Faster Deterministic Fully-Dynamic Graph Connectivity

Christian Wulff-Nilsen

Department of Computer Science

University of Copenhagen

Denmark

Fully-dynamic graph connectivity

- Find an efficient data structure supporting the following operations in a dynamic graph G:

Fully-dynamic graph connectivity

- Find an efficient data structure supporting the following operations in a dynamic graph G:
  - `insert`(u, v): inserts edge (u, v) in G

Fully-dynamic graph connectivity

- Find an efficient data structure supporting the following operations in a dynamic graph G:
  - `insert`(u, v): inserts edge (u, v) in G
  - `delete`(u, v): deletes edge (u, v) from G

Fully-dynamic graph connectivity

- Find an efficient data structure supporting the following operations in a dynamic graph G:
  - `insert`(u, v): inserts edge (u, v) in G
  - `delete`(u, v): deletes edge (u, v) from G
  - `connected`(u, v): reports whether vertices u and v are connected in G

Fully-dynamic graph connectivity

- Find an efficient data structure supporting the following operations in a dynamic graph G:
  - `insert`(u, v): inserts edge (u, v) in G
  - `delete`(u, v): deletes edge (u, v) from G
  - `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

Worst-case bounds

- Eppstein, Galil, Italiano, Nissenzweig, 1992: $O(\sqrt{n})$ update, $O(1)$ query

Worst-case bounds

- Eppstein, Galil, Italiano, Nissenzweig, 1992: $O(\sqrt{n})$ update, $O(1)$ query
- Kapron, King, Mountjoy, 2013: $O(\mathrm{polylog}\, n)$ update and query (Monte Carlo)

Amortized bounds

- Randomized:

Amortized bounds

- Randomized:
  - Thorup, 2000: $O(\log n(\log\log n)^3)$ update and $O(\log n/\log\log\log n)$ query

Amortized bounds

- Randomized:
  - Thorup, 2000: $O(\log n (\log\log n)^3)$ update and $O(\log n / \log\log\log n)$ query
- Deterministic:

Amortized bounds

- Randomized:
  - Thorup, 2000: $O(\log n(\log\log n)^3)$ update and $O(\log n/\log\log\log n)$ query
- Deterministic:
  - Holm, de Lichtenberg, Thorup, 1998: $O(\log^2 n)$ update, $O(\log n/\log\log n)$ query

Amortized bounds

- Randomized:
  - Thorup, 2000: $O(\log n(\log \log n)^3)$ update and $O(\log n / \log \log \log n)$ query
- Deterministic:
  - Holm, de Lichtenberg, Thorup, 1998: $O(\log^2 n)$ update, $O(\log n / \log \log n)$ query
  - New result: $O(\log^2 n / \log \log n)$ update, $O(\log n / \log \log n)$ query

Model of computation

- We assume a pointer machine with words (bitmaps) containing at least $\lfloor \log n \rfloor + 1$ bits

Model of computation

- We assume a pointer machine with words (bitmaps) containing at least $\lfloor \log n \rfloor + 1$ bits
- We allow standard AC^0 instructions:

Model of computation

- We assume a pointer machine with words (bitmaps) containing at least $\lfloor \log n \rfloor + 1$ bits
- We allow standard AC^0 instructions:
  - Addition

Model of computation

- We assume a pointer machine with words (bitmaps) containing at least $\lfloor \log n \rfloor + 1$ bits
- We allow standard AC^0 instructions:
  - Addition
  - Subtraction

Model of computation

- We assume a pointer machine with words (bitmaps) containing at least $\lfloor \log n \rfloor + 1$ bits
- We allow standard AC^0 instructions:
  - Addition
  - Subtraction
  - Comparison

Model of computation

- We assume a pointer machine with words (bitmaps) containing at least $\lfloor \log n \rfloor + 1$ bits
- We allow standard AC^0 instructions:
  - Addition
  - Subtraction
  - Comparison
  - Bit shifts

Model of computation

- We assume a pointer machine with words (bitmaps) containing at least $\lfloor \log n \rfloor + 1$ bits
- We allow standard AC^0 instructions:
  - Addition
  - Subtraction
  - Comparison
  - Bit shifts
  - Boolean 'and', 'or', and 'xor'

The traditional approach

- Maintain a spanning forest of G

The traditional approach

- Maintain a spanning forest of G
- For an update $\texttt{insert}(u, v)$, find trees T_u and T_v containing u and v, respectively

The traditional approach

- Maintain a spanning forest of G
- For an update `insert`(u, v), find trees T_u and T_v containing u and v, respectively
- If $T_u \neq T_v$, add (u, v) as a tree edge; otherwise do nothing

The traditional approach

- Maintain a spanning forest of G
- For an update `insert`(u, v), find trees T_u and T_v containing u and v, respectively
- If $T_u \neq T_v$, add (u, v) as a tree edge; otherwise do nothing
- For an update `delete`(u, v), if (u, v) is a non-tree edge, do nothing

The traditional approach

- Maintain a spanning forest of G
- For an update `insert`(u, v), find trees T_u and T_v containing u and v, respectively
- If $T_u \neq T_v$, add (u, v) as a tree edge; otherwise do nothing
- For an update `delete`(u, v), if (u, v) is a non-tree edge, do nothing
- Otherwise, letting T be the tree containing (u, v), look for an edge (u', v') reconnecting $T \setminus \{(u, v)\}$

The traditional approach

- Maintain a spanning forest of G
- For an update $\texttt{insert}(u, v)$, find trees T_u and T_v containing u and v, respectively
- If $T_u \neq T_v$, add (u, v) as a tree edge; otherwise do nothing
- For an update $\texttt{delete}(u, v)$, if (u, v) is a non-tree edge, do nothing
- Otherwise, letting T be the tree containing (u, v), look for an edge (u', v') reconnecting $T \setminus \{(u, v)\}$
- If (u', v') exists, add it as a tree edge; otherwise do nothing

Clusters

- Assign an integer level $\ell(e)$ to each edge $e \in G$, $0 \le \ell(e) \le \ell_{\max} = \lfloor \log n \rfloor$

Clusters

- Assign an integer level $\ell(e)$ to each edge $e \in G$, $0 \leq \ell(e) \leq \ell_{\max} = \lfloor \log n \rfloor$
- For $0 \leq i \leq \ell_{\max}$, let G_i denote the subgraph of G induced by edges e with $\ell(e) \geq i$

Clusters

- Assign an integer level $\ell(e)$ to each edge $e \in G$, $0 \leq \ell(e) \leq \ell_{\max} = \lfloor \log n \rfloor$
- For $0 \leq i \leq \ell_{\max}$, let G_i denote the subgraph of G induced by edges e with $\ell(e) \geq i$
- We have $G = G_0 \supseteq G_1 \supseteq G_2 \supseteq \cdots \supseteq G_{\ell_{\max}}$

Clusters

- Assign an integer level $\ell(e)$ to each edge $e \in G$, $0 \leq \ell(e) \leq \ell_{\max} = \lfloor \log n \rfloor$
- For $0 \leq i \leq \ell_{\max}$, let G_i denote the subgraph of G induced by edges e with $\ell(e) \geq i$
- We have $G = G_0 \supseteq G_1 \supseteq G_2 \supseteq \cdots \supseteq G_{\ell_{\max}}$
- The connected components of G_i are called *level* i*-clusters* or just *clusters*

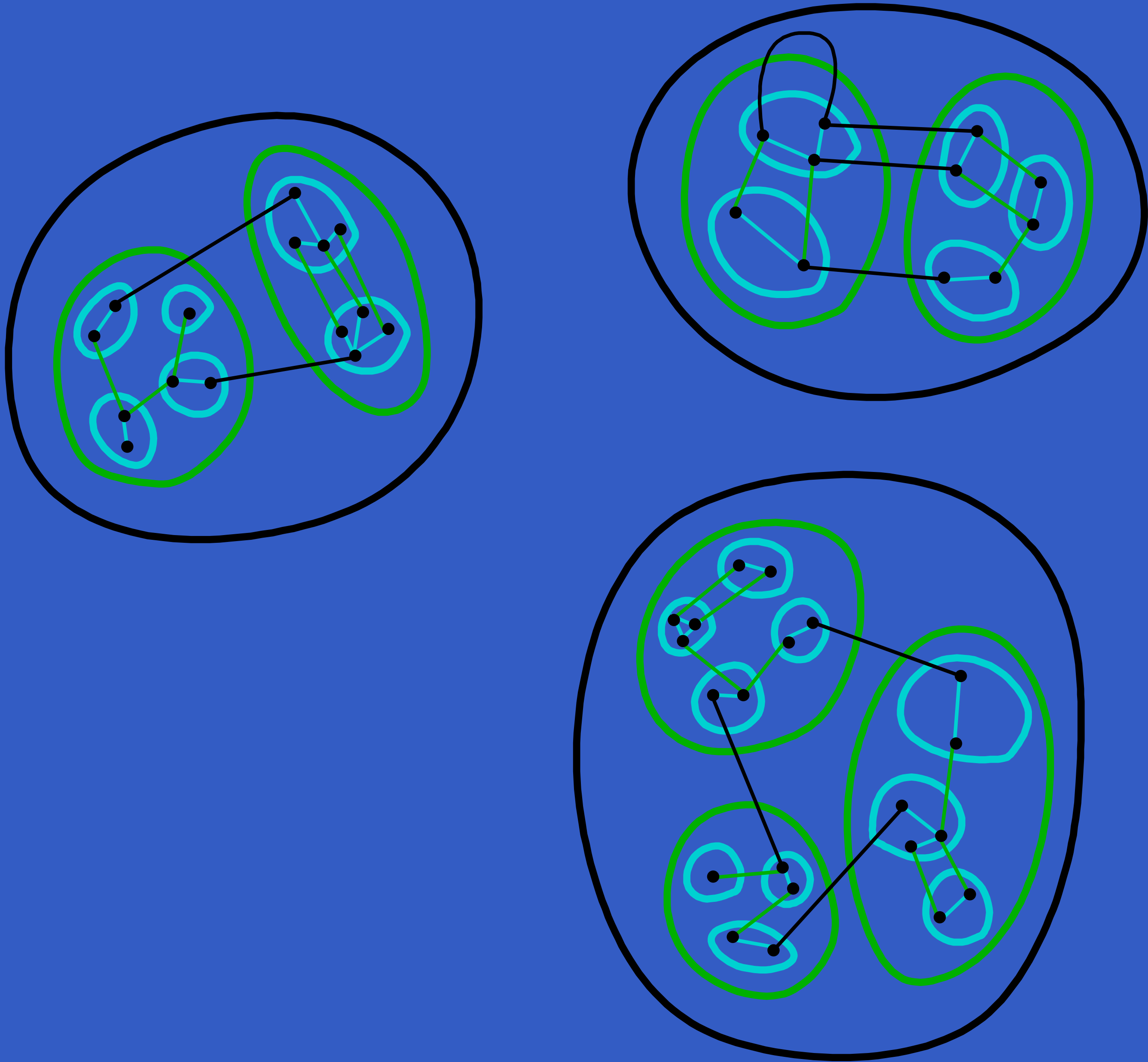
Clusters

- Assign an integer level $\ell(e)$ to each edge $e \in G$, $0 \leq \ell(e) \leq \ell_{\max} = \lfloor \log n \rfloor$
- For $0 \leq i \leq \ell_{\max}$, let G_i denote the subgraph of G induced by edges e with $\ell(e) \geq i$
- We have $G = G_0 \supseteq G_1 \supseteq G_2 \supseteq \cdots \supseteq G_{\ell_{\max}}$
- The connected components of G_i are called *level* i*-clusters* or just *clusters*
- Invariant: any level i-cluster spans at most $\lfloor n/2^i \rfloor$ vertices

Clusters

- Assign an integer level $\ell(e)$ to each edge $e \in G$, $0 \leq \ell(e) \leq \ell_{\max} = \lfloor \log n \rfloor$
- For $0 \leq i \leq \ell_{\max}$, let G_i denote the subgraph of G induced by edges e with $\ell(e) \geq i$
- We have $G = G_0 \supseteq G_1 \supseteq G_2 \supseteq \cdots \supseteq G_{\ell_{\max}}$
- The connected components of G_i are called *level i-clusters* or just *clusters*
- Invariant: any level i-cluster spans at most $\lfloor n/2^i \rfloor$ vertices
- Level 0-clusters are the connected components of G and level $\ell_{\max}$-clusters are vertices of G

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 $C(v) \subseteq C(u)$

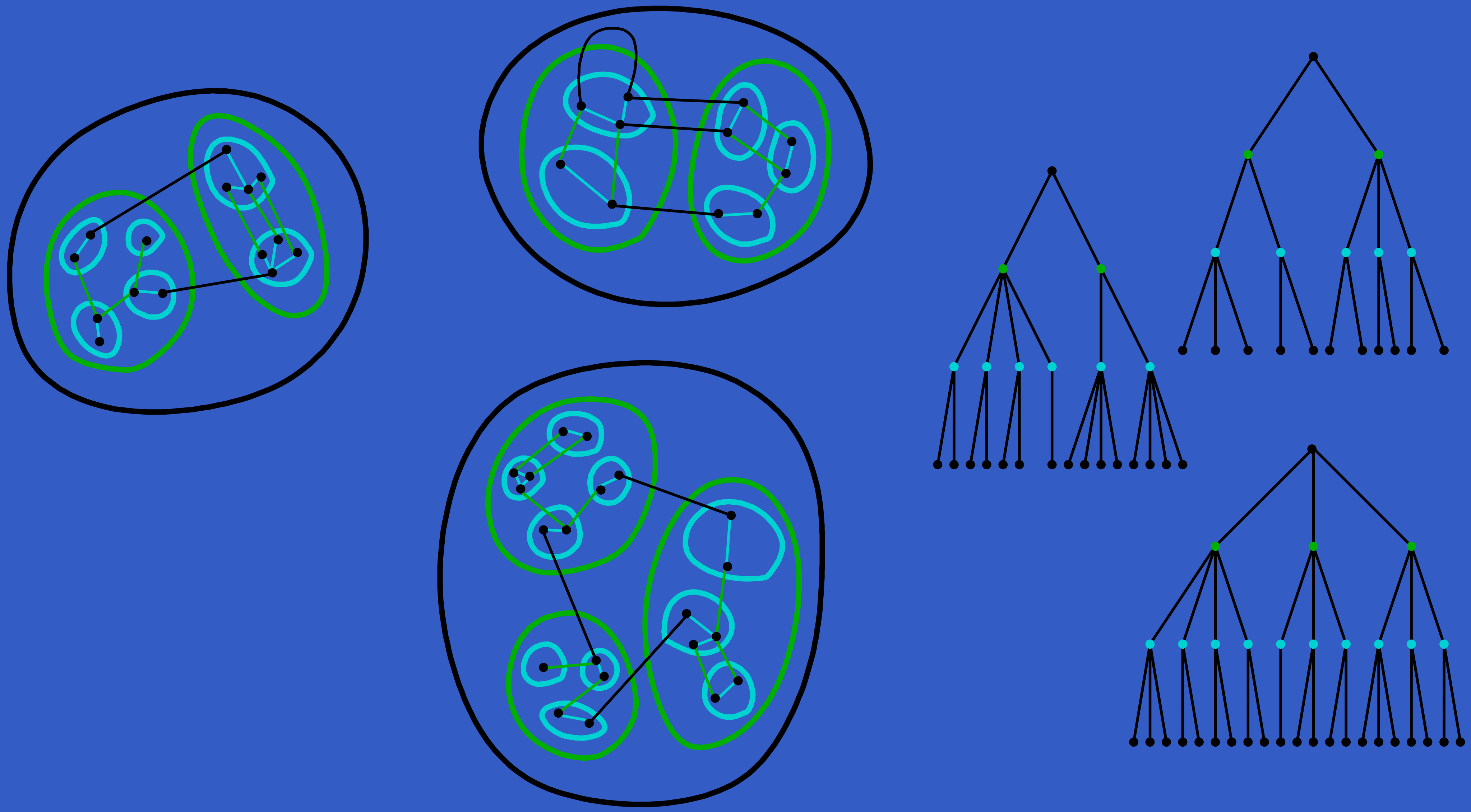
Cluster forest

- 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
- Given $\mathcal{C}$, we can determine whether two vertices u and v are connected in G in $O(\log n)$ time

