Dynamic Connectivity

Christian Wulff-Nilsen
Algorithmic Techniques for Modern Data Models
DTU

November 28, 2025

Problem Definition

- Obtain an efficient data structure supporting the following operations in a dynamically changing graph $G = (V, E)$:

Problem Definition

- Obtain an efficient data structure supporting the following operations in a dynamically changing graph $G = (V, E)$:
  - `insert`(u, v): inserts edge (u, v) in E

Problem Definition

- Obtain an efficient data structure supporting the following operations in a dynamically changing graph $G = (V, E)$:
  - `insert`(u, v): inserts edge (u, v) in E
  - `delete`(u, v): deletes edge (u, v) from E

Problem Definition

- Obtain an efficient data structure supporting the following operations in a dynamically changing graph $G = (V, E)$:
  - `insert`(u, v): inserts edge (u, v) in E
  - `delete`(u, v): deletes edge (u, v) from E
  - `connected`(u, v): reports whether vertices u and v are connected in G

Problem Definition

- Obtain an efficient data structure supporting the following operations in a dynamically changing graph $G = (V, E)$:
  - $\texttt{insert}(u, v)$: inserts edge (u, v) in E
  - $\texttt{delete}(u, v)$: deletes edge (u, v) from E
  - $\texttt{connected}(u, v)$: reports whether vertices u and v are connected in G
- We refer to `insert` and `delete` as update operations and to `connected` as a query operation

Problem Definition

- Obtain an efficient data structure supporting the following operations in a dynamically changing graph $G = (V, E)$:
  - `insert`(u, v): inserts edge (u, v) in E
  - `delete`(u, v): deletes edge (u, v) from E
  - `connected`(u, v): reports whether vertices u and v are connected in G
- We refer to `insert` and `delete` as update operations and to `connected` as a query operation
- Initial graph: $|V| = n$ vertices, $E = \emptyset$

Problem Definition

- Obtain an efficient data structure supporting the following operations in a dynamically changing graph $G = (V, E)$:
  - $\texttt{insert}(u, v)$: inserts edge (u, v) in E
  - $\texttt{delete}(u, v)$: deletes edge (u, v) from E
  - $\texttt{connected}(u, v)$: reports whether vertices u and v are connected in G
- We refer to `insert` and `delete` as update operations and to `connected` as a query operation
- Initial graph: $|V| = n$ vertices, $E = \emptyset$
- Updates and queries are revealed one by one in an online sequence

Problem Definition

- Obtain an efficient data structure supporting the following operations in a dynamically changing graph $G = (V, E)$:
  - $\texttt{insert}(u, v)$: inserts edge (u, v) in E
  - $\texttt{delete}(u, v)$: deletes edge (u, v) from E
  - $\texttt{connected}(u, v)$: reports whether vertices u and v are connected in G
- We refer to `insert` and `delete` as update operations and to `connected` as a query operation
- Initial graph: $|V| = n$ vertices, $E = \emptyset$
- Updates and queries are revealed one by one in an online sequence
- We give a data structure with:
  - $O(\log n)$ *worst-case* query time
  - $O(\log^2 n)$ *amortized* update time

Edge Levels and Clusters

- Our data structure will maintain a *level* $\ell(e)$ for each $e \in E$ where $0 \leq \ell(e) \leq \ell_{\max} = \lfloor \log n \rfloor$

Edge Levels and Clusters

- Our data structure will maintain a *level* $\ell(e)$ for each $e \in E$ where $0 \leq \ell(e) \leq \ell_{\max} = \lfloor \log n \rfloor$
- For $0 \leq i \leq \ell_{\max}$, let $G_i = (V, E_i)$ denote the subgraph of G containing edges e with $\ell(e) \geq i$

Edge Levels and Clusters

- Our data structure will maintain a *level* $\ell(e)$ for each $e \in E$ where $0 \leq \ell(e) \leq \ell_{\max} = \lfloor \log n \rfloor$
- For $0 \leq i \leq \ell_{\max}$, let $G_i = (V, E_i)$ denote the subgraph of G containing edges e with $\ell(e) \geq i$
- We have $E = E_0 \supseteq E_1 \supseteq E_2 \supseteq \cdots \supseteq E_{\ell_{\max}}$

Edge Levels and Clusters

- Our data structure will maintain a *level* $\ell(e)$ for each $e \in E$ where $0 \leq \ell(e) \leq \ell_{\max} = \lfloor \log n \rfloor$
- For $0 \leq i \leq \ell_{\max}$, let $G_i = (V, E_i)$ denote the subgraph of G containing edges e with $\ell(e) \geq i$
- We have $E = E_0 \supseteq E_1 \supseteq E_2 \supseteq \cdots \supseteq E_{\ell_{\max}}$
- Connected components of G_i are called i-*clusters* or just *clusters*

Edge Levels and Clusters

- Our data structure will maintain a *level* $\ell(e)$ for each $e \in E$ where $0 \leq \ell(e) \leq \ell_{\max} = \lfloor \log n \rfloor$
- For $0 \leq i \leq \ell_{\max}$, let $G_i = (V, E_i)$ denote the subgraph of G containing edges e with $\ell(e) \geq i$
- We have $E = E_0 \supseteq E_1 \supseteq E_2 \supseteq \cdots \supseteq E_{\ell_{\max}}$
- Connected components of G_i are called i-*clusters* or just *clusters*
- Invariant: any i-cluster contains at most $\lfloor n/2^i \rfloor$ vertices

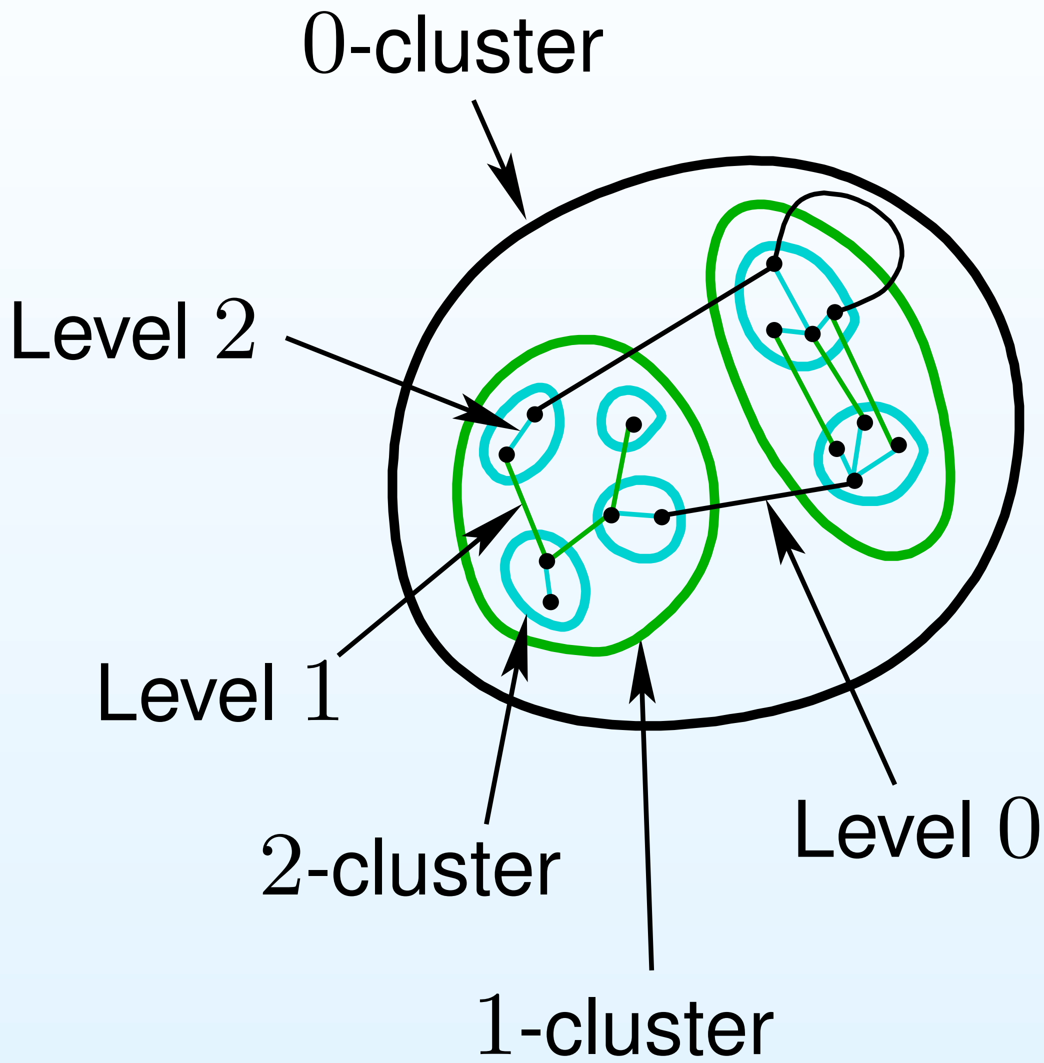
Edge Levels and Clusters

- Our data structure will maintain a *level* $\ell(e)$ for each $e \in E$ where $0 \le \ell(e) \le \ell_{\max} = \lfloor \log n \rfloor$
- For $0 \le i \le \ell_{\max}$, let $G_i = (V, E_i)$ denote the subgraph of G containing edges e with $\ell(e) \ge i$
- We have $E = E_0 \supseteq E_1 \supseteq E_2 \supseteq \cdots \supseteq E_{\ell_{\max}}$
- Connected components of G_i are called i*-clusters* or just *clusters*
- Invariant: any i-cluster contains at most $\lfloor n/2^i \rfloor$ vertices
- 0-clusters are the connected components of G

Edge Levels and Clusters

- Our data structure will maintain a *level* $\ell(e)$ for each $e \in E$ where $0 \leq \ell(e) \leq \ell_{\max} = \lfloor \log n \rfloor$
- For $0 \leq i \leq \ell_{\max}$, let $G_i = (V, E_i)$ denote the subgraph of G containing edges e with $\ell(e) \geq i$
- We have $E = E_0 \supseteq E_1 \supseteq E_2 \supseteq \cdots \supseteq E_{\ell_{\max}}$
- Connected components of G_i are called i*-clusters* or just *clusters*
- Invariant: any i-cluster contains at most $\lfloor n/2^i \rfloor$ vertices
- 0-clusters are the connected components of G
- $\ell_{\max}$-clusters are vertices of V (why?)

Clusters



Cluster forest

- The *cluster forest* of G is a forest $\mathcal{C}$ of rooted trees where each node u corresponds to a cluster $C(u)$

Cluster forest

- The *cluster forest* of G is a forest $\mathcal{C}$ of rooted trees where each node u corresponds to a cluster $C(u)$
- A node u at level $i < \ell_{\max}$ has as children the level $(i+1)$-nodes v such that $C(v) \subseteq C(u)$

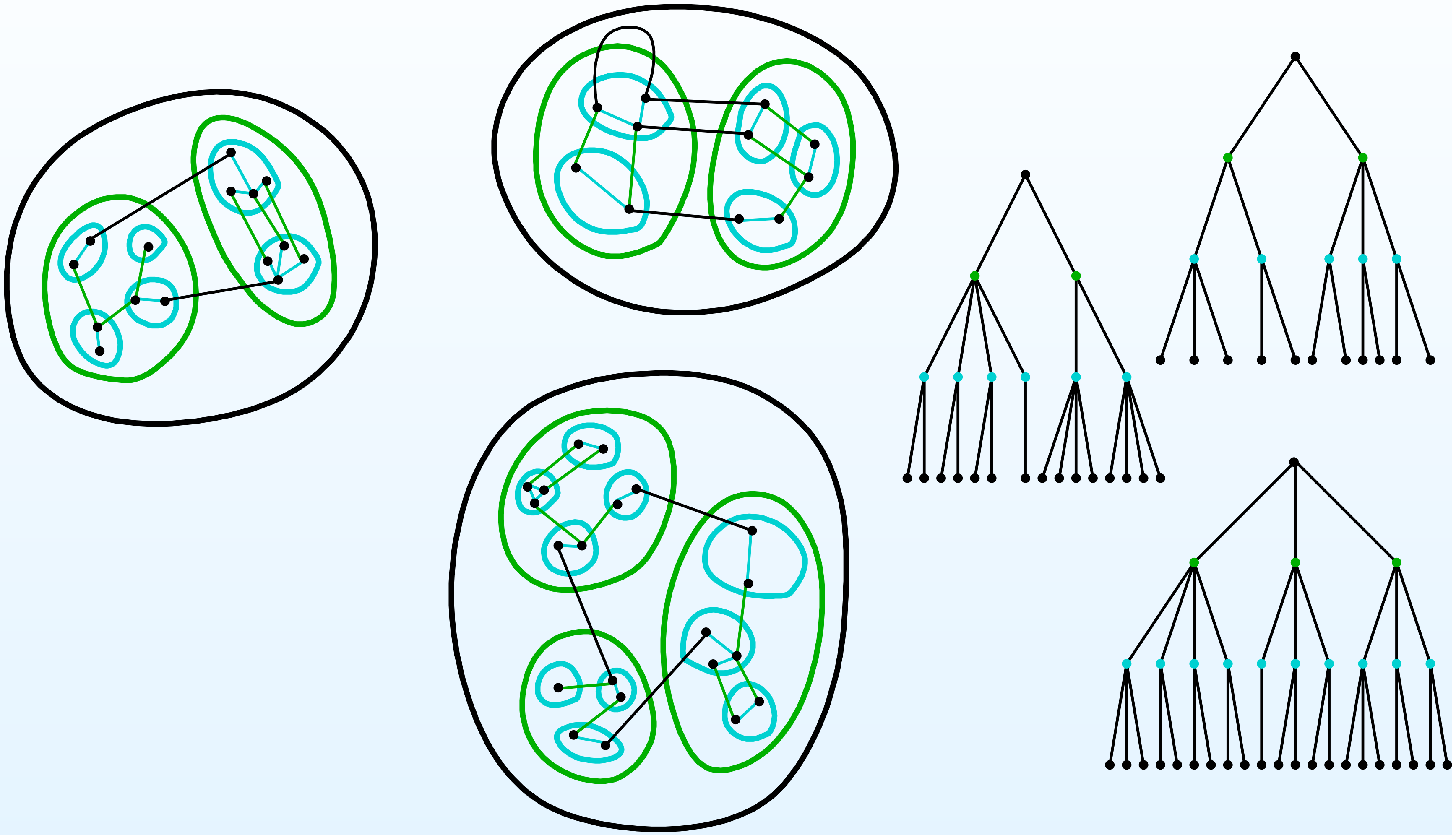
Cluster forest

- The *cluster forest* of G is a forest $\mathcal{C}$ of rooted trees where each node u corresponds to a cluster $C(u)$
- A node u at level $i < \ell_{\max}$ has as children the level $(i+1)$-nodes v such that $C(v) \subseteq C(u)$
- Roots of $\mathcal{C}$ correspond to connected components of G and leaves of $\mathcal{C}$ correspond to vertices of G

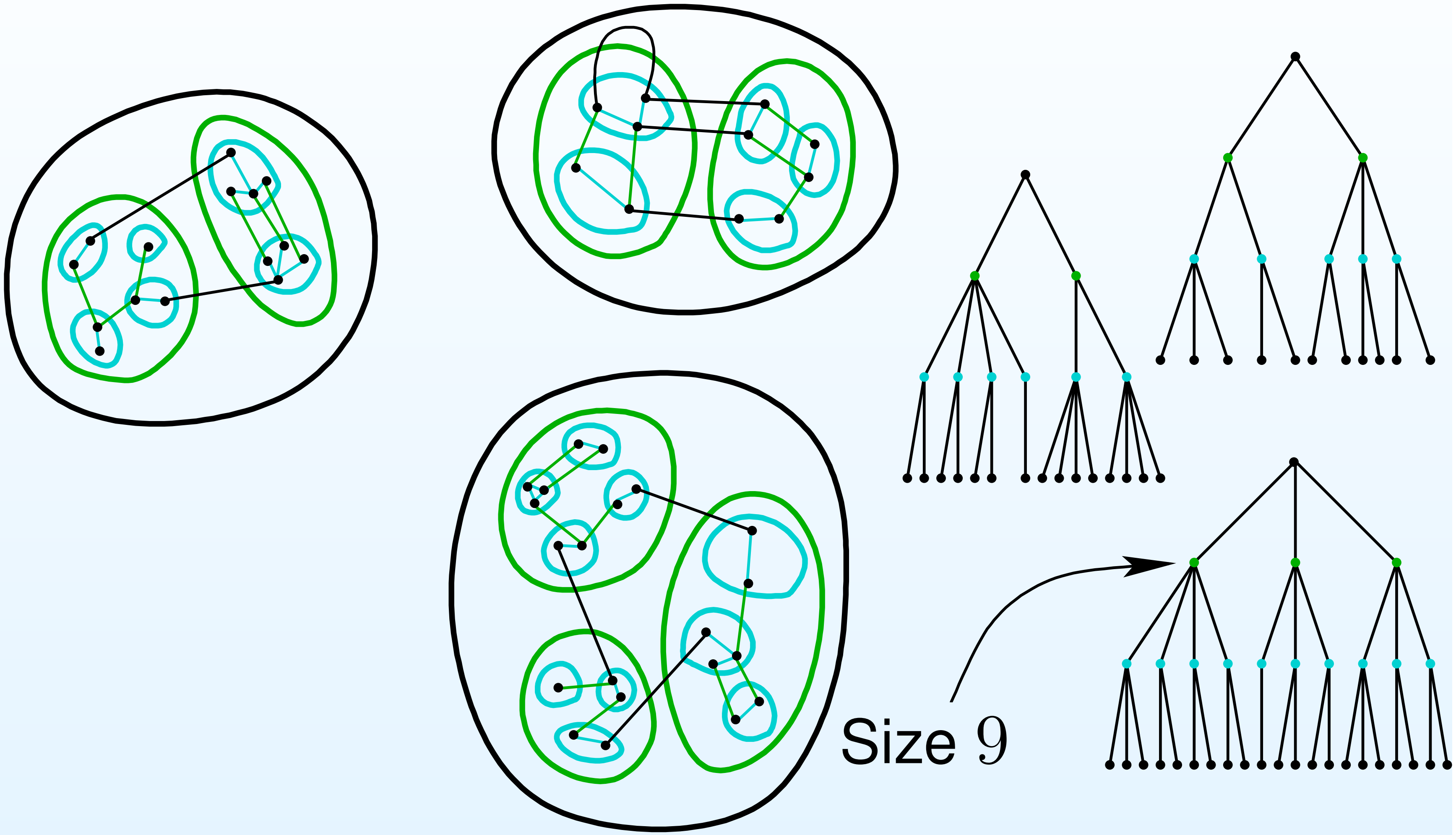
Cluster forest

- The *cluster forest* of G is a forest $\mathcal{C}$ of rooted trees where each node u corresponds to a cluster $C(u)$
- A node u at level $i < \ell_{\max}$ has as children the level $(i+1)$-nodes v such that $C(v) \subseteq C(u)$
- Roots of $\mathcal{C}$ correspond to connected components of G and leaves of $\mathcal{C}$ correspond to vertices of G
- Each node u of $\mathcal{C}$ is associated with its *size* $n(u)$ which is the number of leaves in the subtree of $\mathcal{C}$ rooted at u

Cluster forest



Cluster forest



Answering Queries

Answering Queries

- To determine if vertices u and v are connected in G, traverse the leaf-to-root paths from u and v in $\mathcal{C}$

Answering Queries

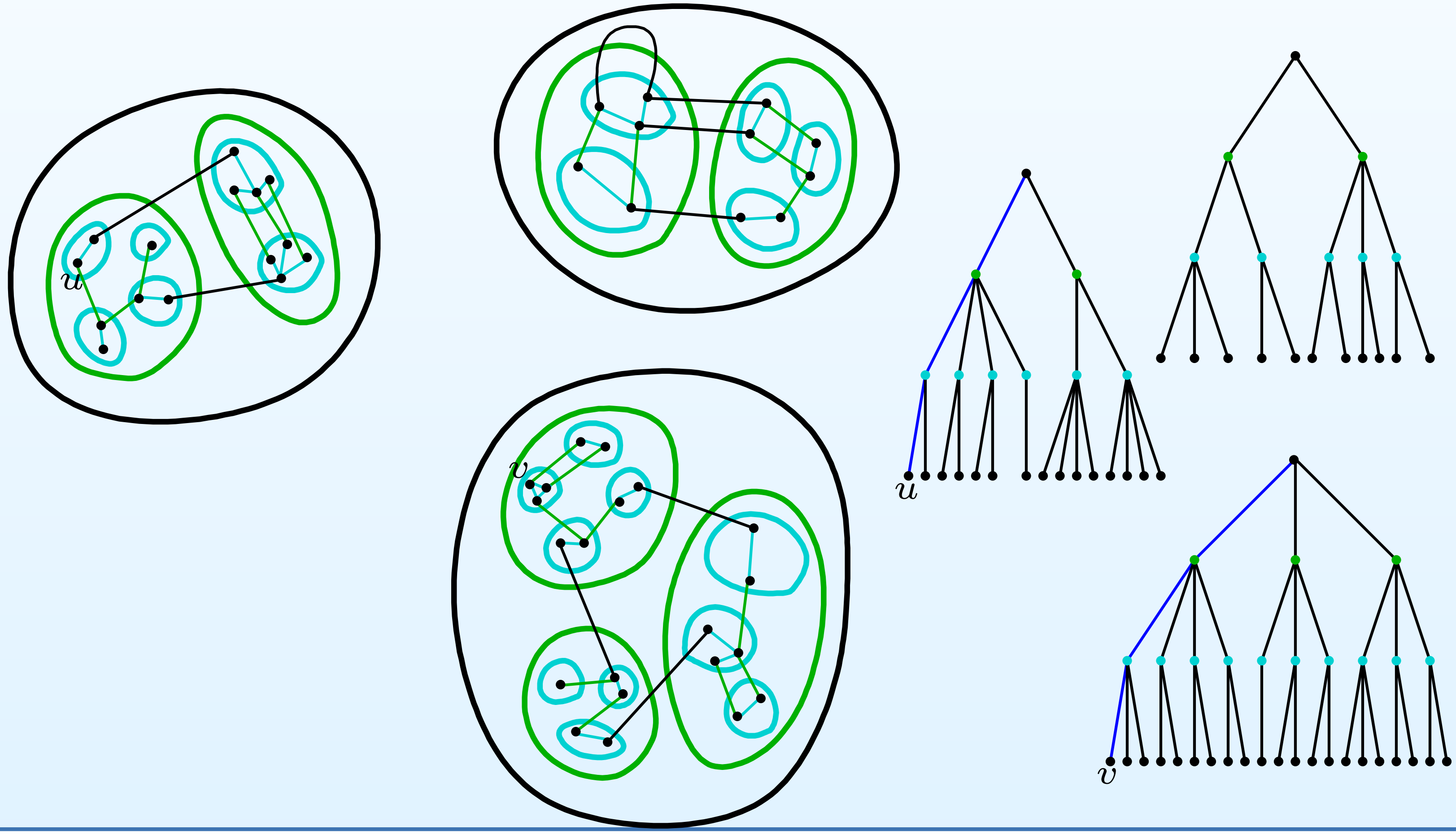
- To determine if vertices u and v are connected in G, traverse the leaf-to-root paths from u and v in $\mathcal{C}$
- Then u and v are connected in G iff the roots are the same

