Kinetic conflict-free coloring

Leijssen, T.N.P.

Award date:
2014

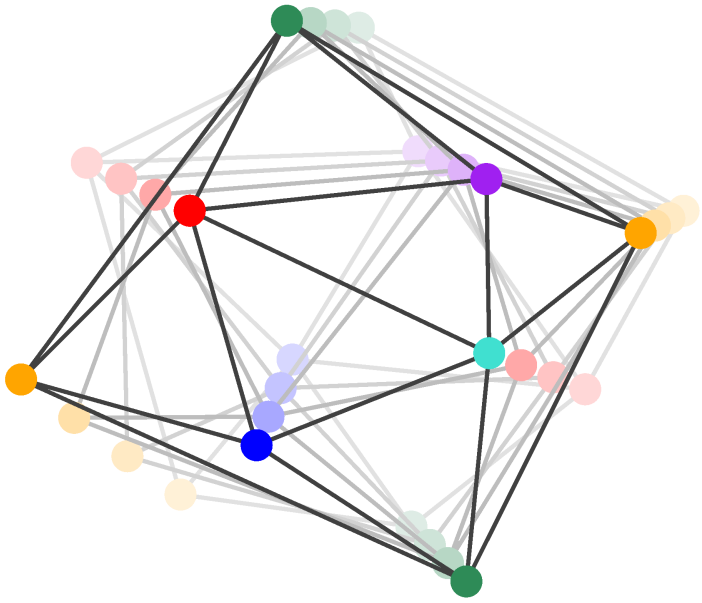
Eindhoven University of Technology

Master thesis

Kinetic Conflict-Free Coloring

Author:
Tim Leijssen

Supervisors:
Mark de Berg
Marcel Roeloffzen



Publishing date: November 24, 2014

Abstract

A *conflict-free coloring* (CF-coloring) of a set of disks D in the plane is an assignment of colors to each disk such that for every point p in the plane covered by a non-empty subset $D' \subseteq D$ of disks, we have that D' contains at least one disk with an unique color among the colors in D'. The problem arises in frequency assignment in cellular networks. *Base stations* with the same frequency might interfere, so we try to assign frequencies to each base station such that a *client* can always access a base station with a unique frequency.

In this thesis we study kinetic CF-coloring. We try to maintain a CF-coloring as the objects move, and doing so while limiting both the total number of colors used and the number of recolorings needed. We initially focus on the one-dimensional case, namely the CF-coloring of a set of moving intervals. We find a solution using the KDS model which efficiently maintains the CF-coloring using four colors and $O(1)$ recolorings per event.

We then study the problem of maintaining a CF-coloring of moving points with respect to disks, using $O(\log n)$ colors. We present two methods, both modified from a known CF-coloring algorithm for the static problem. First, the *Bounded IS algorithm* which usually has only one recoloring per event, but occasionally needs to recolor a larger subset of points. We show that this method requires an amortized $O(\log n)$ recolorings per event, and we show an example where an amortized $\Omega(\log n)$ recolorings per event is needed. Secondly, the *Maximal IS algorithm* which tries to minimize the number of colors used, at the cost of requiring more recolorings than the Bounded IS algorithm. We show a worst-case example where a single event causes an "avalanche" of recolorings.

Finally, we've run experiments to test the Bounded IS and the Maximal IS algorithm on moving point sets distributed uniformly at random. We compare the two algorithms for point sets of different sizes. By setting different parameters we find a trade-off between the total number of colors used and the number of recolorings needed.

Contents

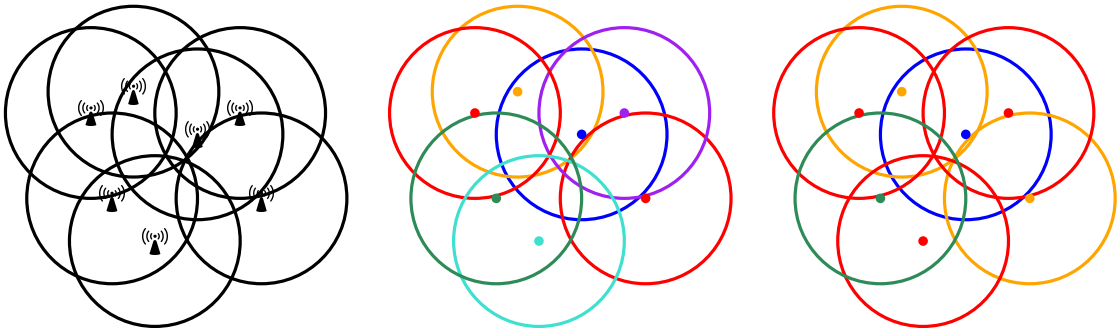


Figure 1.1: **Left**: Frequency assignment of cellular network solved as a coloring problem. **Middle**: A coloring such that no two disks of the same color overlap, using six colors. **Right**: A CF-coloring of the disks using four colors, for every point p contained in a disk there exists a disk containing p with a unique color. Note that every location where two disks of the same color overlap is contained by another disk of a unique color.

1 Introduction

1.1 Motivation

Wireless communication is used in many different situations. Typically, in order to handle interference among radio signals, different frequencies are used. However, there are usually only a small number of different frequencies available, thus one should try to minimize the number of frequencies being used. In this thesis we study so-called conflict-free colorings, which can be seen as an abstract version of certain frequency assignment problems. Before we define conflict-free colorings more formally, we first discuss two applications in which they play a role.

As a first example, consider a cellular network that consists of two types of nodes: *base stations* and *clients*. Base stations generally have a fixed frequency assigned to them, but clients can scan different frequencies to find one which is available. Of course, for a client to communicate with a base station, it has to be within its range. If two base stations close together have the same frequency, interference occurs. To be precise: if a client is within range of two base stations with the same frequency, they interfere and the client cannot communicate with either of them. If the client is within range of a third base station with a unique frequency the client will still be able to communicate with it. Typically we want that for every client within range of at least one base station, there exists at least one base station with which it can communicate without interference.

We can model this as a coloring problem. Namely, we have a set D of n disks in the plane, each representing a base station, such that the radius of the disk is the range of this base station. The different frequencies are represented by different colors, and the goal is to assign a color to every disk. The easiest thing to do would be to simply assign a different color to every disk, but this would result in a coloring of n colors. Since we want to limit the number of frequencies this isn't a very good solution. Another option is to avoid overlapping disks of the same color altogether. For example, if two disks intersect one another they can't be assigned the same color. Again this gives us a valid solution, but it is too restrictive. In the worst case we may still require n colors. We can create another valid solution by coloring the

disks in such a way that for every point p contained by D, we have that p intersects at least one disk with a unique color. This is called a *conflict-free coloring*, which we define more precisely below. See Figure 1.1 for an example.

Our second example of an application of conflict-free colorings involves RFID networks. Radio Frequency IDentification (RFID) is a technology where a *reader device* can detect close-by *reader tags*. Typically, these tags do not carry their own battery. The power needed for these tags to transmit their IDs is supplied wirelessly by the reader itself. Instead of different frequencies, each reader performs a "read" action at a different *time slot*. Namely, there is a fixed time interval T consisting of a sequence of equal-length time slots $1 \dots k$. During each of the k time slots a different subset of readers performs a "read" action. If a tag is within range of two readers which are active in the same time slot, these readers interfere, so for every tag there has to be at least one time slot in T during which there is exactly one reader that can read the tag. As in the example above, we can model the readers as a set D of n disks in the plane, and each time slot corresponding to a color. We want to minimize the total number of time slots used, this corresponds to trying to find the minimum number of colors with which we can create a conflict-free coloring of D.

In this thesis we look into the kinetic version of the problem. Namely, what happens if the base stations or RFID readers move around? We want to maintain a conflict-free coloring at all times, while limiting the total number of colors used. Since the disks move, we may need to change the coloring over time to maintain it. We assume that changing the color of one of the disks is an expensive operation so we would like to limit the number of recolorings as well.

1.2 Problem statement

Let $H = (V, \mathcal{E})$ be a hypergraph, where V is a set and $\mathcal{E}$ is a collection of non-empty subsets of V. The elements of V are called *vertices* and the elements of $\mathcal{E}$ are called *hyperedges*. An induced sub-hypergraph $H' \subseteq H$ is a hypergraph $H' = (V', \mathcal{E}')$ with vertices $V' \subseteq V$ and edges $\mathcal{E}' = \{e \cap V' | e \in \mathcal{E}\}$. That is, every edge from $\mathcal{E}$ with the vertices $V \setminus V'$ removed.

A *k-coloring* (for some natural number k) of H is a function $\phi : V \to \{1, \dots, k\}$. In other words, a k-coloring of H is an assignment of one of k colors to each of the vertices in V. Such a k-coloring of H is called *proper* or *non-monochromatic* if every hyperedge $e \in \mathcal{E}$ with $|e| \geq 2$ has two vertices $x, y \in e$ such that x and y have a different color. We use $\chi(H)$ to denote the least integer k for which H admits a proper k-coloring.

Definition 1. *A* conflict-free *coloring (often shortened to* CF-coloring*) of a hypergraph H is a coloring such that every hyperedge $e \in \mathcal{E}$ has at least one uniquely colored vertex. More precisely, there exists a vertex $x \in e$ such that for all $y \in e \setminus \{x\}$, x has a different color than y. A* unique-maximum *coloring (*UM-coloring *for short) is a further restriction on a CF-coloring where the maximum numbered color in every hyperedge is unique.*

The least integer k for which H admits a CF- or UM-coloring is denoted by $\chi_{\text{cf}}(H)$ and $\chi_{\text{um}}(H)$ respectively. Since the different colorings are restrictions of one another, every UM-coloring is a CF-coloring and every CF-coloring is a proper coloring, and we have:

$$\chi(H) \leq \chi_{\text{cf}}(H) \leq \chi_{\text{um}}(H)$$

Since the CF-coloring problem has mainly been studied as a means to assign frequencies in wireless networks, we focus on hypergraphs that naturally arise in geometry, namely:

- **Hypergraphs induced by regions:** Let $\mathcal{R}$ be a finite collection of regions in $\mathbb{R}^d$. The hypergraph induced by $\mathcal{R}$, is defined as $H(\mathcal{R}) = (\mathcal{R}, \{\mathcal{R}(p)\}_{p\in\mathbb{R}^d})$, where for a point $p \in \mathbb{R}^d$, we have $\mathcal{R}(p) = \{r \in \mathcal{R} : p \in r\}$. That is, $H(\mathcal{R})$ contains a vertex for every region $r \in \mathcal{R}$, and a hyperedge e consisting of a subset of $\mathcal{R}$ if there exists a point that is contained by all regions in e and no other regions in $\mathcal{R} \setminus e$.
- **Hypergraphs induced by points with respect to regions:** Given a set of points $P \subset \mathbb{R}^d$ and a (possibly infinite) set $\mathcal{R}$ of regions in $\mathbb{R}^d$, a hypergraph induced by P with respect to regions $\mathcal{R}$ is defined as $H_{\mathcal{R}}(P) = (P, \{P \cap r \mid r \in \mathcal{R}\})$. That is, $H_{\mathcal{R}}(P)$ has a vertex for every point, and a hyperedge between points $P' \subseteq P$ only if there exists a region r in $\mathcal{R}$ such that $r \cap P = P'$.

In this thesis we study the kinetic version of the CF-coloring problem. This means the regions or points move over time and as a result the induced hypergraph changes as well. We would like to maintain a CF-coloring at all times, so we may need to recolor the regions or points. Our goal is to minimize both the total number of colors required, and the number of recolorings.

1.3 Results

Section 3 concerns a one-dimensional version of kinetic CF-coloring, namely the CF-coloring of moving intervals. We present a solution requiring three colors for the static case and four colors for the kinetic case, using the KDS model. This method requires $O(1)$ recoloring per event, where an event happens whenever two endpoints of different intervals cross one another.

In Section 4 we look at the CF-coloring of points with respect to disks. We adapt a static CF-coloring algorithm in two different ways (the Bounded IS algorithm and the Maximal IS algorithm) to work for the kinetic case, and we analyze the number of colors used and the number of recolorings required when using these methods. For a point-set of n moving points, both methods limit the total number of colors used to $O(\log n)$. However, we show an example where the Bounded IS algorithm requires an amortized $\Omega(\log n)$ recolorings per event. For the Maximal IS algorithm we show an example where a single event causes an "avalanche" of recolorings.

We have run experiments to see how these methods perform in practice for moving point sets created uniformly at random. The results of these experiments can be found in Section 5. We try out different parameters for the algorithms used in these experiments and compare them. In these experiments we confirm the logarithmic bounds for the total number of colors for both algorithms. By setting different parameters, we find a trade-off between the total number of colors and the number of recolorings per event.

2 Previous work

CF-coloring was initially studied in 2003 by Even et al. [5] for simple geometric regions. A framework is presented for the CF-coloring of n points with respect to a set of ranges using a Delaunay graph. Specifically, for coloring n points in the plane with respect to disks (i.e. any possible disk should should have a CF-coloring of its internal points), an algorithm is presented which uses a series of Delaunay triangulations.

