UNIVERSITY OF LJUBLJANA
FACULTY OF MATHEMATICS AND PHYSICS

Financial mathematics – 1st cycle

Tanja Luštrek, Anej Rozman

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.

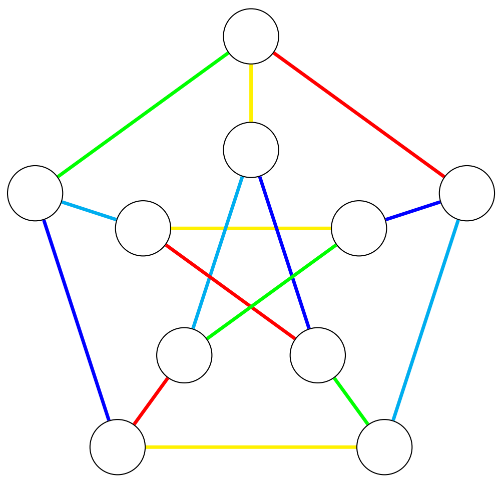
Definition 1.1. In an edge coloring, an edge e is called *rich* if all edges adjacent to e have different colors. An edge coloring is called a *rich-neighbor edge coloring* if every edge is adjacent to some rich edge.

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.*

In the paper we focus on analysing the conjecture only for regular graphs. We plan to do this by implementing an integer programming model that finds a rich-neighbor edge coloring for a given graph using the smallest number of colors possible.

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. $\diamondsuit$

2. Algorithms

2.1. Integer Programming

Using SageMath we implement an integer program that finds a rich-neighbor edge coloring for a given graph using the smallest number of colors possible. Writen matematically, 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

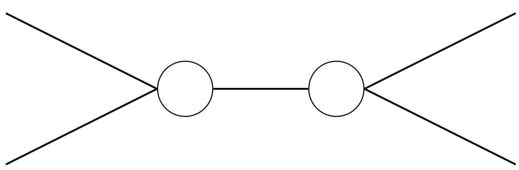
$\forall e: \quad 0 \leq y_e \leq 1,\ y_e \in \mathbb{Z}$ — solutions are binary variables

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

where

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

In our implementation of the ILP we fix the number of colors to $2\Delta - 1$ by adding the following constraint $t = 2\Delta - 1$. We do this since the coloring can't be made with less colors, because every edge has $2\Delta - 2$ neighboring edges and the edge itself has to be a different color.



And since any coloring with more than $2\Delta - 1$ colors does not satisfy the conjecture, we only check if the program has a solution that satisfies all the constraints. In theory this doesn't make the program faster, but in our practice tests it did make a significant differnece.

Our actual implementation of the ILP is demonstrated in the following code.

Algorithm 1: richNeighbor

```
Input: graph G
Output: colors of edges colors, rich edges richEdges
p ← MixedIntegerLinearProgram(maximization = False)
x ← p.new_variable(binary = True)
y ← p.new_variable(binary = True)
t ← p.new_variable(integer = True)
p.set_objective(t[0])
maxCol ← 2 · G.degree()[0] − 1
p.add_constraint(t[0] = maxCol)
for e in G.edges(labels = False) do
    p.add_constraint(∑_{i=1}^{maxCol} x[Set(e), i] = 1)
for (u, v) in G.edges(labels = False) do
    p.add_constraint(∑_{j∈G[u]} y[Set((u, j))] + ∑_{l∈G[v]} y[Set((l, v))] −
      2y[Set((u, v))] ≥ 1)
for e in G.edges(labels = False) do
    for i in 1 to maxCol do
        p.add_constraint(i · x[Set(e), i] ≤ 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[Set((u, v)), i] + x[Set((u, w)), i] ≤ 1)
        for z in G[v] do
            if z = u then
                continue
            p.add_constraint(x[Set((u, v)), i] + x[Set((v, z)), i] ≤ 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[Set((u, w)), i] + x[Set((v, z)), i] +
                  y[Set((u, v))] ≤ 2)
return colors, richEdges
```

Example 2.1. Using the `richNeighbor` algorithm on the Petersen graph gives us the following output.

$$\begin{aligned}
colors = \{&(\{0,1\},1) : 0.0, (\{0,1\},2) : 0.0, (\{0,1\},3) : 0.0, (\{0,1\},4) : 0.0,\\
&(\{0,1\},5) : 1.0, (\{0,4\},1) : 0.0, (\{0,4\},2) : 0.0, (\{0,4\},3) : 1.0,\\
&\dots\\
&(\{9,6\},3) : 0.0, (\{9,6\},4) : 0.0, (\{9,6\},5) : 0.0, (\{9,7\},1) : 0.0,\\
&(\{9,7\},2) : 0.0, (\{9,7\},3) : 0.0, (\{9,7\},4) : 1.0, (\{9,7\},5) : 0.0\}
\end{aligned}$$

$$\begin{aligned}
richEdges = \{&\{0,1\} : 1.0, \{0,4\} : 0.0, \{0,5\} : 0.0, \{1,2\} : 1.0, \{1,6\} : 0.0,\\
&\{3,4\} : 0.0, \{9,4\} : 1.0, \{5,7\} : 0.0, \{8,5\} : 1.0, \{2,3\} : 1.0,\\
&\{2,7\} : 0.0, \{8,6\} : 0.0, \{9,6\} : 0.0, \{8,3\} : 1.0, \{9,7\} : 1.0\}
\end{aligned}$$

