A Detail Analysis on Graph-Coloring Problems

Abstract:

Graphs are considered as an excellent modeling tool which is used to model many type of relations amongst any physical situation. Many problems of real world can be represented by graphs. This paper explores concept of Graph coloring involved in graph theory and its applications in computer science to demonstrate the utility of graph theory. The paper will analysis two greedy algorithms which provided an optimal solution for the graph coloring problems. The paper also analysis the different parameters like how order of the vertices's and graph topologies will effect the performance of the graphs.

Introduction:

Graph theoretical ideas are highly utilized by computer science applications. Especially in research areas of computer science such data mining, image segmentation, clustering, image capturing, networking etc., For example a data structure can be designed in the form of tree which in turn utilized vertices's and edges. Similarly modeling of network topologies can be done using graph concepts. In the same way the most important concept of graph coloring is utilized in resource allocation, scheduling. Also, paths, walks and circuits in graph theory are used in tremendous applications say traveling salesman problem, database design concepts, resource networking.

The paper progresses by introducing the graphs which will be using in the analysis and a brief introduction to graph coloring problems. Then the paper progresses with an in depth analysis of two greedy algorithms along with some of the graph topologies.

Graph Coloring Problems:

A **graph** is an ordered pair G = (V, E) comprising a set V of vertices or nodes or points together with a set E of edges or arcs or lines; this type of graph may be described precisely as undirected and simple.

There are various types of graphs. But some of the most commonly used types of graphs which we will discuss are bipartite graph, undirected graphs and complete graphs.

1. **Bipartite graph:**

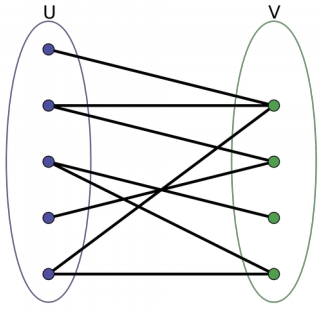
   A bipartite graph, also called a **bigraph**, is a set of graph vertices decomposed into two disjoint sets such that no two graph vertices within the same set are adjacent.

2. **Undirected graph:**

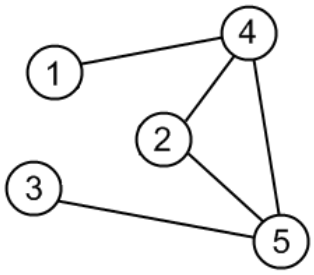
An undirected graph is graph with a set of objects (called vertices or nodes) that are connected together, where all the edges are bidirectional. An undirected graph is sometimes called an **undirected network or Simple graph**.

3. **Complete graph:**

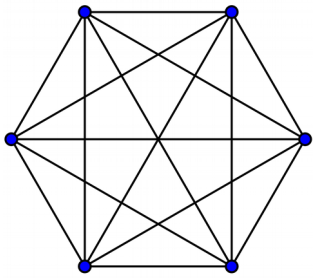
A complete graph is a simple undirected graph in which every pair of distinct vertices is connected by a unique edge.



(1)



(2)



(3)

A vertex coloring is an assignment of labels or colors to each vertex of a graph such that no edge connects two identically colored vertices. The most common type of vertex coloring seeks to minimize the number of colors for a given graph. The graph coloring is also called as vertex coloring.

Greedy Algorithms:

The Graph Coloring Problem (GCP) is a well- known NP-complete problem that has been studied extensively. Heuristics have been widely used for the GCP. The well-known Greedy method is the simplest algorithm which takes an ordering of nodes of a graph and colors these with the smallest color satisfying the constraints that no adjacent nodes are assigned same colors. However, the Greedy method performs poorly in practice. DSATUR uses a heuristic which changes the ordering of nodes and then uses the Greedy method to color these nodes.

The paper will be analyzing two of the two greedy algorithms: simple greedy algorithm and DSATUR algorithm and come up with some case studies.

Simple Greedy algorithm:

This is one of the simplest but most fundamental heuristic algorithm for graph coloring. The algorithm operates by taking vertices one by one according to some ordering and assign each vertex its first available colour.

