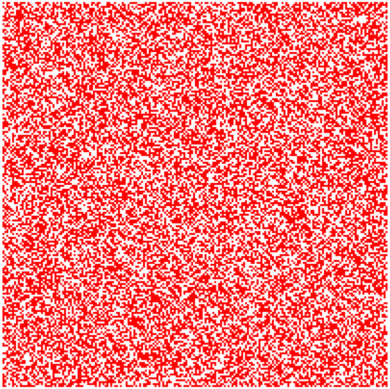
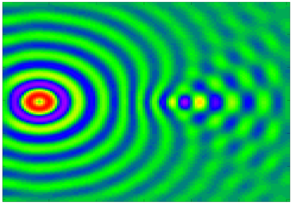


Sparse-direct factorizations and DPPs

Jack Poulson (Hodge Star Scientific Computing)
Aussois, France, June 20, 2019

Sparse-direct DPP factorizations

We have so-far discussed analogues of **dense** factorizations, and **sparse-direct** analogues are a natural extension.

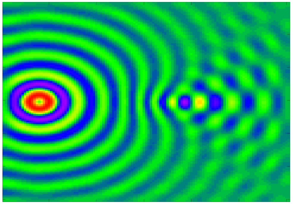


Catamari implements templated, real and complex, Cholesky / LDL^H / LDL^T – switching between DAG-scheduled, **right-looking supernodal** and **up-looking simplicial** based upon arithmetic intensity [Chen/Davis/Hager/Rajamanickam-2008].

A variant of the sparse-direct LDL^H is provided for sparse DPPs. (Unpivoted) sparse-direct LU and LDL^T DPP sampling is a straight-forward extension.

Sparse-direct DPP factorizations

We have so-far discussed analogues of **dense** factorizations, and **sparse-direct** analogues are a natural extension.

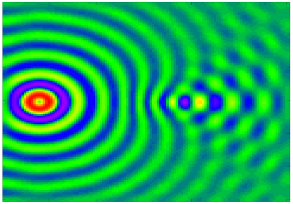


Catamari implements templated, real and complex, Cholesky / LDL^H / LDL^T – switching between DAG-scheduled, **right-looking supernodal** and **up-looking simplicial** based upon arithmetic intensity [Chen/Davis/Hager/Rajamanickam-2008].

A variant of the sparse-direct LDL^H is provided for sparse DPPs. (Unpivoted) sparse-direct LU and LDL^T DPP sampling is a straight-forward extension.

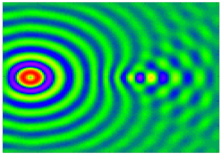
Sparse-direct DPP factorizations

We have so-far discussed analogues of **dense** factorizations, and **sparse-direct** analogues are a natural extension.



Catamari implements templated, real and complex, Cholesky / LDL^H / LDL^T – switching between DAG-scheduled, **right-looking supernodal** and **up-looking simplicial** based upon arithmetic intensity [Chen/Davis/Hager/Rajamanickam-2008].

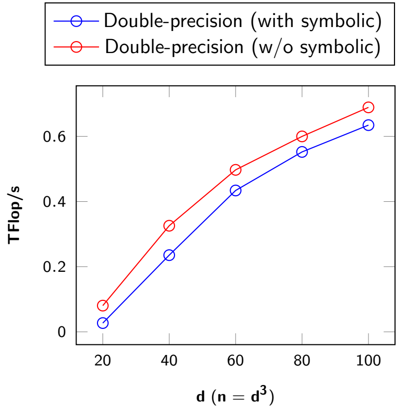
A variant of the sparse-direct LDL^H is provided for sparse DPPs. (Unpivoted) sparse-direct LU and LDL^T DPP sampling is a straight-forward extension.

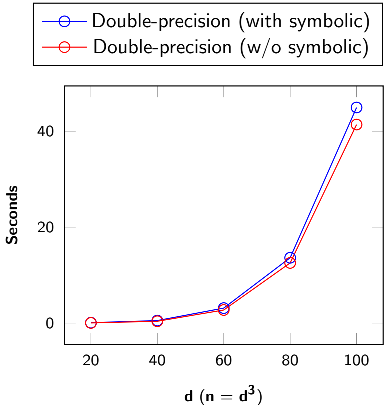


Complex sparse LDL^T on i9-7960x (16-core)

3D Helmholtz w/ PML and trilinear, hexahedral elements

```
$ OMP_NUM_THREADS=16 ./helmholtz_3d_pml
```





Minimum Degree reorderings

Consider the Schur complement elimination of the first index of a matrix with sparsity pattern:

$$\left(\begin{array}{c|cccccc} x & x & x & & x & & x \\ \hline x & x & & & & & \\ x & & x & & & & \\ & & & x & & & \\ x & & & & x & & \\ & & & & & x & \\ x & & & & & & x \end{array}\right) \mapsto \left(\begin{array}{c|cccccc} x & & & & & & \\ \hline x & x & \color{red}{x} & & \color{red}{x} & & \color{red}{x} \\ x & \color{red}{x} & x & & \color{red}{x} & & \color{red}{x} \\ & & & x & & & \\ x & \color{red}{x} & \color{red}{x} & & x & & \color{red}{x} \\ & & & & & x & \\ x & \color{red}{x} & \color{red}{x} & & \color{red}{x} & & x \end{array}\right),$$

where the **fill-in** has been marked in red.

If we refer to the number of edges connected to a node as its **degree** – in this case, the **degree** of the first index is 4 – we notice the square of the degree is an upper-bound for the amount of fill from its elimination.

Minimum Degree reorderings

Consider the Schur complement elimination of the first index of a matrix with sparsity pattern:

$$\left(\begin{array}{c|cccccc} x & x & x & & x & & x \\ \hline x & x & & & & & \\ x & & x & & & & \\ & & & x & & & \\ x & & & & x & & \\ & & & & & x & \\ x & & & & & & x \end{array}\right) \mapsto \left(\begin{array}{c|cccccc} x & & & & & & \\ \hline x & x & \textcolor{red}{x} & & \textcolor{red}{x} & & \textcolor{red}{x} \\ x & \textcolor{red}{x} & x & & \textcolor{red}{x} & & \textcolor{red}{x} \\ & & & x & & & \\ x & \textcolor{red}{x} & \textcolor{red}{x} & & x & & \textcolor{red}{x} \\ & & & & & x & \\ x & \textcolor{red}{x} & \textcolor{red}{x} & & \textcolor{red}{x} & & x \end{array}\right),$$

where the **fill-in** has been marked in red.

If we refer to the number of edges connected to a node as its **degree** – in this case, the **degree** of the first index is 4 – we notice the square of the degree is an upper-bound for the amount of fill from its elimination.

Minimum Degree reorderings

Whereas, if we swapped the first and last indices as:

$$\begin{pmatrix} x & & & & & & x \\ & x & & & & & x \\ & & x & & & & x \\ & & & x & & & \\ & & & & x & & x \\ & & & & & x & \\ x & x & x & & x & & x \end{pmatrix},$$

a full factorization could complete with no fill-in.

Such an ordering can be seen to have sequentially chosen an index of minimal degree after each step of elimination. This is the idea behind **minimum degree reordering**, which was introduced by [Tinney/Walker-1967] as a symmetric analogue of [Markowitz-1957].