Note that for a point set P with n points, a CF-coloring of P with respect to disks with radius r is equivalent to a CF-coloring of a set of n disks with radius r, each centered around a

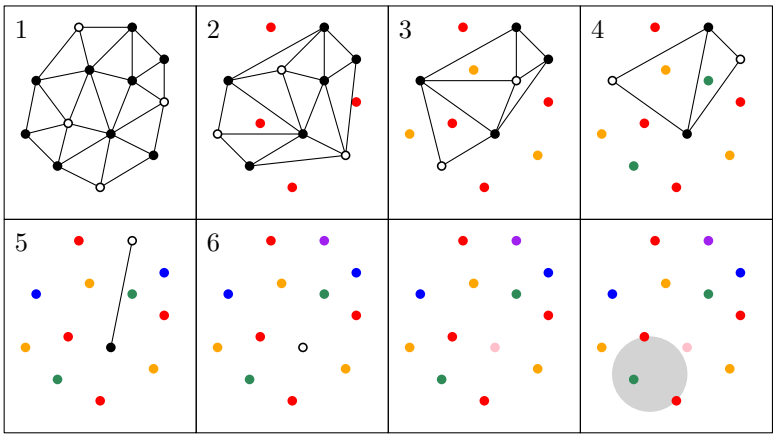


Figure 2.1: Coloring a set of points with respect to disks, using Delaunay triangulations. In the figure six iterations are shown. The white points indicate the new independent set to be colored. In the last figure a disk is shown; any disk containing at least two points of the same color contains a point with a unique color (which was colored in a later iteration).

point in P. As a result, any CF-coloring of P with respect to every disk in $\mathbb{R}^2$ is a CF-coloring of the corresponding set of disks with the same radius r, regardless of the value of r. The reason for this is that for a set of disks with radius r, a point p is contained by a disk d if and only if it is within a distance r of the center of d. This means the center of d is contained by a disk of radius r centered around p.

A sketch of the algorithm: Each iteration, we compute the Delaunay triangulation of the set of points. We compute a "sufficiently large" independent set of this graph, and color it with color i, and we remove these points from the point set. See Figure 2.1 for a visual example.

This algorithm results in a valid CF-coloring. Namely if two points are colored with the same color, they were not connected by an edge in the Delaunay triangulation, which means that any disk containing these points contains another point with a higher color.

The total number of iterations (and therefore the number of colors used) depends on the sizes of the independent sets. Fortunately, since the Delaunay triangulation is a planar graph, the average degree of the vertices is at most 6, which means we can greedily pick vertices with degree less than 6. The result is that our chosen independent set is some constant fraction of the available vertices, which means the total number of colors used is in $O(\log n)$.

There exist arrangements of n disks such that a CF-coloring of these disks requires $\Omega(\log n)$ colors. Namely for a sequence of disks of diameter d all placed with their center on a line segment of length less than d (see Figure 2.2), the hypergraph induced by these disks corresponds to the *discrete-intervals hypergraph*. This hypergraph consists of the vertices $V = \{1, \ldots, n\}$ and its hyperedges being all possible discrete ranges $[i, j]$, with $1 \leq i \leq j \leq n$. In [11] it is shown that for such hypergraphs, $\chi_{\text{cf}}(H) = \chi_{\text{um}}(H) = \lfloor \log n \rfloor + 1$. Pach and Tóth [9] proved that for points with respect to disks, any CF-coloring uses at least $c \log n$ colors for some constant $c > 0$.

In 2005 the above results were extended by Har-Peled et al. [6]. Their paper shows a probabilistic algorithm for CF-coloring any set of "simple" (but not necessarily convex)

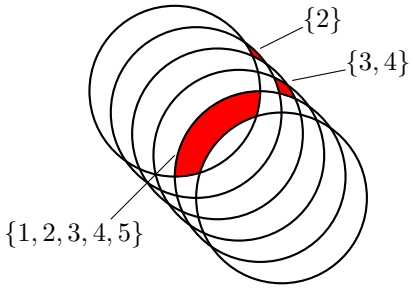


Figure 2.2: It is possible to arrange a set of n disks D such that $H(D)$ is equal to the discrete-intervals hypergraph of size n, so a CF-coloring of $H(D)$ requires $\lfloor \log n \rfloor + 1$.

regions with "low" union complexity. Results are refined for particular cases of axis-parallel rectangles ($O(\log^2 n)$ colors), and range spaces where the underlying ranges are axis-parallel rectangles (n points can be CF-colored with $O(\sqrt{n})$ colors w.r.t. axis-parallel rectangles), and more specialized cases.

Besides that, their paper also studies *k-CF-coloring*. Going back to our use case of wireless communication, it may be the case that two nearby base stations with the same wireless frequency don't interfere that much, but more than some constant k will. This corresponds to the k-CF-coloring problem. This is a more relaxed notion of CF-coloring, where for every hyperedge there must be a color that appears at least once and at most k times among the vertices of this hyperedge.

The kinetic variant of CF-coloring has not been studied yet. There has been some research into *online CF-coloring* however, where points are added one by one and a CF-coloring has to be maintained. In 2006, online CF-coloring of intervals was studied by Chen et al. [3]. More precisely: points are inserted on a line, and each newly inserted point must be assigned a color upon insertion. At all times should every interval I be conflict-free. Their paper provides various deterministic and randomized algorithms, the best deterministic algorithm using a maximum of $\Theta(\log^2 n)$ colors and the best randomized algorithm has an expected $O(\log n \log\log n)$ colors. Finally, the paper also demonstrates that in two dimensions, where the relevant ranges are disks, n colors may be required.

Bar-Noy et al. [1] present a framework for online CF-coloring any hypergraph. Using this framework an randomized online algorithm is obtained, with which any k-degenerate hypergraph can be colored using $O(k \log n)$ colors with high probability. Their paper also studies deterministic online CF-coloring with recoloring, that is, using as few colors as possible while also limiting the number of recolorings. An algorithm is provided for CF-coloring with respect to halfplanes using $O(\log n)$ colors and $O(n)$ recolorings.

In 2007, Smorodinsky [10] presented a method of CF-coloring a hypergraph H using proper colorings of H and its sub-hypergraphs. A sketch of the algorithm is as follows: Every iteration i we search for a proper coloring φ_i of H_i with "few" colors. We pick the largest color class of this φ_i and color it with color i, and remove these vertices from $H_{i+1} \subset H_i$ until there are no vertices left.

This results in a valid CF-coloring (and UM-coloring) for each hyperedge e. Namely if e contains $x > 1$ uncolored vertices during an iteration step of the algorithm, the proper coloring ensures that at most $x - 1$ of the uncolored vertices of e are colored during this

iteration step. If e contains only one uncolored vertex, then after coloring this vertex with a new color, the resulting coloring is conflict free.

The size of these largest color classes directly influences the amount of colors used in the resulting CF-coloring. Specifically, if every sub-hypergraph $H' \subset H$ satisfies $\chi(H') < k$ the algorithm produces a UM-coloring (and therefore a CF-coloring) of $\log_{1+\frac{1}{k-1}} n = O(k \log n)$ colors.

In 2009, Lev-Tov et al. [8] presents an $O(1)$ approximation algorithm for the CF-coloring of unit disks.

Other than k-CF-coloring described above, a few other variations of CF-coloring have been studied recently. For example the *k-strong CF-coloring problem* [7], namely where for each hyperedge containing at least k vertices, at least k of these vertices must have an unique color. Any set of n disks in the plane admits a k-strong CF-coloring of at most $(k \log n)$ colors. This bound also holds for pseudo-disks and regions with linear union complexity. The paper also provides bounds for other types of regions.

It might be the case that a single base station can not use every possible frequency, but can only select from a small list of frequencies. *List CF-colorings* are a specialization of the general CF-coloring problem, in this case each vertex only has a small set of at least k colors to choose from. Cheilaris et al. [2] provides bounds and an algorithm for this problem.

3 CF-coloring of kinetic intervals

Let V be a set n of intervals on the x-axis. A *conflict-free k-coloring* of V is an assignment of one of k colors to each of the intervals in V such that for any point p on the x-axis contained in one or more intervals, we have that p is contained in at least one interval from V with a unique color (that is, no other interval covering that point has the same color). We solve the problem of CF-coloring of intervals in both the static and kinetic case using *chains*.

We make the assumption that every endpoint is unique. That is, two intervals u and v cannot share an endpoint, with the exception of event points in the kinetic case. We use $l(v)$ and $r(v)$ to denote the position of the left endpoint and the right endpoint of interval v respectively.

Definition 2 (**Interval graph**). *Let V be a set of intervals. We define $G(V) = (V, E)$ as the* interval graph *of V. An interval graph is the simple graph whose vertices are the intervals in V and whose set of edges E contains (u, v) if and only if intervals u and v intersect.*

Definition 3 (**Connected component**). *Let V be a set of intervals. We define a subset $V' \subseteq V$ as* connected component *of V if the vertices from V' form a connected component in $G(V)$. That is, V' is a maximal subset such that there is a path between every two vertices $u, v \in V'$ in $G(V)$.*

For a connected component $V' \subseteq V$ it holds that for every point p that is not contained in any interval from V, we have that p is either to the left of all intervals in V' or to the right of all intervals in V'.

3.1 Static case

The main idea is as follows. If V is our set of intervals, we find a subset of V such that every point contained in an interval of V is also contained in an interval in this subset. If we

CF-color this subset with $(k-1)$ colors, we can color the remaining intervals in V with the k-th color. This results in a CF-coloring of V.

We define a *chain* as follows: A chain $C = \{c_1, \ldots, c_m\}$ is an ordered subset of a connected component of intervals V such that the following invariants hold:

INV1 For two subsequent intervals c_i and c_{i+1} in the chain, the right endpoint of c_i must be contained in c_{i+1}, and the left endpoint of c_{i+1} must be contained in c_i.

INV2 Two non-subsequent intervals in the chain (e.g. c_i, c_j such that $|i-j| > 1$) must not intersect.

INV3 The chain spans the union of the connected component V. That is, $\bigcup_{c_i \in C} c_i = \bigcup_{v \in V} v$.

INV4 The chain is the unique chain of the connected component V, in fact, each connected component has a unique chain.

INV5 An interval c_i in the chain must not be fully contained in any other interval v.

It is worth noting that INV5 is not strictly necessary in the static case, but is needed for the kinetic case.

From the above invariants, we can deduce some additional properties:

- Both the left and right endpoints of the intervals in a chain are ordered by this chain. That is, for all c_i and c_{i+1} we have $l(c_i) < l(c_{i+1})$ and $r(c_i) < r(c_{i+1})$.
- For any point p on the x-axis, p is contained in at most 2 intervals in a single chain. This follows directly from INV2 and INV4.
- Any interval v that is not part of a chain is fully covered by a chain. That is, there is no point $p \in v$ for which $p \notin \bigcup_{c_i \in C} c_i$. This follows from INV3.
- From INV4 we have that two different chains can not intersect.

3.1.1 Constructing the chains

We will sketch the algorithm used to construct the chains. We start with the interval in V with the leftmost endpoint, and make this the start c_1 of a new chain C. We then iteratively build a chain, left-to-right, as follows:

We search for a subset of intervals $S \subset V$, consisting of intervals $s \in S$ for which c_i contains the left endpoint of s but not the right endpoint of s. If S is empty, it means c_i is the last interval of the current chain, and we can continue with a new chain in the next leftmost endpoint. Otherwise, we choose the interval in S with the rightmost right endpoint and append it to the current chain as c_{i+1}.

We now prove that the algorithm fulfills all the invariants of a chain:

INV1 Since we pick the next interval in a chain such that its left endpoint is contained in c_i but its right endpoint does not, we satisfy this invariant.

INV2 Let c_h and c_j with $(j-h > 1)$ be two non-subsequent intersecting intervals in a chain. This means there exists an interval c_i in the chain such that $h < i < j$. By INV1 we have that the left endpoint of c_j intersects c_h, and we have that the $r(c_h) > r(c_i)$. This

however would mean that from c_h, the next interval chosen by the algorithm would be c_j, which is a contradiction. Therefore, using this algorithm, two non-subsequent intervals do not intersect.

INV3 Since we start each chain at the leftmost endpoint and end at the rightmost endpoint of a connected component, and because every pair of subsequent chain intervals intersects each other, the chain will span the entire connected component.

INV4 The algorithm only starts a new chain at a new component if no new interval intersects the current rightmost right endpoint, so there is one chain per connected component.

INV5 By contradiction: Let v be an interval not in a chain C which completely contains interval c_i part of C. Then (by INV3) the left endpoint of v is contained in an interval in the chain c_h with $h < i$. By INV1 we have that $r(c_h) < r(c_i) < r(v)$. This however would mean that v would have been chosen as next interval from $r(c_h)$, contradicting the statement that v is not in a chain. So we have that no interval in a chain is contained in another interval.

The algorithm can be implemented efficiently as follows: We can sort the set of intervals by their left endpoint. This means that when finding a suitable next interval for a chain, we can efficiently find the subset S_i of intervals whose left endpoints are in the current c_i. We have that the sum of all subsets S_i is $O(n)$. Each interval can only have its left endpoint in two intervals in a chain, by INV2. This means that even if we use linear search (starting from the leftmost endpoint of the current interval) to find rightmost endpoint in S_i, this results in a total running time of $O(n \log n)$.

Lemma 1. *For a set of n intervals, we can construct a series of chains in $O(n \log n)$ time.*

Using chains, we can create a CF-coloring of a set of intervals as follows using only 3 colors as follows. Let C be a chain of connected component V. We color C as follows: every even interval c_{2i} in the chain is colored blue and every odd interval c_{2i+1} is colored red. We color the remaining intervals in $V \setminus C$ green. Because of INV3, every point p which is contained in an interval from V will be contained in the chain. From INV2 follows that p is contained in a single red interval, a single blue interval, or both a red interval and a blue interval. In any of these cases p is contained in an interval with a unique color, therefore the coloring of V is conflict-free. Figure 3.1 shows an example of such a CF-coloring using three colors.

Theorem 1. *A set of intervals can be CF-colored using three colors in $O(n \log n)$ time.*

3.2 Kinetic case

We now discuss the kinetic case, where intervals move over time. We will keep a valid CF-coloring by maintaining the chains as the intervals move, and for this we use the KDS model in which we assume we know the trajectories of the intervals.