Pseudo code for Simple Greedy method:

```
GREEDY (S ← ∅, π)
(1) for i ← 1 to |π| do
(2)     for j ← 1 to |S|
(3)         if (S_j ∪ {π_i}) is an independent set then
(4)             S_j ← S_j ∪ {π_i}
(5)             break
(6)         else j ← j + 1
(7)     if j > |S| then
(8)         S_j ← {π_i}
(9)         S ← S ∪ S_j
```

1. The algorithm takes an empty solution *S*=Ø and an arbitary permutaion of vertices ∏.
2. In each of the outer loop the algorithm takes the *i*th vertex in the given order and attempts to find the colour class *Sj* ∈ *S* into which it can be inserted.
3. If such a colour class curently exists in S, then the vertex is added to it and the process moves on to consider the next vertex ∏*i+1*
4. If not the algorithm will create a new color class for the algorithm.

Time complexity:

The algorithm colors each vertex at each iteration,i.e., n iterations are required in total. At the *i*th iteration the algorithm is concerned with finding the feasible colour for the vertex. In the words case the algorithm will have to consider all the vertices that have been proceeded. Such cases will occur if the graph is a complete graph. The worst case will occur for all the vertices and hence a total of 0+1+2+...+*(n-1)* checks will be performed. This gives the overall worst case complexity of O(n^2).

Analysis:

I wrote a python program (Graph_Coloring.py) based on the the above Pseudo code. I represented the vertices in the form of a adjacency list.

Case 1:

I have considered the 50 states of USA as my vertices and created a adjacency list with respect to their neighboring states.

```
colors = ['Red', 'Blue', 'Green', 'Yellow', 'Black']

states=['WA', 'DE', 'DC', 'WI', 'WV', 'HI', 'FL', 'WY', 'NH', 'NJ', 'NM', 'TX',
        'LA', 'NC', 'ND', 'NE', 'TN', 'NY', 'PA', 'RI', 'NV', 'VA', 'CO', 'CA',
        'AL', 'AR', 'VT', 'IL', 'GA', 'IN', 'IA', 'OK', 'AZ', 'ID', 'CT', 'ME', 'MD',
        'MA', 'OH', 'UT', 'MO', 'MN', 'MI', 'KS', 'MT', 'MS', 'SC', 'KY', 'OR', 'SD']
```

Colors is the list of colors that the algorithm can use and states is the list of 50 states of America according to their state initials.

Output:

{'WA': 'Red', 'DE': 'Red', 'DC': 'Red', 'WI': 'Red', 'WV': 'Red', 'HI': 'Red', 'FL': 'Red', 'WY': 'Red', 'NH': 'Red', 'NJ': 'Blue', 'NM': 'Red', 'TX': 'Blue', 'LA': 'Red', 'NC': 'Red', 'ND': 'Red', 'NE': 'Blue', 'TN': 'Blue', 'NY': 'Red', 'PA': 'Green', 'RI': 'Red', 'NV': 'Red', 'VA': 'Green', 'CO': 'Green', 'CA': 'Blue', 'AL': 'Green', 'AR': 'Green', 'VT': 'Blue', 'IL': 'Blue', 'GA': 'Yellow', 'IN': 'Red', 'IA': 'Green', 'MA': 'Green', 'AZ': 'Yellow', 'ID': 'Blue', 'CT': 'Blue', 'ME': 'Blue', 'MD': 'Blue', 'OK': 'Yellow', 'OH': 'Blue', 'UT': 'Black', 'MO': 'Red', 'MN': 'Blue', 'MI': 'Green', 'KS': 'Black', 'MT': 'Green', 'MS': 'Yellow', 'SC': 'Blue', 'KY': 'Yellow', 'OR': 'Green', 'SD': 'Yellow'}

Case 2:

Now I added some more colors to the list.

```
colors = ['Red', 'Blue', 'Green', 'Yellow', 'Black','Purple','Brown','White']
```

