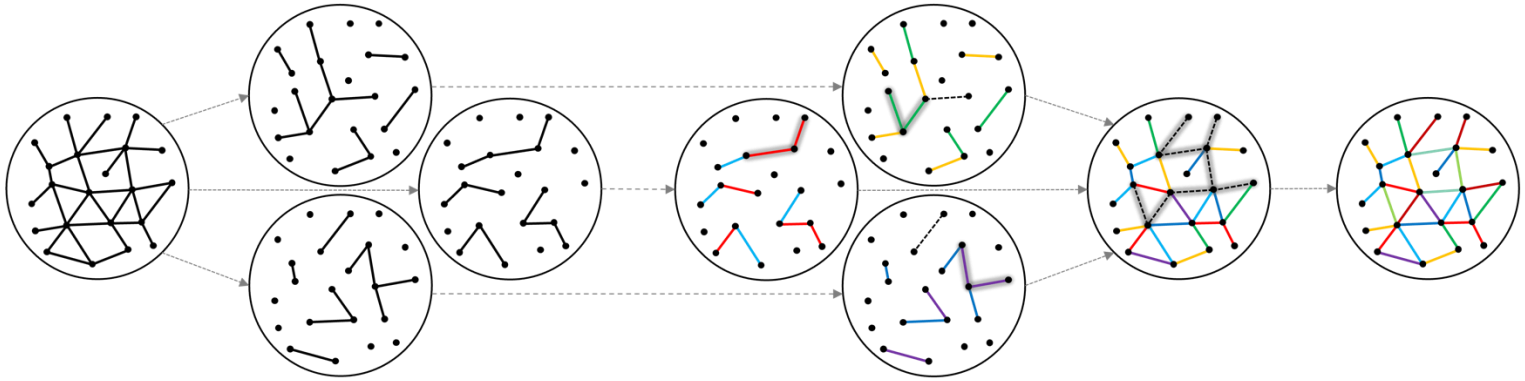


Lecture 5:
Dynamic Edge Coloring via the Nibble Method

Martín Costa

AlgUW: Workshop on Dynamic and Almost Linear-Time Algorithms

Q: How fast can we edge color a graph?

Near-Linear Time Coloring

Theorem. Greedy $2\Delta - 1$ coloring in $O(m \log \Delta)$ time.

Theorem [Duan, He, Zhang, SODA'19]**.** $(1+\epsilon)\Delta$ coloring in $\tilde{O}(m/\epsilon^2)$ time for $\epsilon > 0$, when $\Delta = \Omega(\log n / \epsilon)$.

Theorem [Elkin, Khuzman, arXiv'24]**.** $(1+\epsilon)\Delta$ coloring in $\tilde{O}(m/\epsilon)$ time for $\epsilon > 0$.

Theorem [ABBCSZ, STOC'25]**.** $\Delta + 1$ coloring in $O(m \log \Delta)$ time.

Q: Are there any coloring algorithms that run in $O(m)$ (i.e. **exact-linear**) time?

Exact-Linear Time Coloring

- For $(2+\epsilon)\Delta$ coloring, a trivial algorithm is known:

Randomized Greedy:

For each edge e, sample colors u.a.r. until a color α available at e is found and set $\chi(e) \leftarrow \alpha$.

Analysis:

1. $\Omega(\epsilon\Delta)$ colors available at each edge ⇒ each iteration takes $O(1/\epsilon)$ expected time.
2. Runs in $O(m/\epsilon)$ time w.h.p. for $\epsilon > 0$.

- All known implementations of greedy $2\Delta - 1$ coloring algorithm take $\Omega(m \log \Delta)$ time.

Q1: For any constant $\epsilon > 0$, is there a static $(1+\epsilon)\Delta$ coloring algorithm with $O(m)$ running time?

The Dynamic Setting

The Dynamic Setting

Input: A sequence of *updates* (edge *insertions/deletions*) in a graph G and a parameter Δ such that $\Delta(G)$ never exceeds Δ.

Want to (explicitly) maintain a coloring χ of G. Let $G^{(t)}$ denote the graph G after the t^{th} update.

After the t^{th} update: Update χ to an edge coloring of the graph $G^{(t)}$.

update time = the time taken to handle an update.

recourse = the number of edges that change color during an update.

recourse $\leq$ update time.

Dynamic Edge Coloring

Theorem. Dynamic $(1+\epsilon)\Delta$ coloring in $O(\log^7 n/\epsilon^2)$ update time, when $\Delta = \Omega(\log^2 n/\epsilon^2)$.
[Duan, He, Zhang, SODA'19]

Theorem. Dynamic $(1+\epsilon)\Delta$ coloring in $O(\log^9 n \log^6 \Delta/\epsilon^6)$ update time.
[Christiansen, STOC'22]

Q2: For any constant $\epsilon > 0$, can we maintain a $(1+\epsilon)\Delta$ coloring in *constant* update time?

- A dynamic algorithm with $O(1)$ update time ⇒ a static algorithm with $O(m)$ running time.

Barrier 1: A (Soft) Lower Bound

Theorem [Chang, He, Li, Pettie, Uitto, SODA'18]**.** For any $0 < \epsilon \leq 1/3$, there exists a graph G of max degree Δ and a $(1+\epsilon)\Delta$-edge coloring χ of G *with exactly* 1 *uncolored edge* such that:

Extending χ to be a coloring of the whole graph requires changing the colors of $\Omega(\log(\epsilon n)/\epsilon)$ edges.

- Any algorithm that works by ***extending arbitrary colorings*** has $\Omega(\log n)$ recourse.
- [Duan et al., SODA'19] and [Christiansen, STOC'22] take this approach (based on multi-step Vizing chains).
- We need a different approach...

Dynamic Edge Coloring via the Nibble Method

Theorem. Dynamic $(1+\epsilon)\Delta$ coloring with $\text{poly}(1/\epsilon)$ recourse, when $\Delta = \Omega(\log n \,/\, \text{poly}(\epsilon))$.
[Bhattacharya, Grandoni, Wajc, SODA'21]

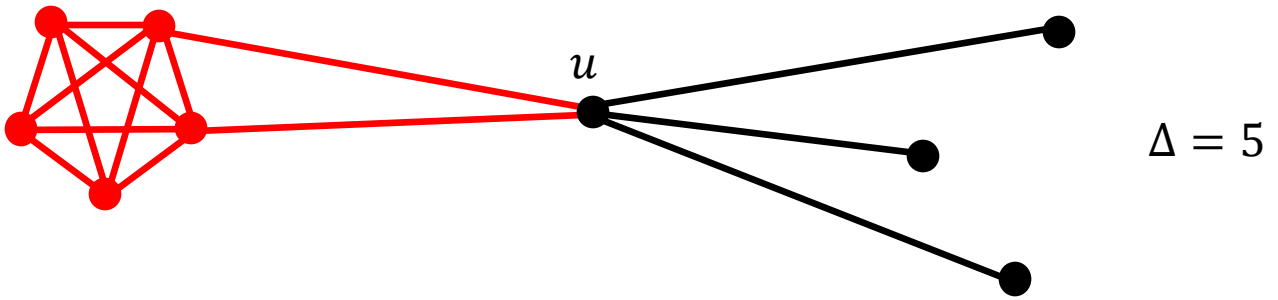
- [Bhattacharya et al., SODA'21] maintain a coloring from a 'nice' distribution → not subject to the soft lower bound.

- Their algorithm is based on the static **NIBBLE** algorithm for edge coloring.
  - A technique that has been well studied in various settings.
  - Completely different to algorithms based on Vizing chains!

- Can we implement this algorithm efficiently?

Barrier 2: Regularization Gadgets

NIBBLE only works on *near-regular graphs*: Need to use 'regularization gadgets':



Create a clique of size $\Delta - 1$ out of dummy nodes for each 'real' node u.

Connect u to $\Delta - \deg(u)$ of its dummy nodes ⇒ near-regular supergraph of G.

- Leads to $\Omega(n\Delta^2)$ running time overhead in the static setting
- *Can we bypass the need for these gadgets?*

Dynamic Edge Coloring via the Nibble Method

Theorem. Dynamic $(1+\epsilon)\Delta$ coloring with $\text{poly}(1/\epsilon)$ update time, when $\Delta \geq (\log n)^{\Theta(\text{poly}(1/\epsilon))}$.
[Bhattacharya, C, Panski, Solomon, SODA'24]

- **Bypass the need for regularization** by using the **subsampling** technique of [Kulkarni, Liu, Sah, Sawhney, Tarnawski, STOC'22].