In order to preserve the CF-coloring of the chains, we have to add and remove intervals from the chains while doing as few recolorings as possible. If we use only two colors for the chains we run into trouble here, but three colors (so four colors for the total CF-coloring) is sufficient. For example, imagine an update to a chain causing it to contain two subsequent intervals c_i and c_{i+1} of the same color. If we use only two colors for the chains we might have to recolor a large part of this chain, but if we use three colors we can always resolve this

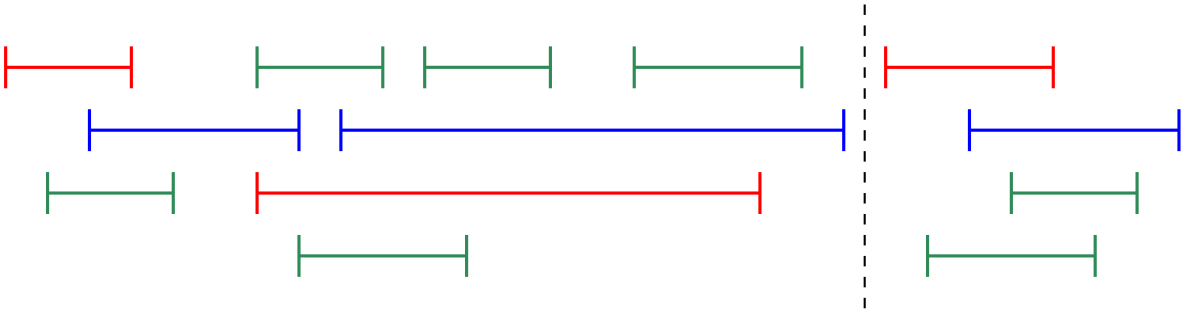


Figure 3.1: CF-coloring of a set of intervals with 3 colors, using chains. In the figure, the intervals make up two connected components (the "gap" between the two is indicated by a dashed line). The chains are colored red-blue, and the remaining intervals are colored green.

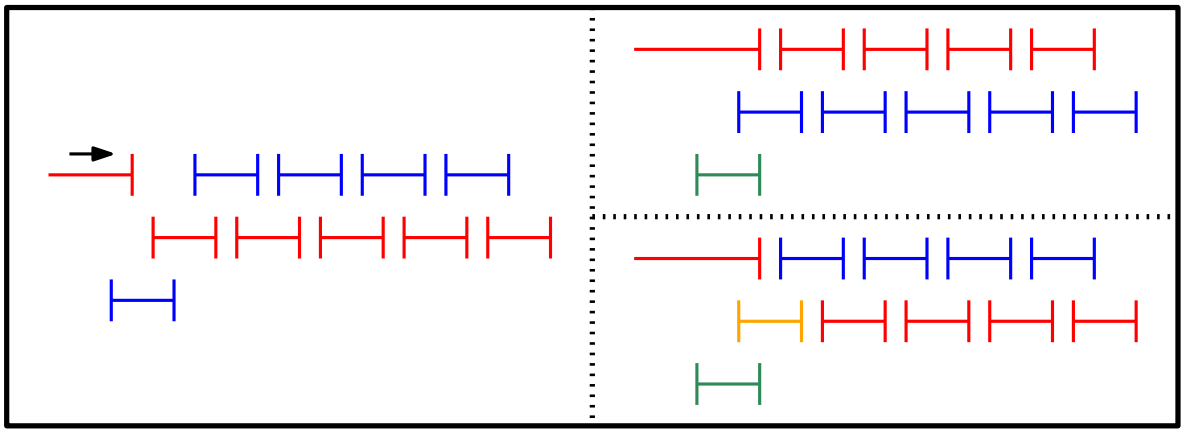


Figure 3.2: Using only two colors for a chain might cause $O(n)$ recolorings in a single event. **Left**, one of the blue intervals will be removed from the chain when the red interval moves farther to the right. **Upper right**: Using only two colors, we need to recolor the entire rest of the chain. **Lower right**: Using three colors we can always limit the number of recolorings to a constant.

conflict by coloring c_i or c_{i+1} a color different from its neighbours in the chain. See Figure 3.2 for an example. Because of this, our CF-coloring of the kinetic set of intervals uses four colors rather than three.

The general approach is as follows. We keep track of an ordered list of interval endpoints. An event happens when two endpoints cross, at which point some of the chain invariants may no longer hold. Depending on the types of endpoints which cross, we need to "fix" the chains in different ways. In section 3.2.2 we describe the different cases of endpoints crossing. In some of these cases we will need to add an interval to a chain using the AddToChain procedure we describe in section 3.2.1 below.

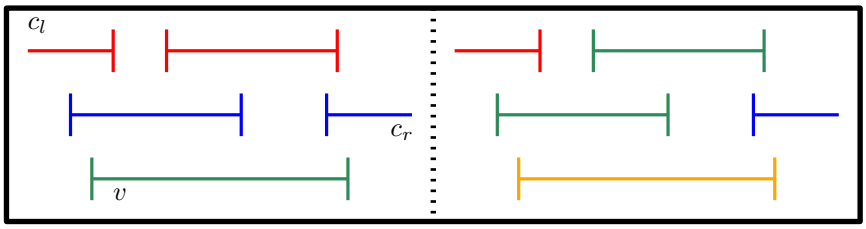


Figure 3.3: Example of the AddToChain procedure of an interval v. In the figure, the colors red, blue and yellow are used for the chain, and green is used for the remaining intervals. On the left side the state before insertion is shown, the right side shows the state after insertion.

3.2.1 The AddToChain procedure

We define a procedure AddToChain, which adds an interval $v \in V$ to a chain, possibly replacing some of the intervals already in the chain. This procedure is used to repair violations of inv3 and inv5. Calling the procedure on an interval v will result in a valid chain under the following preconditions:

- Interval v is not part of the chain.
- Interval v contains a single violation of either inv3 or inv5 (in fact, v might be the violating interval itself). That means:
  - If inv3 is violated (the chain does not fully contain all intervals), the parts of the intervals not covered by the chain are contained in v.
  - If inv5 is violated (an interval c_i in the chain is fully contained in an interval not part of the chain), v contains this interval c_i.
- If there are multiple intervals containing the violation of inv3 or inv5, then v is an interval which is not contained in any of these other intervals.

The procedure works by finding the leftmost chain interval c_l containing the left endpoint of v, and the rightmost chain interval c_r containing the right endpoint of v. We distinguish the following cases:

- If there exists no c_l (no interval containing the left endpoint of v), v becomes the leftmost interval in the chain, and we remove all intervals c_i with $i < r$ from the chain.
- Symmetrically, if there exists no c_r, then v becomes the rightmost interval in the chain and we remove all intervals c_i with $i > l$ from the chain.
- If both c_l and c_r exist, v will be placed between them in the chain, and we remove all intervals c_i with $l < i < r$.

Figure 3.3 shows an example of this procedure.

Theorem 2. *If the preconditions are true when calling* AddToChain *on an interval v, all invariants will hold afterwards.*

Proof. The AddToChain procedure affects the invariants as follows:

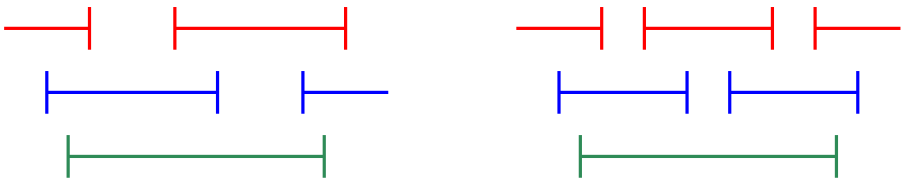


Figure 3.4: Examples of INV5. **Left**: Any interval which is not part of a chain (green) can intersect with at most four intervals in a chain. **Right**: This situation is not allowed because the green interval fully contains one of the intervals of a chain.

INV1 Simply, $l(v)$ is contained in c_l and $r(v)$ is contained in c_r, and v contains both $l(c_r)$ and $r(c_l)$, so this invariant holds.

INV2 Intervals c_l and c_r don't intersect because otherwise there wouldn't be a violation of INV3 or INV5 before calling the procedure. Since we're picking the leftmost chain interval containing $l(v)$ as c_l and the rightmost chain interval containing $r(v)$ as c_r, we have that v does not intersect any intervals in the chain other than c_l and c_r.

INV3 Interval v is now part of the chain, and it contains the previously uncovered part that violated INV3 before calling AddToChain (if any), so this invariant now holds.

INV4 Since c_l, v and c_r are all part of the same chain, the status of this invariant remains unchanged.

INV5 If INV5 was violated with chain interval c_i prior to calling the procedure, v is not contained in any other interval. If INV3 was violated before calling the procedure, interval v contains the range that violated the invariant and again v is not contained in any other interval.

So, given the preconditions, all invariants hold after calling the AddToChain procedure. □

Earlier we mentioned INV5 is only necessary for the kinetic case. We will repeat this invariant here:

INV5 An interval c_i in the chain must not be fully contained in any other interval v.

The reasoning behind this invariant is as follows. Note that above, in the AddToChain procedure, we remove all intervals between c_l and c_r from the chain. Without INV5 this could mean as many as $\Theta(n)$ recolorings after adding a single interval, but with INV5 we have that any interval v not part of a chain can intersect at most four chain intervals (see Figure 3.4). This means there can be at most two intervals between c_l and c_r, which means at most three color changes are needed when adding an interval (two for the intervals to remove, one for interval v).

3.2.2 Cases

Let the endpoints of two intervals u and v cross at position x. There are three different cases depending on which endpoints cross each other, each with subcases depending on whether or not u and v are part of a chain, and depending on whether x is contained in any other intervals. Figures 3.5 through 3.11 show examples of the following cases.

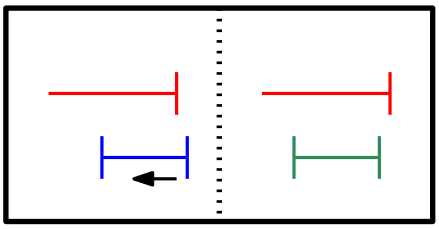


Figure 3.5: Case 1.a: Two left or two right endpoints of intervals u and v cross each other, both intervals being part of the same chain. This means v is the last (or first) interval in the chain, and can simply be removed.

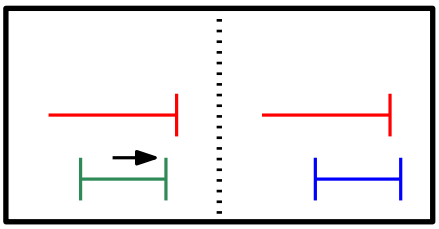


Figure 3.6: Case 1.b.i: Two left or two right endpoints of intervals u and v cross each other at position x, only v being part of a chain. If there is no other interval containing x we have that u should be appended to the chain.

1. Two left endpoints or two right endpoints cross each other at position x. We describe how to handle the case where two right endpoints cross, the case where two left endpoints cross is symmetrical. We define u and v such that after the event, interval u "reaches past" interval v. We have the following subcases:

   (a) Both intervals are part of a chain. This can happen only if v is the last interval in a chain, and u is the interval in the chain before it. This means INV1 breaks and to resolve this, interval v must be removed from this chain. It is easy to see this restores the invariants. See Figure 3.5 for an example.

   (b) Interval v is part of a chain, interval u is not. There are a few subcases here:

      i. If there is no third interval which contains x it means v is the last interval in its chain, which breaks INV3. Since u is the interval violating the invariant, we have that u is not contained by any other interval, so can add u to the chain using AddToChain. See Figure 3.6 for an example.

      ii. If there is a third interval in the chain which contains x and if u is longer than v, then u will be fully containing v (breaking INV5). Since there is no bigger interval containing v, and since before the event happened all invariants were satisfied, we have that u is not contained by any other interval, we should add u to the chain using AddToChain. For an example, see Figure 3.7.

      iii. Otherwise, we have that there is a third interval in the chain intersecting x and we have that u is shorter than v. In this case no invariants are violated.

   (c) Otherwise, we have that interval v is not part of a chain, which means no invariants can be violated.

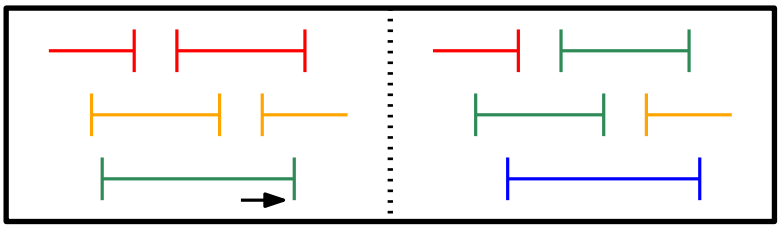


Figure 3.7: Case 1.b.ii: Two left or two right endpoints of intervals u and v cross each other at position x, only v being part of a chain. If there is a third interval containing x and if u is longer than v, then u will be fully containing v, which means we should add u to the chain.

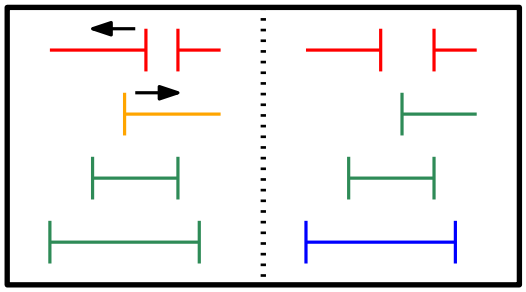


Figure 3.8: Case 2.a.i: Two endpoints cross each other, causing two intervals from a chain to stop intersecting. If there are other intervals containing x, we should seek out an interval that is not contained in any other interval and add it to the chain.

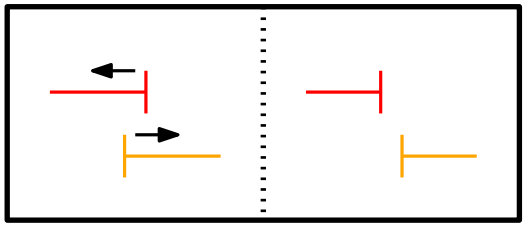


Figure 3.9: Case 2.a.ii: Two endpoints cross each other, causing two intervals from a chain to stop intersecting. If there is no other interval containing x, we split the chain.