Cluster forest



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

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

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

- Initialize $\ell(u, v) \leftarrow 0$
- Let r_u resp. r_v be the root of the tree of $\mathcal{C}$ containing u resp. v

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

- Initialize $\ell(u, v) \leftarrow 0$
- Let r_u resp. r_v be the root of the tree of $\mathcal{C}$ containing u resp. v
- If $r_u = r_v$, $\mathcal{C}$ need not be updated

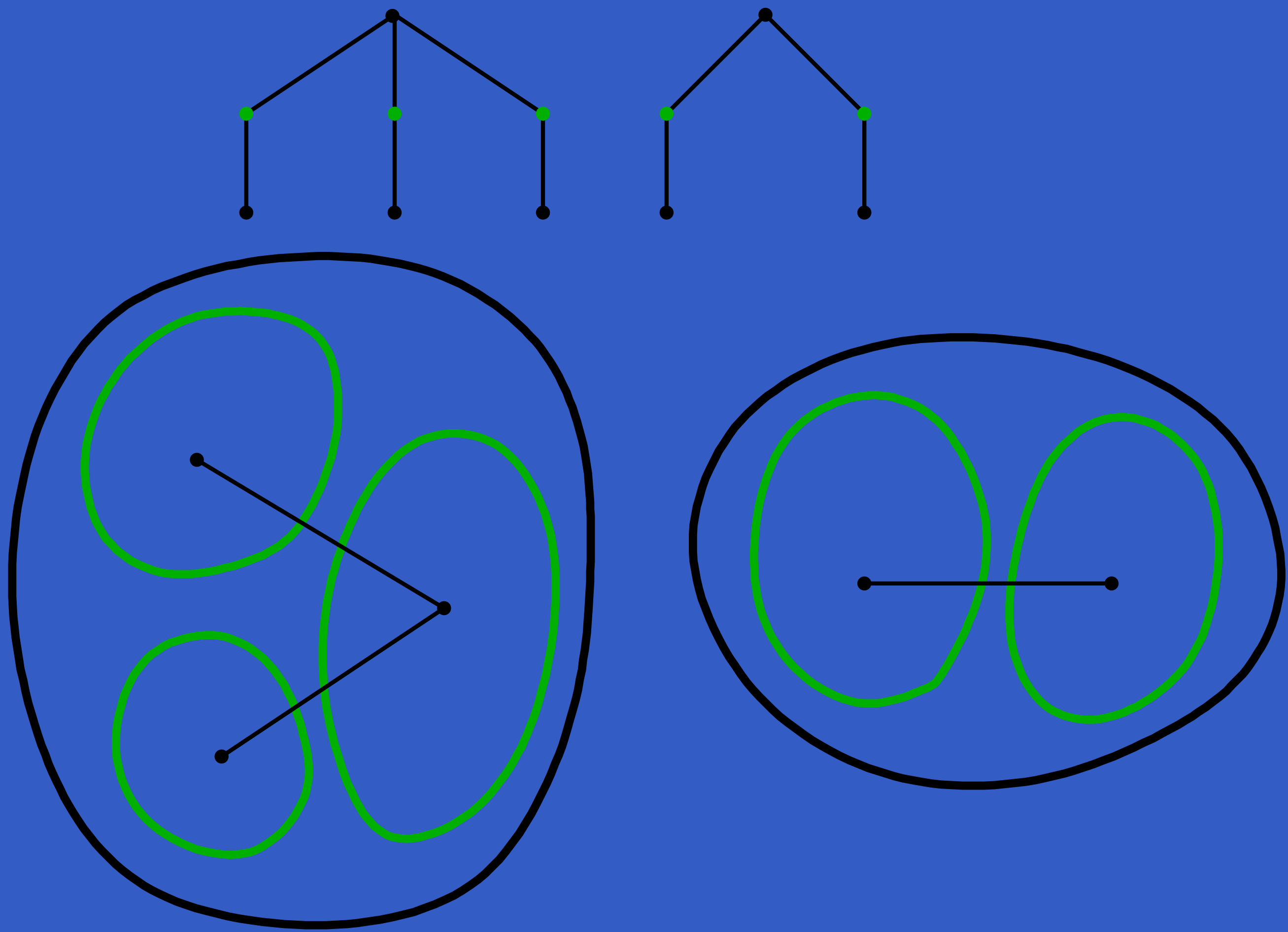
Handling an update $\texttt{insert}(u, v)$

- Initialize $\ell(u, v) \leftarrow 0$
- Let r_u resp. r_v be the root of the tree of $\mathcal{C}$ containing u resp. v
- If $r_u = r_v$, $\mathcal{C}$ need not be updated
- Otherwise, r_u and r_v are *merged* into r_u

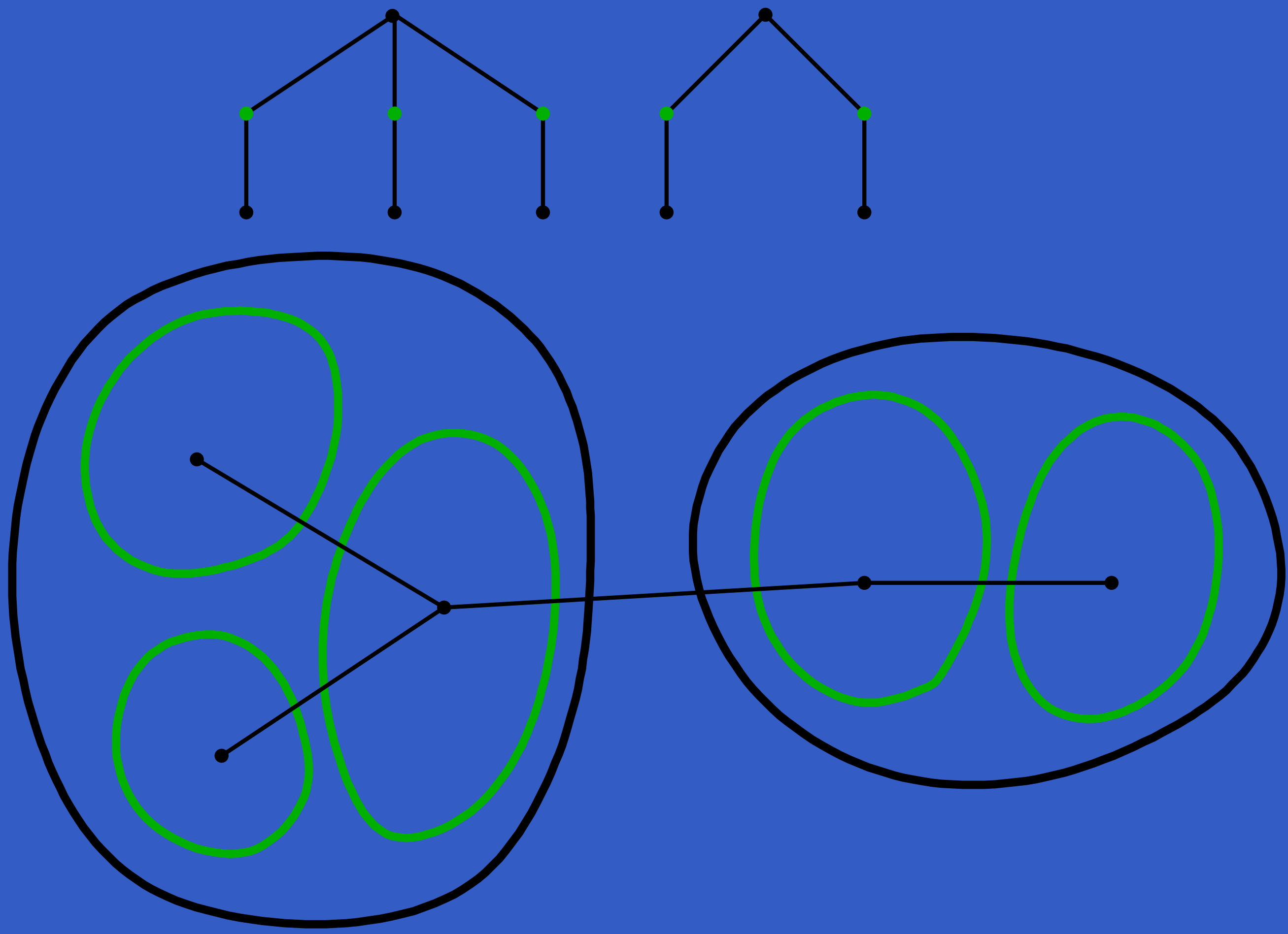
Handling an update $\texttt{insert}(u, v)$

- Initialize $\ell(u, v) \leftarrow 0$
- Let r_u resp. r_v be the root of the tree of $\mathcal{C}$ containing u resp. v
- If $r_u = r_v$, $\mathcal{C}$ need not be updated
- Otherwise, r_u and r_v are *merged* into r_u
- This corresponds to merging $C(r_u)$ and $C(r_v)$

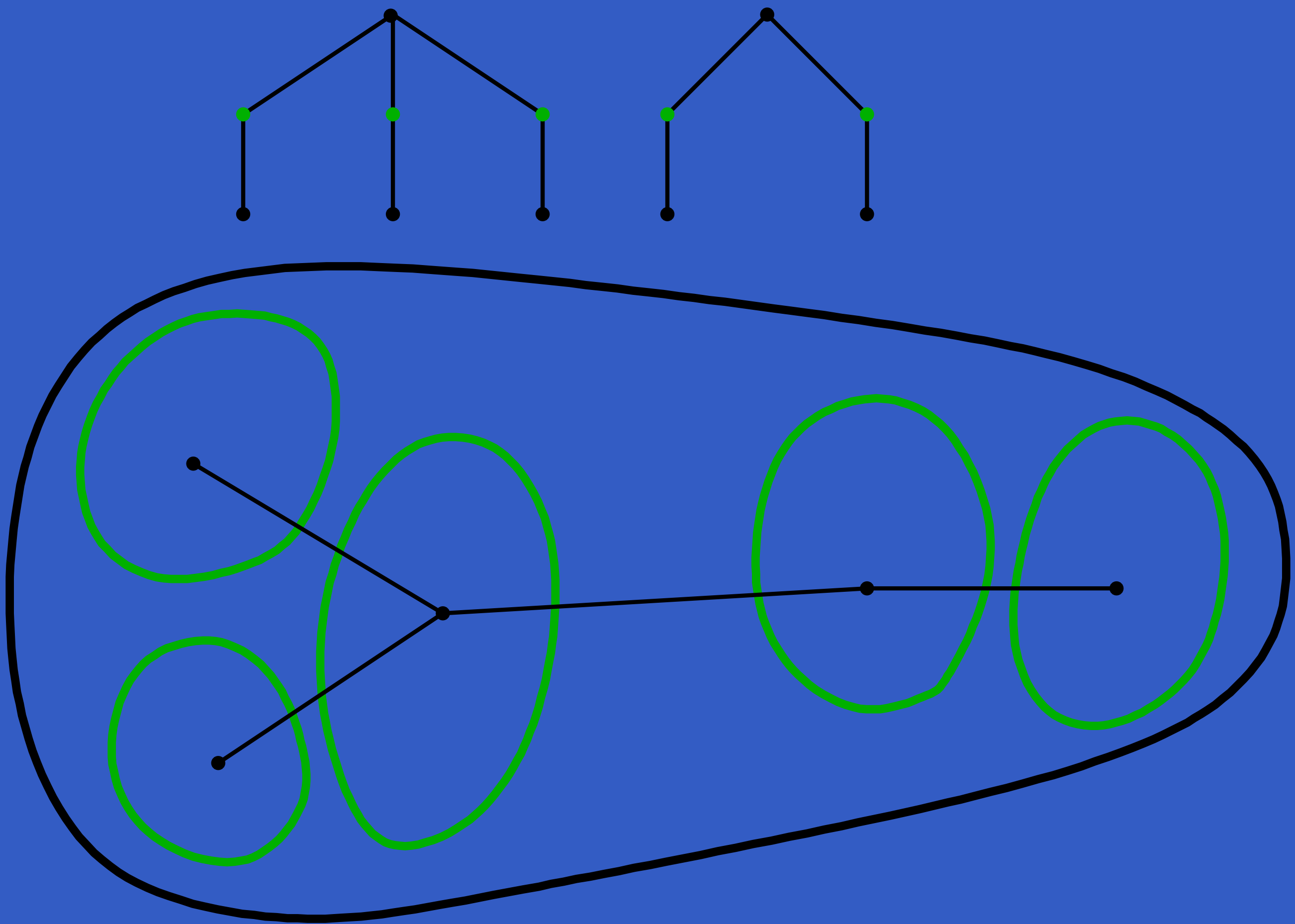
Handling an update `insert`(u, v)



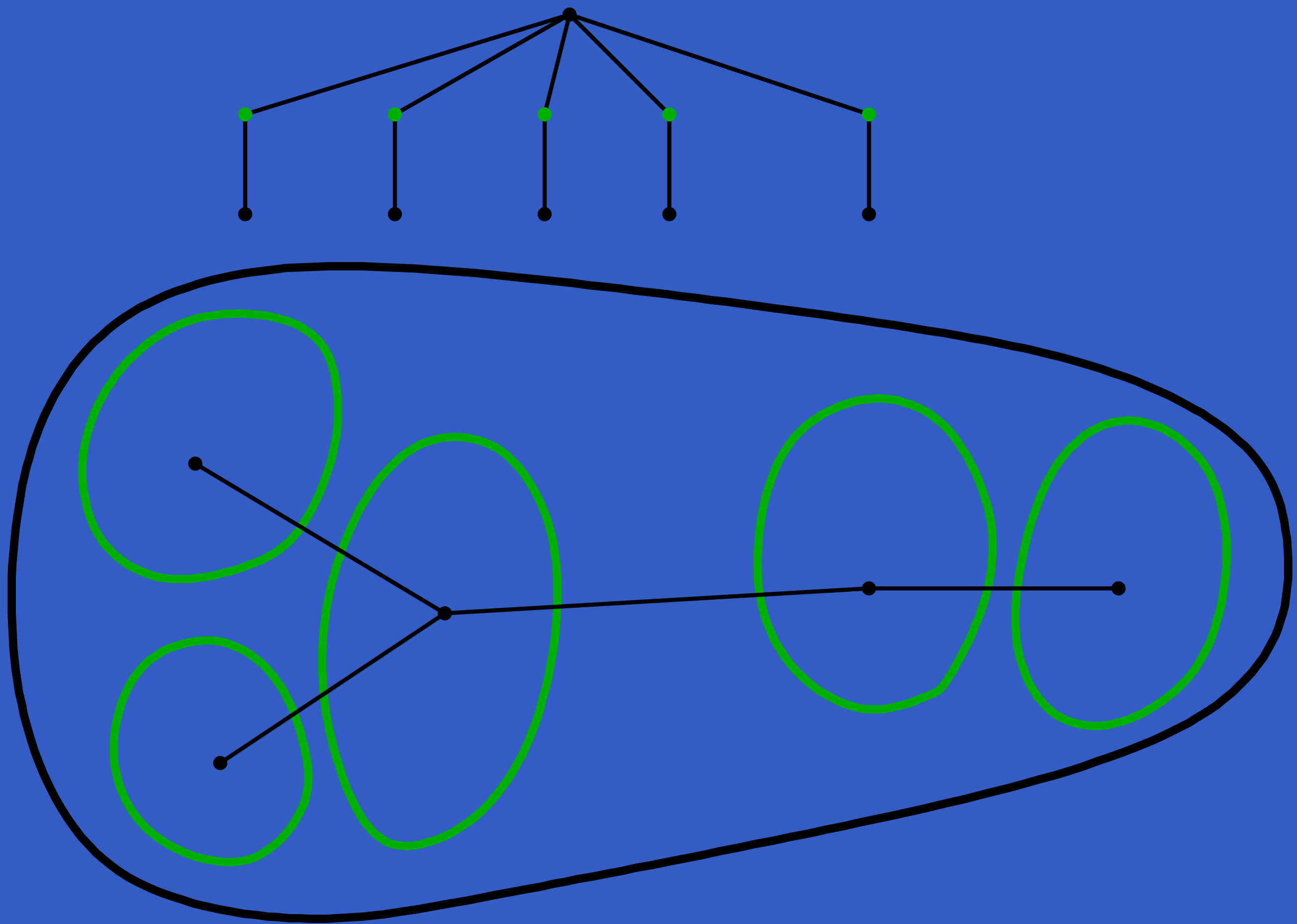
Handling an update $\texttt{insert}(u, v)$



