a color label that is not a nonnegative integer.

If `precol` contains a pair of adjacent faces assigned to the same color.

If `pos` has missing or invalid entries.

If `pos` does not specify a valid planar embedding of `G`.

If `G` is not planar or contains bridges.

**TypeError**
If `precol` is not a dict.

**See also**

*face_coloring*
*dual_graph*
*gcol.node_coloring.node_precoloring()*

### Notes

As mentioned, in this implementation face colorings of a graph $G$ are determined by forming its dual and then passing this to the `node_precoloring()` method. All details are therefore the same as those in the latter method, where they are documented more fully.

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()
>>> pos = nx.planar_layout(G)
>>> precol = {(0, 10, 9, 8, 1): 0, (6, 5, 4, 3, 2): 1}
>>> c = gcol.face_precoloring(G, pos, precol)
>>> print(c)
{(0, 10, 9, 8, 1): 0, (0, 19, 18, 11, 10): 1,..., (2, 6, 5, 4, 3): 1}
```

## 7.3 Node Coloring

Node coloring functions.

gcol.node_coloring.**chromatic_number**($G$)

Return 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.

If G contains any self-loops.

**See also**

*node_coloring*
*gcol.edge_coloring.chromatic_index()*

### 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.node_coloring.**equitable_node_k_coloring**(*G*, *k*, *weight=None*, *opt_alg=None*, *it_limit=0*, *verbose=0*)

Attempt to color the nodes of a graph using `k` colors.

This is done 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 standard deviation 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 standard deviation 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.
- `4` : A hybrid evolutionary algorithm (HEA) that evolves a small population of solutions. During execution, when each new solution is created, the local search method used in Option `2` above is applied for a fixed number of iterations. Each iteration of this HEA therefore has a complexity of $O(m + kn)$, as above.
- `5` : A hybrid evolutionary algorithm is applied (as above), using the local search method from Option `3`.
- `None` : No optimization is performed.

Further details of these algorithms are given in the notes section of the *node_coloring()* method.

**it_limit**
[int, optional (default=0)] Number of iterations of the local search procedure. Not applicable when using `opt_alg=1`.

**verbose**
[int, optional (default=0)] If set to a positive value, information is output during the optimization process. The higher the value, the more information.

**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.

If `G` contains any self-loops.

**ValueError**
If `opt_alg` is not among the supported options.

If `it_limit` is not a nonnegative integer.

If `verbose` 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*
*kempe_chain*
*gcol.edge_coloring.equitable_edge_k_coloring()*

### 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 standard deviation 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 standard deviation. This process repeats until there are no interchanges that reduce the standard deviation. 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], ..., [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 =",
```

(continues on next page)

(continued from previous page)

```
...     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.node_coloring.**kempe_chain**($G$, $c$, $v$, $i$, $j$)

Return the set of nodes in a Kempe chain.

Given a proper node coloring of graph $G$, a Kempe chain is a connected component in the graph induced by nodes of color $i$ and $j$. This method returns the Kempe chain containing the prescribed node $v$, where the color of $v$ is $i$. Any uncolored nodes (i.e., those whose colors are set to `-1`) are ignored [1].

The colors $i$ and $j$ alternate along any path in a Kempe chain. In a proper coloring, interchanging the colors of all nodes in a Kempe chain creates a new proper coloring. 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. It is known that, if $k$ is larger than the degeneracy of the graph $G$, then all $k$-colorings of $G$ 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]` set to `-1`.

**v**
: [node] The node the Kempe chain is generated from.

**i**
: [int] The first color to use. This is the current color of `v`.

**j**
: [int] The second color to use. Must be different to `i`.

**Returns**

**set**
: The set of nodes in the corresponding Kempe chain.

**Raises**

**ValueError**
: If `i` and `j` are equal.

  If `v` is not assigned to color `i` in `c`.

  If `v` is not present in `G`.

**See also**

*s_chain*
*equitable_node_k_coloring*

### Notes

A Kempe chain is simply an $s$-chain using $s = 2$ colors. As such, this method applies the *s_chain()* method.

Kempe-chains can also be constructed for edge and face colorings of a graph $G$ by using the corresponding node colorings of the line graph $L(G)$ and dual graph $G^*$, respectively.

### References

[1], [2]

### Examples

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

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

Attempt to identify the largest independent set of nodes in a graph.

Here, 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 as to whether this is an 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.

**verbose**
: [int, optional (default=0)] If set to a positive value, information is output during the optimization process. In this output, the cost refers to the number of nodes not in the independent set.

**Returns**

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

**Raises**

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

If `G` contains any self-loops.

**ValueError**
If `it_limit` is not a nonnegative integer.

If `verbose` 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)
```

