An Application of Bernoulli Graphings

MSc thesis

Author: **Levente Szemerédi**

Supervisor: **Endre Csóka**

Eötvös Loránd University
Faculty of Science
Institute of Mathematics
2023

NYILATKOZAT

Név: Szemerédi Levente

ELTE Természettudományi Kar, szak: Matematikus MSc

NEPTUN azonosító: EIORSE

Diplomamunka címe:
An Application of Bernoulli Graphings

A **diplomamunka** szerzőjeként fegyelmi felelősségem tudatában kijelentem, hogy a dolgozatom önálló szellemi alkotásom, abban a hivatkozások és idézések standard szabályait következetesen alkalmaztam, mások által írt részeket a megfelelő idézés nélkül nem használtam fel.

Budapest, 2023. 06. 07.

[illegible]

a hallgató aláírása

Contents

1 Introduction

In the summer of 2022 I participated in a workshop at Erdős Center. The project of my group was to prove that a property holds for large 5-regular graphs with high probability. The problem was already transformed to a seemingly easier one and there were some numerical results showing that it is a good line of attack. We had to modify the algorithm to get a better upper bound. We found an improvement and by running it on some random large graphs we saw that it should be enough. We had to come up with a technique with which we could prove the theorem in general – not just on a few examples. For the original algorithm, the general result was proved by approximating with differential equation systems but we could not reconstruct their calculations. Endre Csóka suggested us to try to use Bernoulli graphings for the calculations instead. I learned how to use Bernoulli graphings for calculating results of local algorithms, and I implemented the solution for this problem. With slight modifications, my solution could follow the calculations of other local algorithms as well. Now, we will state the problem and take a look at some previous results.

Let G be a regular graph. An *internal partition* of G is a partition of $V(G)$ into two nonempty sets such that each vertex has at least as many neighbours in its own class as in the other one. The question is whether there is an internal partition in a given regular graph. For the 3, 4 and 6-regular case the following theorem holds:

Theorem 1.1 (Shafique-Dutton [5], Ban-Linial [6]). *Let $d \in \{3, 4, 6\}$. Then apart from finitely many counterexamples all d-regular graphs have internal partitions. The list of the counterexamples is the following.*

- *for $d = 3$, K_4 and $K_{3,3}$ do not have an internal partition. [5]*
- *for $d = 4$, K_5 does not have an internal partition. [5]*
- *for $d = 6$, each graph on at least 12 vertices has an internal partition, the counterexamples have at most 11 vertices. [6]*

For even-regular, graphs it is verified that they asymptotically almost surely have an internal partition.

Theorem 1.2 (Linial-Louis [7]). *Asymptotically almost every $2d$-regular graph has an internal partition.*

The odd-case is still open for $3 < d$. We will prove the following theorem:

Theorem 1.3. *Asymptotically almost every 5-regular graph has an internal partition.*

Let G be a (finite) graph. A *bisection* of G is a partition of $V(G)$ into two equal-sized parts. The size of the bisection is the number of edges between the two parts. The *bisection width* of G is the minimum of the possible bisection sizes. It is known that computing the bisection width is NP-complete [8].

Bärnkopf, Nagy and Paulovics [9] showed that the bisection with of a 5-regular graph of size n is at most $n/2 + 5$ then it has an internal partition. We will show that the bisection with of a 5-regular graph of size n is asymptotically almost surely below $0.498n$ which concludes the theorem 1.3.

The thesis is organised as follows: in section 2 we discuss the greedy algorithm of Diaz, Serna and Wormald and improve it. Next, in section 3 we will discuss the general notions in the limit theory of bounded degree graphs and discuss how it can be used to approximate calculations on finite graphs. In section 4 we look at how we can do the calculations to solve the problem of the internal partitions. Finally, in section 5 we give the concrete implementation.

Acknowledgements. I am grateful to my thesis advisor, Enrde Csóka and the group of people who I was working with during the workshop: Zoltán Lóránd Nagy, Panna Fekete and Marcin Briański.

2 The algorithm

For estimating the bisection width we will give a refined version of the Diaz-Serna-Wormald algorithm [1]: first we run the original, locally greedy algorithm then do some modification on the result.

The first part is a randomized greedy algorithm. The vertices will be put into two classes: red and blue, in pairs. If there is a partial colouring of the vertices of the graph G we classify its uncoloured vertices according to the number of their red and blue neighbours: a vertex is of type (r, b) if it has r red neighbours and b blue neighbours. Two uncoloured vertices form a *symmetric pair* if one of them is of type (r, b) and the other is of type (b, r).

The colouring of G goes step by step: at each step, we choose a symmetric pair of types (r, b) and (b, r) and colour the one with more red neighbours red, the other blue. If they have the same amount of red and blue neighbours, then one of them is coloured red, the other blue, uniformly at random. The values of r and b are determined according to a priority ordering for types, and vertices are picked uniformly at random from those which have the highest priority.

The priority order of types is defined as follows:

- Type(r, b) has the same priority as Type(b, r).
- If $r_1 \leq b_1$ and $r_2 \leq b_2$ then the priority of Type(r_1, b_1) is less than the priority of Type(r_2, b_2) if and only if $b_1 - r_1 < b_2 - r_2$ or $b_1 - r_1 = b_2 - r_2$ and $r_1 < r_2$.

For example, for a 5-regular graph the priority order (from highest to lowest) is as follows: $(0, 5), (1, 4), (2, 3), (0, 4), (0, 3), (1, 3), (0, 2), (1, 2), (0, 1), (2, 2), (1, 1), (0, 0)$.

If there are no more symmetric pairs left, we colour half of the remaining vertices red and the other half blue uniformly at random.

The original Diaz-Serna-Wormald algorithm ends here. Unfortunately, for $d = 5$ it will give a bound $0.503n$ on the bisection size which is not good enough for us.

In the second part, we swap wrongly placed vertices. Because of the techniques we will use for the actual implementation, we have to be careful at this part.

Let us call a vertex of *terminal type* (r', b') if it has r' red neighbours and b' blue neighbours when the first part of the algorithm terminates. Let us call a red vertex *lost red vertex* if it is of terminal type (r', b') with $r' < b'$. Similarly, a blue vertex is a *lost blue vertex* if it is of terminal type (r', b') with $r' > b'$.

Lemma 2.1. *If we can find an independent set I in the induced bipartite graph* *spanned by the lost red and lost blue vertices then we can improve the bisection size by at least $|I|$ when we flip the colour of the vertices in I.*

Proof. Let us do the recolouring one by one. We have to show that in each step all the non-recoloured vertices remain lost. When we recolour a lost red vertex to be blue then it is not connected to lost blue vertices in I thus all of them remain lost. It can be connected to some lost red vertices in I but it will make those more lost. In each step, the bisection size decreases by at least 1 so we improved at least $|I|$ during the procedure. □

2.1 Doing steps simultaneously

As one can see, the first part of the algorithm described above has the same order of steps as the number of vertices of the graph. First, we address this problem in the following way. Fix an arbitrary N positive integer – and denote $1/N$ by ϵ. We will denote the number of vertices of the graph G by n.

Modify the first part of the algorithm such that in each step – there will be approximately N – we will colour $\epsilon n/2$ vertices of G red and $\epsilon n/2$ vertices blue *simultaneously* according to the rule of the original algorithm. The coloured vertices are chosen uniformly at random from vertices of type (r, b) and (b, r) respectively. We will pick the values r and b almost the same way as before but now we will only consider those types which have at least ϵn vertices in total (in this condition vertices of type (r, b) and (b, r) are united). After we terminated, we colour half of the remaining vertices red and the other half blue uniformly at random.

For the second part, we pick a (big) independent set with the same amount of red and blue vertices in the graph spanned by the lost red and the lost blue vertices and flip the colour of the matched vertices.

What happens with the complexity of the algorithm (and its optimality)? In the first part, now we have N steps for any (big enough) graph. Unfortunately, the run time of each step is of order ϵn – but as we will see, we will be able to give a good approximation in constant time. As we take bigger steps this algorithm probabliy give a slightly worse cut than the first one but it will turn out that this is not a big problem. In the second part, we can guarantee a big enough matching according to the following theorem:

Theorem 2.2 (Axenovich-Sereni-Snyder-Weber [10]). *Suppose that H is a balanced bipartite graph on $n + n$ vertices and has maximal degree 3. If n is big enough then H contains a balanced independent set of size $k + k$ where $k > 0.34116n$.*

As a corollary, at the last step, we improve by 0.34116 times the number of lost vertices. We will not be interested in the concrete set with which the improvement is made.

2.2 Localizing

For the calculations, we do not want our graph to have a huge number of cycles of size at most $2N$. It is known that if we tend to infinity with the size of the graph, the expected number of cycles at most $2N$ converges. This means that with high probability most of the points are far from the cycles of size at most $2N$, meaning that most of the N balls are trees.

The first part of the algorithm is almost local. With randomization, it can be made to be local in the following way:

At each step, every vertex will know the *expected* distribution of the types of vertices. In this context, the expected distribution is the distribution on the infinite tree. Before the first step, every vertex is of Type$(0, 0)$. The type of each step is calculated from this expected distribution (not the actual one) thus it is the same for almost all the vertices. In a step if a vertex is of the appropriate type then it decides randomly whether it will be coloured or not – with the appropriate probability. After that, every vertex updates its type and the expected distribution of the types. Because we took at most N steps, it is enough for each vertex to only communicate in its N-neighbourhood.

