Research in Graph Colorings

Abdulhai Naqvi

These past two quarters, Winter and Spring 2017, I worked on a problem in graph theory. The problem could be stated as follows: One is given a graph, an associated coloring and a starting vertex. Our goal is to see if the coloring can be changed to all the same colors provided a target color, given there are a finite number of possible total colors. The only rules are that one can move only to a neighboring vertex and that the color of the vertex that is left is increased by 1. If the number reaches the maximum possible color, it wraps around to 0. The question we ask is if this is possible for all graphs, and if not, then are there certain kinds of graphs for which this can be done.

I wrote a simulation of this problem in Python. I used this simulation to solve the case of cycle graphs. I was able to find a proof that this property holds for cycle graphs using induction. This prove is specifically for the case when backtracking is allowed in the algorithm. The case without backtracking is unsolved as of yet. However, the program provided here can be used to see if this property holds for other graph families or not.

The program, written in the Python programming language, Version 3, follows.

```python
#Author: Abdulhai Naqvi
def fromBaseTen(n, base): # Adapted from an SO answer. To be used to decode array index back to coloring
    if n == 0:
        return ''
    else:
        e = n//base
        q = n%base
        return fromBaseTen(e, base) + str(q)


class state:
    def __init__(self,  vertex,  coloring,  parent):
        self.vertex = vertex
        self.coloring = coloring
        self.parent = parent


def findpos(states, newstate, k, n): # hash function to store entry in states
    # divide states into n comaprtments for each starting position
    pos = int(''.join(map(str,newstate.coloring)), k) + int(newstate.vertex*len(states)/n)
    return pos

def checkstate(states,  newstate, k, n): # check if given state is in states
    pos = findpos(states, newstate, k, n)
    return states[pos]


# usage: BFS(graph, starting_vertex, max_num_colors, coloring, backtrack_option)

def BFS(G, v, k,  colors, allow_backtrack):
    if len(G) == 0 or len(G) == 1:
        return 0;
    # No states visited yet
    G = G[:-1]
    n = len(colors) # no. of vertices
    states = [False for i in range(n*(k**n))]
    # Create a queue for BFS
    queue = []
    s = state(v,  colors,  None) # encode color info at time of visiting vertex

    # Mark the source node as visited and enqueue it
    queue.append(s)
    ##
    #print("------------\n\n",k, n)

    states[findpos(states, s, k, n)] = True

    while queue:
        s = queue.pop(0)
        coloring = s.coloring
        # increase colors by one
        # Get all adjacent vertices of the dequeued
        # vertex v. If an adjacent vertex has not been visited,
        # then mark it visited and enqueue it
        for i in G[s.vertex]:
            if not allow_backtrack and i == s.parent: # don't jump back to parent node
                continue
            newColoring = coloring[:]
            newColoring[s.vertex] = (newColoring[s.vertex] - 1)% k # increase current vertex's color by one
            newState = state(i,  newColoring,  s.vertex)
            if checkstate(states, newState, k, n):
                continue # discard current state as it's already been visited

            queue.append(newState)
```

```
65              states[findpos(states, newState, k, n)] = True # find pos of state in array
                (find hash value)
66      return states
67
68      # return list of all colorings that can reach the target coloring
69  # defining cycle graphs
70  c3 = [[1,2], [0,2], [0,1], 'c3']
71  c4 = [[1,3], [0,2], [1,3], [0,2],'c4'] # notice the repetition.
72  c5 = [[4,1], [2,0], [1,3], [2,4], [3,0],'c5']
73  c6 = [[5,1], [2,0], [1,3], [2,4], [3,5], [4,0], 'c6']
74  k3 = [[1, 2], [0, 2], [0, 1],  'k3']
75  k4 = [[1, 2, 3], [0, 2, 3], [0, 1, 3], [0, 1, 2], 'k4']
76  w3 = [[1,2,3], [0,2,3], [0,1,3], [0,1,2], 'w3']
77  w4 = [[1,3,4], [0,2,4], [1,3,4], [0,2,4], [0,1,2,3], 'w4']
78  w5 = [[4,1,5], [2,0,5], [1,3,5], [2,4,5], [3,0,5], [0,1,2,3,4], 'w5']
79  w6 = [[5,1,6], [2,0,6], [1,3,6], [2,4,6], [3,5,6], [4,0,6], [0,1,2,3,4,5], 'w6']
80
81
82  #states = BFS(c5, 0, 4, (len(c5)-1)*[0], 1)
83  #print some results
84  for graph in [w3, w4, c3, k4]:
85      for k in range(2,3):
86          states = BFS(graph, 0, k, (len(graph)-1)*[0], 0)
87          unruly_states = list([index for index, item in enumerate(states) if item is
            False])
88          print('graph: ',graph[-1], ', k: ', k, ', # of states: ', len(states), ', # of
            states reachable: ', sum(states), ', unreachable states: ', ','.join(list((
            fromBaseTen(n%(k**(len(graph)-1)), k)) for n in unruly_states)), ', start
            vertex: ', ','.join(list((str(n//(k**(len(graph)-1))) for n in unruly_states))))
89
90
```

Proof For Cycle Graphs

1 Conjecture

Given a Cyclic graph, C_n, with n vertices, and an initial coloring; for any graph that is solvable, a graph of $n+2$ vertices is also solvable. Solvable here means that, starting from an initial coloring, the graph can be changed to an all zero coloring given the rules that one can go to any neighbor starting at an arbitrary vertex and increase the color of this neighbor by one modulo k, where k is the maximum number of colors.

2 Proof: By Induction

For $n = 3$, the companion Python program shows that all colorings of C_3 are solvable for a given k and if n is odd. For $n = 4$, the program solves nk^n/k of all nk^n possible colorings or $1/k$ of all possible colorings.

We assume the induction hypothesis, that a graph with n vertices is solvable.

We need to show that a graph with $n+2$ vertices is also solvable. Since we assume the induction hypothesis, we know that the graph with n vertices is solvable. If we take out one of the edges and insert 2 more vertices to form C_{n+2} from C_n, we know that n of the vertices have the same color, zero. We can call these new vertices a and b.

Since the newly added vertices are adjacent to each other and all other vertices have color zero, we can contract all the vertices with color zero to one vertex and perform all operations simultaneously. This reduces down to the base case if we stopped at a vertex adjacent to one of the newly added vertices in the induction hyppthesis. Since we know that every vertex except a and b has color zero, we can contract the graph to have five vertices, assuming we start at a vertex that is not adjacent to either a or b. We can call the starting vertex i. We can contract all the vertices between i and a and call it u and the vertices between i and b and call it v. We go from i to v and back to i. We do this $k-1$ times. The color of v and i is now $k-1$. We are still on i. Now, we go to u and change it's color to one. We now go from u to a and back $k-1$ times, leaving the color of u to be zero. Now, we go back from u to i, changing it's color to zero and finally we move on to v, changing it's color to zero as well. At this point, we can contract v, i and u and treat them as one vertex. This also boils down to the base case.