- This allows for a new variant of **NIBBLE**: works for arbitrary graphs, not just near-regular graphs.

- No need for regularization gadgets – admits an efficient implementation.

Corollary. Static $(1+\epsilon)\Delta$ coloring with $O(m\,\text{poly}(1/\epsilon))$ running time, when $\Delta \geq (\log n)^{\Theta(\text{poly}(1/\epsilon))}$.

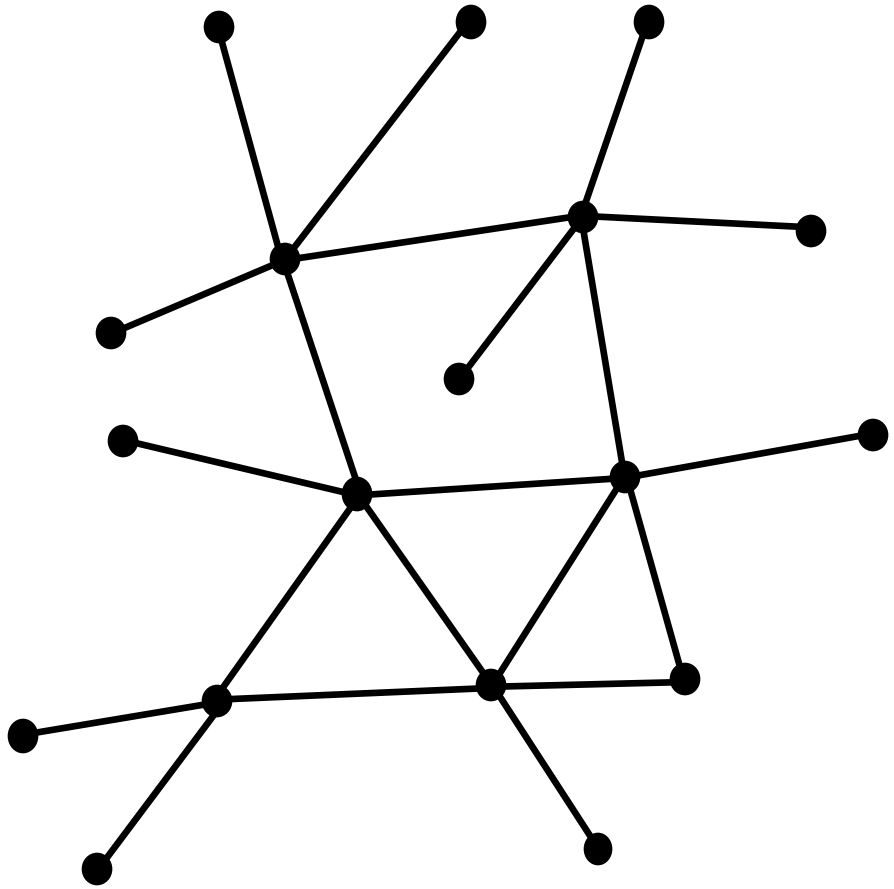
Follow up work removed restriction on Δ in static setting (based on MSVs): [Bernshteyn, Dhawan, arXiv'24]

Rest of the Talk

- The static **NIBBLE** algorithm and its analysis.
- The dynamization of **NIBBLE** and its implementation.
- Open problems.

The Static Algorithm

Static Algorithm: How Does NIBBLE Work?

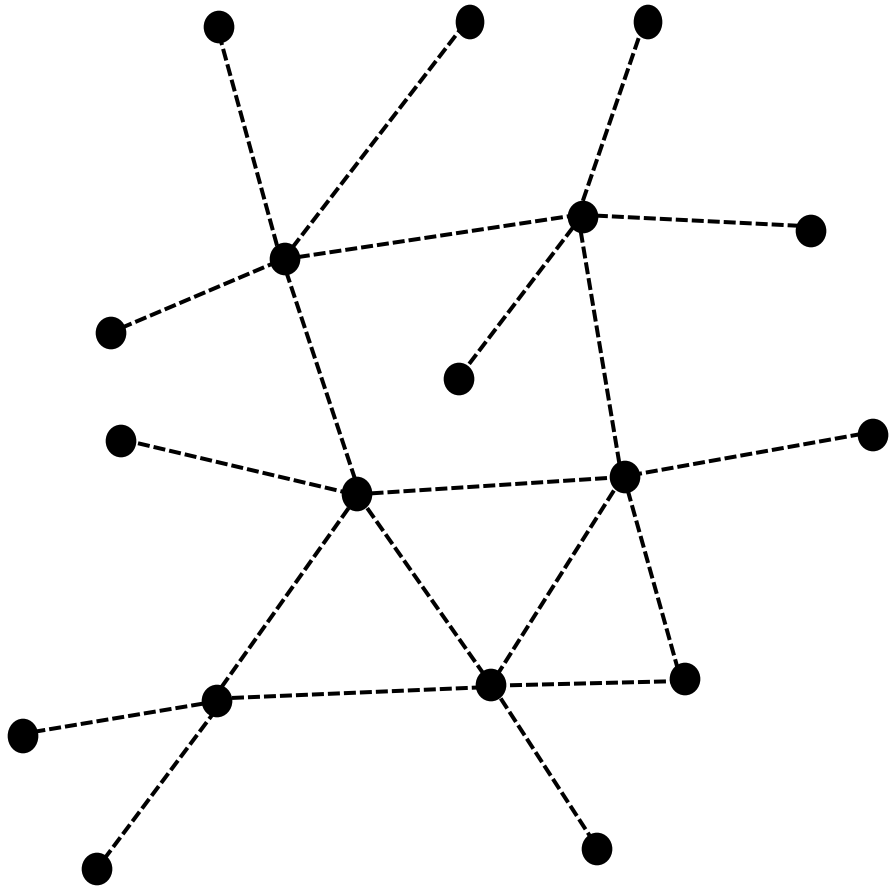


Static Algorithm: How Does NIBBLE Work?

All edges are initially uncolored

Runs in $T = (1/\epsilon)\log(1/\epsilon)$ rounds, uses $(1+\epsilon)\Delta$ colors

For each round $i = 1, \ldots, T$:



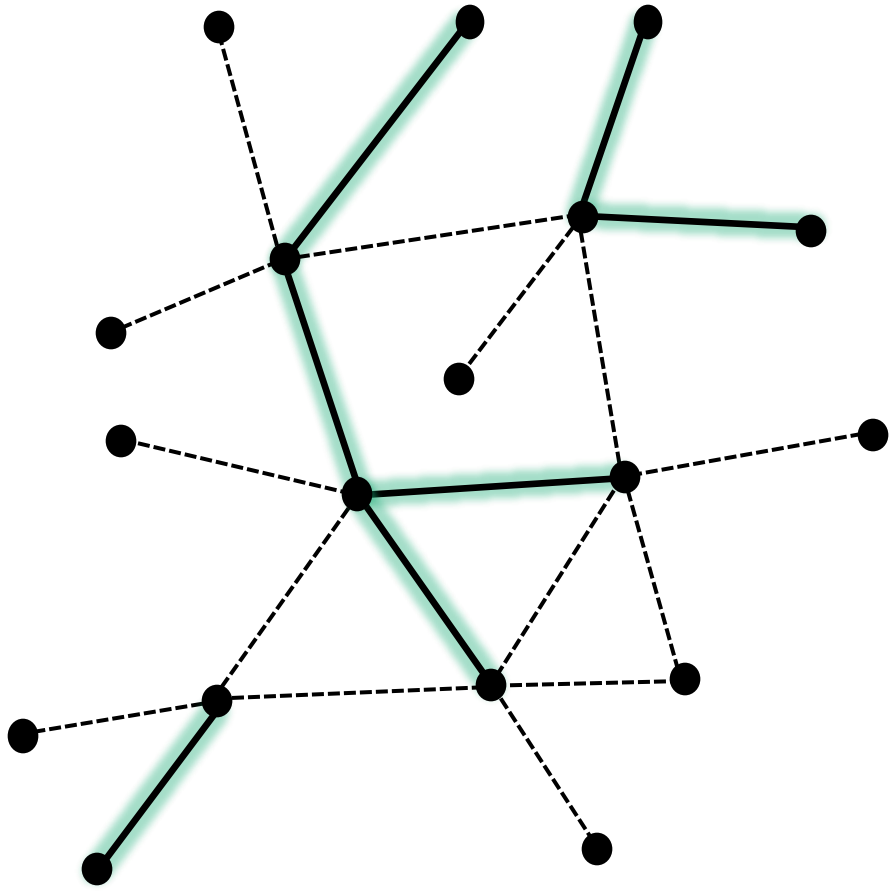
Static Algorithm: How Does NIBBLE Work?