Minimum Degree reorderings

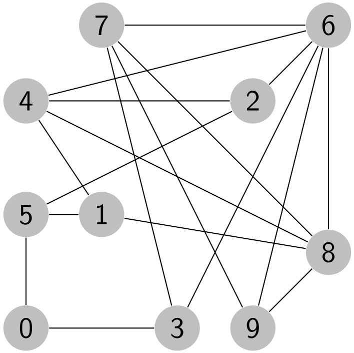
Whereas, if we swapped the first and last indices as:

$$\begin{pmatrix} x & & & & & & x \\ & x & & & & & x \\ & & x & & & & x \\ & & & x & & & \\ & & & & x & & x \\ & & & & & x & \\ x & x & x & & x & & x \end{pmatrix},$$

a full factorization could complete with no fill-in.

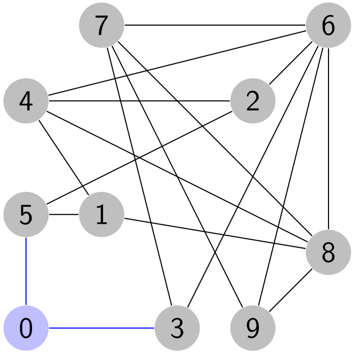
Such an ordering can be seen to have sequentially chosen an index of minimal degree after each step of elimination. This is the idea behind **minimum degree reordering**, which was introduced by [Tinney/Walker-1967] as a symmetric analogue of [Markowitz-1957].

Minimum Degree reorderings



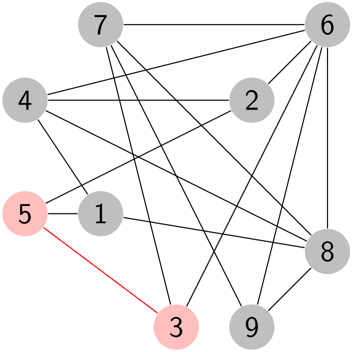
Eliminated nodes:
[]

Minimum Degree: select 0



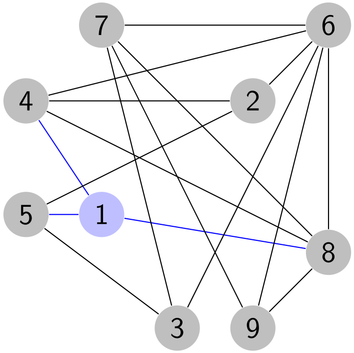
Eliminated nodes:
[]

Minimum Degree: eliminate 0



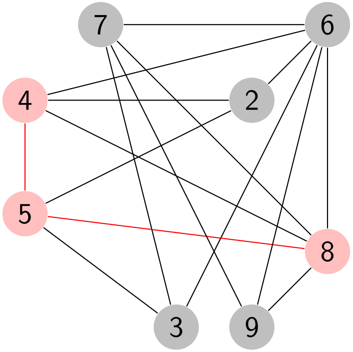
Eliminated nodes:
[0]

Minimum Degree: select 1



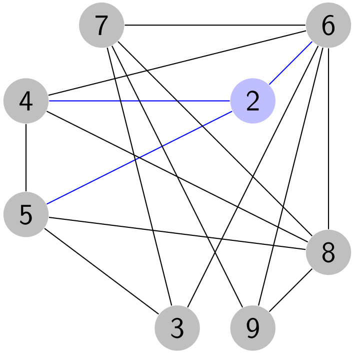
Eliminated nodes:
[0]

Minimum Degree: eliminate 1



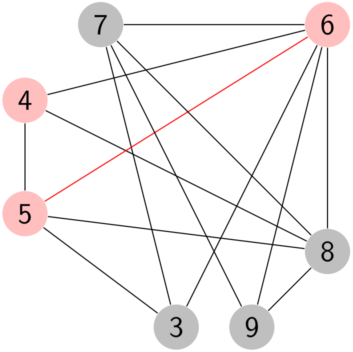
Eliminated nodes:
$[0, 1]$

Minimum Degree: select 2



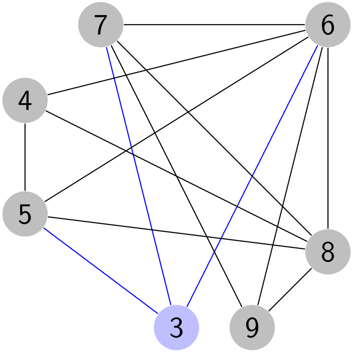
Eliminated nodes:
$[0, 1]$

Minimum Degree: eliminate 2



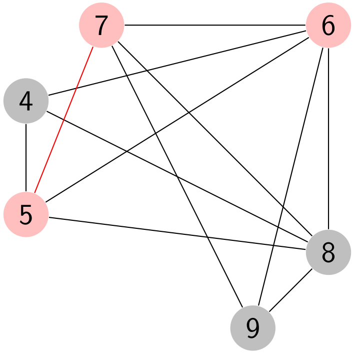
Eliminated nodes:
$[0, 1, 2]$

Minimum Degree: select 3



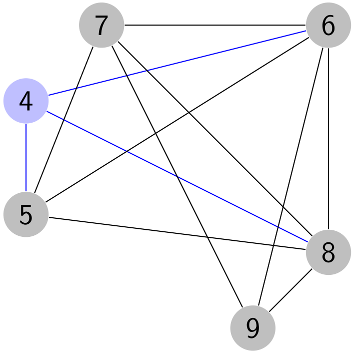
Eliminated nodes:
$[0, 1, 2]$

Minimum Degree: eliminate 3



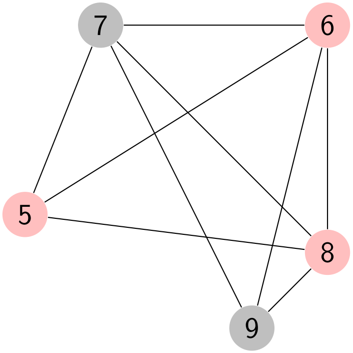
Eliminated nodes:
$[0, 1, 2, 3]$

Minimum Degree: select 4



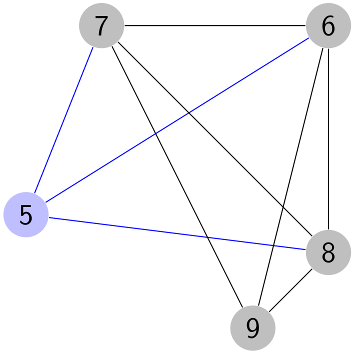
Eliminated nodes:
$[0, 1, 2, 3]$

Minimum Degree: eliminate 4



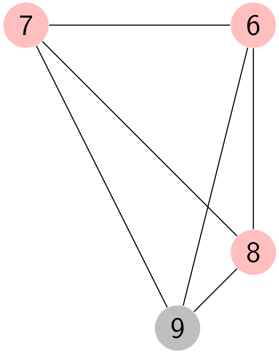
Eliminated nodes:
$[0, 1, 2, 3, 4]$

Minimum Degree: select 5



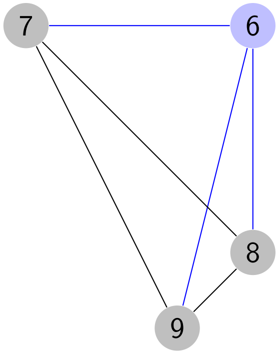
Eliminated nodes:
$[0, 1, 2, 3, 4]$

Minimum Degree: eliminate 5



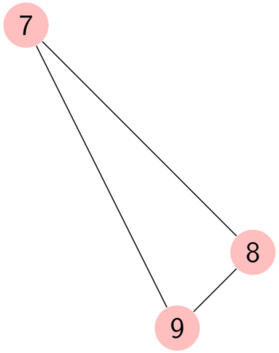
Eliminated nodes:
$[0, 1, 2, 3, 4, 5]$

Minimum Degree: select 6



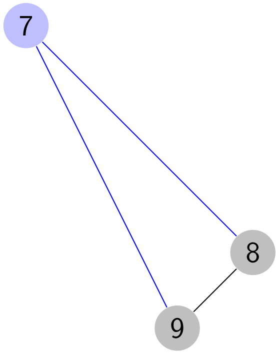
Eliminated nodes:
$[0, 1, 2, 3, 4, 5]$

Minimum Degree: eliminate 6



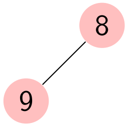
Eliminated nodes:
$[0, 1, 2, 3, 4, 5, 6]$

