Python Programming for Graph Coloring Algorithms

[1]Ganga K. M. and Sreehari A., [2]K. A. Germina,

[1]DepartmentofMathematics, ChristDeemedtobeUniversity, Bangalore,
[2]DepartmentofMathematics, CentralUniversityofKerala, Kasargod, India

Abstract—In this paper, we discuss different heuristic graph coloring algorithms specifically; First fit algorithm, Welsh Powell algorithm, Largest degree ordering algorithm, Incidence Degree Ordering Algorithm, Degree of Saturation Algorithm and Recursive Largest First Algorithm. This paper pro-vides the python program that results in the coloring given by the above-mentioned algorithms.

IndexTerms*—Pythonprogram, Graphcoloringalgorithms. *(Keywords)

I. Introduction

A Graph coloring problem is a problem that involves assigning colors to the vertices of a graph. Graph coloring has a wide range of applications in scheduling, Mobile radio frequency assignment, Register Allocation, and so on. Propercoloringofagraphisassigningacolortothevertexofthegraphsuchthatno two adjacent vertices get the same color. We can color a graph properly in many ways. The question is to find the minimum number of colors used to color the graph properly. The chromatic number of the graph G given by $\chi(G)$is the minimum number of colors used to color the graph properly. The graph coloring algorithm is the steps involve dincoloring the vertice so fa graph such that the adjacent vertices do not get the same color. There are different types of vertex coloring algorithms.

II. Python Implementation of Graph Coloring Algorithms

In this paper, we concentrate on python programming of the algorithms mentionedbelow:

- **First Fit Algorithm** [1]: Arrange the vertices in some arbitrary orderand color them in the arranged order by using proper coloring.haveprovidedthestepsinvolvedinthisalgorithm.
- **Welsh Powell Algorithm** [4]: It arranges the vertices in descending order of their degrees and assigns a color to each vertex based on itsadjacency.
- **Largest Degree Ordering Algorithm** [2]: In this algorithm the verticesare ordered in descending order and proper coloring of the graph is followed in this sequential manner(Largest first colored).
- **Incidence Degree Ordering Algorithm** [3]:The idea of coloring uses arranging the vertices in descending order of their degrees and finding the number of adjacent colored vertices.
- **Degree of Saturation Algorithm** [3]: This idea is similar to that of the one used in Incidence degree ordering except that here we find the number of adjacent vertices that are colored by different colors.
- **Recursive Largest First Algorithm** [3]: It uses a recursive structure for coloring the vertices.

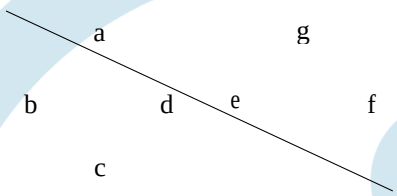
Program for Construction of Graph

```
#required modules
import networkx as nx
import numpy as np

G=nx.Graph() # define the graph G
G.add_node("a")  #define vertices
G.add_node("b")
G.add_node("c")
G.add_node("d")
G.add_node("e")
G.add_node("f")
G.add_node("g")
G.add_edge("a","b") #edge between the vertices
```

```
G.add_edge("a","d")
G.add_edge("a","g")
G.add_edge("b","c")
G.add_edge("b","d")
G.add_edge("d","e")
G.add_edge("d","c")
G.add_edge("e","f")
G.add_edge("e","g")
G.add_edge("f","g")
nx.draw_networkx(G)
```

This graph issued to test the outcome of different algorithms. We test the out come of different algorithm in the graph.



ImplementationofFirstfitAlgorithm

```
def first_fit_algo(G): # def function
  node=list(G.nodes())
  order=len(node)
  color_list=[order + 1]*order
  color_list[0]=1
  adj=nx.adjacency_matrix(G)
  Natural=np.arange(1,order+1) #list of natural numbers
  K=[]
  for i in range(1,order):
    for j in range(order):
      if adj[i,j]==1:
        K.append(color_list[j])
    diff_list=list(set(Natural)-set(K))
    color_list[i]=min(diff_list)
    K.clear()
    diff_list.clear()
    result=zip(node,color_list)
  return print(tuple(result))

first_fit_algo(G)
```

Output: Vertices and its corresponding colors are obtained by using the above method.

```
(('a', 1), ('b', 2), ('c', 1), ('d', 3), ('e', 1), ('f', 2), ('g', 3))
```

Timecomplexity: The time complexity of this programis O(n^K) .Wheren is the order of the graph. So this program has polynomial time complexity.

