Local Computation Algorithms for Hypergraph Coloring – Following Beck's Approach

Andrzej Dorobisz
Theoretical Computer Science Department, Faculty of Mathematics and Computer Science, Jagiellonian University, Kraków, Poland

Jakub Kozik
Theoretical Computer Science Department, Faculty of Mathematics and Computer Science, Jagiellonian University, Kraków, Poland

Abstract

We investigate local computation algorithms (LCA) for two-coloring of k-uniform hypergraphs. We focus on hypergraph instances that satisfy strengthened assumption of the Lovász Local Lemma of the form $2^{1-\alpha k}(\Delta+1)\mathrm{e}<1$, where Δ is the bound on the maximum edge degree. The main question which arises here is for how large α there exists an LCA that is able to properly color such hypergraphs in polylogarithmic time per query. We describe briefly how upgrading the classical sequential procedure of Beck from 1991 with Moser and Tardos' Resample yields polylogarithmic LCA that works for α up to $1/4$. Then, we present an improved procedure that solves wider range of instances by allowing α up to $1/3$.

2012 ACM Subject Classification Mathematics of computing → Hypergraphs; Mathematics of computing → Probabilistic algorithms; Theory of computation → Streaming, sublinear and near linear time algorithms

Keywords and phrases Local Computation Algorithms, Hypergraph Coloring, Property B

Digital Object Identifier 10.4230/LIPIcs.ICALP.2023.48

Category Track A: Algorithms, Complexity and Games

Related Version *Full Version*: https://arxiv.org/abs/2305.02831 [8]

Funding This work was partially supported by Polish National Science Center (2016/21/B/ST6/02165).

1 Introduction

The problem of hypergraph coloring often serves as a benchmark for various probabilistic techniques. The task is to answer whether there exist (or to explicitly find) *a proper coloring*, that is, such an assignment of colors to the vertices of a hypergraph that no edge contains vertices all of the same color. In fact, the problem of two-coloring[1] of linear hypergraphs was one of the main motivations for introducing Local Lemma in the seminal paper of Erdős and Lovász [9]. It is well known that determining whether the given hypergraph admits proper two-coloring is NP-complete [15]. This result holds even for hypergraphs with all edges of size 3. In this work, we discuss sublinear algorithms for two-coloring of uniform hypergraphs within the framework of Local Computation Algorithms.

We are going to work with k-uniform hypergraphs[2]. For the rest of the paper, n is used to denote the number of vertices of considered uniform hypergraph, m its number of edges, and k size of the edges. We assume that k is fixed (but sufficiently large to avoid technical

[1] In two-coloring problem we can assign to each vertex one of two available colors.
[2] In *k-uniform hypergraph* each edge contains exactly k vertices.

details) and that n tends to infinity. For a fixed hypergraph, we denote by Δ its maximum edge degree. In the instances with which we are going to work, Δ is bounded by a function of k, so in terms of n it is $\mathcal{O}(1)$. This implies that the number of edges m is at most linear in n. We also assume that the considered hypergraphs do not have isolated vertices. Then, we also have $m = \Theta(n)$.

1.1 Local Computation Algorithms

Rubinfeld, Tamir, Vardi and Xie proposed in [21] a general model of sublinear sequential algorithms called Local Computation Algorithms (LCA). The model is intended to capture the situation where some computation has to be performed on a large instance but, at any specific time, only parts of the answer are required. The interaction with a local computation algorithm is organized in the sequence of queries about fragments of a global solution. The algorithm shall answer each consecutive query in sublinear time (wrt the size of the instance), systematically producing a partial answer that is consistent with some global solution. The model allows for randomness, and algorithm may occasionally fail.

For example, for the hypergraph two-coloring problem, the aim of an LCA procedure is to find a proper coloring of a given hypergraph. The algorithm can be queried about any vertex, and in response, it has to assign to the queried vertex one of the two available colors. For any sequence of queries, with high probability, it should be possible to extend the returned partial coloring to a proper one.

Formally, for a fixed problem, a procedure is a (t, s, δ)-*local computation algorithm*, if for any instance of size n and any sequence of queries, it can consistently answer each of them in time $t(n)$ using up to $s(n)$ space for computation memory. The time $t(n)$ has to be sublinear in n, but a polylogarithmic dependence is desirable. The value $\delta(n)$ shall bound the probability of failure for the whole sequence of queries. It is usually demanded to be small. The computation memory, the input, and the source of random bits are all represented as tapes with random access (the last two are not counted in $s(n)$ limit). The computation memory can be preserved between queries. In particular, it can store some partial answers determined in the previous calls. For the precise general definition of the model consult [21].

A procedure is called *query oblivious* if the returned solution does not depend on the order of the queries (i.e. it depends only on the input and the random bits). It usually indicates that the algorithm uses computation memory only to answer the current query and that there is no need to preserve information between queries. It is a desirable property, since it allows to run queries to algorithm in parallel. In a follow-up paper [3], Alon, Rubinfeld, Vardi, and Xie presented generic methods of removing query order dependence and reducing necessary number of random bits in LCA procedures. In the same paper, these techniques were applied to the example procedures (including hypergraph coloring) from [21] converting them to query oblivious LCAs. The improved procedures work not only in polylogarithmic time but also in polylogarithmic space. Mansour, Rubinstein, Vardi, and Xie in [16] improved analysis of this approach.

1.2 Constructive Local Lemma and LCA

The Lovász Local Lemma (LLL) is one of the most important tools in the field of local algorithms. In its basic form, it allows one to non-constructively prove the existence of combinatorial objects omitting a collection of undesirable properties, so-called bad events. A brief introduction to this topic and a summary of various versions of LLL can be found in the recent survey by Faragó [11].

For a fixed k-uniform hypergraph, let $p = 2^{-k}$ denote the probability that, in a uniformly random coloring, a fixed edge is monochromatic in a specific color. A straighforward application of the symmetric version of Local Lemma (see e.g., [11]) proves that the condition $2p\ (\Delta + 1)\ \mathrm{e} < 1$, is sufficient for a hypergraph with the maximum edge degree Δ, to be two-colorable.

For many years, Local Lemma resisted attempts to make it efficiently algorithmic. The first breakthrough came in 1991, when Beck [5], working on the example of hypergraph two-coloring, showed a method of converting some of LLL existence proofs into polynomial-time algorithmic procedures. However, in order to achieve that, the assumptions of Local Lemma had to be strengthened and took form

$$2\ p^{\alpha}\ (\Delta + 1)\ \mathrm{e} < 1. \tag{1}$$

For $\alpha = 1$ the inequality reduces to the standard assumption. The above inequality constraints Δ, and the constraint becomes more restrictive as α gets smaller. The original proof of Beck worked for $\alpha < 1/48$. From that time, a lot of effort has been put into studying applications to specific problems and pushing α forward, as close as possible to standard LLL criterion [2, 18, 7, 22, 19].