In the previous paragraph, I cheated a bit because it is not enough for the vertices to follow the distribution of types (i.e. 1-balls) but rather the types of stars (i.e. 2-balls). At this point, it is not necessarily needed to know all the details, we will discuss them later.

After (the first part) of the algorithm terminates, we have every vertex coloured but also we have (the expected) distribution of the types of the vertices. How much error do we make if we calculate only the latter?

Proposition 2.3. *The distribution of the types of vertices tends to the expected distribution in probability as the size of the graphs tends to infinity.*

For the proof of proposition 2.3 and the computations we need to introduce the notion of Bernoulli graphings.

3 Graphings

In this section, we discuss the basic notions of the limit theory of bounded degree graph sequences and how can we use it to approximate calculations on finite graphs. During this section, we will follow the book [2] by Lovász.

Before we see any specifics let us discuss where large graphs can emerge and why it is useful in general to define a limit theory for them. We are surrounded by large graphs (or networks) everywhere. The most common examples are the Internet and social networks. But whenever there are some objects which are somewhat connected we can think about them as graphs. These graphs may have more structure on them corresponding to additional information about the system we try to model: we can label its vertices or edges.

Now we have some large graphs – occasionally with some additional data – and want to calculate some of their parameters or run some algorithms on them. The problem is that usually, we do not know exactly how our graph looks like because real networks are constantly changing. For this reason, some classical questions of graph theory are not even meaningful. For example, we cannot tell the parity of the number of vertices of the graph of 'connections between people' because at any time there are people who are currently being born or dying. Even if somehow we manage to solve this problem and 'know' the graph we will still have a hard time during (theoretical) calculations. We will see that we can do calculations on some kind of limit object with a small error.

So when we work with the limit objects we do not have to know the exact structure of the graph we would like to work with. But to know what our limit object will be we have to know some local properties. Let us discuss some of the potential techniques which could work for this purpose. For this, there are basically two (three) types of graph-sequences: dense, bounded degree (and in between).

Definition 3.1. Let $G_1, G_2, \dots$ be a graph sequence with $|V(G_n)| \to \infty$.

- This graph sequence is *dense* if $|E(G_n)| = \Omega(|V(G_n)|)$.
- If there is a D number such that for every G_n graph in the sequence has maximal degree at most D, then the graph sequence is a *bounded degree graph sequence.*

In this thesis, we will focus on the bounded degree case. From now on, we will assume that every graph is a bounded degree graph with maximal degree at most D.

I will follow the notations of the book [2] by Lovász. In particular,

- Finite graphs are denoted by the letters F and G and their decorated versions (eg. F', G_1).
- Families of finite graphs are denoted by calligraphic letters.
- $\mathcal{G}$ denotes the family of all finite graphs.
- Countable graphs are denoted by letters H and its decorated versions (eg. H', H_1).
- Families of countable graphs are denoted by gothic letters.
- $\mathfrak{G}$ denotes the family of all countable graphs (with bounded degree).
- Boldface letters will denote graphs with larger cardinality (eg. $\mathbf{G}$)
- For a non-negative r the set $\mathfrak{B}_r$ is the set of rooted r-balls, i.e. graphs where each vertex is at a distance of at most r from the root.

Our limit objects will be measures on some infinite graphs. Before we can define these measures we must introduce the notion of Borel graphs.

Definition 3.2. Let $(\Omega, \mathcal{B})$ be a Borel sigma-algebra. Let $\mathbf{G}$ a graph with $V(\mathbf{G}) = \Omega$. If the edge set $E(\mathbf{G})$ is a Borel subset then $\mathbf{G}$ is a Borel graph. Because we will use bounded degree graphs only, we will also assume that a Borel graph is also a bounded degree graph.

The following lemma gives a characterisation or alternative definition for Borel graphs which will be useful in the future.

Lemma 3.3. *[2][Lemma 18.2] Let $\mathbf{G}$ be a graph on a Borel space $(\Omega, \mathcal{B})$. It is a Borel graph if and only if for every Borel set $B \in \mathcal{B}$, the neighbourhood $N(B)$ is Borel.*

On Borel graphs, we can define Borel functions. The most important for us is the degree function: $\deg_A^{\mathbf{G}}(x)$ which is the degree of x in a Borel set A.

Finally, we can define the objects which will be the limit objects of bounded degree graph sequences.

Definition 3.4. Let $(\Omega, \mathcal{B})$ be a Borel sigma-algebra with a probability measure λ on it. A Borel graph $\mathbf{G}$ is called a *graphing* on the node set Ω if for any two measurable sets A and B

$$\int_A \deg_B(x)\, \mathrm{d}\,\lambda(x) = \int_B \deg_A(x)\, \mathrm{d}\,\lambda(x)$$

holds. This means that if we take the measure of edges from A to B then we get the same number as if we would 'count' from B to A.

From the measure λ some other measures can be computed naturally. The first one is the *volume* measure which is just the integral of the degree function:

$$\operatorname{Vol}(A) = \int_A \deg \mathrm{d}\,\lambda.$$

If we take the volume of the whole underlying set we get the average degree $d_0 = \operatorname{Vol}(\Omega)$. If we normalize the volume measure with the average degree we get another probability measure λ^* which is the *stationary distribution* on $\mathbf{G}$ – which refers to the random walk on G. So the definition of λ^* is

$$\lambda^*(A) = \operatorname{Vol}(A)/\operatorname{Vol}(\Omega).$$

We can connect λ and λ^* in the following way: generate a random point x from the distribution λ and keep it with probability $\deg(x)/D$, otherwise reject it and generate a new one. The point which is accepted will be a random point from the distribution λ^*.

We can also define a measure on the edges of the graphing in the following way: let η be a measure on $(\Omega \times \Omega, \mathcal{B} \times \mathcal{B})$ satisfying the following:

$$\eta(A \times B) = \int_A \deg_B \mathrm{d}\,\lambda$$

for product sets. By Caratheodory's Theorem η can be extended to the sigma algebra generated by the product sets. If we normalize it with the average degree we get a probability measure η/d_0 on the edges of $\mathbf{G}$. By the definition of graphings, this measure is symmetric.

3.1 Bernoulli graphings

Previously, we defined graphings in general. Now we will introduce a special kind of them: the Bernoulli graphings and discuss some of its properties. As before, all the graphs discussed here are bounded degree graphs with maximal degree at most D.

First, let us define the graph of graphs as follows. Let $\mathfrak{G}^*$ be the set of all connected, countable rooted graphs with maximal degree at most D. A rooted graph $\mathbf{G}_x$ is a graph with a special node $x \in V(\mathbf{G})$ which we call its root. Similarly, $\mathcal{G}^*$ is the set of all connected, finite rooted graphs with maximal degree at most D. Two rooted graphs are the same if there is a root preserving isomorphism between them.

Definition 3.5. Let $\mathbf{H}$ be the following graph: its node set $V(\mathbf{H})$ is the previously defined $\mathfrak{G}^*$ and the edges adjacent to a rooted graph $H_x \in \mathbf{H}$ are $H_xH_{x'}$ where xx' are an edge in H. This $\mathbf{H}$ is the *graph of graphs.*

We want to have a Borel structure on $\mathbf{H}$. For this, let us consider the following notion of distance which will turn out to be a metric.

Definition 3.6. Let $H_1, H_2 \in \mathbf{G}^*$. Let their *ball distance* be

$$d^*(H_1, H_2) = \inf\{2^{-r} : B_{H_1,r} \cong B_{H_2,r}\}$$

where $B_{H,r}$ is the ball of radius r in H centered at its root.

Proposition 3.7. *For the ball distance defined above the following are true:*

1. *The ball distance is a metric on $\mathfrak{G}^*$.*
2. *Let F be a finite r-ball. $\mathfrak{G}^*_F$ denotes those graphs H for which $B_{H,r} \cong F$ hold. Then the sets $\mathfrak{G}^*_F$ are clopen and they form an open basis.*
3. *The space $(\mathfrak{G}^*, d^*)$ is compact and totally disconnected.*

Proof. The only non-trivial part is compactness. It follows from the fact that $\mathfrak{G}^*$ is complete and totally bounded. $\square$

We will denote the sigma-algebra of Borel sets of $(\mathfrak{G}^*, d^*)$ by $\mathfrak{A}$.

Proposition 3.8. *The graph of graphs $\mathbf{H}$ is a Borel graph.*

Proof. It is enough to prove that the edge set of $\mathbf{H}$ is closed. We will show that in $\mathfrak{G}^* \times \mathfrak{G}^*$ a small neighbourhood of a non-edge of $\mathbf{H}$ does not contain any edge of $\mathbf{H}$.

Let us take two rooted graphs $H, H' \in \mathbf{H}$ which do not form an edge in $\mathbf{H}$. Let us denote the neighbours of H in $\mathbf{H}$ by $H_1, \ldots, H_d$ (because we only consider bounded degree graphs, clearly $d \leq D$ holds). Since d^* is a metric there is a radius r such that $B_{H',r} \not\cong B_{H_i,r}$ for $i = 1, \ldots, d$.

