A Simple Heuristic for Finding k-Nowhere Zero Flows

Arnaud Lequen
Ecole Normale Supérieure de Rennes
arnaud.lequen@ens-rennes.fr

ABSTRACT

This papers focuses on finding a $\mathbb{Z}/k\mathbb{Z}$-nowhere-zero flow on a given graph. To achieve that, we propose several heuristics that rely on graph algorithms, to find such a flow as efficiently as possible, but not deterministically.

CCS CONCEPTS

• **Mathematics of computing** → **Graph algorithms**.

KEYWORDS

Nowhere-zero-flows, Graphs, Solver, Heuristic

ACM Reference Format:
Arnaud Lequen. 2018. A Simple Heuristic for Finding k-Nowhere Zero Flows.

1 INTRODUCTION

The common notion of flows in $\mathbb{N}$ or $\mathbb{R}^+$ can be generalized to any finite abelian group Γ, in some respect. Informally, a Γ-flow over the graph G is an assignation of a value of Γ to every edge of G, such that, for every vertex v, the added value of the edges out of v is equal to the added value of the edges into v.

As no order can be found on finite groups, the problem of maximizing a flow over a flow network is irrelevant. Yet, another interesting problem emerges: for a given graph G, is there a Γ-flow over G such that no edge is assigned the value 0?

A Γ-flow that never takes the value 0 is called a Γ-nowhere-zero flow (Γ-NZF). Various results on the existence of such a flow on a given graph have been proven in the second half of the 19^{th} century, and the NP-hardness of the problem of computing a Γ flow for certain groups has been proven.

In this paper, we will focus on $\mathbb{Z}/k\mathbb{Z}$ flows, for $k \geq 2$. Indeed, we present an algorithm that is based on several heuristics, whose aim is to find a $\mathbb{Z}/k\mathbb{Z}$-NZF for a given graph G and a given k.

Firstly, we give a few formal definitions, as well as some results that help understands some of our choices and the interest of the paper. Then, we present our algorithm, detailing the various heuristics that we use. After that, we present the results of the experimental evaluation of an implementation of our algorithm. Then, we mention further ideas to improve the program.

SPA Project, December 2018,

2 STATE OF THE ART

For a start, let us define formally what a Γ-NZF is.

Definition 1. Let Γ a finite abelian group and $G(V, E)$ a graph. A Γ-Nowhere-Zero-Flow is a function $\Phi : E \longmapsto \Gamma \setminus \{0_\Gamma\}$ such that, for every vertex $v \in E$,

$$\sum_{e \in \delta^+(v)} \Phi(e) = \sum_{e \in \delta^-(v)} \Phi(e)$$

where $\delta^+(v)$ denotes the edges into v and $\delta^-(v)$ the edges out of v. Such a constraint is called the Kirchoff's law for v.

We have several interesting results about Γ-NZFs. Tutte proved, in 1953, that the orientations of the edges of G don't have an incidence on the existence of a Γ-NZFs over that graph. In particular, if we can find a Γ-NZF for some orientation of G, then every orientation of G admits a Γ-flow. A seminal idea to show that lies in the fact that, if we find a Γ-flow for some orientation of G, then we can change the orientation of $e \in E$, and the flow remains correct if we replace $\Phi(e)$ by $-\Phi(e)$.

Tutte also proved that the existence of a Γ-NZF doesn't depend on the structure of the group Γ. Indeed, it has been shown that if Γ and Γ' are two finite abelian groups such that $|\Gamma| = |\Gamma'|$, then G admits a Γ-NZF iff it admits a Γ'-NZF. This is why we have choosen to develop a solver for $\mathbb{Z}/k\mathbb{Z}$-NZF: such a family of groups is easy to work with, and the existence of a $\mathbb{Z}/k\mathbb{Z}$-NZF solely depends on k. A constructive demonstration of the result relies on the existence of a bijection between $\mathbb{Z}/k\mathbb{Z}$ and every group of the form $\prod_i \mathbb{Z}/k_i\mathbb{Z}$, with $\prod_i k_i = k$.