Answering Queries

- To determine if vertices u and v are connected in G, traverse the leaf-to-root paths from u and v in $\mathcal{C}$
- Then u and v are connected in G iff the roots are the same
- Query time $O(\log n)$

Answering Queries

- To determine if vertices u and v are connected in G, traverse the leaf-to-root paths from u and v in $\mathcal{C}$
- Then u and v are connected in G iff the roots are the same
- Query time $O(\log n)$



Handling `insert`(u, v)

Handling `insert`(u, v)

- Initialize $\ell(u, v) \leftarrow 0$

Handling `insert`(u, v)

- Initialize $\ell(u, v) \leftarrow 0$
- r_u, r_v: roots of trees of $\mathcal{C}$ containing u and v, respectively

Handling $\texttt{insert}(u, v)$

- Initialize $\ell(u, v) \leftarrow 0$
- r_u, r_v: roots of trees of $\mathcal{C}$ containing u and v, respectively
- If $r_u = r_v$, $\mathcal{C}$ is not changed

Handling `insert`(u, v)

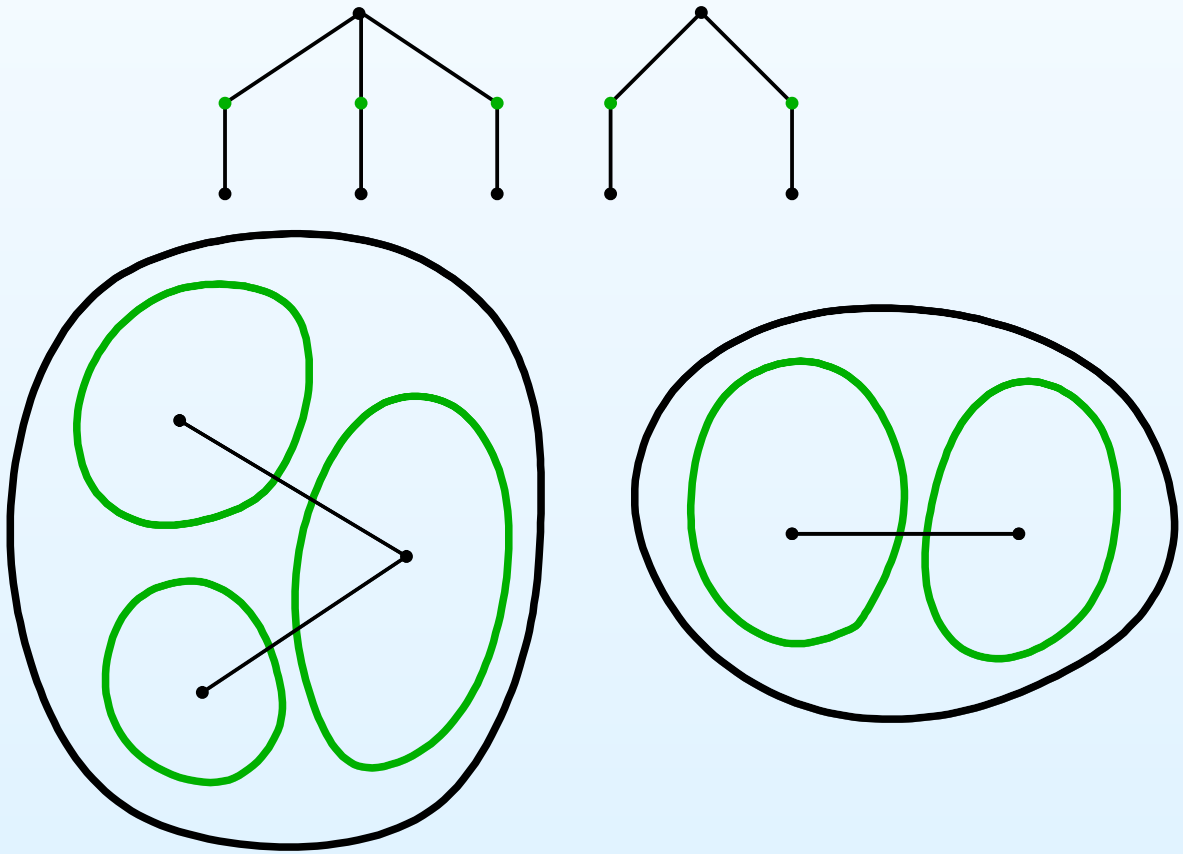
- Initialize $\ell(u, v) \leftarrow 0$
- r_u, r_v: roots of trees of $\mathcal{C}$ containing u and v, respectively
- If $r_u = r_v$, $\mathcal{C}$ is not changed
- Otherwise, r_u and r_v are *merged*

Handling `insert`(u, v)

- Initialize $\ell(u, v) \leftarrow 0$
- r_u, r_v: roots of trees of $\mathcal{C}$ containing u and v, respectively
- If $r_u = r_v$, $\mathcal{C}$ is not changed
- Otherwise, r_u and r_v are *merged*
- This corresponds to merging $C(r_u)$ and $C(r_v)$

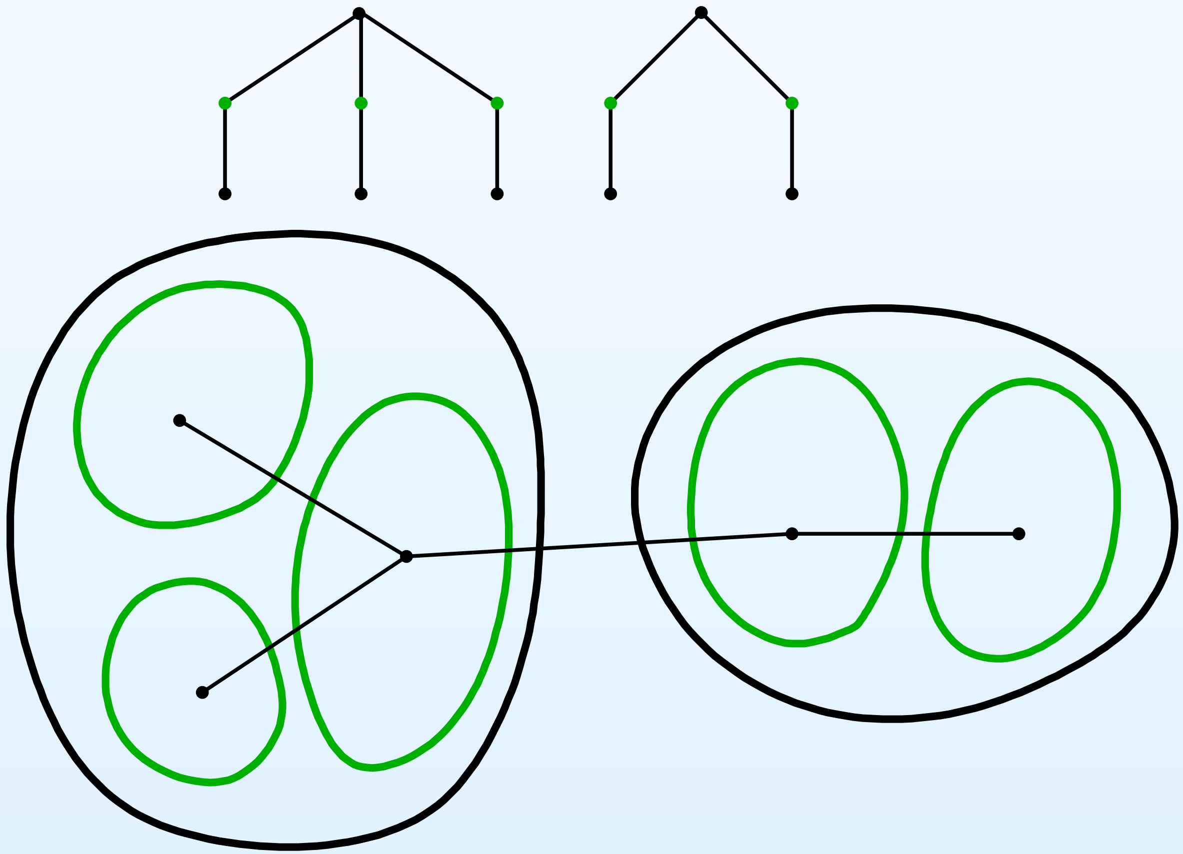
Handling `insert`(u, v)

- Initialize $\ell(u, v) \leftarrow 0$
- r_u, r_v: roots of trees of $\mathcal{C}$ containing u and v, respectively
- If $r_u = r_v$, $\mathcal{C}$ is not changed
- Otherwise, r_u and r_v are *merged*
- This corresponds to merging $C(r_u)$ and $C(r_v)$



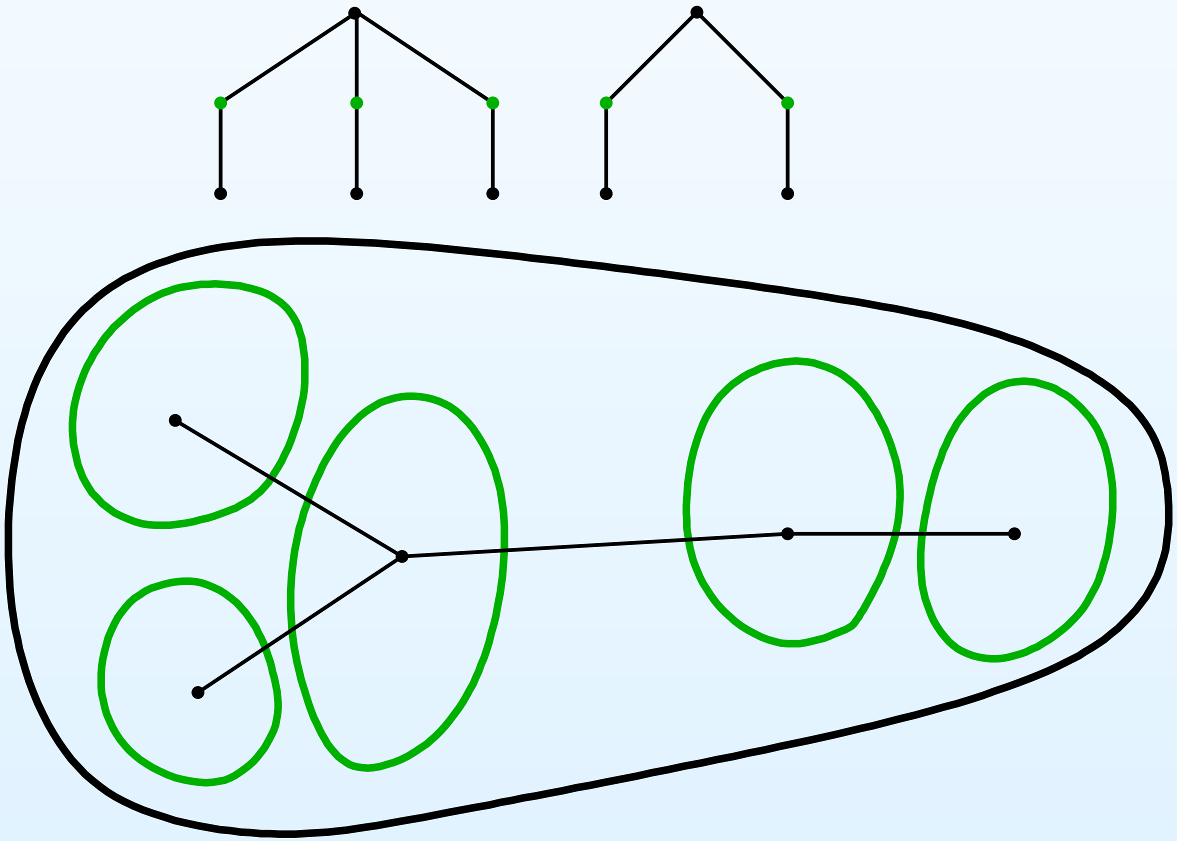
Handling `insert`(u, v)

- Initialize $\ell(u, v) \leftarrow 0$
- r_u, r_v: roots of trees of $\mathcal{C}$ containing u and v, respectively
- If $r_u = r_v$, $\mathcal{C}$ is not changed
- Otherwise, r_u and r_v are *merged*
- This corresponds to merging $C(r_u)$ and $C(r_v)$



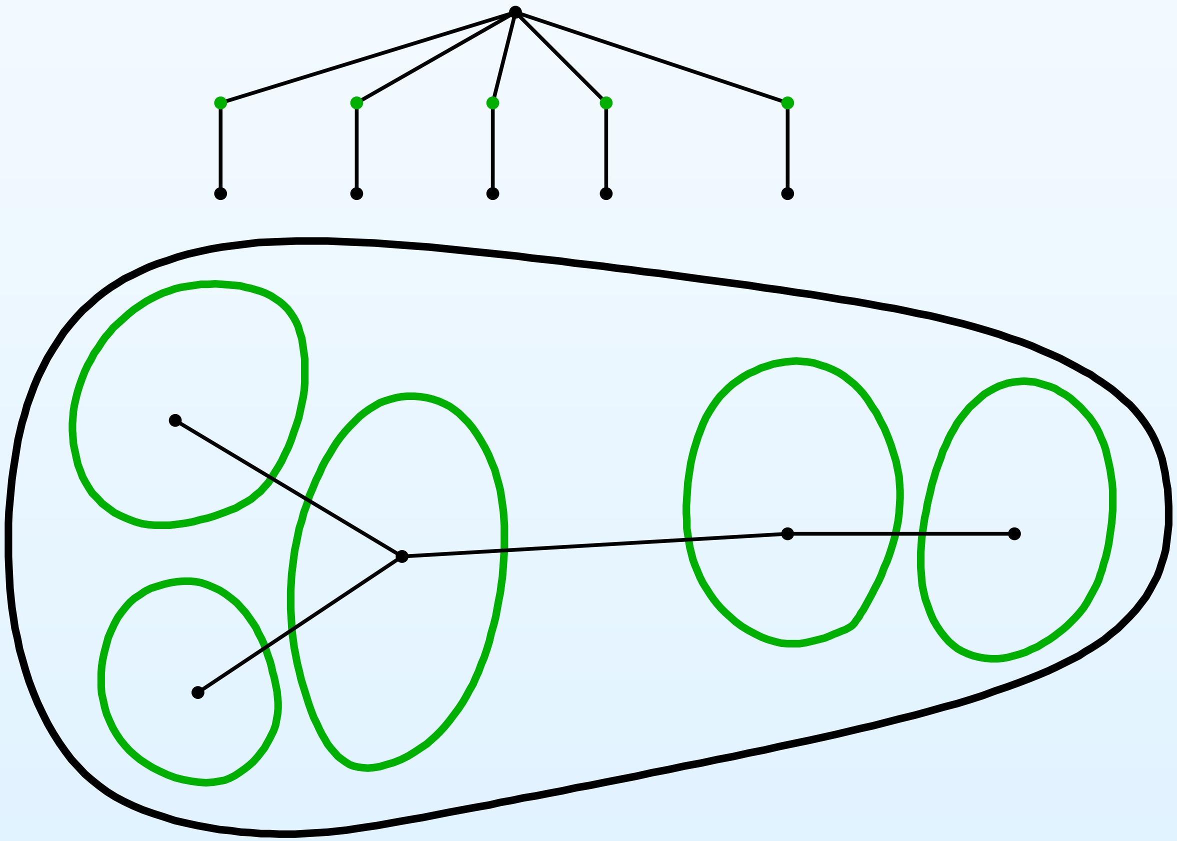
Handling `insert`(u, v)

- Initialize $\ell(u, v) \leftarrow 0$
- r_u, r_v: roots of trees of $\mathcal{C}$ containing u and v, respectively
- If $r_u = r_v$, $\mathcal{C}$ is not changed
- Otherwise, r_u and r_v are *merged*
- This corresponds to merging $C(r_u)$ and $C(r_v)$



Handling `insert`(u, v)

- Initialize $\ell(u, v) \leftarrow 0$
- r_u, r_v: roots of trees of $\mathcal{C}$ containing u and v, respectively
- If $r_u = r_v$, $\mathcal{C}$ is not changed
- Otherwise, r_u and r_v are *merged*
- This corresponds to merging $C(r_u)$ and $C(r_v)$



Handling `delete`(u, v)

Handling `delete`(u, v)

- Let $i = \ell(u, v)$ and let C_u and C_v be the $(i + 1)$-clusters containing u and v

Handling `delete`(u, v)

- Let $i = \ell(u, v)$ and let C_u and C_v be the $(i + 1)$-clusters containing u and v
- Assume $C_u \neq C_v$ since otherwise, $\mathcal{C}$ is not changed

Handling `delete`(u, v)

- Let $i = \ell(u, v)$ and let C_u and C_v be the $(i+1)$-clusters containing u and v
- Assume $C_u \neq C_v$ since otherwise, $\mathcal{C}$ is not changed
- Let M_i be the multigraph with $(i+1)$-clusters as vertices and level i-edges of G as edges

Handling `delete`(u, v)

- Let $i = \ell(u, v)$ and let C_u and C_v be the $(i+1)$-clusters containing u and v
- Assume $C_u \neq C_v$ since otherwise, $\mathcal{C}$ is not changed
- Let M_i be the multigraph with $(i+1)$-clusters as vertices and level i-edges of G as edges
- In M_i, execute two standard search procedures in parallel, one starting in C_u, the other starting in C_v

Handling `delete`(u, v)

- Let $i = \ell(u, v)$ and let C_u and C_v be the $(i + 1)$-clusters containing u and v
- Assume $C_u \neq C_v$ since otherwise, $\mathcal{C}$ is not changed
- Let M_i be the multigraph with $(i + 1)$-clusters as vertices and level i-edges of G as edges
- In M_i, execute two standard search procedures in parallel, one starting in C_u, the other starting in C_v
- Terminate both procedures when in one of the following two cases:

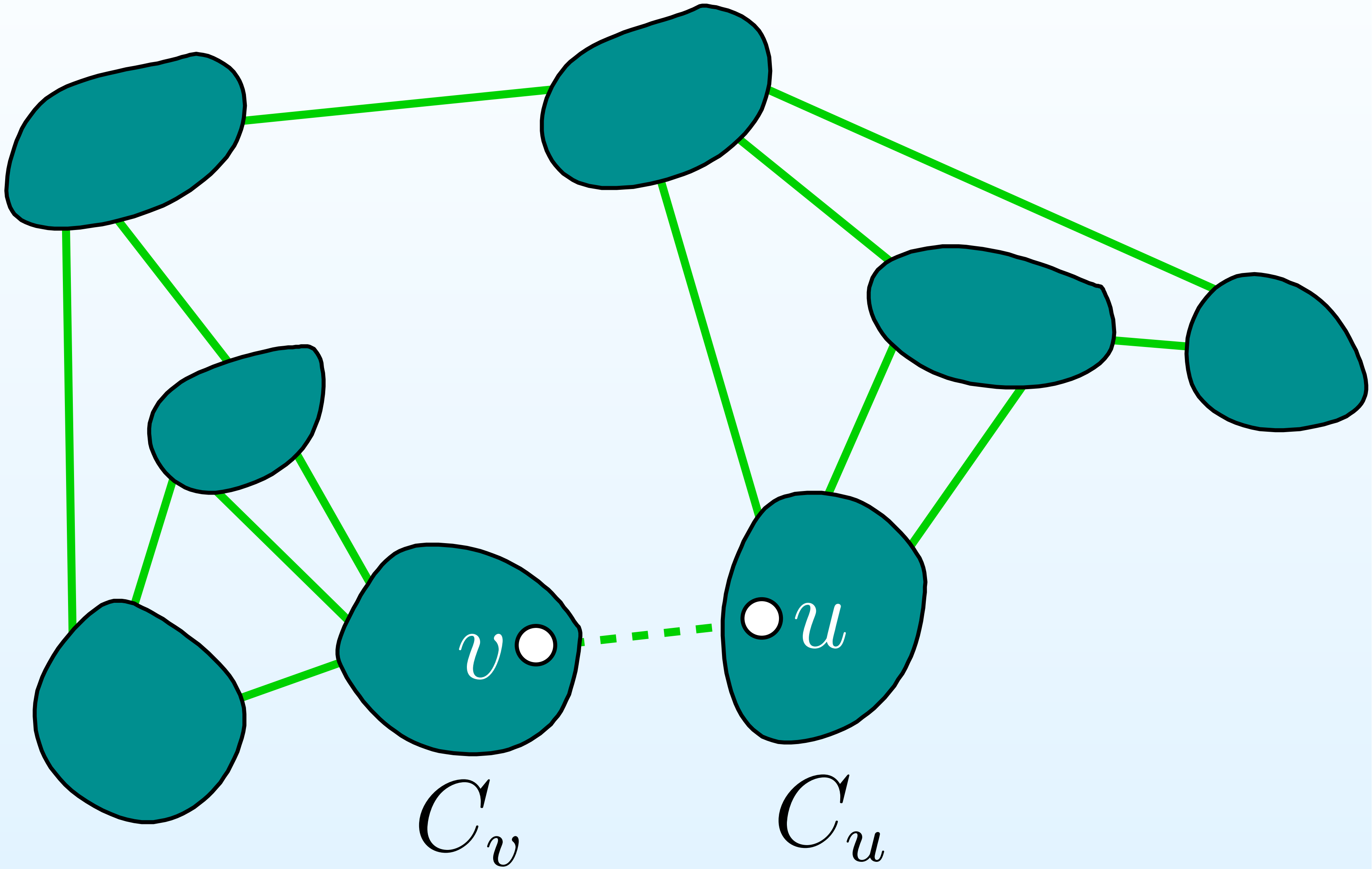
Handling `delete`(u, v)

- Let $i = \ell(u, v)$ and let C_u and C_v be the $(i + 1)$-clusters containing u and v
- Assume $C_u \neq C_v$ since otherwise, $\mathcal{C}$ is not changed
- Let M_i be the multigraph with $(i + 1)$-clusters as vertices and level i-edges of G as edges
- In M_i, execute two standard search procedures in parallel, one starting in C_u, the other starting in C_v
- Terminate both procedures when in one of the following two cases:
    - a vertex of M_i is explored by both search procedures