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



Handling an update `insert`(u, v)



Handling an update $\texttt{delete}(u, v)$

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

Handling an update `delete`(u, v)

- Let $i = \ell(u, v)$ and let $C_u \neq C_v$ be the level $(i+1)$-clusters containing u and v
- Let M_i be the multigraph with level $(i+1)$-clusters as vertices and level i-edges of G as edges

Handling an update `delete`(u, v)

- Let $i = \ell(u, v)$ and let $C_u \neq C_v$ be the level $(i+1)$-clusters containing u and v
- Let M_i be the multigraph with level $(i+1)$-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 an update $\texttt{delete}(u, v)$

- Let $i = \ell(u, v)$ and let $C_u \neq C_v$ be the level $(i+1)$-clusters containing u and v
- Let M_i be the multigraph with level $(i+1)$-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:

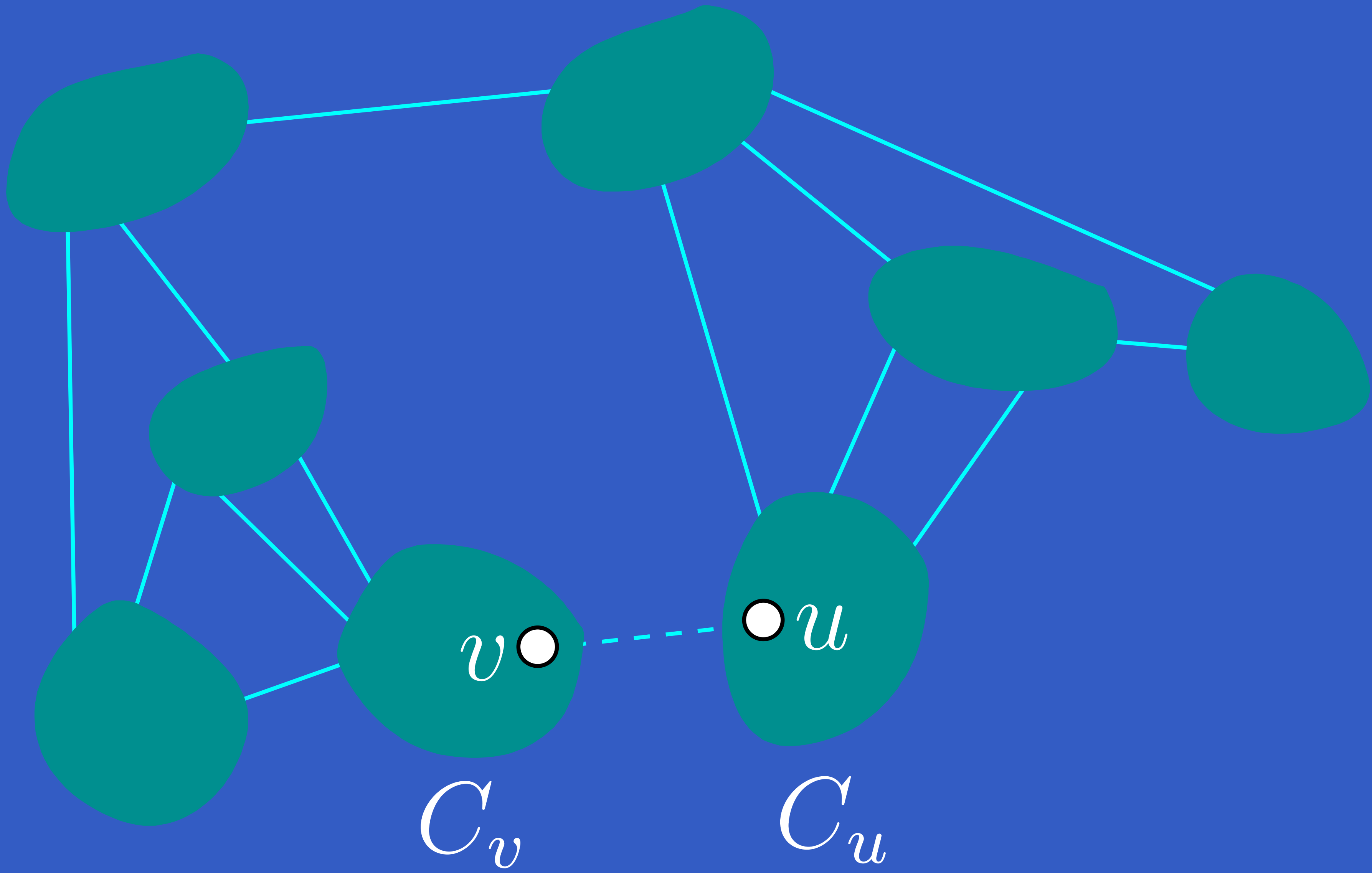
Handling an update `delete`(u, v)

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

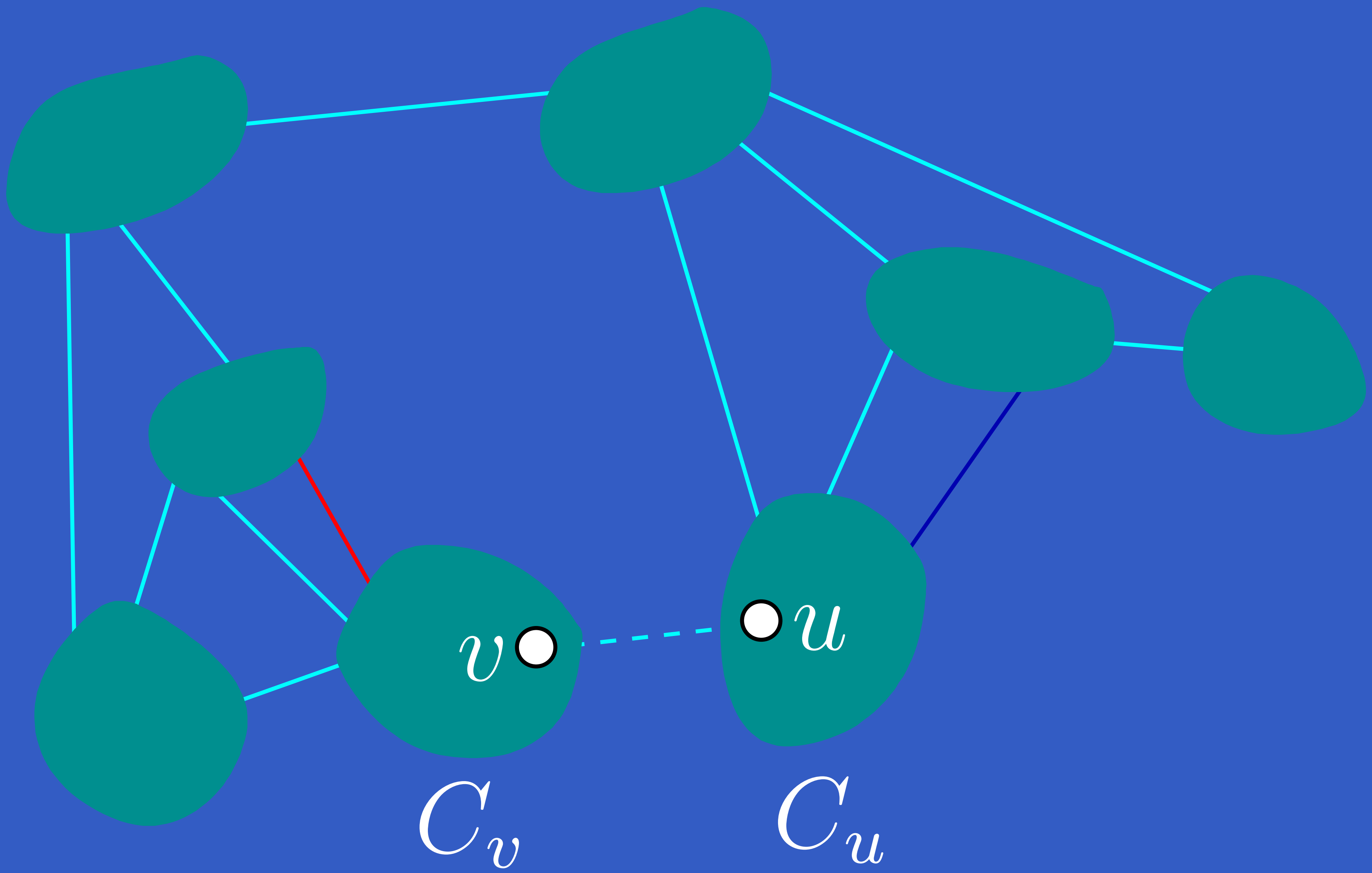
Handling an update `delete`(u, v)

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

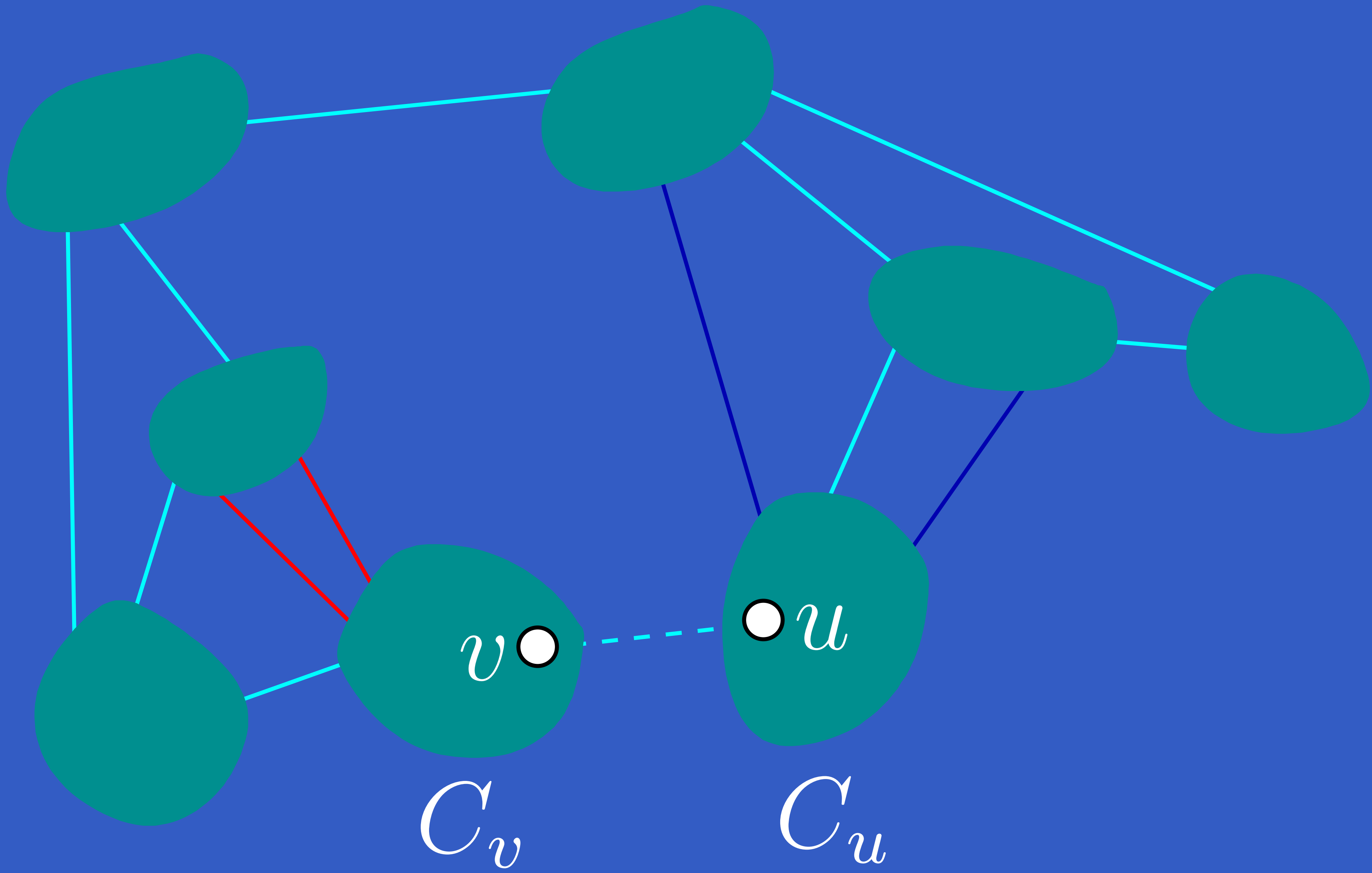
A vertex of M_i is explored by both search procedures



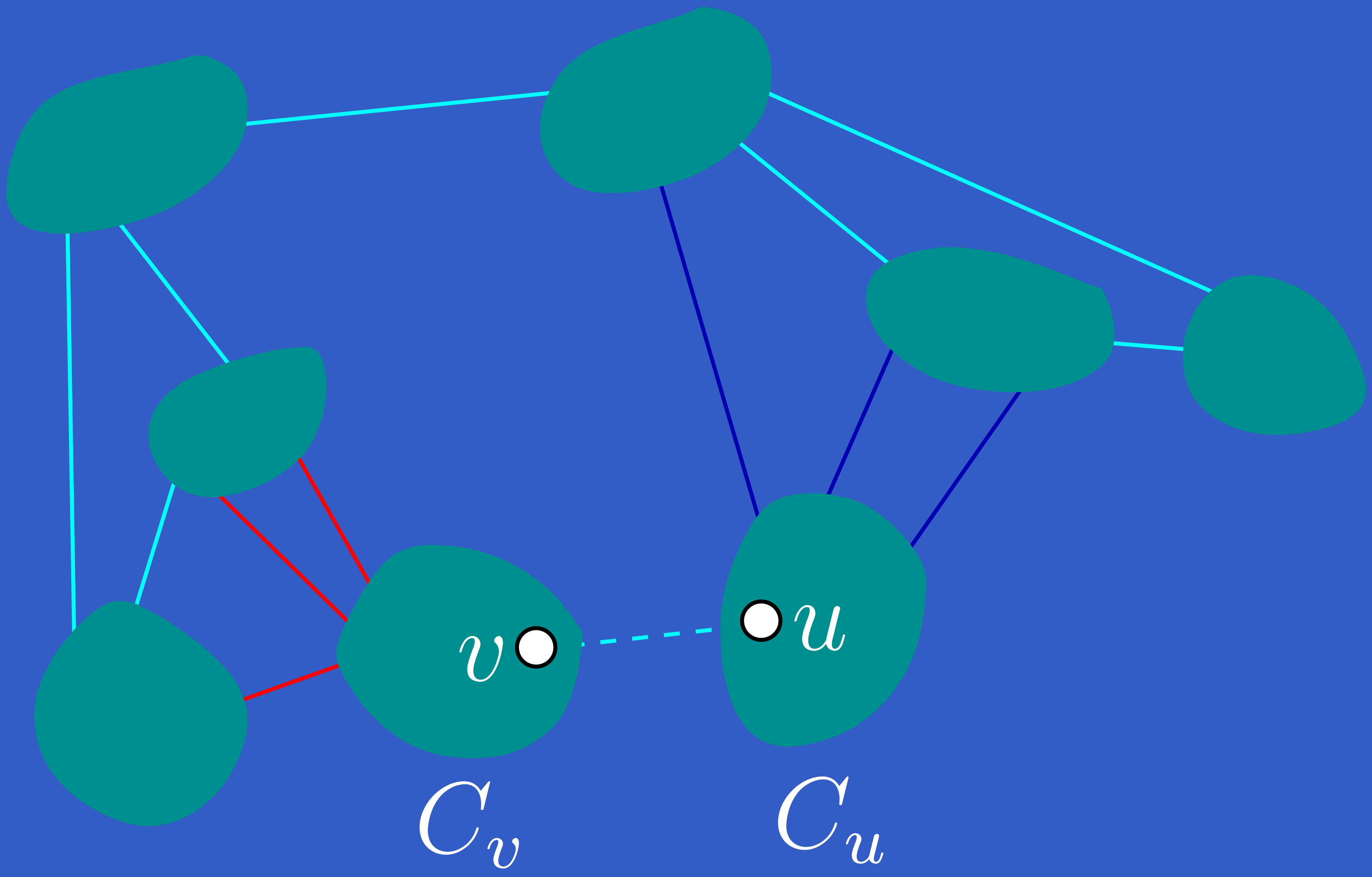
A vertex of M_i is explored by both search procedures