A few properties can help us decide whether a graph G admits a $\mathbb{Z}/k\mathbb{Z}$-NZF or not. A direct consequence of the definition allows us to say that if there is a bridge in G, then the graph admits no NZF, whatever the graph. A bridge is an edge such that, when removed, a new connex component is created.

In the general case, it is NP-hard to decide the existence of and compute a Γ-NZF on a random graph G. But Seymour showed in 1981 that, if a graph is bridgeless, then for every $k \geq 6$, it admits a $\mathbb{Z}/k\mathbb{Z}$-NZF. Although it is the best result we have so far, Tutte conjectued, in 1954, that every bridgeless graph admits a $\mathbb{Z}/5\mathbb{Z}$-NZF. This conjecture remains an open question to this day.

When G is planar, there is an interesting way to compute a $Z/4\mathbb{Z}$-NZF (and thus a $\mathbb{Z}/5\mathbb{Z}$-NZF). In 1954, Tutte showed

that the dual problem of computing a $\mathbb{Z}/k\mathbb{Z}$-NZF on G is to color its faces using k colors. It has also been showed that converting a k-coloration of the dual to a $\mathbb{Z}/k\mathbb{Z}$-NZF on G can be made in $O(m)$. Using the four-colors theorem, we can deduce that every planar graph admits a $\mathbb{Z}/k\mathbb{Z}$-NZF.

As the problem can be easily reduced to a set of linear equations and passed to a finite fields solver, no solver for this particular problem exists. Nevertheless, we decided to focus on this problem, and to solve it using graph algorithms.

3 APPROACH

Let G a graph without asymetric edges (for the sake of simplicity) and $k \geq 2$ a natural number. The outline of the algorithm is to start from an initial valuation of the edges, and then to refine it so much so that it becomes a $\mathbb{Z}/k\mathbb{Z}$-NZF. For the later part, many different heuristics are combined, and they try to satisfy both Kirchoff's law and the non-zero constraints.

3.1 Eliminating unsolvable graphs

Previous results showed that in some cases, it is fairly easy to conclude that a graph admits no $\mathbb{Z}/k\mathbb{Z}$-NZF. If a bridge is found in a graph G, the we can directly prune out the existence of any $\mathbb{Z}/k\mathbb{Z}$-NZF. Thus, the first step of the algorithm is to check for the existence of bridges in G.

A naive algorithm is in $O(|E| \cdot |V|)$. For every edge, we remove it from the graph and then check if the number of connex components of the graph is greater than before. If not, then we put it back and proceed to do so for every other edge. A better, yet more complex, algorithm exists, as found by Tarjan. Deciding whether a bridge exists in a graph can be done in $O(n)$.

In the particular case where $k = 2$, it is easy to check if a bridgeless graph admits a $\mathbb{Z}/2\mathbb{Z}$-NZF. In $O(n)$, we compute the in and out degrees of every vertex. If the out degree of any vertex is of a different parity than its in degree, then the graph admits no $\mathbb{Z}/2\mathbb{Z}$-NZF. Else, there is only one NZF, that can easily be found by putting 1 on every edge.

3.2 Finding an initial flow

The algorithm requires an initial valuation of the edges be found. Even though, at first, we opted for a naive idea, where random numbers would be put on every edge, we rather chose to implement an algorithm that would give us a valuation that is as close as possible to being a $\mathbb{Z}/k\mathbb{Z}$-NZF.

We chose to reduce our problem to a set intersection one, with the aim of passing it to the difference map algorithm. We need to describe the solutions of our problem as the intersection between two sets, named A and B, onto which we can easily find a projection. As we have two kinds of constraints to satisfy, both sets will roughly describe one of those types of constraints.