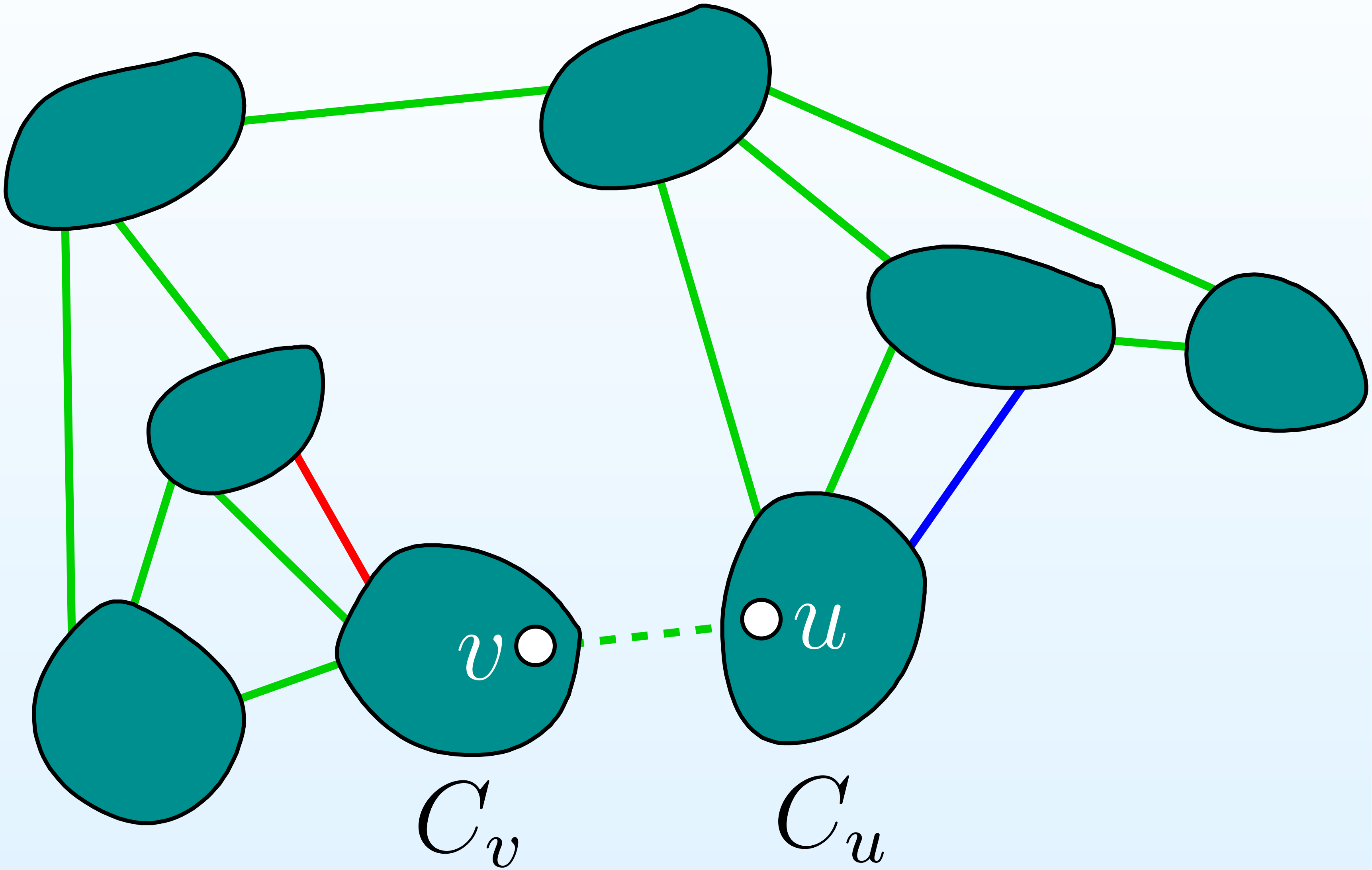
Handling `delete`(u, v)

- Let $i = \ell(u, v)$ and let C_u and C_v be the $(i+1)$-clusters containing u and v
- Assume $C_u \neq C_v$ since otherwise, $\mathcal{C}$ is not changed
- Let M_i be the multigraph with $(i+1)$-clusters as vertices and level i-edges of G as edges
- In M_i, execute two standard search procedures in parallel, one starting in C_u, the other starting in C_v
- Terminate both procedures when in one of the following two cases:
    - a vertex of M_i is explored by both search procedures
    - one of the search procedures has no more edges to explore

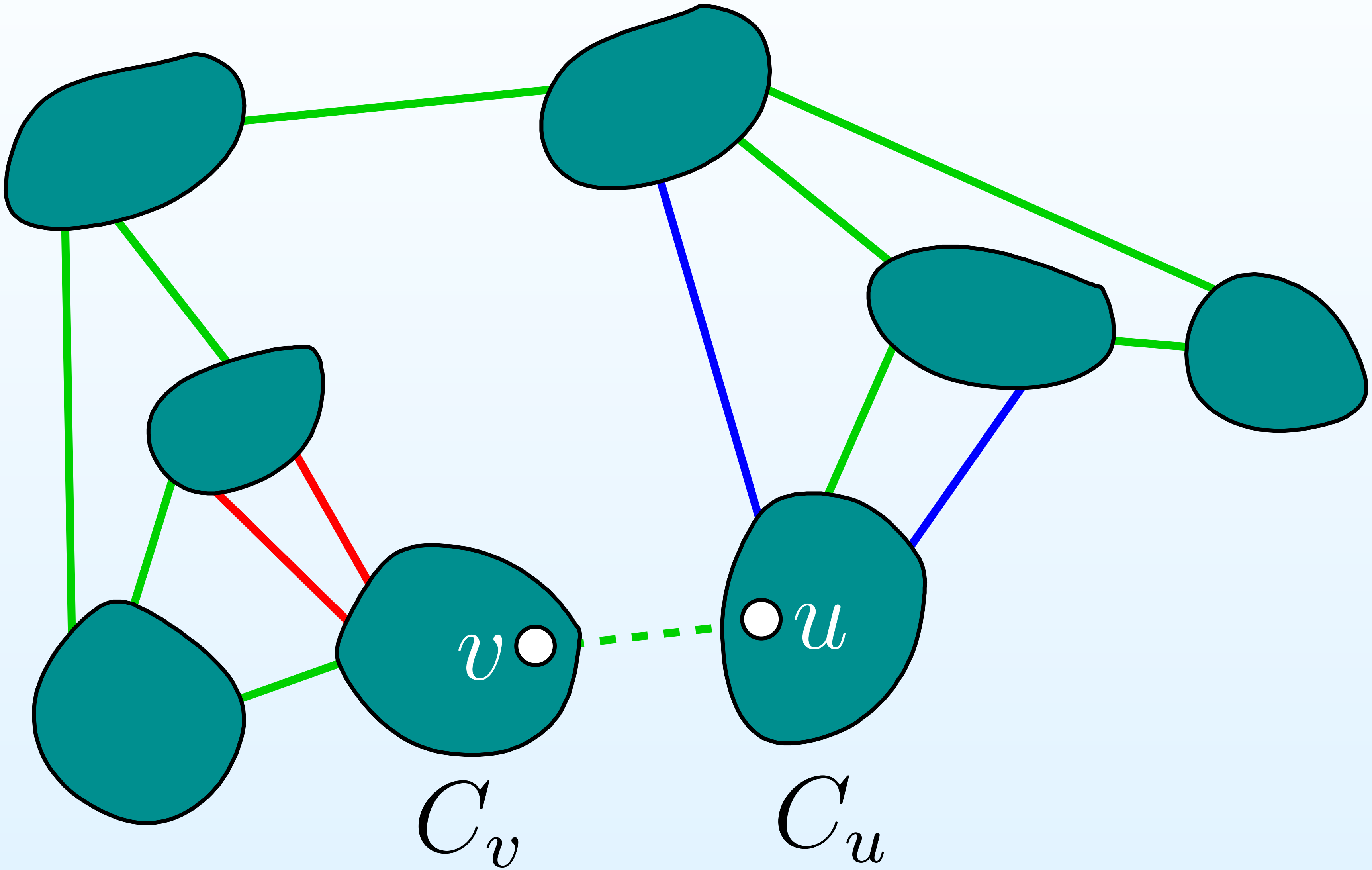
A vertex of M_i is explored by both search procedures



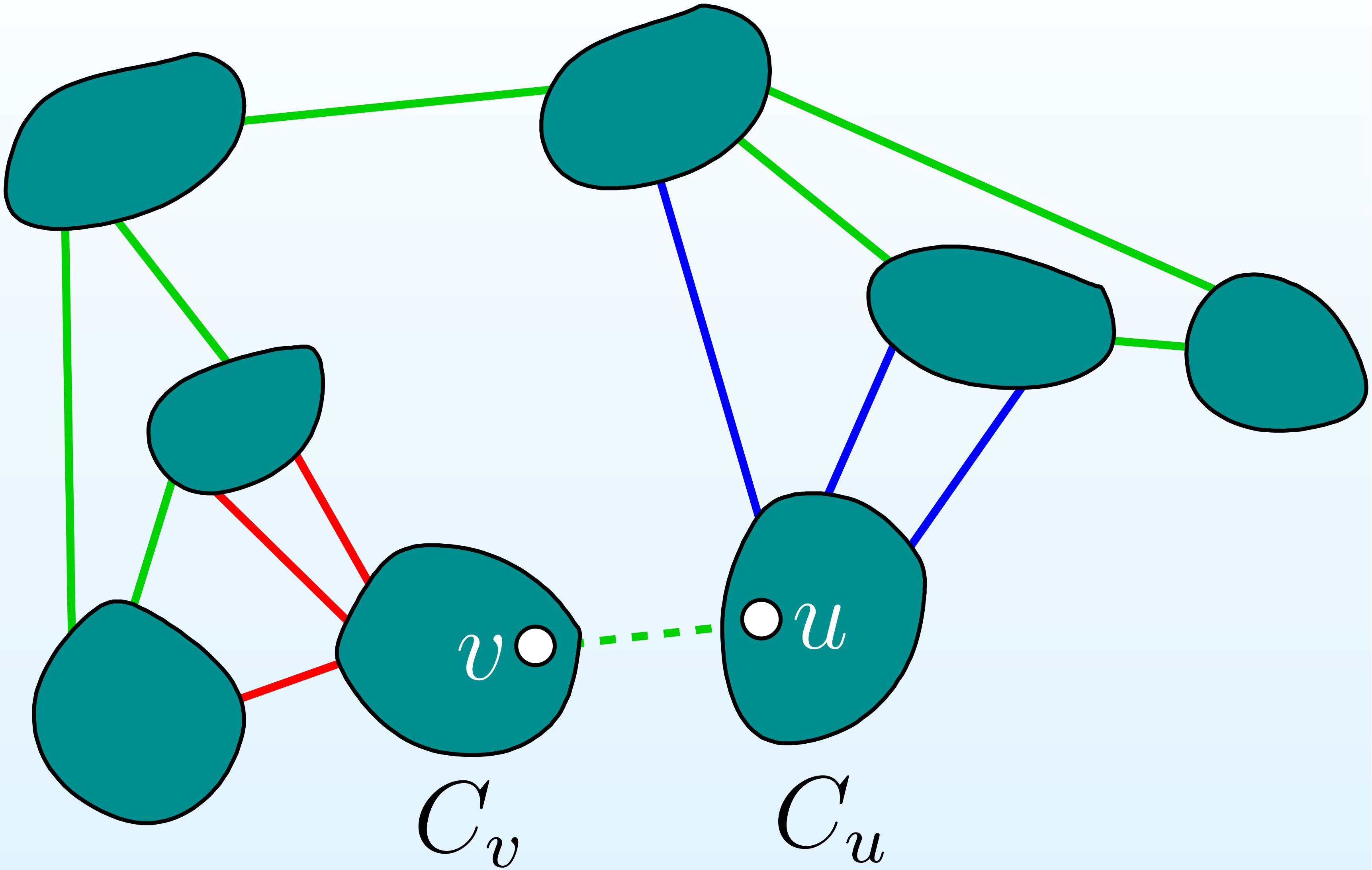
A vertex of M_i is explored by both search procedures



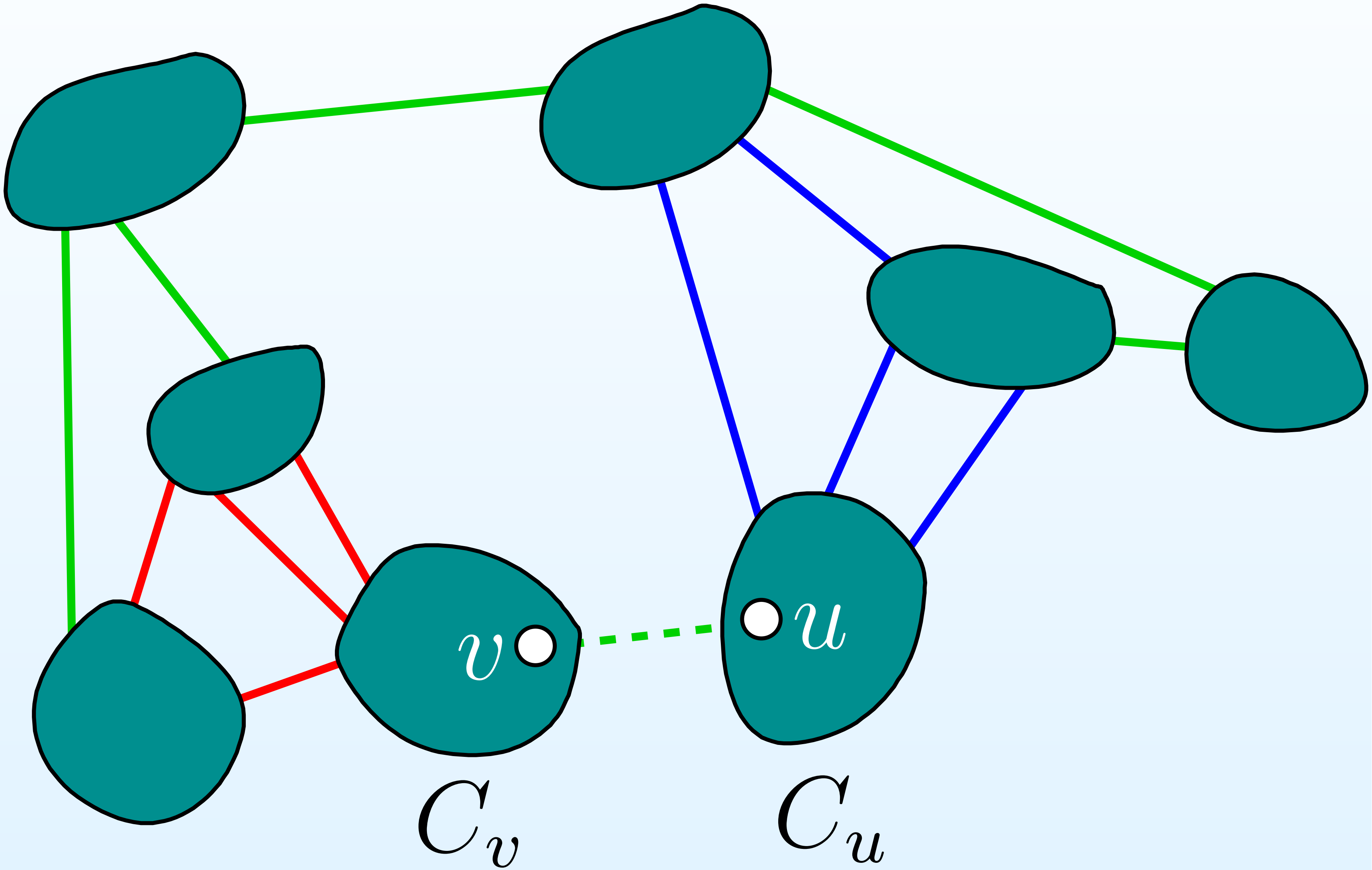
A vertex of M_i is explored by both search procedures



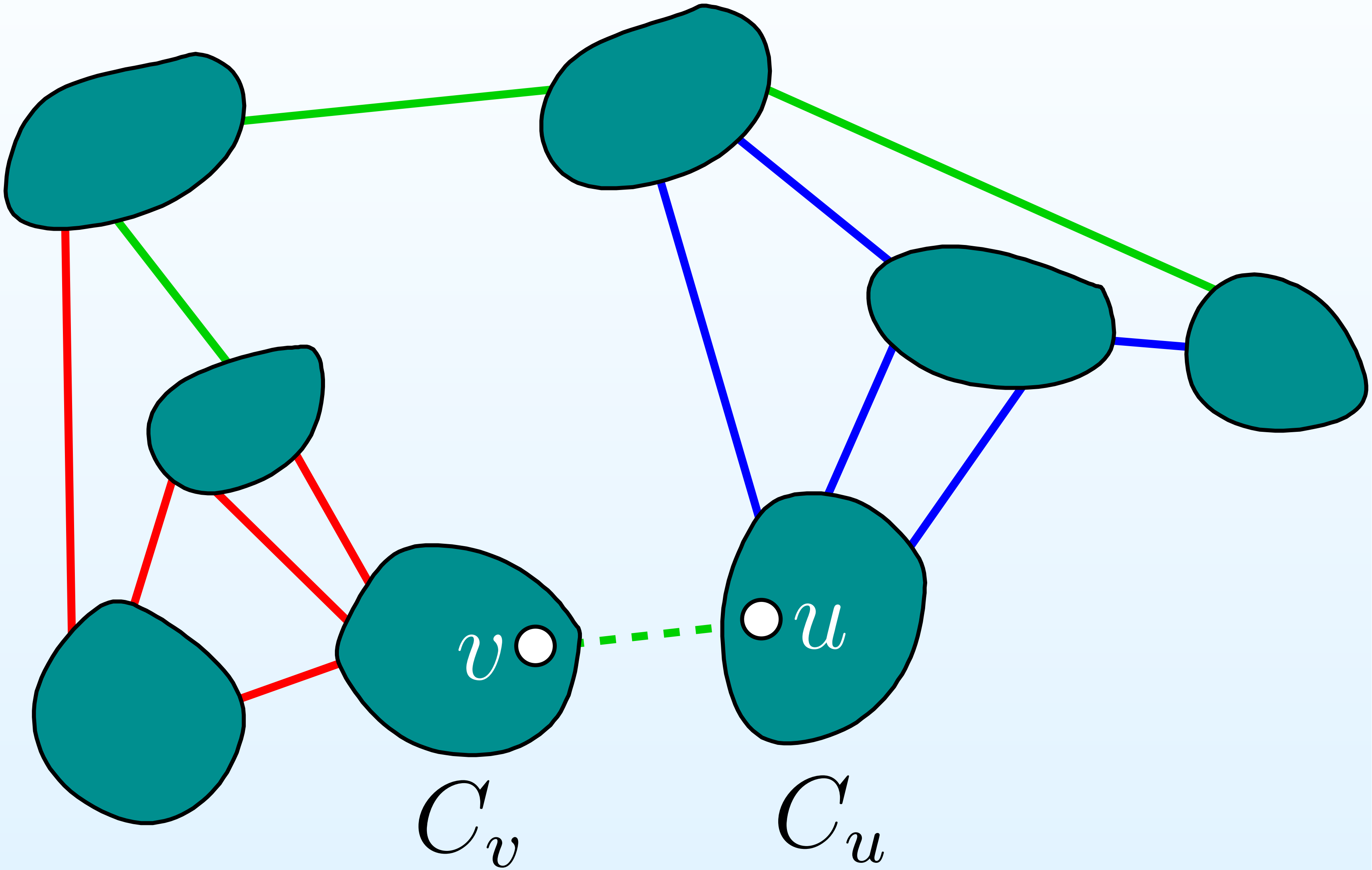
A vertex of M_i is explored by both search procedures



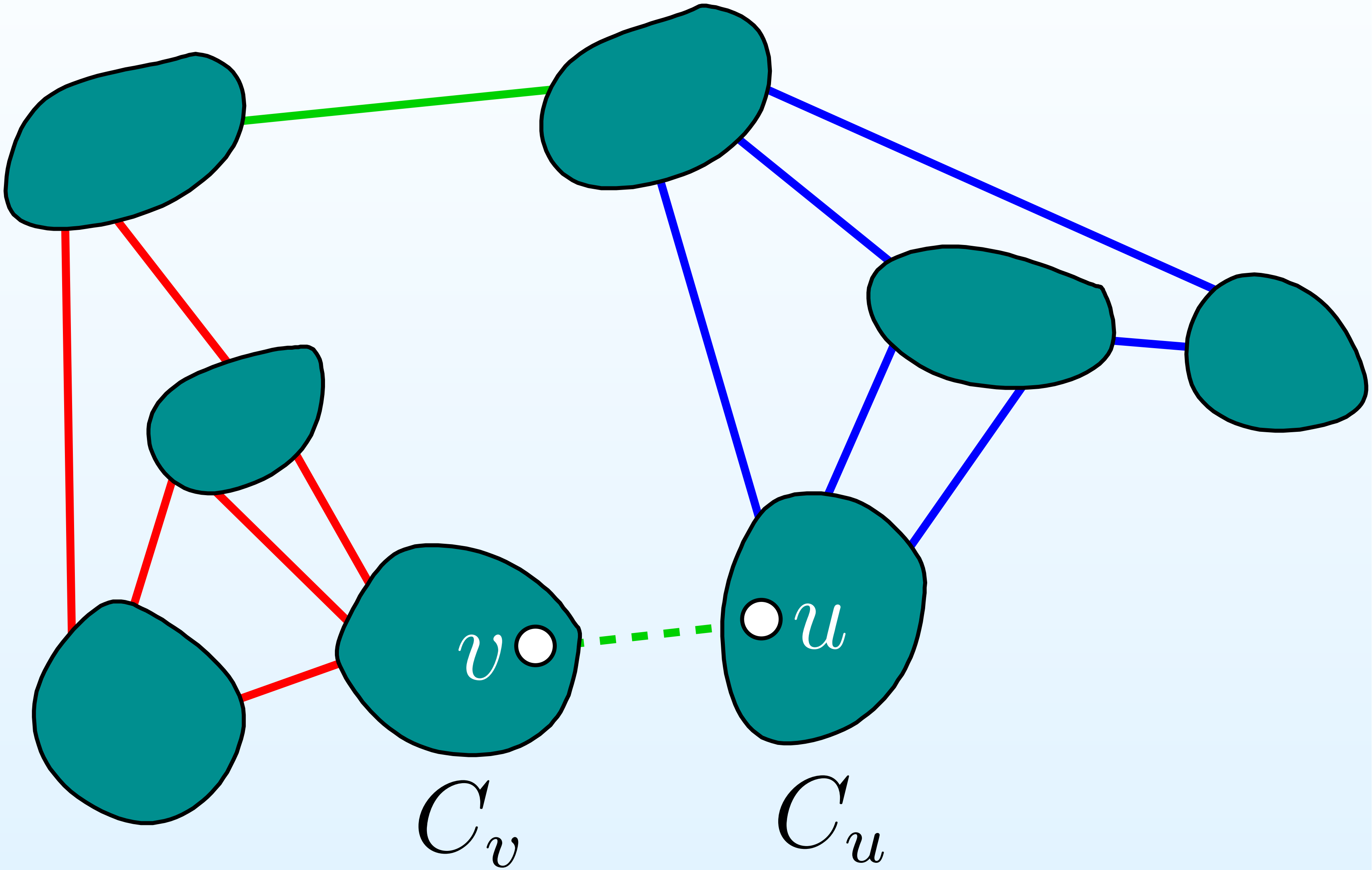
A vertex of M_i is explored by both search procedures



