Advancements in Graph Bandwidth Reduction

Luis M. B. Varona[1,2,3] and Nathaniel Johnston[2]

Mount Allison University

[1]Dept. of Politics & International Relations, MtA
[2]Dept. of Mathematics & Computer Science, MtA
[3]Dept. of Economics, MtA

Science Atlantic – Mathematics, Statistics, & Computer Science (MSCS) 2025
Cape Breton University, Sydney, NS, Canada

October 18, 2025

Graph Bandwidth

Definition (Layout)

Let $G = (V, E)$ be a graph with $|V| = n$. A* ***layout*** *of G is a bijection from $V \hookrightarrow\!\!\!\!\rightarrow \{0, 1, \dots, n-1\}$.*

Definition (Graph bandwidth)

Let $G = (V, E)$ be a graph with $|V| = n$. The* ***bandwidth*** *$\beta(G, \pi)$ of G under a layout π is the maximum $k \in \{0, 1, \dots, n-1\}$ such that $|\pi(u) - \pi(v)| \leq k$ for all $\{u, v\} \in E$. The* ***minimum bandwidth*** *$\beta(G)$ of G is $\min \beta(G, \pi)$ taken over all layouts π.*

- Empty graphs: $\beta\left(\overline{K_n}\right) = 0$
- Path graphs: $\beta(P_n) = 1$
- Complete graphs: $\beta(K_n) = n - 1$

*We use "graph" to refer specifically to a **simple undirected graph**.

Graph Bandwidth

Definition (Layout)

Let $G = (V, E)$ be a graph with $|V| = n$. A **layout** of G is a bijection from $V \hookrightarrow\!\!\!\!\rightarrow \{0, 1, \ldots, n-1\}$.*

Definition (Graph bandwidth)

Let $G = (V, E)$ be a graph with $|V| = n$. The **bandwidth** $\beta(G, \pi)$ of G under a layout π is the maximum $k \in \{0, 1, \ldots, n-1\}$ such that $|\pi(u) - \pi(v)| \leq k$ for all $\{u, v\} \in E$. The **minimum bandwidth** $\beta(G)$ of G is $\min \beta(G, \pi)$ taken over all layouts π.*

- Empty graphs: $\beta\left(\overline{K_n}\right) = 0$
- Path graphs: $\beta(P_n) = 1$
- Complete graphs: $\beta(K_n) = n - 1$

*We use "graph" to refer specifically to a **simple undirected graph**.

Graph Bandwidth

Definition (Layout)

*Let $G = (V, E)$ be a graph** *with* $|V| = n$. *A* ***layout*** *of G is a bijection from* $V \hookrightarrow\!\!\!\!\rightarrow \{0, 1, \ldots, n-1\}$.

Definition (Graph bandwidth)

*Let $G = (V, E)$ be a graph** *with* $|V| = n$. *The* ***bandwidth*** $\beta(G, \pi)$ *of G under a layout π is the maximum* $k \in \{0, 1, \ldots, n-1\}$ *such that* $|\pi(u) - \pi(v)| \leq k$ *for all* $\{u, v\} \in E$. *The* ***minimum bandwidth*** $\beta(G)$ *of G is* $\min \beta(G, \pi)$ *taken over all layouts* π.

- Empty graphs: $\beta\left(\overline{K_n}\right) = 0$
- Path graphs: $\beta(P_n) = 1$
- Complete graphs: $\beta(K_n) = n - 1$

*We use "graph" to refer specifically to a **simple undirected graph**.

Matrix Bandwidth

Theoreticians usually think in terms of graphs, but computational scientists often formulate bandwidth as a matrix-theoretic concept:

Definition (Matrix bandwidth)

Let A be an $n \times n$ matrix. The ***bandwidth*** $\beta(A, P)$ *of A under a permutation matrix P is the maximum $k \in \{0, 1, \ldots, n-1\}$ such that $[PAP^\mathsf{T}]_{i,j} \neq 0 \implies |i - j| \leq k$. The* ***minimum bandwidth*** $\beta(A)$ *of A is $\min \beta(A, P)$ taken over all permutation matrices P.*

- Often, A is **structurally symmetric**: $A_{i,j} = 0 \iff A_{j,i} = 0$
- If A is the adjacency matrix of a graph G (ignoring weights and diagonal entries), then $\beta(A) = \beta(G)$