(continues on next page)

(continued from previous page)

```
>>> 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.node_coloring.**min_cost_k_coloring**(*G*, *k*, *weight=None*, *weights_at='nodes'*, *it_limit=0*, *HEA=False*, *verbose=0*)

Color the nodes of the graph using `k` colors.

This is done 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.

**HEA**
[bool, optional (default=False)] If set to `True`, a hybrid evolutionary algorithm is used in conjunction with local search; otherwise, only local search is used.

**verbose**
[int, optional (default=0)] If set to a positive value, information is output during the optimization process.

**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.

If `G` contains any self-loops.

**ValueError**
If `weights_at` is not among the supported options.

If `it_limit` is not a nonnegative integer.

If `verbose` 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.

Further details on the local search and hybrid evolutionary algorithms, can be found in the notes section of the *node_coloring()* method.

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)
>>> 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"], ") clashes")
Edge 3 19 ( cost = 16 ) clashes
Edge 5 6 ( cost = 1 ) clashes
Edge 7 8 ( cost = 1 ) clashes
Edge 9 10 ( cost = 1 ) clashes
Edge 11 18 ( cost = 7 ) clashes
Edge 15 16 ( cost = 1 ) clashes
```

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

Return 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 four 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.
- `4` : A hybrid evolutionary algorithm (HEA) that evolves a small population of solutions. During execution, when each new solution is created, the local search method used in Option `2` above is applied for a fixed number of iterations. Each iteration of this HEA therefore has a complexity of $O(m + kn)$, as above.
- `5` : A hybrid evolutionary algorithm is applied (as above), using the local search method from Option `3`.
- `None` : No optimization is performed.

Further details of these algorithms are given below.

**it_limit**
[int, optional (default=0)] Number of iterations of the local search procedure. Not applicable when using `opt_alg=1`.

**verbose**
[int, optional (default=0)] If set to a positive value, information is output during the optimization process. The higher the value, the more information.

**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.

If `G` contains any self-loops.

**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 `verbose` is not a nonnegative integer.

**See also**

*chromatic_number*
*node_k_coloring*
*gcol.edge_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` is set to `2`, `3`, `4`, or `5`), the algorithm removes a color class and uses the chosen local search routine to seek a proper coloring using the remaining colors. This process repeats until a solution using $|C|$ colors has been identified (as above), or until the iteration limit (defined by `it_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 a complexity of $O(nk + m)$, 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.

If `opt_alg` is set to `4` or `5`, a hybrid evolutionary algorithm (HEA) is used [5]. This method maintains a small population of $k$-colored solutions that is evolved using selection, recombination, local search and replacement. Specifically, in each HEA cycle, two parent solutions are selected from the population, and these are used in conjunction with a specialized recombination operator to produce a new offspring solution. Local search is this applied to the offspring for a fixed number of iterations, and the resultant solution is inserted back into the population, replacing its weaker parent. If `opt_alg=4`, TabuCol is used as the local search operator; if `opt_alg=5`, PartialCol is used. Each iteration of the HEA has complexity $O(nk + m)$, as above. Note that the HEA is often able to produce solutions using fewer colors compared to when using `opt_alg=2` or `opt_alg=3`; however, larger iteration limits will usually be needed to see these improvements.

As stated above, if `verbose` is set to a positive integer, output is produced during the execution of the chosen optimization algorithm. If the backtracking algorithm is being used, the stated iterations refer to the number of calls to its recursive function. Otherwise, iterations refer to the $O(nk + m)$ processes mentioned above. If no optimization is performed, no output is produced.