Now we will prove that if $\bar{H}, \bar{H}' \in \mathbf{H}$ two rooted graphs for which $d^*(H, \bar{H}) < 2^{-r}$ and $d^*(H', \bar{H}') < 2^{-r}$ holds then $\bar{H}$ and $\bar{H}'$ are not connected by an edge in $\mathbf{H}$.

Let us suppose that there is an edge between $\bar{H}$ and $\bar{H}'$. Because of $d^*(H, \bar{H}) < 2^{-r}$, their $r+1$-balls are isomorphic: $B_{H,r+1} \cong B_{\bar{H},r+1}$. Because of the edge between $\bar{H}$ and $\bar{H}'$ and the definition of edges in $\mathbf{H}$, if we shift the roots, we have $B_{H_i,r} \cong B_{\bar{H}',r}$ for some $1 \leq i \leq d$. By $d^*(H', \bar{H}') < 2^{-r}$ we also have that $B_{H',r} \cong B_{\bar{H}',r}$. Combining these together we get $B_{H_i,r} \cong B_{H',r}$ which is a contradiction. $\square$

We want to be able to choose from $\mathfrak{G}^*$ randomly, i.e. to have a probability measure. Now, let us take and arbitrary probability measure σ on $(\mathfrak{G}^*, \mathfrak{A})$. The degree function deg of the root is a measurable function on $\mathfrak{G}^*$, so we can introduce the following measure on $\mathfrak{G}^*$:

$$\sigma^*(A) = \frac{\int_A \deg \mathrm{d}\,\sigma}{\int_{\mathfrak{G}^*} \deg \mathrm{d}\,\sigma}.$$

We want the measure to respect the graph in some sense. For this let us introduce the following notion.

Definition 3.9. Take a random graph H from the distribution σ^* and select a uniformly random edge e from the root of H. e is considered oriented away from the root of H. This gives a probability distribution $\sigma^{\rightarrow}$ on $\mathfrak{G}^{\rightarrow}$ which is the set of rooted graphs from $\mathfrak{G}^*$ with an oriented edge, a root edge, specified. Let us call σ *involution invariant* if the map $\mathfrak{G}^{\rightarrow} \to \mathfrak{G}^{\rightarrow}$ which reverses the orientation of the root edge if measure preserving with respect to $\sigma^{\rightarrow}$. An *involution invariant random graph* is a random rooted graph from an involution invariant probability measure on $\mathfrak{G}^*$.

To see why involution invariant measures are important, consider the following construction.

Construction 3.10. Let $\mathbf{G}$ be an arbitrary graphing and $x \in V(\mathbf{G})$ is one of its vertices chosen randomly according to the measure λ. The connected component of x in $\mathbf{G}$ is a $\mathbf{G}_x$ graph which is in $\mathfrak{G}^*$ which will be called a *random rooted component of* $\mathbf{G}$. So the *component map* $x \mapsto \mathbf{G}_x$ is a measurable map which defines a probability distribution σ on $(\mathfrak{G}^*, \mathfrak{A})$. If x was selected from the distribution λ^*, then $\mathbf{G}_x$ is selected from the distribution σ^*. If we choose one of the edges coming from x in G_x, we get an edge from the distribution η/d_0 with an orientation. Since the measure η is symmetric, if we change the orientation of the edge the distribution does not change. Thus σ is an involution invariant measure.

With the construction above, we can make an involution invariant measure from any graphing. From the other direction, this graphing *represents* the involution invariant measure σ on $\mathfrak{G}^*$. The inverse of this statement also holds:

Theorem 3.11. *Every involution invariant probability distribution σ can be represented by a graphing.*

We will prove this statement by giving such a representant.

If the graphs from σ have no symmetries as unrooted graphs then it is not so hard to give a representant by the following

Lemma 3.12. *If σ is an involution invariant measure on $\mathfrak{G}^*$ such that graphs from the distribution σ has no automorphisms with probability 1, then $(\mathbf{H}, \sigma)$ is a graphing and it is a representant of σ.*

If graphs from σ do have symmetries the previous construction does not work. However, with some modifications, we can break the symmetries:

Definition 3.13. Let us denote the set of triplets (H, v, α) by $\mathfrak{G}^+$ where $H_v \in \mathfrak{G}^*$ is a rooted graph and $\alpha : V(H) \to [0, 1]$ is a weighting on its vertices. Two such triplets are considered equivalent if there is an isomorphism between the graphs which preserves both the root and the weighting. The following sets will be the generator of the sigma-algebra $\mathfrak{A}^+$: fix the isomorphism type of the r-ball around the root and for every vertex in the r-ball and fix a Borel set of $[0, 1]$ from which the weights can be chosen on the r-ball (these could be different Borel sets for different vertices). Define the *graph of weighted graphs* $\mathbf{H}^+$ as follows. The vertex set of $\mathbf{H}^+$ is $\mathfrak{G}^+$ and two vertices are connected if they are isomorphic as non-rooted weighted graphs and their roots are neighbours. For a given probability distribution σ on $\mathfrak{G}^*$ we can create a probability distribution σ^+ on $\mathfrak{G}^+$ by choosing a random rooted graph according to σ and assigning independent uniform random weights from $[0, 1]$ to its vertices. With the construction above for any σ involution invariant measure we can associate a graphing. This is called the *Bernoulli graphing representing* σ

Proposition 3.14. *The construction above is indeed a graphing representing σ.*

Proof. First, we prove that it is a graphing. Similarly to the definition of σ^*, let us define $\sigma^{+*}(A) = \frac{\int_A \deg \mathrm{d}\,\sigma^+}{\int_{\mathfrak{G}^+} \deg \mathrm{d}\,\sigma^+}$. Take a rooted, weighted graph (H, v, α) from the distribution σ^{+*} and uniformly at random a neighbour u of v in H. With probability 1, the values of α are all different, so the rooted, weighted graph (H, u, α) is almost surely different from (H, v, α). These two also form an edge in $\mathbf{H}^+$ by definition. This gave us a way to get a random (oriented) edge from $\mathbf{H}^+$. Because σ was involution invariant, this distribution is invariant under the flipping of the orientation of the edges. Thus $(\mathfrak{G}^+, \sigma^+)$ is a graphing.

For the representation we have to argue that if we take the graphing $(\mathfrak{G}^+, \sigma^+)$ and apply the steps of 3.10 then we get back the distribution σ. As before, those rooted, weighted graphs (H, v, α) where any two weights are equal form a set of measure 0. So almost surely the connected component of (H, v, α) is isomorphic to H_v as a rooted graph. This gives us a distribution on $\mathfrak{G}^*$. But we constructed σ^+ in such a way that first we select a graph and then a weighting so the construction must give back σ. □

3.2 Local convergence

Let us discuss the relation between bounded degree graph sequences and graphings. In particular, we will answer the question of when we can say that a graph sequence converges to a given graphing in some sense.

The convergence is a topological notion, so we have to give a topology on the space of graphs and graphings. We will do this by introducing a metric on it. But before that, we need some new definitions.

Definition 3.15. Let $G \in \mathcal{G}$ be a graph. Let us denote the distribution of its r-balls $\rho_{G,r}$. We will call this *sample distribution of size r*.

For two different graphs (not necessarily of the same size) the sample distribution of size r is supported on the same set, the set of r-balls, $\mathfrak{B}_r$. Thus we can take their variational distance as distributions. Using this we have the following definition.

Definition 3.16. Let $G, G' \in \mathcal{G}$ be two graphs. Then their *sampling distance of depth r* is the variational distance of their sample distribution of size r denoted by $\delta^r(G, G')$. We can combine these distances together to get the *sampling distance* of the two graphs by the following formula:

$$\delta(G, G') = \sum_{r=0}^{\infty} \frac{1}{2^r} \delta^r(G, G').$$

Let us use the previous definition for defining the convergence of bounded degree graph sequences as follows.

Definition 3.17. A graph sequence G_n with $|V(G_n)| \to \infty$ is *locally convergent* if for every r and every r-ball F the r-neighbour densities $\rho_{G_n,r}(F)$ converges.

Note that this is equivalent to (G_n) being a Cauchy-sequence in the sampling distance.

The definition does not give us directly a limit object. We will construct one but it will be not unique. First, let us discuss the properties which a limit object should have.

Definition 3.18. For every r, let us denote the limit of the distribution $\rho_{G_n,r}$ σ_r. σ_r is on the set of $\mathfrak{B}_r$ and it is *consistent*, i.e. if we take a random r-ball from σ_r and delete those vertices which are at distance r from the root, we get a random $r-1$-ball from σ_{r-1}.