Output:

{'WA': 'Red', 'DE': 'Red', 'DC': 'Red', 'WI': 'Red', 'WV': 'Red', 'HI': 'Red', 'FL': 'Red', 'WY': 'Red', 'NH': 'Red', 'NJ': 'Blue', 'NM': 'Red', 'TX': 'Blue', 'LA': 'Red', 'NC': 'Red', 'ND': 'Red', 'NE': 'Blue', 'TN': 'Blue', 'NY': 'Red', 'PA': 'Green', 'RI': 'Red', 'NV': 'Red', 'VA': 'Green', 'CO': 'Green', 'CA': 'Blue', 'AL': 'Green', 'AR': 'Green', 'VT': 'Blue', 'IL': 'Blue', 'GA': 'Yellow', 'IN': 'Red', 'IA': 'Green', 'MA': 'Green', 'AZ': 'Yellow', 'ID': 'Blue', 'CT': 'Blue', 'ME': 'Blue', 'MD': 'Blue', 'OK': 'Yellow', 'OH': 'Blue', 'UT': 'Black', 'MO': 'Red', 'MN': 'Blue', 'MI': 'Green', 'KS': 'Black', 'MT': 'Green', 'MS': 'Yellow', 'SC': 'Blue', 'KY': 'Yellow', 'OR': 'Green', 'SD': 'Yellow'}

Conclusion 1:

The result remains the same. From the above two case it can be seen that the Greedy approach give an optimal solution by considering the minimum number of colors.

Case 3:

In this case I will be considering a complete graph. Practically not possible, but lets use the above states to create a complete graph. (view Complete_Graph.txt)

Output:

{'WA': 'Red', 'DE': 'Blue', 'DC': 'Green', 'WI': 'Yellow', 'WV': 'Black', 'HI': 'Purple', 'FL': 'Brown', 'WY': 'White', 'NH': 'Megenta', 'NJ': 'Safforon', 'NM': 'Gold', 'TX': 'Silver', 'LA': 'Red-blue', 'NC': 'Greem-yello', 'ND': 'Purple-Black', 'NE': 'Brown-White', 'TN': 'Mengenta-saffrin', 'NY': 'safforon-GoldSilver-Grey', 'PA': 'Grey', 'RI': 'Orange', 'NV': 'Grey-Orange', 'VA': 'Grey-Red', 'CO': 'Orange-Red', 'CA': 'Red-Green', 'AL': 'Red-Black', 'AR': 'Red-Brown', 'VT': 'Red-Purple', 'IL': 'Red-Megenta', 'GA': 'Red-safrron', 'IN': 'Yello-white', 'IA': 'Saffron-White', 'MA': 'Blue-Silver', 'AZ': 'Blue-Purple', 'ID': 'Blue-Brown', 'CT': 'Blue-White', 'ME': 'Blue-Safforon', 'MD': 'Blue-Gold', 'OK': 'Blue-Yellow', 'OH': 'Blue-Grey', 'UT': 'Blue-OrangeYellow-Purple', 'MO': 'Yellow-White', 'MN': 'Yellow-Safforon', 'MI': 'Yellow-Gold', 'KS': 'Yellow-Silver', 'MT': 'Yello-Grey', 'MS': 'Yellow-Orange', 'SC': 'Megenta-Grey', 'KY': 'Megenta-Orange', 'OR': 'Megenta-Purple', 'SD': 'Lime'}

Conclusion 2:

From this case we can conclude that, if the Graph with n number of vertices is a complete Graph; the n number of colors are required to get the solution.

Execution Time in seconds:

	Simple Greedy
Simple Undirected Graph	0.0003521442
Complete Graph	0.0149400234

DSatur Algorithm:

The Dsatur algorithm (abbreviated from “degree of saturation”) was originally proposed by Brelaz. It is very similar in behavior to the Simple Greedy algorithm.