The next breakthrough was made by Moser in 2009. In cooperation with Tardos, Moser's ideas have been recasted in [20] into general constructive formulation of the lemma. They showed that, assuming so called variable setting of LLL, a natural randomized procedure called Resample[3] quickly finds an evaluation of involved random variables for which none of the bad events hold. They also proved that, in typical cases, the expected running time of the procedure is linear in the size of the instance. For the problem of two-coloring of k-uniform hypergraphs, the total expected number of resamplings is bounded by m/Δ (see Theorem 7 in [11]).

Adjusting constructive LLL to LCA model remains one of the most challenging problems in the area. It turns out, however, that previous results on algorithmization of Local Lemma can be adapted in the natural way. In fact, the first LCA algorithm for the hypergraph coloring from [21], is built on the variant of Beck's algorithm that is described in the book by Alon and Spencer [4]. That version works for $\alpha < 1/11$, and runs in polylogarithmic time per query. Later refinements focused on optimizing space and time requirements ([3], [16]), however, for polylogarithmic LCAs the bound on α has not been improved. In a recent work, Achlioptas, Gouleakis, and Iliopoulos [1] showed how to adjust Resample to LCA model. They did not manage, however, to obtain a polylogarithmic time. Their version answers queries in time $t(n) = n^{\beta(\alpha)}$. They establish some trade-off between the bound on α and the time needed to answer a query. In particular, when α approaches $1/2$ then $\beta(\alpha)$ tends to 1, which results in a very weak bound on the running time per query.

1.3 Main result

Our research focuses on the following general question in the area of local constructive versions of the Lovász Local Lemma: up to what value of α there exists a polylogarithmic LCA for the problem of two-coloring of k-uniform hypergraphs satisfying condition $2(\Delta + 1)\mathrm{e} < 2^{\alpha k}$. We prove the following theorem:

[3] As long as some bad events are violated, the procedure picks any such event and resamples all variables on which that event depends.

▶ **Theorem 1** (main result). *For every $\alpha < 1/3$ and all large enough k, there exists a local computation algorithm that, in polylogarithmic time per query, with probability $1 - O(1/n)$ solves the problem of two-coloring for k-uniform hypergraphs with maximum edge degree Δ, that satisfies $2\mathrm{e}(\Delta + 1) < 2^{\alpha k}$.*

Within the notation of [21] we present $(\mathrm{polylog}(n), \mathcal{O}(n), \mathcal{O}(1/n))$-local computation algorithm that properly colors hypergraphs that satisfy the above assumption. Our algorithm is not query oblivious. Moreover, typical methods of eliminating the dependence on the order of queried vertices do not seem to be applicable without sacrificing constant α. Consult the full version of this paper [8] for the complete proof of the theorem.

For comparison, Alon et al. [3] after Rubinfeld et al. [21] present a query oblivious $(\mathrm{polylog}(n), \mathrm{polylog}(n), \mathcal{O}(1/n))$-local computation algorithm working for hypergraphs satisfying

$$\begin{aligned} 16\,\Delta(\Delta-1)^3(\Delta+1) &< 2^{k_1}, \\ 16\,\Delta(\Delta-1)^3(\Delta+1) &< 2^{k_2}, \\ 2e(\Delta+1) &< 2^{k_3}, \end{aligned} \tag{2}$$

where k_1, k_2 and k_3 are positive integers such that $k = k_1 + k_2 + k_3$. These assumptions correspond to $\alpha < 1/11$.

The analysis of the LCA procedure from [3] guarantees only that the running time is of the order $\mathcal{O}\Big(\log^{\Delta}(n)\Big)$. Mansour et al. in [16] focus on improving time and space bounds within polylogarithmic class, removing the dependency on the maximal edge degree from the exponent. They obtain an LCA working in $\mathcal{O}\big(\log^4(n)\big)$ time and space, assuming that $k \geq 16\log(\Delta) + 19$, so it requires even stronger bound on α.

1.4 LOCAL distributed algorithms

The model of Local Computation Algorithms is related to the classical model of local distributed computations by Linial [14] (called LOCAL). For comparison of these two models, see work of Even, Medina, and Ron [10]. Chang and Pettie observed recently in [6] that within LOCAL model, the general problem of solving Local Lemma instances with a dependency graph of bounded degree is in some sense complete for a large class of problems (these are the problems which can be solved in sublogarithmic number of rounds). They also conjectured that for sufficiently strengthened condition of Local Lemma (like taking small enough α in (1)) there exists a distributed LOCAL algorithm that solves the problem in $\mathcal{O}(\log\log n)$ rounds. The straightforward simulation of such an algorithm within LCA framework would yield a procedure that, at least for fixed maximum degree, answers queries in polylogarithmic time.

Recently, progress towards this conjecture has been made by Fischer and Ghaffari [12], who proved that there exists an algorithm for Local Lemma instances that works in $2^{\mathcal{O}\left(\sqrt{\log\log n}\right)}$ rounds. The influence of the degree of underlying dependency graph on running time has been later improved by Ghaffari, Harris and Kuhn in [13]. In particular, for sufficiently constrained problem of hypergraph two-coloring, that result allows one to obtain an LCA procedure that answers queries in sublinear time. The time, however, would be superpolylogarithmic. Moreover, the necessary strengthening of Local Lemma assumptions appears to be much stronger than the one required to apply the result of Rubinfeld et al. [21].

The possibility of simulation of LOCAL algorithms within LCA model implies that if Chang and Pettie conjecture holds, then any problem satisfying sufficiently strengthened LLL conditions can be solved in LCA model in polylogarithmic time per query. We can therefore

formulate a weaker conjecture that for some α every such α-strengthened problem can be solved in LCA in polylogarithmic time per query. For the specific problem of hypergraph coloring, this property is known to hold. We can, however, ask what is the maximum such α for a fixed problem. That is precisely the general problem stated at the beginning of Section 1.3. It is interesting to note that our algorithms make essential use of the sequential nature of LCA. For that reason, they cannot be translated to $\mathcal{O}(\log \log n)$ LOCAL algorithms. This also illustrates an important difference between the models.

2 Main techniques and ideas of the proof

The algorithmic procedure of Beck [5] is divided into two phases. In the first one, which we call *the shattering phase*, it builds a random partial coloring that guarantees that a fraction of all edges are already properly colored. Moreover, the edges which are not yet taken care of have sufficiently many non-colored vertices to make sure that the partial coloring can be completed to a proper one. They also form connected components of logarithmic sizes which can be colored independently. Then, in the second phase, which we call *the final coloring phase*, an exhaustive search is used to complete the coloring of each component. This results in a sequential procedure with polynomial running time. In order to reduce the running time to almost linear, the shattering phase can be applied twice. Then, the final components w.h.p. are of size $\mathcal{O}(\log \log(n))$. The polylogarithmic LCA procedure for hypergraph coloring from [21] followed that approach and simulates locally two shattering phases and an exhaustive search when answering a single query. Division into these three phases is directly reflected in the conditions (2) required by the procedure.