2. A left and right endpoint cross each other at position x, causing intervals u and v to stop intersecting.

   (a) Both intervals are part of a chain. There are the following subcases:

      i. If there exists a third interval which contains x, we should find an interval v containing x that is not contained in any other interval, and add it to the chain (using AddToChain) in order to avoid breaking inv3. See Figure 3.8 for an example.

      ii. Otherwise inv1 breaks. This means the connected component splits, so we need to split the chain into two smaller chains. In this case u and v become the first and last intervals of separate chains. Clearly, no further action is needed. See Figure 3.9 for an example.

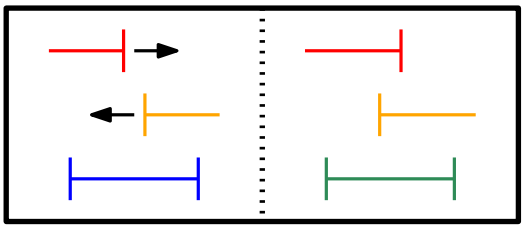


Figure 3.10: Case 3.a.i: Two endpoints from chain intervals u and v cross each other at position x, causing the two intervals to start intersecting. If there exists a third interval w in the chain which contains x, it means u and v are part of the same chain and w can be removed from the chain.

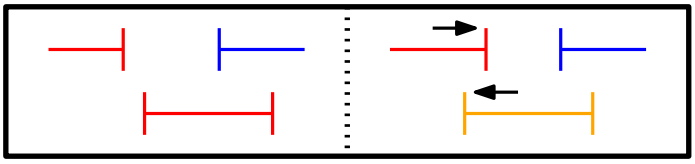


Figure 3.11: Case 3.a.ii: Two endpoints from chain intervals u and v cross each other at position x, causing the two intervals to start intersecting. If there exists no third interval which contains x, it means u and v are from different chains, and these two chains should be joined.

(b) Otherwise we have only one or none of u and v are part of a chain, so no invariants are violated.

3. A left and right endpoint cross each other at position x, causing intervals u and v to start intersecting.

   (a) Both intervals are part of a chain. There are the following subcases:

      i. If there exists a third interval w in the chain which contains x, it means INV2 breaks, so this interval w should be removed from the chain (since u and v intersect, this will restore the invariants). If u and v both have the same color, either u or v should be recolored to a color not used for its neighbours in the chain. See Figure 3.10 for an example.

      ii. Otherwise, if there exists no interval containing x it means u and v are from different chains, and INV4 breaks. This can be resolved by joining the two chains together. If u and v both have the same color, either u or v should be recolored to a color not used for its neighbours in the new chain. It is easy to see that this restores the invariant. For an example, see Figure 3.11.

   (b) Otherwise either u or v is not part of a chain, we can easily see that no invariants are violated in these cases.

3.2.3 Implementation in the KDS model

In the KDS model, we need to maintain the x-order of the endpoints. For every pair of consecutive endpoints we have a certificate, which breaks when the endpoints cross each other. An event occurs whenever a certificate breaks.

In order to efficiently maintain the chains, we need to use the following data structures:

- An array $A[1 \dots 2n]$, containing all the endpoints of the intervals ordered by their x-position. When two endpoints cross each other, we swap them in the array.
- A linked list L containing the chain intervals. This linked list is used to find chain intervals containing a given endpoint for the AddToChain function. Since this function is local to at most four chain intervals, we can follow the linked list from the interval where the event occurs.
- An *interval tree* T as described in [4], containing all chain intervals. An interval tree is an augmented red-black tree containing a node for every interval, sorted by their left endpoint. Each node also stores the rightmost right endpoint of any interval in its subtree. An interval tree supports insertion, deletion and searching (e.g. find an interval in T containing a query point x) in $O(\log n)$ time.

  In [4], intervals are static and sorted by their left endpoint. In our kinetic case we can still do this, instead of storing the endpoint positions we store the functions describing the endpoint positions over time. We need to ensure that the order of the intervals in the tree (and the rightmost right endpoint positions) remains correct as the endpoints swap, which we can do by deleting and reinserting one of the intervals. In our case, we use the tree to find the non-chain interval containing a query point x with the leftmost left endpoint, so we insert/remove intervals from T when they are removed from or added to a chain.

  Searching for the interval containing x with the leftmost left endpoint can be done as follows: From a node N storing an interval v, we always go down the left subtree if our point x is smaller than the rightmost right endpoint in the left subtree. If this is not the case, we do one of the following:

  - If x is left of v, this means there is no interval which contains x and we return nil.
  - If x lies inside v, we return v since it is the interval containing x with the leftmost left endpoint.
  - If x is right of v, we go down the right subtree of N.

Our KDS, which consists of the three data structures discussed above (the array A, linked list L and interval tree T), has the following properties:

- **Responsive:** An event simply means swapping the order of two endpoints in the data structure. This may require inserting/removing intervals from the interval tree. In addition this requires updating a constant number of certificates in the event queue, so this takes $O(\log n)$ time.
- **Compact:** It can be easily seen there are $O(n)$ certificates, namely one certificate for every adjacent pair of endpoints. The total amount of storage for the supporting data structures is also $O(n)$.
- **Local:** Each endpoint has certificates only for the endpoints directly left and right of it, so each endpoint participates in only $O(1)$ certificates.

- **Efficient:** For two large intervals, each fully containing a set of small intervals moving at exactly the same speed, there are $\Theta(n^2)$ events and only $\Theta(1)$ recolorings, so the worst case efficiency is $\Theta(n^2)$.

The following theorem summarizes the results from this section.

Theorem 3. *Let V be a set of intervals moving in $\mathbb{R}^1$. We can maintain a CF-coloring for V using only four colors with a KDS such that per event (i.e. two endpoints crossing one another), only $O(1)$ recolorings are needed.*

4 CF-coloring of kinetic points with respect to disks

In this section we will be focusing on the problem of computing the CF-coloring of a set P of n points in the plane with respect to disks. That is, for any disk in the plane the subset of points contained by that disk should be CF-colored. As mentioned in Section 2, we can solve the static version of this problem using a recursive algorithm. In iteration i of the algorithm, we are given a set V_i (initially $V_1 = V$).

- We compute a Delaunay triangulation $DT(V_i)$ of V_i.
- We find an independent set $IS_i \subseteq V_i$ of vertices in this triangulation.
- We color the vertices in IS_i with color i.
- We call the algorithm recursively on the set $V_i \setminus IS_i$ with $i := i + 1$.

We've seen that this algorithm produces a valid CF-coloring of the point set P. By sorting the vertices in V_i by their degree, we can greedily find an independent set in any planar graph of size at least $\frac{1}{6}V_i$. This means each iteration we have at most $\frac{5}{6}$th the number of vertices of the previous iteration. Because of this the number of iterations in the algorithm, and therefore the number of colors in the resulting CF-coloring is at most $\log_{\frac{6}{5}} n$.

In the remainder of this section we will be adapting this algorithm for the kinetic case. To do this we will be maintaining the Delaunay triangulation and the independent set of every iteration of the algorithm, which in turn means our coloring of the points remains a valid CF-coloring. We will be using the following terminology. We refer to the subset $V_i \subseteq V$ that are handled in the i-th iteration of the algorithm as a *layer*. We will shorten *independent set* as IS. Specifically, we use IS_i to denote the IS of the i-th layer.

4.1 Bounded IS algorithm

We can easily see that if the Delaunay triangulations of none of the layers changes, the CF-coloring remains valid. However, as the points move, eventually the triangulation might no longer be a valid Delaunay triangulation. Recall that a triangle u, v, w is a Delaunay triangle if and only if its circumcircle does not contain any other point from P. Thus when a point moves in the interior of a circumcircle of three points of a triangle, this triangle is no longer a valid Delaunay triangle. To resolve this, we need to *flip* one of the edges of the triangle to fix the triangulation. See Figure 4.1 for an example.

In this structure, we define an event as when the moving vertices cause an edge flip in the Delaunay triangulation of any layer. During an event in a layer p, two vertices in the IS of this

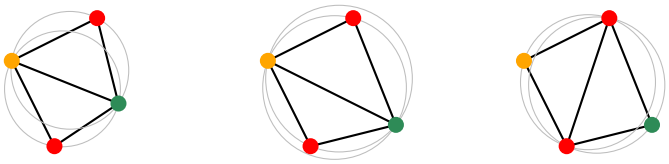


Figure 4.1: Example of an *edge flip* in a Delaunay triangulation. The red points are the IS of this layer. **Left**: The original Delaunay triangulation. Note that the circumcircles have no points in their interior. **Middle**: The green point has moved a bit to the right, causing the circumcircles to contain a point in their interiors. This is not a valid Delaunay triangulation anymore, so we need to flip an edge. **Right**: The triangulation after the edge has flipped. The new circumcircles are empty, so this is a valid Delaunay triangulation. Note however that there are two adjacent red vertices, meaning the IS of this layer is no longer valid, which means we have to recolor.

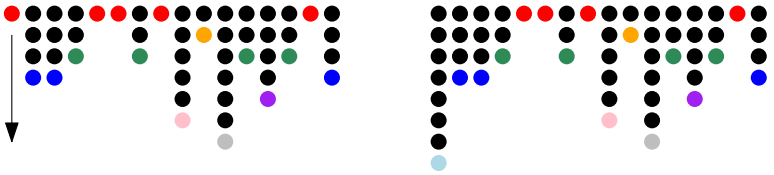


Figure 4.2: Recoloring a vertex to resolve a conflict. The i-th horizontal row of points represents the vertices which are still available in iteration i (that is, the set V_i). On the left, the leftmost red vertex has to be recolored to resolve a conflict. On the right, the result of this recoloring is shown. Note that in the second layer on the right the bound of $|V_i|/d$ has been broken for $d = 12$, so this might prompt a full recoloring from layer 2 onwards.

layer might become adjacent in $DT(V_p)$, which means the IS is no longer a valid independent set and therefore the CF-coloring is no longer valid. To rectify this, we arbitrarily pick one of the conflicting vertices and color it with a new color q, not used for any other vertex. This adds the vertex to the Delaunay triangulation to every layer between p and q (see Figure 4.2). Note that this can not invalidate the independent sets of the intermediate Delaunay triangulations, since inserting a vertex cannot create a new edge between existing vertices.

We choose two constants c and d. The constant c is chosen such that we can always find an IS for any layer i of size at least $|V_i|/c$, if we compute the IS from scratch, e.g. $c = 6$. The constant $d > c$ determines the slack we allow ourselves on the size of the IS as the points move and we may have to remove points from the IS. That is, IS_i is considered "large enough" as long as it contains more than $|V_i|/d$ vertices, where we usually take $d = 12$. If all the IS's remain within this bound, we will have a logarithmic number of colors. Specifically, the number of colors is bounded by $\log_{\frac{d}{d-1}} n$. Of course this means that for larger d we will be using more colors.

If an edge flip occurs in layer j, the two vertices connected by the new edge might both be in an independent set, so one of these vertices needs to be removed from the IS, we add the vertex to V_i for all $i > j$, and append a new layer consisting only of this vertex. Edge flips might occur in multiple layers simultaneously. When this happens we simply process the edge flips in order of their layer number, so edge flips in less deep layers are processed first.

When we "shift" a vertex to a new layer like this, it might happen that the bound of $|V_p|/d$ is broken for a layer p. When this happens, we can recolor V_p using the static algorithm, such that V_p and the deeper layers all have IS's of size at least $|V_i|/c$ again. We will call such a recoloring a *reset* of layer p. A reset of layer p might recolor every vertex in that layer, so the total number of recolorings can be as much as $|V_p|$. Informally speaking, the deeper layers are reset more frequently, but since they are smaller the total number of recolorings is not too large.

The implementation of this method is fairly simple. When an edge flip happens in a layer p and a new edge (u, v) is created, HANDLEEVENT(p, u, v) should be called.

```
procedure HANDLEEVENT(p, u, v)
    if COLOR(u) = COLOR(v) then
        q ← the current number of colors +1
        create a new layer V_q
        for i = p + 1 to q do
            add u to V_i
        end for
        set the color of u to q
        for i = p to q do
            if d · |IS_i| ≤ |V_i| then
                perform a reset from layer i
                return
            end if
        end for
    end if
end procedure
```

In the pseudocode, COLOR(v) returns the color of a vertex v. We only store implicitly whether a vertex v is in IS_i of a layer i, namely $v \in IS_i \iff$ COLOR(v) $= i$.

4.1.1 An upper bound on the average number of recolorings

For a given layer p with initial IS size $|IS_p^{init}| \geq |V_p|/c$, if every event causes a vertex to shift from IS_p to the IS of a deeper layer q, we have that every event shrinks the IS of this layer by one, while the total number of vertices in the layer remains the same. Because of this, the number of events it takes for our bound of $|IS_p| > |V_p|/d$ to be broken is at least

$$|IS_p^{init}| - |IS_p^{end}| \geq \frac{|V_p^{init}|}{c} - \frac{|V_p^{init}|}{d} \geq |V_p^{init}|\frac{d-c}{cd},$$

where $|V_p^{init}|$ is the initial size of V_p (right after a reset).

If we look at a deeper layer V_i with $p < i < q$, every event where we "shift" a vertex from IS_p to IS_q will grow the number of vertices in V_i by one, while the size of the IS_i remains the same. Therefore the number of these events it takes for the bound of $|IS_i| > |V_i|/d$ to be broken is at least

$$|V_i^{end}| - |V_i^{init}| \geq |IS_i^{init}|(d-c) \geq |V_i^{init}|\frac{d-c}{c},$$

where $|V_i^{init}|$ is the initial size of V_i. This of course only happens if a less deep layer doesn't reset before layer i does.