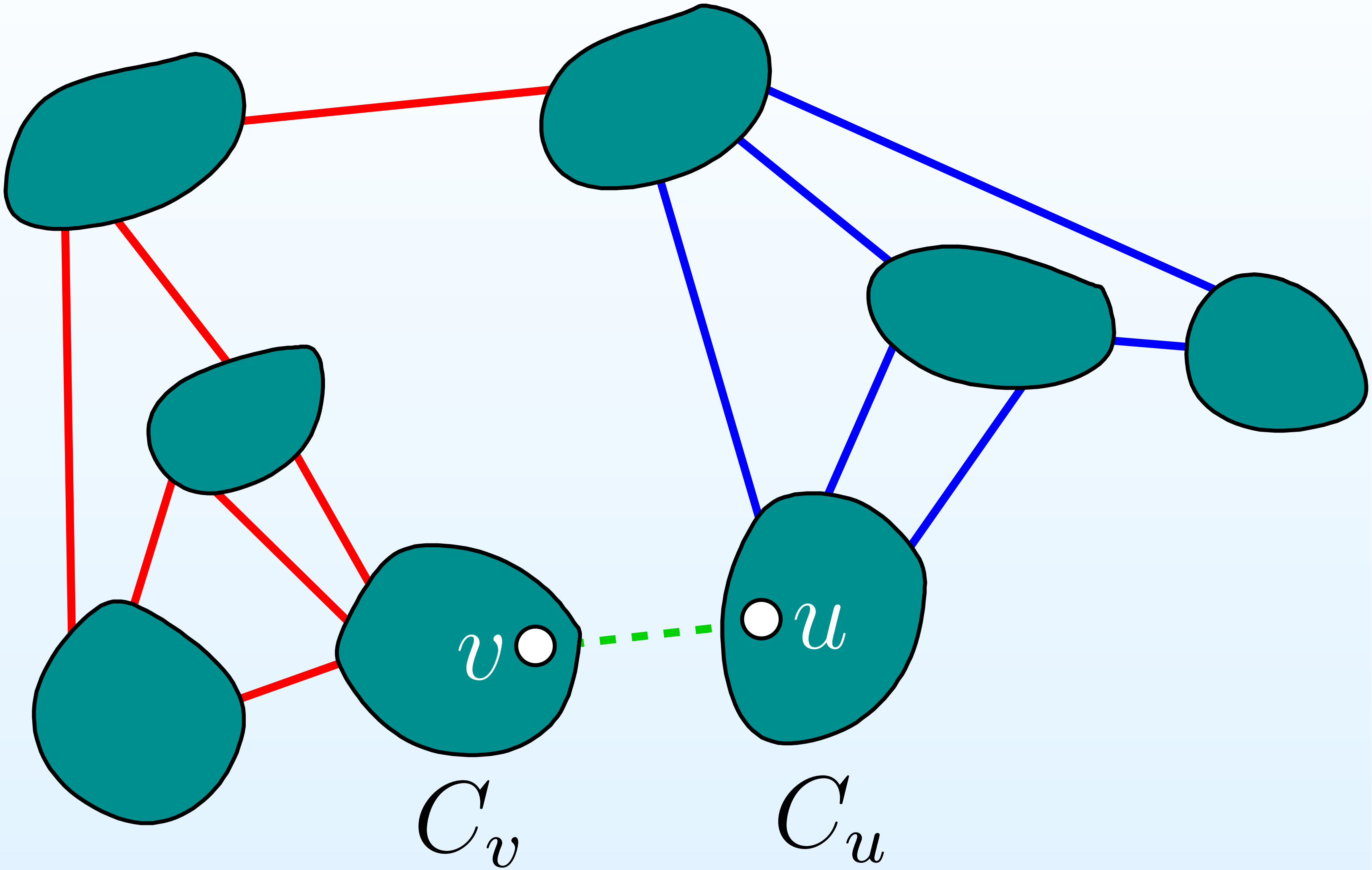
A vertex of M_i is explored by both search procedures



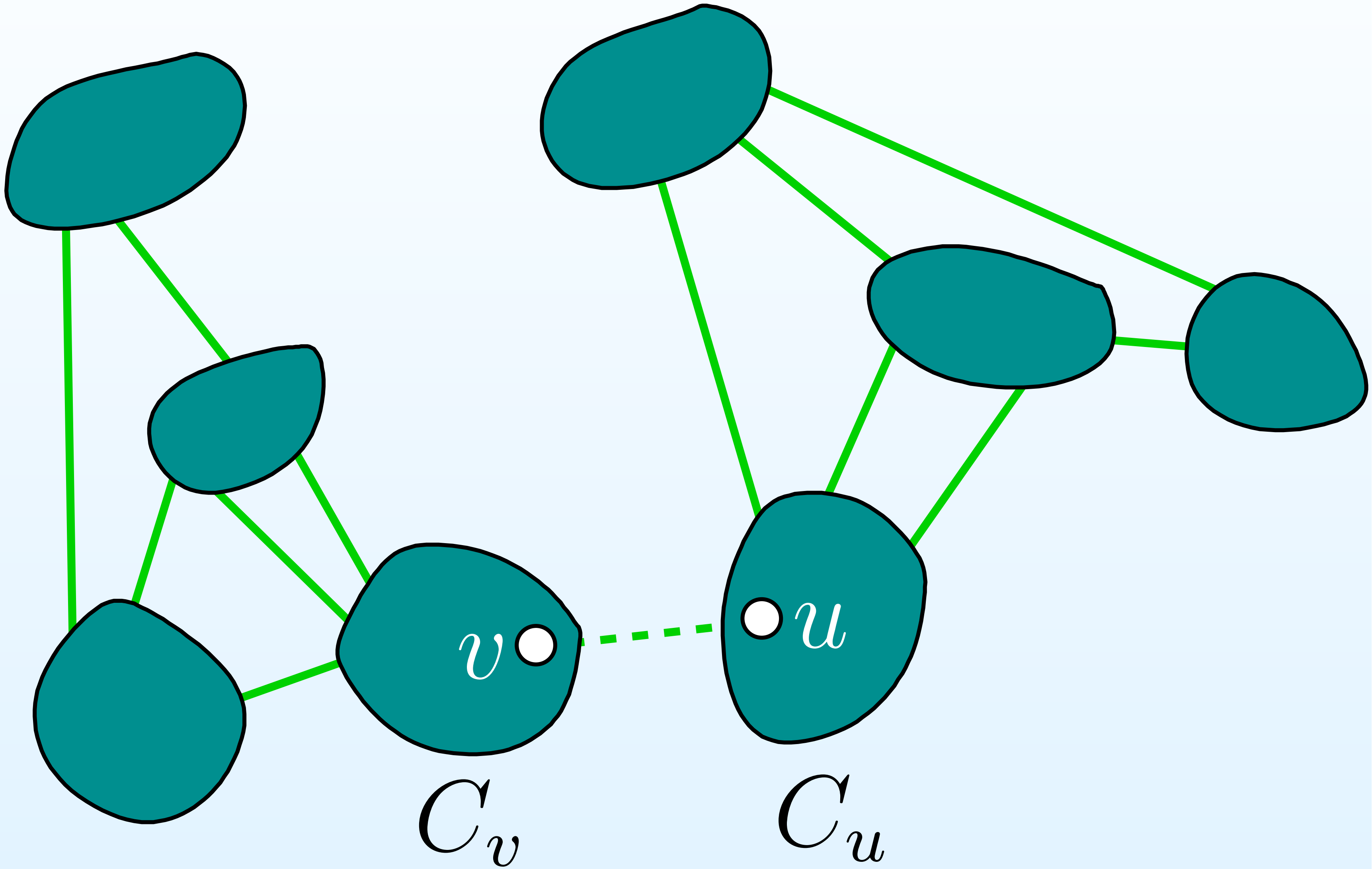
A vertex of M_i is explored by both search procedures



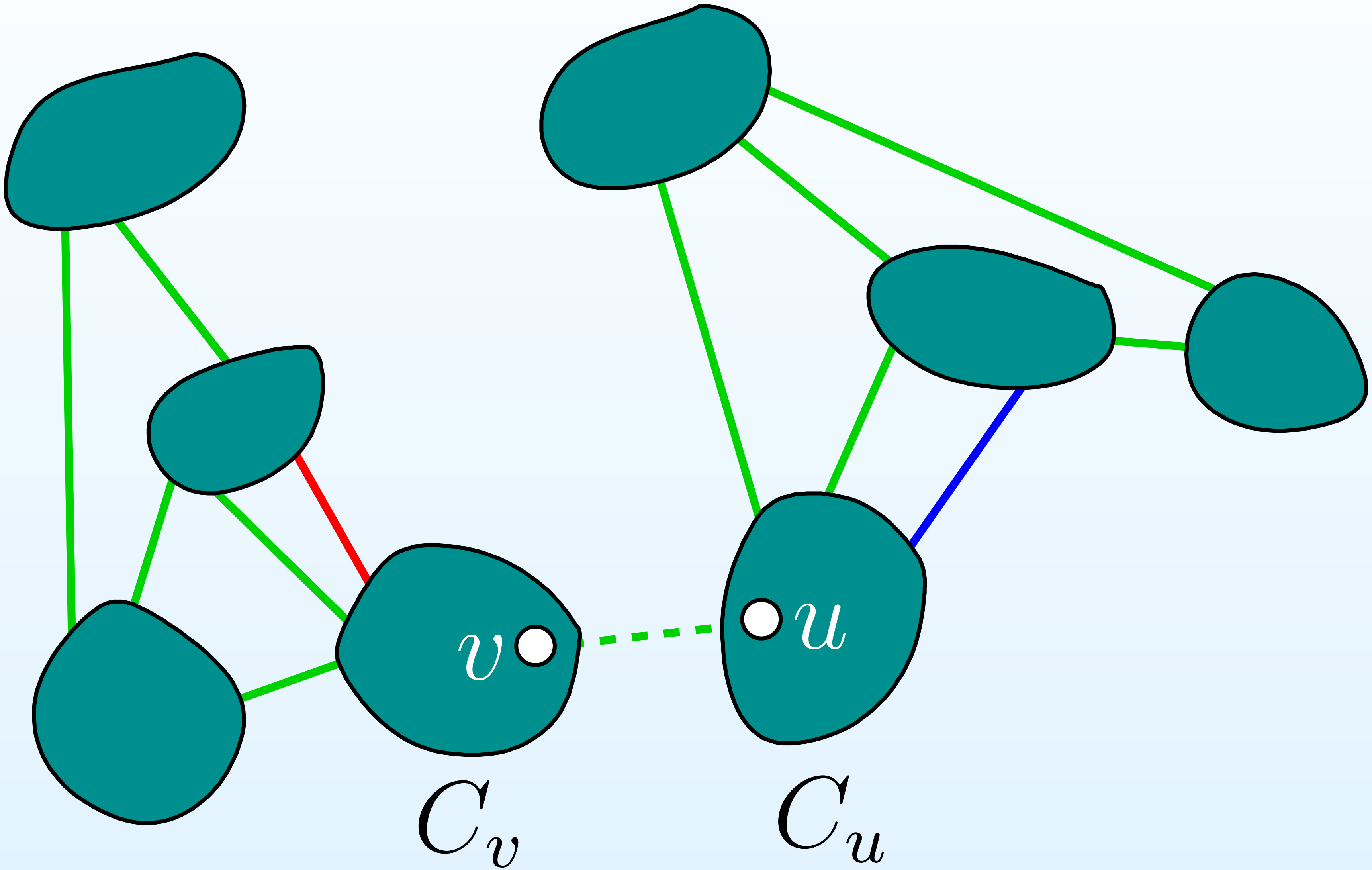
A vertex of M_i is explored by both search procedures



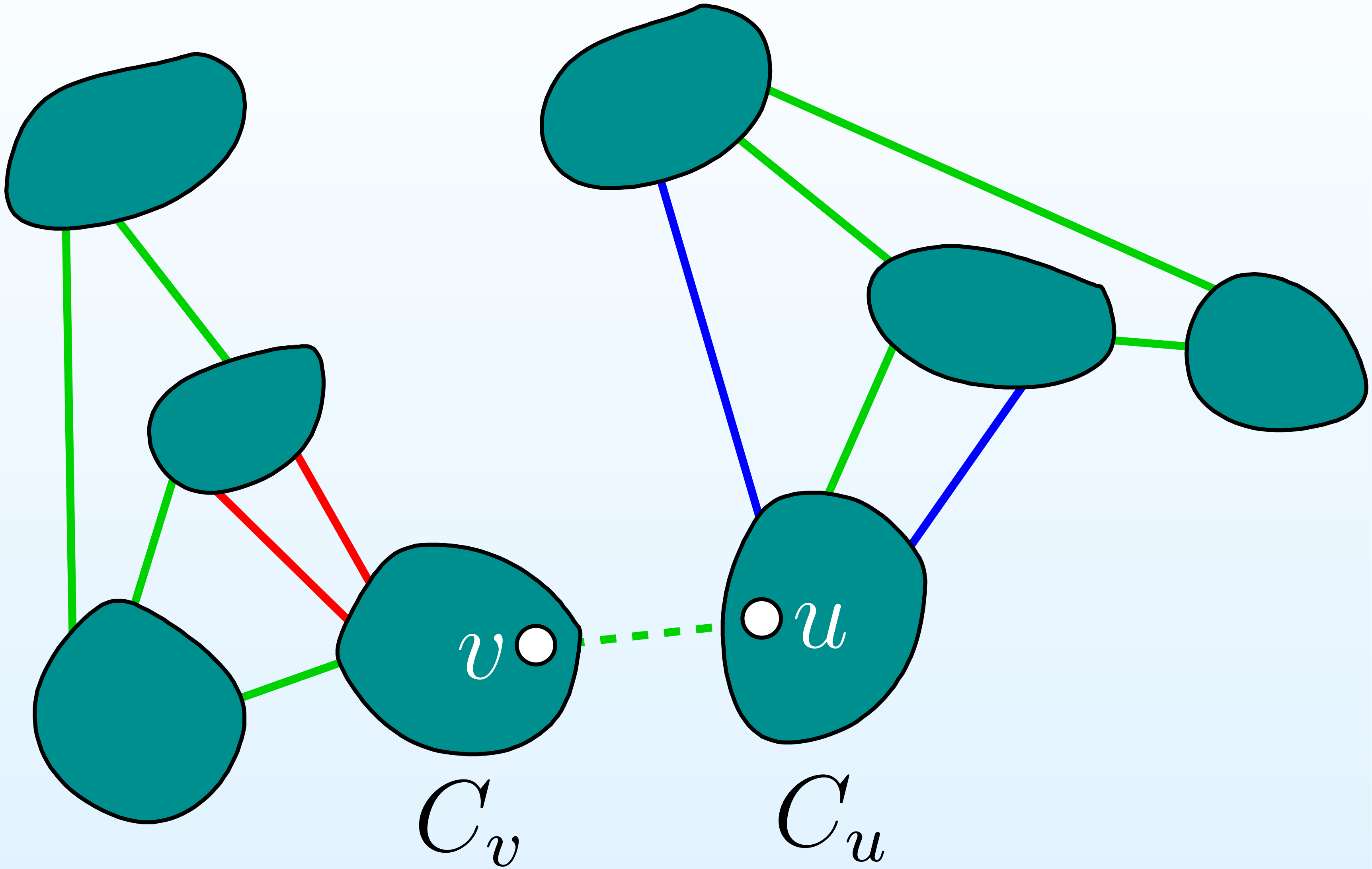
A search procedure has no more edges to explore



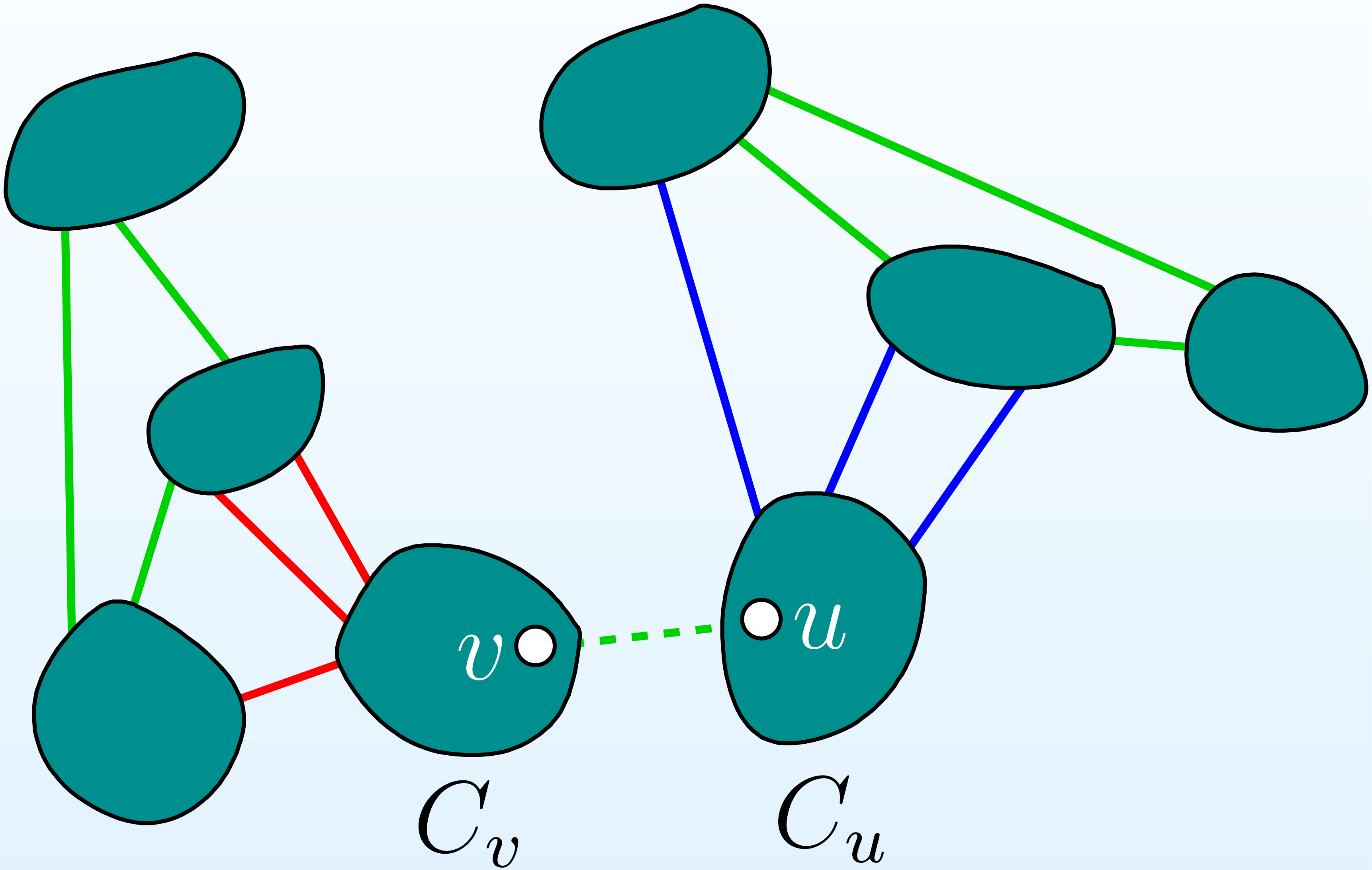
A search procedure has no more edges to explore



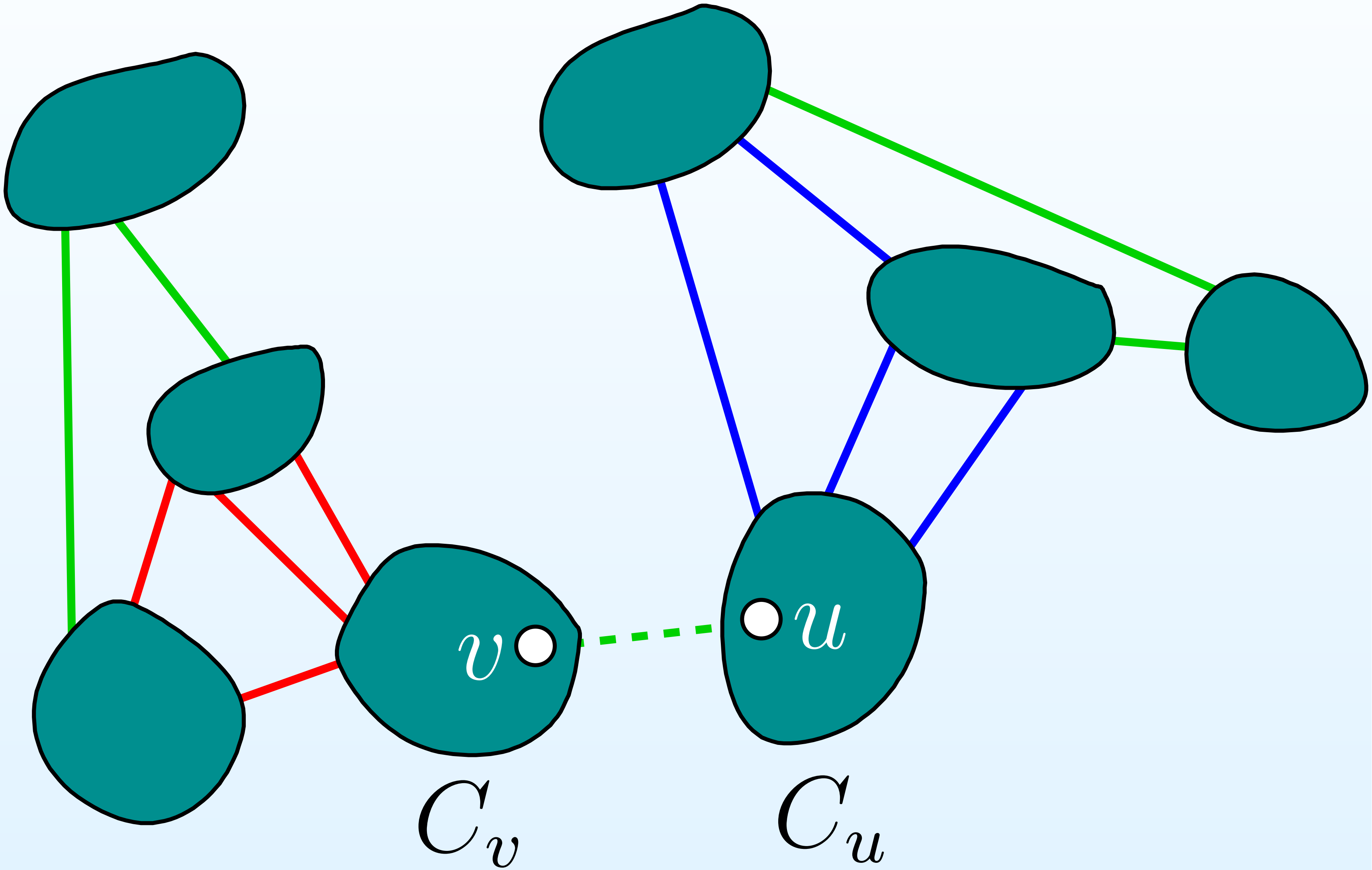
A search procedure has no more edges to explore