Matrix Bandwidth

Theoreticians usually think in terms of graphs, but computational scientists often formulate bandwidth as a matrix-theoretic concept:

Definition (Matrix bandwidth)

Let A be an $n \times n$ matrix. The ***bandwidth*** *$\beta(A, P)$ of A under a permutation matrix P is the maximum $k \in \{0, 1, \ldots, n-1\}$ such that $[PAP^\mathsf{T}]_{i,j} \neq 0 \implies |i-j| \leq k$. The* ***minimum bandwidth*** *$\beta(A)$ of A is $\min \beta(A, P)$ taken over all permutation matrices P.*

- Often, A is **structurally symmetric**: $A_{i,j} = 0 \iff A_{j,i} = 0$
- If A is the adjacency matrix of a graph G (ignoring weights and diagonal entries), then $\beta(A) = \beta(G)$

Matrix Bandwidth

Theoreticians usually think in terms of graphs, but computational scientists often formulate bandwidth as a matrix-theoretic concept:

Definition (Matrix bandwidth)

Let A be an $n \times n$ matrix. The ***bandwidth*** *$\beta(A, P)$ of A under a permutation matrix P is the maximum $k \in \{0, 1, \ldots, n-1\}$ such that $[PAP^\mathsf{T}]_{i,j} \neq 0 \implies |i-j| \leq k$. The* ***minimum bandwidth*** *$\beta(A)$ of A is $\min \beta(A, P)$ taken over all permutation matrices P.*

- Often, A is **structurally symmetric**: $A_{i,j} = 0 \iff A_{j,i} = 0$
- If A is the adjacency matrix of a graph G (ignoring weights and diagonal entries), then $\beta(A) = \beta(G)$

Matrix Bandwidth

Theoreticians usually think in terms of graphs, but computational scientists often formulate bandwidth as a matrix-theoretic concept:

Definition (Matrix bandwidth)

Let A be an $n \times n$ matrix. The ***bandwidth*** $\beta(A, P)$ *of A under a permutation matrix P is the maximum $k \in \{0, 1, \ldots, n-1\}$ such that $\left[PAP^{\mathsf{T}}\right]_{i,j} \neq 0 \implies |i-j| \leq k$. The* ***minimum bandwidth*** $\beta(A)$ *of A is $\min \beta(A, P)$ taken over all permutation matrices P.*

- Often, A is **structurally symmetric**: $A_{i,j} = 0 \iff A_{j,i} = 0$
- If A is the adjacency matrix of a graph G (ignoring weights and diagonal entries), then $\beta(A) = \beta(G)$

Bandwidth Reduction Example

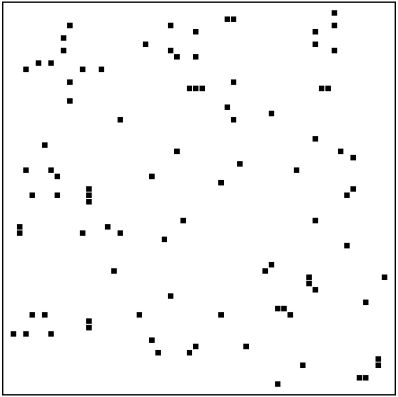

Figure 1: A 60×60 sparse matrix with original bandwidth 51.

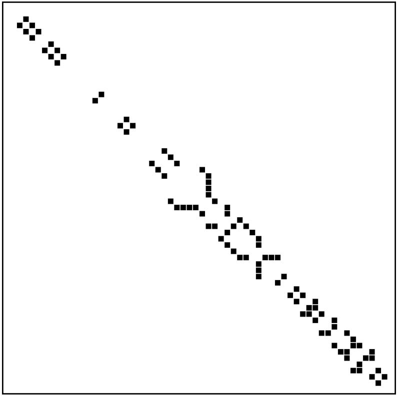

Figure 2: Bandwidth reduced to 5 by the **Gibbs–Poole–Stockmeyer** heuristic algorithm.

Practical Applications

Applications in engineering, scientific computing, and more:

- approximating **partial differential equations**...
- optimizing **circuit layout**...
- training **recurrent neural networks**...
- even investigating operators in **quantum information theory**!