All edges are initially uncolored

Runs in $T = (1/\epsilon)\log(1/\epsilon)$ rounds, uses $(1+\epsilon)\Delta$ colors

For each round $i = 1, \ldots, T$:

1. Sample each uncolored edge e w.p. ϵ



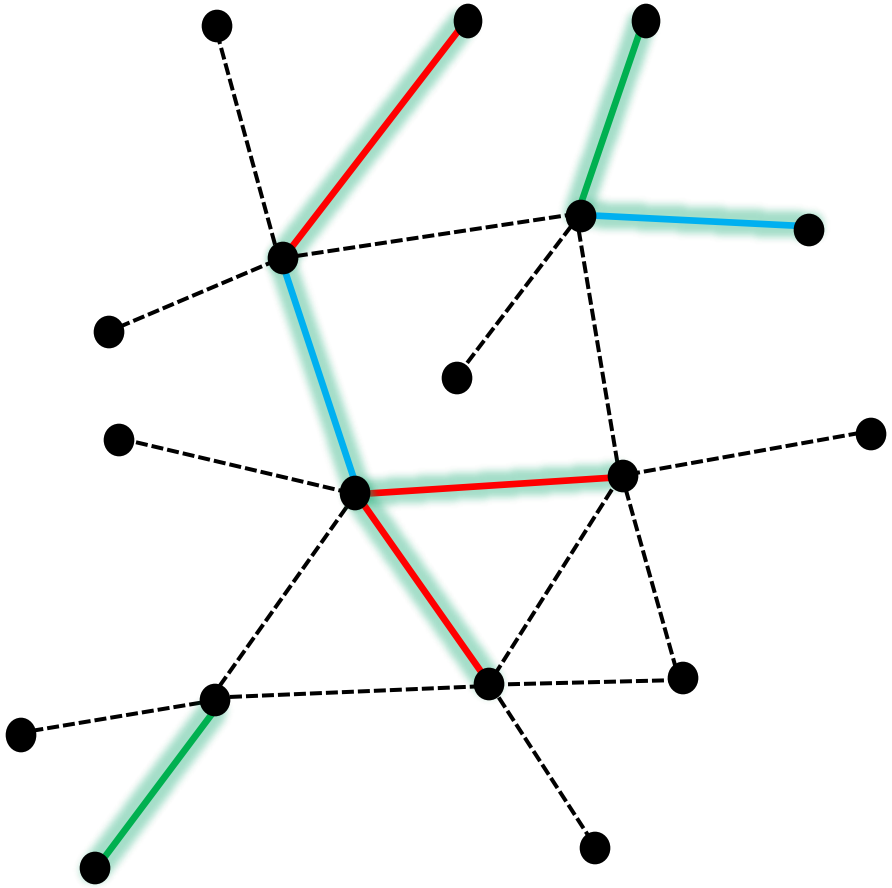
Static Algorithm: How Does NIBBLE Work?

All edges are initially uncolored

Runs in $T = (1/\epsilon)\log(1/\epsilon)$ rounds, uses $(1+\epsilon)\Delta$ colors

For each round $i = 1, \ldots, T$:

1. Sample each uncolored edge e w.p. ϵ
2. Sample a ***tentative*** *color* $\tilde{\chi}(e)$ for each sampled edge u.a.r. from its ***palette*** $P_i(e)$



$P_i(e)$ = colors *available* to e at round i

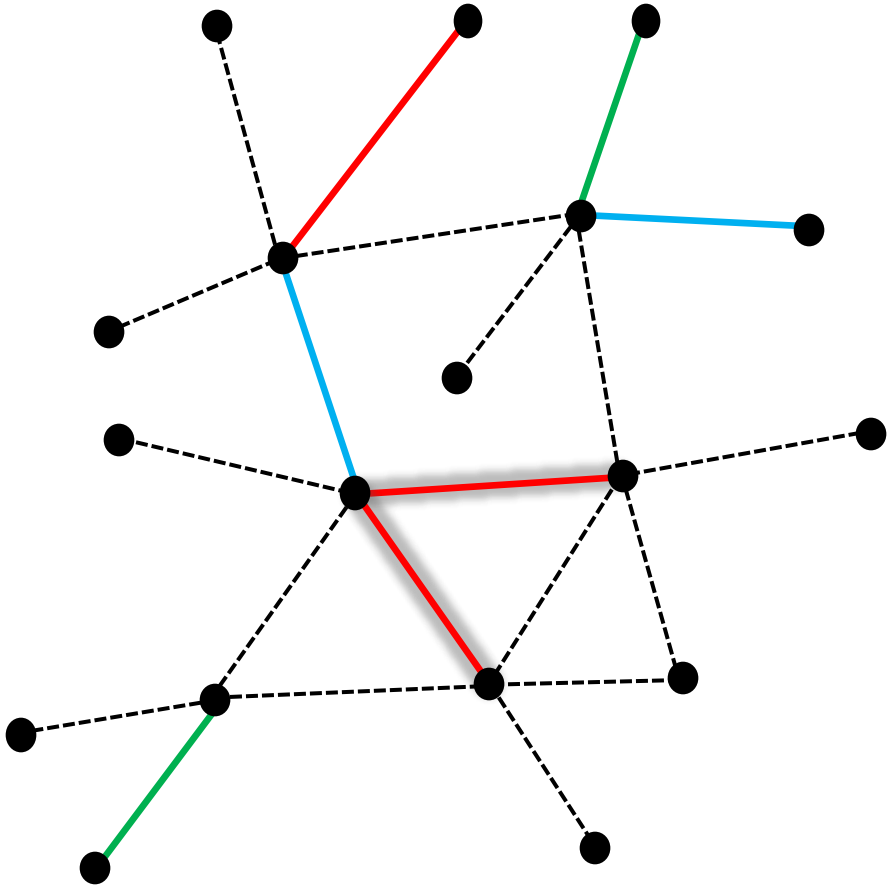
Static Algorithm: How Does NIBBLE Work?

All edges are initially uncolored

Runs in $T = (1/\epsilon)\log(1/\epsilon)$ rounds, uses $(1+\epsilon)\Delta$ colors

For each round $i = 1, \ldots, T$:

1. Sample each uncolored edge e w.p. ϵ
2. Sample a ***tentative*** *color* $\tilde{\chi}(e)$ for each sampled edge u.a.r. from its ***palette*** $P_i(e)$
3. The edges with ***conflicts*** FAIL



$P_i(e)$ = colors *available* to e at round i

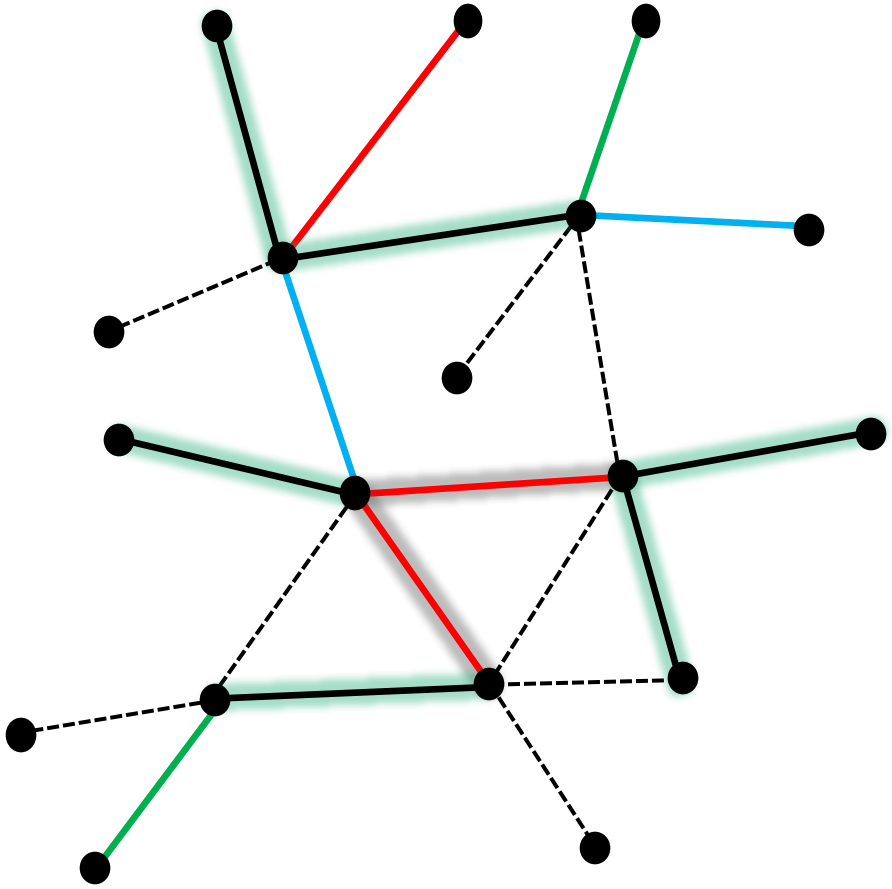
Static Algorithm: How Does NIBBLE Work?