Informally, we can find an upper bound on the average number of recolorings as follows. When shifting a vertex to a new layer, we have that each layer in between moves a bit closer to its bound. We've seen above that the number of events it takes for a layer V_i to break its bound is at least $\Theta(|V_i|)$. Of course, the number of recolorings during a reset of a layer V_i is also $\Theta(|V_i|)$. This means the average number of recolorings per event per layer is $O(1)$, and since we have a logarithmic number of layers this means the average number of recolorings per event is $O(\log n)$.

We will prove an upper bound on the average number of recolorings per event using the accounting method [4]. Imagine a scheme where we have a wallet $W(v, i)$ for every vertex $v \in V_i$. The goal is to make sure that whenever we reset a layer V_i, the total amount of money in each of its wallets $W(v, i)$ (for all $v \in V_i$) is at least $\in 1$. We will show that this can be ensured if we spend $O(\log n)$ at each event, thus proving that we recolor $O(\log n)$ vertices per event, amortized.

To simplify the proof, we wish to maintain the following invariants:

- For every layer, vertices in the same IS have the same amount of money in each of their wallets. In other words, for any two vertices in the same IS, e.g. $u, v \in IS_l$, we have that $W(u, i) = W(v, i)$ for all i. The reason for this is to ensure that vertices in the same IS are able to go through the same resets.

- A wallet can not hold more than $\in 1$. If adding money to a wallet $W(v, i)$ would cause its contents to be higher than $\in 1$, we will set the contents to $\in 1$. We can do this because after a reset in a layer i, this layer and any deeper layers are back to their starting bounds, at which point there is no reason to have any money left in these wallets.

At each event a vertex v is shifted from a layer p to a new layer q (or in other words, v is removed from IS_p, added to all layers V_i for $i > p$ and a new layer q is created containing only v, which of course becomes the only element in IS_q as well).

When shifting a vertex v from layer p to a new layer q we spend money as follows:

1. We spend $\in 1$ to immediately recolor this vertex.

2. We add $\in \frac{cd}{|V_p|(d-c)}$ to $W(v', p)$ for each vertex $v' \in V_p$.

3. Note that when we shift v to a new layer q, then v will be added to all sets V_i with $p < i < q$. For each such layer i we create a wallet $W(v, i)$ and we put $\in X_i$ into $W(v, i)$, where X_i is the amount of money in wallet $W(v', i)$ for $v' \in V_i$. (Recall that all $v' \in V_i$ have the same amount of money in their wallets.)

4. After that, we add $\in \frac{c}{|V_i|(d-c)}$ to all wallets $W(v', i)$ for all $i > p$, $v' \in V_i$.

The first rule is trivial. As for the second rule, if all events shift a point from layer p to a deeper layer, it takes at least $\frac{|V_p|(d-c)}{cd}$ events before a reset happens in layer p, hence we should give $\in \frac{cd}{|V_p|(d-c)}$ to each vertex in V_p. Note that the total amount of money spent on the vertices is $\in \frac{cd}{d-c}$, which is a constant.

The third rule is there to ensure that the wallets for two points in the same IS hold the same amount of money. For example, if a layer i is close to resetting, and a point v is added to V_i, then $W(v, i)$ should contain enough money to recolor during the upcoming reset. Note that since the number of layers is logarithmic, and since the wallets are "capped" at $\in 1$, the amount of money we spend in this rule is bounded by $O(\log n)$.

Finally, the fourth rule. For a layer i, if each event adds a vertex to this layer, it takes at least $\frac{|V_i|(d-c)}{c}$ events for this layer before a reset is needed. To ensure the wallets for this layer contain enough money, we give $\in \frac{c}{|V_i|(d-c)}$ to each wallet in this layer. Per layer, the total amount of money spent per event is $\in \frac{c}{d-c}$, which is a constant. Since we have a logarithmic number of layers the total amount of money spent by this rule is bounded by $O(\log n)$.

Since each of the rules above spends at most $\in O(\log n)$, and since this is enough to facilitate all recolorings in the structure, we have the following theorem:

Theorem 4. *The Bounded IS method requires an average of $O(\log n)$ recolorings per event.*

4.1.2 A worst case example

We now give an illustrative example that requires $\Omega(\log n)$ recolorings per event on average. For simplicity, we use as bounds $c = 2$ and $d = 4$, so a layer i resets when its IS is smaller than (or equal to) $|V_i|/4$ and when this occurs the layer is recolored such that the IS is exactly $|V_i|/2$.

To be able to have these bounds, we will be using a raster-like structure as shown in Figure 4.3. For simplicity we assume that recolorings only happen from the first layer. We can imagine the points to move within a small distance ϵ from their starting point, causing the diagonal edges to flip, which results in violations in the independent set.

Since the raster is not infinitely large there is a boundary, where the IS's aren't structured quite as neatly and the example doesn't work. We can still focus on any square region in the interior of the raster where the bounds do hold.

Let V be a set of $n = 2^k$ vertices. This means initially, the layers have respectively $2^k, 2^{k-1}, 2^{k-2}, \ldots, 2, 1$ vertices, of which respectively $2^{k-1}, 2^{k-2}, 2^{k-3}, \ldots, 1, 1$ are in the IS's. Each step, we remove a vertex from the IS in the first layer and append it as new layer at the end. See Figure 4.4 for an example.

Lemma 2. *In this scheme, all layers (with exception of the first layer) have an IS size which is a power of two. In fact, for every i with $0 \leq i \leq k-2$ there is at least one layer with IS size 2^i.*

Proof. We prove by induction that every power of two (up to 2^{k-2}) occurs as the size of a layer. Initially, this is trivially true. Recall that in our scheme we only move vertices from the top layer down, we never move a vertex from another layer down. Because of this, when shifting vertices down the recursion, the independent sets in the existing layers (with the exception of the first layer) don't change. Only a reset can cause the IS in a layer to change.

If every layer $i > 1$ has an IS size which is a power of two, say 2^j, we have that the layer resets when $|V_i|$ equals exactly 2^{j+2}. This means the new IS of this layer has size 2^{j+1}, and the IS's of the deeper layers becomes $2^j, 2^{j-1}, 2^{j-2}, \ldots, 1, 1$, so every power of two still appears as an IS size. □

If there are multiple layers with the same IS size, the one that is the least deep reaches its bound before the other layers, and therefore these other layers won't reach their bounds. We will call each of these least deep layers a *critical layer*. Specifically, we will call the least deep layer with an IS size of 2^i the i-th *critical layer*. Note that the ordering of the critical layers is reversed compared to the ordering of the layer numbers. We have an i-th critical layer for $0 \leq i \leq k-2$.

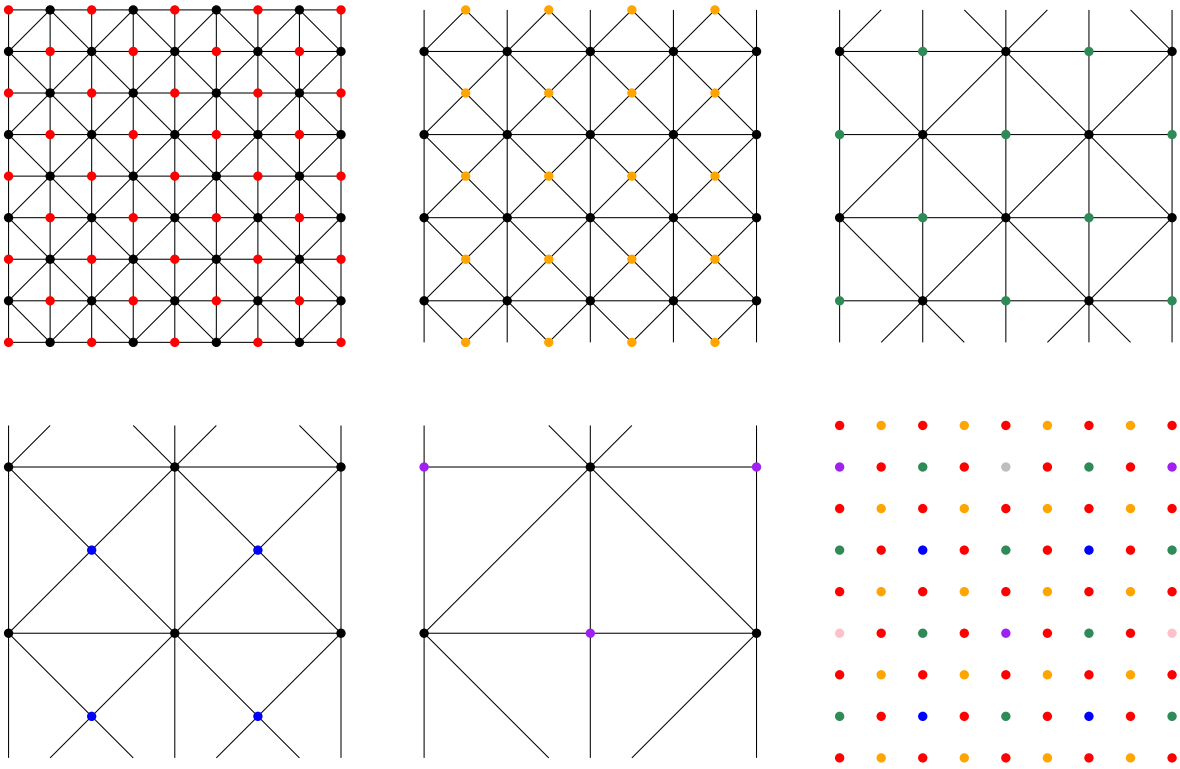


Figure 4.3: A raster of points. The first five figures show the first five layers of a CF-coloring, with the independent sets colored. The lower right figure shows the resulting CF-coloring.

Lemma 3. *The i-th critical layer resets every 2^{i+2} steps.*

Proof. Note that after it has been reset, the i-th critical layer has a total size of 2^{i+1}, and therefore it takes 2^{i+1} steps until enough vertices are added for its bounds to break. It takes the $(i+1)$th critical layer exactly twice as long for its bounds to break, and because of this we have that if the bounds of the i-th critical layer are broken, the bounds of all deeper critical layers break simultaneously.

In fact, of all the times the i-th critical layer breaks its bounds, the $(i+1)$th critical layer breaks its bounds exactly half of those times, so for each critical layer only half of the times the bounds break results in a reset in this layer. This means the i-th critical layer causes a reset every 2^{i+2} steps. □

At the moment the i-th critical layer breaks its bounds, it has 2^{i+2} vertices total, so the number of recolorings is 2^{i+2}.

Every i-th critical layer resets every 2^{i+2} steps, with $0 \leq i \leq k-4$. This means that during the 2^{k-2} steps it takes for the first layer to recolor, this critical layer resets $\frac{2^{k-2}}{2^{i+2}}$ times. This gives us a total number of recolorings due to resets of:

$$\sum_{i=0}^{k-4} \frac{2^{k-2}}{2^{i+2}} \times 2^{i+2} = (k-3) \times 2^{k-2}$$

step 0	step 1	step 2	step 3	step 4	step 5	step 6	step 7	step 8
16/32	15/32	14/32	13/32	12/32	11/32	10/32	9/32	16/32
8/16	8/17	8/18	8/19	8/20	8/21	8/22	8/23	8/16
4/8	4/9	4/10	4/11	4/12	4/13	4/14	4/15	4/8
				4/8	4/9	4/10	4/11	
2/4	2/5	2/6	2/7	2/4	2/5	2/6	2/7	2/4
		2/4	2/5			2/4	2/5	
1/2	1/3	1/2	1/3	1/2	1/3	1/2	1/3	1/2
1/1	1/2	1/1	1/2	1/1	1/2	1/1	1/2	1/1
	1/1		1/1		1/1		1/1	

Figure 4.4: Progression of the layers and their IS's in our worst case example. Each x/y pair shows the IS size compared to the total layer size, so $|IS_i|/|V_i|$. Each column represents a step where a vertex is shifted down. Layers with the same IS size are grouped together. Blue cells indicate the bound will be broken the next step, causing a reset. Green cells show the layers which were recolored in a reset.

in 2^{k-2} steps, so the average number of recolorings per step is

$$k - 3 = \Theta(k) = \Theta(\log n).$$

4.2 Maximal IS algorithm

We will now give an alternative approach to the Bounded IS algorithm described above. Again, we keep track of the Delaunay triangulations of every layer, an event happens when in any one of the layers an edge flip occurs. The difference here is that instead of waiting for resets, we try to keep the IS of a layer maximal. More precisely, we impose the following restrictions on the IS of a layer i:

- IS_i should be a valid independent set in the Delaunay triangulation of that layer.
- IS_i should not contain any vertices with a degree of 12 or greater.
- IS_i should be maximal (with exception of vertices with degree 12 or greater). IS_i is *maximal* if all vertices in $V_i \setminus IS_i$ with a degree less than 12 are neighbours to a vertex in IS_i. In other words, there are no more vertices in V_i we can add to the independent set.

For any vertex v neighbouring a vertex from IS_i, we will say v is *covered* by IS_i.

The maximal degree avoids situations where the IS of a layer consists of a few very high-degree vertices. More precisely, we have that at most half the vertices in a triangulation have a degree of 12 or greater, and for all the other lower-degree vertices we can say that choosing a vertex for the IS eliminates at most 11 other choices. This means the IS size of a layer i has a lower bound of $\frac{1}{24}|V_i|$, and as a result we have a logarithmic bound in the number of layers.