Minimum Degree: select 7



Eliminated nodes:
$[0, 1, 2, 3, 4, 5, 6]$

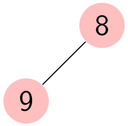
Minimum Degree: eliminate 7



Eliminated nodes:
$[0, 1, 2, 3, 4, 5, 6, 7]$

Then we select 8, then 9...

Minimum Degree: eliminate 7



Eliminated nodes:
$[0, 1, 2, 3, 4, 5, 6, 7]$

Then we select 8, then 9...

Approximate Minimum Degree reorderings

Such a naive Minimum Degree reordering is not competitive in practice: the cost of explicitly maintaining the **elimination graph** would rival that of a numeric factorization.

The de facto standard (e.g., in MATLAB and in most mathematical programming) is the **Approximate Minimum Degree** reordering algorithm [Amestoy/Davis/Duff-1996], which uses a fast but accurate approximation of the degree and continually probes for cliques within the elimination graph (actually, in the **quotient graph**) in order to reduce complexity.

Approximate Minimum Degree reorderings

Such a naive Minimum Degree reordering is not competitive in practice: the cost of explicitly maintaining the **elimination graph** would rival that of a numeric factorization.

The de facto standard (e.g., in MATLAB and in most mathematical programming) is the **Approximate Minimum Degree** reordering algorithm [Amestoy/Davis/Duff-1996], which uses a fast but accurate approximation of the degree and continually probes for cliques within the elimination graph (actually, in the **quotient graph**) in order to reduce complexity.

Quotient graphs

Let us reconsider the elimination

$$\left(\begin{array}{c|cccccc} x & x & x & & x & & x \\ \hline x & x & & & & & \\ x & & x & & & & \\ & & & x & & & \\ x & & & & x & & \\ & & & & & x & \\ x & & & & & & x \end{array}\right) \mapsto \left(\begin{array}{c|cccccc} x & & & & & & \\ \hline x & x & x & & x & & x \\ x & x & x & & x & & x \\ & & & x & & & \\ x & x & x & & x & & x \\ & & & & & x & \\ x & x & x & & x & & x \end{array}\right),$$

which we see requires much more storage for the resulting elimination graph than the original graph.

But each Schur complement introduces a **clique** of size k in the elimination graph, which we could represent with k indices instead of k^2.

[George/Liu-1981] formalized such an approach called the **quotient graph** (originally **elimination graph**) and showed that it never requires more storage than the original graph.

The key idea is to maintain a graph with two separate types of vertices: **variables**, which are of the standard type, and **elements**, which are the mechanism for efficiently representing cliques.

Quotient graphs

Let us reconsider the elimination

$$\left(\begin{array}{c|cccccc} x & x & x & & x & & x \\ \hline x & x & & & & & \\ x & & x & & & & \\ & & & x & & & \\ x & & & & x & & \\ & & & & & x & \\ x & & & & & & x \end{array}\right) \mapsto \left(\begin{array}{c|cccccc} x & & & & & & \\ \hline x & x & x & & x & & x \\ x & x & x & & x & & x \\ & & & x & & & \\ x & x & x & & x & & x \\ & & & & & x & \\ x & x & x & & x & & x \end{array}\right),$$

which we see requires much more storage for the resulting elimination graph than the original graph.

But each Schur complement introduces a **clique** of size k in the elimination graph, which we could represent with k indices instead of k^2.

[George/Liu-1981] formalized such an approach called the **quotient graph** (originally **elimination graph**) and showed that it never requires more storage than the original graph.

The key idea is to maintain a graph with two separate types of vertices: **variables**, which are of the standard type, and **elements**, which are the mechanism for efficiently representing cliques.

Quotient graphs

Let us reconsider the elimination

$$\left(\begin{array}{c|cccccc} x & x & x & & x & & x \\ \hline x & x & & & & & \\ x & & x & & & & \\ & & & x & & & \\ x & & & & x & & \\ & & & & & x & \\ x & & & & & & x \end{array}\right) \mapsto \left(\begin{array}{c|cccccc} x & & & & & & \\ \hline x & x & x & & x & & x \\ x & x & x & & x & & x \\ & & & x & & & \\ x & x & x & & x & & x \\ & & & & & x & \\ x & x & x & & x & & x \end{array}\right),$$

which we see requires much more storage for the resulting elimination graph than the original graph.

But each Schur complement introduces a **clique** of size k in the elimination graph, which we could represent with k indices instead of k^2.

[George/Liu-1981] formalized such an approach called the **quotient graph** (originally **elimination graph**) and showed that it never requires more storage than the original graph.

The key idea is to maintain a graph with two separate types of vertices: **variables**, which are of the standard type, and **elements**, which are the mechanism for efficiently representing cliques.

Quotient graphs

Let us reconsider the elimination

$$\left(\begin{array}{c|cccccc} x & x & x & & x & & x \\ \hline x & x & & & & & \\ x & & x & & & & \\ & & & x & & & \\ x & & & & x & & \\ & & & & & x & \\ x & & & & & & x \end{array}\right) \mapsto \left(\begin{array}{c|cccccc} x & & & & & & \\ \hline x & x & x & & x & & x \\ x & x & x & & x & & x \\ & & & x & & & \\ x & x & x & & x & & x \\ & & & & & x & \\ x & x & x & & x & & x \end{array}\right),$$

which we see requires much more storage for the resulting elimination graph than the original graph.

But each Schur complement introduces a **clique** of size k in the elimination graph, which we could represent with k indices instead of k^2.

[George/Liu-1981] formalized such an approach called the **quotient graph** (originally **elimination graph**) and showed that it never requires more storage than the original graph.

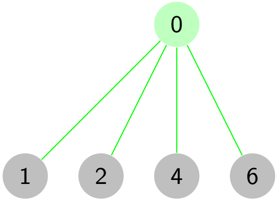
The key idea is to maintain a graph with two separate types of vertices: **variables**, which are of the standard type, and **elements**, which are the mechanism for efficiently representing cliques.