All edges are initially uncolored

Runs in $T = (1/\epsilon)\log(1/\epsilon)$ rounds, uses $(1+\epsilon)\Delta$ colors

For each round $i = 1, \ldots, T$:

1. Sample each uncolored edge e w.p. ϵ
2. Sample a ***tentative*** *color* $\tilde{\chi}(e)$ for each sampled edge u.a.r. from its ***palette*** $P_i(e)$
3. The edges with ***conflicts*** FAIL



$P_i(e)$ = colors *available* to e at round i

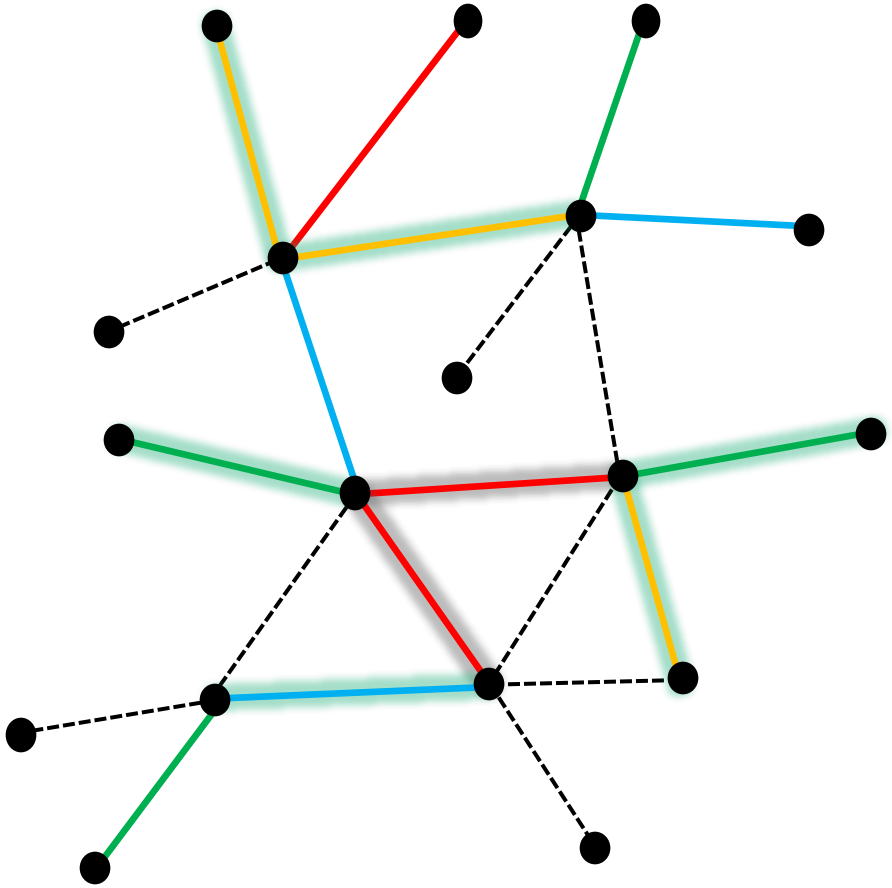
Static Algorithm: How Does NIBBLE Work?

All edges are initially uncolored

Runs in $T = (1/\epsilon)\log(1/\epsilon)$ rounds, uses $(1+\epsilon)\Delta$ colors

For each round $i = 1, \ldots, T$:

1. Sample each uncolored edge e w.p. ϵ
2. Sample a ***tentative*** *color* $\tilde{\chi}(e)$ for each sampled edge u.a.r. from its ***palette*** $P_i(e)$
3. The edges with ***conflicts*** FAIL



$P_i(e)$ = colors *available* to e at round i

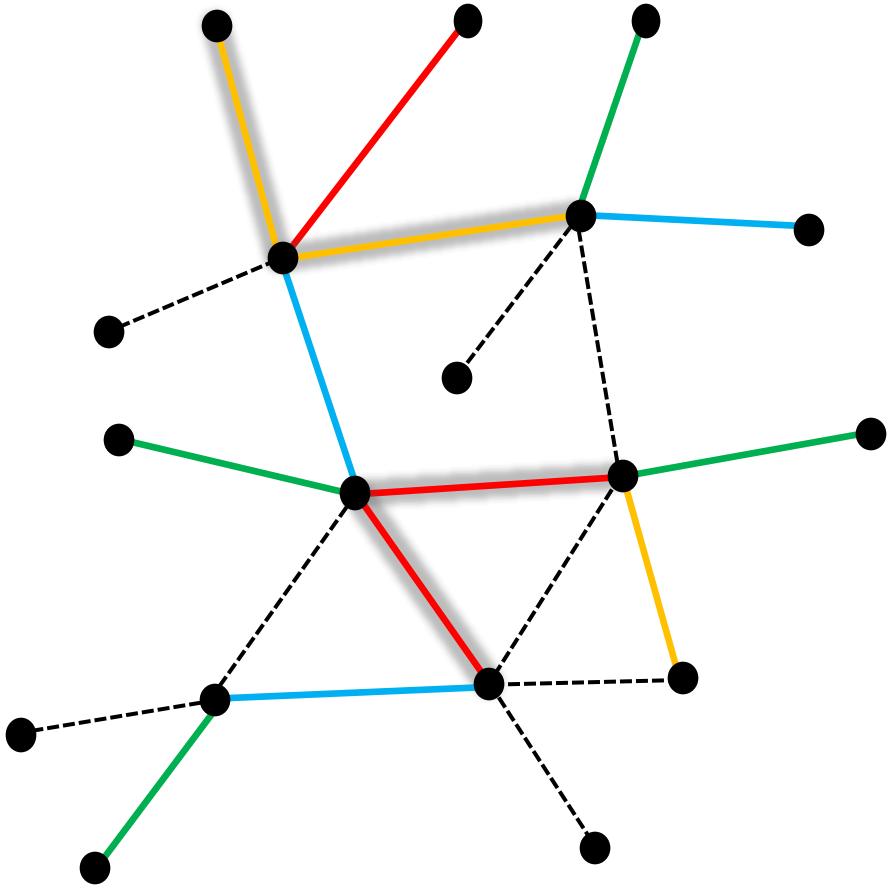
Static Algorithm: How Does NIBBLE Work?

All edges are initially uncolored

Runs in $T = (1/\epsilon)\log(1/\epsilon)$ rounds, uses $(1+\epsilon)\Delta$ colors

For each round $i = 1, \ldots, T$:

1. Sample each uncolored edge e w.p. ϵ
2. Sample a ***tentative*** *color* $\tilde{\chi}(e)$ for each sampled edge u.a.r. from its ***palette*** $P_i(e)$
3. The edges with ***conflicts*** FAIL



$P_i(e)$ = colors *available* to e at round i

Static Algorithm: How Does NIBBLE Work?