While it is not known whether it is possible to design an LCA algorithm based solely on Resample, combining it with previous local algorithms brings significant improvements. It turns out that, within polylogarithmic time, after only one shattering phase, the coloring can be completed with the use of Resample. This simple modification, with slightly improved analysis, is sufficient to derive Theorem 1 for $\alpha \leq 1/4$. This is our first contribution. That procedure provides a reference point for explaining the intuitions and motivations that underlie the further improvements that we derive. In particular, we define a notion of *component-hypergraph* that allows for a more fine-grained analysis of the components of the residual hypergraph. For that reason, we present our base algorithm in detail in Section 3.

The first modification that we make in order to improve the base algorithm is that within the shattering phase we sample colors for all vertices. Then, for some vertices, the color is final, and for others, it is allowed to change the assigned color in the final coloring phase. Coloring all the vertices during the first phase somehow blurs the border between the shattering and final coloring phases. Its main purpose is to enable a more refined partition of the residual hypergraph into independent fragments. It also allows to determine some components of the residual hypergraph for which no recoloring would be necessary. This corresponds to a situation in which the first sampled colors in Resample happen to define a proper coloring. Altogether, we managed to significantly reduce the pessimistic size of the independent fragments colored in the final coloring phase, which enables further relaxation of the necessary conditions on α to $\alpha < 1/3$. The improved procedure is described in Section 4.

In order to analyze the procedures, we employ a common technique of associating some tree-like *witness structures* with components that require recoloring. Every such structure describes a collection of events associated with some edges of the hypergraph. All these events are determined by the colors assigned in the shattering phase. For the base algorithm, these structures are quite typical. However, in order to achieve the better bound on α, we

developed more sophisticated structures that are capable of tracking different kinds of events, which can also depend on the colors that are allowed to be recolored. Different kinds of events come with different bounds on probability. An important aspect of the analysis concerns amortization of different kinds of events within a single structure. The construction of these structures is our main technical contribution. Its detailed description can be found in the full version of this paper [8].

We finally note that, while our methods are not general enough to work for all instances satisfying the strengthened assumptions of LLL, they can be applied to a number of problems similar to hypergraph coloring, like, e.g. k-SAT.

3 Establishing base result

In this section we show how the Beck's algorithm can be combined with Resample to construct a local computation algorithm that works in polylogarithmic time per query for α up to $1/4$. In other words, we prove Theorem 1 under the stronger assumption that $\alpha \leq 1/4$. To keep the exposition simple, we first present a global randomized algorithm. Then, we comment on how to adapt this procedure to LCA model. The analysis of the procedure can be found in the full version of this paper [8].

Let $H = (V, E)$ be a hypergraph that satisfies the assumptions of Theorem 1 for a fixed $\alpha \leq 1/4$. For technical convenience, we assume that αk is an integer[4]. By assigning a random color, we mean choosing uniformly one of the two available colors. For a set of edges S, by $V(S)$ we mean all vertices covered by the edges from S. For an edge f, $N(f)$ denotes the set of edges intersecting f. We use a naming convention that is similar to other works on the subject – in particular, our view of Beck's algorithm is influenced by its descriptions by Alon and Spencer [4] and Molloy and Reed [17], as well as LCA realization given in [21].

3.1 Global coloring procedure

The algorithm starts with choosing an arbitrary order of vertices. Then, it proceeds in two phases: *the shattering phase* and *the final coloring phase*. The shattering phase colors some vertices of the input hypergraph and then splits the edges of the hypergraph that are not properly colored yet into *final components* – subhypergraphs that can be colored independently. The final coloring phase completes the coloring by considering the final components separately, one by one.

3.1.1 The shattering phase

The procedure processes vertices sequentially according to the fixed ordering. For every vertex, it either assigns a random color to the vertex or leave it non-colored in case it belongs to a *bad* edge. An edge is called *bad* if it contains $(1 - \alpha)k$ colored vertices and is still not colored properly (that is, all these vertices have the same color). Once an edge becomes bad, no more vertices from that edge will be colored – such vertices are called *troubled*. Vertices with assigned colors are called *accepted*.

Upon completion of the shattering phase, there are three types of edges:

- *safe edges* – properly colored by the accepted vertices,
- *bad edges* – containing exactly $(1 - \alpha)k$ accepted vertices, all of the same color,
- *unsafe edges* – containing fewer than $(1 - \alpha)k$ accepted vertices, all of the same color.

[4] In fact, for the given k it is only reasonable to take α in the form of t/k, where t is an integer $2 \leq t \leq k$.

Observe that in the resulting (partial) coloring, every edge that is not colored properly has at least αk troubled vertices, which will be colored in the next phase. Note also that it might happen that some unsafe edge has no colored vertices at all.

The colors of accepted vertices are not going to be changed, so the safe edges are already taken care of. Therefore, we focus on bad and unsafe edges. Let E_{bad} denote the set of all bad edges. Consider hypergraph $(V(E_{bad}), E_{bad})$. It is naturally decomposed into connected components.

▶ **Definition 2.** *Every component of the hypergraph* $(V(E_{bad}), E_{bad})$ *is called a* bad-component.

Note that every troubled vertex belongs to some bad-component. On top of them we build an abstract structure to express dependencies between bad-components through unsafe edges.

▶ **Definition 3.** *A* component-hypergraph *is constructed as follows: its vertices are bad-components of* H *and for every unsafe edge* f *intersecting more than one bad-component, an edge that contains all bad-components intersected by* f *is added to it.*

For each connected component of the component-hypergraph (that is, a maximal set of bad-components that is connected in the component-hypergraph) we construct a *final component* by taking the union of those bad-components (hence a final component is a subhypergraph of H). The shattering phase is *successful* if each final component contains at most $2(\Delta+1)\log(m)$ bad edges. If this is not the case, the procedure declares a failure. It turns out that this is very unlikely to happen.

3.1.2 The final coloring phase

For each final component $\mathcal{C}$ determined during the shattering phase, we add to $\mathcal{C}$ all unsafe edges intersecting it, and then, we restrict $\mathcal{C}$ to troubled vertices[5]. We obtain a hypergraph $\mathcal{C}'$ containing at most $2(\Delta+1)^2\log(m)$ edges, and each of them has at least αk vertices. The maximum edge degree in $\mathcal{C}'$ cannot be larger than Δ, which is the maximum edge degree in H. Since $2e(\Delta+1) < 2^{\alpha k}$ (by the assumptions of Theorem 1), Lovász Local Lemma ensures that $\mathcal{C}'$ is two-colorable. Hence, by the theorem of Moser and Tardos Resample finds a proper coloring of it using on average $|E(\mathcal{C}')|/\Delta$ resamplings (see Theorem 7 in [11]).

When the final coloring phase is over, all final components are properly colored. Since each bad or unsafe edge is dealt within some final component, and each safe edge was properly colored during the shattering phase, it is now guaranteed that the constructed coloring is proper for the whole H.