We define the problem as follows. Let $G(V, E)$ a graph, and let us consider the vector $\nu \in \mathbb{R}^{2\cdot|E|}$, that we denote, for convenience, $(e_1^- \; e_1^+ \; e_2^- \; e_2^+ \; \dots \; e_{|E|}^- \; e_{|E|}^+)^T$. Intuitively, e_i^- represents the flow at the tail of edge i, and e_i^+ represents the flow at the end of edge i.

We try to define A as the set of points of $\mathbb{R}^{2\cdot|E|}$ that describe correct nowhere-zero valuations of the edges of G. It is easy to see that, for such a representation to describe a valuation of the edges, we need to have, for every $i \in \{1, \dots, |E|\}$, $e_i^+ = e_i^-$. If it is the case, then $e_i^+ = e_i^-$ is the value that we would associate to edge $e_i \in E$. The values that we allow for e_i^+ are every integer in $\{-k+1, \dots, -1, 1, \dots, k-1\}$. Thus, we formally define A as the following subset of $\mathbb{R}^{2\cdot|E|}$.

$$A = \left\{ (e_1^- \; e_1^+ \; \dots \; e_{|E|}^- \; e_{|E|}^+)^T \;\middle|\; \begin{array}{l} \forall i, e_i^+ = e_i^- \\ e_i^+ \in \{-k+1, \dots, k-1\} \setminus \{0\} \end{array} \right\}$$

We note $P_a : \mathbb{R}^{2\cdot|E|} \longleftarrow A$ the projection of any point to A. Computing computing such a function is fairly easy, and the main difficulty lies in finding a way to make sure no zero appear during the projection.

The valuation of the edges associated to a point in set A doesn't necessarily satisfy Kirchoff's law. This is why set B focuses on the vertices of the graph, and makes sure that this set of constraints is satisfied. Thus, set a set is easily computed:

$$B = \left\{ (e_1^- \; e_1^+ \; \dots \; e_{|E|}^- \; e_{|E|}^+)^T \;\middle|\; \forall a \in \{1, \dots, |V|\}, \sum_{\exists b, e_i = (a,b)} e_i^- = \sum_{\exists b, e_i = (b,a)} e_i^+ \right\}$$

Computing the projection is fairly easy, yet interesting as it is a common constraints optimization problem. Using the Karush-Kuhn-Tücker theorem, we have an explicit expression of the projection.

Using two seemingly unrelated values to describe an edge is a way to make the projection onto B easier. Indeed, such a choice allows us to relax slightly Kirchoff's constraints. This way, we have a set of linear equations, such that every variable appears only once. Thus, we can independently satisfy every equation without interfering with the others, making the computation easier.

It is important to notice that both sets are not convex. It makes it harder to compute a point that belongs to both sets. If both sets were, recursively applying $P_A \circ P_B$ would ensure that we find a solution to our problem, at one moment.

Instead, we need to use a slightly more complex formula, that we apply recursively. Let $\beta \neq 0$:

$$Step : x \longmapsto x + \beta[P_A(f_B(x)) - P_B(f_A(x))]$$

where

$$f_A : x \longmapsto P_A(x) - \frac{1}{\beta}(P_A(x) - x)$$

$$f_B : x \longmapsto P_B(x) - \frac{1}{\beta}(P_B(x) - x)$$

If, for any x, $Step(x) = x$, then it means that $P_A(f_B(x)) - P_B(f_A(x)) = 0$, which means that x belongs to both sets.

The algorithm proceeds as follows: first, we randomly select an initial point in A. Then, we recursively apply *Step* to it, until we find a fixed point for *Step*, or until the allocated computation time is exceeded.

If no correct NZF exists, the algorithm will not be able to find out by itself. Even so, the difference map algorithm is not guaranteed to terminate, even if a correct NZF actually exists, hence the time limit on its execution. Its goal is only to find an initial valuation of the edges that is fairly close to being a correct NZF, so that the heuristics that we use after can refine the valuation and find an NZF.