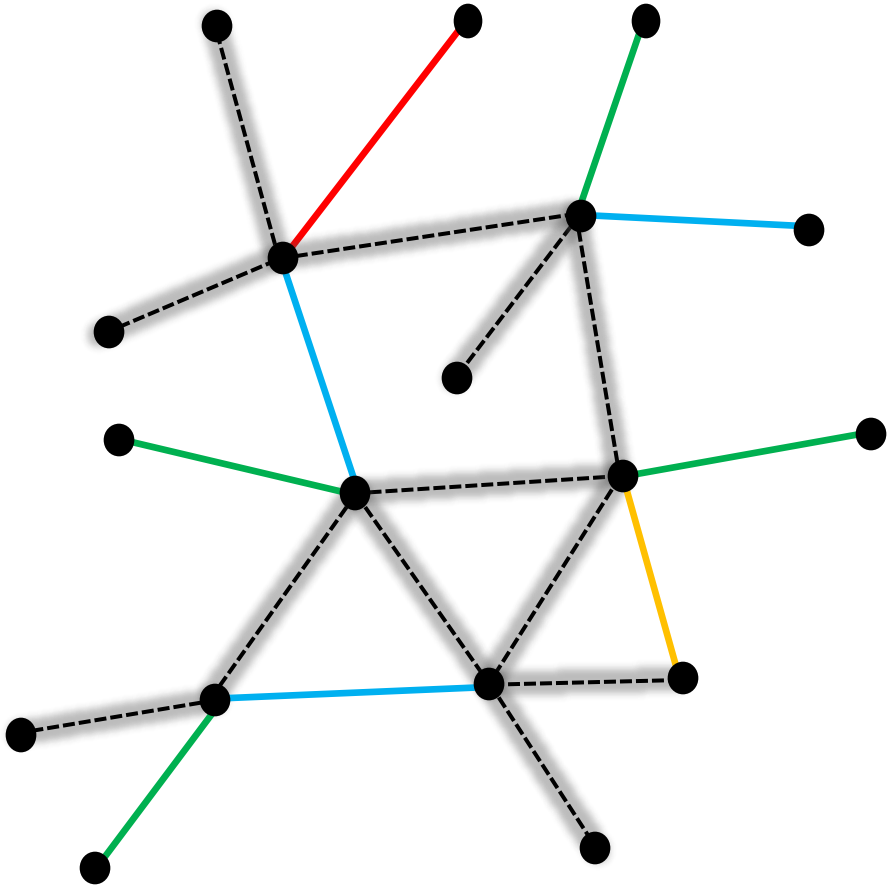
All edges are initially uncolored

Runs in $T = (1/\epsilon)\log(1/\epsilon)$ rounds, uses $(1+\epsilon)\Delta$ colors

For each round $i = 1, \ldots, T$:

1. Sample each uncolored edge e w.p. ϵ
2. Sample a ***tentative*** *color* $\tilde{\chi}(e)$ for each sampled edge u.a.r. from its ***palette*** $P_i(e)$
3. The edges with ***conflicts*** FAIL

Greedily color the subgraph F of all edges that were never sampled or failed using different colors



$P_i(e)$ = colors *available* to e at round i

Static Algorithm: How Does NIBBLE Work?

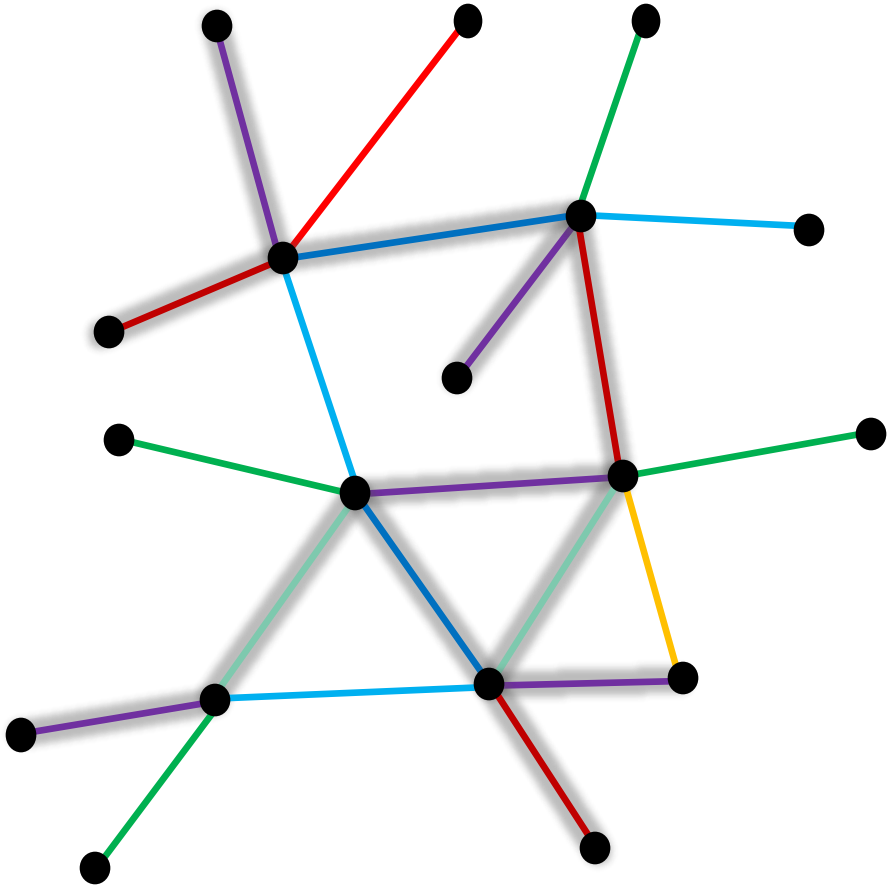
All edges are initially uncolored

Runs in $T = (1/\epsilon)\log(1/\epsilon)$ rounds, uses $(1+\epsilon)\Delta$ colors

For each round $i = 1, \ldots, T$:

1. Sample each uncolored edge e w.p. ϵ
2. Sample a ***tentative*** *color* $\tilde{\chi}(e)$ for each sampled edge u.a.r. from its ***palette*** $P_i(e)$
3. The edges with ***conflicts*** FAIL

Greedily color the subgraph F of all edges that were never sampled or failed using different colors



$P_i(e)$ = colors *available* to e at round i

Static Algorithm: How Does NIBBLE Work?

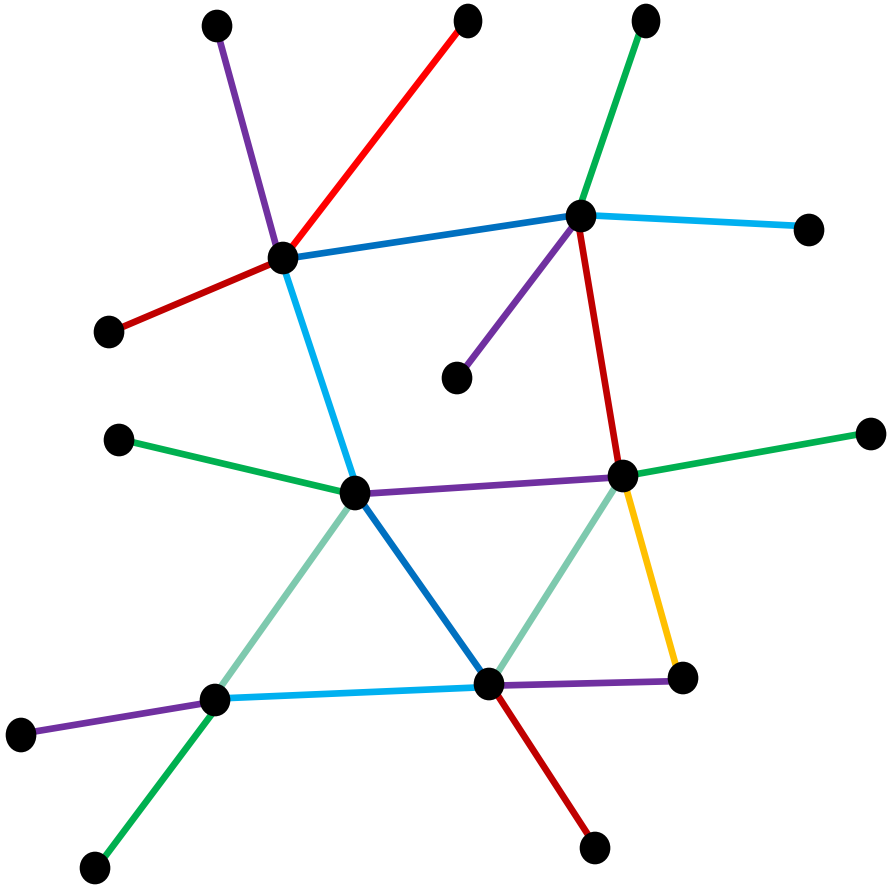
All edges are initially uncolored

Runs in $T = (1/\epsilon)\log(1/\epsilon)$ rounds, uses $(1+\epsilon)\Delta$ colors

For each round $i = 1, \ldots, T$:

1. Sample each uncolored edge e w.p. ϵ
2. Sample a ***tentative*** *color* $\tilde{\chi}(e)$ for each sampled edge u.a.r. from its ***palette*** $P_i(e)$
3. The edges with ***conflicts*** FAIL

Greedily color the subgraph F of all edges that were never sampled or failed using different colors



$P_i(e)$ = colors *available* to e at round i

Static Algorithm: Analysis of NIBBLE

Suppose we can show that:

1. For each edge e sampled during round i, $|P_i(e)| \geq \Omega(\Delta \operatorname{poly}(\epsilon))$.
2. For each node u, $\deg_F(u) \leq O(\epsilon\Delta)$.

