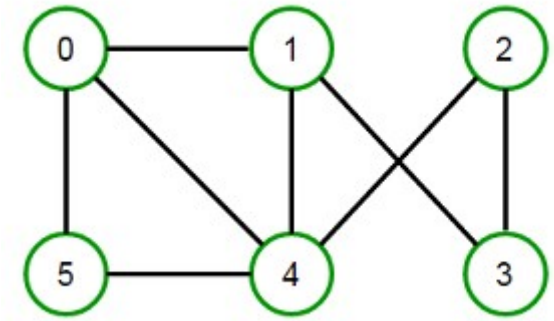
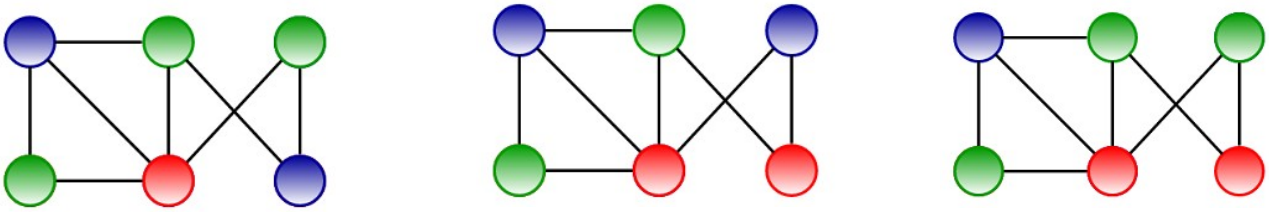
Graph Coloring Problem (or Map Coloring)

Graph coloring (also called vertex coloring) is a way of coloring a graph's vertices such that no two adjacent vertices share the same color.
For example, consider the following graph:



We can color it in many ways by using the minimum of 3 colors.
Please note that we can't color the above graph using two colors.



K–colorable graph:

A coloring using at most k colors is called a (proper) *k*–coloring, and a graph that can be assigned a (proper) *k*–coloring is *k*–colorable.

K–chromatic graph:

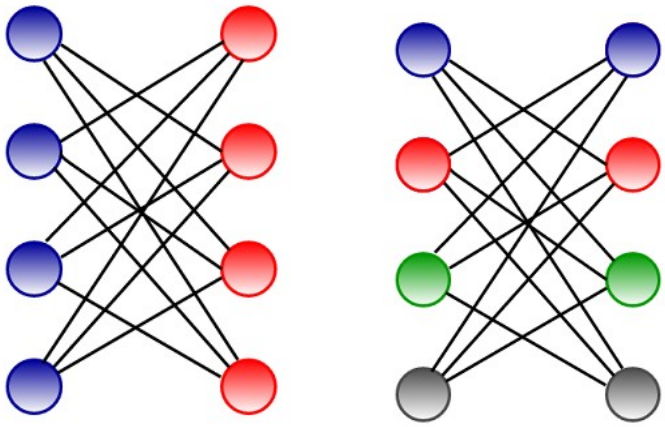
The smallest number of colors needed to color a graph G is called its chromatic number, and a graph that is *k*–chromatic if its chromatic number is exactly k.

Greedy coloring *considers the vertices of the graph in sequence and assigns each vertex its first available color*, i.e., vertices are considered in a specific order v1, v2, … vn, and vi and assigned the smallest available color which is not used by any of vi's neighbors.

Greedy coloring doesn't always use the minimum number of colors possible to color a graph.

For a graph of maximum degree x, greedy coloring will use at most x+1 color. Greedy coloring can be arbitrarily bad; for example, the following crown graph (a complete bipartite graph), having n

vertices, can be 2–colored (refer left image), but greedy coloring resulted in n/2 colors (refer right image).



The algorithm can be implemented as follows in Python: GraphColoring1.py

Using backtracking: GraphColoring2.py

The idea is to assign colors one by one to different vertices, starting from vertex 0. Before assigning a color, check for safety by considering already assigned colors to the adjacent vertices i.e check if the adjacent vertices have the same color or not. If there is any color assignment that does not violate the conditions, mark the color assignment as part of the solution. If no assignment of color is possible then backtrack and return false

Follow the given steps to solve the problem:

- Create a recursive function that takes the graph, current index, number of vertices, and output color array.
- If the current index is equal to the number of vertices. Print the color configuration in the output array.
- Assign a color to a vertex (1 to m).
- For every assigned color, check if the configuration is safe, (i.e. check if the adjacent vertices do not have the same color) recursively call the function with the next index and number of vertices
- f any recursive function returns true break the loop and return true
- If no recursive function returns true then return false

Networks in python with library networkx

Run the following code:

```
import networkx as nx
```

```
network = nx.Graph()
network.add_nodes_from([4,5,6,7])
network.add_edge(1,4)
network.add_edge(1,5)
network.add_edge(2,4)
network.add_edge(3,5)
network.add_edge(5,6)
network.add_edge(7,3)
network.add_edge(7,6)
color_list = ["gold", "violet", "violet", "violet",
"limegreen", "limegreen", "darkorange"]
nx.draw_networkx(network,node_color=color_list, with_labels=True)
```

Okay this looks like a possible solution to the graph coloring problem. But you might ask yourself, how do I know how many colors I am gonna need?

CryptArithmetic Problem: SEND + MORE = MONEY

The Send More Money Problem consists in finding distinct digits for the letters D, E, M, N, O, R, S, Y such that S and M are different from zero (no leading zeros) and the equation
SEND + MORE = MONEY

is satisfied.

Code1:

```
from itertools import combinations, permutations

a, b, c = 'SEND', 'MORE', 'MONEY'
#a, b, c = 'hi','hello', 'goodby'

for comb in combinations(range(10), 8):
      for perm in permutations(comb):
            d = dict(zip('SENDMORY', perm))
            f = lambda x: sum(d[e] * 10**i for i, e in enumerate(x[::-1]))
            if f(a) + f(b) == f(c):
                  print("{} + {} = {}".format(f(a), f(b), f(c)))
```

code2:

```
from itertools import combinations, permutations

def replacements():
      for comb in combinations(range(10), 8):
            for perm in permutations(comb):
                  if perm[0] * perm[1] != 0:
                        yield dict(zip('SMENDORY', perm))

a, b, c = 'SEND', 'MORE', 'MONEY'
```

```
for replacement in replacements():
	f = lambda x: sum(replacement[e] * 10**i for i, e in enumerate(x[::-
1]))
	if f(a) + f(b) == f(c):
		print('{} + {} = {}'.format(f(a), f(b), f(c)))
```

code3:

```
import itertools


def get_value(word, substitution):
	s = 0
	factor = 1
	for letter in reversed(word):
		s += factor * substitution[letter]
		factor *= 10
	return s


def solve2(equation):
	# split equation in left and right
	left, right = equation.lower().replace(' ', '').split('=')
	# split words in left part
	left = left.split('+')
	# create list of used letters
	letters = set(right)
	for word in left:
		for letter in word:
			letters.add(letter)
	letters = list(letters)

digits = range(10)
for perm in itertools.permutations(digits, len(letters)):
	sol = dict(zip(letters, perm))

	if sum(get_value(word, sol) for word in left) == get_value(right,
sol):
		print(' + '.join(str(get_value(word, sol)) for word in left) + "
= {} (mapping: {})".format(get_value(right, sol), sol))

if __name__ == '__main__':
	#solve2('SEND + MORE = MONEY')
	solve2('SHOW + ME + THE = MONEY')
```

Exercize: try to solve this problem as a CSP problem.