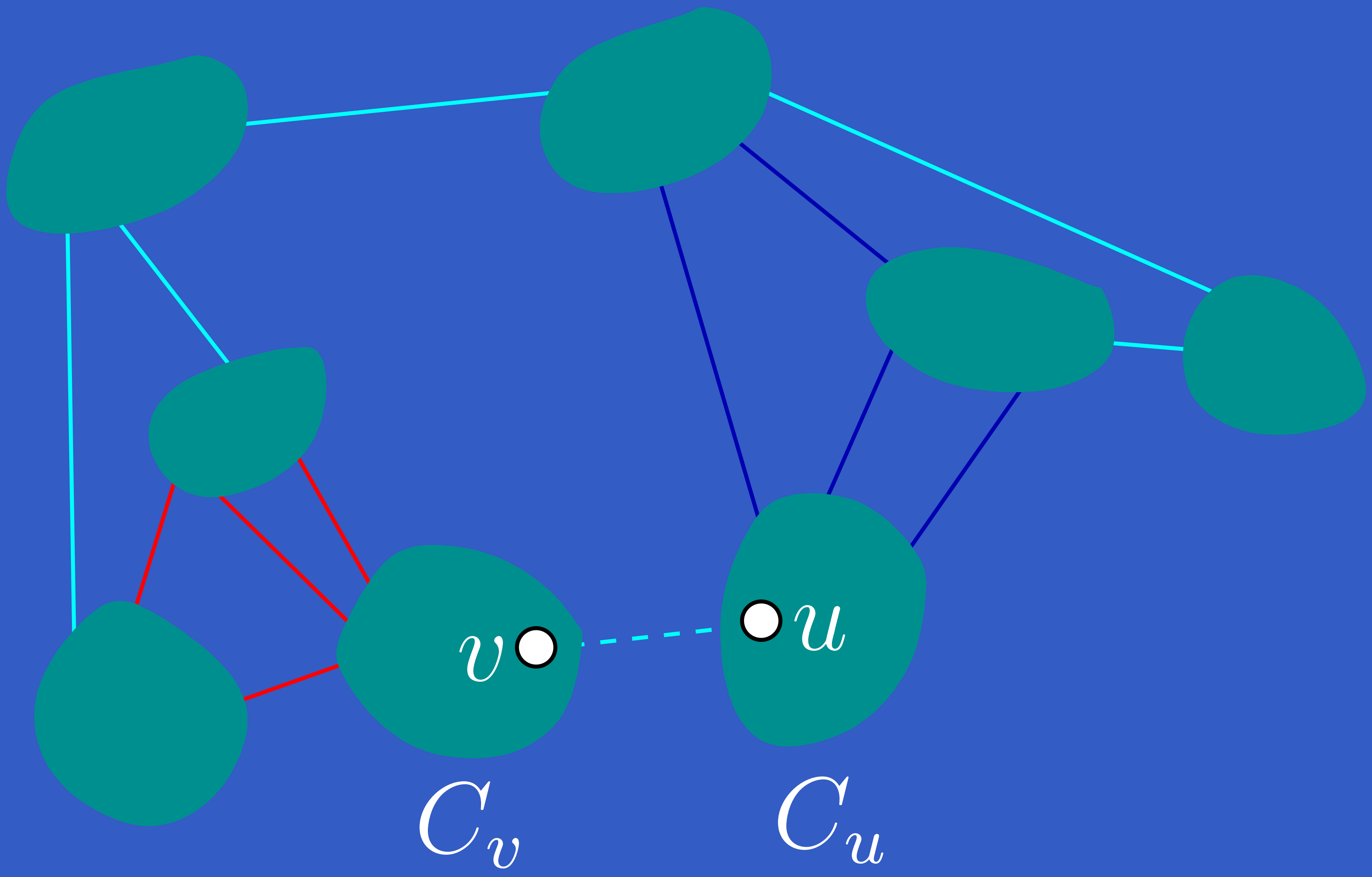
A vertex of M_i is explored by both search procedures



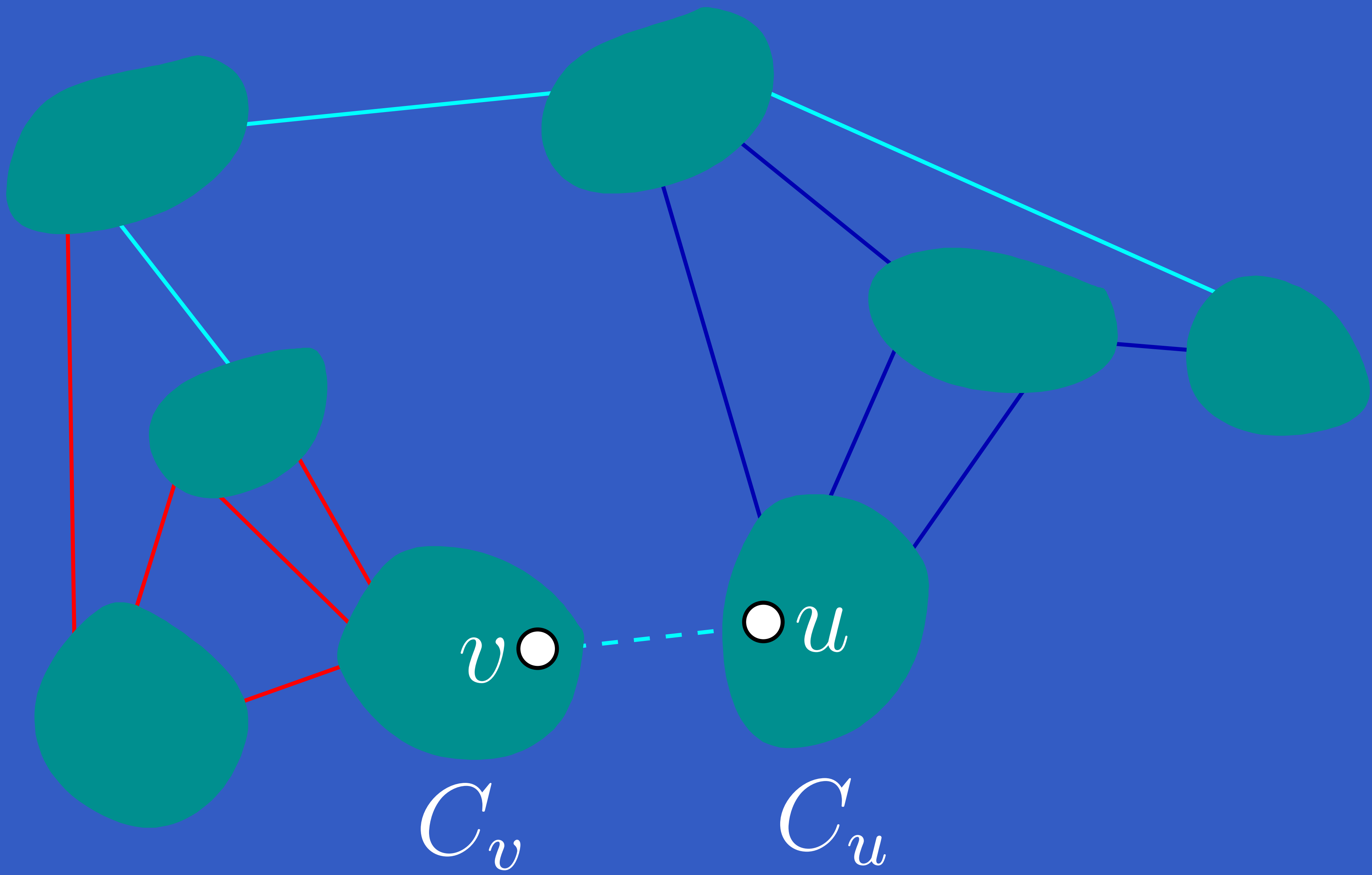
A vertex of M_i is explored by both search procedures



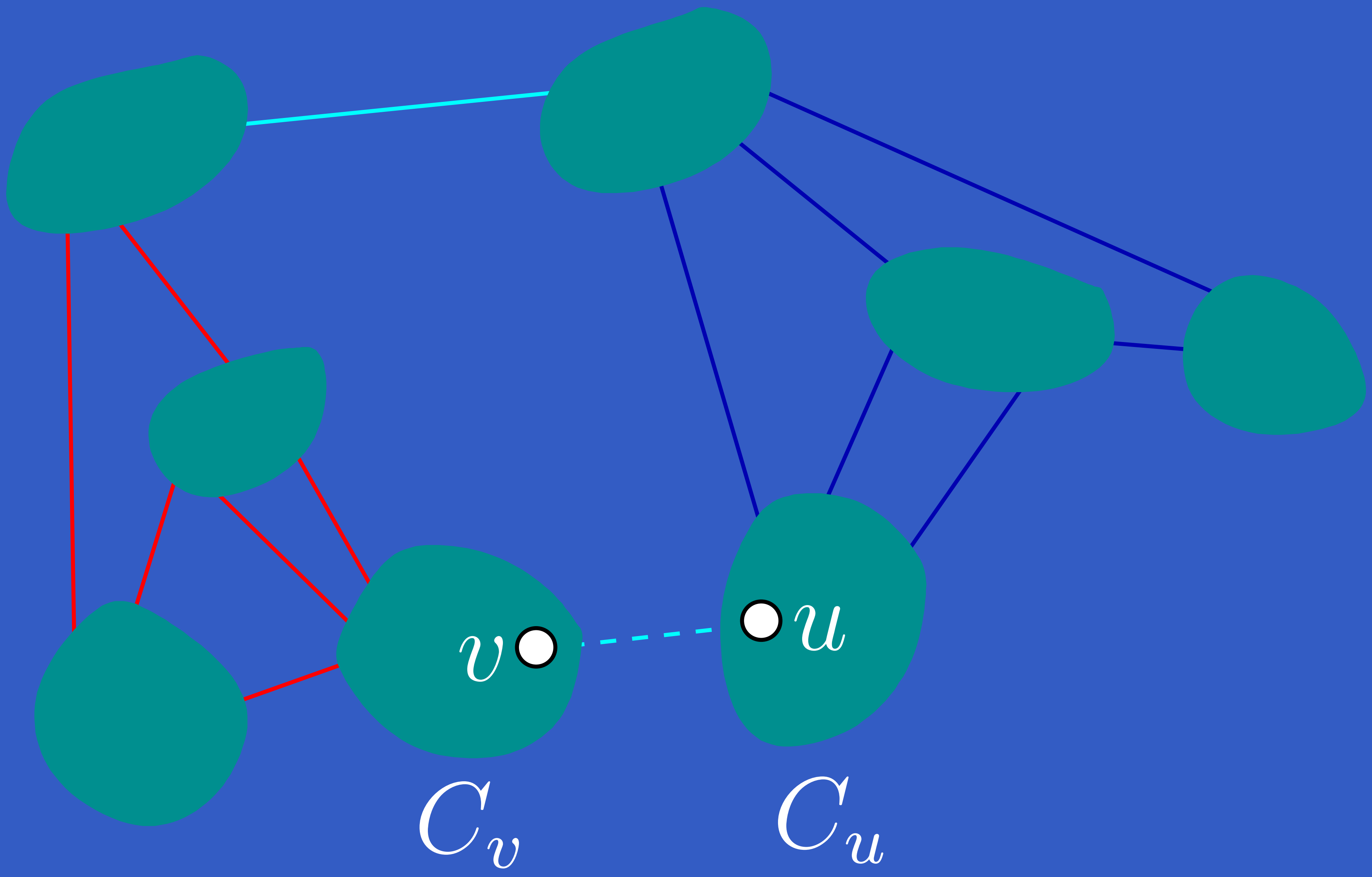
A vertex of M_i is explored by both search procedures



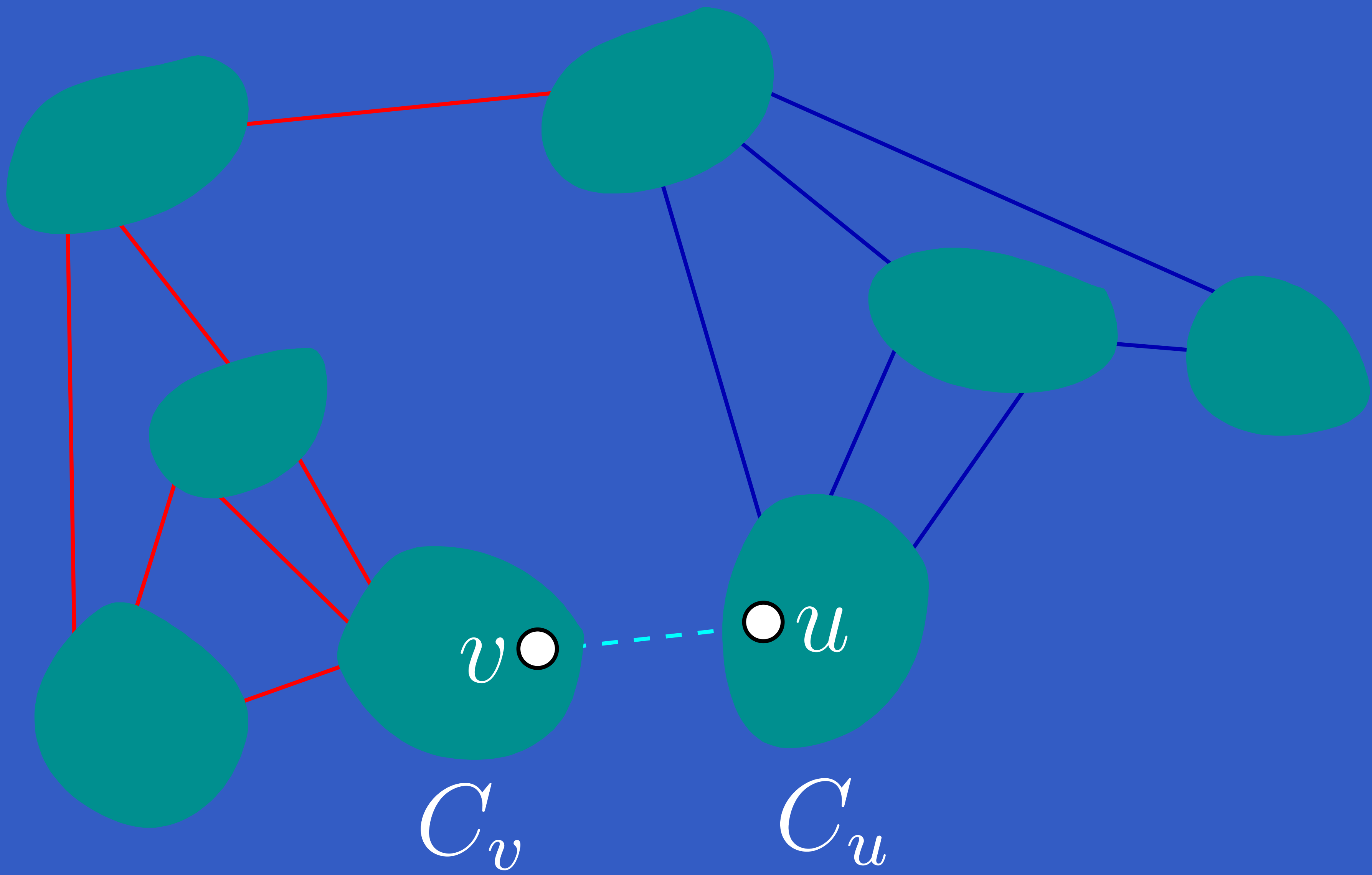
A vertex of M_i is explored by both search procedures