But there are $r-1$-balls in a r-ball centered at the neighbours of the original root. These also should give back the distribution σ_{r-1}. Because different r-balls could have different root-degrees when we write the induced distribution on the $r-1$ balls we have to do a weighting according to these. Let us define the following modified distribution with this in mind:

$$\sigma_r^*(F) = \frac{\deg(F)\sigma_r(F)}{\sum_{G \in \mathfrak{B}_r} \deg(G)\sigma_r(G)}.$$

Now let us take a random ball F from the distribution σ^* and a random edge from the root of F. We create two $r-1$-balls: one with the original root and one with the other end of the edge. We also remember, which edge was the 'connecting' edge. This construction gives us two distributions on $\mathfrak{B}_{r-1}$ with a root edge. If these distributions are the same for all $r > 0$, then the sequence $(\sigma_1, \sigma_2, \dots)$ is *involution invariant.*

The other direction also works: if there is a consistent sequence $(\sigma_1, \sigma_2, \dots)$ then this gives us a distribution σ on $(\mathfrak{G}^*, \mathfrak{A})$ by $\sigma(\mathfrak{G}^*_F) = \sigma_r(F)$ for every r-ball F. By the definition of involution invariance it follows that $(\sigma_1, \sigma_2, \dots)$ is involution invariant if and only if σ is involution invariant. This is the *local limit* of the sequence (also called as *Benjamini-Schramm limit*).

With these definitions at hand, we can state the following lemma.

Lemma 3.19. *Let G_n be a sequence of random d-regular graphs where each G_n has n vertices. We generate the random graphs with the configuration model: at each vertex, there are d half edges and we take a random pairing of these. Then the sequence G_n is almost surely locally convergent and its limit is the infinite d-regular tree.*

Proof. Let us fix an r radius. We will show that the distribution $\rho_{G_n,r}$ almost surely tends to the probability distribution concentrated on the regular tree of depth r. The expected number of at most $2r$-cycles in G_n is convergent, bounded by a function of d and r. So most vertices will be farther than r from any at most $2r$ cycles thus almost all r-neighbourhood is a tree.

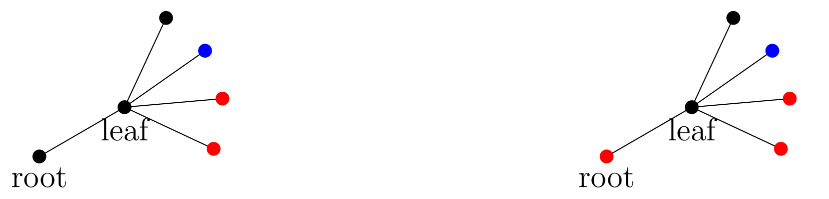
This holds for each r so the sequence is locally convergent. Its limit is concentrated on the infinite d-regular tree. And it is represented by a Bernoulli graphing concentrated on the d-regular tree. $\square$

4 Details of the calculation

Now, we can prove 2.3. Take a sequence of random 5-regular graphs with increasing size. By lemma 3.19 it is convergent almost surely and its local limit is represented by the Bernoulli graphing concentrated on the 5-regular infinite tree. With the number of steps N fixed, we have that the N-neighbourhood of almost all vertex is a d-tree, in these neighbourhoods, the root has the same type distribution as the root of the infinite tree. With high probability the number of neighbourhoods containing small cycles is bounded by a constant so as we tend to infinity with the size of the graph, their ratio tends to 0, thus their contribution to the average as well.

So we just have to calculate the expected distribution of the final types. We will do the calculation on a Bernoulli graphing representing the involution invariant measure which is concentrated on the infinite d-regular tree. With this technique, we can calculate directly the limit distribution and avoid dealing with the (finitely) many parameters from the weak laws which control the error terms. However, we will have some error due to ignoring events with low probability in exchange for significantly faster run time but this will be controllable.

Now it is time to discuss which 'vertex types' we will follow. These are not the same as the types defined in the beginning so from now on, we will call them *star types*. A star type decodes a coloured two-neighbourhood up to isomorphism. We can represent this as a coloured one-neighbourhood where the root can be red, blue or uncoloured and the leaves can be red, blue or typed uncoloured. For easier implementation, the type of an uncoloured leaf encodes how many red and blue neighbours it has outside the star, i.e. if we colour the root the type of the uncoloured leaves does not change. From the star types, we can easily get back the (uncoloured) vertex types just by taking the appropriate marginals.



(a) The root is uncoloured (b) The root is red

Figure 1: Both leaves have the same leaf type $(2, 1)$.

To be able to continue our calculations, we need the following lemma.

Lemma 4.1. *Suppose that at one point in the algorithm, there is an edge with both endpoints uncoloured. If we take out this edge, the infinite tree falls to two parts.*

Then the colouring (red, blue or uncoloured on each vertex) of the two parts are identical and independent in distribution.

Proof. It is enough to prove that during the algorithm, no information goes through that edge other than both of its endpoints are uncoloured. But this is true: in each step, every vertex asks the colours of its neighbours and if it got coloured tells this information to its neighbours. Since both endpoints of the edge are uncoloured, the first type of information is always 'uncoloured' and the second type never occurs. □

From this lemma, we can conclude that when we want to update the distribution of the stars, the distribution of an uncoloured 'outer' neighbour of an uncoloured leaf is identical to the distribution of (uncoloured) vertex types conditional to having at least one uncoloured neighbours and these are independent of each other.

Now we just have to calculate in each step how these star types develop, i.e. for each star type S and S' we have to calculate what ratio of stars of type S will become of type S'. And we have to do this in a smart way because the number of star types increasing very fast in d.

Lemma 4.2. *Let S and S' be two star types with fixed leaf-order. At a step of the algorithm where the type of the colourable vertices is* $\mathrm{Type}(r, b)$ *a star of type S can become a star of type S' if and only if all the following conditions hold:*

- *if the root of S is coloured then the root of S' is also coloured with the same colour*
- *if the root of S is uncoloured then either*
  - *the root of S' is uncoloured*
  - *the root of S is of* $\mathrm{Type}(r, b)$ *and the root of S' is of the appropriate colour (this can be red, blue and red or blue)*
- *for each pair of leaves in S and S' with the same index*
  - *if the leaf in S is coloured then the leaf in S' is also coloured with the same colour*
  - *if the leaf in S is uncoloured with outer type (r', b') then either*
    - *the leaf in S' is uncoloured and it has at least r' red outer neighbours and at least b' blue outer neighbours*
    - *the leaf in S is of* $\mathrm{Type}(r, b)$ *and the leaf in S' is of the appropriate colour*

Let us calculate the probability of a star of type S developing to be a star of type S' at a step of Type(r,b). Let us denote the probability of an uncoloured vertex of Type(r,b) gets coloured red p, and the probability of an uncoloured vertex with at least one uncoloured neighbours becomes red q. This probability is the product of a root part and a leaf part. The root part is 1, if the root of S is coloured or it is uncoloured but not of Type(r,b) and it is p or $1-2p$ if the root of S is uncoloured and the root of S' is coloured or uncoloured respectively.

Each leaf has a contribution to the leaf part and these are multiplied together. For one leaf the contribution is calculated as follows.

- if the leaf in S was coloured then it is 1.
- if the leaf in S was uncoloured but in S' it is coloured then it is p.
- if the leaf in S was uncoloured and in S' it is still uncoloured then
  - let us suppose that S has r_1, b_1, u_1 red, blue and uncoloured outer neighbours respectively and S' has r_2, b_2 and b_2 red, blue and uncoloured outer neighbours respectively. Let us call the differences $r_2 - r_1$ and $b_2 - b_1$ d_r and d_b respectively. From the lemma, we know that they are nonnegative numbers.
  - the contribution of this leaf is $\binom{u_1}{u_1-d_r-d_b} \cdot \binom{d_r+d_b}{d_b} \cdot q^{d_r+d_b} \cdot (1-2q)^{u_1-d_r-d_b}$.
  - if the leaf in S was of Type(r,b) then the previous expression must by multiplied with $1-2p$.

Now the algorithm goes as follows: first, everything is uncoloured so the appropriate star type is of measure 1, and everything else is of measure 0. In each of the N (or so) steps we take every pair of ordered star types S and S' then see if a star of type S could become S'. If so, we calculate its probability and we take that portion of S to be S' after the step. Unfortunately, it is terribly slow: in the 5-regular case, there are more than 4 million ordered star types. Let us consider what could be done to optimize the algorithm.

Preprocessing. If we look at the probabilities calculated above, we can notice that if we fix the type of the colourable vertex then these probabilities are of form $c_0 \cdot p_1^c \cdot (1-p)_2^c \cdot q_3^c \cdot (1-q)_4^c$. So as a 0th step calculate these five numbers from all pairs of star types and create a list for all star types containing the pairs: (what it can become, the coefficients in the formula). In each step, we just have to go through the list of all star types and plug in the appropriate values of p and q to the formula. We will call these polynomials *transition polynomials*.

Just calculate with unordered star types. We can use unordered star types, but it makes the calculations a bit trickier because of the multiplicities. If $d = 5$

then there are 'only' about 60 thousand unordered star types instead of the 4 million ordered one.

Ignore events with low probability. We will ignore those transition polynomials where the exponent of q is greater than a threshold. We will calculate, that if we choose 2 as this threshold then the total error we make is of order ϵ. These ignored probabilities will be added to the 'nothing happens' case.

Ignore certainly wrong star types. Because of the previous simplifications, we know that a very little portion of star type pairs will have a nonzero transition polynomial. So when taking these pairs let us do the following: fix S and take only those S'-s which are different in at most 2 (the threshold above) colouring to S.