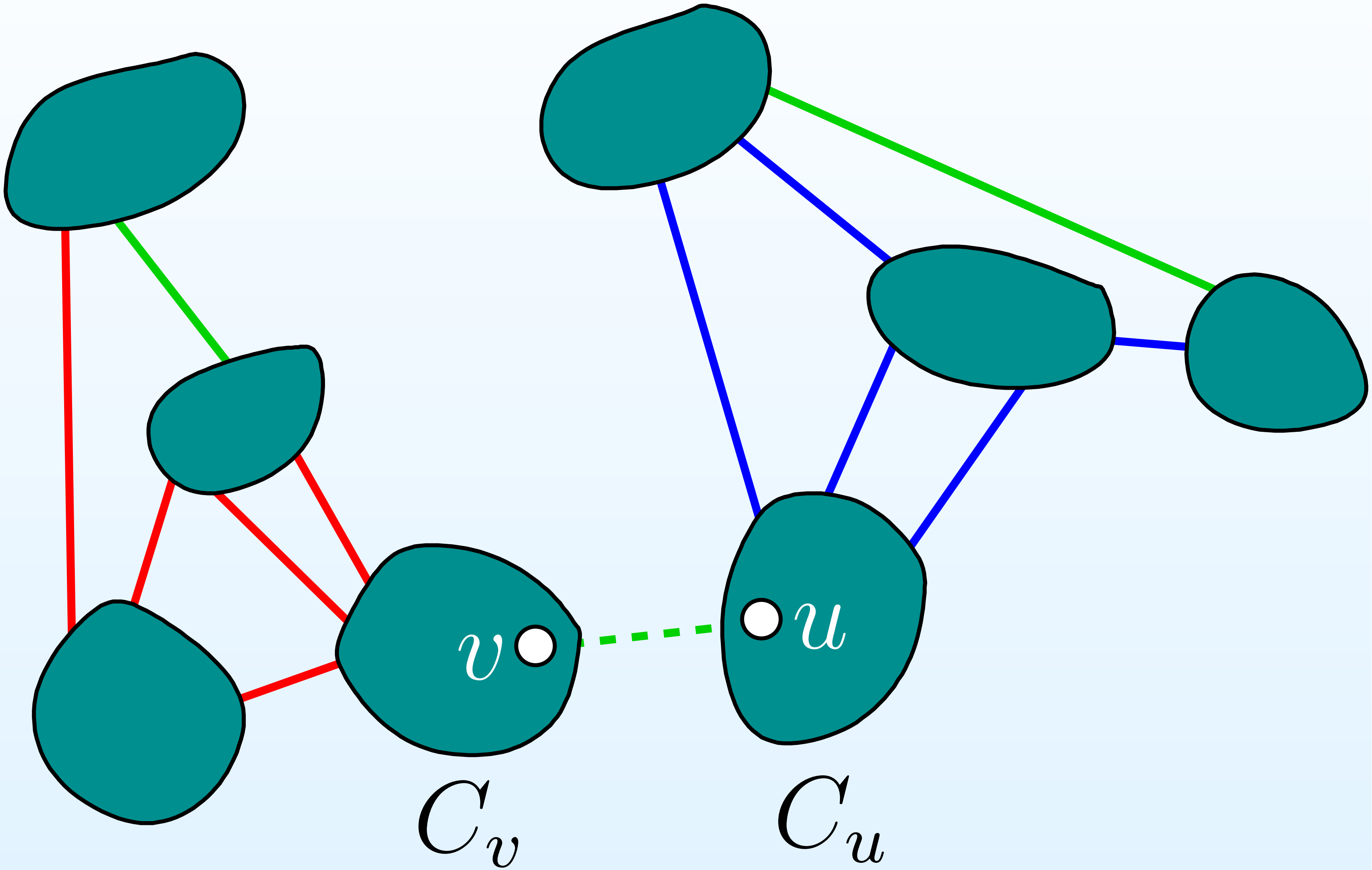
A search procedure has no more edges to explore



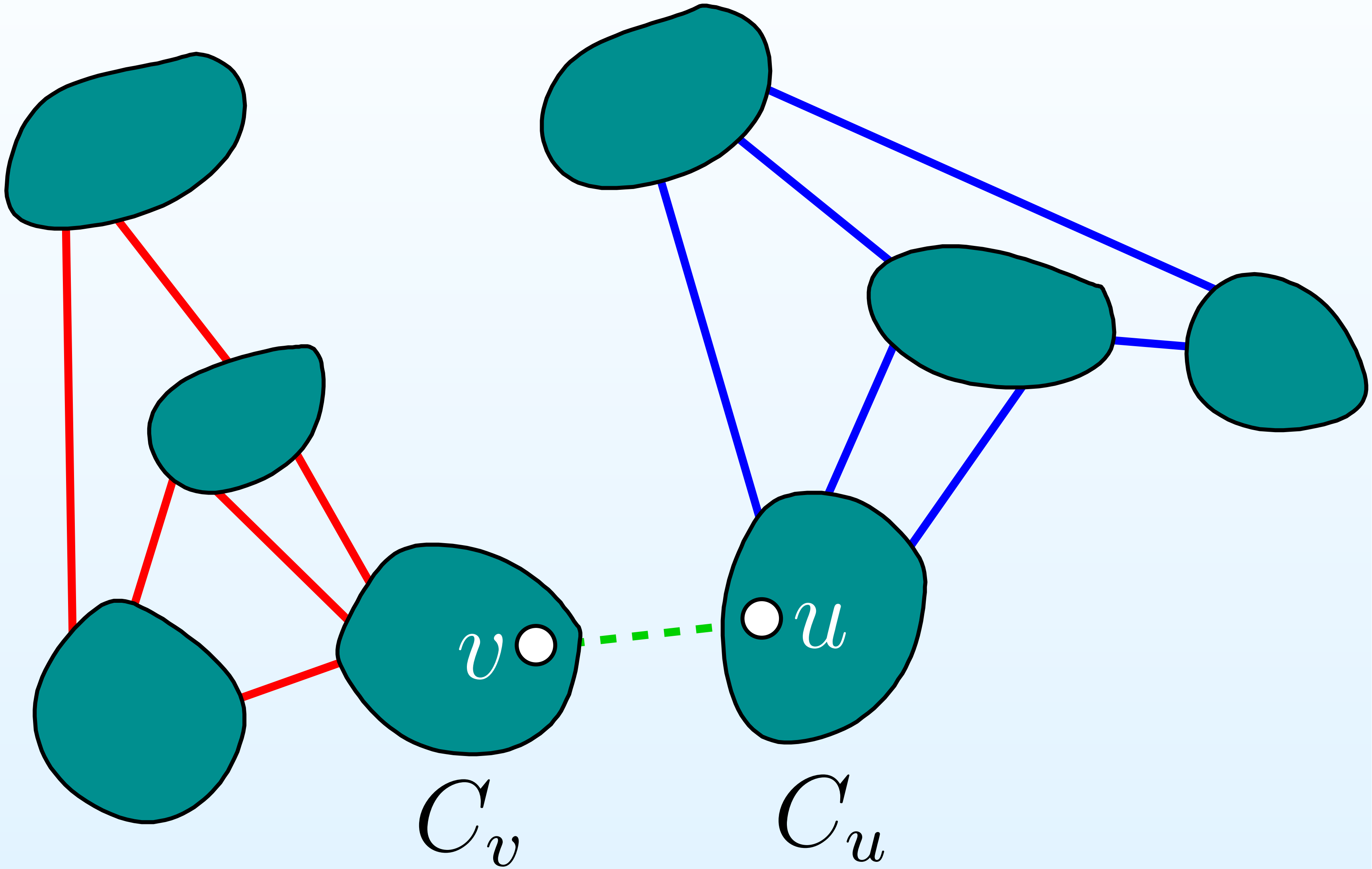
A search procedure has no more edges to explore



A search procedure has no more edges to explore



A search procedure has no more edges to explore

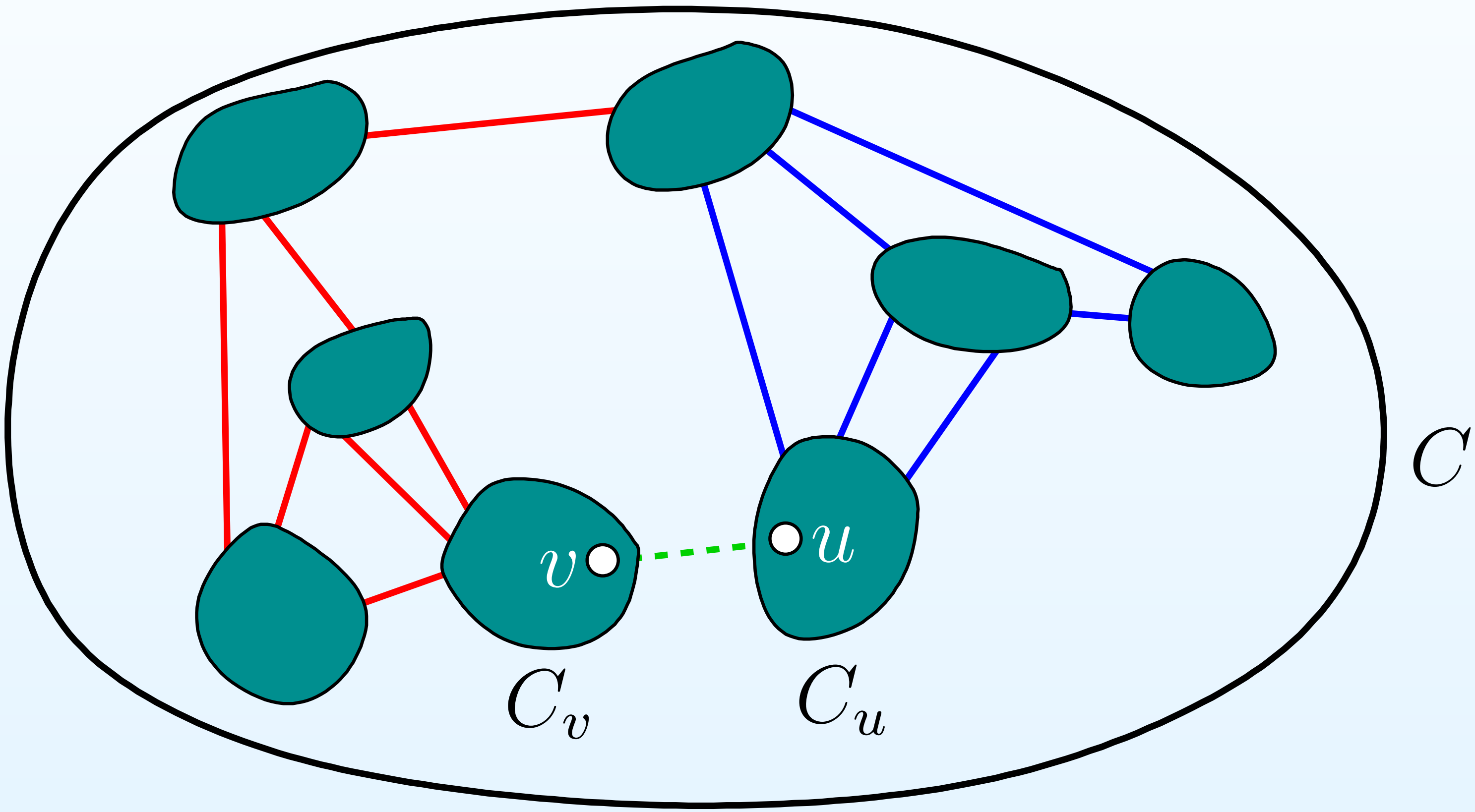


Updates to $\mathcal{C}$

- If the two search procedures meet, the level i-cluster C containing (u, v) is still connected so C remains a level i-cluster

Updates to $\mathcal{C}$

- If the two search procedures meet, the level i-cluster C containing (u, v) is still connected so C remains a level i-cluster



Updates to $\mathcal{C}$

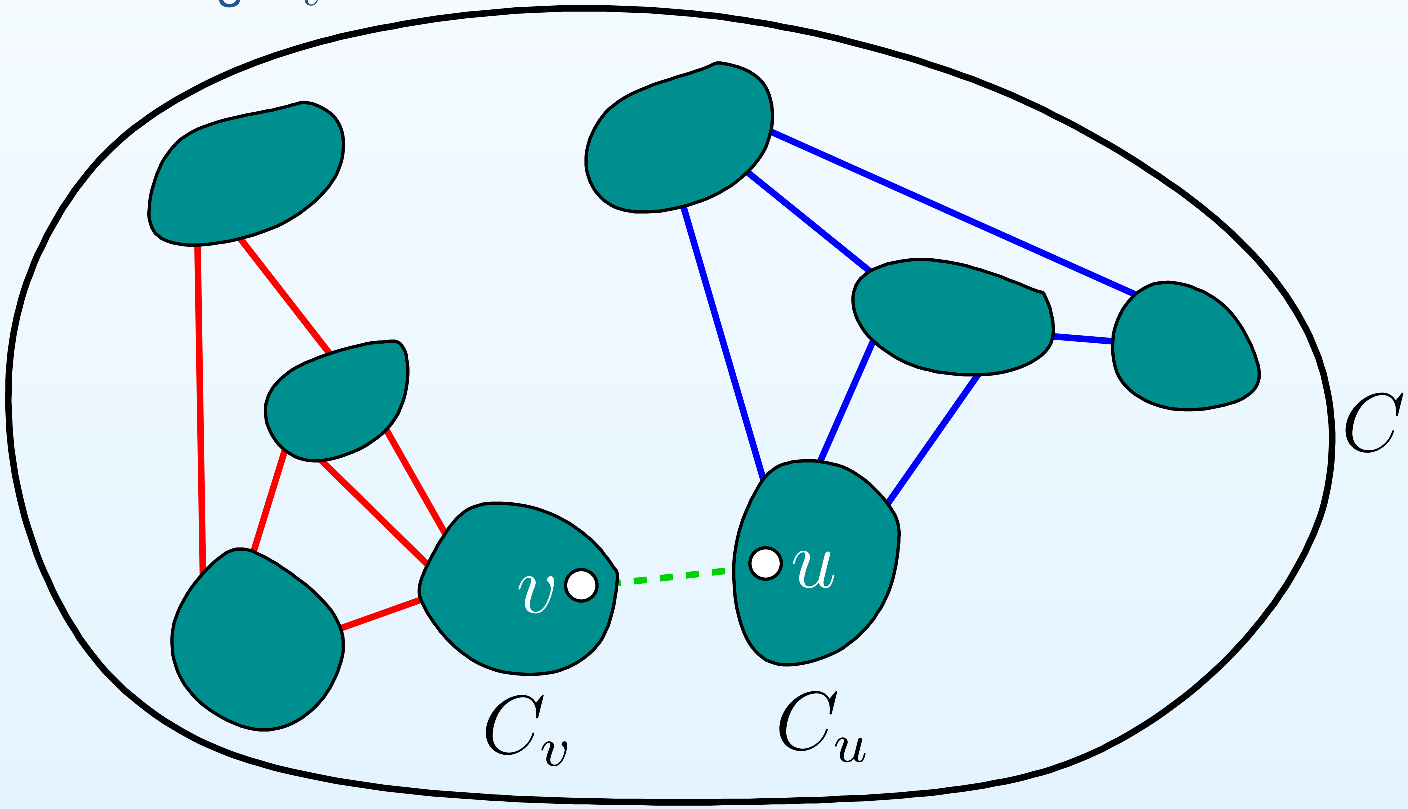
- If the two search procedures meet, the level i-cluster C containing (u, v) is still connected so C remains a level i-cluster

Updates to $\mathcal{C}$

- If the two search procedures meet, the level i-cluster C containing (u, v) is still connected so C remains a level i-cluster
- Otherwise, C is be split in two, one part containing C_u and one containing C_v

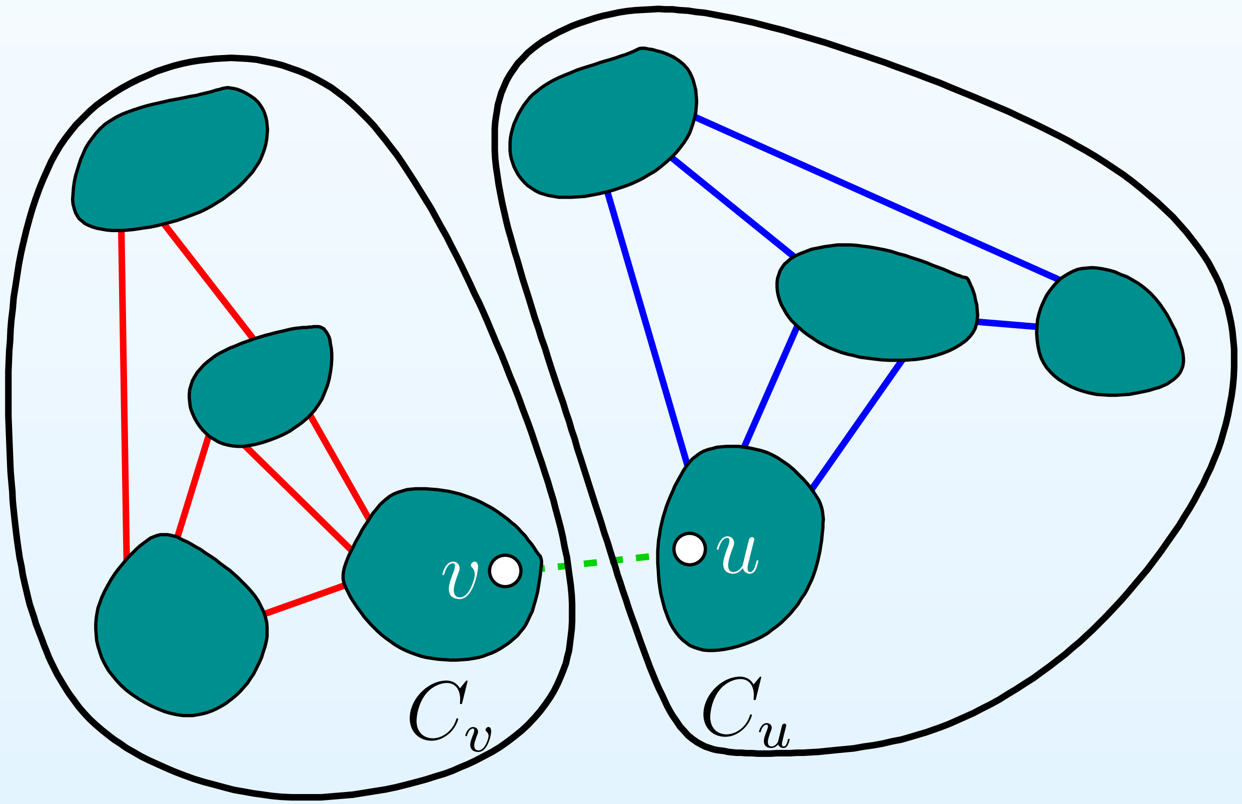
Updates to C

- If the two search procedures meet, the level i-cluster C containing (u, v) is still connected so C remains a level i-cluster
- Otherwise, C is be split in two, one part containing C_u and one containing C_v



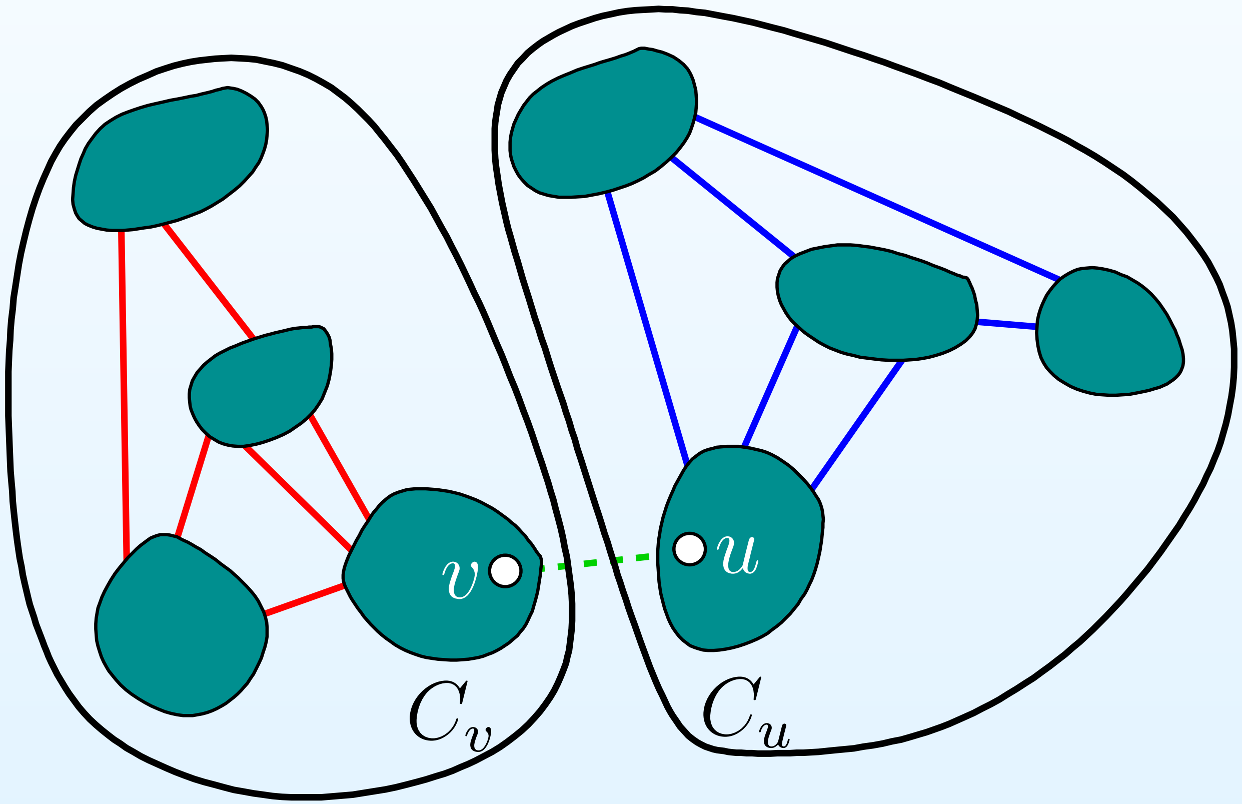
Updates to $\mathcal{C}$

- If the two search procedures meet, the level i-cluster C containing (u, v) is still connected so C remains a level i-cluster
- Otherwise, C is be split in two, one part containing C_u and one containing C_v



Updates to $\mathcal{C}$

- If the two search procedures meet, the level i-cluster C containing (u, v) is still connected so C remains a level i-cluster
- Otherwise, C is be split in two, one part containing C_u and one containing C_v