Figure 3: My cat, **Ash**, playing with circuit-related things... <3

Practical Applications

Applications in engineering, scientific computing, and more:

- approximating **partial differential equations**...
- optimizing **circuit layout**...
- training **recurrent neural networks**...
- even investigating operators in **quantum information theory**!

Figure 3: My cat, **Ash**, playing with circuit-related things... <3

Practical Applications

Applications in engineering, scientific computing, and more:

- approximating **partial differential equations**...
- optimizing **circuit layout**...
- training **recurrent neural networks**...
- even investigating operators in **quantum information theory**!

Figure 3: My cat, **Ash**, playing with circuit-related things... <3

Practical Applications

Applications in engineering, scientific computing, and more:

- approximating **partial differential equations**...
- optimizing **circuit layout**...
- training **recurrent neural networks**...
- even investigating operators in **quantum information theory**!

Figure 3: My cat, **Ash**, playing with circuit-related things... <3

Practical Applications

Applications in engineering, scientific computing, and more:

- approximating **partial differential equations**...
- optimizing **circuit layout**...
- training **recurrent neural networks**...
- even investigating operators in **quantum information theory**!

Figure 3: My cat, **Ash**, playing with circuit-related things... <3

Types of Reduction Problems

Let $G = (V, E)$ be a graph with $|V| = n$ nodes, $|E| = m$ edges.

- **Bandwidth recognition:** For a fixed k, determine whether there exists a layout π of G with $\beta(G, \pi) \leq k$ — $O(n^k)$
- **Bandwidth minimization:** Find a layout π of G that minimizes (or gets close to minimizing) $\beta(G, \pi)$
    - **Exact algorithms:** Find a layout π of G that truly minimizes $\beta(G, \pi)$ — NP-complete
    - **(Meta)heuristic algorithms:** Find a layout π of G that approximately minimizes $\beta(G, \pi)$ — typically $O(mn)$ or $O(n^3)$

Types of Reduction Problems

Let $G = (V, E)$ be a graph with $|V| = n$ nodes, $|E| = m$ edges.

- **Bandwidth recognition:** For a fixed k, determine whether there exists a layout π of G with $\beta(G, \pi) \leq k$ — $O(n^k)$
- **Bandwidth minimization:** Find a layout π of G that minimizes (or gets close to minimizing) $\beta(G, \pi)$
  - **Exact algorithms:** Find a layout π of G that truly minimizes $\beta(G, \pi)$ — NP-complete
  - **(Meta)heuristic algorithms:** Find a layout π of G that approximately minimizes $\beta(G, \pi)$ — typically $O(mn)$ or $O(n^3)$

Types of Reduction Problems

Let $G = (V, E)$ be a graph with $|V| = n$ nodes, $|E| = m$ edges.

- **Bandwidth recognition:** For a fixed k, determine whether there exists a layout π of G with $\beta(G, \pi) \leq k$ — $O(n^k)$
- **Bandwidth minimization:** Find a layout π of G that minimizes (or gets close to minimizing) $\beta(G, \pi)$
  - **Exact algorithms:** Find a layout π of G that truly minimizes $\beta(G, \pi)$ — NP-complete
  - **(Meta)heuristic algorithms:** Find a layout π of G that approximately minimizes $\beta(G, \pi)$ — typically $O(mn)$ or $O(n^3)$

Open-Source Interface

- In open-source, only **reverse Cuthill-McKee** (older, less efficient heuristic algorithm from 1971) is widely available
- In **industry**: Want to apply performant modern alternatives
- In **academia**: Want to benchmark new algorithm ideas; often also need recognition/exact minimization (e.g., to bound a density matrix's factor width in quantum information theory)
- I have created **MatrixBandwidth.jl***, a unified Julia interface for recognition, exact minimization, and (meta)heuristic minimization algorithms for bandwidth reduction
- Now using it to investigate **partial layout priority heuristics** for bandwidth recognition algorithms...

*Available on the official Julia package registry and at `https://www.github.com/Luis-Varona/MatrixBandwidth.jl`

Open-Source Interface

- In open-source, only **reverse Cuthill-McKee** (older, less efficient heuristic algorithm from 1971) is widely available
- In **industry**: Want to apply performant modern alternatives
- In **academia**: Want to benchmark new algorithm ideas; often also need recognition/exact minimization (e.g., to bound a density matrix's factor width in quantum information theory)
- I have created **MatrixBandwidth.jl**[*], a unified Julia interface for recognition, exact minimization, and (meta)heuristic minimization algorithms for bandwidth reduction
- Now using it to investigate **partial layout priority heuristics** for bandwidth recognition algorithms...