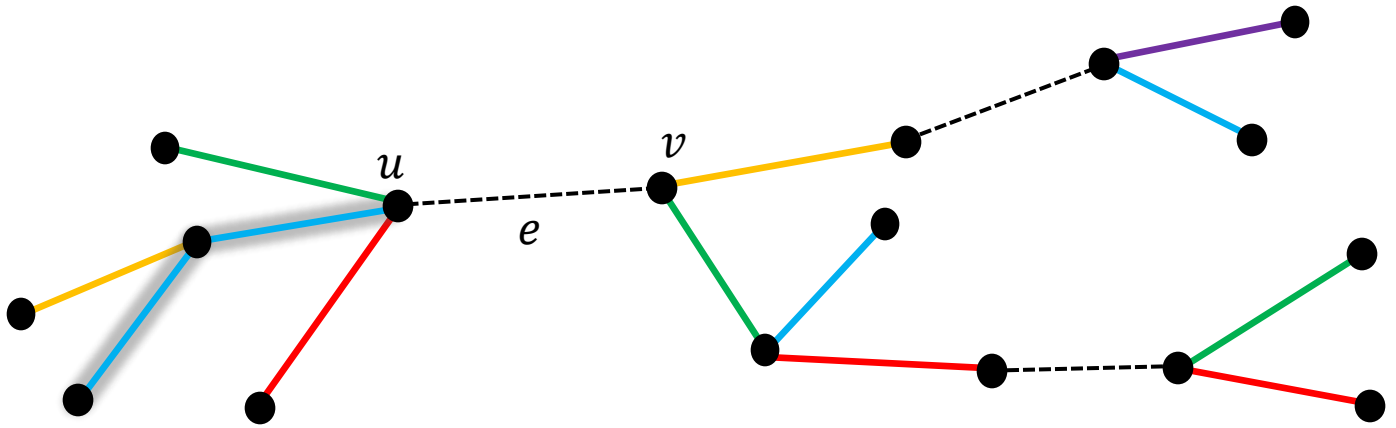
(1) ⇒ can sample colors from $P_i(e)$ in $O(\operatorname{poly}(1/\epsilon))$ time.

(2) ⇒ can color the failed and leftover edges with $O(\epsilon\Delta)$ extra colors.

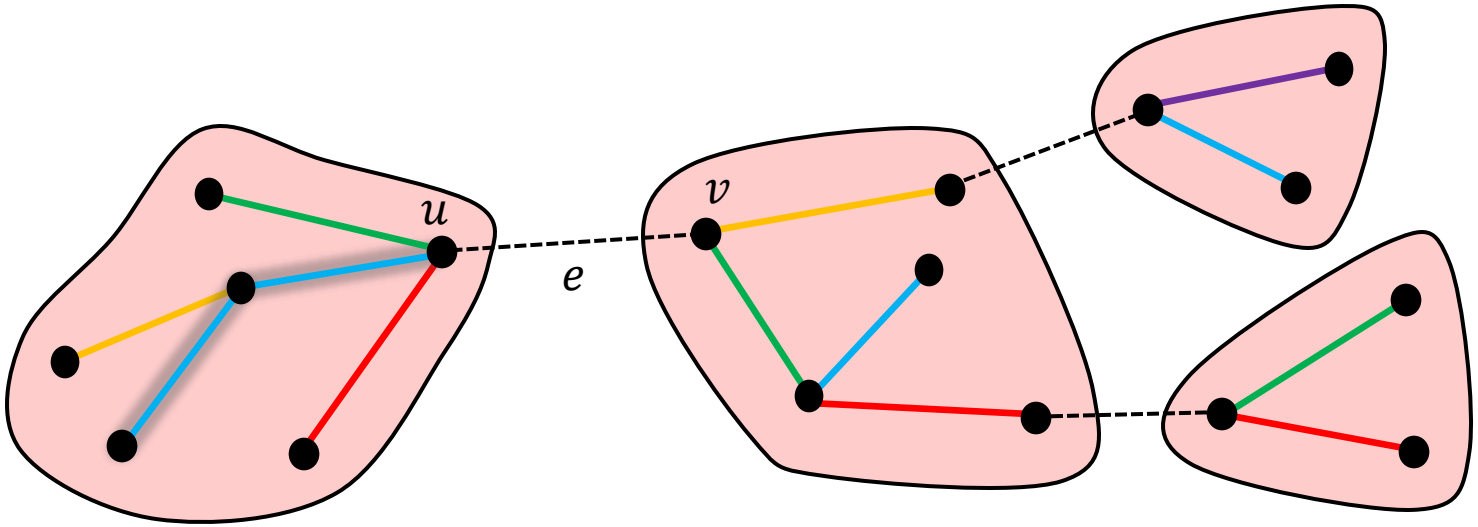
- Leads to a $O(m \operatorname{poly}(1/\epsilon))$ time algorithm that uses $(1+\epsilon)\Delta$ colors.
- **Near-regularity necessary for concentration bounds in standard analysis.**
- *Can we perform a different analysis?*

Static Algorithm: NIBBLE on Forests

- If input graph G is a forest, **NIBBLE** is much easier to analyze.
- Objects that are usually not independent, ***are independent*** when G is a forest.
- Let e be an edge sampled during round i.
- Consider the state of **NIBBLE** after the first $i - 1$ iterations:



Static Algorithm: NIBBLE on Forests



- Consider the graph with the edges appearing in the first $i-1$ rounds.
- u and v are in different connected components.
- $\Rightarrow$ the palettes of $P_i(u)$ and $P_i(v)$ independent.
- Hence:

$$\mathrm{E}[|P_i(e)|] = \sum_c \mathrm{P}[c \in P_i(e)] = \sum_c \mathrm{P}[c \in P_i(u)] \cdot \mathrm{P}[c \in P_i(v)] = \frac{|P_i(u)| \cdot |P_i(v)|}{(1+\epsilon)\Delta} \geq \Omega(\Delta\epsilon^2).$$

Static Algorithm: NIBBLE on Locally Treelike Graphs

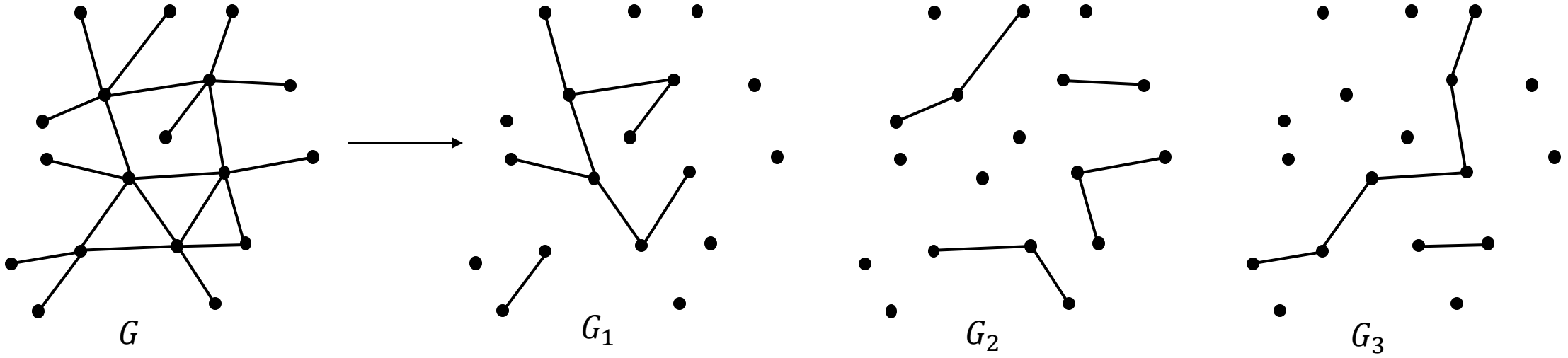
- Are there more general graphs that can be analyzed in this simple manner?