With the ideas above the algorithm can run in a reasonable time. But before we discuss the results, we need to calculate the error due to the approximation at the step *Ignore events with low probability.*

4.1 Error calculation

Now we will estimate the probability that for a given star there are at least three 2-neighbours which would got coloured if we not ignore this event. Let us denote the number of uncoloured 2-neighbour (which are a 2-neighbours through an uncoloured 1-neighbour) u.

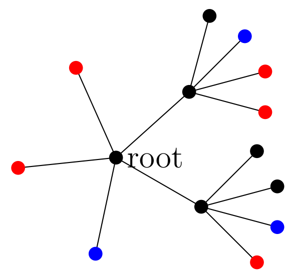


Figure 2: In this star the root has 3 uncoloured 2-neighbours.

$$\begin{aligned}\mathbb{P}\left(>2\text{ coloured}\right)=&1-\mathbb{P}\left(<3\text{ coloured}\right)\\=&1-(1-2q)^{u-2}\cdot\left((1-2q)^2+u\cdot 2q\cdot(1-2q)+\binom{u}{2}\cdot(2q)^2\right)\\=&1-\left(1-(u-2)\cdot 2q+\binom{u-2}{2}\cdot(2q)^2-\binom{u-2}{3}\cdot(2q)^3+o(q^3)\right)\\&\cdot\left((1-2q)^2+u\cdot 2q\cdot(1-2q)+\binom{u}{2}\cdot(2q)^2\right)\\=&q^3\cdot\left(\frac{4}{3}u^3-4u^2+\frac{16}{3}u\right)+o(q^3)<4/3(u^3+1)\cdot q^3+o(q^3).\end{aligned}$$

If T is an arbitrary vertex type then let us denote $\mu(T)$ the ratio of vertices which are of type T and denote the number of uncoloured neighbours $U(T)$. Note that $\mu(T)$ is not the true value, it has already accumuluted errors in the previous steps. It can be shown that if the step type is $T(r,b)$, then the value of q is as follows:

$$q=\frac{\epsilon/2\cdot(d-r-b)}{\sum_{T\text{ vertex type}}\mu(T)\cdot U(T)}.$$

Now it is enough to show that $\prod_1^{1/\epsilon}(1+4/3(u^3+1)\cdot q^3)$ is not much bigger than 1. If we could guarantee $4/3(u^3+1)\cdot q^3<\epsilon$ for each step then we could conclude that the product is less than $\exp\epsilon\approx 1+\epsilon$.

By reordering the terms, we have the following formula: $\left(1/6(u^3+1)\right)^{1/3}\cdot(d-r-b)\cdot\epsilon^{2/3}<\sum_{T\text{ vertex type}}\mu(T)\cdot U(T)$. In the first step, inequality holds. After that there will be no steps of Type$(0,0)$, so the left hand side is at most $\left(1/6((d(d-1))^3+1)\right)^{1/3}\cdot 4\cdot\epsilon^{2/3}\approx 44\cdot\epsilon^{2/3}$ for $d=5$. Those steps where we colour vertices which have a fully determined neighbourhood, i.e. only the root is uncoloured, the leaves are coloured, does not contribute to the error calculation – in these cases, $q=0$ automatically. So if there are at least k real steps ahead then the right hand side is at least $k\epsilon$. So we need $44\cdot\epsilon^{2/3}<k\epsilon$. After rearranging, we have $44\cdot N^{1/3}<k$. Thus our stated inequality hold before the last $44\cdot N^{1/3}$ real steps. In this calculation we used the calculated values of the ratio of the different star types. Note that finnally we managed to ask for a lower bound on the remaining true steps. But – at least not until the very end – we cannot have that much error that the number of remaining true steps would differ significantly in the calculated and in the real case.

In each step the ratio of each vertex type (maybe with coloured root) can change with at most 6ϵ. So the last, possibly wrong real steps could modify the final vertex distribution by $264\cdot\epsilon^{2/3}$ for each vertex type. So this type of error also tends to 0. I believe this second type of error could be estimated in a better way such that it would be much less than the first type. Because of that in the table below (and in the code) I did not calculat with that.

4.2 Results

I implemented the algorithm discussed above in C++ (the source files are in section 5, it was compiled with g++ 9.4.0 with c++14). The table below contains the output values for $d = 5$.

ϵ	cut size without improvement	lost vertices	improved cut size
10^{-3}	0.51009	0.018628	0.50425
10^{-4}	0.50347	0.016906	0.49775
10^{-5}	0.50282	0.016500	0.49720

So we got that for $\epsilon = 10^{-5}$ the improved cut size is at most 0.49720 which concludes the proof of 1.3.

Which in turn concludes the proof of theorem 1.3.

5 Implementation

In this section I will provide my implementation of the methods discussed in the previous sections. I compiled with g++ 9.4.0 with c++14.

The interesting thing about this implementation is that with slight modifications it could do calculations for other local algorithms as well – where in each step it is enough to know the 2-neighbourhood of the root.

Here are some example of possible modifications:

- the parameters of the algorithm: d, ϵ and the degree of the approximation can be modified in config.h
- the priority order of the vertex types can be modified by changing the function Priority::operator¡
- the other similar local algorithm can be implemented by modifying the functions Transition::isTransitionable and Transition::getTransitionData
- new colours or other decorations can be introduced in the class Point and SimpleStar

main.cpp

```
#include "main.h"

#include "math.h"
#include <iostream>
#include <algorithm>
#include <numeric>

int main() {
    auto stars = generateStars();

    auto priorities = generatePriorities();
    cout << "Order␣of␣priorities:\n";
    for(auto it=priorities.begin(); it!=priorities.end(); ++it)
        it−>printType();
    cout << "\n−−−−−−−−−−\n";
    vector<Transition> transitions;
    while(update(stars, priorities, transitions));
    report(stars, priorities);

    return 0;
}

bool incLeafOutType(vector<int>& lot, int maxVal) {
    int idx = lot.size()−1;
    while(idx>0 && lot[idx]==lot[idx−1])
        idx−−;
    if(idx == 0) {
        if(lot[0] == maxVal−1) {
            fill(lot.begin(), lot.end(), 0);
            return false;
        }
        lot[lot.size()−1]++;
        fill(lot.begin(), lot.end()−1, 0);
        return true;
    }
    lot[idx−1]++;
    fill(lot.begin()+idx,lot.end()−1,lot[idx−1]);
    return true;
}

vector<Star> generateStars() {
    vector<Star> stars;
    int maxLeaf = DEGREE∗(DEGREE+1)/2+2;
    for(int c=0; c<3; c++) {
        vector<int> lot(DEGREE, 0);
        do {
            stars.push_back(Star(c,lot));
        } while(incLeafOutType(lot, maxLeaf));
    }
    stars[SimpleStar::getNewStarIndex(SimpleStar(0, vector<int>(DEGREE, 2)))].
        setMeas(1);
    auto s = stars[SimpleStar::getNewStarIndex(SimpleStar(0, vector<int>(DEGREE,
         2)))];
    cout << s.getCenter().getColour() << "␣" << s.getLeafs()[0].getColour() << "\n";
    return stars;
}

vector<Priority> generatePriorities() {
    vector<Priority> p;
    for(int r=0; r<=DEGREE; r++)
        for(int b=r; b+r<=DEGREE; b++)
            p.push_back(Priority(PointType(r,b)));
    sort(p.rbegin(), p.rend());
    return p;
}
```

```
bool update(vector<Star>& stars, vector<Priority>& priorities, vector<Transition>&
    transitions) {
    udouble sumOfMargs = calculateMargs(stars, priorities);
    auto colouredIt = find_if(priorities.begin(), priorities.end(),
        [](auto p) {return p.isColourable();});
    if(colouredIt == priorities.end())
        return false;
    cout << sumOfMargs/EPS << "␣";
    colouredIt->printType();
    cout << endl;
    auto transitionIt = find_if(transitions.begin(), transitions.end(),
        [colouredIt](auto t) {return *colouredIt == t.getColoured();});
    if(transitionIt == transitions.end()) {
        transitions.push_back(Transition(stars, *colouredIt));
        transitionIt = --transitions.end();
    }
    udouble probKnown = EPS*(2-colouredIt->isSymmetric())/(2* (colouredIt->
        getMeas()));
    udouble probUnknown = probKnown/(2-colouredIt->isSymmetric())*colouredIt
        ->numOfUncol()*colouredIt->getMeas()/
        accumulate(priorities.begin(), priorities.end(), udouble(0),
            [](udouble s, auto p){return s+p.numOfUncol()*p.getMeas();});
    transitionIt->doTheTransition(stars, probKnown, probUnknown);
    return true;
}

udouble calcCut(const vector<Star>& stars) {
    vector<udouble> c;
    for(auto s: stars) {
        if(s.getCenter().getColour() == 1)
            c.push_back(s.getMeas()*s.getMultiplicity()*s.getCenter().numOfNeig()[2]);
        else if(s.getCenter().getColour() == 2)
            c.push_back(s.getMeas()*s.getMultiplicity()*s.getCenter().numOfNeig()[1]);
    }
    sort(c.begin(), c.end());
    return accumulate(c.begin(), c.end(), udouble(0))/2;
}

udouble calculateMargs(const vector<Star>& s, vector<Priority>& p) {
    vector<udouble> m;
    for(auto it = p.begin(); it!=p.end(); ++it)
        m.push_back(it->calculateMeas(s));
    sort(m.begin(), m.end());
    udouble res = accumulate(m.begin(), m.end(), udouble(0));
    return res;
}
```

```
void report(const vector<Star>& s, vector<Priority>& p) {
    cout << "\n−−−−−␣REPORT␣−−−−−\n";
    cout << "DEGREE:␣" << DEGREE << endl;
    cout << "EPS:␣" << EPS << endl;
    udouble e0 = exp(−EPS);
    udouble e1 = exp(EPS);
    cout << "Error␣multiplier␣due␣to␣not␣considering␣higher␣order␣terms:␣" << e0
        << "␣...␣" << e1 << "\n";
    udouble cut = calcCut(s);
    cout << "Size␣of␣cut:␣" << cut << endl;
    cout << "Remaining␣uncoloured␣vertices:␣" << calculateMargs(s, p) << endl;
    cout << "Maximal␣contribution␣to␣cut:␣";
    udouble cont = 0;
    for(int i=0; i<s.size(); i++) {
        if(s[i].getCenter().getColour() != 0)
            continue;
        if(s[i].getCenter().numOfNeig()[1]<s[i].getCenter().numOfNeig()[2]) {
            cont += (DEGREE−s[i].getCenter().numOfNeig()[2])∗s[i].getMeas()∗s[i].
                getMultiplicity();
        } else if(s[i].getCenter().numOfNeig()[1]>s[i].getCenter().numOfNeig()[2]) {
            cont += (DEGREE−s[i].getCenter().numOfNeig()[1])∗s[i].getMeas()∗s[i].
                getMultiplicity();
        } else {
            cont += (DEGREE−s[i].getCenter().numOfNeig()[1])∗s[i].getMeas()∗s[i].
                getMultiplicity()/2;
        }
    }
    cout << cont << "\n";
    cout << "So␣the␣total␣cut␣is␣" << cut+cont << "\n";
    cout << "Distribution␣of␣fully␣coloured␣stars:\n";
    cout << "Center\tRed\tBlue\tMeas\n";
                udouble lost = 0;
    for(int i=0; i<s.size(); i++) {
        if(s[i].getCenter().getColour()==0 or s[i].getCenter().numOfUncol()>0)
            continue;
        if(s[i].getCenter().getColour()==1)
            cout << "Red\t";
        else
            cout << "Blue\t";
        cout << s[i].getCenter().numOfNeig()[1] << "\t"
             << s[i].getCenter().numOfNeig()[2] << "\t"
             << s[i].getMeas()∗s[i].getMultiplicity() << "\n";
        if(s[i].getCenter().numOfNeig()[s[i].getCenter().getColour()] < s[i].getCenter().
            numOfNeig()[3−s[i].getCenter().getColour()])
            lost += s[i].getMeas()∗s[i].getMultiplicity();
    }
    cout << "Measure␣of␣lost␣vertices:␣" << lost << "\n";
    cout << endl;
    cout << "So␣the␣improved␣cut␣size␣is␣at␣most␣" << e1*cut+e1*cont−e0*lost*
        impConst << endl;
}
```