[*] Available on the official Julia package registry and at `https://www.github.com/Luis-Varona/MatrixBandwidth.jl`

Open-Source Interface

- In open-source, only **reverse Cuthill-McKee** (older, less efficient heuristic algorithm from 1971) is widely available
- In **industry**: Want to apply performant modern alternatives
- In **academia**: Want to benchmark new algorithm ideas; often also need recognition/exact minimization (e.g., to bound a density matrix's factor width in quantum information theory)
- I have created **MatrixBandwidth.jl***, a unified Julia interface for recognition, exact minimization, and (meta)heuristic minimization algorithms for bandwidth reduction
- Now using it to investigate **partial layout priority heuristics** for bandwidth recognition algorithms...

*Available on the official Julia package registry and at `https://www.github.com/Luis-Varona/MatrixBandwidth.jl`

Open-Source Interface

- In open-source, only **reverse Cuthill-McKee** (older, less efficient heuristic algorithm from 1971) is widely available
- In **industry**: Want to apply performant modern alternatives
- In **academia**: Want to benchmark new algorithm ideas; often also need recognition/exact minimization (e.g., to bound a density matrix's factor width in quantum information theory)
- I have created **MatrixBandwidth.jl***, a unified Julia interface for recognition, exact minimization, and (meta)heuristic minimization algorithms for bandwidth reduction
- Now using it to investigate **partial layout priority heuristics** for bandwidth recognition algorithms...

*Available on the official Julia package registry and at `https://www.github.com/Luis-Varona/MatrixBandwidth.jl`

Base Recognition Algorithm

Definition (Partial layout)

Let $G = (V, E)$ be a graph. A ***partial layout*** *of G is a bijection from $U \hookrightarrow\!\!\!\!\!\twoheadrightarrow \{0, 1, \ldots, m - 1\}$ for some $U \subseteq V$ with $|U| = m$.*

Base $O(n^k)$ Saxe–Gurari–Sudborough algorithm to find a π with $\beta(G, \pi) \leq k$ [Journal of Algorithms **5** (1984), no. 4, 531–46]:

1. If G violates an $O(n^3)$ "density" lower bound, return `null`.
2. Initialize an empty queue Q for partial layouts of G and insert the empty partial layout $\varphi : \emptyset \to \emptyset$.
3. While Q is not empty: Extract a partial layout φ from Q.* If φ is a full layout, return φ. If φ does not violate certain constraints, extend it by one node to φ' and insert φ' into Q.
4. If Q is emptied, return `null`.

*Step 3 is simplified here—state data associated with each partial layout is also stored in Q, used to validate bandwidth constraints. (See next slide.)

Base Recognition Algorithm

Definition (Partial layout)

Let $G = (V, E)$ be a graph. A ***partial layout*** *of G is a bijection from $U \hookrightarrow\!\!\!\!\!\!\twoheadrightarrow \{0, 1, \ldots, m-1\}$ for some $U \subseteq V$ with $|U| = m$.*

Base $O(n^k)$ Saxe–Gurari–Sudborough algorithm to find a π with $\beta(G, \pi) \leq k$ [Journal of Algorithms **5** (1984), no. 4, 531–46]:

1. If G violates an $O(n^3)$ "density" lower bound, return `null`.
2. Initialize an empty queue Q for partial layouts of G and insert the empty partial layout $\varphi : \emptyset \to \emptyset$.
3. While Q is not empty: Extract a partial layout φ from Q.* If φ is a full layout, return φ. If φ does not violate certain constraints, extend it by one node to φ' and insert φ' into Q.
4. If Q is emptied, return `null`.

*Step 3 is simplified here—state data associated with each partial layout is also stored in Q, used to validate bandwidth constraints. (See next slide.)