The difference between the two algorithms lies in the way that these vertex orderings are generated. With the simple Greedy the ordering is decided before any coloring takes place; on the other hand, for the Dsatur algorithm the choice of which vertex to colour next is decided heuristically based ob the characteristics of the current partial coloring of the graph. The choice is primarily based on the saturation degree of the vertices.

The saturation degree of a vertex is defined as the number of colored adjacent vertex associated with it.

Pseudo code for Dsatur method:

```
DSATUR (S ← ∅, X ← V)
(1)  while X ≠ ∅ do
(2)      Choose v ∈ X
(3)      for j ← 1 to |S|
(4)          if (S_j ∪ {v}) is an independent set then
(5)              S_j ← S_j ∪ {v}
(6)              break
(7)          else j ← j + 1
(8)      if j > |S| then
(9)          S_j ← {v}
(10)         S ← S ∪ S_j
(11)     X ← X − {v}
```

- The major difference between Simple Greedy and Dsatur is that, here a set *X* is used to define the set of vertices currently not assigned to a colour.
- At the beginning of the execution *X=V.*
- In each iteration of the algorithm the next vertex to be coloured is selected from *X* (line 2). and once coloured, it is removed from *X* (line 11).
- The algorithm terminated at *X* = Ø
- **In line 2 the next vertex to be colored is chosen as the vertex in *X* that has a maximum saturation degree. If there is more than one vertex with the maximum saturation degree, then ties are broken by choosing the vertex among these with the largest degree.** Any further ties can then be broken randomly.

The idea behind the maximum saturation degree hurestic is that it prioritieses vertices that are seen to be the most “constrained”- that is, vertices that currently have the fewest color options available to them. Consequently, these “more constrainded ” vertices are delt with the algorithm first, allowing the less constrained vertices to be colored later.

Time complexity:

It can be seen that the majority of the algorithm is the same as the Simple Greedy algorithm in that once a vertex has been selected, a colour is found by simply going through each of the colour class in turn and stop when a suitable one has been found. Consequently, the worst-case complexity of Dsature is the same as simple Greedy at O(n^2), although in practice some extra bookkeeping is required to keep track of the saturation degrees of the uncolored vertices.

Analysis:

I implemented a python program based on the the above Pseudo code. I represented the vertices in the form of a adjacency list.

Case 4:

I generated a bipartite graph for the 50 states by making a two sets of 25 states each. I ran the program for all the graph topologies.

Execution Time in seconds:

	Simple Greedy	Dsatur
Simple Undirected Graph	0.0003521442	0.0057249069
Complete Graph	0.0149400234	2.1097390651
Bipartite Graph	0.0008020454	0.0090484399

Conclusion 3:

From the above analysis it can be seen that Dsatre algorithm takes more time for complete graph. It can be concluded that simple greedy technique is best for complete graphs.

Case 5:

In order to check the performance (stability) of the algorithms I conducted following executions on bipartite graph.

Execution Time in seconds:

Execution #	Simple Greedy	Dsatur
1	0.0006752	0.0061521
2	0.0011229	0.0078082
3	0.0006347	0.0072014
4	0.0059729	0.0092070
5	0.0028514	0.0058782

Conclusion 4:

The number of colors assigned by the Simple Greedy algorithm depends on the order that the vertices are fed into the procedure, with results potentially varying a great deal. While Dsatur algorithm reduces this variance by generating the vertex ordering during a run according to its heuristics. As a result. Dsatur performance is more predictable. Hence provides more stable results for different graph topologies.

It can be see that, for a Simple graph and Bipartite graph Dsatur algorithm has a slightly more execution time due to the lookup cost. Considering the stable performance of the algorithm it is evident that Dsatur is best for both the topologies.

References:

1. *"The Vertex Coloring Problem and its Generalizations" by Enrico Malaguti.*
2. *"The guide to graph coloring" by R.M.R Lewis*
3. *S.G. Shrinivas. "APPLICATIONS OF GRAPH THEORY IN COMPUTER SCIENCE AN OVERVIEW"*
   International Journal of Engineering Science and Technology,Vol. 2(9), 2010.