3.2 LCA realization

We employ quite standard techniques to obtain an LCA realization of the described algorithm. We articulate it below to provide a context for the description of our main algorithm. An important property of the described procedure is that the ordering of vertices does not have to be fixed a priori. In fact it can be even chosen in an on-line manner by an adversary. Following [21], we are going to exploit the freedom of choice of ordering. The LCA version of the algorithm is going to simulate the global version run with a specific ordering. That ordering is constructed dynamically during the evaluation and is driven by the queries. Apart

[5] Restriction of $H=(V,E)$ to $V'\subseteq V$ is defined as $H'=(V',\{e\cap V' |\ e\in E, e\cap V'\neq\emptyset\})$.

from some minor adjustment (resulting from adaptation to LCA model) when the algorithm is queried about vertex v, it performs all the work of the standard algorithm needed to assign a final color to v. The LCA version is presented in Listings 1, 2, 3, and 4. All colors assigned during work of the algorithm are stored in the computation memory (which is preserved between queries). For convenience, we also store there the status of each vertex – *uncolored*, *accepted* or *troubled*. Initially all vertices are uncolored.

Algorithm 1 LCA for uniform hypergraph coloring – main function.

```
1  Procedure QUERY(v - vertex):
2      if v is uncolored then
3          if all edges containing v are not bad then
4              assign a random color to v and mark it as accepted    // shattering
5          else mark v as troubled
6      if v is troubled then
7          C_v ← BUILD_FINAL_COMPONENT(v)                            // shattering
8          COLOR_FINAL_COMPONENT(C_v)                              // final coloring
9      return color assigned to v
```

3.2.1 query

When a vertex v has been already marked as accepted, its color is immediately returned. If it has not been processed before, the algorithm checks whether v belongs to any bad edge (that requires inspecting the current statuses of all the edges that contain v). If not, a random color is assigned to v, the vertex is marked as accepted, and the procedure returns the assigned color. On the other hand, when v belongs to a bad edge, it is marked as troubled. The algorithm then determines the final component containing v in procedure BUILD_FINAL_COMPONENT. These steps can be viewed as the shattering phase. Afterwards, the final coloring phase is performed for the final component in procedure COLOR_FINAL_COMPONENT.

Algorithm 2 Building the final component for v that belongs to some bad edge.

```
1  Procedure BUILD_FINAL_COMPONENT(v - troubled vertex):
2      B ← ∅                  // initialize set of bad edges of the component
3      U ← ∅                     // initialize set of unsafe edges to process
4      e ← any bad edge containing v
5      mark e as explored and run EXPAND_BAD_COMPONENT(e, B, U)
6      // process surrounding unsafe edges
7      while U is not empty do
8          f ← next edge from U (remove it from U)
9          EXPAND_VIA_UNSAFE(f, B, U)
10     // return hypergraph built on set of bad edges
11     return C = (V(B), B)
```

3.2.2 build_final_component

This procedure builds the set B of bad edges of the final component of v, exploring the line graph of H[6]. It uses a temporary flag *explored* to mark visited edges (this flag is not preserved between queries). The construction starts from a bad edge containing troubled vertex v and expands it to a bad-component. Then, as long as possible, set B is extended by edges of neighboring bad-components, which can be reached through unsafe edges adjacent to B. If at some point the number of bad edges in B exceeds the prescribed bound $2(\Delta+1)\log(m)$, then the procedure declares a failure (note that it cannot be restarted since LCA model does not allow to change colors returned for previous queries). Construction of the final component is done when there are no more bad edges to add. Then, the hypergraph $\mathcal{C} = (V(B), B)$ built on the collected bad edges is returned.

The expansion of bad-components is done within subprocedure `EXPAND_BAD_COMPONENT`. It starts from the given bad edge and explores the line graph by inspecting the adjacent edges. For each adjacent edge, its type (safe, unsafe, or bad) is determined using `DETERMINE_EDGE_STATUS`. Determining status of an edge may require processing some uncolored vertices of that edge. For each of them, the procedure check whether it is troubled. If it is not, a random color is assigned to the vertex and the vertex is marked as accepted.

Algorithm 3 Subprocedures for the final component construction.

```
1  Procedure EXPAND_BAD_COMPONENT(e - bad edge, B - bad edges, U - unsafe edges):
2      Q ← {e}                          // initialize set of bad edges to process
3      while Q is not empty do
4          f ← next edge from Q (remove it from Q)
5          add f to B and if |B| > 2(Δ + 1) log(m) then FAIL
6          for g ∈ N(f) which are not explored do
7              mark g as explored and DETERMINE_EDGE_STATUS(g)
8              if g is bad then add g to Q
9              if g is unsafe then add g to U
10
11 Procedure EXPAND_VIA_UNSAFE(f - unsafe edge, B - bad edges, U - unsafe edges):
12     for g ∈ N(f) which are not explored do
13         DETERMINE_EDGE_STATUS(g)
14         if g is bad then
15             mark g as explored and run EXPAND_BAD_COMPONENT(g, B, U)
16
17 Procedure DETERMINE_EDGE_STATUS(g - edge):
18     for each w in g that is uncolored unless g becomes safe do
19         if some edge containing w (including g) is bad then mark w as troubled
20         else assign a random color to w and mark it as accepted
21     count accepted vertices and check their colors to determine status of g
```

During the expansion through unsafe edges we keep a set U of not processed unsafe edges that intersects any edge of B. As long as U is not empty, we pick any unsafe f from U and process it by `EXPAND_VIA_UNSAFE`. Here we determine the statuses of all edges

[6] The line graph $L(H)$ is the graph built on $E(H)$ in which two distinct vertices (representing edges of H) are adjacent if the corresponding edges intersect.

adjacent to f and if we encounter a bad edge which is not in B, then we add it and expand a bad-component containing it. For technical convenience, during bad-component expansion we collect non-explored adjacent unsafe edges and add them to U.

Algorithm 4 Finding coloring inside the final component.

```
1  Procedure COLOR_FINAL_COMPONENT(C - hypergraph):
2      add to C all unsafe edges intersecting C
3      C' ← restriction of C to troubled vertices
4      t_e ← |E(C')|/Δ                    // expected time of one RESAMPLE trial
5      for trial = 1 to 2 log(m) do
6          // RESAMPLE with limited number of steps
7          assign random colors to V(C')
8          for step = 1 to 2t_e do
9              if there is monochromatic f ∈ E(C') then
10                 assign new random colors to all vertices of f
11             else
12                 // C' is properly colored
13                 mark all vertices of C' as accepted and return
14     FAIL
```

3.2.3 color_final_component

Final component $\mathcal{C}$ is extended with unsafe edges that intersect it. Then it is restricted to the set of its troubled vertices. The resulting hypergraph is denoted by $\mathcal{C}'$. The algorithm tries to find a proper coloring of $\mathcal{C}'$ using Resample procedure. To ensure polylogarithmic time, it is run only for the limited number of resampling steps. To decrease the probability of a failure, the procedure may be restarted a few times. When a proper coloring is found, each vertex of $\mathcal{C}'$ is marked as accepted. From now on, all edges of $\mathcal{C}$ are treated as safe. However, if all trials were unsuccessful, the procedure declares a failure.