When the points move and a Delaunay edge flips, the invariants can be broken as follows:

- Two vertices in IS_i become neighbours. This means one of the vertices v should be removed from the independent set.
- A vertex $v \in V_i \setminus IS_i$ (with degree < 12) is no longer covered by IS_i. This means it can be added to IS_i.
- A vertex $v \in V_i \setminus IS_i$ not covered by IS_i loses a neighbour such that v's degree is now < 12. This means v can be added to IS_i.
- A vertex $v \in IS_i$ gains a neighbour and reaches a degree ≥ 12, which means it should be removed from IS_i.

Adding or removing a vertex to an IS will have some side effects. If a vertex v is removed from IS_p it is added to the deeper layers until we find a layer q where v can be added to IS_q without violating the invariants. On top of that, IS_p might no longer be maximal, which means one or more vertices need to be added to IS_p. If a vertex is added to an IS then the vertex is removed from all the deeper layers.

Adding or removing a vertex v to/from a layer q can have side-effects as well:

- v is added to layer q, and causes a vertex in IS_q to get a degree of 12 or greater.
- v is added to layer q, and makes it possible to add a new vertex (possibly v itself, but not necessarily) to IS_q.
- v is removed from layer q, and makes it possible to add a different vertex into IS_q.
- v is removed from layer q, and causes two vertices in IS_q to become neighbours.

Each of these effects will require IS_q to change, which in turn may cause similar effects in layers even deeper. This means one single edge flip can cause several recolorings to happen in a single event.

4.2.1 Implementation

First of all, these four procedures are used in the pseudocode below. Since they are fairly trivial, we will not elaborate on them any further.

- COLOR(v) returns the color of point v.
- DEGREE(l, v) returns the degree of point v in $DT(V_l)$.
- ISCOVERED($l, v, ignore$) returns whether v is covered by IS_l. Note that we do not check v itself, nor any vertices in the set $ignore$.
- NEIGHBOURS(l, v) returns a set of neighbours of v in the current $DT(V_l)$.

The following pseudocode illustrates the implementation of processing an edge flip. After flipping an edge in layer l we should call VERTEXCHANGE(l,v) on all vertices v involved in this edge flip. For a single edge flip this would be four vertices, but in some cases multiple edge flips might happen at the same moment in time. This procedure will check if the vertex v violates the restrictions of the maximal independent set in layer l and fix this violation.

```
procedure VERTEXCHANGE(l, v)
    if COLOR(v) ≠ l then
        CHECKADDIS(l, v, ∅)
    else if DEGREE(l, v) ≥ 12 or ISCOVERED(l, v, ∅) then
        REMOVEFROMIS(l, v)
    end if
end procedure
```

The CHECKADDIS procedure is called on layer l and vertex v to add v to IS_l if possible. In addition, CHECKADDIS takes an *ignore* parameter which contains a set of vertices to ignore when checking whether v is covered. Doing this requires v to be removed from deeper layers.

```
procedure CHECKADDIS(l, v, ignore)
    if not ISCOVERED(l, v, ignore) and DEGREE(l, v) < 12 then
        for i = COLOR(v) downto l + 1 do
            REMOVEFROMLAYER(i, v)
        end for
        set the color of v to l
    end if
end procedure
```

The REMOVEFROMIS procedure is called on a layer l and vertex v to remove v from IS_l. When this happens, we need to see if the neighbours of v can be added to IS_l, and finally add v to the next layer.

```
procedure REMOVEFROMIS(l, v)
    for all n ∈ NEIGHBOURS(l, v) do
        CHECKADDIS(l, n, {v})
    end for
    FITINIS(l + 1, v)
end procedure
```

The FITINIS procedure is called on a layer l and vertex v. This procedure adds v to V_l and tries to fit v in IS_l. If this is not possible, it recursively calls itself to fit v in the next layer, and afterwards handles the changes that adding v to V_l have caused in the Delaunay triangulation.

```
procedure FITINIS(l, v)
    add v to V_l
    if ISCOVERED(l, v, ∅) or DEGREE(l, v) ≥ 12 then
        FITINIS(l + 1, v)
        for all n ∈ NEIGHBOURS(l, v) do
            if COLOR(n) = l and DEGREE(l, n) ≥ 12 then
                REMOVEFROMIS(l, n)
            else if COLOR(n) ≠ l then
                CHECKADDIS(l, n, ∅)
            end if
        end for
    else
        set the color of v to l
    end if
end procedure
```

The REMOVEFROMLAYER procedure removes a vertex v form a layer l and fixes any voilations of the IS which may occur. In particular, if a neighour n of v has to be removed from IS_l as a result of removing v, then we need to check for every neighbour m of n if they can be added to IS_l instead. To avoid conflict we temporarily set the color of n to an invalid value.

```
procedure REMOVEFROMLAYER(l, v)
    nb_v ← NEIGHBOURS(l, v)
    remove v from V_l
    for all n ∈ nb_v do
        if COLOR(n) ≠ l then
            CHECKADDIS(l, n)
        else if ISCOVERED(l, n, ∅) or DEGREE(l, n) ≥ 12 then
            for all m ∈ NEIGHBOURS(l, n) do
                CHECKADDIS(l, m, {n})
            end for
            set the color of v to −1
            FITINIS(l + 1, n)
        end if
    end for
end procedure
```

4.2.2 A worst case example

A theoretical example of things escalating is the following: A vertex is removed from a layer V_i, which causes the addition of two new vertices to IS_i. These two vertices were originally in IS_{i+1}, but since they are now removed in V_{i+1} these removals will cause twice as many new vertices to be added to IS_{i+1}, which removes them from V_{i+2} (and IS_{i+2}), etc.

This example will double the number of recolorings each layer, so if this keeps up for l layers, the total number of recolorings in a single event is $O(2^l)$. However, the question is: is

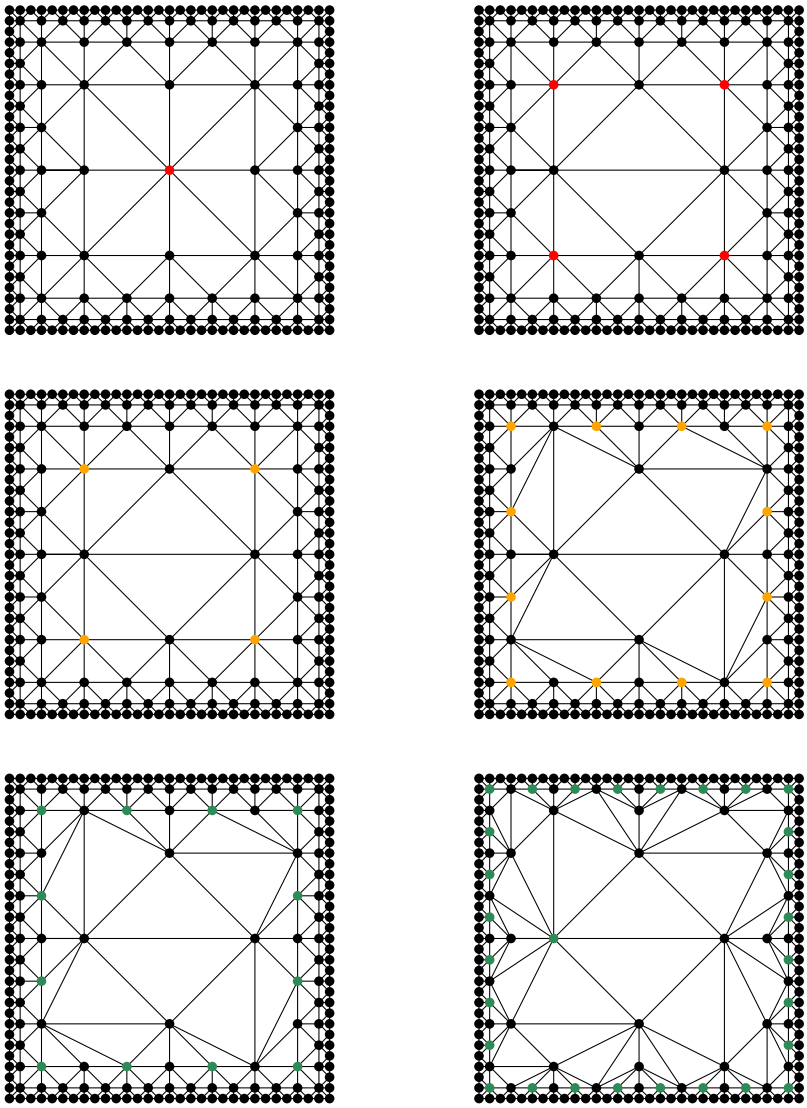


Figure 4.5: A set of points arranged in squares. On the left, three layers are shown, on the right the same layers are shown after removal of the center vertex. Each next layer, the number of recolorings doubles. Not shown are "padding" vertices close to the vertices in the outer squares to ensure the IS's are maximal.

such an example geometrically possible?

For an actual example of something like this happening, see Figure 4.5. The figure shows a set of points arranged on l squares centered around a single center point. The i-th square has dimensions $1 - \frac{1}{2^i}$, and has a point on each corner. The edges of the square consist of points spaced $\frac{1}{2^{i+1}}$ apart. For $i \geq 1$, we call the subset of points on the i-th square S_i. Note that if S_i has m points, S_{i+1} will have $2m + 8$ points, so the number of vertices doubles each square.

The center point is in the first IS and for all i in $\{1 \dots l\}$, exactly half of the points of the S_i are in IS_{i+1}. The other half of the points on each square are in deeper independent sets

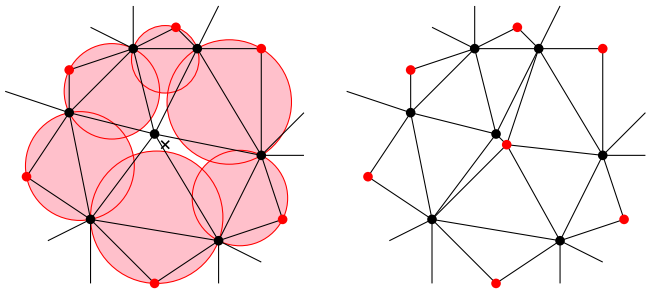


Figure 4.6: For a point set in general position (no 4 points lie on the same circumcircle), we can always add a new IS vertex to a Delaunay triangulation to cover a yet uncovered vertex v.

and are unimportant for this example.

The "escalation" is as follows: if we remove the center point from IS_1, we have to add 4 vertices from S_1 to IS_1. If these 4 vertices were originally in IS_2 their removal from this layer prompts 12 vertices from S_2 to be added to IS_2. If we again pick the 12 vertices which were originally in IS_3, which removes them from V_3, etc. In other words, if we "badly" choose new vertices for each IS, then we can cause twice the number of changes in every next layer.

We have however ignored a problem with this example, namely that our independent sets aren't maximal. If we add vertices in the outer squares to an IS such that it is maximal, our scheme no longer works out, so we have to do something different. For each layer i we add a set X_i of padding vertices to V_i, which are become become part of IS_i such that this independent set becomes maximal.

Lemma 4. *Given a set of vertices V, of which a subset $C \subseteq V$ forms an independent set in the Delaunay triangulation of V, it is possible to add a set of vertices X such that $X \cup C$ is a maximal independent set of the Delaunay triangulation of $X \cup V$.*

Proof. We can solve this problem greedily, by adding the vertices of X one by one to the triangulation until $X \cup C$ is a maximal independent set. Let C be the current independent set in V. If C maximal, we are done. However if C is not maximal, this means there are vertices in $V \setminus C$ which are not neighbouring any vertices in C. In other words, there are vertices in V still not *covered* by C. Let v be such an uncovered vertex. We try to add a vertex c which covers v. Of course we should ensure that c does not become a neighbour to any IS vertex in the new Delaunay triangulation.

Let S be the set of circumspheres through the vertices in C, that is, S contains a circumsphere of a Delaunay triangle if and only if one of the vertices is in C. If we add c inside any of the circumspheres of S, it will become a neighbour of a vertex in C. However since the vertex v is uncovered by the independent set, and since our points are in general position, there is always some amount of space around v not intersecting any of the spheres in S. See Figure 4.6 for an example. This means we can place our vertex c inside this space, close to v, which results in v being covered. □

Vertices with degree of 12 or greater of course do not need to be covered by the independent set. However, padding vertices should not have a degree of 12 or greater either. A padding

vertex c covering v can be neighbour only to neighbours of v and v itself, so by placing c opposite to some neighbour of v we can guarantee c is not a neighbour to all neighbours of v. This means the degree of c is less than 12.

This means our total construction consists of the center vertex, the square vertices S_i and the padding vertices X_i for $1 \leq i \leq l$. Each independent set IS_i consists of exactly half the vertices in S_{i+1}, the padding vertices X_i and possibly some leftover vertices in the earlier squares S_j for $j < i$. With this construction it is possible that removing the center vertex will cause an "avalanche" effect, doubling the number of recolorings in each layer until the l-th one.

5 Experimental results

In the previous sections, we have determined a few theoretical bounds for the number of recolorings and the number of colors. For the Bounded IS algorithm with bound d, each layer has at most $\frac{d-1}{d}$ times the number of vertices of the previous layer, which means the number of colors is at most $\log_{\frac{d}{d-1}} n$. We also have argued that this algorithm requires an amortized $O(\log n)$ recolorings per event. For the Maximal IS algorithm we've seen that a single event can cause an "avalanche" effect, namely in every next layer twice as many vertices need to be recolored. In this section we experimentally investigate the number of recolorings and the number of colors used by these two algorithms.

Unless otherwise indicated, we use the following settings when running the experiments:

- Our point-set consists of 100 points, placed uniformly at random in a domain D which is an axis-aligned square. Each point has a velocity vector chosen uniformly at random from a unit square.
- When a point intersects a boundary of the domain D its trajectory in the x- or y-direction is reversed. The result of this is that the point distribution will stay uniformly at random.
- Our tests end after processing 10000 events.
- For the Bounded IS algorithm, the bound d after which a reset happens is set to $d = 12$.
- For the Bounded IS algorithm, we resolve conflicts (neighbouring points in the IS, making the IS invalid) by shifting one of the points to a new layer (as opposed to the first layer where the point would fit in the IS).
- For the Maximal IS algorithm, the maximal degree d is set to $d = 12$, so a point in an IS can not have a degree of 12 or greater.