Quotient graphs

The result of the elimination

$$\left(\begin{array}{c|cccccc} x & x & x & & x & & x \\ \hline x & x & & & & & \\ x & & x & & & & \\ & & & x & & & \\ x & & & & x & & \\ & & & & & x & \\ x & & & & & & x \end{array}\right) \mapsto \left(\begin{array}{c|cccccc} x & & & & & & \\ \hline x & x & x & & x & & x \\ x & x & x & & x & & x \\ & & & x & & & \\ x & x & x & & x & & x \\ & & & & & x & \\ x & x & x & & x & & x \end{array}\right),$$

could be represented via the quotient graph



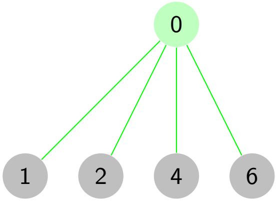
To determine the adjacency of a variable in the elimination graph from the quotient graph, each connection to an element is replaced with connections to all of the variables adjacent to the element.

Quotient graphs

The result of the elimination

$$\left(\begin{array}{c|cccccc} x & x & x & & x & & x \\ \hline x & x & & & & & \\ x & & x & & & & \\ & & & x & & & \\ x & & & & x & & \\ & & & & & x & \\ x & & & & & & x \end{array}\right) \mapsto \left(\begin{array}{c|cccccc} x & & & & & & \\ \hline x & x & x & & x & & x \\ x & x & x & & x & & x \\ & & & x & & & \\ x & x & x & & x & & x \\ & & & & & x & \\ x & x & x & & x & & x \end{array}\right),$$

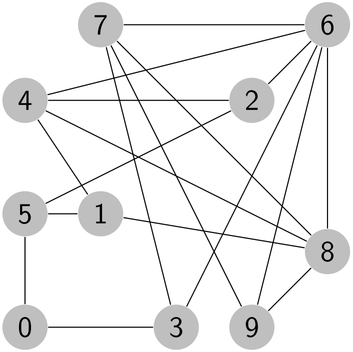
could be represented via the quotient graph



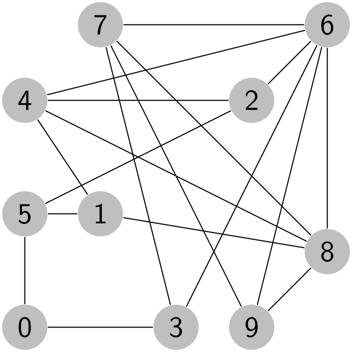
To determine the adjacency of a variable in the elimination graph from the quotient graph, each connection to an element is replaced with connections to all of the variables adjacent to the element.

Quotient graph elimination

Elimination graph:

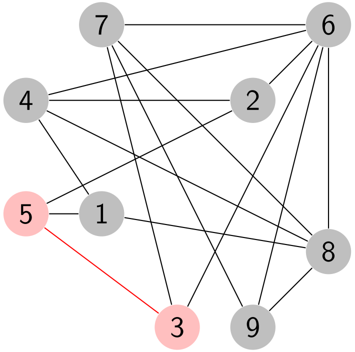


Quotient graph:

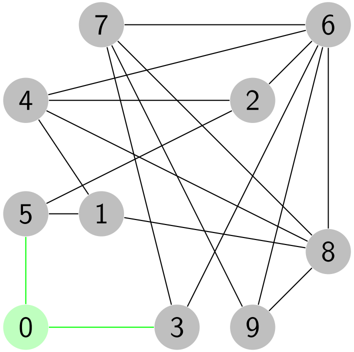


Quotient graph elimination: eliminate 0

Elimination graph:

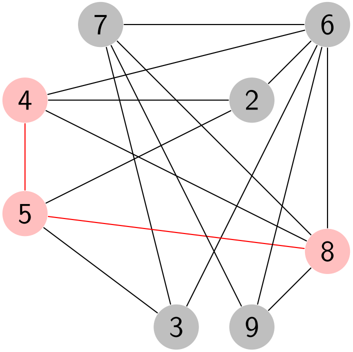


Quotient graph:

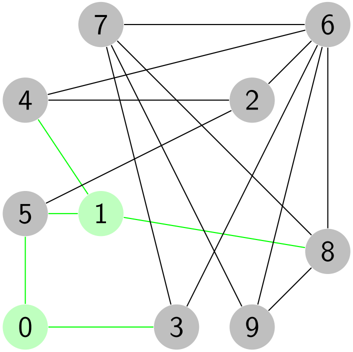


Quotient graph elimination: eliminate 1

Elimination graph:



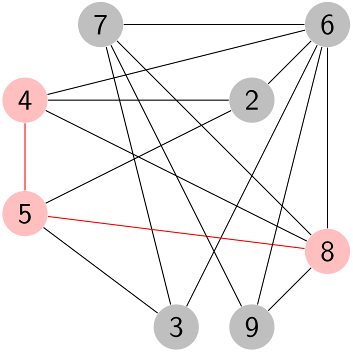
Quotient graph:



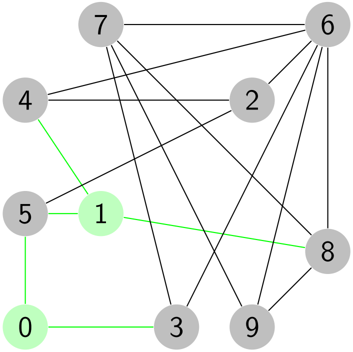
Notice that we were able to delete the $(4, 8)$ edge in the quotient graph.

Quotient graph elimination: eliminate 1

Elimination graph:



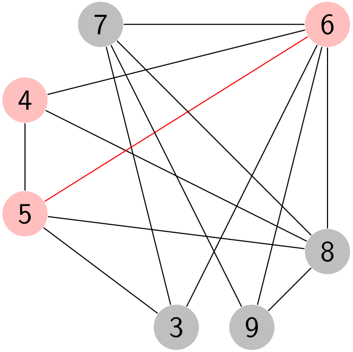
Quotient graph:



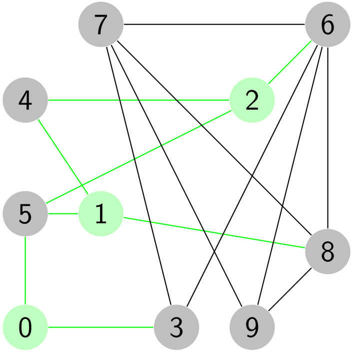
Notice that we were able to delete the $(4, 8)$ edge in the quotient graph.

Quotient graph elimination: eliminate 2

Elimination graph:



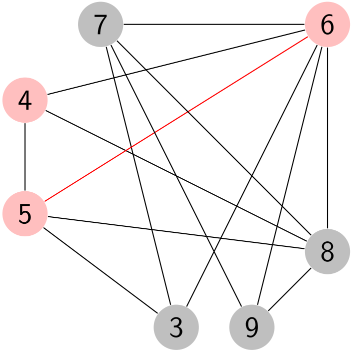
Quotient graph:



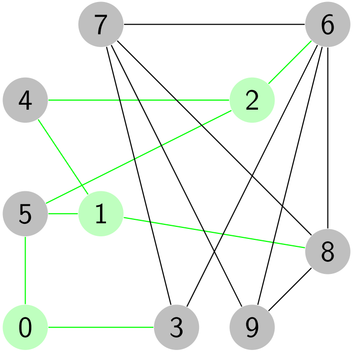
This time, we could delete the $(4, 6)$ edge from the quotient graph.

