OBJECTIVES:

- Find shortest number of colors in the graph can be colored with at most m colors such that no two adjacent vertices of the graph are colored with same color.
- Implementing Graph Coloring algorithm using your preferred programming language.

BACKGROUND STUDY

- Should have prior knowledge on any programming language to implement the algorithm.
- Input a graph.
- Graph Coloring algorithm that would be covered in theory class and that knowledge will help you here to get the implementation idea.

RECOMMENDED READING

- https://en.wikipedia.org/wiki/Graph_coloring
- BOOK

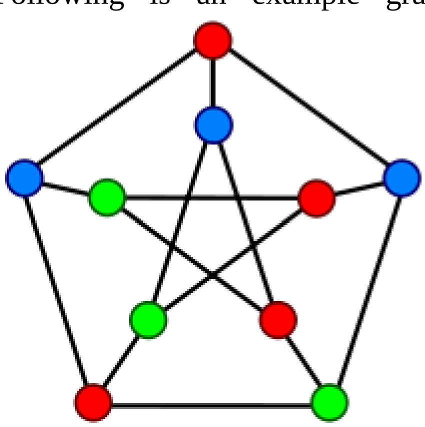
Graph Coloring Problem

Given an undirected graph and a number m, determine if the graph can be colored with at most m colors such that no two adjacent vertices of the graph are colored with same color. Here coloring of a graph means assignment of colors to all vertices.

Input:
1) A 2D array graph[V][V] where V is the number of vertices in graph and graph[V][V] is adjacency matrix representation of the graph. A value graph[i][j] is 1 if there is a direct edge from i to j, otherwise graph[i][j] is 0.
2) An integer m which is maximum number of colors that can be used.

Output:
An array color[V] that should have numbers from 1 to m. color[i] should represent the color assigned to the ith vertex. The code should also return false if the graph cannot be colored with m colors.

Following is an example graph (from Wiki page) that can be colored with 3 colors.



Naive Algorithm

Generate all possible configurations of colors and print a configuration that satisfies the given constraints.

```
while there are untried configuration
{
   generate the next configuration
   if no adjacent vertices are colored with same color
   {
      print this configuration;
   }
}
```

There will be V^m configurations of colors.

Backtracking Algorithm

The idea is to assign colors one by one to different vertices, starting from the vertex 0. Before assigning a color, we check for safety by considering already assigned colors to the adjacent vertices. If we find a color assignment which is safe, we mark the color assignment as part of solution. If we do not a find color due to clashes then we backtrack and return false.

The most obvious solution to this problem is arrived at through a design referred to as *backtracking*.

Recall that the essence of backtracking is:

1. Number the solution variables [v_0 v_1, …, v_{n-1}].
2. Number the possible values for each variable [c_0 c_1, …, c_{k-1}].
3. Start by assigning c_0 to each v_i.
4. If we have an acceptable solution, stop.
5. If the current solution is not acceptable, let i = n-1.
6. If i < 0, stop and signal that no solution is possible.
7. Let j be the index such that $v_i = c_j$. If j < k-1, assign c_{j+1} to v_i and go back to step 4.
8. But if j ≥ k-1, assign c_0 to v_i, decrement i, and go back to step 6.

Although this approach will find a solution eventually (if one exists), it isn't speedy. Backtracking over n variables, each of which can take on k possible values, is $O(k^n)$.

For graph coloring, we will have one variable for each node in the graph. Each variable will take on any of the available colors.