Anej Rozman, Tanja Luštrek

Rich-Neighbor Edge Colorings

Term Paper in Finance Lab

Long Presentation

Advisers: Assistant Professor Janoš Vidali,
Professor Riste Škrekovski

Ljubljana, 2023

Contents

1. Introduction

In this paper we set out to analyse an open conjecture in a modern graph theory problem known as rich-neighbor edge coloring.

Definition 1.1. In an edge coloring, an edge e is called *rich* if all edges adjacent to e have different colors. An edge coloring is called a *rich-neighbor edge coloring* if every edge is adjacent to some rich edge.

Definition 1.2. $X'_{rn}(G)$ denotes the smallest number of colors for which there exists a rich-neighbor edge coloring.

Conjecture 1.3. *For every graph G of maximum degree Δ, $X'_{rn}(G) \leq 2\Delta - 1$ holds.*

Example 1.4. Let's take a look at the Petersen graph and an example of a rich-neighbor edge coloring.
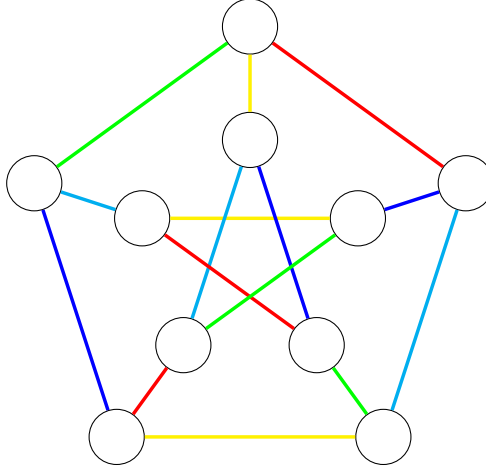
We can see that for the Petersen graph (which is 3-regular) we can find an appropriate coloring with 5 colors so $X'_{rn} \leq 5 \leq 2 \cdot 3 - 1 = 5$. This shows that the conjecture holds for this graph. $\diamond$

2. ALGORITHMS

2.1. INTEGER PROGRAMMING

Using SageMath we constructed an integer programming model that finds a rich-neighbor edge coloring for a given graph. Our interger program looks like this:

minimize t we minimize the number of colors we need

subject to $\forall e: \quad \sum_{i=1}^{2\Delta-1} x_{ei} = 1$ each edge is exactly one color

$\forall i \ \forall u \ \forall v, w \sim u, v \neq u: \quad x_{uv,i} + x_{uw,i} \leq 1$

edges with the same vertex are a different color

$\forall e \ \forall i: \quad x_{ei} \cdot i \leq t$ we use less or equal to t colors

$\forall i \ \forall uv \ \forall w \sim u, w \neq v \ \forall z \sim v, z \neq u, w: \quad x_{uw,i} + x_{vz,i} + y_{uv} \leq 2$

uv is a rich edge $\Leftrightarrow$ all adjacent edges are a different color

$\forall e: \quad \sum_{f \sim e} y_f \geq 1$ every edge is adjacent to some rich edge

$\forall e: \quad 0 \leq y_e \leq 1, \ y_e \in \mathbb{Z}$

$\forall e \ \forall i: \quad 0 \leq x_{ei} \leq 1, \ x_{ei} \in \mathbb{Z},$

where

$$x_{ei} = \begin{cases} 1, & \text{if edge } e \text{ has color } i \\ 0, & \text{otherwise} \end{cases} \quad \text{and} \quad y_e = \begin{cases} 1, & \text{if edge } e \text{ is rich} \\ 0, & \text{otherwise.} \end{cases}$$

In our implementation of the ILP we fix the number of colors to $2\Delta - 1$, since the coloring can't be made with less colors (every edge has $2\Delta - 2$ neighboring edges and the edge itself has to be a different color) and any more colors do not satisfie the conjecture. Therefore we only check if the program has a solution that satisfies all the constraints and the conjecture. In theory this doesn't make the program faster, but in our practice tests it did make a significant differnece.

Our implementation of the IPL is demonstrated in the following code.

Algorithm 1: richNeighbor

Input: Graph G

Output: Colors $colors$, Rich edges $richEdges$

$p \leftarrow \text{MixedIntegerLinearProgram}(\text{maximization} = \text{False})$