- If $i > 0$, recurse on level $i - 1$

Edge Level Increases after Search

- Recall: each node w of $\mathcal{C}$ is associated with its size $n(w)$

Edge Level Increases after Search

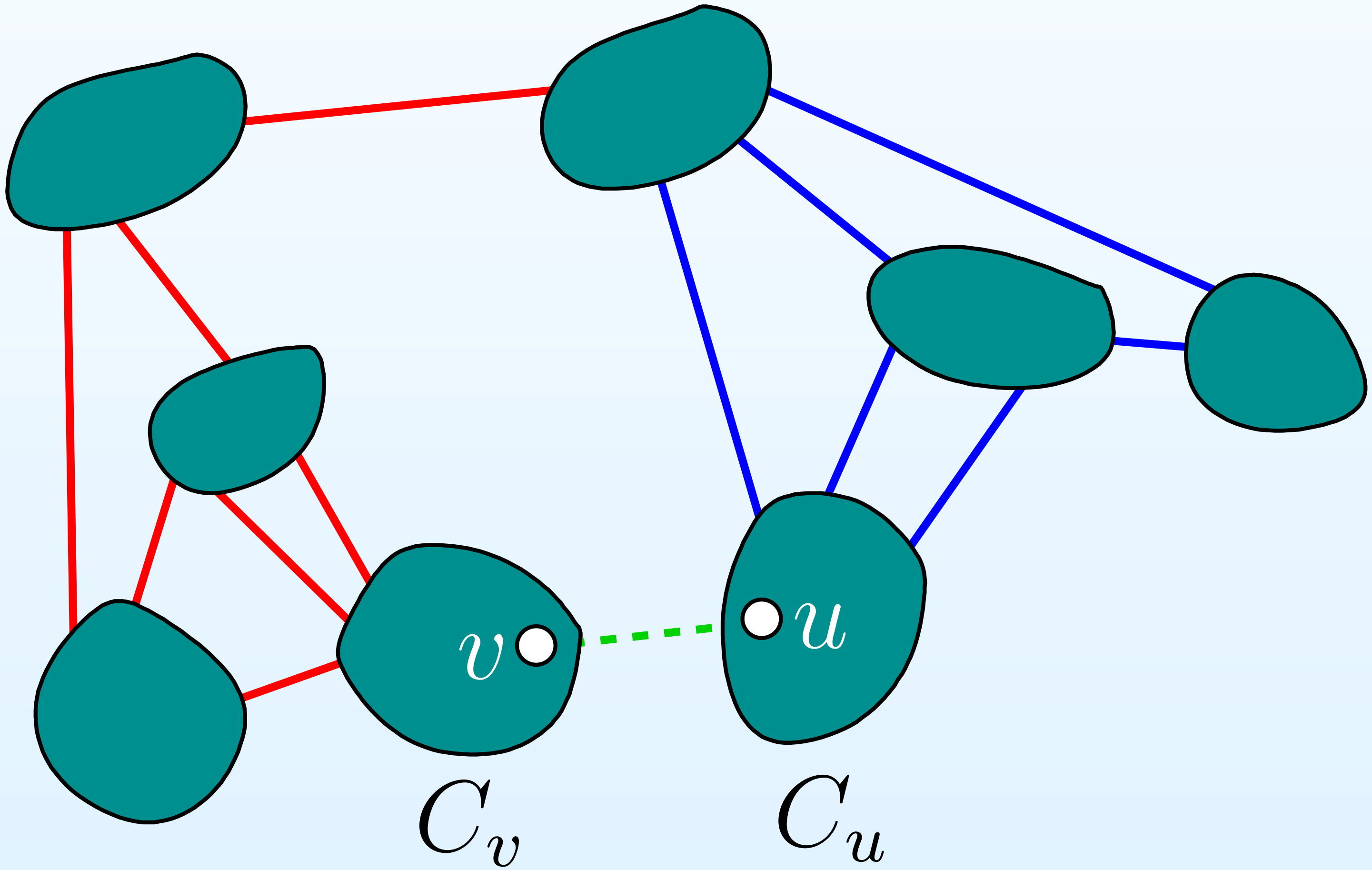
- Recall: each node w of $\mathcal{C}$ is associated with its size $n(w)$
- $n(w)$ is the number of vertices of V in cluster $C(w)$

Edge Level Increases after Search

- Recall: each node w of $\mathcal{C}$ is associated with its size $n(w)$
- $n(w)$ is the number of vertices of V in cluster $C(w)$
- For the search procedure that explored clusters of smallest total size, all its visited edges have their levels increased

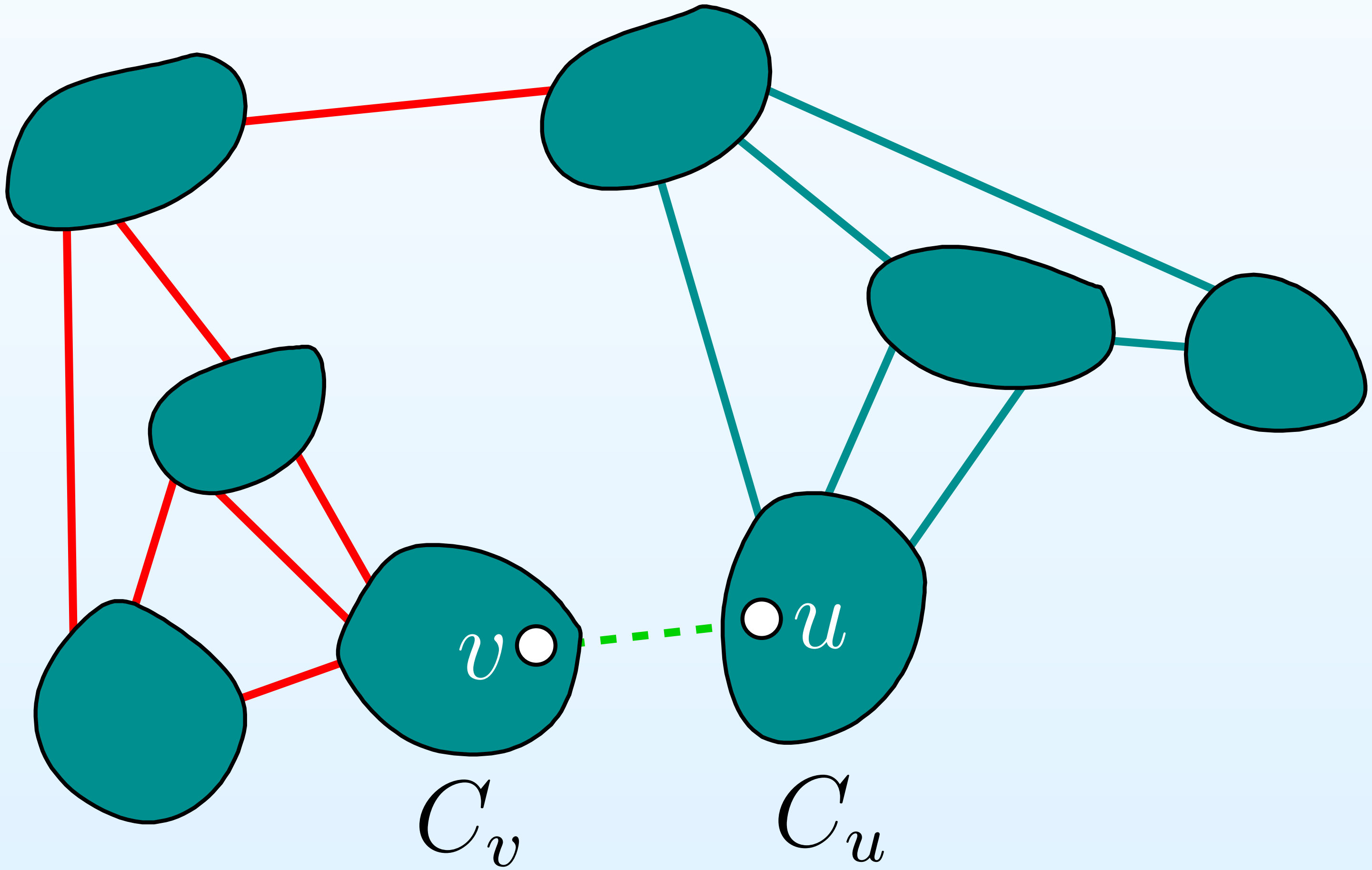
Edge Level Increases after Search

- Recall: each node w of $\mathcal{C}$ is associated with its size $n(w)$
- $n(w)$ is the number of vertices of V in cluster $C(w)$
- For the search procedure that explored clusters of smallest total size, all its visited edges have their levels increased



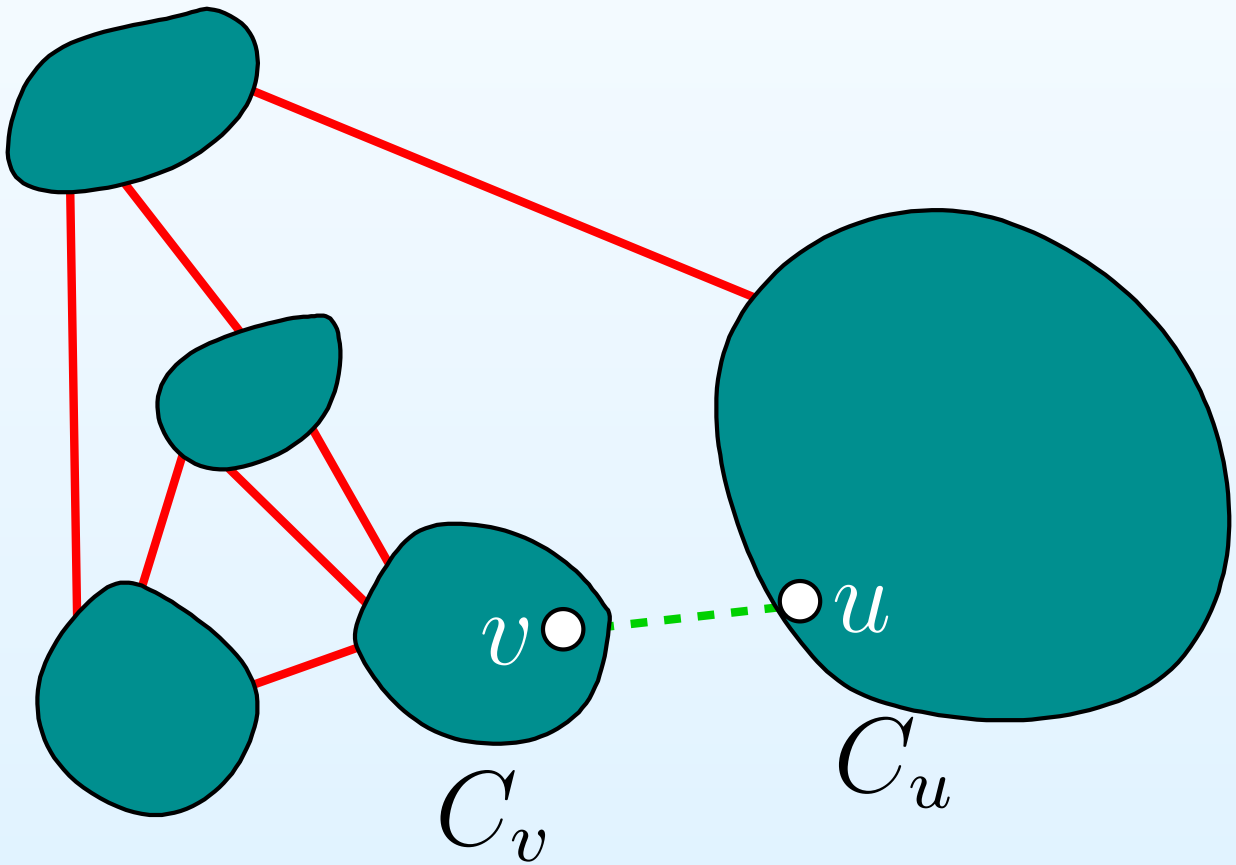
Edge Level Increases after Search

- Recall: each node w of $\mathcal{C}$ is associated with its size $n(w)$
- $n(w)$ is the number of vertices of V in cluster $C(w)$
- For the search procedure that explored clusters of smallest total size, all its visited edges have their levels increased



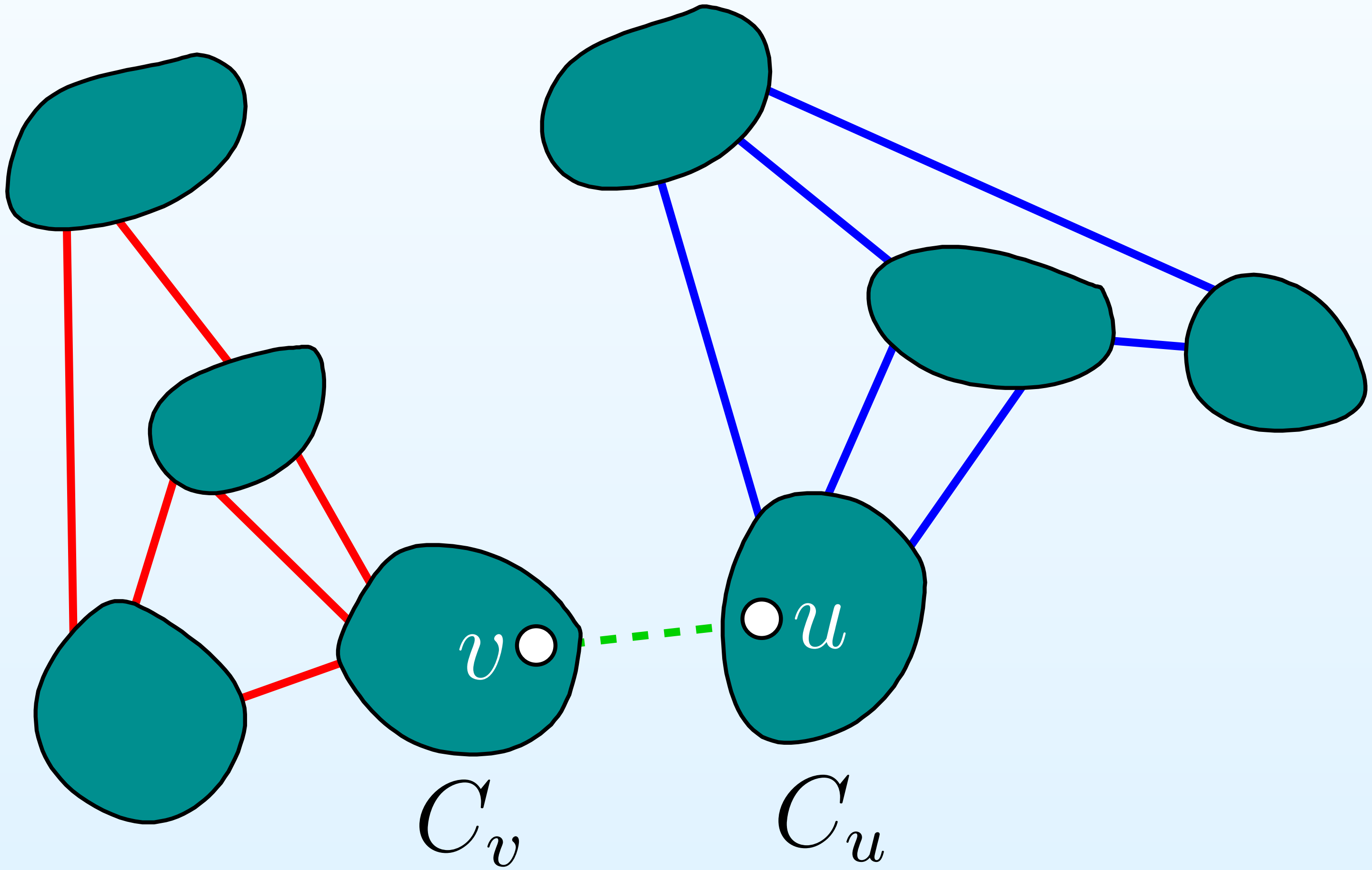
Edge Level Increases after Search

- Recall: each node w of $\mathcal{C}$ is associated with its size $n(w)$
- $n(w)$ is the number of vertices of V in cluster $C(w)$
- For the search procedure that explored clusters of smallest total size, all its visited edges have their levels increased



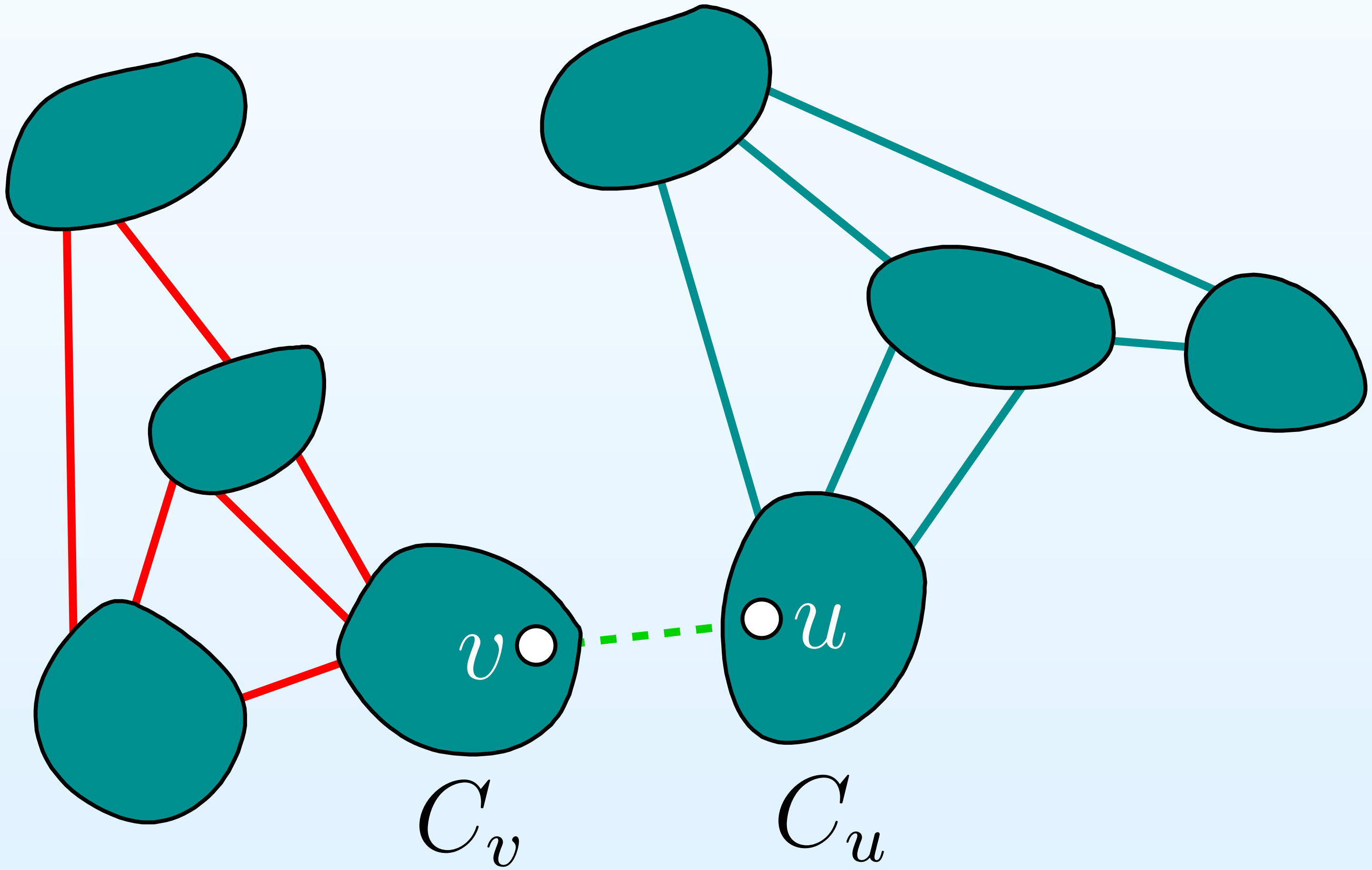
Edge Level Increases after Search

- Recall: each node w of $\mathcal{C}$ is associated with its size $n(w)$
- $n(w)$ is the number of vertices of V in cluster $C(w)$
- For the search procedure that explored clusters of smallest total size, all its visited edges have their levels increased



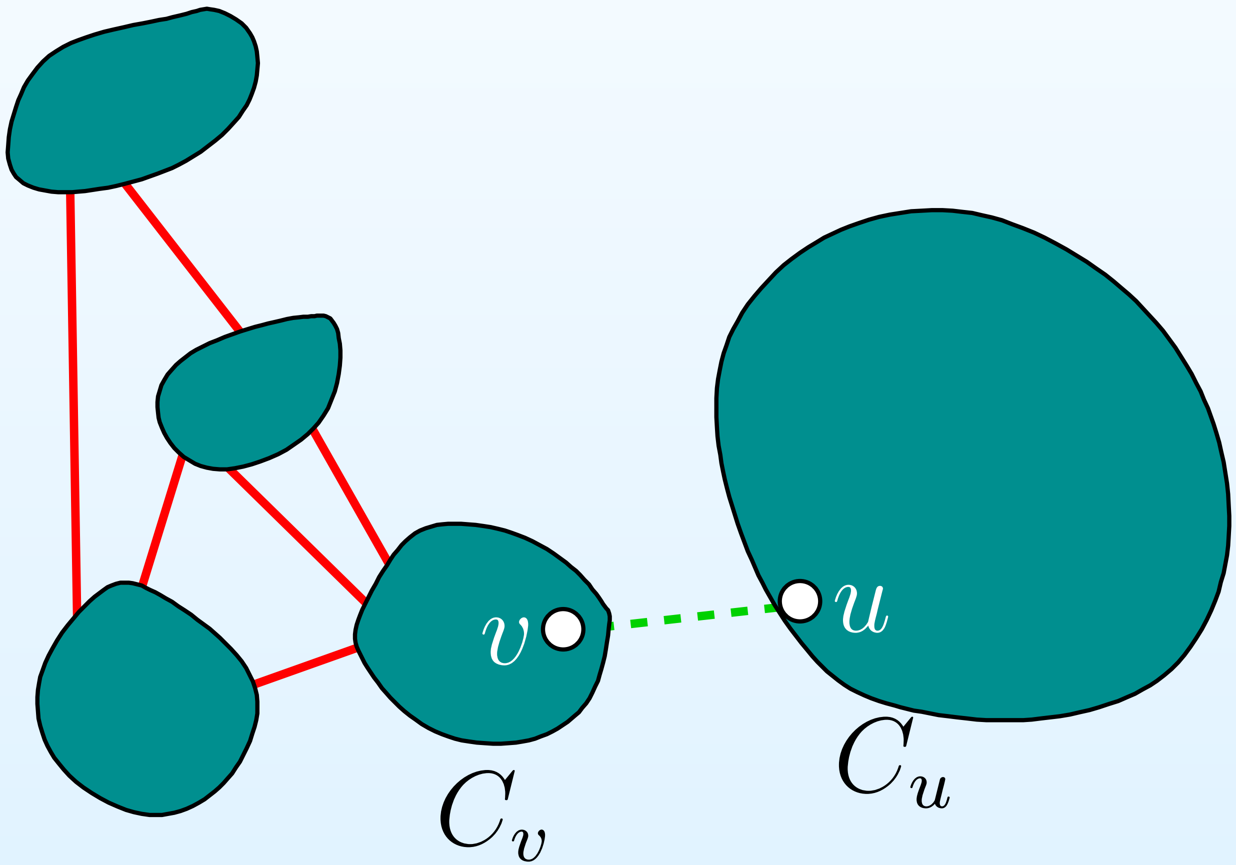
Edge Level Increases after Search

- Recall: each node w of $\mathcal{C}$ is associated with its size $n(w)$
- $n(w)$ is the number of vertices of V in cluster $C(w)$
- For the search procedure that explored clusters of smallest total size, all its visited edges have their levels increased