Claim: The color assigned to an edge e by **NIBBLE** depends only on the structure of the $(T+1)$-neighborhood of the edge e.

- If the $(T+1)$-neighborhood of every edge e is a tree, the same analysis goes through.
- We call such graphs 'locally treelike'.
- Reduction to locally treelike graphs [Kulkarni et al., STOC'22].

Static Algorithm: Subsampling to Locally Treelike Graphs

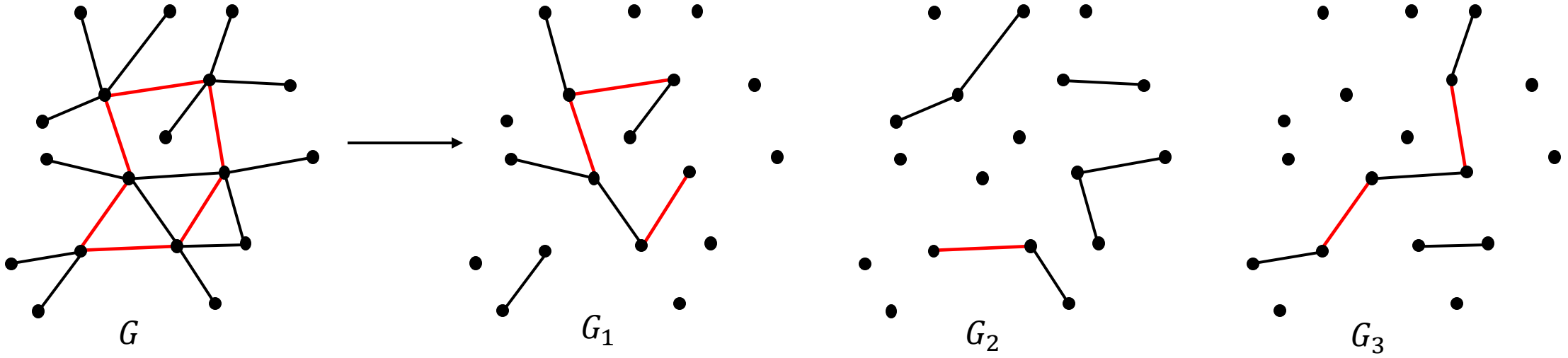
Subsampling: Split the graph G into graphs $G_1, \dots, G_\eta$ by placing each edge $e \in G$ into one of the G_i independently and u.a.r. (η is a parameter that depends on Δ and ϵ).



- Cycles in G are unlikely to appear in any of the G_j.

Static Algorithm: Subsampling to Locally Treelike Graphs

Subsampling: Split the graph G into graphs $G_1, \dots, G_\eta$ by placing each edge $e \in G$ into one of the G_i independently and u.a.r. (η is a parameter that depends on Δ and ϵ).



- Cycles in G are unlikely to appear in any of the G_j.

Static Algorithm: Subsampling to Locally Treelike Graphs

Subsampling: Split the graph G into graphs $G_1, \dots, G_\eta$ by placing each edge $e \in G$ into one of the G_i independently and u.a.r. (η is a parameter that depends on Δ and ϵ).

- Cycles in G are unlikely to appear in any of the G_j.

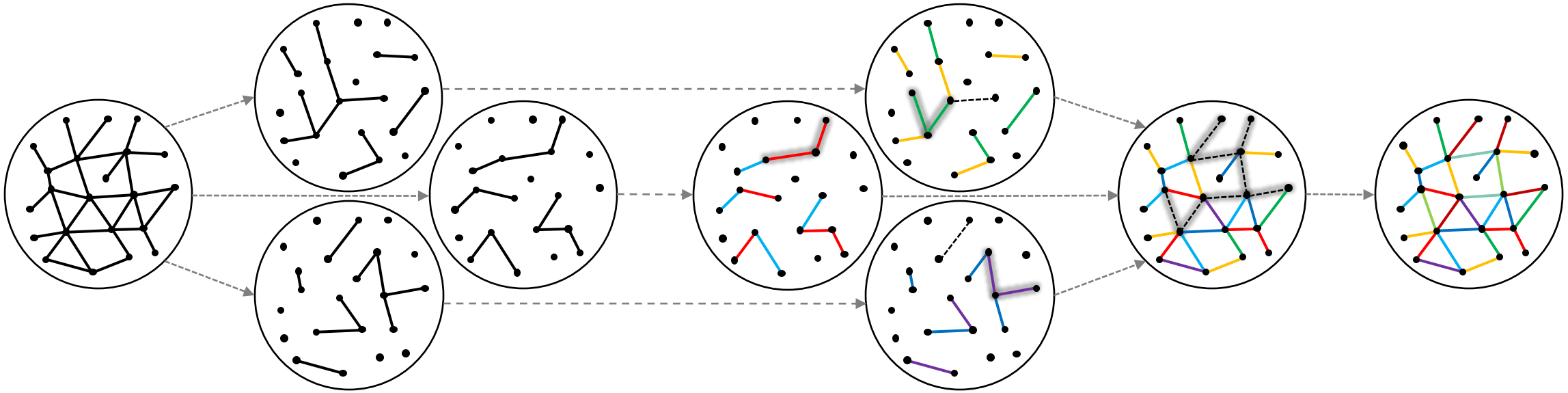
More Precisely:

- Let $G^\star$ be the graph that contains an edge $e \in G_j$ if the $(T+1)$-neighborhood of e in G_j is *not* a tree.
- Then $G^\star$ has maximum degree $\leq \epsilon\Delta$ w.h.p.
- Run **NIBBLE** on the G_j and combine the colorings.
- *Intuitively, for large enough η, the graphs $G_1, \dots, G_\eta$ are locally treelike.**

*Literally false, but morally true

Static Algorithm: Final Algorithm

1. Split the graph G into graphs $G_1, \ldots, G_\eta$ using the subsampling technique.
2. Run **NIBBLE** on each of the G_j (using different colors) to obtain tentative colorings.
3. Run the greedy algorithm on edges that fail/are never sampled across *all* the G_j.



The Dynamic Algorithm

Dynamic Algorithm: High Level Approach

The main idea: Maintain the output of our static algorithm as the input changes.

Fix random bits so output of static algorithm depends only on edges in G.

After an update: Iterate through rounds propagating changes in colors.

The update procedure needs to:

1. Change as few colors as possible (*low recourse*).
2. Be efficient to implement (*update time proportional to recourse*).

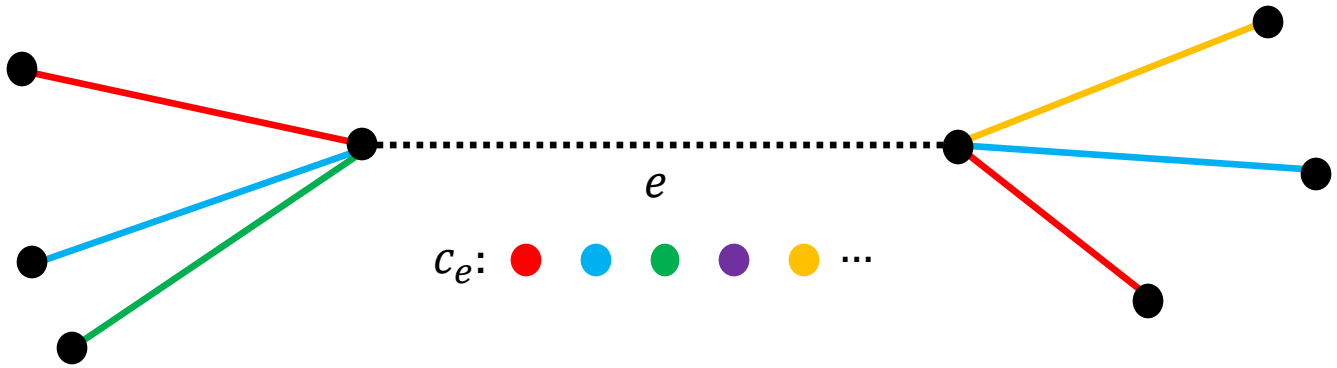
- [Bhattacharya et al., SODA'21] obtain (1) but not (2).