Implementation of Welsh-Powell Algorithm

```
def welsh_powell_algo(G):
  node=list(G.nodes())
  order=len(node)
  color_list=[order + 1]*order
  adj=nx.adjacency_matrix(G)
  Natural=np.arange(1,order+1)
  colored_vertex=[]
  K=[]
  z=[]
  for c in range(order):
    for i in range (order):
     for j in range (order):
       if adj[i,j]==1:
         if i not in colored_vertex:
           z.append(i)
    w=max(set(z),key=z.count)
    for j in range(order):
      if adj[w,j]==1:
        K.append(color_list[j])
    diff_list=list(set(Natural)-set(K))
    color_list[w]=min(diff_list)
    K.clear()
    diff_list.clear()
    colored_vertex.append(w)
    for j in range(order):
      if adj[w,j]!=1:
         if j not in colored_vertex:
           if color_list[j]== order +1:
             for i in range(order):
               if adj[i,j]==1:
                 K.append(color_list[i])
             if color_list[w] not in K:
               color_list[j]=color_list[w]
               colored_vertex.append(j)
               w=j
      K.clear()
    j=j+1
  c=c+1
  result=zip(node,color_list)
  return print(tuple(result))

welsh_powell_algo(G)
```

Output: Vertices and its corresponding colors are obtained by using the above method.

```
(('a', 2), ('b', 3), ('c', 2), ('d', 1), ('e', 2), ('f', 1), ('g', 3))
```

Timecomplexity: The time complexity of this programis O(n^K). Wheren is the order of the graph.So this program has polynomial time complexity.

Implementation of Largest Degree Ordering Algorithm

```
def largest_degree_order_algo(G):
  node=list(G.nodes())
  order=len(node)
  color_list=[order + 1]*order
  color_list[0]=1
  deg_seq=sorted(G.degree, key=lambda  x: x[1], reverse=True)
  n=0
  vertex_deg_seq=[x[n] for x in deg_seq]
  adj=nx.adjacency_matrix(G,nodelist=vertex_deg_seq)
  Natural=np.arange(1,order+1)
  dup_list=[]
  K=[]
  for i in range(1,order):
    for j in range(order):
      if adj[i,j]==1:
        K.append(color_list[j])
    diff_list=list(set(Natural)-set(K))
    color_list[i]=min(diff_list)
    K.clear()
    diff_list.clear()
  result=zip(vertex_deg_seq,color_list)
  return print(tuple(result))

largest_degree_order(G)
```

Output: Vertices and its corresponding colors are obtained by using the above method.

```
(('d', 1), ('a', 2), ('b', 3), ('e', 2), ('g', 1), ('c', 2), ('f', 3))
```

Timecomplexity: The time complexity of this programis O(n^K). Wheren is the order of the graph. So this program has polynomial time complexity.

Implementation of Incidence Degree Ordering Algorithm

```
def incident_deg_algo(G):
  node=list(G.nodes())
  order=len(node)
  color_list=[order + 1]*order
  deg_seq=sorted(G.degree, key=lambda  x: x[1], reverse=True)
  n=0
  vertex_deg_seq=[x[n] for x in deg_seq]
  color_list[0]=1
  adj=nx.adjacency_matrix(G,nodelist=vertex_deg_seq)
  Natural=np.arange(1,order+2)
  dup_list=[]
  colored_vertex=[]
  colored_vertex.append(0)
```

```
  s=[]
  x=[]
  for v in range(order):
    p=[]
    for i in range(order):
      for j in range(order):
        if adj[i,j]==1:
          if color_list[j]!=order+1:
            p.append(i)
      j=j+1
    i=i+1
    for n in p:
      if n not in colored_vertex:
       x.append(n)
    if len(x)!=0:
      w=max(set(x),key=x.count)
    colored_vertex.append(w)
    for m in range(order):
        if adj[w,m]==1:
          s.append(color_list[m])
    diff_list=list(set(Natural)-set(s))
    color_list[w]=min(diff_list)
    diff_list.clear()
    s.clear()
    x.clear()
  v=v+1
  result=zip(vertex_deg_seq,color_list)
  return print(tuple(result))

incident_deg_algo(G)
```

Output: Vertices and its corresponding colors are obtained by using the above method.

```
(('d', 1), ('a', 2), ('b', 3), ('e', 2), ('g', 1), ('c', 2), ('f', 3))
```

Timecomplexity: Thetimecomplexityofthisprogramis$O(n^K)$.Wherenistheorderofthegraph.So thisprogramhaspolynomialtimecomplexity.