Edge Level Increases after Search

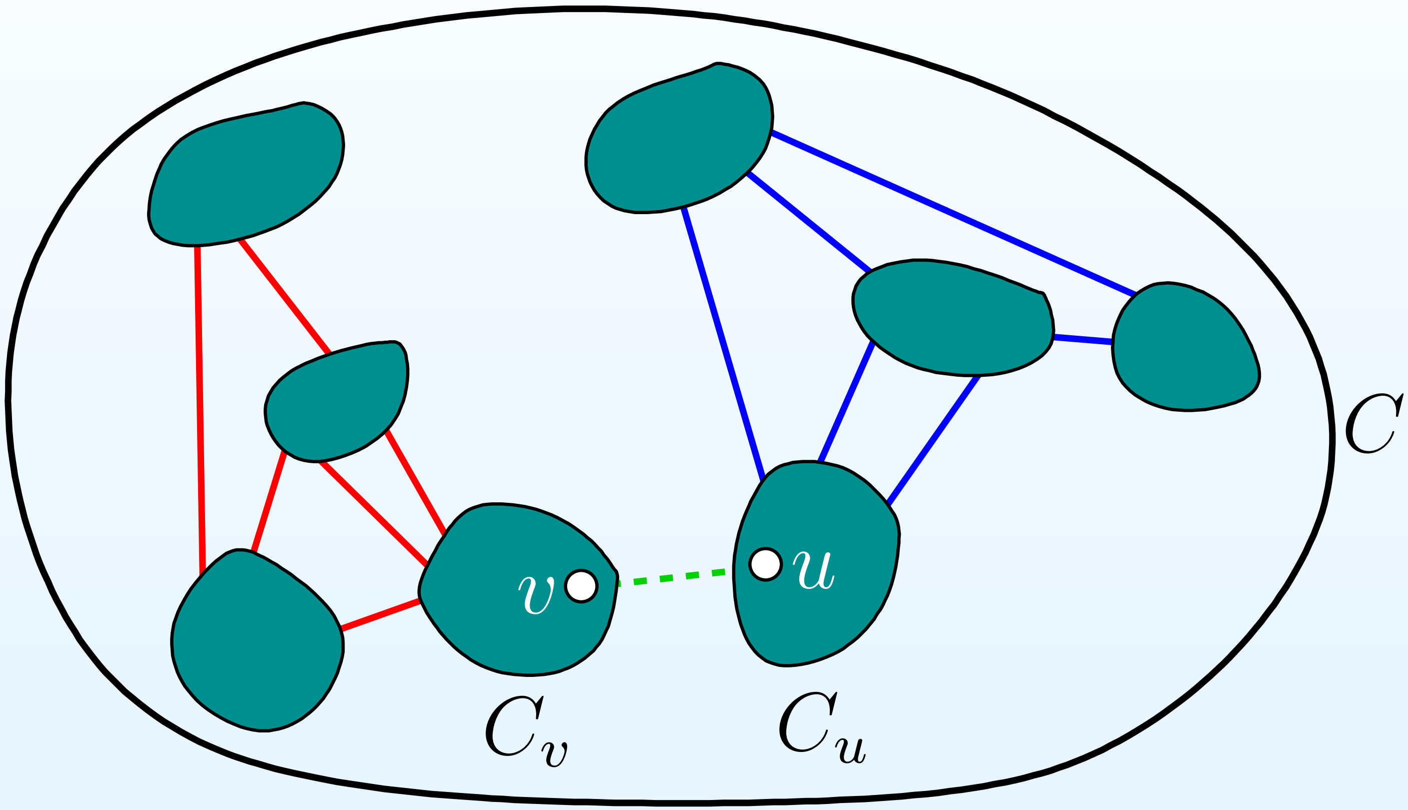
- Recall: each node w of $\mathcal{C}$ is associated with its size $n(w)$
- $n(w)$ is the number of vertices of V in cluster $C(w)$
- For the search procedure that explored clusters of smallest total size, all its visited edges have their levels increased



Maintaining the Invariant

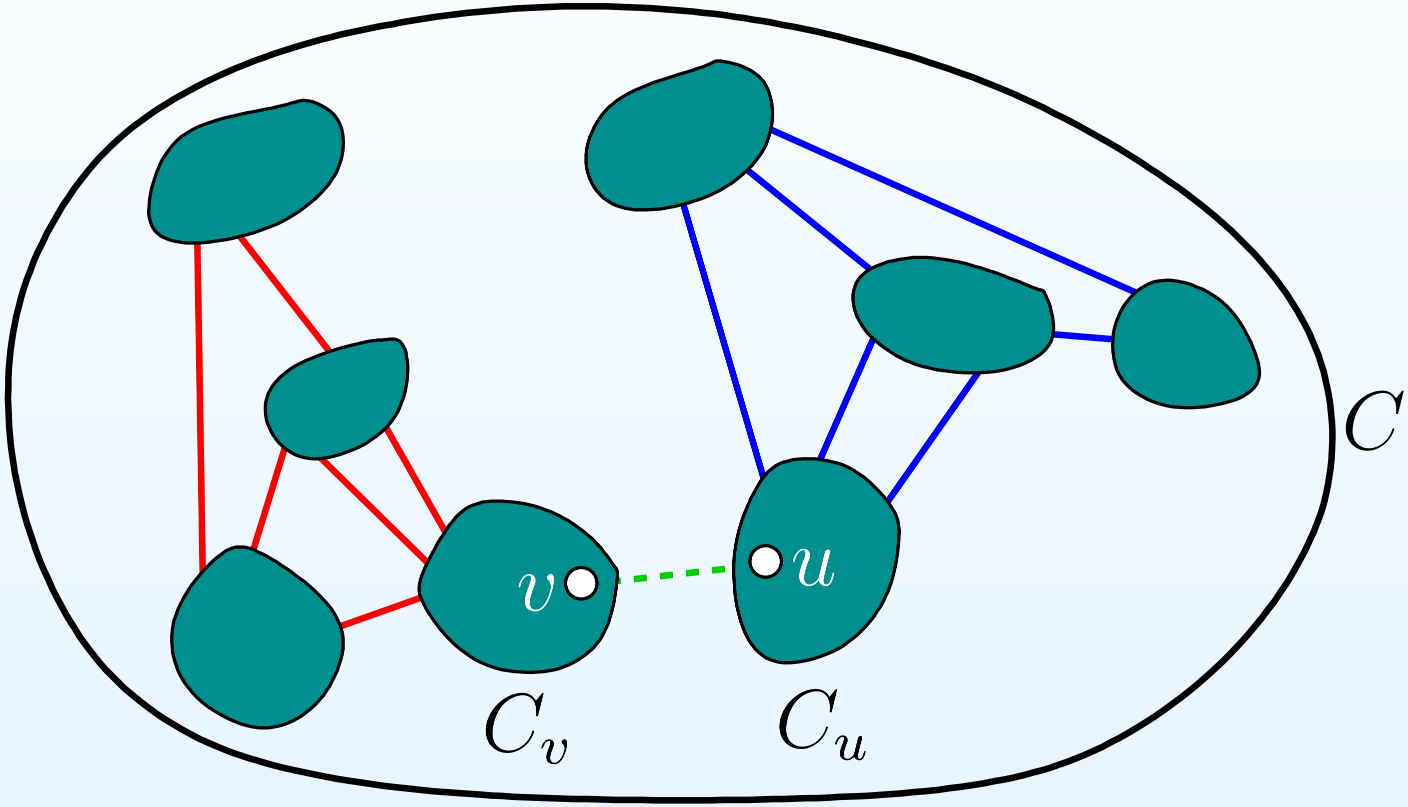
Maintaining the Invariant

- Parent level i-cluster C has size at most $\lfloor n/2^i \rfloor$



Maintaining the Invariant

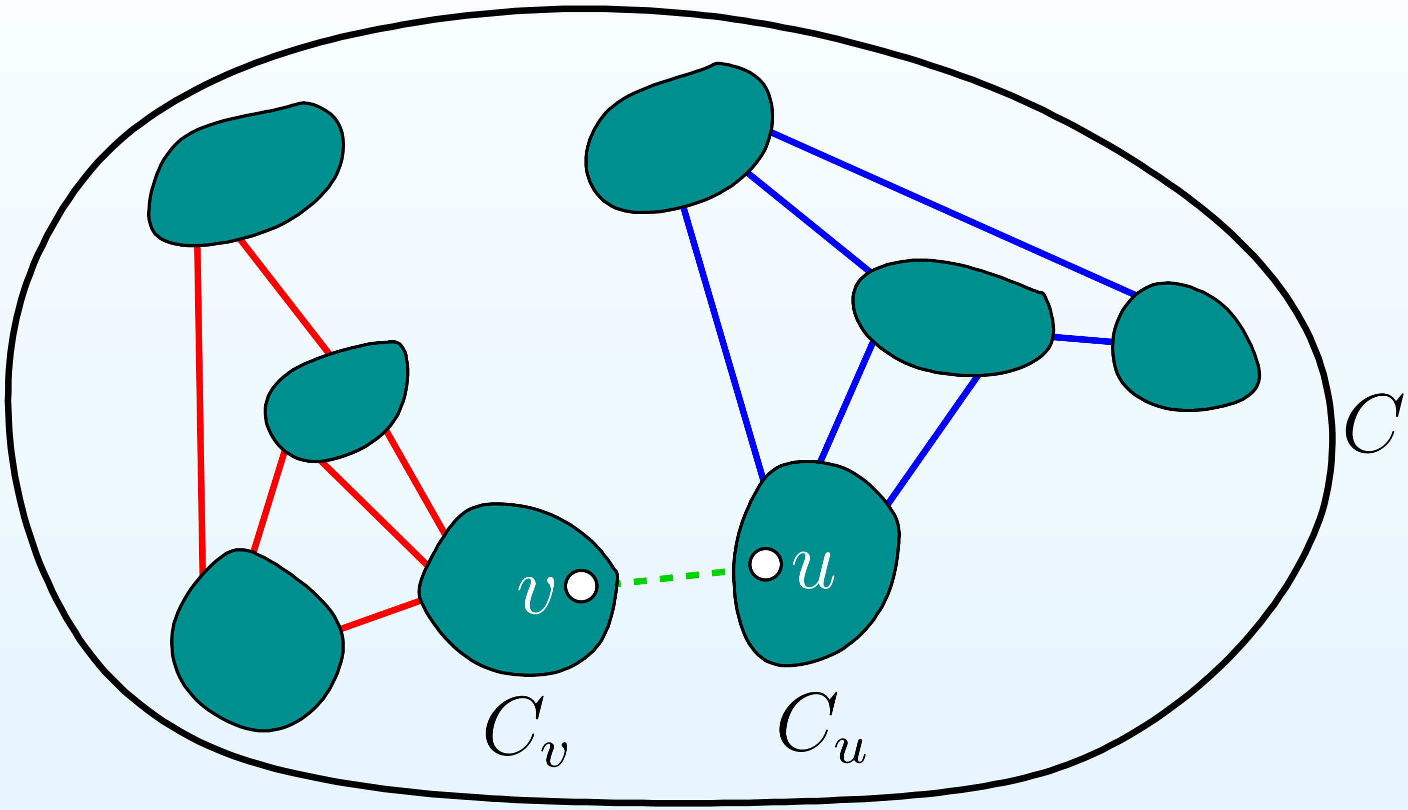
- Parent level i-cluster C has size at most $\lfloor n/2^i \rfloor$



- The smaller side has size at most $\lfloor n/2^{i+1} \rfloor$

Maintaining the Invariant

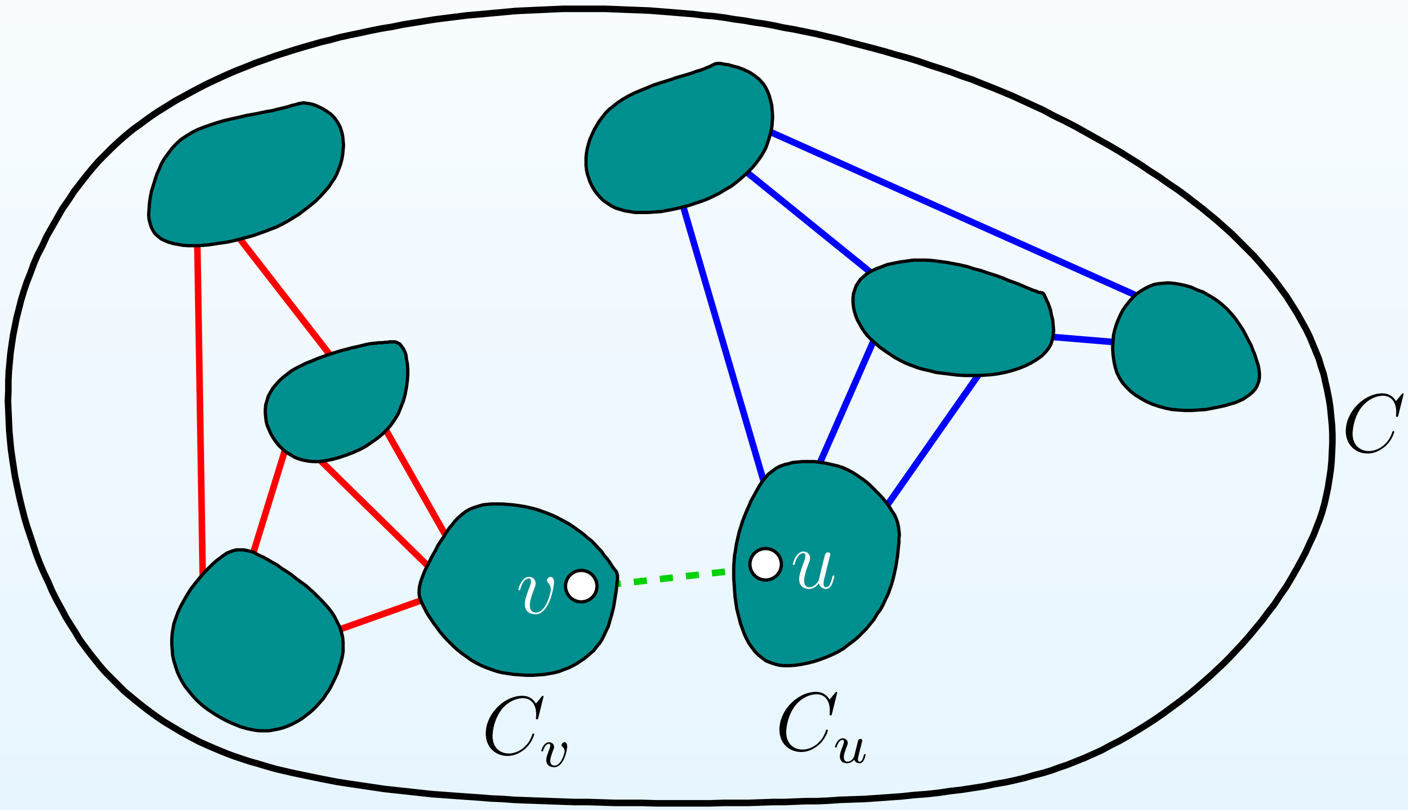
- Parent level i-cluster C has size at most $\lfloor n/2^i \rfloor$



- The smaller side has size at most $\lfloor n/2^{i+1} \rfloor$ since otherwise, C would have size $\geq 2(\lfloor n/2^{i+1} \rfloor + 1) > 2 \cdot n/2^{i+1} \geq \lfloor n/2^i \rfloor$

Maintaining the Invariant

- Parent level i-cluster C has size at most $\lfloor n/2^i \rfloor$



- The smaller side has size at most $\lfloor n/2^{i+1} \rfloor$ since otherwise, C would have size $\geq 2(\lfloor n/2^{i+1} \rfloor + 1) > 2 \cdot n/2^{i+1} \geq \lfloor n/2^i \rfloor$
- Thus, the invariant is still satisfied after merging level $(i+1)$-clusters

Overall Amortized Analysis

- Suppose each search procedure uses $O(1)$ time per edge visited

Overall Amortized Analysis

- Suppose each search procedure uses $O(1)$ time per edge visited
- For the analysis, we let each edge pay $O(1)$ credits when its level is increased

Overall Amortized Analysis

- Suppose each search procedure uses $O(1)$ time per edge visited
- For the analysis, we let each edge pay $O(1)$ credits when its level is increased
- The search on the smaller side is thus paid for by its visited edges

Overall Amortized Analysis

- Suppose each search procedure uses $O(1)$ time per edge visited
- For the analysis, we let each edge pay $O(1)$ credits when its level is increased
- The search on the smaller side is thus paid for by its visited edges
- The other search visits the same number of edges (plus/minus 1)

Overall Amortized Analysis

- Suppose each search procedure uses $O(1)$ time per edge visited
- For the analysis, we let each edge pay $O(1)$ credits when its level is increased
- The search on the smaller side is thus paid for by its visited edges
- The other search visits the same number of edges (plus/minus 1)
- Hence, the edge level increases can pay for both search procedures

Overall Amortized Analysis

- Suppose each search procedure uses $O(1)$ time per edge visited
- For the analysis, we let each edge pay $O(1)$ credits when its level is increased
- The search on the smaller side is thus paid for by its visited edges
- The other search visits the same number of edges (plus/minus 1)
- Hence, the edge level increases can pay for both search procedures
- Max level of an edge: $\ell_{\max} = \lfloor \log n \rfloor = O(\log n)$

Overall Amortized Analysis

- Suppose each search procedure uses $O(1)$ time per edge visited
- For the analysis, we let each edge pay $O(1)$ credits when its level is increased
- The search on the smaller side is thus paid for by its visited edges
- The other search visits the same number of edges (plus/minus 1)
- Hence, the edge level increases can pay for both search procedures
- Max level of an edge: $\ell_{\max} = \lfloor \log n \rfloor = O(\log n)$
- Amortized time per update is thus $O(\log n)$

Overall Amortized Analysis

- Suppose each search procedure uses $O(1)$ time per edge visited
- For the analysis, we let each edge pay $O(1)$ credits when its level is increased
- The search on the smaller side is thus paid for by its visited edges
- The other search visits the same number of edges (plus/minus 1)
- Hence, the edge level increases can pay for both search procedures
- Max level of an edge: $\ell_{\max} = \lfloor \log n \rfloor = O(\log n)$
- Amortized time per update is thus $O(\log n)$
- What is the problem with this analysis?

Overall Amortized Analysis

- Suppose each search procedure uses $O(1)$ time per edge visited
- For the analysis, we let each edge pay $O(1)$ credits when its level is increased
- The search on the smaller side is thus paid for by its visited edges
- The other search visits the same number of edges (plus/minus 1)
- Hence, the edge level increases can pay for both search procedures
- Max level of an edge: $\ell_{\max} = \lfloor \log n \rfloor = O(\log n)$
- Amortized time per update is thus $O(\log n)$
- What is the problem with this analysis?
    - The multigraph M_i is not stored explicitly

Overall Amortized Analysis

- Suppose each search procedure uses $O(1)$ time per edge visited
- For the analysis, we let each edge pay $O(1)$ credits when its level is increased
- The search on the smaller side is thus paid for by its visited edges
- The other search visits the same number of edges (plus/minus 1)
- Hence, the edge level increases can pay for both search procedures
- Max level of an edge: $\ell_{\max} = \lfloor \log n \rfloor = O(\log n)$
- Amortized time per update is thus $O(\log n)$
- What is the problem with this analysis?
  - The multigraph M_i is not stored explicitly
  - Thus, we cannot ensure $O(1)$ time per edge visited

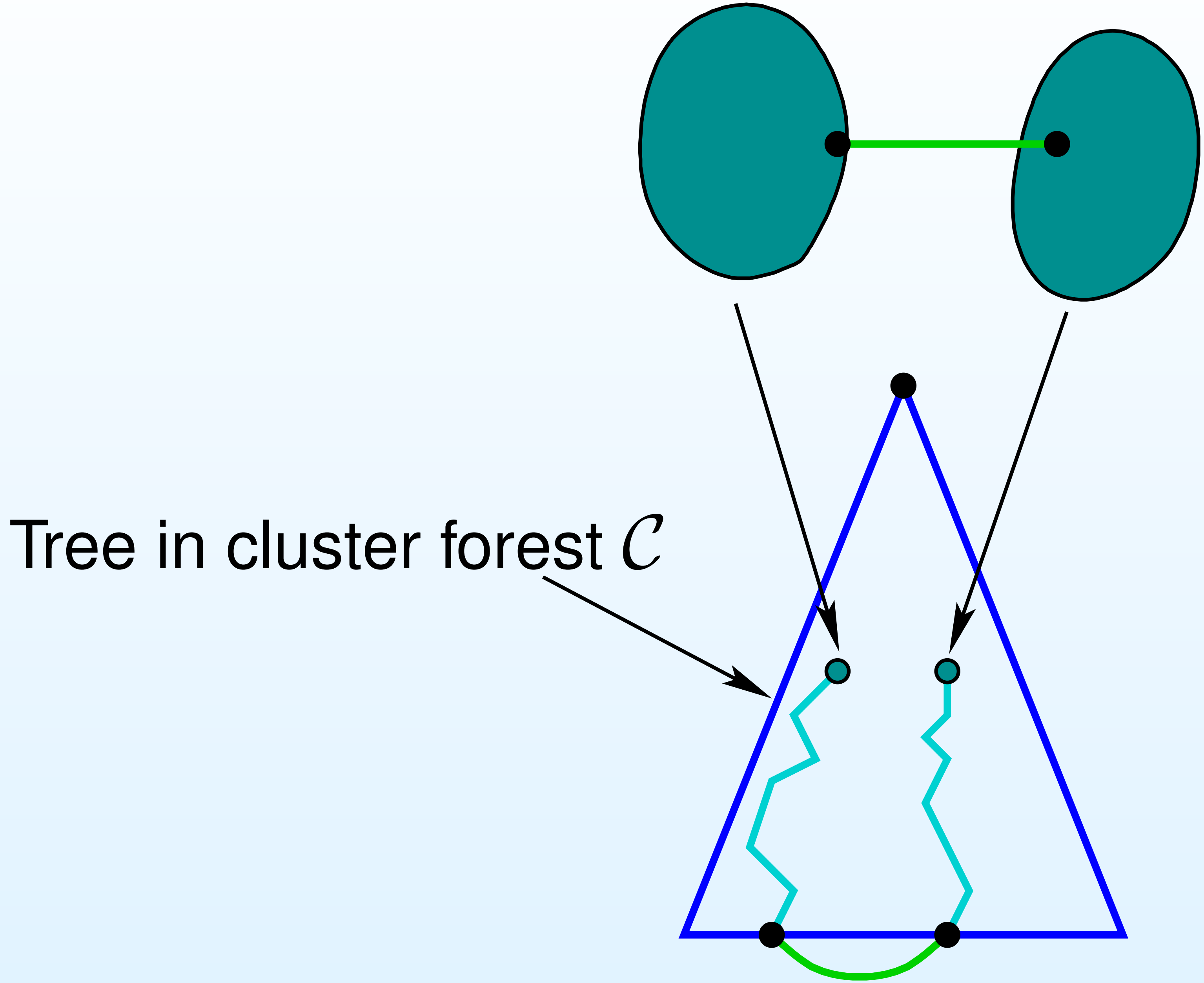
Overall Amortized Analysis

- Suppose each search procedure uses $O(1)$ time per edge visited
- For the analysis, we let each edge pay $O(1)$ credits when its level is increased
- The search on the smaller side is thus paid for by its visited edges
- The other search visits the same number of edges (plus/minus 1)
- Hence, the edge level increases can pay for both search procedures
- Max level of an edge: $\ell_{\max} = \lfloor \log n \rfloor = O(\log n)$
- Amortized time per update is thus $O(\log n)$
- What is the problem with this analysis?
  - The multigraph M_i is not stored explicitly
  - Thus, we cannot ensure $O(1)$ time per edge visited
  - We will instead show how to get $O(\log n)$ time per edge visited

Overall Amortized Analysis

- Suppose each search procedure uses $O(1)$ time per edge visited
- For the analysis, we let each edge pay $O(1)$ credits when its level is increased
- The search on the smaller side is thus paid for by its visited edges
- The other search visits the same number of edges (plus/minus 1)
- Hence, the edge level increases can pay for both search procedures
- Max level of an edge: $\ell_{\max} = \lfloor \log n \rfloor = O(\log n)$
- Amortized time per update is thus $O(\log n)$
- What is the problem with this analysis?
  - The multigraph M_i is not stored explicitly
  - Thus, we cannot ensure $O(1)$ time per edge visited
  - We will instead show how to get $O(\log n)$ time per edge visited
  - This will give $O(\log^2 n)$ amortized update time