3.3 Fixing edges

Both following techniques choose a vertex a and try to find a way to change the valuation of one of its vertices to satisfy its Kirchoff's law, as well as the one of one or more of its adjacent vertices.

First, we define the excess of a vertex as the function Δ, which indicates how much there is to substract to the total flow passing by a so that its Kirchoff's law is satisfied. Formaly, if Φ is the current valuation, it is defined as follows:

$$\Delta : a \longmapsto \sum_{e \in \delta^-(a)} \Phi(e) - \sum_{e \in \delta^+(a)} \Phi(e)$$

The first function, that we named simpleFixVertex(a), takes as an input a vertex a whose Kirchoff's law is not satisfied. It then computes its excess $\Delta(a)$. After that, it selects an adjacent vertex b: if $\Delta(b) = -\Delta(a)$, it substracts $\Delta(a)$ from the edge (a, b), or $\Delta(b)$ from the edge (b, a), depending on which one exists. This way, both a and b see that their Kirchoff's law is satisfied. If it is not possible to find such a b, the routine fails.

Note that calling simpleFixVertex may cause a zero to appear on an edge. This is something that is fixed later during the algorithm.

Even though such cases do occur when trying to compute an NZF, fixing them is not enough to find a correct NZF. Thus, a slightly more elaborated way of fixing vertex a Kirchoff's law is to see if there is a set of edges in or out of a, whose valuation can be changed to fix not only a's, but also its neighbors' Kirchoff's laws.

The function that does so is called multipleFixVertex(a). It is similar to the previous function, except that it computes, beforehand, for every subset of vertices adjacent to a, the sum of the excess of its vertices, and then tries to find a subset whose total excess is equal to $-\Delta(a)$. It then proceeds to change the valuations of the associated edges, to satisfy a and its selected neighbors' Kirchoff's laws.

Computing that many values is quadratic in the degree of a, and there is no approach more efficient than the dynamic programming one that we implemented. Still, calling that function is more costly than calling simpleFixVertex, which is why the later is still relevant. This is discussed in the last part of this paper.

3.4 Finding paths

The above heuristics are powerless in greater graphs, in which it often happens that the last few vertices whose Kirchoff's laws are left unsatistied are far apart. This is why we generalized simpleFixVertex, so that it is able to fix distance vertices.

The main idea of PathFixVertex is to find two vertices, a and b, such that $\Delta(a) = -\Delta(b)$. Then, we proceed to find a path between a and b, such that it passes by no edge whose valuation is $\Delta(a)$. This way, we can substract $\Delta(a)$ from every edge on the path, without changing the excess of any vertex on the path, but a and b, whose Kirchoff's laws are now satisfied.

An important remark is that the orientation of the edges can be changed at will. Thus, when looking for a path from i, when can hop to j using the edge (j, i) by considering that it is the edge (i, j), but with valuation $-\Phi((j, i))$. This almost doubles the number of paths that can be considered.

To avoid, as much as possible, deadlocks, and to try to generalize multipleFixVertex to sets of distant vertices, we do the following. When, and only when, there is no pair of vertices whose excess correlate, we fix only one of them, by reporting the excess of a to b, by finding a path between them. If this can not be done, then we randomly choose a number to substract along a path from a to b. The values of the excess of a and b are consequently changed, although it don't fix any of the vertices. Yet, it can be seen as a way to exit a deadlock.

Finding a path in itself can be done efficiently using Dijkstra's algorithm. Switching the orientation of an edge, if needed, is done during the phase where edges are added into the stack. Moreover, during that same phase, before adding an edge to the stack, we check if its current valuation is not $\Delta(a)$. Using a heap, the complexity of this step is $O(|V| \cdot log(|E|))$, which makes it the most costly of the heuristics to fix edges. This is why this step is only done after and fewer times than the others.

We chose not to let this method create new zeros, because the paths that are found can be surprisingly long, and many zeros could be created if allowed. This would greatly complicate the next step, and we chose to make this step longer and slightly more complex to prevent this.