These are the default settings of our experiments. We typically use the default settings for all variables except one, which we vary in the experiment. Thus we investigate the dependency of the quality of our solution on the following parameters.

- The number of points for both algorithms.
- The bound d in the Bounded IS algorithm.
- The shifting method in the Bounded IS algorithm.

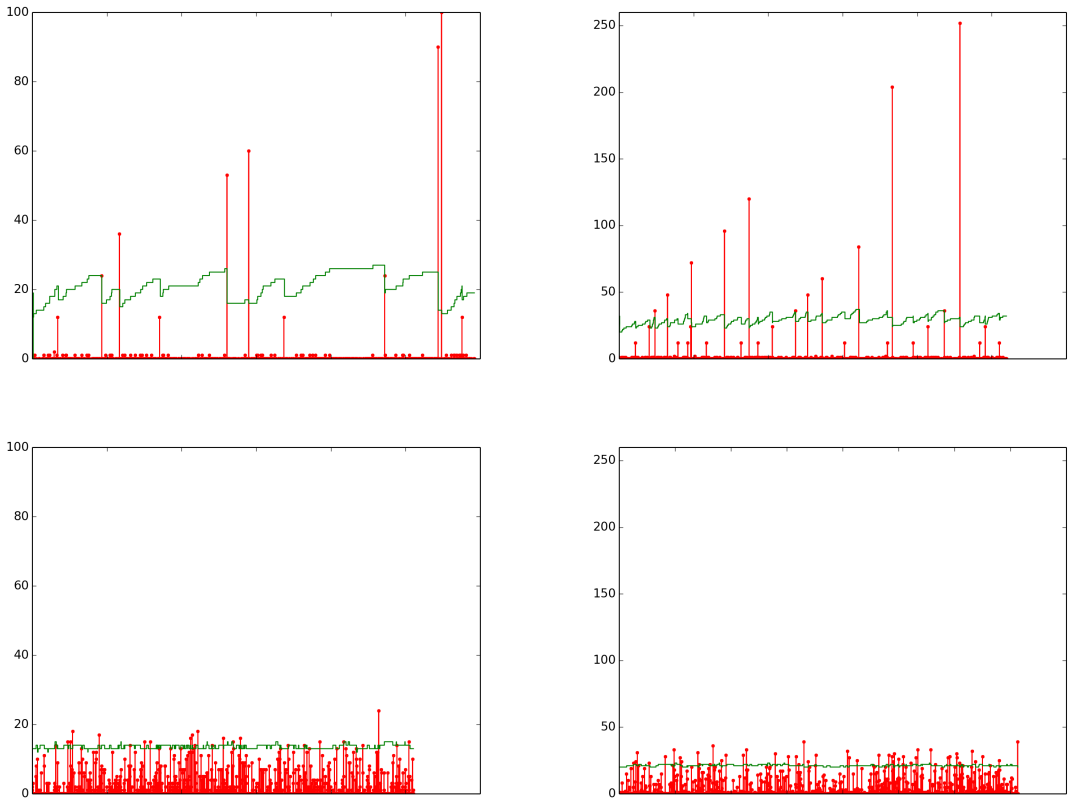


Figure 5.1: Examples of the number of recolorings (red) and the number of colors (green) during 2500 events. **Upper left:** Bounded IS algorithm with 100 points. **Upper right:** Bounded IS algorithm with 1000 points. In both the Bounded IS graphs the reset size appears to ascend until the reset of the first layer. **Lower left:** Maximal IS algorithm with 100 points. **Lower right:** Maximal IS algorithm with 1000 points.

- The maximal degree in the Maximal IS algorithm.
- Whether or not the x- or y-direction of a point is reversed when it collides with the boundary of the domain D.

We look at the effect changing these variables has on the number of colors and the number of recolorings.

5.1 Varying the point-set sizes

Our first experiments are to analyze the effect of different point-set sizes. We've run experiments for point-sets ranging from 20 to 2000 points, for both the Bounded IS and the Maximal IS algorithm. For the Bounded IS algorithm, since we use a constant of $d = 12$, each layer is at most $\frac{11}{12}$th of the previous one in size, the upper bound of the number of colors is given by $\log_{\frac{12}{11}} n$. Similarily, for the Maximal IS algorithm the upper bound of the number of

	Colors					Recolorings	
n	avg	max	bound	$\frac{avg}{\log n}$	$\frac{max}{\log n}$	total	max
20	12.4	15	34.4	2.9	3.5	1232	20
50	17.2	22	45.0	3.0	3.8	1569	50
100	20.9	28	52.9	3.1	4.2	2017	100
200	24.6	31	60.9	3.2	4.1	2275	200
500	28.3	35	72.4	3.2	3.9	2879	337
1000	31.3	38	79.4	3.1	3.8	4064	444
2000	34.0	42	87.4	3.1	3.8	5659	696

Table 5.1: Varying the number of points for the Bounded IS algorithm. For the number of colors, **avg** is the average number of colors per event, **max** is the maximal number of colors that occurred during the experiment, and **bound** is the theoretical upper bound. For the number of recolorings **total** is the total number of recolorings during the experiment, and **max** is the highest number of recolorings in a single event.

	Colors					Recolorings	
n	avg	max	bound	$\frac{avg}{\log n}$	$\frac{max}{\log n}$	total	max
20	8.2	10	69.3	1.9	2.3	6414	12
50	11.2	14	91.9	2.0	2.5	8036	19
100	13.5	16	108.2	2.0	2.4	9834	24
200	15.9	18	124.5	2.1	2.4	12034	27
500	19.1	21	146.0	2.1	2.3	15248	34
1000	21.3	23	162.3	2.1	2.3	18406	62
2000	23.5	25	178.6	2.1	2.3	22164	60

Table 5.2: Varying the number of points for the Maximal IS algorithm.

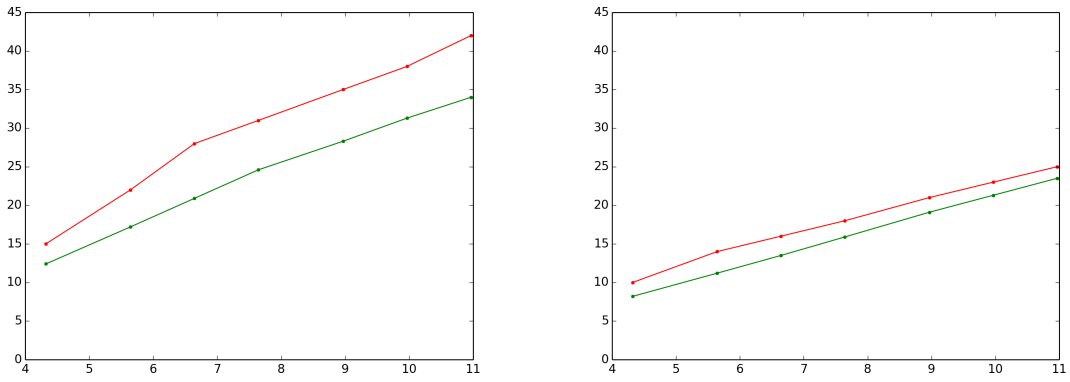


Figure 5.2: The average (green) and maximal (red) number of colors for different point-set sizes. Left the Bounded IS algorithm is shown, right the Maximal IS algorithm. The x-axis is the logarithm of the number of points.

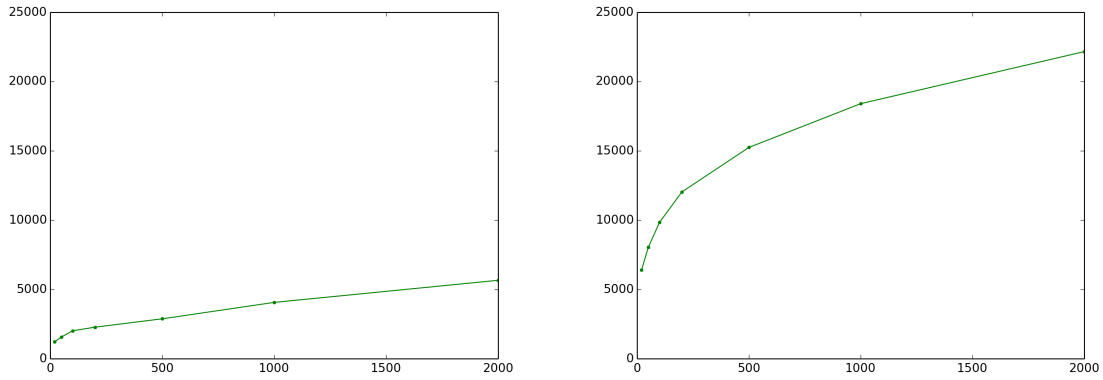


Figure 5.3: The total number of recolorings plotted against the number of points. Left the Bounded IS algorithm is shown, right the Maximal IS algorithm.

colors is given by $\log_{\frac{24}{23}} n$. In both cases we will likely never get close to these bounds since they require the IS fractions of every layer to be close to $|V|/12$ or $|V|/24$ respectively.

Figure 5.1 shows some examples of the progression of the number of colors and number of recolorings during these experiments, and Tables 5.1 and 5.2 show the results of these experiments.

One interesting pattern in the figures is the "staircase effect" that appears during the Bounded IS algorithm. Namely, the recoloring spikes seem to get bigger and bigger towards the end of the graph. In fact, for two subsequent layers V_i and V_{i+1}, we have that V_{i+1} tends to reset shortly before V_i resets. The reason that this occurs is that V_{i+1} is "almost" the size of V_i, so its bounds break shortly before V_i's bounds break. This means we expect a slightly smaller spike shortly before a large spike, which gives us these increasing reset sizes. After a reset of the first layer we expect the spikes to start small again.

It is of course a bit wasteful to have a layer V_{i+1} recolor shortly before V_i, since it means most of the points in V_i are recolored twice in a row, and if V_i would have broken its bounds before V_{i+1} then V_{i+1} would not have to recolor again for a while.

Figure 5.2 shows the average and maximum number of colors plotted against $\log n$. The linear behaviour of the graphs indicates indeed that in practice, the number of colors is logarithmic in the number of points. Figure 5.3 shows the number of recolorings plotted against n.

Comparing the two algorithms, we can see that although the Maximal IS algorithm uses less colors than the Bounded IS algorithm, the Maximal IS algorithm requires many more recolorings. The Bounded IS algorithm has some events with a very large number of recolorings (occasionally the entire point-set), though most events only cause 1 recoloring or less. The Maximal IS algorithm has many recolorings of smaller size, but recolorings where only a single point is recolored are rare. Around 70% of the recolorings here have a size greater than one.

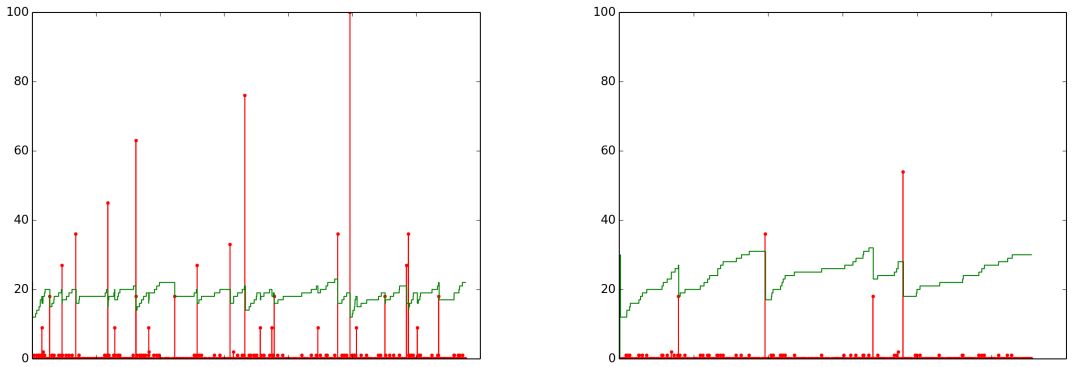


Figure 5.4: Examples of the number of recolorings (red) and the number of colors (green) during 2500 events for the Bounded IS algorithm. In the left graph we use a bound of $d = 9$, in the right graph $d = 18$. As expected, the number of recolorings for higher d is lower, but the average number of colors is higher.

	Colors			
d	avg	max	bound	Recolorings
7	16.2	19	29.9	6960
8	17.4	21	32.5	4292
9	18.2	24	39.1	3899
10	19.1	24	42.7	2979
11	19.8	25	48.3	2819
12	20.6	27	52.9	1986
14	22.8	29	62.1	1609
16	23.8	33	71.4	1289
18	27.2	35	80.6	731
20	27.6	36	89.8	680
22	28.6	37	99.0	477
24	30.9	41	108.2	458

Table 5.3: Varying the bound d for the Bounded IS algorithm.

5.2 Bounded IS: Varying the reset bound

In the previous sections we've seen that the upper bound of the number of colors when using the Bounded IS algorithm is given by $\log_{\frac{d}{d-1}} n$. In practice, we expect the number of colors to be lower, since this upper bound only occurs if all layers have an IS size close to $|V_i|/d$.

For higher values of d, the IS of a layer needs to be a smaller fraction of the layer vertices for a full reset to take place, so we expect the number of recolorings to decrease. Similarily, for lower d resets take place earlier so the number of recolorings is greater.

Figure 5.4 shows examples of the algorithm running with different bounds. As expected, a higher value of d results in less frequent resets and a higher number of colors.