$x \leftarrow p.new\_variable(binary = True)$

$y \leftarrow p.new\_variable(binary = True)$

$t \leftarrow p.new\_variable(integer = True)$

$p.set\_objective(t[0])$

$\text{maxCol} \leftarrow 2 \cdot G.degree()[0] - 1$

$p.add\_constraint(t[0] = \text{maxCol})$

for e *in* $G.edges(labels = False)$ **do**

    $p.add\_constraint(\sum_{i=1}^{\text{maxCol}} x[\text{Set}(e), i] = 1)$

for (u, v) *in* $G.edges(labels = False)$ **do**

    $p.add\_constraint(\sum_{j \in G[u]} y[\text{Set}((u, j))] + \sum_{l \in G[v]} y[\text{Set}((l, v))] - 2y[\text{Set}((u, v))] \geq 1)$

for e *in* $G.edges(labels = False)$ **do**

    for i *in* 1 *to* $maxCol$ **do**

        $p.add\_constraint(i \cdot x[\text{Set}(e), i] \leq t[0])$

for i *in* 1 *to* $maxCol$ **do**

    for (u, v) *in* $G.edges(labels = False)$ **do**

        for w *in* $G[u]$ **do**

            if $w = v$ **then**

                continue

            $p.add\_constraint(x[\text{Set}((u, v)), i] + x[\text{Set}((u, w)), i] \leq 1)$

        for z *in* $G[v]$ **do**

            if $z = u$ **then**

                continue

            $p.add\_constraint(x[\text{Set}((u, v)), i] + x[\text{Set}((v, z)), i] \leq 1)$

for (u, v) *in* $G.edges(labels = False)$ **do**

    for w *in* $G.neighbors(u)$ **do**

        for z *in* $G.neighbors(v)$ **do**

            if $w = v$ *or* $z = u$ **then**

                continue

            for i *in* 1 *to* $maxCol$ **do**

                $p.add\_constraint(x[\text{Set}((u, w)), i] + x[\text{Set}((v, z)), i] + y[\text{Set}((u, v))] \leq 2)$

return $colors$, $richEdges$

2.2. Iterative Algorithm

We also implemented an iterative algorithm that finds a rich-neighbor edge coloring for a given graph. The algorithm works by assigning a color to an edge and then checking if the coloring is valid. If it is, we move on to the next edge, otherwise we try a different color. If we can't find a valid coloring with $2\Delta - 1$ colors, we increase the number of colors by one and try again.

The algorithm is not guaranteed to find a coloring with $2\Delta - 1$ colors, but it is guaranteed to find a coloring with 2Δ colors.

The algorithm is very fast for footnotesize graphs, but it becomes too slow for graphs with more than 10 vertices. Ta tekst so smeti ma tle bi pc mogli mal blefirat da je iterativn algoritm

pocasnejsi od ILPja.

3. Complete search

Vertices	Degree 4	Degree 5	Degree 6	Degree 7
5	1	0	0	0
6	1	1	0	0
7	2	0	1	0
8	6	3	1	1
9	16	0	4	0
10	59	60	21	5
11	265	0	266	0
12	1544	7848	7849	1547
13	10778	0	367860	0
14	88168	3459383	21609300	21609301
15	805491	0	1470293675	0
16	8037418	2585136675	113314233808	733351105934

TABLE 1. Number of k-regular graphs on n vertices

Magari vkluci se tabelo ki kaze kako hitro raste stevilo vseh povezanih grafov na n vozlih. Nasploh pa mal raztegni besedilo.

3.1. Graph generation

Na dolgo, povej da je riste dal njegove .txt datoteke kjer so vsi k-regularni grafi na n-vozliscih.

4. Random Search

Along with testing the hypothesis for smaller graphs, we were also interested in checking if the conjecture holds for larger graphs. Since we can't even begin to check all for exmaple 20-regular graphs on 100 vertices, a random search algorytm seemed as a natural continuation of our problem. And since the we aren't looking for an optimum solution (anything more than the minimum $2\Delta - 1$ disproves the conjecture), we opted for a modification approach. We start with a random graph, check if the conjecture holds, and then modify it in a way that preserves the regularity and connectedness. We repeat this process indefinetly. Since ....dodaj razlog......., on every iteration, there is a footnotesize probability that we generate a completely new random graph and start again. Razlog je, da ce je v grafu obstaja rich neighbor edge coloring potem je vecja verjetnost da bo v bliznjih grafih prav tako obstajala, zato dopuscamo manjso moznost da popolnoma na novo generiramo graf.

Opisi algoritem

4.1. Graph generation

Since we only need one graph to start our iterative process, we can generate it randomly using the built in sage function `graphs.RandomRegular`.

5. Preventiva

tukej daj algoritma chack coloring in check richness. Ostalo bom js napisu

6. Findings

7. Conclusion

Algorithm 2: tweak

Input: Graph G
Output: Tweaked graph T

1 $T \leftarrow$ graph.copy()
2 $e_1 \leftarrow T$.random_edge()
3 $u_1, v_1, \text{extra}_1 \leftarrow e_1$
4 T.delete_edge(e_1)
5 $e_2 \leftarrow T$.random_edge()
6 $u_2, v_2, \text{extra}_2 \leftarrow e_2$
7 T.delete_edge(e_2)
8 $p \leftarrow$ random()
9 if $p < 0.5$ **then**
10 $\quad T$.add_edge(u_1, u_2)
11 $\quad T$.add_edge(v_1, v_2)
12 else
13 $\quad T$.add_edge(u_1, v_2)
14 $\quad T$.add_edge(v_1, u_2)
15 if *not T.is_connected*() **then**
16 $\quad$ **return** tweakGraph(graph)
17 return T