3.5 Removing zeros

Even though the projection onto set A guarantees us that, at the end of the diff map algorithm, no zero remain, sometimes, new zeros appear, when calling simpleFixVertex or multipleFixVertex.

In our case, they are fairly easy to remove, which is done by the fixZeros(a, b) function, which takes, as an input, edge (a, b), whose valuation is zero. Because we have checked that there is no bridge in the graph, (a, b) belongs to at least one cycle. The function finds a cycle where there exists a number $l \in \{1, \ldots, k-1\}$ that never appears on any of the edges of the cycle. Then, l is substracted from every edge of the cycle:

this way, (a, b) is assigned the value $-l \neq 0$ and no other zero has been created.

Algorithmically speaking, finding such a cycle can be done using a breadth first search, in $O(|E| \cdot |V|)$. Even though this seems costly, fixZeros is called very few times, which makes this step reasonable in time.

3.6 Full algorithm

For a start, we define the following higher-order function, which helps us decide to cut a heuristic when we deem it necessary.

Every function is called a bound amount of times. There are also several cases where we will cut a step, if it is clear that it fails to yield interesting results. For instance, if calling simpleFixVertex fails $|V|$ times in a row, we can proceed directly to the next step.

```
def executeHeuristicVertex(heuristic, MAX,
 MAXFAILS):
    iterations = 0;
    failsInARow = 0;
    while iterations ≤ MAX and failsInARow ≤
     MAXFAILS do
        Choose vertex i not satisfied;
        heuristic(i);
        if heuristic(i) succeeded then
            failsInARow = 0;
        else
            failsInARow++;
        end
        iterations += 1;
    end
```

Algorithm 1: A helper function to cut a heuristic when needed

A similar helper function is defined as executeHeuristicEdge, which randomly chooses an edge instead of a vertex. Every heuristic function changes the valuation ν, although it is not mentione everytime, for convenience. Putting these parts together, we give the algorithm described in Algorithm 2.

It is important to notice that every arbitrary choice in the algorithm is non-deterministic. For instances, the choice of the vertex to fix, the edge to use to fix it, or even the path from one vertex to another is randomly choosen. This way, the algorithm can be run multiple times, and find a correct solution where on instances on which it failed previously, be cause of a poor choice that caused a deadlock.

The algorithm is not based on an exhaustive exploration heuristic. Thus, it is possible that it inevitably fails to find a correct NZF for a given instance, even though there is one.

4 EXPERIMENTAL EVALUATION

It seems mandatory to conduct an experimental evaluation of the algorithm, given that it is based on a set of heuristics.

```
Data: A graph G, a group ℤ/kℤ
Result: A NZF ν over G
Check if the graph is bridgeless;
Choose a random valuation ν;
Run the difference map algorithm;
if error(ν) == 0 then
    return ν
end
executeHeuristicVertex(simpleFixVertex,
 MAXSIMP, MAXSIMPFAILS);
executeHeuristicVertex(multipleFixVertex,
 MAXMULTI, MAXMULTIFAILS);
executeHeuristicVertex(pathFixVertex, MAXPATH,
 MAXPATHFAILS);
executeHeuristicEdge(fixZero, MAXZEROS,
 MAXZEROSFAILS)
if noZero(ν) and error(ν) == 0 then
    return ν;
end
return "No valuation has been found";
```

Algorithm 2: The outline of the main algorithm

4.1 Generating benchmarks

An instance of the problem we are condidering simply consists of a directed graph, and of a number k, which will decide in which $\mathbb{Z}/k\mathbb{Z}$ we are going to work. Yet, even though graphs are fairly easy to generate randomly, it is a little harder to make sure that it is interesting for our problem. For instance, we found out that most of the graphs we generated were either extremely dense or had a bridge. As the algorithm prunes out right away graphs with bridges, they are of very little interest.

The benchmark generator thus has an option to reduce the probability with which a generated graph contains a bridge. We found out that adding edges so that the degree of a vertex is at least 3 reduces greatly the odds of the instance being unsolvable.