Quotient graph elimination: eliminate 2

Elimination graph:



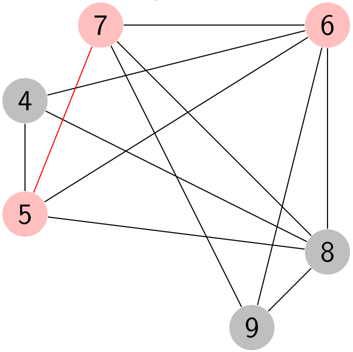
Quotient graph:



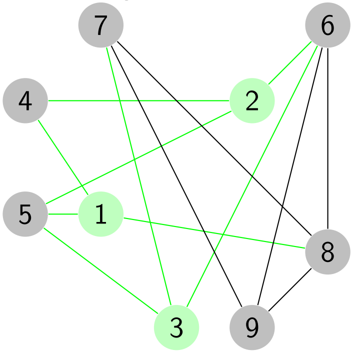
This time, we could delete the $(4, 6)$ edge from the quotient graph.

Quotient graph elimination: eliminate 3

Elimination graph:



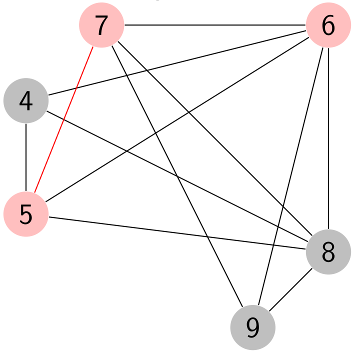
Quotient graph:



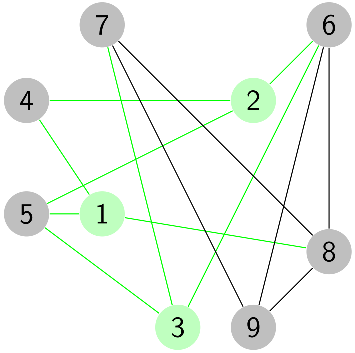
We were able to delete element 0 since its contribution to the elimination graph was contained in that of element 3. We also deleted redundant edge $(6, 7)$ from quotient graph.

Quotient graph elimination: eliminate 3

Elimination graph:



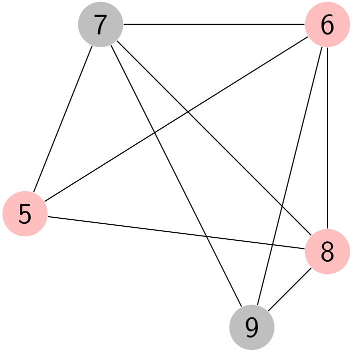
Quotient graph:



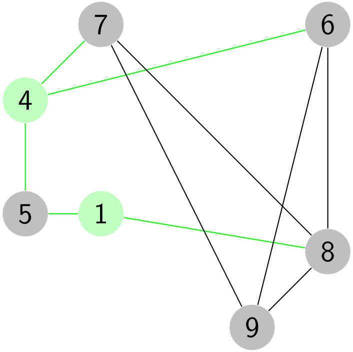
We were able to delete element 0 since its contribution to the elimination graph was contained in that of element 3. We also deleted redundant edge $(6, 7)$ from quotient graph.

Quotient graph elimination: eliminate 4

Elimination graph:



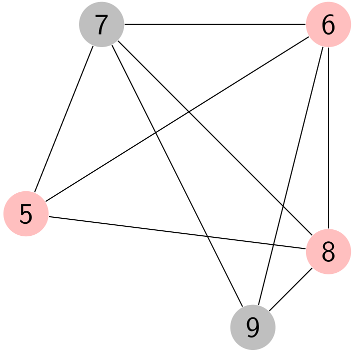
Quotient graph:



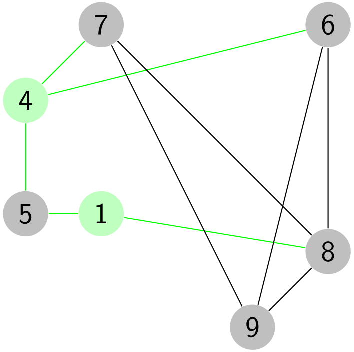
We were able to absorb elements 2 and 3 into element 4.

Quotient graph elimination: eliminate 4

Elimination graph:



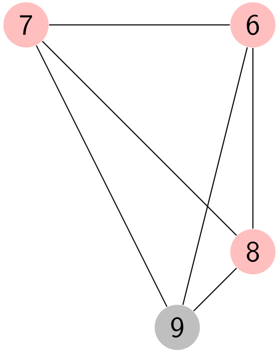
Quotient graph:



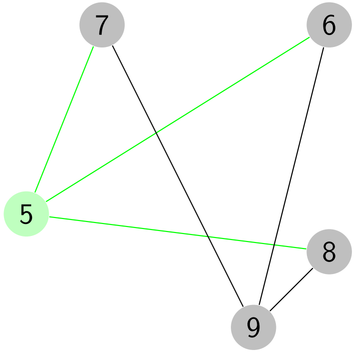
We were able to absorb elements 2 and 3 into element 4.

Quotient graph elimination: eliminate 5

Elimination graph:



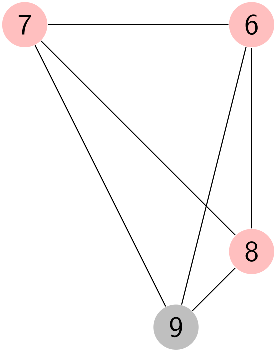
Quotient graph:



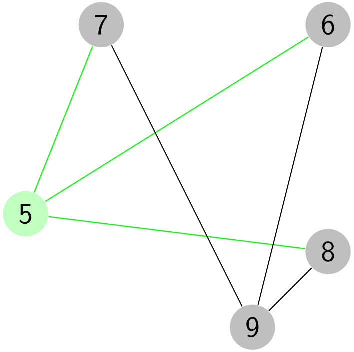
We were able to absorb elements 1 and 4 and delete edges $(6, 8)$ and $(7, 8)$.

Quotient graph elimination: eliminate 5

Elimination graph:



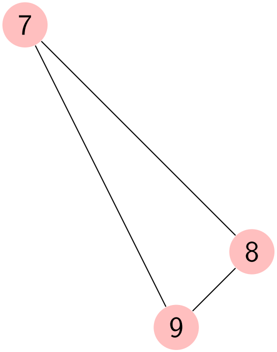
Quotient graph:



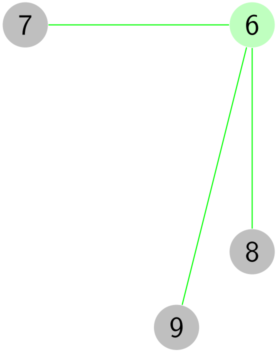
We were able to absorb elements 1 and 4 and delete edges $(6, 8)$ and $(7, 8)$.

Quotient graph elimination: eliminate 6

Elimination graph:



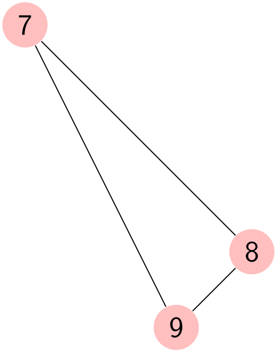
Quotient graph:



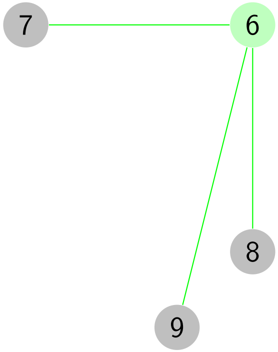
We were able to absorb element 5 and delete edges $(7, 9)$ and $(8, 9)$.