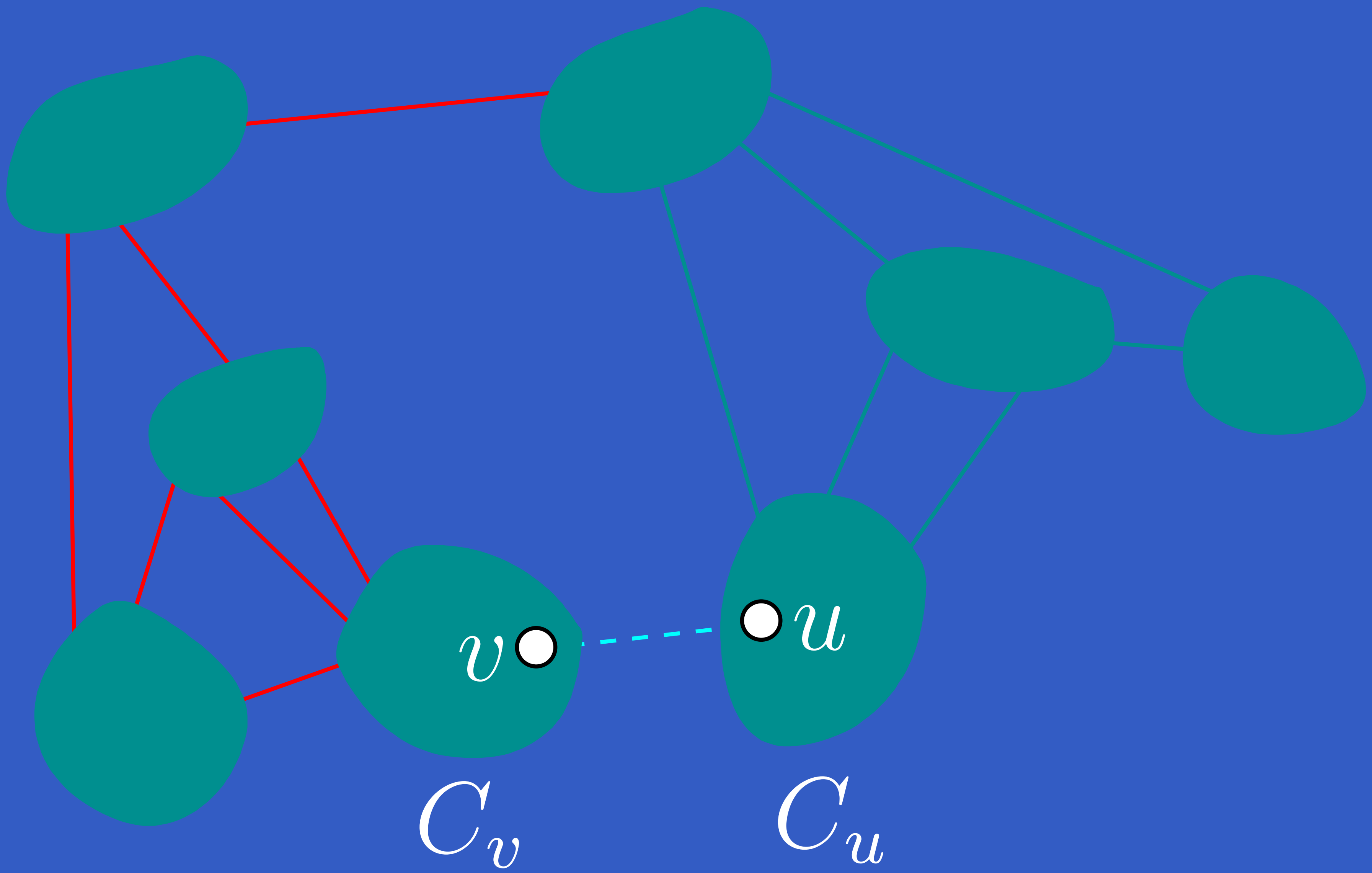
A vertex of M_i is explored by both search procedures



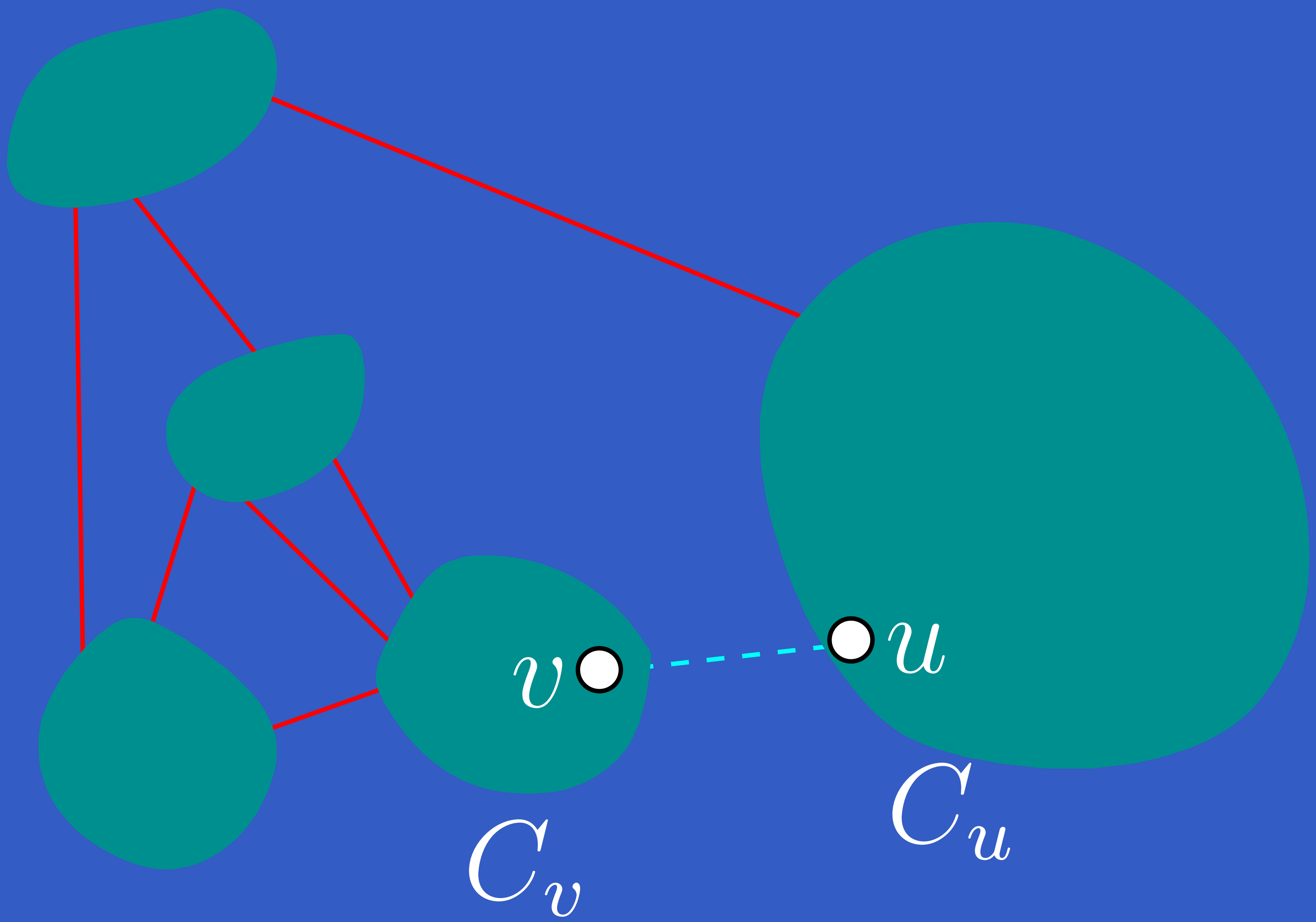
A vertex of M_i is explored by both search procedures



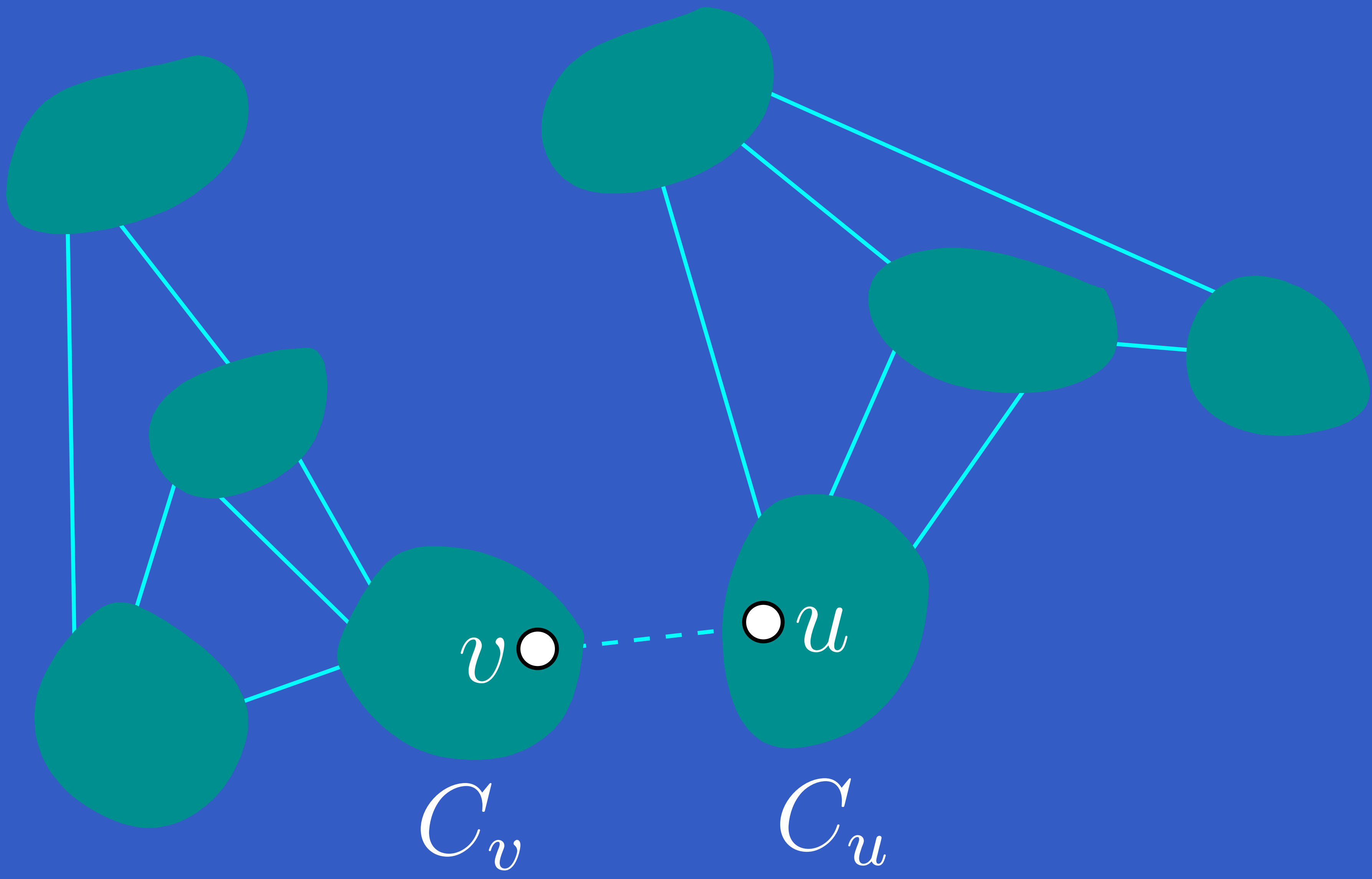
A vertex of M_i is explored by both search procedures



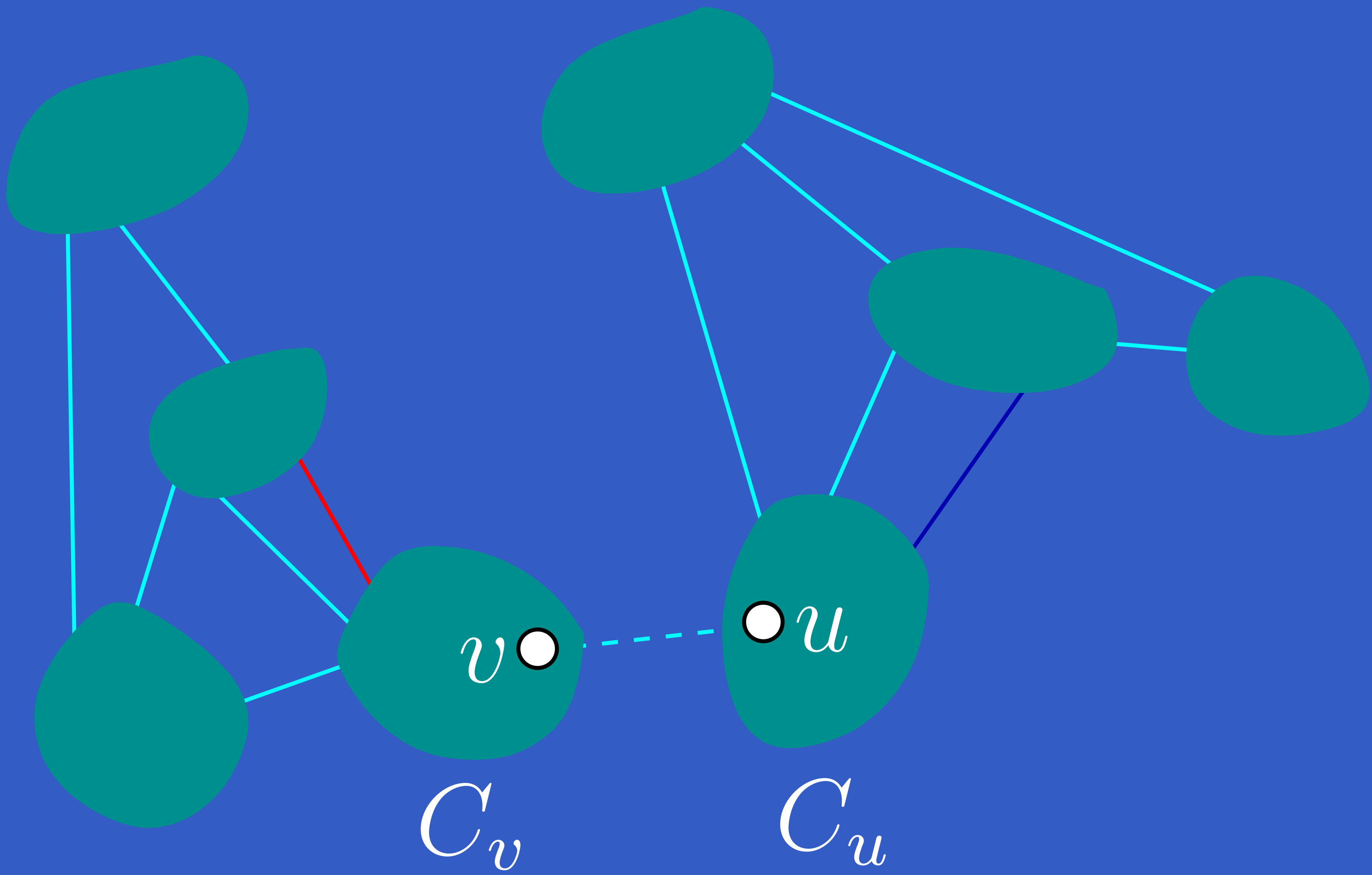
A vertex of M_i is explored by both search procedures