All the above algorithms and bounds are described in detail in [4]. The c++ code used in [4] 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.node_coloring(G)
>>> print("Coloring is", c)
Coloring is {0: 0, 1: 1, 19: 1, 10: 1, 2: 0, ..., 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], ..., [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, ..., 27: 2}
>>>
>>> print("Number of colors =", max(c.values()) + 1)
Number of colors = 5
```

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

Attempt to color the nodes of a graph using `k` colors.

This is done 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 four 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.
- `4` : A hybrid evolutionary algorithm (HEA) that evolves a small population of solutions. During execution, when each new solution is created, the local search method used in Option `2` above is applied for a fixed number of iterations. Each iteration of this HEA therefore has a complexity of $O(m + kn)$, as above.
- `5` : A hybrid evolutionary algorithm is applied (as above), using the local search method from Option `3`.
- `None` : No optimization is performed.

Further details of these algorithms are given in the notes section of the *node_coloring()* method.

**it_limit**
[int, optional (default=0)] Number of iterations of the local search procedure. Not applicable when using `opt_alg=1`.

**verbose**
[int, optional (default=0)] If set to a positive value, information is output during the optimization process. The higher the value, the more information.

**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.

If `G` contains any self-loops.

**ValueError**
If `opt_alg` is not among the supported options.

If `it_limit` is not a nonnegative integer.

If `verbose` 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*
*gcol.edge_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, ..., 15: 3}
>>>
>>> c = gcol.node_k_coloring(G, 3)
```

(continues on next page)

(continued from previous page)

```
>>> print(c)
{0: 0, 1: 1, 19: 2, 10: 1, 2: 0, ..., 12: 1}
```

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

Return a solution to the node list coloring problem on `G`.

In the list coloring problem, each node $v$ is associated with a list of allowed colors $L(v)$. A solution is an assignment of colors to all nodes so that adjacent nodes have different colors, and the color of each node $v$ belongs to the list $L(v)$.

The list coloring problem is NP-hard. This method therefore includes options for using an exponential-time exact algorithm (based on backtracking), or a choice of four 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, colors are defined using labels belonging to $\{0, 1, 2, \ldots\}$. Assuming $k$ is the largest color label appearing across all lists $L(v)$, the aim is to find a solution using at most $k$ colors. If this cannot be done, an exception is raised (see below).

Here, solutions are found by adding a clique of $k$ dummy nodes, one for each color. Additional edges are then added between a node $v$ and a dummy node $i$ whenever $i$ does not occur in $L(v)$. The modified graph is then passed to the *node_k_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.

**allowed_cols**
: [None or dict, optional (default=None)] A dictionary, keyed by the nodes of `G`. `allowed_cols[v]` should be a list of nonnegative integers that defines the permissible colors for node `v`. If None, a node coloring with the minimum number of colors is returned.

**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.

  - `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.
- `4` : A hybrid evolutionary algorithm (HEA) that evolves a small population of solutions. During execution, when each new solution is created, the local search method used in Option `2` above is applied for a fixed number of iterations. Each iteration of this HEA therefore has a complexity of $O(m + kn)$, as above.
- `5` : A hybrid evolutionary algorithm is applied (as above), using the local search method from Option `3`.
- `None` : No optimization is performed.

Further details of these algorithms are given in the notes section of the *node_coloring()* method.

**it_limit**
[int, optional (default=0)] Number of iterations of the local search procedure. Not applicable when using `opt_alg=1`.

**verbose**
[int, optional (default=0)] If set to a positive value, information is output during the optimization process. The higher the value, the more information.

**Returns**

**dict**
A dictionary with keys representing nodes and values representing their colors. Colors are identified by the integers $0, 1, 2, \ldots$. If `c[v]==j` then `j` is an element of `allowed_cols[v]`.

**Raises**

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

If `G` contains any self-loops.

**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 `verbose` is not a nonnegative integer.

If `G` contains a node with the name `'dummy'`.

If `allowed_cols` has a node not in G, or is missing an entry for a node in `G`.

If `allowed_cols` contains a color label that is not a nonnegative integer.

If `allowed_cols` contains a list that is empty.

If `allowed_cols` contains a pair of adjacent nodes that have the same single allowed color.