4 Main result – algorithm

We show how to improve the base procedure described in the previous section to obtain an algorithm that can be used to prove Theorem 1, that is, an algorithm that works in polylogarithmic time per query on input hypergraphs that satisfy strengthened LLL condition (1) for $\alpha < 1/3$. Actually, our procedure can be used to find a proper coloring also for instances that satisfy that condition with any $\alpha \in (0, 1)$, but the running time is not guaranteed for $\alpha \geq 1/3$. We start with introducing the main ideas behind algorithm improvement and describe its global version. Then, we discuss how to adapt it to the model of the local computation algorithms, and finally we present a description of the LCA procedure. The analysis of the algorithm can be found in the full version of this paper [8].

4.1 A general idea

It is a common approach in randomized coloring algorithms to start from an initial random coloring and then make some correction to convert it to a proper one (like in Resample [20] or in Alon's parallel algorithm [2]). This is not the case of Beck's procedure, in which a

proper coloring is constructed incrementally, but coloring of some vertices (those marked as troubled) is postponed to the later phase. Our approach lies somewhere in between. We generally try to follow the latter one, but we sample colors for the troubled vertices already in the shattering phase. Such colors are considered as *proposed*, and we reserve the possibility of changing them in the final coloring phase. We use the information about the proposed colors to shrink the area that will be processed in the final coloring phase. In particular, if we look at the colors proposed for troubled vertices, then only those final components that contain a monochromatic edge require recoloring. Moreover, if we carefully track dependencies between bad-components (see Definition 2), it is also possible to decrease the sizes of the final components. We explain this idea in more detail in the following subsections.

4.1.1 Activation of bad-components

Imagine that all the vertices were colored in the shattering phase and we want to determine the final components. We look at the component-hypergraph (see Definition 3) and have to decide which of the bad-components should be recolored. We start from bad-components that are intersected by monochromatic edges - we mark them as *initially active* and treat them as seeds of final components. The remaining ones are currently *inactive*. Our intention is to recolor only active components in the final coloring phase. Note that it might not be sufficient to alter the coloring in a way that makes initially active components properly colored, because after their recoloring, it is possible that some unsafe edge which get both colors in the shattering phase becomes monochromatic. That is why the activation has to be propagated. We use the following *propagation rule*:

- let A_t be the set of troubled vertices that are covered by active bad-components, and f be an unsafe edge that intersects A_t; if $f \setminus A_t$ is monochromatic, then all inactive bad-components that intersect f become active and all bad-components that intersect f are merged into one (eventually final) component.

The above propagation rule is applied as long as possible. When it stops, it is guaranteed that all monochromatic edges are inside active components and all unsafe and bad edges outside of active components are properly colored by the vertices that are outside of active bad-components. In particular, we can accept all the colors proposed for inactive vertices.

4.1.2 Edge trimming

We employ an additional technique, which can further reduce the area of the final components. Observe that, in order to guarantee two-colorability of the final components, it is enough to ensure that each edge has at least αk vertices to recolor inside one final component. It means that if some active component already contains αk troubled vertices of some edge, then it is not necessary to propagate activation through that edge. Thus, we can improve the propagation rule in the following way. Consider an unsafe edge f for which $f \setminus A_t$ is monochromatic (recall that A_t denotes the set of currently active troubled vertices). If some active component contains at least αk troubled vertices of f, then f is trimmed to that active component. Otherwise, all bad-components intersected by f are activated and merged into one component (as described in the previous section).

We point out that the direct inspiration for this technique came from the work of Czumaj and Scheideler [7] in which the edge trimming is actively used during the construction of the area to be recolored. One of the consequences of using it is that the shapes of the final components depend on the specific order in which activation is propagated.

4.2 Global coloring procedure

Similarly to the base algorithm from Section 3.1, the improved procedure performs the shattering phase and then the final coloring phase. The former is modified according to the ideas described in the previous subsection. In particular, each vertex gets a color but we use the notions of *proposed* and *accepted* colors to distinguish colors that can be changed. The latter phase is almost the same. Pseudocode of the whole procedure can be found in Listing 5 in Appendix A.

4.2.1 The shattering phase

The procedure processes the vertices in a fixed order. For each vertex, it marks it as *accepted* or *troubled*, and then chooses a random color for it. A vertex is accepted if, at the time of processing, it does not belong to any of the bad edges. Otherwise, it is troubled. An edge becomes *bad* when its set of accepted vertices reaches size $(1-\alpha)k$ and is still monochromatic.

After processing all the vertices, *safe* and *unsafe* edges are determined in the same way as in the base algorithm. Additionally, by a *monochromatic* edge, we mean an edge for which all its vertices (accepted and troubled) have the same color. The colors of the accepted vertices are called *accepted colors*. The colors of the troubled vertices are called *proposed colors*. By accepting a color assigned to a vertex, we mean changing its status to accepted.

The next step involves determining the final components. We work with the component-hypergraph. We are going to mark some bad-components and unsafe edges as *active*. By an *active component*, we mean a maximal set of active bad-components which is connected in the component-hypergraph via active unsafe edges. We start with marking as active all monochromatic unsafe edges and all bad-components that are intersected by any (bad or unsafe) monochromatic edge. Let A_t denote the set of troubled vertices that are currently covered by active bad-components. Then, as long as there exists an inactive unsafe edge f satisfying the following conditions:

- f is monochromatic outside the active troubled area (i.e., $f \setminus A_t$ is monochromatic), and
- each active component contains less than αk troubled vertices of f,

we activate f and activate all bad-components intersected by f. When this propagation rule can no longer be applied, we accept the colors of all the troubled vertices from inactive bad-components. At that time, each active component determines a final component as the union of its bad-components. Just like in the base algorithm, the shattering phase is *successful* if each final component contains at most $2(\Delta+1)\log(m)$ bad edges. Otherwise, the procedure declares a failure.

4.2.2 The final coloring phase

We implement one modification at the beginning of the final coloring phase. For each final component $\mathcal{C}$, we add to $\mathcal{C}$ not all unsafe edges intersecting it, but only those that have at least αk troubled vertices in $V(\mathcal{C})$. Then, we proceed exactly as in the base algorithm: we restrict $\mathcal{C}$ to the troubled vertices and apply Resample.

4.3 Ideas behind LCA realization

In the base case, the conversion of the global algorithm to LCA is straightforward. In fact, the LCA version determines the same area to recolor (assuming that both versions process the vertices in the same order). For the improved algorithm described in the previous subsection, conversion to LCA is more complex and alters the behavior of the algorithm. The main

difficulty is that for a bad-component alone that is not initially active, it is not easy to quickly decide whether it is going to be activated or not. There might exist a long chain of activation leading to an activation of the considered bad-component, and we do not know in which direction to search for the sources of this eventual activation. Moreover, even if we find out that it will be activated, it is not obvious what the shape of the final component containing it will be, since it requires performing activation propagation and determining activation statuses of neighboring bad-components as well. To address these problems, when a troubled vertex of some bad-component is queried, we focus on finding an area containing that vertex that can be recolored independently from the remaining part of the input hypergraph. It means that from the beginning of the procedure the component of that vertex is treated as active and we allow trimming unsafe edges to that component. Moreover, we use additional techniques described below to limit the expansion of the processed area in a single query.