Quotient graph elimination: eliminate 6

Elimination graph:



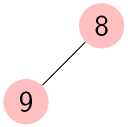
Quotient graph:



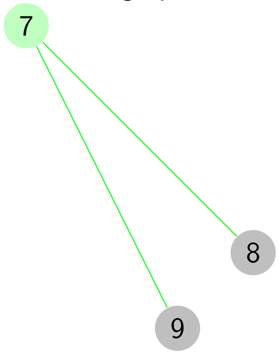
We were able to absorb element 5 and delete edges $(7, 9)$ and $(8, 9)$.

Quotient graph elimination: eliminate 7

Elimination graph:



Quotient graph:

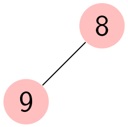


We were able to absorb element 6.

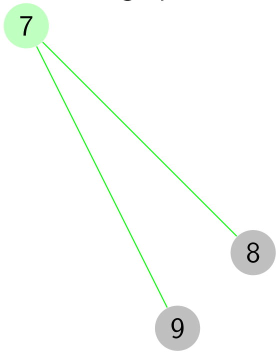
Etc.

Quotient graph elimination: eliminate 7

Elimination graph:



Quotient graph:

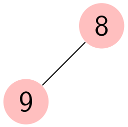


We were able to absorb element 6.

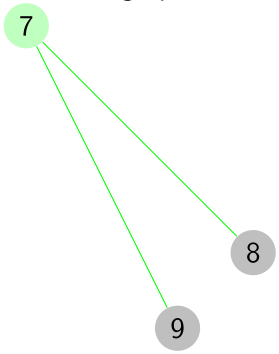
Etc.

Quotient graph elimination: eliminate 7

Elimination graph:



Quotient graph:



We were able to absorb element 6.

Etc.

Supervariable detection

An important performance optimization is detecting **indistinguishable variables**, or **supervariables**: cliques within the elimination graph where each member has the same external adjacencies.

In practice, since we will only explicitly store the quotient graph, we only attempt to detect the subset whose adjacency and element lists are equivalent (modulo members of the supervariable).

Rather than performing all-pairs equivalence checks, we can compute hashes of each variable's adjacency and element lists and only perform a full equivalence check if the hashes compare.

Supervariable detection

An important performance optimization is detecting **indistinguishable variables**, or **supervariables**: cliques within the elimination graph where each member has the same external adjacencies.

In practice, since we will only explicitly store the quotient graph, we only attempt to detect the subset whose adjacency and element lists are equivalent (modulo members of the supervariable).

Rather than performing all-pairs equivalence checks, we can compute hashes of each variable's adjacency and element lists and only perform a full equivalence check if the hashes compare.

Supervariable detection

An important performance optimization is detecting **indistinguishable variables**, or **supervariables**: cliques within the elimination graph where each member has the same external adjacencies.

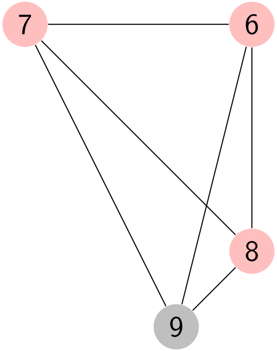
In practice, since we will only explicitly store the quotient graph, we only attempt to detect the subset whose adjacency and element lists are equivalent (modulo members of the supervariable).

Rather than performing all-pairs equivalence checks, we can compute hashes of each variable's adjacency and element lists and only perform a full equivalence check if the hashes compare.

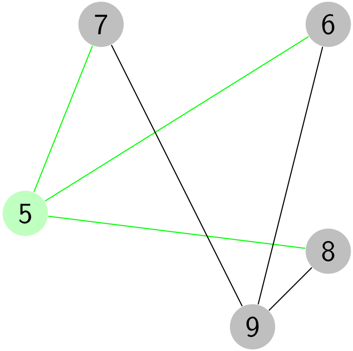
Supervariable detection

In our full elimination/quotient graph example process, a supervariable of nodes $\{6, 7, 8\}$ would be detected after the elimination of variable 5:

Elimination graph:



Quotient graph:



Nested Dissection reordering

Minimum degree reordering is a *greedy* method, as it chooses which index to eliminate based upon the fill-in upper bound, not on its overall impact on the cost of the factorization.

On the opposite side of the spectrum is **nested dissection** orderings, which, roughly speaking, recursively find small bisectors which are placed at the *end* of the elimination order. Typically, these approaches are most effective for graphs with low-dimensional topology (e.g., from a finite element discretization).

For regular grids, these reorderings can easily be analytically computed. More generally, multilevel graph partitioners such as Chaco, METIS, and SCOTCH are the norm.

Nested Dissection reordering

Minimum degree reordering is a *greedy* method, as it chooses which index to eliminate based upon the fill-in upper bound, not on its overall impact on the cost of the factorization.

On the opposite side of the spectrum is **nested dissection** orderings, which, roughly speaking, recursively find small bisectors which are placed at the *end* of the elimination order. Typically, these approaches are most effective for graphs with low-dimensional topology (e.g., from a finite element discretization).

For regular grids, these reorderings can easily be analytically computed. More generally, multilevel graph partitioners such as Chaco, METIS, and SCOTCH are the norm.

Nested Dissection reordering

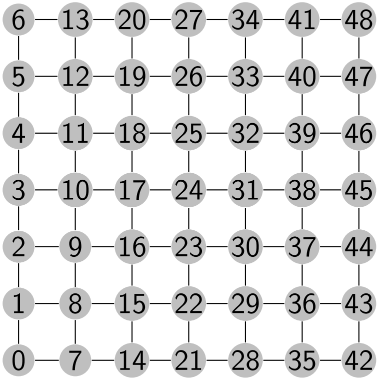
Minimum degree reordering is a *greedy* method, as it chooses which index to eliminate based upon the fill-in upper bound, not on its overall impact on the cost of the factorization.

On the opposite side of the spectrum is **nested dissection** orderings, which, roughly speaking, recursively find small bisectors which are placed at the *end* of the elimination order. Typically, these approaches are most effective for graphs with low-dimensional topology (e.g., from a finite element discretization).

For regular grids, these reorderings can easily be analytically computed. More generally, multilevel graph partitioners such as Chaco, METIS, and SCOTCH are the norm.