If a node list k-coloring could not be determined (the lists of allowed colors are too restrictive).

**TypeError**
If `allowed_cols` is not a dict.

**See also**

*node_coloring*
*gcol.edge_coloring.edge_precoloring()*

### Notes

As mentioned, in this implementation, solutions are formed by passing a modified version of the graph to the *node_k_coloring()* method. Details are therefore the same as those in the latter.

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.cycle_graph(4)
>>> V = {0: [0, 1], 1: [1], 2: [0, 3], 3: [0, 1, 3]}
>>> c = gcol.node_list_coloring(G, V)
>>> print(c)
{1: 1, 0: 0, 2: 3, 3: 1}
```

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

Return a coloring of a graph's nodes where some nodes are precolored.

A node coloring of a graph is an assignment of colors to nodes so that adjacent nodes have different colors. In the node precoloring problem, some of the nodes have already been assigned colors. Colors are defined using labels belonging to the set $\{0, 1, 2, \ldots\}$. Assuming $k$ is the largest color label used in the precoloring, the aim is to color all remaining nodes using $l$ colors, where $l$ is minimized and $k \leq l$. 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 four 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 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 (nonnegative 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.
- `4` : A hybrid evolutionary algorithm (HEA) that evolves a small population of solutions. During execution, when each new solution is created, the local search method used in Option `2` above is applied for a fixed number of iterations. Each iteration of this HEA therefore has a complexity of $O(m + kn)$, as above.
- `5` : A hybrid evolutionary algorithm is applied (as above), using the local search method from Option `3`.
- `None` : No optimization is performed.

Further details of these algorithms are given in the notes section of the *node_coloring()* method.

**it_limit**
[int, optional (default=0)] Number of iterations of the local search procedure. Not applicable when using `opt_alg=1`.

**verbose**
[int, optional (default=0)] If set to a positive value, information is output during the optimization process. The higher the value, the more information.

**Returns**

**dict**
A dictionary with keys representing nodes and values representing their colors. Colors are identified by the integers $0, 1, 2, \ldots$. If `precol[v]==j` then `c[v]==j`.

**Raises**

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

If `G` contains any self-loops.

**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 `verbose` 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 color label that is not a nonnegative integer.

If `precol` contains a pair of adjacent nodes assigned to the same color.

**TypeError**
If `precol` is not a dict.

**See also**

*node_coloring*
*gcol.edge_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, ..., 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, ..., 12: 2}
```

gcol.node_coloring.**s_chain**($G$, $c$, $v$, $L$)

Return the set of nodes in an $s$-chain.

An $s$-chain is a generalization of a Kempe chain that allows more than two colors. Given a proper node coloring of a graph $G = (V, E)$, an $s$-chain is defined by a prescribed node $v \in V$ and sequence of unique colors $j_0, j_1, \ldots, j_{s-1}$, where the current color of $v$ is $j_0$. The result is the set of nodes that are reachable from $v$ in the digraph $G' = (V', A)$ in which:

- $V' = \{u \ : \ u \in V \ \wedge \ c(u) \in \{j_0, j_1, \ldots, j_{s-1}\}\}$, and
- $A = \{(u, w) \ : \ \{u, w\} \in E \ \wedge \ c(u) = j_i \ \wedge \ c(w) = j_{(i+1) \bmod s}\}$,

where $c(u)$ gives the color of a node $u$.

In a proper coloring, interchanging the colors of all nodes in an $s$-chain via the following mapping

- $j_i \leftarrow j_{(i+1) \bmod s}$

results in a new proper coloring [1].

In this method, uncolored nodes are ignored.

**Parameters**

**G**
[NetworkX graph] The graph that we want to compute an $s$-chain for.

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

**v**
[node] The node the $s$-chain is to be generated from.

**L**
[list] A sequence of unique colors, represented by integers. The first color in `L` should be the current color of `v`.

**Returns**

**set**
The set of nodes in the corresponding $s$-chain.

**Raises**

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

If `G` contains any self-loops.

**ValueError**
If `v` is not present in `G`.

If `G` has a node that is not present in `c`.

If `c` contains a pair of adjacent nodes assigned to the same color.