4.3.1 Trimming to bad-component

We extend edge trimming to the case when an unsafe edge f has at least αk troubled vertices in some bad-component S, and the set of those vertices together with the accepted vertices of f is not monochromatic. In such a case, f can be trimmed by removing from it the troubled vertices that do not belong to S. Note that we do not check here whether S is active or not. The idea behind this step is that from now on S is responsible for the proper coloring of f. If at some point, the colors of the vertices of S get accepted without any resamplings, then f will be obviously colored properly. Otherwise, if S becomes active, then f will be trimmed anyway, and S has enough troubled vertices of f to not break two-colorability of S.

4.3.2 Activation exclusion

The necessary condition for an inactive bad-component S to be activated is that there is an unsafe edge f whose accepted vertices and troubled vertices in $f \cap V(S)$ are of the same color. When there is no such edge or all such edges were trimmed to other components, then S cannot be activated. Therefore if it is not initially active, it stays inactive. In such a case, we can accept all the proposed colors for the vertices of S. As a result, some unsafe edges become properly colored, and we can treat them as safe. This, in turn, may enable proving that neighboring bad-components will also not be activated. The same reasoning can be applied to a set C of bad-components. If none of the bad-components in C is initially active and there are no unsafe edges intersecting some bad-component outside C that may activate bad-component from C, then we can conclude that all bad-components in C remain inactive.

4.3.3 Conditional expansion

The idea described in the previous subsection can be used for a bad-component to perform some kind of search for a potential reason of activation. If S_1 is not initially active, we inspect unsafe edges that may cause the activation of S_1. We can select any such f, and ask whether other bad-component S_2 intersected by f may become active. We can continue that procedure as long as there is a risk of activating any S_i from the group of bad-components visited so far. In the end, we either find some initially active component or we prove that all the considered bad-components cannot be activated. It turns out that, if we do not follow the edges that can be trimmed with the trimming to bad-component technique, then the processed area during such a search is unlikely to be large.

The possibility of finding an initially active bad-component can be used in expansion of the component to extend it by a neighboring area. For a selected bad-component adjacent to the currently constructed eventually final component, we launch a search and either we find

some monochromatic edge (initially active component) and extend the component with the whole searched area, or convince ourselves that this area cannot be activated. In the latter case we can simply accept the proposed colors in that area. In the former we can perform the expansion because the occurrence of a monochromatic edge, as an unlikely event, in a sense amortizes the expansion of the component. In fact, we can stop the search procedure not only when we find a monochromatic edge but also in a less restrictive case when we find an unsafe edge intersecting at least two disjoint bad edges outside the search area. This possibility follows from the technical details of the analysis.

4.4 LCA procedure

We describe the improved LCA procedure in reference to the base algorithm presented in Section 3.2. As previously, the ordering of the vertices is constructed dynamically and is driven by the queries and the work of the algorithm. For a set of edges S, by $V_t(S)$ we mean all troubled vertices in $V(S)$. For an edge f, we denote by $f|_t$ the set of troubled vertices of f, and by $f|_a$ the set of accepted vertices of f.

4.4.1 query

The main procedure is almost identical to its counterpart in the base algorithm (Listing 1). The only difference is that when processing a vertex v of a bad edge, it is not only marked as troubled, but also a random color is assigned to v.

4.4.2 build_final_component

This procedure is the heart of the algorithm and is substantially more complex than its analogue in the base version. It is presented in Listings 6 and 7 available in Appendix A. It also makes use of subprocedures defined earlier (see Listing 3), with one modification in `DETERMINE_EDGE_STATUS` – once a vertex w is marked as troubled, a random color is also assigned to w. As previously, the procedure works on the line graph of H and grows a set B of bad edges that will be converted to a final component at the end of the procedure. It always starts from the bad-component containing the queried vertex v, and expands it by neighbor bad-components via unsafe edges. The main change is that in the base algorithm each unsafe edge causes expansion of the component, here unsafe edges are processed more carefully. Throughout the procedure we make sure that the size of B does not exceed $2(\Delta+1)\log(m)$ bound on number of edges – if that happens, the procedure stops and declares a failure.

Let U be the set of not processed unsafe edges intersecting $V(B)$. If some edge can be trimmed to $V(B)$, it can be safely removed from U. Thus, we may assume that each f in U has fewer than αk troubled vertices in $V(B)$. Since every unsafe edge has more than αk troubled vertices, each f from U has to intersect at least one bad-component outside $V(B)$. The procedure applies the following *extension rules* as long as possible:

- (r1) if there exists f in U that intersects at least two disjoint bad edges outside B, or
- (r2) if there exists f in U for which all the vertices of f outside of $V_t(B)$ are monochromatic,

then B is extended with all bad edges from the bad-components intersected by f;

- (r3) if there are no edges in U that meet the conditions (r1) or (r2), but there exists f in U that has fewer than αk troubled vertices outside $V(B)$,

then call `EXPAND_OR_ACCEPT` procedure (described in the following subsection) for f, which implements the conditional expansion technique, and extend B with the returned set of bad edges (which may happen to be empty).

Note that, when there are no edges that meet conditions (r1) or (r2), then for any remaining f from U it is guaranteed that f intersects exactly one bad-component outside $V(B)$ and $f \setminus V_t(B)$ is not monochromatic. If such f does not satisfy condition (r3), it has at least αk troubled vertices in that external bad-component, so it can be trimmed to it (according to trimming to bad-component technique). Thus, f can be removed from U.

After each extension rule, the processed edge is removed from U. On the other hand, when B is extended, new unsafe edges may be added to U, but we remove those that can now be trimmed to $V(B)$. Since edges which do not fulfill any of the extension rules are also removed from U, finally U becomes empty and the procedure stops. At this point, B is a set of bad edges which are surrounded only by safe and trimmed unsafe edges.

4.4.3 expand_or_accept

This procedure is an implementation of the conditional expansion technique, through a given unsafe edge e. Similarly to BUILD_FINAL_COMPONENT, it grows a set A of bad edges, which we call a *search area*, and makes sure that its size does not exceed $2(\Delta+1)\log(m)$ bound (if that happens, the whole algorithm stops and declares a failure). Initially, A is empty. Then it becomes expanded by bad-components which may lead to initially active bad-component, starting from the not explored bad-component intersected by e. The expansion naturally stops when there are no more candidate bad-components. The procedure, however, can also stop earlier in case when some monochromatic edge or unsafe edge intersecting two disjoint not explored bad edges is found.