main.h

```
#include "star.h"
#include "transition.h"
#include "config.h"
#include "priority.h"

#include <vector>

using namespace std;

vector<Star> generateStars();
vector<Priority> generatePriorities();
bool update(vector<Star>&, vector<Priority>&, vector<Transition>&);
udouble calcCut(const vector<Star>&);
udouble calculateMargs(const vector<Star>&, vector<Priority>&);
void report(const vector<Star>&, vector<Priority>&);
```

transition.cpp

```
#include "transition.h"
#include "binom.h"

#include <iostream>
#include <algorithm>

void Transition::initTransitionTable(const vector<Star>& stars) {
    m_transitionTable.resize(stars.size());
    for(int from = 0; from < stars.size(); from++) {
        auto neigh = SimpleStar::firstNeighs(stars[from]);
        auto sneigh = SimpleStar::secondNeighs(stars[from]);
        neigh.insert(neigh.end(), sneigh.begin(), sneigh.end());

        for(int i=0; i<neigh.size(); i++) {
            auto to = neigh[i];
            if(!isTransitionable(stars[from], to))
                continue;
            auto tData = getTransitionData(stars[from], to);
            if(!isNegligible(tData))
```

```
                m_transitionTable[from].push_back(pair<int, vector<tDataType>>(
                    SimpleStar::getNewStarIndex(to.getOrderedVersion()), tData));

            }
        }
}

bool Transition::isTransitionable(const SimpleStar& from, const SimpleStar& to)
    const {
    if( from.getCenter().getColour() > 0 and
        from.getCenter().getColour() != to.getCenter().getColour())
        return false;
    else if(from.getCenter().getColour() == 0) {
        if(m_coloured == from.getCenter()) {
            if(to.getCenter().getColour() > 0) {
                if( from.getCenter().numOfNeig()[to.getCenter().getColour()] <
                    from.getCenter().numOfNeig()[3-to.getCenter().getColour()])
                    return false;
                }
        } else if(to.getCenter().getColour() > 0) {
            return false;
        }
    }
    auto l1 = from.getLeafs();
    auto l2 = to.getLeafs();
    for(int i=0; i<DEGREE; i++) {
        if(l1[i].getColour() != 0) { // it is coloured
            if(l1[i].getColour() != l2[i].getColour())
                return false;
        } else { // it was not coloured
            if(l2[i].getColour() != 0) { // it get coloured
                if(not(l1[i] == m_coloured)) // it couldnt get coloured
                    return false;
                if( l1[i].numOfNeig()[l2[i].getColour()] <
                    l1[i].numOfNeig()[3-l2[i].getColour()])
                    return false;
            } else {
                int dr = (l2[i].numOfNeig()[1]-(to.getCenter().getColour()==1))
                        -(l1[i].numOfNeig()[1]-(from.getCenter().getColour()==1));
                int db = (l2[i].numOfNeig()[2]-(to.getCenter().getColour()==2))
                        -(l1[i].numOfNeig()[2]-(from.getCenter().getColour()==2));
                if(dr<0 or db<0)
                    return false;
            }
        }
    }
    return true;
}

vector<tDataType> Transition::getTransitionData(const SimpleStar& from, const
    SimpleStar& to) const {
    vector<tDataType> tData(5, 0);
    if(from.getCenter().getColour() == 0) {
        if(m_coloured == from.getCenter()) {
            if(to.getCenter().getColour() == 0)
                tData[2]++;
            else
                tData[1]++;
        }
    }
    auto l1 = from.getLeafs();
    auto l2 = to.getLeafs();
    tData[0] = 1;
    for(int i=0; i<DEGREE; i++) {
        if(l1[i].getColour() == 0) { // it was not coloured
            if(l2[i].getColour() != 0) { // it get coloured
                tData[1]++;
            } else {
                if(l1[i] == m_coloured) // could get coloured but it didnt
                    tData[2]++;
                int dr = (l2[i].numOfNeig()[1]−(to.getCenter().getColour()==1))
                        −(l1[i].numOfNeig()[1]−(from.getCenter().getColour()==1));
                int db = (l2[i].numOfNeig()[2]−(to.getCenter().getColour()==2))
                        −(l1[i].numOfNeig()[2]−(from.getCenter().getColour()==2));
                int tu = (l1[i].numOfNeig()[0]−(from.getCenter().getColour()==0));
                tData[0] ∗= binom(tu, tu−dr−db)∗binom(dr+db,db);
                tData[3] += dr+db;
                tData[4] += tu−dr−db;
            }
        }
    }
    return tData;
}

bool Transition::isNegligible(const vector<tDataType>& tData) const {
    return tData[0] == 0 or tData[3]>=HOT;
}

udouble Transition::calculateTransitionValue(const vector<tDataType>& tData, const
     udouble probKnown, const udouble probUnknown) const {
    udouble result = tData[0];
    for(int i=0; i<tData[1]; i++)
```

```
        result *= probKnown;
    for(int i=0; i<tData[2]; i++)
        result *= 1-(1+m_coloured.isSymmetric())*probKnown;
    for(int i=0; i<tData[3]; i++)
        result *= probUnknown;
    for(int i=0; i<tData[4]; i++)
        result *= 1-2*probUnknown;
    return result;
}


Transition::Transition(const vector<Star>& stars, const PointType& coloured) :
    m_coloured{coloured} {
    initTransitionTable(stars);
}

void Transition::doTheTransition(vector<Star>& stars, const udouble probKnown,
    const udouble probUnknown) const {
    for(int from = 0; from < m_transitionTable.size(); from++)
        for(auto t: m_transitionTable[from]) {
            int to = t.first;
            auto tData = t.second;
            udouble diff = stars[from].getMeas() * calculateTransitionValue(tData,
                probKnown, probUnknown);
            stars[from].addDiff(-diff);
            stars[to].addDiff(diff*stars[from].getMultiplicity()/stars[to].getMultiplicity())
                ;
        }
    for(auto it = stars.begin(); it != stars.end(); ++it)
        it->updateMeas();
}

PointType Transition::getColoured() const {
    return m_coloured;
}
```