If `L` is not a list or tuple, has a length of less than two, or contains repeated values.

If the first value of `L` is not equal to `c[v]`

If `L` contains values that are not in the set $\{0, 1, 2, \ldots\}$.

**See also**

*kempe_chain*
*equitable_node_k_coloring*
*gcol.face_coloring.dual_graph()*

### Notes

This method uses a modified version of breadth-first search and operates in $O(m)$ time.

$s$-chains can also be constructed for edge and face colorings of a graph $G$ by using the corresponding node colorings of the line graph $L(G)$ and dual graph $G^*$, respectively.

### References

[1]

### Examples

```
>>> import networkx as nx
>>> import gcol
>>>
>>> G = nx.dodecahedral_graph()
>>> c = gcol.node_coloring(G)
>>> C = gcol.s_chain(G, c, 0, [0, 1, 2])
>>> print("s-chain =", C)
s-chain = {0, 1, 2, ..., 19}
>>> C = gcol.s_chain(G, c, 0, [0, 2, 1])
>>> print("s-chain =", C)
s-chain = {0}
```

## 7.4 Output

Output functions.

`gcol.output.coloring_layout`($G$, $c$)

Arrange the nodes of the graph in a circle.

Nodes of the same color are put 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_colors(G, c)
... )
>>> plt.show()
```

gcol.output.**draw_face_coloring**(*c*, *pos*, *external=False*, *palette=None*)

Draw the face coloring defined by `c` and `pos`.

The RGB color of each face is determined by its color label in `c` and the chosen palette. If a face is marked as uncolored (i.e., assigned a value of `-1`) it is painted white.

**Parameters**

**c**
[dict] A dictionary where keys represent the sequence of nodes occurring in each face (polygon), and values represent the face's color. Colors are identified by the integers $0, 1, 2, \ldots$. The first element in `c` defines the external face of the embedding; the remaining elements define the internal faces.

**pos**
[dict] A dict specifying the (x,y) coordinates of each node in the embedding.

**external**
[bool, optional (default=False)] If set to `True`, the background (corresponding to the external face of the embedding) is also colored, else it is left blank.

**palette**
[None or dict, optional (default=None)] A dictionary that maps the integers $-1, 0, 1, \ldots$ to RGB values. The in-built options are as follows

- `gcol.tableau` : A collection of 21 colors provided by Tableau.
- `gcol.colorful` : A collection of 57 bright colors that are chosen to contrast each other as much as possible.
- `gcol.colorblind` : A collection of 11 colors, provided by Tableau, that are intended to help colorblind users.
- If `None`, then `gcol.tableau` is used.

**Returns**

**None**

**Raises**

**ValueError**
If `c` uses more colors than available in the palette.

**See also**

*get_set_colors*
*get_edge_colors*
*get_node_colors*

### Notes

User generated palettes can also be passed into this method. In such cases it is good practice to map the value `-1` to the color white. Descriptions on how to specify valid colors can be found at [1].

### References

[1]

### Examples