Base Recognition Algorithm

Definition (Partial layout)

Let $G = (V, E)$ be a graph. A ***partial layout*** *of G is a bijection from $U \hookrightarrow\!\!\!\!\rightarrow \{0, 1, \ldots, m-1\}$ for some $U \subseteq V$ with $|U| = m$.*

Base $O(n^k)$ Saxe–Gurari–Sudborough algorithm to find a π with $\beta(G, \pi) \leq k$ [Journal of Algorithms **5** (1984), no. 4, 531–46]:

1. If G violates an $O(n^3)$ "density" lower bound, return `null`.
2. Initialize an empty queue Q for partial layouts of G and insert the empty partial layout $\varphi : \emptyset \to \emptyset$.
3. While Q is not empty: Extract a partial layout φ from Q.* If φ is a full layout, return φ. If φ does not violate certain constraints, extend it by one node to φ' and insert φ' into Q.
4. If Q is emptied, return `null`.

*Step 3 is simplified here—state data associated with each partial layout is also stored in Q, used to validate bandwidth constraints. (See next slide.)

Base Recognition Algorithm

Definition (Partial layout)

Let $G = (V, E)$ be a graph. A ***partial layout*** *of G is a bijection from $U \hookrightarrow\!\!\!\!\rightarrow \{0, 1, \ldots, m-1\}$ for some $U \subseteq V$ with $|U| = m$.*

Base $O(n^k)$ Saxe–Gurari–Sudborough algorithm to find a π with $\beta(G, \pi) \leq k$ [Journal of Algorithms **5** (1984), no. 4, 531–46]:

1. If G violates an $O(n^3)$ "density" lower bound, return `null`.
2. Initialize an empty queue Q for partial layouts of G and insert the empty partial layout $\varphi : \emptyset \to \emptyset$.
3. While Q is not empty: Extract a partial layout φ from Q.* If φ is a full layout, return φ. If φ does not violate certain constraints, extend it by one node to φ' and insert φ' into Q.
4. If Q is emptied, return `null`.

*Step 3 is simplified here—state data associated with each partial layout is also stored in Q, used to validate bandwidth constraints. (See next slide.)

Base Recognition Algorithm

Definition (Partial layout)

Let $G = (V, E)$ be a graph. A ***partial layout*** *of G is a bijection from $U \hookrightarrow\!\!\!\!\rightarrow \{0, 1, \ldots, m-1\}$ for some $U \subseteq V$ with $|U| = m$.*

Base $O(n^k)$ Saxe–Gurari–Sudborough algorithm to find a π with $\beta(G, \pi) \leq k$ [Journal of Algorithms **5** (1984), no. 4, 531–46]:

1. If G violates an $O(n^3)$ "density" lower bound, return `null`.
2. Initialize an empty queue Q for partial layouts of G and insert the empty partial layout $\varphi : \emptyset \to \emptyset$.
3. While Q is not empty: Extract a partial layout φ from Q.* If φ is a full layout, return φ. If φ does not violate certain constraints, extend it by one node to φ' and insert φ' into Q.
4. If Q is emptied, return `null`.

*Step 3 is simplified here—state data associated with each partial layout is also stored in Q, used to validate bandwidth constraints. (See next slide.)

Base Recognition Algorithm

Definition (Partial layout)

Let $G = (V, E)$ be a graph. A ***partial layout*** *of G is a bijection from $U \hookrightarrow\!\!\!\!\rightarrow \{0, 1, \dots, m-1\}$ for some $U \subseteq V$ with $|U| = m$.*

Base $O(n^k)$ Saxe–Gurari–Sudborough algorithm to find a π with $\beta(G, \pi) \le k$ [Journal of Algorithms **5** (1984), no. 4, 531–46]:

1. If G violates an $O(n^3)$ "density" lower bound, return `null`.
2. Initialize an empty queue Q for partial layouts of G and insert the empty partial layout $\varphi : \emptyset \to \emptyset$.
3. While Q is not empty: Extract a partial layout φ from Q.* If φ is a full layout, return φ. If φ does not violate certain constraints, extend it by one node to φ' and insert φ' into Q.
4. If Q is emptied, return `null`.

*Step 3 is simplified here—state data associated with each partial layout is also stored in Q, used to validate bandwidth constraints. (See next slide.)