Let Q be the set of unsafe edges to be processed (initially it is empty). Let C be the set of bad edges of the currently expanded bad-component. Let U_C denote the set of unsafe edges intersecting $V(C)$ but not adjacent to the edges of B and A (these are simply those unsafe edges adjacent to the edges in C that were not explored before expansion of C). The procedure extends A with all edges from C, and then looks for the following *amortizing configuration*:

- (e1) if C contains monochromatic edge f

then the procedure stops and returns set A;

- (e2) if U_C contains a monochromatic edge f, or
- (e3) if U_C contains an edge f, which intersects at least two disjoint bad edges outside C,

then first set A is extended with all the bad edges of the bad-components intersected by f, and then the procedure stops and returns A.

When no such configuration is found, all unsafe edges in U_C are not monochromatic and, moreover, each intersects at most one bad-component outside A. We focus on the edges from U_C that can cause an activation of C – these are the edges whose troubled vertices in $V(C)$ together with accepted vertices are monochromatic. Each such an edge f has to intersect exactly one external bad-component and troubled vertices of that component together with $f|_a$ ensure a proper coloring of f. If there are at least αk troubled vertices of f in that external bad-component, f can be trimmed to it (according to the technique of trimming to bad-component). That is why we add to Q only those edges from U_C that may cause activation of C and have fewer than αk troubled vertices outside of $V(C)$.

When processing of C is finished, we pick any edge from Q (the set of unsafe edges to be processed) and repeat the above steps for the external bad-component intersected by the selected edge. It may happen that this component has already been added to A, in a such case the procedure continues picking edges from Q. When the procedure finishes without encountering amortizing configuration, there are no monochromatic edges in A and all unsafe edges intersecting $V(A)$ are either properly colored by the colors of the accepted vertices and

the vertices from $V_t(A)$, or are trimmed to bad-components outside it. Thus, an activation of whole A is excluded. Then we mark all vertices in $V_t(A)$ as accepted and treat edges properly colored by their colors as safe. In that case, the procedure returns the empty set.

Note that during this procedure, we do not apply edge trimming to $V(A)$ when it covers at least αk troubled vertices of some unsafe edge, since it can result in a false activation (in case the edge is monochromatic inside $V(A)$). We also ignore all unsafe edges intersecting $V(B)$ (they were explored before call to `EXPAND_OR_ACCEPT`) since, due to not satisfying (r1) and (r2) they cannot be used in an amortizing configuration or cause an activation (it is guaranteed that they are not monochromatic outside $V_t(B)$).

4.4.4 color_final_component

The last procedure is almost identical to its counterpart in the base algorithm (Listing 4). Recall that the only change is at the beginning of the procedure. Instead of extending $\mathcal{C}$ with all unsafe edges intersecting it, only those unsafe edges that have at least αk troubled vertices in $V(\mathcal{C})$ are added. Then we proceed as in the base algorithm.

References

1 Dimitris Achlioptas, Themis Gouleakis, and Fotis Iliopoulos. Simple local computation algorithms for the general Lovász Local Lemma. In Christian Scheideler and Michael Spear, editors, *SPAA '20: 32nd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, July 15-17, 2020*, pages 1–10. ACM, 2020. doi:10.1145/3350755.3400250.

2 Noga Alon. A parallel algorithmic version of the local lemma. *Random Structures Algorithms*, 2(4):367–378, 1991. doi:10.1002/rsa.3240020403.

3 Noga Alon, Ronitt Rubinfeld, Shai Vardi, and Ning Xie. Space-efficient local computation algorithms. In Yuval Rabani, editor, *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 1132–1139. SIAM, 2012. doi:10.1137/1.9781611973099.89.

4 Noga Alon and Joel H. Spencer. *The Probabilistic Method, Second Edition*. John Wiley, 2000. doi:10.1002/0471722154.

5 József Beck. An algorithmic approach to the Lovász local lemma. I. *Random Structures Algorithms*, 2(4):343–365, 1991. doi:10.1002/rsa.3240020402.

6 Yi-Jun Chang and Seth Pettie. A time hierarchy theorem for the LOCAL model. *SIAM J. Comput.*, 48(1):33–69, 2019. doi:10.1137/17M1157957.

7 Artur Czumaj and Christian Scheideler. Coloring non-uniform hypergraphs: a new algorithmic approach to the general Lovász local lemma. In David B. Shmoys, editor, *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms, January 9-11, 2000, San Francisco, CA, USA*, pages 30–39. ACM/SIAM, 2000. URL: https://dl.acm.org/doi/10.5555/338219.338229.

8 Andrzej Dorobisz and Jakub Kozik. Local computation algorithms for hypergraph coloring – following Beck's approach (full version), 2023. arXiv:2305.02831.

9 Paul Erdős and László Lovász. Problems and results on 3-chromatic hypergraphs and some related questions. In *Infinite and finite sets (Colloq., Keszthely, 1973; dedicated to P. Erdős on his 60th birthday), Vol. II*, volume 10 of *Colloquia Mathematica Societatis János Bolyai*, pages 609–627. North-Holland, Amsterdam, 1975.

10 Guy Even, Moti Medina, and Dana Ron. Best of two local models: Centralized local and distributed local algorithms. *Inf. Comput.*, 262:69–89, 2018. doi:10.1016/j.ic.2018.07.001.

11 András Faragó. A meeting point of probability, graphs, and algorithms: The Lovász Local Lemma and related results – A survey. *Algorithms*, 14(12):355, 2021. doi:10.3390/a14120355.

12 Manuela Fischer and Mohsen Ghaffari. Sublogarithmic distributed algorithms for Lovász Local Lemma, and the complexity hierarchy. In Andréa W. Richa, editor, *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, volume 91 of *LIPIcs*, pages 18:1–18:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.DISC.2017.18.

13 Mohsen Ghaffari, David G. Harris, and Fabian Kuhn. On derandomizing local distributed algorithms. In Mikkel Thorup, editor, *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018, Paris, France, October 7-9, 2018*, pages 662–673. IEEE Computer Society, 2018. doi:10.1109/FOCS.2018.00069.

14 Nathan Linial. Distributive graph algorithms global solutions from local data. In *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*, pages 331–335, 1987. doi:10.1109/SFCS.1987.20.

15 László Lovász. Coverings and coloring of hypergraphs. In *Proceedings of the Fourth Southeastern Conference on Combinatorics, Graph Theory, and Computing (Florida Atlantic Univ., Boca Raton, Fla., 1973)*, pages 3–12, 1973.

16 Yishay Mansour, Aviad Rubinstein, Shai Vardi, and Ning Xie. Converting online algorithms to local computation algorithms. In Artur Czumaj, Kurt Mehlhorn, Andrew M. Pitts, and Roger Wattenhofer, editors, *Automata, Languages, and Programming – 39th International Colloquium, ICALP 2012, Warwick, UK, July 9-13, 2012, Proceedings, Part I*, volume 7391 of *Lecture Notes in Computer Science*, pages 653–664. Springer, 2012. doi:10.1007/978-3-642-31594-7_55.

17 Michael Molloy and Bruce Reed. *Graph colouring and the probabilistic method*. Springer, 2002. doi:10.1007/978-3-642-04016-0.