ImplementationofDegree of Saturation Algorithm

```
def deg_sat_algo(G):
  node=list(G.nodes())
  order=len(node)
  color_list=[order + 1]*order
  deg_seq=sorted(G.degree, key=lambda  x: x[1], reverse=True)
  n=0
  vertex_deg_seq=[x[n] for x in deg_seq]
  color_list[0]=1
  adj=nx.adjacency_matrix(G,nodelist=vertex_deg_seq)
  Natural=np.arange(1,order+1)
  colored_vertex=[]
```

```
  s=[]
  x=[]
  z=[]
  for v in range(1,order):
    p=[]
    for i in range(1,order):
        for j in range(order):
          if adj[i,j]==1:
            if color_list[j]!=order + 1:
              p.append(color_list[j])
              set(p)
        for n in p:
            x.append(i)
        p.clear()
    for q in x:
      if q not in colored_vertex:
        z.append(q)
    if len(z)!=0:
      w=max(set(z),key=z.count)
    colored_vertex.append(w)
    x.clear()
    for m in range(order):
      if adj[w,m]==1:
        s.append(color_list[m])
    diff_list=list(set(Natural)-set(s))
    color_list[w]=min(diff_list)
    s.clear()
    diff_list.clear()
    x.clear()
    z.clear()
  v=v+1
  result=zip(vertex_deg_seq,color_list)
  return print(tuple(result))

deg_sat_algo(G)
```

Output: Vertices and its corresponding colors are obtained by using the above method.

```
(('d', 1), ('a', 2), ('b', 3), ('e', 2), ('g', 1), ('c', 2), ('f', 3))
```

Timecomplexity: The time complexity of this programis O(n^K). Wheren is the order of the graph. So this program has polynomial time complexity.

Implementation of Recursive Largest First Algorithm

```
def recursive_lf_algo(G):
  node=list(G.nodes())
  order=len(node)
  color_list=[order]*order
  deg_seq=sorted(G.degree, key=lambda  x: x[1], reverse=True)
```

```
  n=0
  vertex_deg_seq=[x[n] for x in deg_seq]
  color_list[0]=1
  adj=nx.adjacency_matrix(G,nodelist=vertex_deg_seq)
  Natural=np.arange(1,order+1)
  colored_vertex=[]
  K=[]
  z=[]
  u=[]
  v=[]
  w=0
  colored_vertex.append(0)
  for c in range(1,order):
    for j in range(order):
      if adj[w,j]==1:
         u.append(j)
      else:
         v.append(j)
    for q in v:
      for r in u:
        if adj[q,r]==1:
          if q not in colored_vertex:
            z.append(q)
    if len(z)!=0:
      w=max(set(z),key=z.count)
    colored_vertex.append(w)
    for m in range(order):
        if adj[w,m]==1:
          K.append(color_list[m])
    diff_list=list(set(Natural)-set(K))
    color_list[w]=min(diff_list)
    K.clear()
    diff_list.clear()
    z.clear()
    u.clear()
    v.clear()
  c=c+1
  result=zip(vertex_deg_seq,color_list)
  for b in range(order):
    if color_list[b]== order:
      for i in range(1,order):
        for j in range(order):
            if adj[i,j]==1:
              K.append(color_list[j])
        diff_list=list(set(Natural)-set(K))
        color_list[i]=min(diff_list)
        K.clear()
        diff_list.clear()
  return print(tuple(result))

recursive_lf_algo(G)
```

Output: Vertices and its corresponding colors are obtained by using the above method.
Time complexity: The time complexity of this programis O(n^K). Where n is the order of

```
(('d', 1), ('a', 3), ('b', 2), ('e', 2), ('g', 1), ('c', 3), ('f', 3))
```

the graph. So this program has polynomial time complexity.

III. Conclusion

In this paper, we introduce six different python programming for the heuristic algorithms in which each algorithm gives different colors when applied. It has been found that the programs are in polynomial time and their time complexity is given by $O(n^k)$ where n is the order of the graph.

References

[1] H. Al-Omari and K. E. Sabri, "New graph coloring algorithms," *AmericanJournalofMathematics andStatistics*,vol.2,no.4,pp.739–741,2006.

[2] A. Mansuri, V. Gupta, and R. Chandel, "Coloring programs in graph the-ory,"*Int.Journal ofMath.Analysis*,vol.4,no.50,pp.2473–2479,2010.

[3] M.AslanandN.A.Baykan,"Aperformancecomparisonofgraphcoloringalgorithms,"*InternationalJournal ofIntelligentSystemsandApplicationsinEngineering*,vol.4,no.SpecialIssue-1,pp.1–7,2016.

[4] "Welshpowellgraphcolouringalgorithm."(),[Online].Available:https://www.geeksforgeeks.org/welsh-powell-graph-colouring-algorithm/.(accessed:01.07.2022).