Traversing a single graph edge



Assuming a Binary Cluster Forest $\mathcal{C}$

- Assume $\mathcal{C}$ is binary: every node has at most two children

Assuming a Binary Cluster Forest $\mathcal{C}$

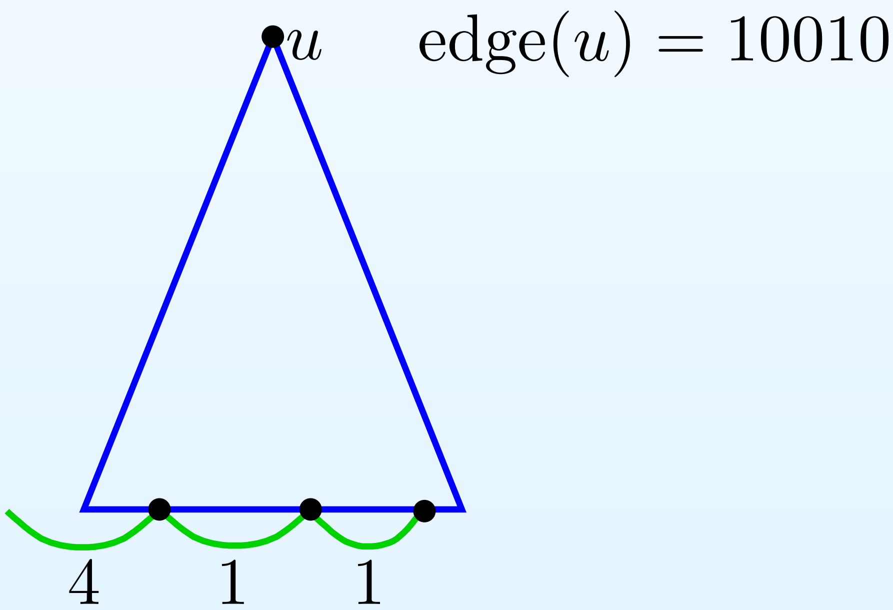
- Assume $\mathcal{C}$ is binary: every node has at most two children
- At each such node u, store an $\ell_{\max}$-bit word, $\mathrm{edge}(u)$

Assuming a Binary Cluster Forest $\mathcal{C}$

- Assume $\mathcal{C}$ is binary: every node has at most two children
- At each such node u, store an $\ell_{\max}$-bit word, $\mathrm{edge}(u)$
- The ith bit $\mathrm{edge}(u)[i]$ is 1 if and only if a level i-edge of E is incident to a leaf of the subtree of $\mathcal{C}$ rooted at u

Assuming a Binary Cluster Forest $\mathcal{C}$

- Assume $\mathcal{C}$ is binary: every node has at most two children
- At each such node u, store an $\ell_{\max}$-bit word, $\mathrm{edge}(u)$
- The ith bit $\mathrm{edge}(u)[i]$ is 1 if and only if a level i-edge of E is incident to a leaf of the subtree of $\mathcal{C}$ rooted at u
- Example with $\ell_{\max} = 5$:



Assuming a Binary Cluster Forest $\mathcal{C}$

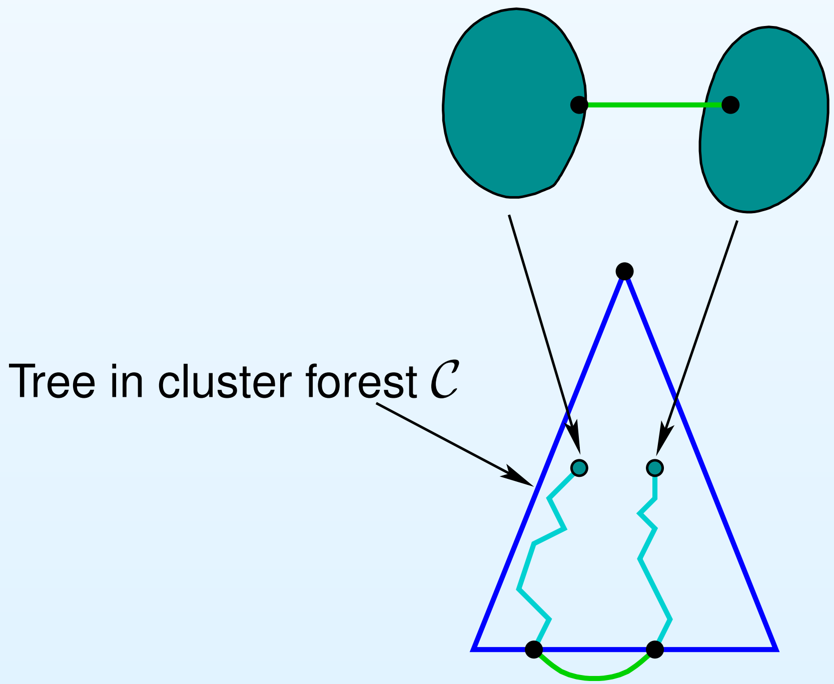
- Assume $\mathcal{C}$ is binary: every node has at most two children
- At each such node u, store an $\ell_{\max}$-bit word, $\mathrm{edge}(u)$
- The ith bit $\mathrm{edge}(u)[i]$ is 1 if and only if a level i-edge of E is incident to a leaf of the subtree of $\mathcal{C}$ rooted at u
- Maintaining these bitmaps can be done efficiently (exercise)

Assuming a Binary Cluster Forest $\mathcal{C}$

- Assume $\mathcal{C}$ is binary: every node has at most two children
- At each such node u, store an $\ell_{\max}$-bit word, $\mathrm{edge}(u)$
- The ith bit $\mathrm{edge}(u)[i]$ is 1 if and only if a level i-edge of E is incident to a leaf of the subtree of $\mathcal{C}$ rooted at u
- Maintaining these bitmaps can be done efficiently (exercise)
- Since $\mathcal{C}$ is binary, we can traverse a single edge of a multigraph in $O(\log n)$ time using the edge-bit maps (how?)

Assuming a Binary Cluster Forest $\mathcal{C}$

- Assume $\mathcal{C}$ is binary: every node has at most two children
- At each such node u, store an $\ell_{\max}$-bit word, $\mathrm{edge}(u)$
- The ith bit $\mathrm{edge}(u)[i]$ is 1 if and only if a level i-edge of E is incident to a leaf of the subtree of $\mathcal{C}$ rooted at u
- Maintaining these bitmaps can be done efficiently (exercise)
- Since $\mathcal{C}$ is binary, we can traverse a single edge of a multigraph in $O(\log n)$ time using the edge-bit maps (how?)



Assuming a Binary Cluster Forest $\mathcal{C}$

- Assume $\mathcal{C}$ is binary: every node has at most two children
- At each such node u, store an $\ell_{\max}$-bit word, $\mathrm{edge}(u)$
- The ith bit $\mathrm{edge}(u)[i]$ is 1 if and only if a level i-edge of E is incident to a leaf of the subtree of $\mathcal{C}$ rooted at u
- Maintaining these bitmaps can be done efficiently (exercise)
- Since $\mathcal{C}$ is binary, we can traverse a single edge of a multigraph in $O(\log n)$ time using the edge-bit maps (how?)
- This gives the desired time bound for the search procedures

Assuming a Binary Cluster Forest $\mathcal{C}$

- Assume $\mathcal{C}$ is binary: every node has at most two children
- At each such node u, store an $\ell_{\max}$-bit word, $\mathrm{edge}(u)$
- The ith bit $\mathrm{edge}(u)[i]$ is 1 if and only if a level i-edge of E is incident to a leaf of the subtree of $\mathcal{C}$ rooted at u
- Maintaining these bitmaps can be done efficiently (exercise)
- Since $\mathcal{C}$ is binary, we can traverse a single edge of a multigraph in $O(\log n)$ time using the edge-bit maps (how?)
- This gives the desired time bound for the search procedures
- However, we need to deal with the case where $\mathcal{C}$ is not binary

Node Ranks

- Recall: for each node u in $\mathcal{C}$, $n(u)$ is the number of leaves in the subtree of $\mathcal{C}$ rooted at u

Node Ranks

- Recall: for each node u in $\mathcal{C}$, $n(u)$ is the number of leaves in the subtree of $\mathcal{C}$ rooted at u
- Define the *rank* of u as $\mathrm{rank}(u) = \lfloor \lg n(u) \rfloor$

Rank Trees

- Let u be a non-leaf node in $\mathcal{C}$

Rank Trees

- Let u be a non-leaf node in $\mathcal{C}$
- Initialize node set R as the children of u in $\mathcal{C}$

Rank Trees

- Let u be a non-leaf node in $\mathcal{C}$
- Initialize node set R as the children of u in $\mathcal{C}$
- *Rank trees* of u are formed by repeating the following procedure as long as two nodes of R have the same rank:

Rank Trees

- Let u be a non-leaf node in $\mathcal{C}$
- Initialize node set R as the children of u in $\mathcal{C}$
- *Rank trees* of u are formed by repeating the following procedure as long as two nodes of R have the same rank:
    - Remove from R two nodes r_1 and r_2 with $\mathrm{rank}(r_1) = \mathrm{rank}(r_2)$

Rank Trees

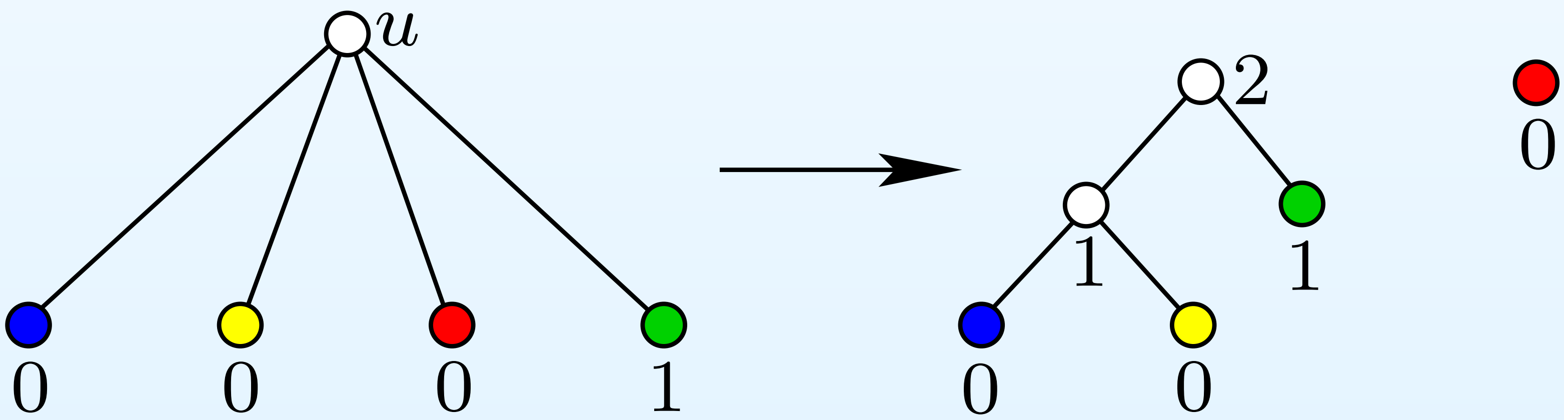
- Let u be a non-leaf node in $\mathcal{C}$
- Initialize node set R as the children of u in $\mathcal{C}$
- *Rank trees* of u are formed by repeating the following procedure as long as two nodes of R have the same rank:
  - Remove from R two nodes r_1 and r_2 with $\mathrm{rank}(r_1) = \mathrm{rank}(r_2)$
  - Attach r_1 and r_2 to a parent r of rank $\mathrm{rank}(r) = \mathrm{rank}(r_1) + 1$

Rank Trees

- Let u be a non-leaf node in $\mathcal{C}$
- Initialize node set R as the children of u in $\mathcal{C}$
- *Rank trees* of u are formed by repeating the following procedure as long as two nodes of R have the same rank:
  - Remove from R two nodes r_1 and r_2 with $\text{rank}(r_1) = \text{rank}(r_2)$
  - Attach r_1 and r_2 to a parent r of rank $\text{rank}(r) = \text{rank}(r_1) + 1$
  - Add r to R

Rank Trees

- Let u be a non-leaf node in $\mathcal{C}$
- Initialize node set R as the children of u in $\mathcal{C}$
- *Rank trees* of u are formed by repeating the following procedure as long as two nodes of R have the same rank:
  - Remove from R two nodes r_1 and r_2 with $\mathrm{rank}(r_1) = \mathrm{rank}(r_2)$
  - Attach r_1 and r_2 to a parent r of rank $\mathrm{rank}(r) = \mathrm{rank}(r_1) + 1$
  - Add r to R



Local trees

- Let $r_1, r_2, \ldots, r_k$ be the final set of rank tree roots in R ordered by decreasing rank:

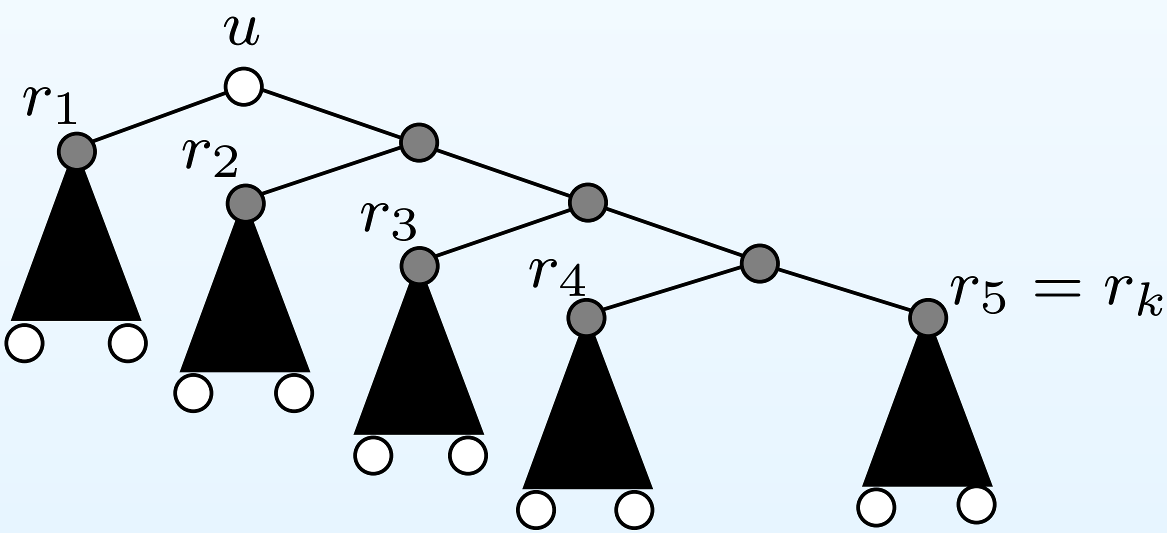
$$\text{rank}(r_1) > \text{rank}(r_2) > \cdots > \text{rank}(r_k)$$

Local trees

- Let $r_1, r_2, \ldots, r_k$ be the final set of rank tree roots in R ordered by decreasing rank:

$$\mathrm{rank}(r_1) > \mathrm{rank}(r_2) > \cdots > \mathrm{rank}(r_k)$$

- *Local tree* $L(u)$ for $k = 5$:

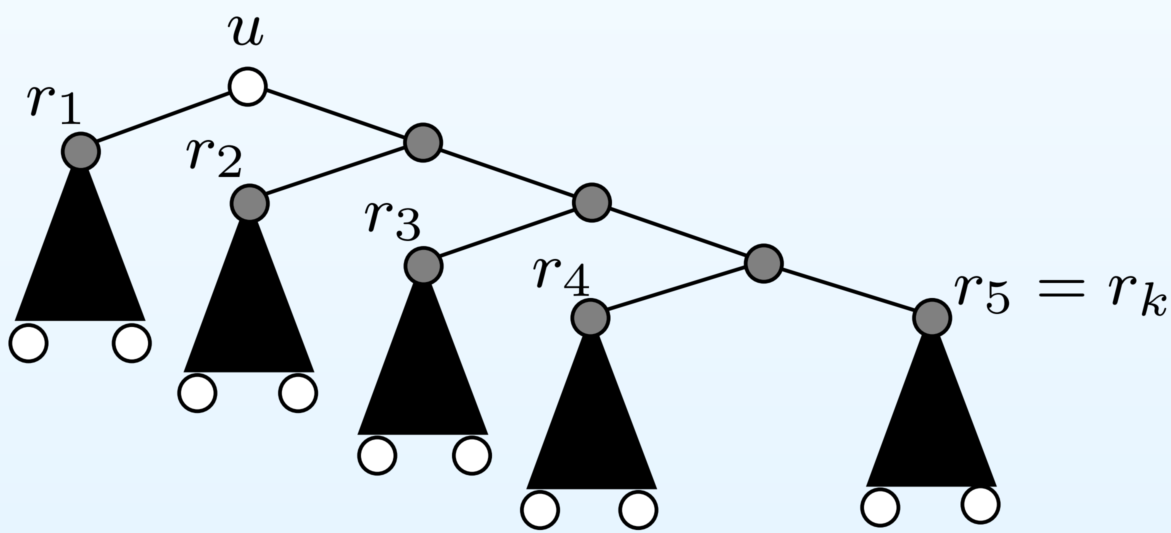


Local trees

- Let $r_1, r_2, \ldots, r_k$ be the final set of rank tree roots in R ordered by decreasing rank:

$$\text{rank}(r_1) > \text{rank}(r_2) > \cdots > \text{rank}(r_k)$$

- *Local tree* $L(u)$ for $k = 5$:



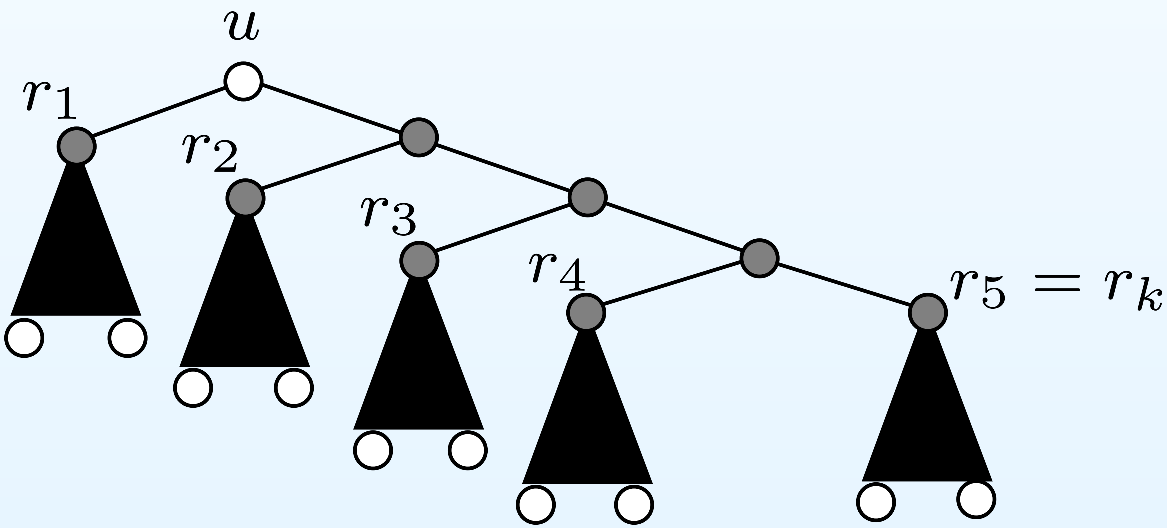
- Replace edges from u to its children in $\mathcal{C}$ by $L(u)$

Local trees

- Let $r_1, r_2, \ldots, r_k$ be the final set of rank tree roots in R ordered by decreasing rank:

$$\mathrm{rank}(r_1) > \mathrm{rank}(r_2) > \cdots > \mathrm{rank}(r_k)$$

- *Local tree* $L(u)$ for $k = 5$:



- Replace edges from u to its children in $\mathcal{C}$ by $L(u)$
- Doing this for all u turns $\mathcal{C}$ into forest $\mathcal{C}_L$ of binary trees

Properties of $\mathcal{C}_L$

- $\mathcal{C}_L$ has height $O(\log n)$ (exercise)

Properties of $\mathcal{C}_L$

- $\mathcal{C}_L$ has height $O(\log n)$ (exercise)
- Merging nodes u and v in $\mathcal{C}$ involves merging $L(u)$ and $L(v)$ in $\mathcal{C}_L$

Properties of $\mathcal{C}_L$

- $\mathcal{C}_L$ has height $O(\log n)$ (exercise)
- Merging nodes u and v in $\mathcal{C}$ involves merging $L(u)$ and $L(v)$ in $\mathcal{C}_L$
- Splitting a node u involves splitting $L(u)$

Properties of $\mathcal{C}_L$

- $\mathcal{C}_L$ has height $O(\log n)$ (exercise)
- Merging nodes u and v in $\mathcal{C}$ involves merging $L(u)$ and $L(v)$ in $\mathcal{C}_L$
- Splitting a node u involves splitting $L(u)$
- This can be done in $O(\log n)$ time per merge/split and will not increase the asymptotic update time (exercise)

Performance of data structure

- Each edge pays $O(\log n)$ credits each time its level increases

Performance of data structure

- Each edge pays $O(\log n)$ credits each time its level increases
- Its level can never decrease

Performance of data structure

- Each edge pays $O(\log n)$ credits each time its level increases
- Its level can never decrease
- Number of levels: $O(\log n)$

Performance of data structure

- Each edge pays $O(\log n)$ credits each time its level increases
- Its level can never decrease
- Number of levels: $O(\log n)$
- Amortized time per update: $O(\log^2 n)$

Performance of data structure

- Each edge pays $O(\log n)$ credits each time its level increases
- Its level can never decrease
- Number of levels: $O(\log n)$
- Amortized time per update: $O(\log^2 n)$
- Query time: $O(\log n)$

Performance of data structure

- Each edge pays $O(\log n)$ credits each time its level increases
- Its level can never decrease
- Number of levels: $O(\log n)$
- Amortized time per update: $O(\log^2 n)$
- Query time: $O(\log n)$
- Space:

Performance of data structure

- Each edge pays $O(\log n)$ credits each time its level increases
- Its level can never decrease
- Number of levels: $O(\log n)$
- Amortized time per update: $O(\log^2 n)$
- Query time: $O(\log n)$
- Space: $O(m + n \log n)$ words

Performance of data structure

- Each edge pays $O(\log n)$ credits each time its level increases
- Its level can never decrease
- Number of levels: $O(\log n)$
- Amortized time per update: $O(\log^2 n)$
- Query time: $O(\log n)$
- Space: $O(m + n \log n)$ words
- Can be improved to $O(m + n)$ by compressing paths in $\mathcal{C}_L$, whose interior nodes have degree 2, to single edges

Performance of data structure

- Each edge pays $O(\log n)$ credits each time its level increases
- Its level can never decrease
- Number of levels: $O(\log n)$
- Amortized time per update: $O(\log^2 n)$
- Query time: $O(\log n)$
- Space: $O(m + n \log n)$ words
- Can be improved to $O(m + n)$ by compressing paths in $\mathcal{C}_L$, whose interior nodes have degree 2, to single edges
- Using a more complicated data structure, both update and query time can be improved by a factor of $\log \log n$

Performance of data structure

- Each edge pays $O(\log n)$ credits each time its level increases
- Its level can never decrease
- Number of levels: $O(\log n)$
- Amortized time per update: $O(\log^2 n)$
- Query time: $O(\log n)$
- Space: $O(m + n \log n)$ words
- Can be improved to $O(m + n)$ by compressing paths in $\mathcal{C}_L$, whose interior nodes have degree 2, to single edges
- Using a more complicated data structure, both update and query time can be improved by a factor of $\log \log n$
- This is still the fastest deterministic data structure known