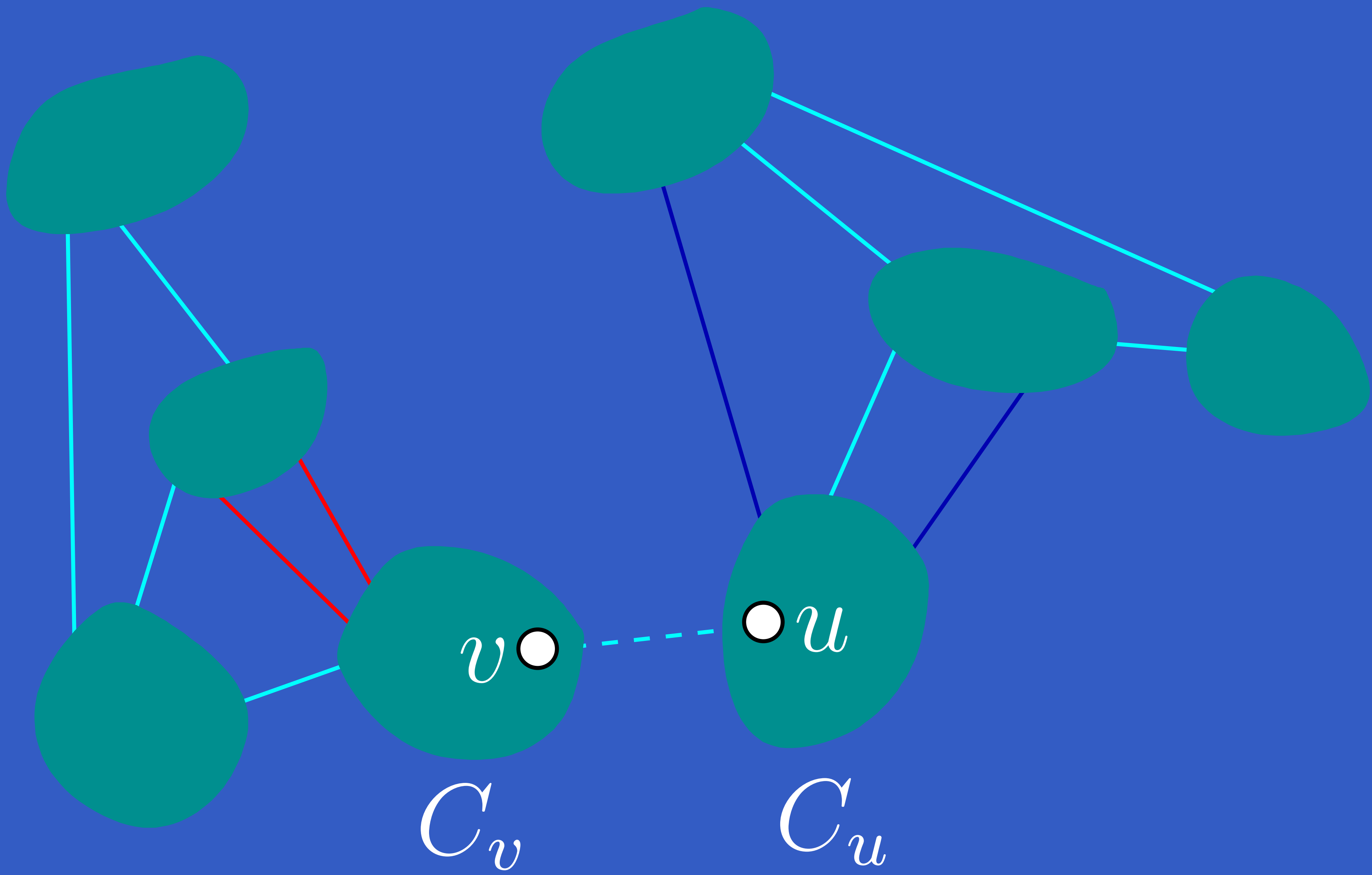
A search procedure has no more edges to explore



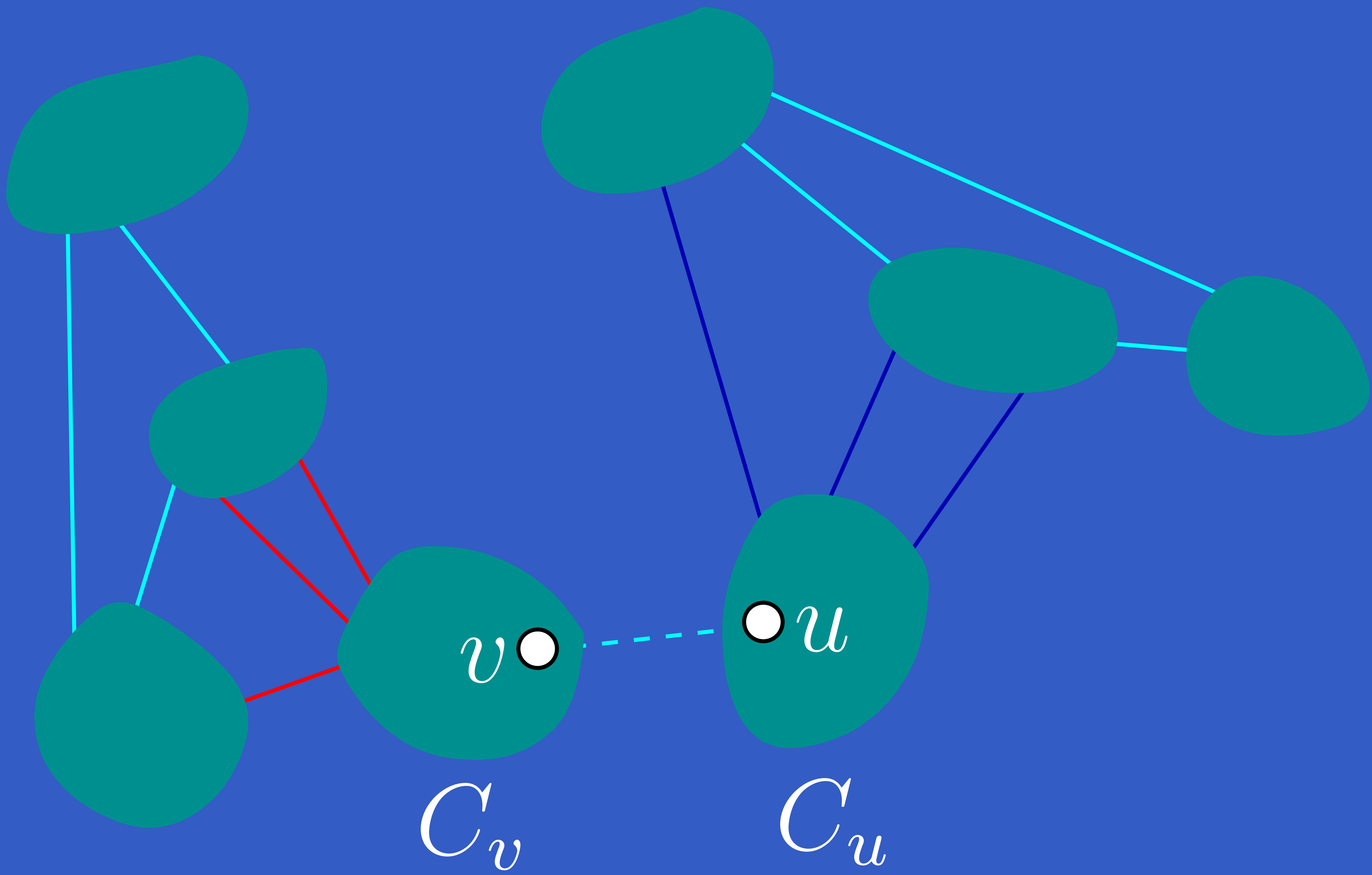
A search procedure has no more edges to explore



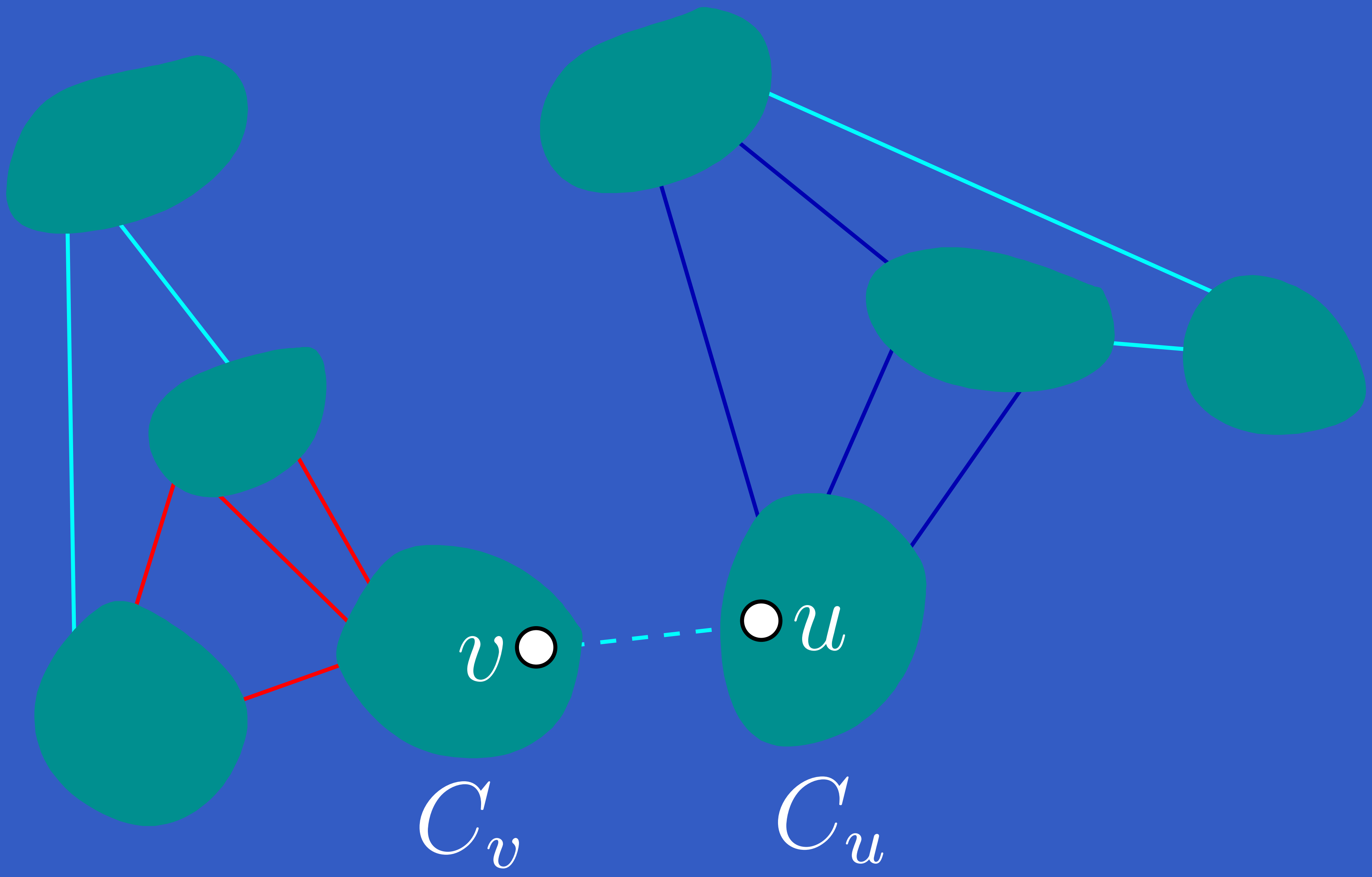
A search procedure has no more edges to explore



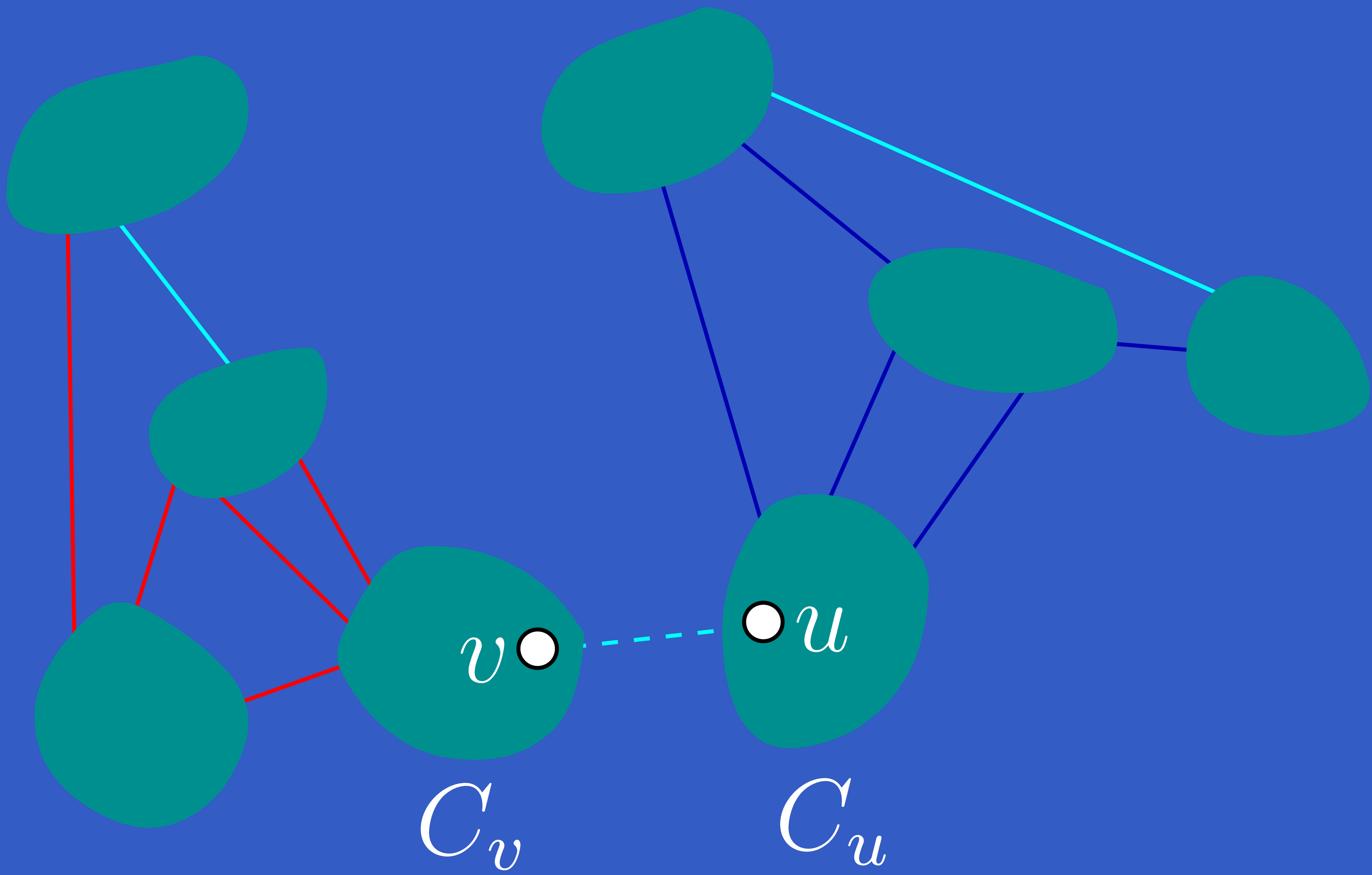
A search procedure has no more edges to explore