```
>>> import networkx as nx
>>> import gcol
>>> import matplotlib.pyplot as plt
>>>
>>> # Construct a small planar graph with defined node positions
>>> G = nx.Graph()
>>> G.add_node(0, pos=(0,0))
>>> G.add_node(1, pos=(1,0))
>>> G.add_node(2, pos=(0,1))
>>> G.add_node(3, pos=(1,1))
>>> G.add_edges_from([(0,1),(0,2),(0,3),(1,3),(2,3)])
>>>
>>> # Color the faces of the embedding and draw to the screen
>>> pos = nx.get_node_attributes(G, "pos")
>>> c = face_coloring(G, pos)
>>> draw_face_coloring(c, pos)
>>> plt.show()
```

`gcol.output.`**`get_edge_colors`**(*G*, *c*, *palette=None*)

Generate an RGB color for each edge in the graph `G`.

The RGB color of an edge is determined by its color label in `c` and the chosen palette. 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 value of `-1` , 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 $0, 1, 2, \ldots$.

**palette**
[None or dict, optional (default=None)] A dictionary that maps the integers $-1, 0, 1, \ldots$ to RGB values. The in-built options are as follows

- `gcol.tableau` : A collection of 21 colors provided by Tableau.
- `gcol.colorful` : A collection of 57 bright colors that are chosen to contrast each other as much as possible.
- `gcol.colorblind` : A collection of 11 colors, provided by Tableau, that are intended to help colorblind users.
- If `None`, then `gcol.tableau` is used.

**Returns**

**dict**

**list**
A sequence of RGB colors in edge order.

**Raises**

**ValueError**
If `c` uses more colors than available in the palette.

**See also**

*get_set_colors*
*get_node_colors*

### Notes

User generated palettes can also be passed into this method. Descriptions on how to specify valid colors can be found at [1].

### References

[1]

### 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.output.`**`get_node_colors`**(*G*, *c*, *palette=None*)

Generate an RGB color for each node in the graph `G`.

The RGB color of a node is determined by its color label in `c` and the chosen palette. 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 value of `-1`, or is 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 $0, 1, 2, \ldots$.

**palette**
[None or dict, optional (default=None)] A dictionary that maps the integers $-1, 0, 1, \ldots$ to RGB values. The in-built options are as follows

- `gcol.tableau` : A collection of 21 colors provided by Tableau.
- `gcol.colorful` : A collection of 57 bright colors that are chosen to contrast each other as much as possible.
- `gcol.colorblind` : A collection of 11 colors, provided by Tableau, that are intended to help colorblind users.
- If `None`, then `gcol.tableau` is used.

**Returns**

**list**
A sequence of RGB colors in node order.

**Raises**

**ValueError**
If `c` uses more colors than available in the palette.

**See also**

*get_set_colors*
*get_edge_colors*

## Notes

User generated palettes can also be passed into this method. In such cases it is good practice to map the value `-1` to the color white. Descriptions on how to specify valid colors can be found at [1].

## References

[1]

## Examples

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

(continues on next page)

(continued from previous page)

```
>>> 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.output.**get_set_colors**(*G*, *S*, *S_color='yellow'*, *other_color='grey'*)

Generate an RGB color for each node based on if it is in 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 sequence of RGB colors, in node order

**See also**

*get_node_colors*
*get_edge_colors*

### Notes

Descriptions on how to specify valid colors can be found at [1].

### References

[1]

### 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(
```

(continues on next page)

(continued from previous page)

```
...     G, pos=nx.spring_layout(G), node_color=gcol.get_set_colors(G, S)
... )
>>> plt.show()
```

gcol.output.**multipartite_layout**($G$, $c$)

Arrange the nodes of the graph into columns.

Nodes of the same color are put in the same column. This method is 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.output.**partition**($c$)

Convert 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.

### 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], ..., [3, 9, 11, 7, 5, 16]]
>>>
>>> c = gcol.edge_coloring(G)
>>> print(gcol.partition(c))
[[(11, 12), (18, 19), (16, 17), ..., (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] 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] 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/>

[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: Four Color Theorem <https://en.wikipedia.org/wiki/Four_color_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: Four Color Theorem <https://en.wikipedia.org/wiki/Four_color_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: Four Color Theorem <https://en.wikipedia.org/wiki/Four_color_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: Four Color Theorem <https://en.wikipedia.org/wiki/Four_color_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] 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/>

[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] 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: 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] Galinier, P. and J. Hao (1999). Hybrid Evolutionary Algorithms for Graph Coloring. Journal of Combinatorial Optimization 3, 379–397.

[6] 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/>

[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/>

[1] Morgenstern, C. and H. Shapiro (1990), Coloration Neighborhood Structures for General Graphs <https://dl.acm.org/doi/pdf/10.5555/320176.320202>

[1] Matplotlib: Specifying colors <https://matplotlib.org/stable/users/explain/colors/colors.html>

[1] Matplotlib: Specifying colors <https://matplotlib.org/stable/users/explain/colors/colors.html>

[1] Matplotlib: Specifying colors <https://matplotlib.org/stable/users/explain/colors/colors.html>

[1] Matplotlib: Specifying colors <https://matplotlib.org/stable/users/explain/colors/colors.html>

# PYTHON MODULE INDEX

## g

# INDEX