Adding a Priority Queue (Pt. 1)

We can replace the queue of partial layouts with a **min-heap priority queue** to find a bandwidth-k layout faster. Associate with each partial layout $\varphi : U \hookrightarrow\!\!\!\!\rightarrow \{0, 1, \ldots, m-1\}$ a 4-tuple **state** of the form $(\texttt{placed}, \texttt{unplaced}, \texttt{region}, \texttt{latest})$:

1. `placed` is an array representation of the mapping (i.e., $\texttt{placed}[i] = v$ implies $\varphi(v) = i$)
2. `unplaced` is a hash set consisting of the nodes in $V \setminus U$ with which we may extend φ to a full layout
3. `latest` is a hash map from each node $v \in V \setminus U$ to the latest position $i \in \{m, m+1, \ldots, n-1\}$ it can occupy without violating the bandwidth-k constraint
4. `region` is a another data structure tracking edges connecting U to $V \setminus U$, used to validate bandwidth constraints in Step 3

Adding a Priority Queue (Pt. 1)

We can replace the queue of partial layouts with a **min-heap priority queue** to find a bandwidth-k layout faster. Associate with each partial layout $\varphi : U \hookrightarrow\!\!\!\!\rightarrow \{0, 1, \ldots, m-1\}$ a 4-tuple **state** of the form $(\texttt{placed}, \texttt{unplaced}, \texttt{region}, \texttt{latest})$:

1. `placed` is an array representation of the mapping (i.e., $\texttt{placed}[i] = v$ implies $\varphi(v) = i$)
2. `unplaced` is a hash set consisting of the nodes in $V \setminus U$ with which we may extend φ to a full layout
3. `latest` is a hash map from each node $v \in V \setminus U$ to the latest position $i \in \{m, m+1, \ldots, n-1\}$ it can occupy without violating the bandwidth-k constraint
4. `region` is a another data structure tracking edges connecting U to $V \setminus U$, used to validate bandwidth constraints in Step 3

Adding a Priority Queue (Pt. 1)

We can replace the queue of partial layouts with a **min-heap priority queue** to find a bandwidth-k layout faster. Associate with each partial layout $\varphi : U \hookrightarrow\!\!\!\!\!\rightarrow \{0, 1, \ldots, m-1\}$ a 4-tuple **state** of the form $(\texttt{placed}, \texttt{unplaced}, \texttt{region}, \texttt{latest})$:

1. `placed` is an array representation of the mapping (i.e., $\texttt{placed}[i] = v$ implies $\varphi(v) = i$)
2. `unplaced` is a hash set consisting of the nodes in $V \setminus U$ with which we may extend φ to a full layout
3. `latest` is a hash map from each node $v \in V \setminus U$ to the latest position $i \in \{m, m+1, \ldots, n-1\}$ it can occupy without violating the bandwidth-k constraint
4. `region` is a another data structure tracking edges connecting U to $V \setminus U$, used to validate bandwidth constraints in Step 3

Adding a Priority Queue (Pt. 1)

We can replace the queue of partial layouts with a **min-heap priority queue** to find a bandwidth-k layout faster. Associate with each partial layout $\varphi : U \hookrightarrow\!\!\!\!\rightarrow \{0, 1, \ldots, m-1\}$ a 4-tuple **state** of the form $(\texttt{placed}, \texttt{unplaced}, \texttt{region}, \texttt{latest})$:

1. `placed` is an array representation of the mapping (i.e., $\texttt{placed}[i] = v$ implies $\varphi(v) = i$)
2. `unplaced` is a hash set consisting of the nodes in $V \setminus U$ with which we may extend φ to a full layout
3. `latest` is a hash map from each node $v \in V \setminus U$ to the latest position $i \in \{m, m+1, \ldots, n-1\}$ it can occupy without violating the bandwidth-k constraint
4. `region` is a another data structure tracking edges connecting U to $V \setminus U$, used to validate bandwidth constraints in Step 3

Adding a Priority Queue (Pt. 1)

We can replace the queue of partial layouts with a **min-heap priority queue** to find a bandwidth-k layout faster. Associate with each partial layout $\varphi : U \hookrightarrow\!\!\!\!\rightarrow \{0, 1, \ldots, m-1\}$ a 4-tuple **state** of the form $(\texttt{placed}, \texttt{unplaced}, \texttt{region}, \texttt{latest})$:

1. `placed` is an array representation of the mapping (i.e., $\texttt{placed}[i] = v$ implies $\varphi(v) = i$)
2. `unplaced` is a hash set consisting of the nodes in $V \setminus U$ with which we may extend φ to a full layout
3. `latest` is a hash map from each node $v \in V \setminus U$ to the latest position $i \in \{m, m+1, \ldots, n-1\}$ it can occupy without violating the bandwidth-k constraint
4. `region` is a another data structure tracking edges connecting U to $V \setminus U$, used to validate bandwidth constraints in Step 3

Adding a Priority Queue (Pt. 2)

We use MatrixBandwidth.jl to compare three **priority functions** for a state s, adapted from **constraint satisfaction problems**:

- **Least-active-nodes** heuristic:

$$\texttt{priority}_{\texttt{lan}}(s) = \left|\left\{v \in s.\texttt{unplaced} : \exists u \in s.\texttt{placed} \text{ with } \{u, v\} \in E\right\}\right|$$

- **Minimum-remaining-values** heuristic:

$$\texttt{priority}_{\texttt{mrv}}(s) = \min\left\{s.\texttt{latest}[v] - |s.\texttt{placed}| : v \in s.\texttt{unplaced}\right\}$$

- **Total-remaining-values** heuristic:

$$\texttt{priority}_{\texttt{trv}}(s) = \sum_{v \in s.\texttt{unplaced}} \left(s.\texttt{latest}[v] - |s.\texttt{placed}|\right)$$

Adding a Priority Queue (Pt. 2)

We use MatrixBandwidth.jl to compare three **priority functions** for a state s, adapted from **constraint satisfaction problems**:

- **Least-active-nodes** heuristic:

$$\texttt{priority}_{\texttt{lan}}(s) = \left|\{v \in s.\texttt{unplaced} : \exists u \in s.\texttt{placed} \text{ with } \{u, v\} \in E\}\right|$$

- **Minimum-remaining-values** heuristic:

$$\texttt{priority}_{\texttt{mrv}}(s) = \min\{s.\texttt{latest}[v] - |s.\texttt{placed}| : v \in s.\texttt{unplaced}\}$$

- **Total-remaining-values** heuristic:

$$\texttt{priority}_{\texttt{trv}}(s) = \sum_{v \in s.\texttt{unplaced}} (s.\texttt{latest}[v] - |s.\texttt{placed}|)$$

Adding a Priority Queue (Pt. 2)

We use MatrixBandwidth.jl to compare three **priority functions** for a state s, adapted from **constraint satisfaction problems**:

- **Least-active-nodes** heuristic:

$$\texttt{priority}_{\texttt{lan}}(s) = \left|\{v \in s.\texttt{unplaced} : \exists u \in s.\texttt{placed} \text{ with } \{u, v\} \in E\}\right|$$

- **Minimum-remaining-values** heuristic:

$$\texttt{priority}_{\texttt{mrv}}(s) = \min\{s.\texttt{latest}[v] - |s.\texttt{placed}| : v \in s.\texttt{unplaced}\}$$

- **Total-remaining-values** heuristic:

$$\texttt{priority}_{\texttt{trv}}(s) = \sum_{v \in s.\texttt{unplaced}} \left(s.\texttt{latest}[v] - |s.\texttt{placed}|\right)$$

Adding a Priority Queue (Pt. 2)

We use MatrixBandwidth.jl to compare three **priority functions** for a state s, adapted from **constraint satisfaction problems**:

- **Least-active-nodes** heuristic:

$$\texttt{priority}_{\texttt{lan}}(s) = \left|\left\{v \in s.\texttt{unplaced} : \exists u \in s.\texttt{placed} \text{ with } \{u, v\} \in E\right\}\right|$$

- **Minimum-remaining-values** heuristic:

$$\texttt{priority}_{\texttt{mrv}}(s) = \min\left\{s.\texttt{latest}[v] - |s.\texttt{placed}| : v \in s.\texttt{unplaced}\right\}$$

- **Total-remaining-values** heuristic:

$$\texttt{priority}_{\texttt{trv}}(s) = \sum_{v \in s.\texttt{unplaced}} \left(s.\texttt{latest}[v] - |s.\texttt{placed}|\right)$$

Next Steps

Figure 4: Rebekka's dogs, **Jonsi** (right) and **Timmy** (left), being cute <3

Thank you!

Figure 5: Art by **Rebekka Jonasson** <3