transition.h

```
#include "point.h"
#include "star.h"
#include "config.h"

#include <vector>
#include <utility>

using namespace std;
```

```
typedef int tDataType;
class Transition {
    private:
        const PointType m_coloured;
        vector<vector<pair<int,vector<tDataType> > > > m_transitionTable;

        void initTransitionTable(const vector<Star>&);
        bool isTransitionable(const SimpleStar&, const SimpleStar&) const;
        vector<tDataType> getTransitionData(const SimpleStar&, const SimpleStar
            &) const;
        bool isNegligible(const vector<tDataType>&) const;
        udouble calculateTransitionValue(const vector<tDataType>&, const udouble,
            const udouble) const;
    public:
        Transition(const vector<Star>&, const PointType&);
        void doTheTransition(vector<Star>&, const udouble, const udouble) const;
        PointType getColoured() const;
};
```

priority.cpp

```
#include "priority.h"

#include <math.h>
#include <algorithm>
#include <numeric>

Priority::Priority(const PointType& center) :
    PointType{center},
    m_meas{0}
{}

bool Priority::operator<(const Priority& p) const {
    int r1 = m_red;
    int b1 = m_blue;
    int r2 = p.m_red;
    int b2 = p.m_blue;
    if(r1+b1 == DEGREE)
        return false;
    if(r2+b2 == DEGREE)
        return true;
    if(abs(r1-b1) < abs(r2-b2))
        return true;
    if(abs(r1-b1) > abs(r2-b2))
        return false;
    if(min(r1, b1) < min(r2, b2))
        return true;
    return false;
}

udouble Priority::calculateMeas(const vector<Star>& stars) {
    vector<udouble> m;
    for(auto s: stars)
        if(s.getCenter().getColour() == 0 and s.getCenter() == *this)
            m.push_back(s.getMeas()*s.getMultiplicity());
    sort(m.begin(), m.end());
    m_meas = accumulate(m.begin(), m.end(), udouble(0));
    return m_meas;
}

udouble Priority::getMeas() const {
    return m_meas;
}

bool Priority::isColourable() const {
    return m_meas >= EPS;
}
```

priority.h

```
#include "point.h"
#include "config.h"
#include "star.h"

#include <vector>
#include <iostream>

using namespace std;

class Priority : public PointType{
    private:
        udouble m_meas;
    public:
        Priority(const PointType&);
        bool operator<(const Priority&) const;
        udouble calculateMeas(const vector<Star>&);
        udouble getMeas() const;
        bool isColourable() const;
        void printType() const {
            cout << "(" << m_red <<", " << m_blue <<"),\t";
        }
};
```

star.cpp

```
#include "star.h"

#include <algorithm>
#include <numeric>
#include <iostream>
#include <utility>
#include <map>
#include "binom.h"

using namespace std;

int numberOfRedLeafs(const vector<int>& leafOutTypes) {
    return count(leafOutTypes.begin(), leafOutTypes.end(), 0);
}

int numberOfBlueLeafs(const vector<int>& leafOutTypes) {
    return count(leafOutTypes.begin(), leafOutTypes.end(), 1);
}

static pair<int, vector<int>> computeInitDataFromIndex(const int idx) {
    int maxLeaf = DEGREE*(DEGREE+1)/2+2;
    int mIdx = 1;
    for(int i=0; i<DEGREE; i++)
        mIdx *= maxLeaf;
    int c = idx / mIdx;
    vector<int> l;
    int ci = idx % mIdx;
    for(int i = 0; i<DEGREE; i++) {
        l.push_back(ci % maxLeaf);
        ci /= maxLeaf;
    }
    return pair<int, vector<int>>(c,l);
}

static int computeMultiplicity(const vector<int>& leafOutTypes) {
    map<int, int> c;
    for(auto i: leafOutTypes)
        c[i]++;
    int res = factorial(leafOutTypes.size());
    for(auto p: c)
        res /= factorial(p.second);
    return res;
}
```

```
SimpleStar::SimpleStar(const int center, const vector<int>& leafOutTypes) :
    m_center{center, numberOfRedLeafs(leafOutTypes), numberOfBlueLeafs(
        leafOutTypes)},
    m_multiplicity{computeMultiplicity(leafOutTypes)}
{
    for(int i=0; i<DEGREE; i++) {
        m_leafs.push_back(Point::calculateLeafPoint(m_center, leafOutTypes[i]));
    }
}

SimpleStar::SimpleStar(const int idx) : SimpleStar{computeInitDataFromIndex(idx).
    first, computeInitDataFromIndex(idx).second} {}

SimpleStar::SimpleStar() : SimpleStar{-1} {cout << "Something went wrong with 
    SimpleStars...";}

Star::Star(const int center, const vector<int>& leafOutTypes, const udouble meas) :
    SimpleStar{center, leafOutTypes},
    m_meas{meas}
{}

Star::Star(const int center, const vector<int>& leafOutTypes) :
    Star{center, leafOutTypes, 0}
{}

Point SimpleStar::getCenter() const {
    return m_center;
}

vector<Point> SimpleStar::getLeafs() const {
    return m_leafs;
}

int SimpleStar::getMultiplicity() const {
    return m_multiplicity;
}

SimpleStar SimpleStar::getOrderedVersion() const {
    int c = getCenter().getColour();
    vector<int> leafOutTypes;
    for(int i=0; i<DEGREE; i++)
        leafOutTypes.push_back(Point::calculateLeafOutType(getCenter(), getLeafs()[i])
            );
    sort(leafOutTypes.begin(), leafOutTypes.end());
    return SimpleStar(c, leafOutTypes);
}
```

```
static int getStarIndex(const SimpleStar& ss) {
    int maxLeaf = DEGREE∗(DEGREE+1)/2+2;
    int mIdx = 1;
    for(int i=0; i<DEGREE; i++)
        mIdx ∗= maxLeaf;
    int idx = 0;
    auto l = ss.getLeafs();
    for(auto it = l.rbegin(); it != l.rend(); it++) {
        idx ∗= maxLeaf;
        idx += Point::calculateLeafOutType(ss.getCenter(), ∗it);
    }
    idx += ss.getCenter().getColour() ∗ mIdx;
    return idx;
}


// only for increasing leaf types
int SimpleStar::getNewStarIndex(const SimpleStar& ss) {
    int maxLeaf = DEGREE∗(DEGREE+1)/2+2;
    vector<int> T;
    auto l = ss.getLeafs();
    for(auto a: l) {
        T.push_back(Point::calculateLeafOutType(ss.getCenter(), a));
    }
    int idx = ss.getCenter().getColour() ∗ binom(DEGREE+maxLeaf−1,DEGREE);
    int S = T[DEGREE−1];
    idx += (S==0) ? 0 : binom(DEGREE+S−1,DEGREE);
    idx += binom(DEGREE+S−1,DEGREE−1) − binom(DEGREE+S−1−T[0],
        DEGREE−1);
    S −= T[0];
    for(int i=1; i<DEGREE; i++) {
        idx += binom(DEGREE+S−i−1,DEGREE−i−1);
        idx −= binom(DEGREE+S−i−1−T[i]+T[i−1],DEGREE−i−1);
        S −= T[i]−T[i−1];
    }
    return idx;
}

vector<SimpleStar> SimpleStar::firstNeighs(const SimpleStar& s) {
    vector<SimpleStar> res;
    int c = s.getCenter().getColour();
    vector<int> leafOutTypes;
    for(int i=0; i<DEGREE; i++)
        leafOutTypes.push_back(Point::calculateLeafOutType(s.getCenter(), s.getLeafs()
            [i]));
    if(c == 0) {
        res.push_back(SimpleStar(1,leafOutTypes));
        res.push_back(SimpleStar(2,leafOutTypes));
    }
    for(int i=0; i<DEGREE; i++) {
        if(leafOutTypes[i] > 1) {
            auto l = leafOutTypes;
            l[i] = 0;
            res.push_back(SimpleStar(c,l));
            l[i] = 1;
            res.push_back(SimpleStar(c,l));
            if(s.getLeafs()[i].numOfUncol()>(s.getCenter().getColour() == 0)) {
                l[i] = Point::calculateLeafOutType(s.getCenter(), s.getLeafs()[i])
                    + DEGREE − (s.getLeafs()[i].numOfNeig()[1]−(s.getCenter().
                        getColour()==1));
                res.push_back(SimpleStar(c,l));
                l[i] = Point::calculateLeafOutType(s.getCenter(), s.getLeafs()[i])
                    + 1;
                res.push_back(SimpleStar(c,l));
            }
        }
    }
    sort(res.begin(), res.end(), [](auto a1, auto a2){return getStarIndex(a1) <
        getStarIndex(a2);});
    auto it = unique(res.begin(), res.end(), [](auto a1, auto a2){return getStarIndex(
        a1) == getStarIndex(a2);});
    res.resize(distance(res.begin(), it));
    return res;
}

vector<SimpleStar> SimpleStar::secondNeighs(const SimpleStar& s) {
    auto fn = firstNeighs(s);
    vector<SimpleStar> res;
    for(auto a: fn) {
        auto f = firstNeighs(a);
        res.insert(res.end(), f.begin(), f.end());
    }
    sort(res.begin(), res.end(), [](auto a1, auto a2){return getStarIndex(a1) <
        getStarIndex(a2);});
    auto it = unique(res.begin(), res.end(), [](auto a1, auto a2){return getStarIndex(
        a1) == getStarIndex(a2);});
    res.resize(distance(res.begin(), it));
    return res;
}

udouble Star::getMeas() const {
    return m_meas;
}

void Star::setMeas(udouble meas) {
    m_meas = meas;
}

void Star::addDiff(udouble diff) {
    m_diffs.push_back(diff);
}

void Star::updateMeas() {
    m_diffs.push_back(m_meas);
    sort(m_diffs.begin(), m_diffs.end(),
        [](udouble a, udouble b){return abs(a)<abs(b);});
    m_meas = accumulate(m_diffs.begin(), m_diffs.end(), udouble(0));
    m_diffs.clear();
}
```