Both *colors* and *richEdges* are dictionaries and their key-value pairs are structured like

$$(e,i) : \begin{cases} 1.0, & \text{if edge } e \text{ is color } i \\ 0.0, & \text{otherwise} \end{cases} \qquad \text{and} \qquad e : \begin{cases} 1.0, & \text{if edge } e \text{ is rich} \\ 0.0, & \text{otherwise.} \end{cases}$$

$\diamondsuit$

2.2. Iterative Algorithm

We debated implementing an iterative algorythm that finds a rich neighbor edge coloring, but we were not able to come up with anything that runs in polynomial time. From [1] we know that integer linear programs are NP-complete problems, so it did't make much sense implementing another slow algorythm, since it wouldn't make much of a difference in the number of graphs we would be able to check.

3. Complete search

As illustrated in Table 1, the number of k-regular graphs on n vertices grows exponentially with n. This poses a computational challenge, as checking all graphs becomes impossible for larger values of n. However, we can still check all graphs for smaller values of n up to 13.

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 [2]

3.1. Graph generation

Graphs for the complete search were taken from a collection of files, provided by Professor Škrekovski. The files contain all k-regular graphs on n vertices up to 15 vertices, since the number of graphs beyond that is too much to handle.

4. Random Search

In addition to examining the hypothesis for smaller graphs, we were also interested in checking if the conjecture holds for larger graphs. The challenge here is testing all possible colorings in graphs with many vertices. Recognizing the enormity of this task, a random search algorithm seemed like a natural continuation of our problem. Here we opted for a modification approach.

We start with a random graph and use our ILP to check if the conjecture holds for it. Then, we modify the graph in a way that preserves its regularity and connectedness. We repeat this process indefinitely, or, actually, until we stop our program. We know that if a rich-neighbor edge coloring exists in a given graph, there is a higher probability that it also exists in similar graphs. Therefore, on every iteration, we keep a small probability that we generate a completely new random graph and start again.

Written below is the algorithm `tweak` for modifying graphs. First it selects two random edges and deletes them. Then it generates a random number p from $[0, 1)$ and based on the value of p it adds back two different edges in one of two possible ways that preserve the regularity of the graph. There is a chance that the newly formed graph is not connected, so we check for that and if it is not, we restart the process until we get a connected graph.

Algorithm 2: tweak

Input: graph G
Output: tweaked graph T
$T \leftarrow$ graph.copy()
$e_1 \leftarrow T$.random_edge()
$u_1, v_1, \leftarrow e_1$
T.delete_edge(e_1)
$e_2 \leftarrow T$.random_edge()
$u_2, v_2, \leftarrow e_2$
T.delete_edge(e_2)
$p \leftarrow$ random()
if $p < 0.5$ **then**
  T.add_edge(u_1, u_2)
  T.add_edge(v_1, v_2)
else
  T.add_edge(u_1, v_2)
  T.add_edge(v_1, u_2)
if *not T.is_connected()* **then**
  return tweak(G)
return T

4.1. Graph generation

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

5. Checking the coloring

In order to make sure that our ILP implentation is without fault, we also implemented the functions `checkColoring` and `checkRichness` which check if our program always returns a proper coloring, and if the coloring is actually a rich-neighbor edge coloring.

`checkColoring` takes a graph G and the output *colors* of our ILP. It iterates over each vertex in the graph, checks its neighbors and returns false if a color repeats on multiple of that edges.

Algorithm 3: checkColoring

```
Input: graph G, colors of edges coloring
Output: Boolean indicating proper coloring
for v in G.vertices() do
    col ← set()
    for w in G.neighbors(v) do
        for i in range(1, 2 · G.degree()[0]) do
            if coloring[(Set((v, w)), i)] = 1 then
                col.add(i)
    if len(col) ≠ len(G.neighbors(v)) then
        return False
return True
```

`checkRichness` takes a graph G and the output *richEdges* of our ILP. It iterates over each edge in the graph, checks its neighbors and returns false if it doesn't find a rich neighboring edge for each edge.

Algorithm 4: checkRichness

```
Input: graph G, rich edges richEdges
Output: Boolean indicating proper rich-neighbor edge coloring
for (u, v) in G.edges(labels = False) do
    S ← 0
    for w in G.neighbors(u) do
        if w = v then
            S ← S + richEdges[Set((u, w))]
    for z in G.neighbors(v) do
        if z = u then
            S ← S + richEdges[Set((v, z))]
    if S = 0 then
        return False
return True
```

6. Findings

7. Conclusion

References

[1] Ravindran Kannan and Clyde L. Monma. On the computational complexity of integer programming problems. In Rudolf Henn, Bernhard Korte, and Werner Oettli, editors, *Optimization and Operations Research*, pages 161–172, Berlin, Heidelberg, 1978. Springer Berlin Heidelberg.

[2] M. Meringer. Fast generation of regular graphs and construction of cages. *Journal of Graph Theory*, 30:137–146, 1999.