A more interesting approach would have been to detect components that are 1-edge-connected, and to add edges so that they become, at least, 2-edge-connected. Unfortunately, such an algorithm makes it longer to generate a single problem, and we had the idea too late to implement it and run new tests.

4.2 Experimental protocol

We designed various sets of problems. Each set regroups a variety of problems that have a property in common, and each set comprises both small and bigger graphs.

The first set contains problems in $\mathbb{Z}/k\mathbb{Z}$ for $k \geq 10$. Its aim is to see if working in greater groups makes it easier for the algorithm to find a correct valuation. Every graph was generated so that is has 40 vertices and 120 edges.

The second set of problems makes the algorithm work with $\mathbb{Z}/5\mathbb{Z}$, but with graphs that can be especially sparse or dense.

Table 1: Variation of the success of the algorithm when changing the group order

k	10	15	20	50	150
G_1	0.9	0.9	0.9	0.9	0.9
G_2	0.6	0.9	0.8	0.8	0.8
G_3	0.8	0.7	0.6	0.7	0.7
G_4	0.6	0.8	0.8	0.8	0.8
G_5	1.0	0.8	0.8	0.9	0.9
Av.	0.78	0.82	0.78	0.82	0.82

Table 2: Variation of the success of the algorithm with various graphs densities

$\lvert E\rvert/\lvert V\rvert^2$	0.1	0.4	0.6	0.8	1
G	0.92	0.88	0.82	0.81	0.78

The tests were run on an Intel Core i5-7200U, using at most 6Go of RAM.

4.3 Results

Table 1 presents the percentage of success of the algorithm when run on graphs G_i, depending on which $\mathbb{Z}/k\mathbb{Z}$ the algorithm is working with. As the algorithm is non-deterministic, it is run ten times on each graph, for each value of k.

It is a little surprising to realize that allowing the algorithm to work with higher order groups doesn't make the problem any easier for the algorithm, as its chances of success remain more or less the same.

Table 2 shows the percentage of success of 10 tries on each graph of a family of size 5, whose density follows the given criterias.

Yet again, it is surprising that, even though extremely sparse graphs seem to be easier for the algorithm, graphs that are extremely dense, or even complete, don't seem to be significantly harder than graphs that are not especially dense. An explaination that we would like to propose, is that the increased number of edges makes it easier for the algorithm to find certain paths and cycles.

5 DISCUSSION

5.1 Tests comparisons

No solver for this particular problem exists. This is mainly due to the fact that the formalized problem is easily describable by a set of constraints, and it would be faster to pass an instance to a finite fields linear equations system solver, than to design an algorithm specificaly for the problem, had the actual need for such a solver arisen.

Nevertheless, developping our solver by reducing the problem to a set of equations would have been of little interest, way less recreational, and a little technical. This is mainly why we didn't have the time to develop a program that would convert an instance of our problem to a linear system for which a finite fields solver would have found an answer.

Such a program would have been interesting, but we would have expected it to outperform greatly our algorithm. As there are as many equations to satisfy as vertices in the graph, the biggest instances that our algorithm can solve - with about 400 vertices - are easily solved by modern linear systems solvers, even though our program struggled.

5.2 Finding correct parameters

As our algorithm relies on heuristics, there are a few different parameters that will determine how the algorithm unfolds, as well as its success.

Some parameters will affect how much time the algorithm will allocate to a certain heuristic. For instance, the constant MAXSIMPLEFIX puts a bound in the maximum number of calls to simpleFixVertex that the algorithm can do before proceeding to the next heuristic. Setting the threshold too high will make the algorithm search for too long for a solution that doesn't exist, and setting it too low will prevent the algorithm from finding a solution that could have been found by an heuristic it implements.