18 Michael Molloy and Bruce A. Reed. Further algorithmic aspects of the Local Lemma. In Jeffrey Scott Vitter, editor, *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23-26, 1998*, pages 524–529. ACM, 1998. doi:10.1145/276698.276866.

19 Robin A. Moser. Derandomizing the Lovasz Local Lemma more effectively. *CoRR*, abs/0807.2120, 2008. arXiv:0807.2120.

20 Robin A. Moser and Gábor Tardos. A constructive proof of the general lovász local lemma. *J. ACM*, 57(2):11:1–11:15, 2010. doi:10.1145/1667053.1667060.

21 Ronitt Rubinfeld, Gil Tamir, Shai Vardi, and Ning Xie. Fast local computation algorithms. In Bernard Chazelle, editor, *Innovations in Computer Science – ICS 2011, Tsinghua University, Beijing, China, January 7-9, 2011. Proceedings*, pages 223–238. Tsinghua University Press, 2011.

22 Aravind Srinivasan. Improved algorithmic versions of the Lovász Local Lemma. In Shang-Hua Teng, editor, *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2008, San Francisco, California, USA, January 20-22, 2008*, pages 611–620. SIAM, 2008. URL: https://dl.acm.org/doi/10.5555/1347082.1347150.

A Listings of the improved procedure

A.1 Listing of the global algorithm

Algorithm 5 Improved algorithm for uniform hypergraph coloring.

Procedure HYPERGRAPH_COLORING(H - *hypergraph*):
  // I. SHATTERING PHASE
  let $(v_1, v_2, ...v_n)$ be an ordering of $V(H)$
  for $i = 1$ **to** n **do**
    if *all edges containing* v_i *are not bad* **then**
      mark v_i as `accepted`
    else
      mark v_i as `troubled`
    assign a random color to v_i
  determine status of each $e \in E(H)$ // e is bad, safe, or unsafe
  explore the line graph and build component-hypergraph $H_C = (V_C, E_C)$
  // activation of bad-components
  // - let U_C be the set of unsafe edges corresponding to E_C
  // - let $U(B)$ denote unsafe edges intersecting component B
  // - let $U_C(B) = U(B) \cap U_C$
  // - let $V_t(\mathcal{C})$ denote set of troubled vertices in component $\mathcal{C}$
  $\mathcal{A} \leftarrow \emptyset$ // initialize set of active components
  $Q \leftarrow \emptyset$ // unsafe edges to process
  // - initial activation
  foreach $B \in V_C$ **do**
    if *some* $e \in E(B)$ *or* $f \in U(B)$ *is monochromatic* **then**
      mark B as *active*
      add B to $\mathcal{A}$ **and** add all edges from $U_C(B)$ to Q
    else mark B as *inactive*
  foreach $f \in U_C$ **do**
    if f *is monochromatic* **then** merge in $\mathcal{A}$ all $\mathcal{C} \in \mathcal{A}$ intersected by f
  // - activation propagation
  while Q *is not empty* **do**
    $f \leftarrow$ next edge from Q (remove it from Q)
    if $\forall_{\mathcal{C} \in \mathcal{A}} |f \cap V_t(\mathcal{C})| < \alpha k$ ***and*** $f \setminus V_t(\bigcup \mathcal{A})$ *is monochromatic* **then**
      // - activate new bad-components through f
      foreach $B \in V_C$ *such that* B *is* inactive *and* f *intersects* B **do**
        mark B as *active*
        add B to $\mathcal{A}$ **and** add all edges from $U_C(B)$ to Q
      // - merge active components through f
      merge in $\mathcal{A}$ all $\mathcal{C} \in \mathcal{A}$ intersected by f

  // II. FINAL COLORING PHASE - color each final component
  foreach $\mathcal{C} \in \mathcal{A}$ **do**
    foreach $f \in U(\mathcal{C})$ *such that* $|f \cap V_t(\mathcal{C})| \geq \alpha k$ **do** add f to $\mathcal{C}$
    $\mathcal{C}' \leftarrow$ restriction of $\mathcal{C}$ to troubled vertices
    RESAMPLE($\mathcal{C}'$)

A.2 Listing of build_final_component (LCA)

Algorithm 6 Improved LCA procedure for the final component construction.

```
1  Procedure BUILD_FINAL_COMPONENT(v - troubled vertex):
2      B ← ∅                    // initialize set of bad edges of the component
3      U ← ∅                    // initialize set of unsafe edges to process
4      U_s ← ∅                  // unprocessed unsafe edges able to launch search
5      e ← any bad edge containing v
6      mark e as explored and run EXPAND_BAD_COMPONENT(e, B, U)
7      // process surrounding unsafe edges according to extension rules
8      while U ≠ ∅ or U_s ≠ ∅ do
9          while U is not empty do
10             f ← next edge from U (remove it from U)
11             if f has < αk troubled vertices in V(B) then
12                 if f satisfies rule (r1) or (r2) then
13                     EXPAND_VIA_UNSAFE(f, B, U)
14                 else if f can satisfy rule (r3) then
15                     add f to U_s        // f \ V(B) has < αk troubled vertices
16         if U_s is not empty then
17             f ← next edge from U_s (remove it from U_s)
18             if f has < αk troubled vertices in V(B) then
19                 // f satisfies rule (r3)
20                 (A, U_A) ← EXPAND_OR_ACCEPT(f, B, U)
21                 B = B ∪ A and if |B| > 2(Δ + 1) log(m) then FAIL
22                 U = U ∪ U_A
23     // return hypergraph built on set of bad edges
24     return C = (V(B), B)
```

A.3 Listing of expand_or_accept (LCA)

Algorithm 7 Conditional expansion via unsafe edge e (exploring a search area).

```
1  Procedure EXPAND_OR_ACCEPT(e - unsafe edge):
2      A ← ∅                    // initialize set of bad edges of the search area
3      U_A ← ∅                  // initialize set of unsafe edges around search area
4      Q ← {e}                  // unprocessed unsafe edges allowing expansion
5      // process selected surrounding unsafe edges
6      while Q is not empty do
7          f ← next edge from Q (remove it from Q)
8          // expand with the external component to which leads f
9          (C, U_C) ← (∅, ∅)
10         EXPAND_VIA_UNSAFE(f, C, U_C)
11         A = A ∪ C and if |A| > 2(Δ + 1) log(m) then FAIL
12         U_A = U_A ∪ U_C
13         // inspect new edges - look for amortizing configuration
14         if (e1) is satisfied (there is a monochromatic edge in C) then
15             return (A, U_A)
16         else if there is an unsafe edge f in U_C satisfying (e2) or (e3) then
17             EXPAND_VIA_UNSAFE(f, A, U_A)
18             return (A, U_A)
19         // select edges that may cause an activation
20         else
21             for g in U_C do
22                 if g|_a ∪ (g|_t ∩ V(C)) is monochromatic then
23                     if g \ V(C) has < αk troubled vertices then add g to Q
24     // activation exclusion
25     mark all troubled vertices in V(A) as accepted
26     return (∅, ∅)
```