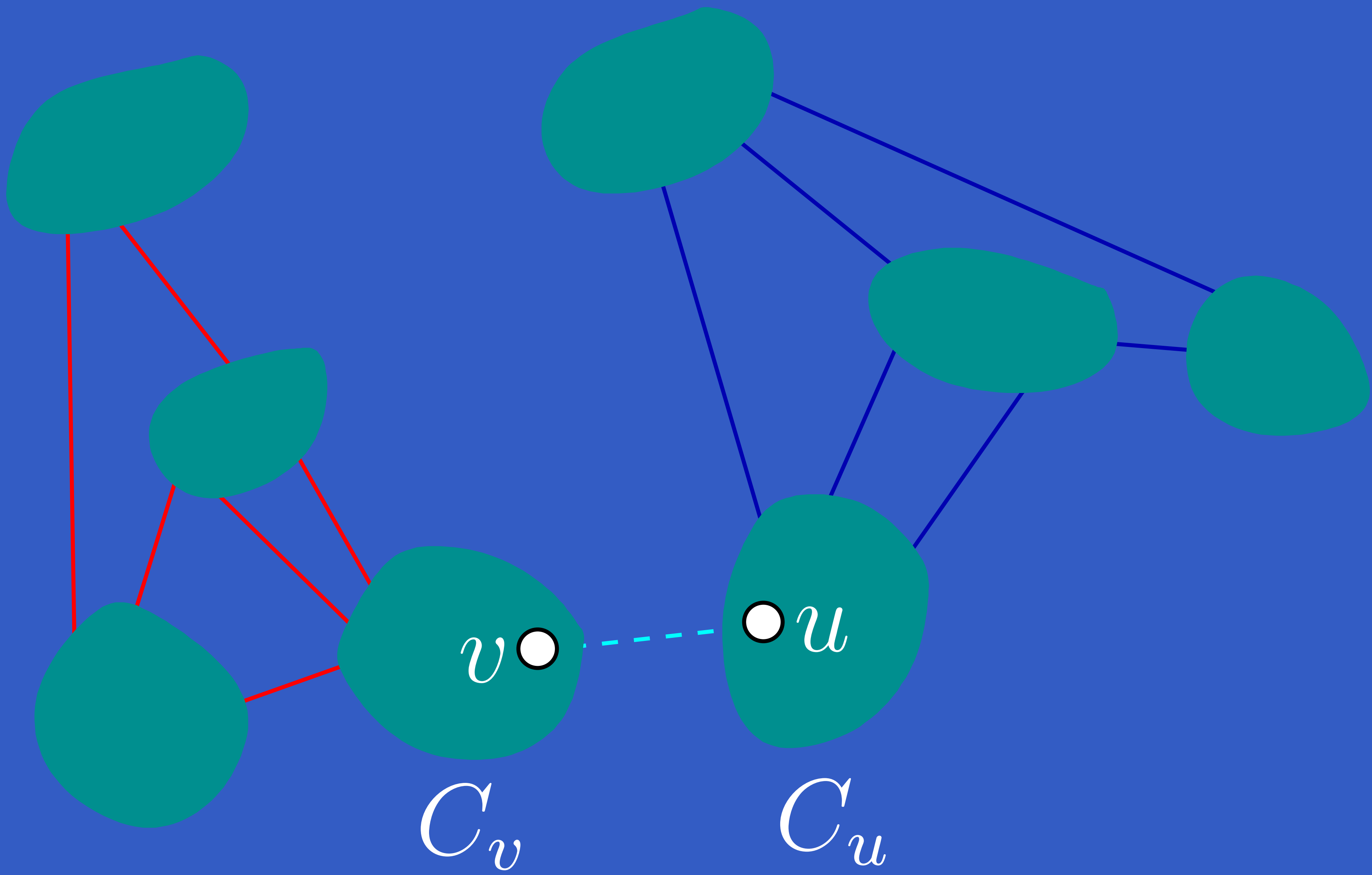
A search procedure has no more edges to explore



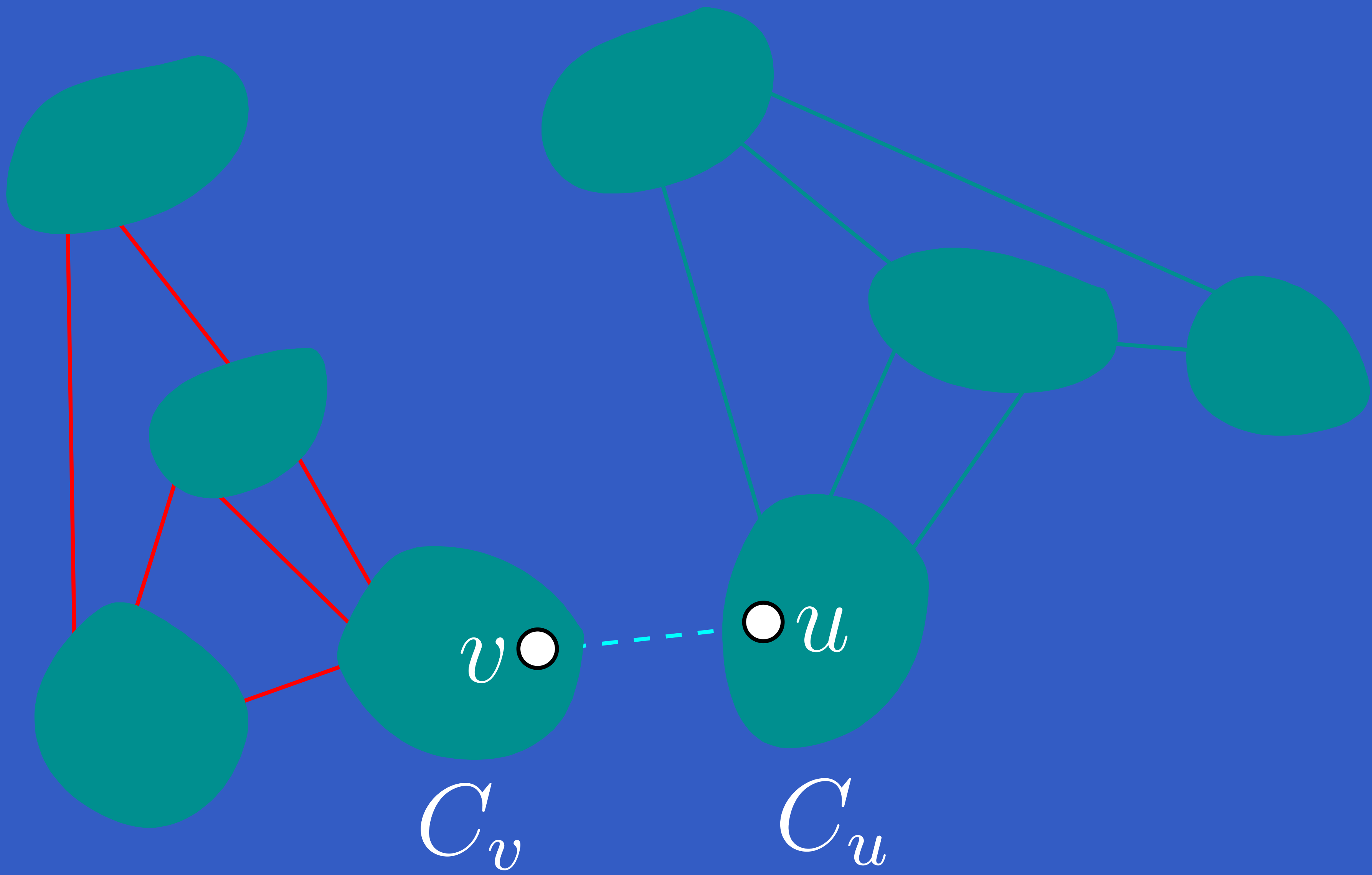
A search procedure has no more edges to explore



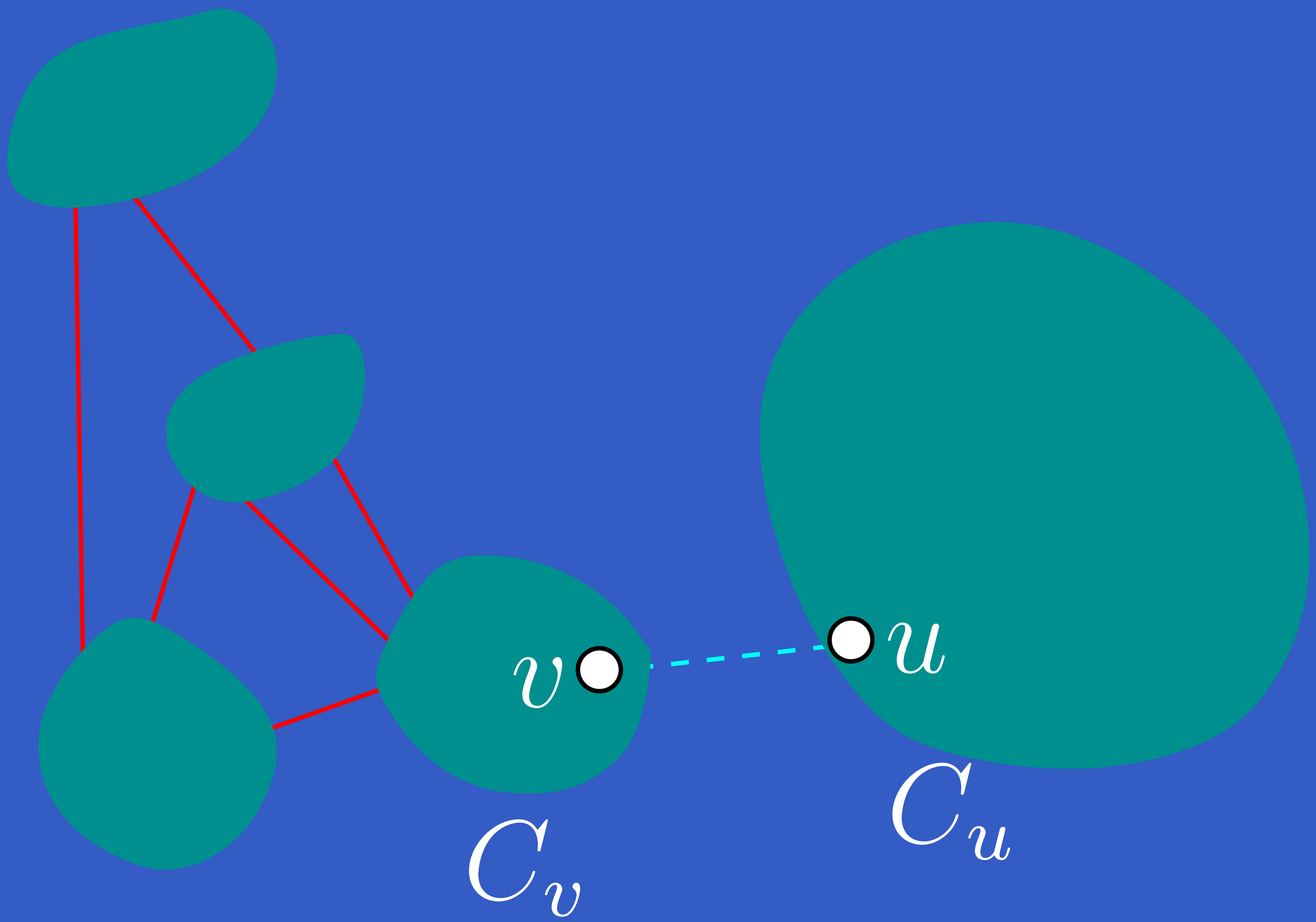
A search procedure has no more edges to explore



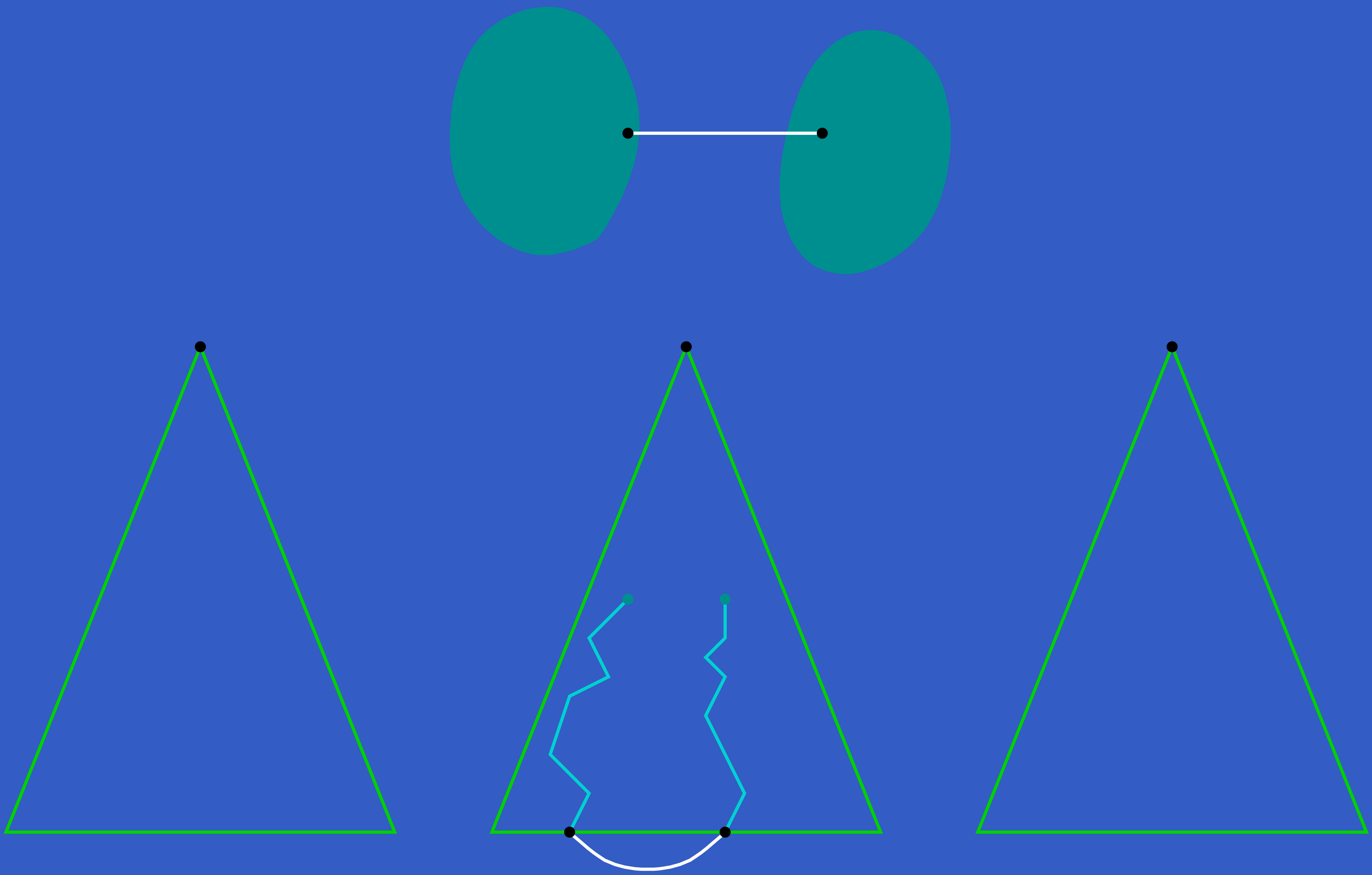
A search procedure has no more edges to explore



A search procedure has no more edges to explore



Traversing a single graph edge

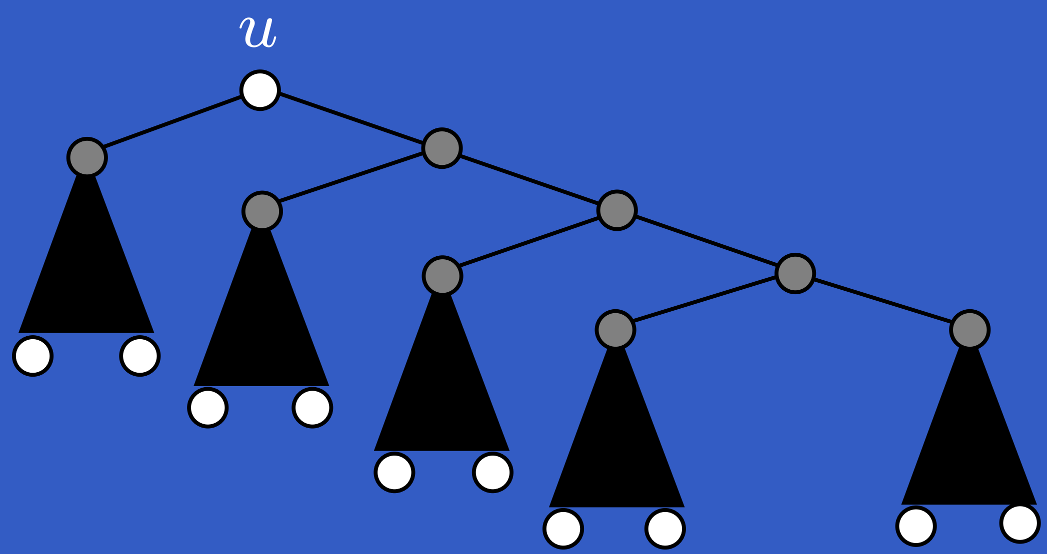


Local trees

- To search for graph edges of a specific level, we replace $\mathcal{C}$ by a forest of binary trees