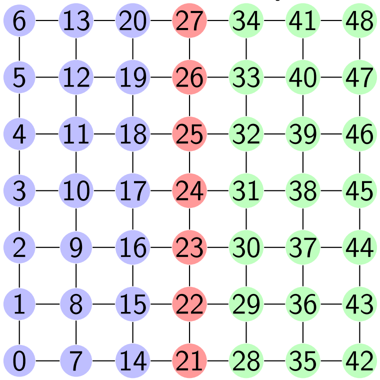
Nested Dissection reordering

As an example, consider the planar graph:



Nested Dissection reordering

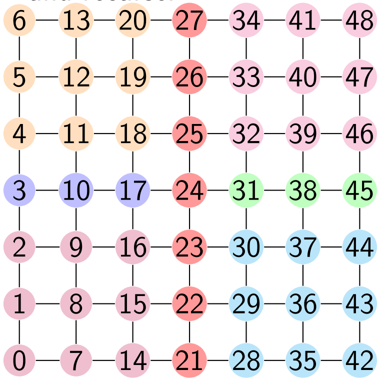
which we can nodally bisect as:



$$\begin{pmatrix} A_{0,0} & 0 & A_{0,2} \\ 0 & A_{1,1} & A_{1,2} \\ A_{2,0} & A_{2,1} & A_{2,2} \end{pmatrix}$$

Nested Dissection reordering

and recurse:



$$\begin{pmatrix} \begin{pmatrix} A^0_{0,0} & 0 & A^0_{0,2} \\ 0 & A^0_{1,1} & A^0_{1,2} \\ A^0_{2,0} & A^0_{2,1} & A^0_{2,2} \end{pmatrix} & 0 & A_{0,2} \\ 0 & \begin{pmatrix} A^1_{0,0} & 0 & A^1_{0,2} \\ 0 & A^1_{1,1} & A^1_{0,2} \\ A^1_{2,0} & A^1_{2,1} & A^1_{2,2} \end{pmatrix} & A_{1,2} \\ A_{2,0} & A_{2,1} & A_{2,2} \end{pmatrix}$$

Scalar forest and structure

Once an ordering has been determined, the precise set of nonzero indices in each column of the factorization – its **structure** – needs to be computed.

The **elimination parent** of column j is the first below-diagonal nonzero index of column j of the factor. Or, in other words, the first column index which is effected by the elimination of column j.

The **elimination forest** is the implied forest structure.

Denoting the original below-diagonal structure of column j of the sparse matrix A as $\mathcal{A}_j$, and the structure of column j of the sparse lower-triangular factor as $\mathcal{L}_j$, we have that

$$\mathcal{L}_j = \mathcal{A}_j \cup_{c \in \text{children}(j)} \mathcal{L}_c \setminus \{j\}.$$

For each leaf column j, we have that $\mathcal{L}_j = \mathcal{A}_j$, and we can work our way up the **elimination forest** to compute each column's structure.

Scalar forest and structure

Once an ordering has been determined, the precise set of nonzero indices in each column of the factorization – its **structure** – needs to be computed.

The **elimination parent** of column j is the first below-diagonal nonzero index of column j of the factor. Or, in other words, the first column index which is effected by the elimination of column j.

The **elimination forest** is the implied forest structure.

Denoting the original below-diagonal structure of column j of the sparse matrix A as $\mathcal{A}_j$, and the structure of column j of the sparse lower-triangular factor as $\mathcal{L}_j$, we have that

$$\mathcal{L}_j = \mathcal{A}_j \cup_{c \in \text{children}(j)} \mathcal{L}_c \setminus \{j\}.$$

For each leaf column j, we have that $\mathcal{L}_j = \mathcal{A}_j$, and we can work our way up the **elimination forest** to compute each column's structure.

Scalar forest and structure

Once an ordering has been determined, the precise set of nonzero indices in each column of the factorization – its **structure** – needs to be computed.

The **elimination parent** of column j is the first below-diagonal nonzero index of column j of the factor. Or, in other words, the first column index which is effected by the elimination of column j.

The **elimination forest** is the implied forest structure.

Denoting the original below-diagonal structure of column j of the sparse matrix A as $\mathcal{A}_j$, and the structure of column j of the sparse lower-triangular factor as $\mathcal{L}_j$, we have that

$$\mathcal{L}_j = \mathcal{A}_j \cup_{c \in \text{children}(j)} \mathcal{L}_c \setminus \{j\}.$$

For each leaf column j, we have that $\mathcal{L}_j = \mathcal{A}_j$, and we can work our way up the **elimination forest** to compute each column's structure.

Scalar forest and structure

Once an ordering has been determined, the precise set of nonzero indices in each column of the factorization – its **structure** – needs to be computed.

The **elimination parent** of column j is the first below-diagonal nonzero index of column j of the factor. Or, in other words, the first column index which is effected by the elimination of column j.

The **elimination forest** is the implied forest structure.

Denoting the original below-diagonal structure of column j of the sparse matrix A as $\mathcal{A}_j$, and the structure of column j of the sparse lower-triangular factor as $\mathcal{L}_j$, we have that

$$\mathcal{L}_j = \mathcal{A}_j \cup_{c \in \text{children}(j)} \mathcal{L}_c \setminus \{j\}.$$

For each leaf column j, we have that $\mathcal{L}_j = \mathcal{A}_j$, and we can work our way up the **elimination forest** to compute each column's structure.

Scalar forest and structure

Once an ordering has been determined, the precise set of nonzero indices in each column of the factorization – its **structure** – needs to be computed.

The **elimination parent** of column j is the first below-diagonal nonzero index of column j of the factor. Or, in other words, the first column index which is effected by the elimination of column j.

The **elimination forest** is the implied forest structure.

Denoting the original below-diagonal structure of column j of the sparse matrix A as $\mathcal{A}_j$, and the structure of column j of the sparse lower-triangular factor as $\mathcal{L}_j$, we have that

$$\mathcal{L}_j = \mathcal{A}_j \cup_{c \in \text{children}(j)} \mathcal{L}_c \setminus \{j\}.$$

For each leaf column j, we have that $\mathcal{L}_j = \mathcal{A}_j$, and we can work our way up the **elimination forest** to compute each column's structure.

Fundamental supernodes

As we know from dense factorizations, unblocked factorizations spend their time in memory-bandwidth constrained level 2 BLAS, whereas blocked algorithms perform most of their work in level 3 BLAS.

One can, in linear time, identify an initial grouping, called a **fundamental supernode partition**, of contiguous columns whose structures – modulo the group itself – are identical.

If the only child of column j is column $j-1$, and the degree of column j is exactly one less than column $j-1$, then they are in the same fundamental supernode.

$$\left(\begin{array}{c|cc} x & & \\ \hline & x & \\ & x & x \\ \hline x & & \\ & x & x \\ x & x & x \end{array}\right), \quad \left(\begin{array}{c|c|c} x & & \\ \hline & x & \\ \hline x & x & x \\ \hline x & & \\ & x & x \\ x & x & x \end{array}\right)$$

Such a process requires knowledge of the elimnination forest and structure degrees, but importantly **not the structure itself**.

Fundamental supernodes

As we know from dense factorizations, unblocked factorizations spend their time in memory-bandwidth constrained level 2 BLAS, whereas blocked algorithms perform most of their work in level 3 BLAS.

One can, in linear time, identify an initial grouping, called a **fundamental supernode partition**, of contiguous columns whose structures – modulo the group itself – are identical.

If the only child of column j is column $j-1$, and the degree of column j is exactly one less than column $j-1$, then they are in the same fundamental supernode.

$$\left(\begin{array}{c|cc} x & & \\ \hline & x & \\ & x & x \\ \hline x & & \\ & x & x \\ x & x & x \end{array}\right), \quad \left(\begin{array}{c|c|c} x & & \\ \hline & x & \\ \hline x & x & x \\ \hline x & & \\ & x & x \\ x & x & x \end{array}\right)$$

Such a process requires knowledge of the elimnination forest and structure degrees, but importantly **not the structure itself**.

Fundamental supernodes

As we know from dense factorizations, unblocked factorizations spend their time in memory-bandwidth constrained level 2 BLAS, whereas blocked algorithms perform most of their work in level 3 BLAS.

One can, in linear time, identify an initial grouping, called a **fundamental supernode partition**, of contiguous columns whose structures – modulo the group itself – are identical.