The algorithm uses the thresholds MAXSIMPLEFIX, MAXMULTIFIX, MAXPATHFIX and MAXZEROFIX, which respectively determine the maximum numbers of calls that can be made to simpleFixVertex, multipleFixVertex, pathFixVertex and fixZeros. Experimental tests show that using values that are proportional to $|E|$ is a correct balance between efficiency and correctness, in the cases that are considered in the benchmarks. Still, assigning MAXSIMPLEFIX to a significantly lesser value than MAXMULTIFIX or MAXPATHFIX seems to make the algorithm more efficient, without loosing in correctness.

A few other thresholds set the maximum number of times a heuristic can fail in a row. The idea was to prevent the algorithm from trying to apply a heuristic in a state were it is ineffective, due to some unpredicted property of the graph or to previous choices. The main difference with the previous thresholds is that, while the former will cut out the heuristic if the progress is not fast enough, the later will only cut it if no progress is struck.

Another choice that was seemingly arbitrary is the choice of the β factor in the difference-map algorithm. As it is hard to interpret, or even hard to predict what a small change will have for effect, finding a satisfying value can only be done through experimentation. Sometimes, a good value will make the difference map algorithm so efficient that it will be enough to find a solution to the whole problem.

In general, we feel that too many arbitrary choices have been done. The only arguments that we have to motivate our changes are experimental results, which have made us significantly change the values of various parameters. For instance, as a first guess, we assigned MAXSIMPLEFIX to $|E|^2$, while experience shows that a value of $3 \cdot |E|$ seems to be a better fit.

5.3 Relative efficiencies of the heuristics

Studying more carefully the relative importances of each heuristic, as well as the way into which they are combined, would have been interesting.

A few trails have still been blazed during the testing phase. Informally, we realized that removing the MultipleFixVertex heuristic makes it significantly harder - but possible, given enough time - for the algorithm to find a correct NZF.

Another interesting thing that we tried was to remove every heuristic but fixZeros, and start from the null valuation. Surprisingly, such a tweaked algorithm falls into a deadlock tremenduously quick, and most of the problems that were, once, easy to solve - even with only the difference map algorithm and simpleFixVertex - become unsolvable. This seems to be especially true for sparse graphs, although not enough tests were made to provide statistical evidence.

6 FURTHER WORK

6.1 Planar graphs and duality

As said before, using the duality for planar graphs reduces the problem to finding a k-coloration of the dual. Although k-coloration is NP-complete in the general case, we still have efficient heuristics to find a coloration that is, sometimes, suboptimal, although most often optimal or close to the optimal. For instance, a greedy coloring algorithm is polynomial, in $O(|E| \cdot |V|)$. Another less naive algorithm, DSATUR, as described by Brélaz in 1979, is often suboptimal, but has the property of finding, in the worst case, a 4-coloration for planar graphs. It is also reasonably efficient, as it is quadratic in the number of vertices of the graph.

A planar graph is not uniquely associated to a dual graph. But computing a dual graph of a planar graph is not a trivial problem. There are some heuristics that allow us to do so, and some of them are polynomial or have an interesting complexity. Then, using, for instance, a quad edge representation would allow us to quickly and efficiently switch from the primal graph to the dual graph, and thus, quickly converting a k-coloration of the dual to find a k-flow on the primal graph. The conversion of the coloration to a flow can be done in $O(|E|)$.

This means that it would result in a way to solve a specific kind of linear systems more efficiently than by using a linear systems solver.

6.2 Code availability

The code is available at https://gitlab.com/arnaudlequen/spa

Everything was developed in Python, and graphs are displayed using GraphViz, to make it visualy confortable to check if the flow that has been computed is correct.

There are a few tests available on common graphs, such as K_5, $K_{3,3}$, Petersen's graph, etc. The generators have been included in the repository.

7 CONCLUSION

We showed that, using only algorithms on graphs, we could find a NZF. Even though the algorithm still needs to be tinkered with, in order to find efficient parameters and to refine the part each heuristic has to play, it seems to be fairly complete, in the sense that it often finds a valuation when there is one.