Dynamic Algorithm: Fixing the Randomness

At the start of the algorithm, each **potential edge** $e \in \binom{V}{2}$ is assigned:

1. A random index $j_e \in [\eta]$, determining its subsampled graph.
2. A random index $i_e \in [T]$, determining its round.
3. A random color sequence $c_e(1), \ldots, c_e(1/\epsilon^2)$, for sampling its tentative color.

Sampling the tentative color $\tilde{\chi}(e)$:

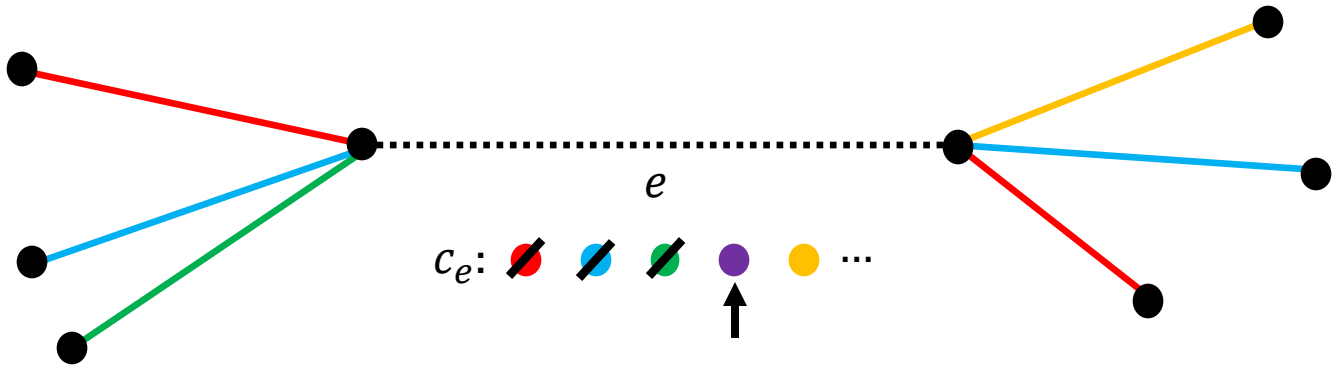


Dynamic Algorithm: Fixing the Randomness

At the start of the algorithm, each **potential edge** $e \in \binom{V}{2}$ is assigned:

1. A random index $j_e \in [\eta]$, determining its subsampled graph.
2. A random index $i_e \in [T]$, determining its round.
3. A random color sequence $c_e(1), \ldots, c_e(1/\epsilon^2)$, for sampling its tentative color.

Sampling the tentative color $\tilde{\chi}(e)$:

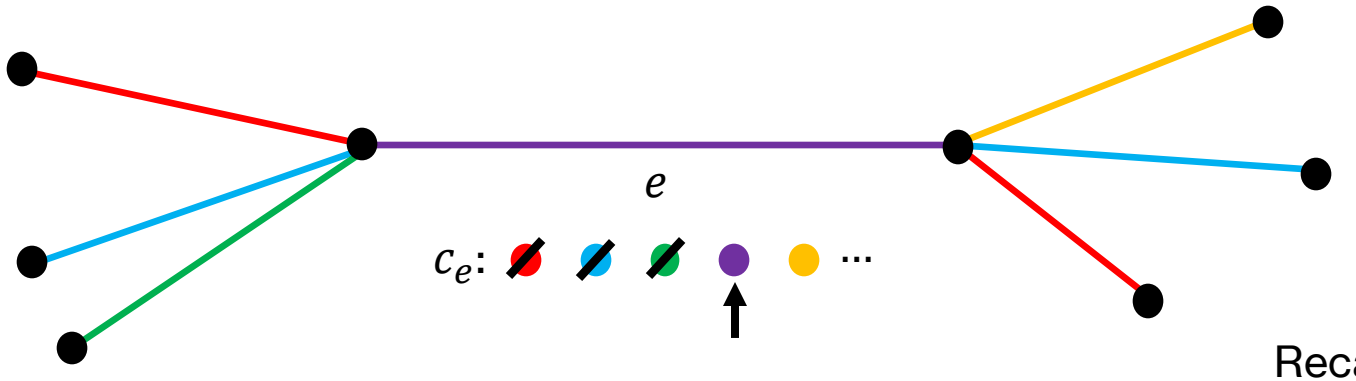


Dynamic Algorithm: Fixing the Randomness

At the start of the algorithm, each **potential edge** $e \in \binom{V}{2}$ is assigned:

1. A random index $j_e \in [\eta]$, determining its subsampled graph.
2. A random index $i_e \in [T]$, determining its round.
3. A random color sequence $c_e(1), \ldots, c_e(1/\epsilon^2)$, for sampling its tentative color.

Sampling the tentative color $\tilde{\chi}(e)$:



Recall that $|P_i(e)| \geq \Omega(\Delta\epsilon^2)$

Dynamic Algorithm: Fixing the Randomness

At the start of the algorithm, each **potential edge** $e \in \binom{V}{2}$ is assigned:

1. A random index $j_e \in [\eta]$, determining its subsampled graph.
2. A random index $i_e \in [T]$, determining its round.
3. A random color sequence $c_e(1), \ldots, c_e(1/\epsilon^2)$, for sampling its tentative color.

- All random bits fixed in advance ⇒ output only depends on edges present in G.

- **We can now bound the recourse of an update.**

Dynamic Algorithm: Bounding the Recourse

$A^{(t)} =$ **edges that change tentative colors** during the t^{th} update.

$A_i^{(t)} =$ edges in $A^{(t)}$ at round i.

Two Key Lemmas:

Lemma. The recourse of the t^{th} update is $O(|A^{(t)}|)$**.**

Lemma. $\mathrm{E}\left[|A_i^{(t)}|\right] \leq 4\epsilon \cdot \mathrm{E}\left[|A_{<i}^{(t)}|\right]$**.**

After the insertion or deletion of an edge e:

$\Rightarrow \mathrm{E}[\text{recourse}] \leq \mathrm{E}\left[|A_{\leq T}^{(t)}|\right] \leq (1+4\epsilon)^T \cdot \mathrm{E}\left[|A_{i_e}^{(t)}|\right] \leq (1+4\epsilon)^T \leq 1/\epsilon^4$ (up to a $O(1)$ factor)

Dynamic Algorithm: Efficiently Propagating Changes

How can we *efficiently* identify changes in the coloring caused by some $e \in A_i^{(t)}$?

Let c and c' denote the *previous* and *new* colors of e respectively.

Let f be an edge sharing an endpoint with e s.t. $i_f \geq i_e$.

$$\mathrm{P}[c \in c_f] \leq \frac{1}{\epsilon^2} \cdot \frac{1}{(1+\epsilon)\Delta} \leq O\left(\frac{1}{\epsilon^2 \Delta}\right) \quad \text{and} \quad \mathrm{P}[c' \in c_f] \leq O\left(\frac{1}{\epsilon^2 \Delta}\right).$$

Thus, in expectation, $O(1/\epsilon^2)$ edges can be directly affected by this change.

With the appropriate data structures, we can identify these edges efficiently and resample their colors.

⇒ update time is proportional to recourse.

Dynamic Algorithm: Summary

Maintain the output of our static algorithm as the input changes.

Fix random bits so output of static algorithm depends only on edges in G.

After an update: Iterate through rounds propagating changes in colors.

Recourse is $O(\text{poly}(1/\epsilon))$

Update time proportional to recourse ⇒ gives $O(\text{poly}(1/\epsilon))$ update time.

Open Problems

Open Problems in Edge Coloring

Q: Can we close the gap between the dynamic and static setting?

Open Problem. Can we get dynamic $\Delta + \tilde{O}(\Delta^{0.99})$ coloring in $\tilde{O}(1)$ update time?

- Even getting $\Delta + \tilde{O}(\Delta^{0.99})$ coloring in $\tilde{O}(1)$ recourse is not known.
- Not clear if this is possible or not, can we get a lower bound?

Open Problem. An incremental $(\Delta + 1)$-coloring algorithm with $\tilde{O}(1)$ update time?

Open Problem. A parallel $(\Delta + 1)$-coloring algorithm with $\tilde{O}(1)$ depth?

Questions?