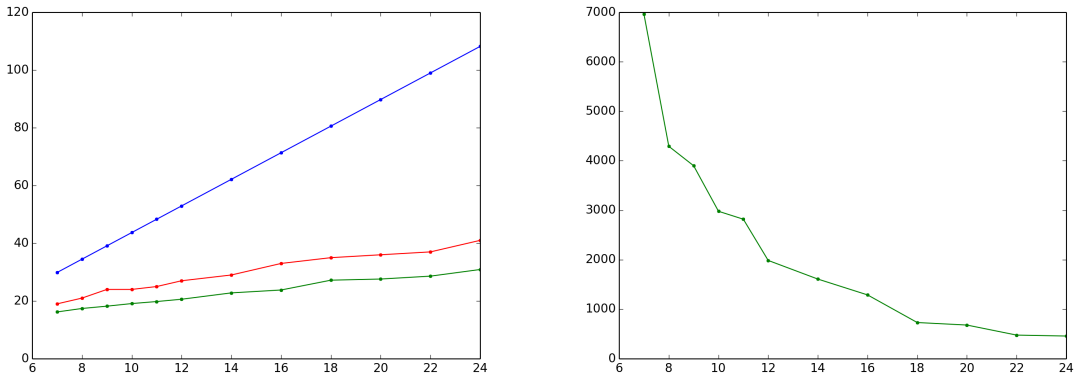


Figure 5.5: Experimental results when varying the reset bound d for the Bounded IS algorithm. **Left:** The average (green) and maximal (red) number of colors plotted against d. The blue points indicate the theoretical upper bound of $\log_{\frac{d}{d-1}} n$. **Right:** The number of recolorings plotted against d.

	Colors		
Shift to	avg	max	Recolorings
new layer	21.7	27	1699
first fit	19.5	23	998

Table 5.4: Comparing shifting to a new layer to shifting to the first fitting layer.

In Table 5.3 and Figure 5.5 we show the results of these experiments. The number of colors stays far below the calculated upper bound. The number of recolorings show a pattern converging to 0. After all, for higher values of d resets will happen less frequently.

It is worth noting that the Maximal IS algorithm has a lower average and maximum number of layers for 100 points, even compared to the Bounded IS with $d = 7$. As a trade-off, the number of recolorings is higher than that of the Bounded IS with $d = 7$. This is because while a maximal independent subset of a layer i can in theory be smaller than $V_i/7$, this rarely happens for uniformly distributed points.

5.3 Bounded IS: Shifting methods

Our next experiments involve the shifting methods for the Bounded IS algorithm. In our other experiments we've always shifted points to a new layer, but it is also possible to shift a point to the first layer where it fits in the IS without causing a violation. This should improve the number of colors, since we don't create a new layer for every shift. The number of recolorings should also improve since adding a point to the IS of an existing layer will delay a reset taking place from that layer.

Table 5.4 and Figure 5.6 shows the results of these experiments. These results indicate that the number of colors for the "first fit" shifting method is slightly less than the number of colors for the "new layer" method. The number of recolorings is significantly less for the

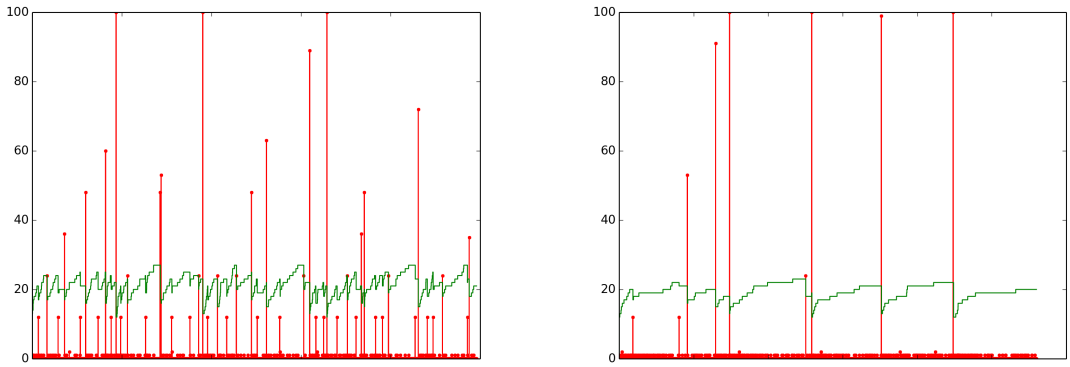


Figure 5.6: The number of colors (green) and recolorings (red) for the Bounded IS algorithm when shifting to a new layer (left) and shifting to the first fitting layer (right). Interestingly, when shifting to the first fitting layer, resets of layers other than the first one seems much less common.

	Colors			Recolorings	
d	avg	max	bound	total	max
7	13.3	15	223.3	21450	31
8	13.4	15	145.1	13153	28
9	13.5	15	122.0	10796	26
10	13.6	16	112.8	10110	25
11	13.6	16	109.1	9360	27
12	13.5	16	108.2	10066	25
14	13.6	16	110.5	9517	23
16	13.6	16	115.6	9852	24
18	13.6	16	122.0	9863	22
20	13.5	16	129.3	9973	23
22	13.6	16	137.0	9356	22
24	13.6	15	145.1	9713	24

Table 5.5: Varying the maximal degree d for the Maximal IS algorithm.

"first fit" shifting method. It seems resets from layers other than the first layer are much less common, which can be explained by the fact that vertices are often added to the IS's of existing layers other than the first one, which means they don't reset as often.

5.4 Maximal IS: Varying the maximal degree

For the Maximal IS algorithm we use a maximal degree denoted as d in order to limit the number of colors. A lower maximal degree means that fewer vertices can be chosen to be in the IS of a layer, whereas a higher maximal degree means that high-degree vertices can be chosen for the IS which covers many other points at once.

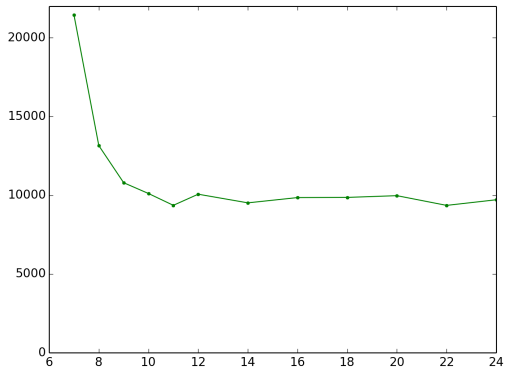


Figure 5.7: The number of recolorings plotted against the maximal degree d for the Maximal IS algorithm.

		Colors		Recolorings		
Algorithm	bound	avg	max	total	max	events
Bounded IS	enabled	21.2	27	1761	100	10000
Bounded IS	disabled	21.0	27	1239	100	5640
Maximal IS	enabled	13.5	16	9987	22	10000
Maximal IS	disabled	13.7	15	3348	20	3139

Table 5.6: Experimental results for both algorithms, with the bounded domain disabled and enabled.

Since vertices reaching this maximal degree may need to be recolored, we expect the number of recolorings to be higher for lower values of d. However since high degrees are fairly rare, especially degrees greater than 12, the number of recolorings for $d > 12$ should not differ too much from $d = 12$.

Table 5.5 and Figure 5.7 show the results of the experiments. As expected, low values of d increase the number of recolorings, but for $d \geq 10$ there are too few vertices with such high degrees to make a significant difference. Interestingly, the number of colors appears to be unaffected by the different values of d. This indicates that for point trajectories chosen uniformly at random, having a maximal degree d at all is unnecessary.

5.5 Bounded domain

In our previous experiments we have used a bounded domain, points which hit the edge of the domain reverse their horizontal or vertical speed. This way we could continue running the experiment indefinitely. Without this boundary the points will move further apart and events will start happening less frequently as time progresses.

Figure 5.8 shows examples of both algorithms running without bounded domain. In Table 5.6 the results of the experiments are shown. It seems that for both algorithms the number of colors is not significantly affected. Also, for both algorithms the average number of recolorings

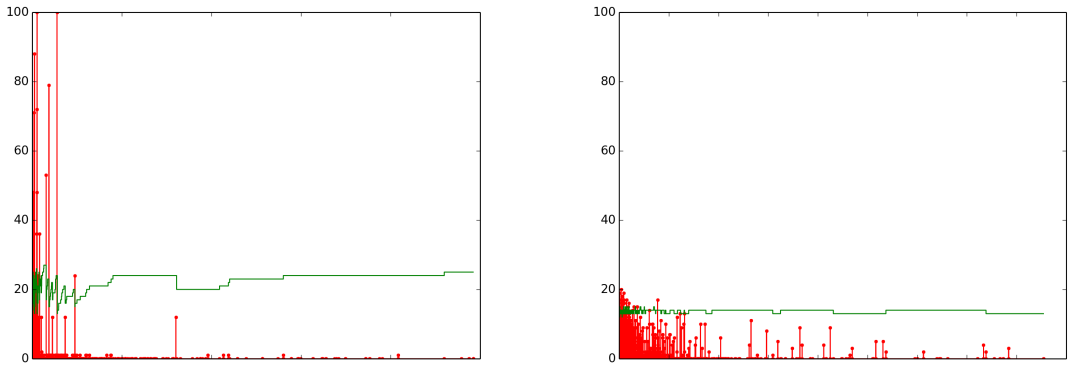


Figure 5.8: The number of colors (green) and recolorings (red) without bounded domain. Left the Bounded IS algorithm is shown, right the Maximam IS algorithm.

per event does not seem to differ much depending on the whether we bound the domain.

6 Conclusions

For the one-dimensional case we have shown a method to find a CF-coloring of intervals using structures called chains, which results in a coloring using three colors. Using the KDS model, we have shown how to maintain these structures when the intervals move which results in a CF-coloring using four colors with $O(1)$ recolorings per event.

In two dimensions we've looked into maintaining a CF-coloring of points with respect to disks, or equivalently the CF-coloring of unit disks. We've mainly looked into adapting the algorithm found in [5] for the kinetic case. We've looked into two different ways of doing this, summarized as the *Bounded IS* and the *Maximal IS* algorithms, both resulting in a $O(\log n)$ number of colors.

For the Bounded IS algorithm, we have shown that the algorithm requires an amortized $O(\log n)$ recolorings per event, and we've seen an example where an average of $\Omega(\log n)$ recolorings per event are required. For the Maximal IS algorithm no clear bounds were found, but we did show an example where the number of recolorings doubles each layer, causing an "avalanche" of recolorings.

We have run experiments of both the Bounded IS and Maximal IS algorithm on point sets chosen uniformly at random from inside a square. We have compared the number of recolorings and the total number of colors during different experiments, varying the number of points and some of the parameters of both the Bounded IS and Maximal IS algorithm.

In these experiments we've confirmed the logarithmic behaviour of the number of colors, though the total number of colors generally does not come close to the theoretical upper bound. For the Bounded IS algorithm we've shown the trade-off between the number of recolorings and the total number of colors when using different bounds. We've also demonstrated that coloring a vertex with the first available color gives quite an improvement over coloring a vertex with the first unused color. For the Maximal IS algorithm we've looked at the effects when changing the maximal degree parameter. It turns out that this parameter only matters

when it is set to a low value, e.g. 10 or lower. This stems from the fact that high-degree vertices are simply not that common when the points are distributed uniformly at random. Finally, we've taken a brief look at the effects of having a bounded domain compared to not having one.

There is still plenty of further research to be done in Kinetic CF-coloring. In this thesis we've only considered CF-coloring of kinetic intervals and CF-coloring of points with respect to disks. For example, we haven't looked into CF-colorings of rectangles or other types of regions. The most generalized version of Kinetic CF-coloring would be maintaining a CF-coloring of a hypergraph as we insert and remove hyperedges. Other than that, there are a few variations of CF-coloring which do not have any kinetic solutions yet, such as k-CF coloring, k-strong CF-coloring, and List CF-coloring.

References

[1] A. Bar-Noy, P. Cheilaris, S. Olonetsky, and S. Smorodinsky. *Online conflict-free colorings for hypergraphs.* 2007.

[2] P. Cheilaris, S. Smorodinsky, and M. Sulovskỳ. The potential to improve the choice: list conflict-free coloring for geometric hypergraphs. In *Proceedings of the 27th Annual ACM Symposium on Computational Geometry*, pages 424–432, 2011.

[3] K. Chen, A. Fiat, H. Kaplan, M. Levy, J. Matoušek, E. Mossel, J. Pach, M. Sharir, S. Smorodinsky, and U. Wagner. Online conflict-free coloring for intervals. *SIAM Journal on Computing*, 36(5):1342–1359, 2006.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, Cambridge, 2001.

[5] G. Even, Z. Lotker, D. Ron, and S. Smorodinsky. Conflict-free colorings of simple geometric regions with applications to frequency assignment in cellular networks. *SIAM Journal on Computing*, 33(1):94–136, 2003.

[6] S. Har-Peled and S. Smorodinsky. Conflict-free coloring of points and simple regions in the plane. *Discrete & Computational Geometry*, 34(1):47–70, 2005.

[7] E. Horev, R. Krakovski, and S. Smorodinsky. Conflict-free coloring made stronger. In *Algorithm Theory-SWAT 2010, LNCS*, pages 105–117. 2010.

[8] N. Lev-Tov and D. Peleg. Conflict-free coloring of unit disks. *Discrete Applied Mathematics*, 157(7):1521–1532, 2009.

[9] J. Pach and G. Tóth. Conflict-free colorings. In *Discrete & Computational Geometry*, pages 665–671. 2003.

[10] S. Smorodinsky. On the chromatic number of geometric hypergraphs. *SIAM Journal on Discrete Mathematics*, 21(3):676–687, 2007.

[11] S. Smorodinsky. Conflict-free coloring and its applications. *arXiv preprint arXiv:1005.3616*, 2010.