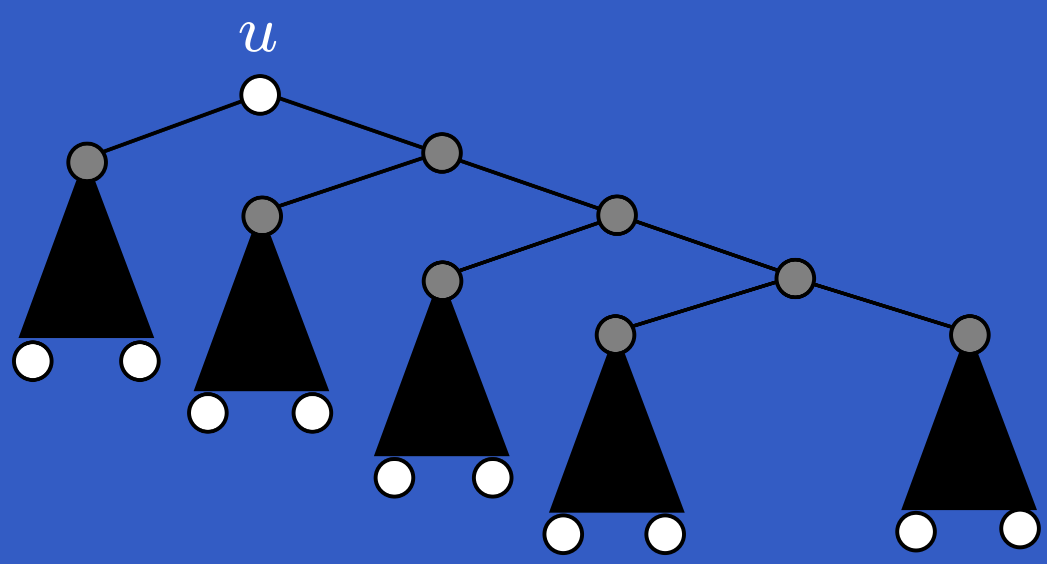
Local trees

- To search for graph edges of a specific level, we replace $\mathcal{C}$ by a forest of binary trees
- For each non-leaf node $u \in \mathcal{C}$, we replace the edges to its children by a *local tree* $L(u)$



Local trees

- To search for graph edges of a specific level, we replace $\mathcal{C}$ by a forest of binary trees
- For each non-leaf node $u \in \mathcal{C}$, we replace the edges to its children by a *local tree* $L(u)$



- Denote by $\mathcal{C}_L$ the resulting forest of binary trees; its height is $O(\log n)$

Searching for multigraph edges

- With each node $u \in \mathcal{C}_L$, associate a bitmap $\mathrm{edge}(u)$ where $\mathrm{edge}(u)[i] = 1$ iff a level i-edge is incident to a leaf of the subtree of $\mathcal{C}_L$ rooted at u

Searching for multigraph edges

- With each node $u \in \mathcal{C}_L$, associate a bitmap $\mathrm{edge}(u)$ where $\mathrm{edge}(u)[i] = 1$ iff a level i-edge is incident to a leaf of the subtree of $\mathcal{C}_L$ rooted at u
- We can use these bitmaps to search for level i-edges to be explored by our search procedures

Searching for multigraph edges

- With each node $u \in \mathcal{C}_L$, associate a bitmap $\mathrm{edge}(u)$ where $\mathrm{edge}(u)[i] = 1$ iff a level i-edge is incident to a leaf of the subtree of $\mathcal{C}_L$ rooted at u
- We can use these bitmaps to search for level i-edges to be explored by our search procedures
- Whenever an edge is explored, we move up $\mathcal{C}_L$ to identify the new level $(i+1)$-cluster visited

Searching for multigraph edges

- With each node $u \in \mathcal{C}_L$, associate a bitmap $\mathrm{edge}(u)$ where $\mathrm{edge}(u)[i] = 1$ iff a level i-edge is incident to a leaf of the subtree of $\mathcal{C}_L$ rooted at u
- We can use these bitmaps to search for level i-edges to be explored by our search procedures
- Whenever an edge is explored, we move up $\mathcal{C}_L$ to identify the new level $(i+1)$-cluster visited
- Since $\mathcal{C}_L$ has height $O(\log n)$, the search procedures run in $O(\log n)$ time per edge explored

Performance of simple data structure

- For each edge level increase, we spend $O(\log n)$ time

Performance of simple data structure

- For each edge level increase, we spend $O(\log n)$ time
- Total time charged to an edge: $O(\log^2 n)$

Performance of simple data structure

- For each edge level increase, we spend $O(\log n)$ time
- Total time charged to an edge: $O(\log^2 n)$
- $O(\log^2 n)$ amortized update time

Performance of simple data structure

- For each edge level increase, we spend $O(\log n)$ time
- Total time charged to an edge: $O(\log^2 n)$
- $O(\log^2 n)$ amortized update time
- Query time: $O(\log n)$

Performance of simple data structure

- For each edge level increase, we spend $O(\log n)$ time
- Total time charged to an edge: $O(\log^2 n)$
- $O(\log^2 n)$ amortized update time
- Query time: $O(\log n)$
- Goal: improve both bounds by a factor of $\log \log n$

Performance of simple data structure

- For each edge level increase, we spend $O(\log n)$ time
- Total time charged to an edge: $O(\log^2 n)$
- $O(\log^2 n)$ amortized update time
- Query time: $O(\log n)$
- Goal: improve both bounds by a factor of $\log \log n$
- Main ideas: add shortcuts to $\mathcal{C}_L$ and use *lazy* local trees

Shortcuts and our search procedure

- We add shortcuts each skipping order $\epsilon \log \log n$ nodes on a leaf-to-root path in $\mathcal{C}_L$

Shortcuts and our search procedure

- We add shortcuts each skipping order $\epsilon \log \log n$ nodes on a leaf-to-root path in $\mathcal{C}_L$
- Can be maintained efficiently

Shortcuts and our search procedure

- We add shortcuts each skipping order $\epsilon \log \log n$ nodes on a leaf-to-root path in $\mathcal{C}_L$
- Can be maintained efficiently
- With the shortcuts, we can traverse a leaf-to-root path in $\mathcal{C}_L$ in $O(\log n / \log \log n)$ time

Shortcuts and our search procedure

- We add shortcuts each skipping order $\epsilon \log \log n$ nodes on a leaf-to-root path in $\mathcal{C}_L$
- Can be maintained efficiently
- With the shortcuts, we can traverse a leaf-to-root path in $\mathcal{C}_L$ in $O(\log n / \log \log n)$ time
- Another system of shortcuts is used to move down $\mathcal{C}_L$ in $O(\log n / \log \log n)$ time per edge explored

Shortcuts and our search procedure

- We add shortcuts each skipping order $\epsilon \log \log n$ nodes on a leaf-to-root path in $\mathcal{C}_L$
- Can be maintained efficiently
- With the shortcuts, we can traverse a leaf-to-root path in $\mathcal{C}_L$ in $O(\log n / \log \log n)$ time
- Another system of shortcuts is used to move down $\mathcal{C}_L$ in $O(\log n / \log \log n)$ time per edge explored
- Hence, an edge pays a total of $O(\log^2 n / \log \log n)$

Shortcuts and our search procedure

- We add shortcuts each skipping order $\epsilon \log \log n$ nodes on a leaf-to-root path in $\mathcal{C}_L$
- Can be maintained efficiently
- With the shortcuts, we can traverse a leaf-to-root path in $\mathcal{C}_L$ in $O(\log n / \log \log n)$ time
- Another system of shortcuts is used to move down $\mathcal{C}_L$ in $O(\log n / \log \log n)$ time per edge explored
- Hence, an edge pays a total of $O(\log^2 n / \log \log n)$
- Problem: maintaining local trees is too expensive

Thorup's lazy local tree

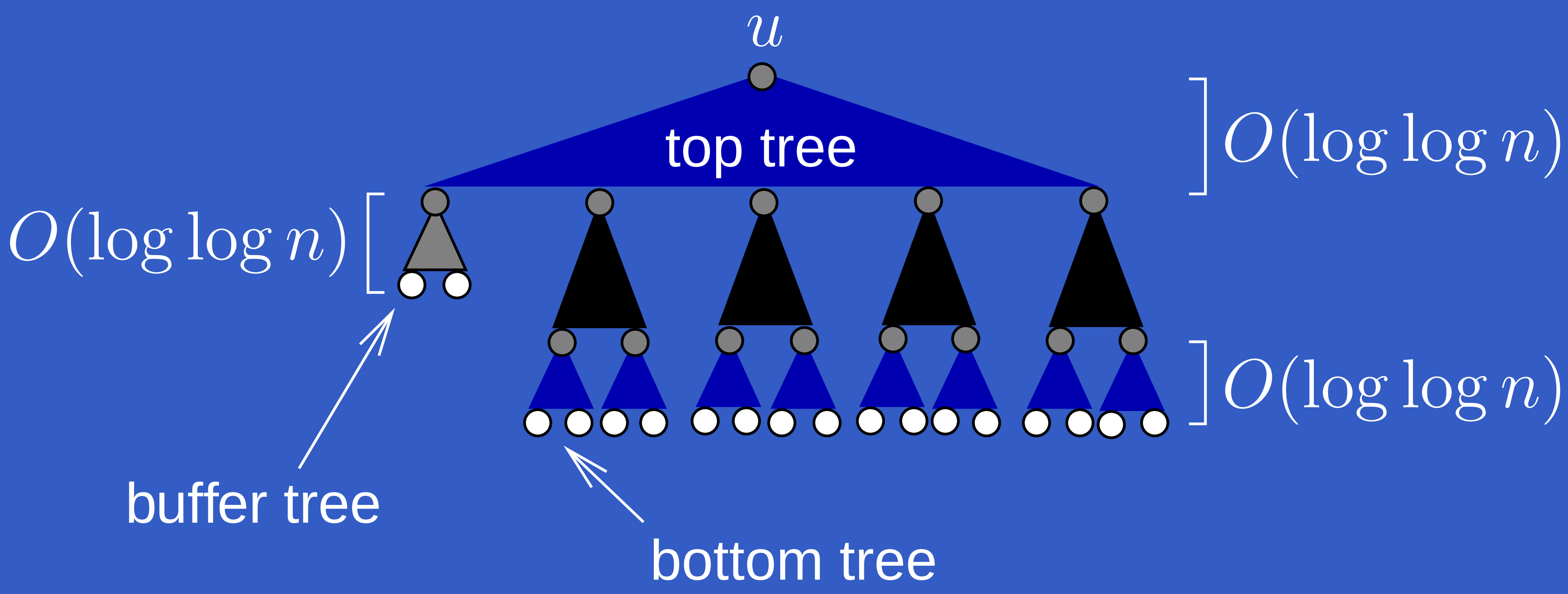
- To speed up tree updates, Thorup introduced the lazy local tree

Thorup's lazy local tree

- To speed up tree updates, Thorup introduced the lazy local tree
- Disadvantage: height of trees in $\mathcal{C}_L$ increases to $O(\log n \log \log n)$

Thorup's lazy local tree

- To speed up tree updates, Thorup introduced the lazy local tree
- Disadvantage: height of trees in $\mathcal{C}_L$ increases to $O(\log n \log\log n)$



New lazy local tree $L(u)$

- Hybrid between a local and a lazy local tree

New lazy local tree $L(u)$

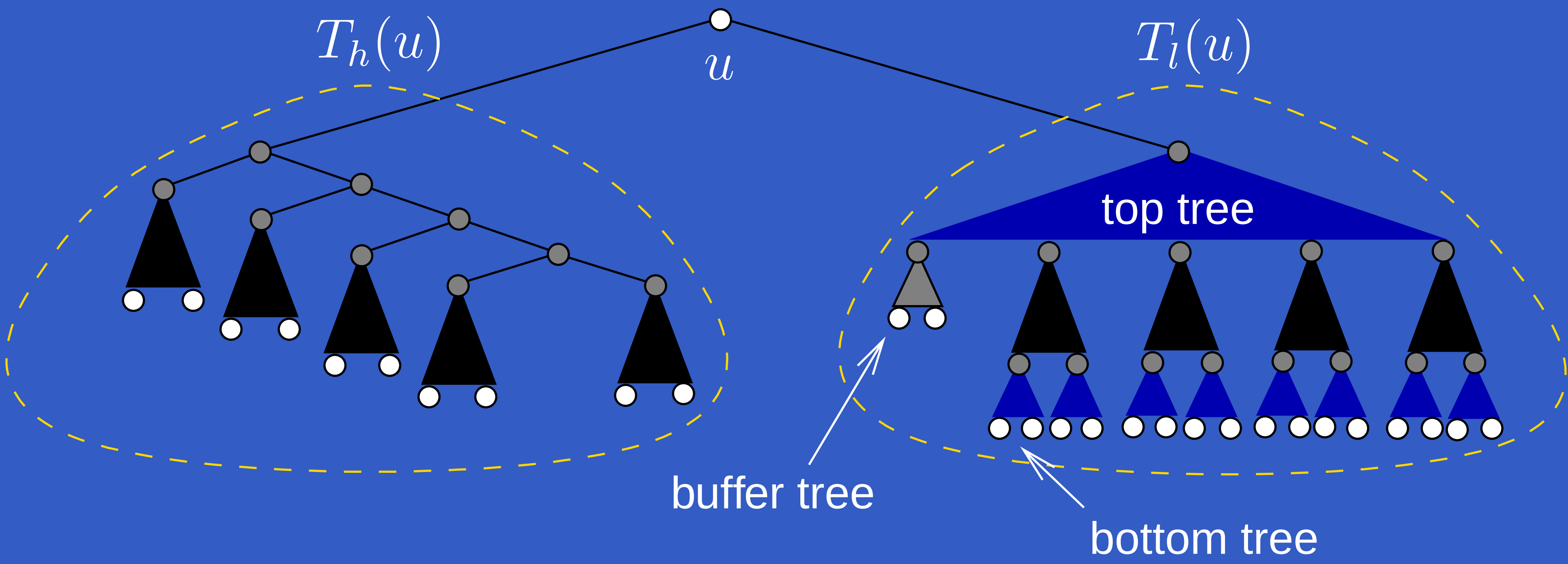
- Hybrid between a local and a lazy local tree
- Keep heavy children in a local tree of size $O(\log^\epsilon n)$

New lazy local tree $L(u)$

- Hybrid between a local and a lazy local tree
- Keep heavy children in a local tree of size $O(\log^{\epsilon} n)$
- Height of trees in $\mathcal{C}_L$: $O(\frac{1}{\epsilon} \log n)$

New lazy local tree $L(u)$

- Hybrid between a local and a lazy local tree
- Keep heavy children in a local tree of size $O(\log^{\epsilon} n)$
- Height of trees in $\mathcal{C}_L$: $O(\frac{1}{\epsilon}\log n)$



Concluding remarks

- We gave a deterministic data structure for fully-dynamic graph connectivity with $O(\log^2 n/\log\log n)$ update time and $O(\log n/\log\log n)$ query time

Concluding remarks

- We gave a deterministic data structure for fully-dynamic graph connectivity with $O(\log^2 n / \log\log n)$ update time and $O(\log n / \log\log n)$ query time
- This improves the update time of the deterministic data structures of Holm, de Lichtenberg, and Thorup by a factor of $\log\log n$

Concluding remarks

- We gave a deterministic data structure for fully-dynamic graph connectivity with $O(\log^2 n / \log\log n)$ update time and $O(\log n / \log\log n)$ query time
- This improves the update time of the deterministic data structures of Holm, de Lichtenberg, and Thorup by a factor of $\log\log n$
- Does improvement extend to fully-dynamic MSF?

Concluding remarks

- We gave a deterministic data structure for fully-dynamic graph connectivity with $O(\log^2 n / \log\log n)$ update time and $O(\log n / \log\log n)$ query time
- This improves the update time of the deterministic data structures of Holm, de Lichtenberg, and Thorup by a factor of $\log\log n$
- Does improvement extend to fully-dynamic MSF?
- $O(\log n)$ time for both updates and queries?