---

star.h

```
#include "point.h"
#include "config.h"

#include <vector>

using namespace std;

class SimpleStar {
    protected:
        Point m_center;
        vector<Point> m_leafs;
        int m_multiplicity;
        //bool m_sorted;
    public:
        SimpleStar(const int, const vector<int>&);
        SimpleStar(const int);
        SimpleStar();
        Point getCenter() const;
        vector<Point> getLeafs() const;
        int getMultiplicity() const;
        SimpleStar getOrderedVersion() const;
        //bool isSorted() const;

        //static int getStarIndex(const SimpleStar&);
        static int getNewStarIndex(const SimpleStar&);
        static vector<SimpleStar> firstNeighs(const SimpleStar&);
        static vector<SimpleStar> secondNeighs(const SimpleStar&);
};

class Star : public SimpleStar {
    private:
        udouble m_meas;
        vector<udouble> m_diffs;
    public:
        Star(const int, const vector<int>&, const udouble);
        Star(const int, const vector<int>&);
        udouble getMeas() const;
        void setMeas(udouble);
        void addDiff(udouble);
        void updateMeas();
};
```

point.cpp

```
#include "point.h"
#include "config.h"
#include <iostream>

using namespace std;

PointType::PointType(const int red, const int blue):
    m_red{red},
    m_blue{blue}
{}

bool PointType::operator==(const PointType& p) const {
    return min(m_red, m_blue) == min(p.m_red, p.m_blue) and
            max(m_red, m_blue) == max(p.m_red, p.m_blue);
}

bool PointType::isSymmetric() const {
    return m_red == m_blue;
}

int PointType::numOfUncol() const {
    return DEGREE - m_red - m_blue;
}

Point::Point(const int center, const int red, const int blue) :
    PointType{red, blue},
    m_center{center}
{}
```

```
int Point::getColour() const {
    return m_center;
}

vector<int> Point::numOfNeig() const {
    return vector<int>{DEGREE−m_red−m_blue, m_red, m_blue};
}

Point Point::calculateLeafPoint(const Point& center, const int leafOutType) {
    if(leafOutType < 2)
        return Point(leafOutType+1,−1,−1);
    int red = center.getColour()==1;
    int blue = center.getColour()==2;
    int m = DEGREE−1;
    int lt = leafOutType−2;
    while(lt>m) {
        lt −= (m+1);
        m−−;
        red++;
    }
    blue += lt;
    return Point(0, red, blue);
}

int Point::calculateLeafOutType(const Point& center, const Point& leafPoint) {
    if(leafPoint.getColour() > 0)
        return leafPoint.getColour() − 1;
    int red = leafPoint.numOfNeig()[1] − (center.getColour() == 1);
    int blue = leafPoint.numOfNeig()[2] − (center.getColour() == 2);
    return 2 + red ∗ DEGREE − (red ∗ (red − 1)) / 2 + blue;
}
```

point.h

```
#include <vector>

using namespace std;

class PointType {
    protected:
        int m_red;
        int m_blue;
    public:
        PointType(const int red, const int blue);
        bool operator==(const PointType&) const;
        bool isSymmetric() const;
        int numOfUncol() const;
};

class Point : public PointType {
    private:
        int m_center;
    public:
        Point(const int center, const int red, const int blue);
        int getColour() const; // 0: uncol, 1: red, 2: blue
        vector<int> numOfNeig() const; // (#uncol, #red, #blue)

        static Point calculateLeafPoint(const Point&, const int);
        static int calculateLeafOutType(const Point&, const Point&);
};
```

---

config.h

```
typedef long double udouble;

const udouble EPS = 0.0001;
const int DEGREE = 5;
const udouble impConst = 0.34116;
const int BINOM_BIG_NUMBER = 50; // for binomial coefficients
const int FACTORIAL_BIG_NUMBER = 10;
const int HOT = 3;
```

---

binom.cpp

```
#include "binom.h"
#include "config.h"
#include <vector>
#include <iostream>

int LOOKUP_TABLE_FOR_BINOMIAL_COEFFICIENTS[BINOM_BIG_NUMBER][
    BINOM_BIG_NUMBER];
int LOOKUP_TABLE_FOR_FACTORIAL[FACTORIAL_BIG_NUMBER];

int binom(int a, int b) {
        if(a>=BINOM_BIG_NUMBER or b>=BINOM_BIG_NUMBER) {
                std::cout << "TOO_BIG_NUMBER_IN_FUNCTION_binom\n";
                return -1;
        }
        if(a < b)
                return 0;
        if(LOOKUP_TABLE_FOR_BINOMIAL_COEFFICIENTS[a][b] == 0) {
                if(a == b or b == 0)
                        LOOKUP_TABLE_FOR_BINOMIAL_COEFFICIENTS[a][b] =
                            1;
        else
            LOOKUP_TABLE_FOR_BINOMIAL_COEFFICIENTS[a][b] =
                binom(a−1,b)+binom(a−1,b−1);
    }
    return LOOKUP_TABLE_FOR_BINOMIAL_COEFFICIENTS[a][b];
}

int invBinomU(int a, int b) {
    int c = 0;
    for(;binom(c,b)<a;c++) {}
    return c;
}

int invBinomD(int a, int b) {
    int c = 0;
    for(;binom(c,b)<=a; c++) {}
    return c;
}

int factorial(int a) {
    if(a>=FACTORIAL_BIG_NUMBER) {
        std::cout << "TOO_BIG_NUMBER_IN_FUNCTION_factorial.\n";
        return −1;
    }
    if(LOOKUP_TABLE_FOR_FACTORIAL[a] == 0) {
        if(a==0) {
            LOOKUP_TABLE_FOR_FACTORIAL[a] = 1;
        } else {
            LOOKUP_TABLE_FOR_FACTORIAL[a] = a ∗ factorial(a−1);
        }
    }
    return LOOKUP_TABLE_FOR_FACTORIAL[a];
}
```

binom.h

```
int binom(int a, int b);   // calculate a choose b
int invBinomU(int a, int b);  // calculate the least int c st. a >= binom(c,b)
int invBinomD(int a, int b);   // calculate the most int c st. a <= binom(c,b)

int factorial(int a);
```

References

[1] Díaz, Josep, Maria J. Serna, and Nicholas C. Wormald. ”Computation of the bisection width for random d-regular graphs.” LATIN 2004: Theoretical Informatics: 6th Latin American Symposium, Buenos Aires, Argentina, April 5-8, 2004. Proceedings 6. Springer Berlin Heidelberg, 2004.

[2] Lovász, László. Large networks and graph limits. Vol. 60. American Mathematical Soc., 2012.

[3] Hatami, Hamed, László Lovász, and Balázs Szegedy. ”Limits of locally–globally convergent graph sequences.” Geometric and Functional Analysis 24.1 (2014): 269-296. 1205.4356

[4] Elek, Gábor, and Gábor Lippner. ”Borel oracles. An analytical approach to constant-time algorithms.” Proceedings of the American Mathematical Society 138.8 (2010): 2939-2947.

[5] Shafique, Khurram H., and Ronald D. Dutton. ”On satisfactory partitioning of graphs.” Congressus Numerantium (2002): 183-194.

[6] Ban, Amir, and Nati Linial. ”Internal partitions of regular graphs.” Journal of Graph Theory 83.1 (2016): 5-18.

[7] Linial, Nathan, and Sria Louis. ”Asymptotically Almost Every 2 r-Regular Graph Has an Internal Partition.” Graphs and Combinatorics 36.1 (2020): 41-50.

[8] Garey, Michael R., and David S. Johnson. Computers and intractability. Vol. 174. San Francisco: freeman, 1979.

[9] Bärnkopf, Pál, Zoltán Lóránt Nagy, and Zoltán Paulovics. ”A note on internal partitions: the 5-regular case and beyond.” arXiv preprint arXiv:2109.14421 (2021).

[10] Axenovich, Maria, et al. ”Bipartite independence number in graphs with bounded maximum degree.” SIAM Journal on Discrete Mathematics 35.2 (2021): 1136-1148.