If the only child of column j is column $j-1$, and the degree of column j is exactly one less than column $j-1$, then they are in the same fundamental supernode.

$$\left(\begin{array}{c|cc} \times & & \\ \hline & \times & \\ & \times & \times \\ \hline \times & & \\ & \times & \times \\ \times & \times & \times \end{array}\right), \quad \left(\begin{array}{c|c|c} \times & & \\ \hline & \times & \\ \hline \times & \times & \times \\ \hline \times & & \\ & \times & \times \\ \times & \times & \times \end{array}\right)$$

Such a process requires knowledge of the elimnination forest and structure degrees, but importantly **not the structure itself**.

Fundamental supernodes

As we know from dense factorizations, unblocked factorizations spend their time in memory-bandwidth constrained level 2 BLAS, whereas blocked algorithms perform most of their work in level 3 BLAS.

One can, in linear time, identify an initial grouping, called a **fundamental supernode partition**, of contiguous columns whose structures – modulo the group itself – are identical.

If the only child of column j is column $j-1$, and the degree of column j is exactly one less than column $j-1$, then they are in the same fundamental supernode.

$$\left(\begin{array}{c|cc} x & & \\ \hline & x & \\ & x & x \\ \hline x & & \\ & x & x \\ x & x & x \end{array}\right), \quad \left(\begin{array}{c|c|c} x & & \\ \hline & x & \\ \hline x & x & x \\ \hline x & & \\ & x & x \\ x & x & x \end{array}\right)$$

Such a process requires knowledge of the elimnination forest and structure degrees, but importantly **not the structure itself**.

Fundamental supernodes

As we know from dense factorizations, unblocked factorizations spend their time in memory-bandwidth constrained level 2 BLAS, whereas blocked algorithms perform most of their work in level 3 BLAS.

One can, in linear time, identify an initial grouping, called a **fundamental supernode partition**, of contiguous columns whose structures – modulo the group itself – are identical.

If the only child of column j is column $j-1$, and the degree of column j is exactly one less than column $j-1$, then they are in the same fundamental supernode.

$$\left(\begin{array}{c|cc} x & & \\ \hline & x & \\ & x & x \\ \hline x & & \\ & x & x \\ x & x & x \end{array}\right), \quad \left(\begin{array}{c|c|c} x & & \\ \hline & x & \\ \hline x & x & x \\ \hline x & & \\ & x & x \\ x & x & x \end{array}\right)$$

Such a process requires knowledge of the elimnination forest and structure degrees, but importantly **not the structure itself**.

Supernode relaxation

The fundamental supernodes imply a **fundamental supernode elimination forest**, which leads to dense Cholesky factorizations on the fundamental diagonal blocks and symmetric/Hermitian rank-k updates when forming Schur complements.

But the fundamental supernodes towards the bottom of the fundamental supernode elimination forest are often very lower cardinality – often even of size 1 – despite neighboring structures being mostly identical.

The process of **supernode relaxation/amalgomation** is combining the supernodes of nodes in the fundamental supernode elimination forest with their parents whenever acceptably small percentages of nonzeros would be introduced.

Importantly, amalgomation does not require knowledge of the fundamental supernode structure, only its degree.

Supernode relaxation

The fundamental supernodes imply a **fundamental supernode elimination forest**, which leads to dense Cholesky factorizations on the fundamental diagonal blocks and symmetric/Hermitian rank-k updates when forming Schur complements.

But the fundamental supernodes towards the bottom of the fundamental supernode elimination forest are often very lower cardinality – often even of size 1 – despite neighboring structures being mostly identical.

The process of **supernode relaxation/amalgomation** is combining the supernodes of nodes in the fundamental supernode elimination forest with their parents whenever acceptably small percentages of nonzeros would be introduced.

Importantly, amalgomation does not require knowledge of the fundamental supernode structure, only its degree.

Supernode relaxation

The fundamental supernodes imply a **fundamental supernode elimination forest**, which leads to dense Cholesky factorizations on the fundamental diagonal blocks and symmetric/Hermitian rank-k updates when forming Schur complements.

But the fundamental supernodes towards the bottom of the fundamental supernode elimination forest are often very lower cardinality – often even of size 1 – despite neighboring structures being mostly identical.

The process of **supernode relaxation/amalgomation** is combining the supernodes of nodes in the fundamental supernode elimination forest with their parents whenever acceptably small percentages of nonzeros would be introduced.

Importantly, amalgomation does not require knowledge of the fundamental supernode structure, only its degree.

Supernode relaxation

The fundamental supernodes imply a **fundamental supernode elimination forest**, which leads to dense Cholesky factorizations on the fundamental diagonal blocks and symmetric/Hermitian rank-k updates when forming Schur complements.

But the fundamental supernodes towards the bottom of the fundamental supernode elimination forest are often very lower cardinality – often even of size 1 – despite neighboring structures being mostly identical.

The process of **supernode relaxation/amalgomation** is combining the supernodes of nodes in the fundamental supernode elimination forest with their parents whenever acceptably small percentages of nonzeros would be introduced.

Importantly, amalgomation does not require knowledge of the fundamental supernode structure, only its degree.

Multifrontal factorization

While the most popular sparse-direct Cholesky factorization implementation, CHOLMOD [Davis et al.], is *left-looking*, we will discuss the *right-looking multifrontal* equivalent because it is more amenable to parallelism.

Given a supernode with column indices $s_j = [j, ..., j + n_j)$ and structure $\mathcal{L}_j$, we initialize its **front** as a symmetric/Hermitian matrix of order $|s_j| + |\mathcal{L}_j|$:

$$F_j = \left(\begin{array}{c|c} A_{s_j} & 0 \\ \hline A_{\mathcal{L}_j, s_j} & 0 \end{array}\right).$$

The term **multifrontal** refers to each supernode's work being captured within their front, which will be left containing

$$\hat{F}_j = \left(\begin{array}{c|c} L_{s_j} & 0 \\ \hline L_{\mathcal{L}_j, s_j} & S_j \end{array}\right),$$

where S_j is the Schur complement update $-L_{\mathcal{L}_j, s_j} L'_{\mathcal{L}_j, s_j}$.

Once all of a node's children's Schur complement updates have been formed, the updates are added into the parent. The parent supernode's portion of L can then be computed in its front, as can its update.

Multifrontal factorization

While the most popular sparse-direct Cholesky factorization implementation, CHOLMOD [Davis et al.], is *left-looking*, we will discuss the *right-looking multifrontal* equivalent because it is more amenable to parallelism.

Given a supernode with column indices $s_j = [j, ..., j + n_j)$ and structure $\mathcal{L}_j$, we initialize its **front** as a symmetric/Hermitian matrix of order $|s_j| + |\mathcal{L}_j|$:

$$F_j = \left(\begin{array}{c|c} A_{s_j} & 0 \\ \hline A_{\mathcal{L}_j, s_j} & 0 \end{array}\right).$$

The term **multifrontal** refers to each supernode's work being captured within their front, which will be left containing

$$\hat{F}_j = \left(\begin{array}{c|c} L_{s_j} & 0 \\ \hline L_{\mathcal{L}_j, s_j} & S_j \end{array}\right),$$

where S_j is the Schur complement update $-L_{\mathcal{L}_j,s_j} L'_{\mathcal{L}_j,s_j}$.

Once all of a node's children's Schur complement updates have been formed, the updates are added into the parent. The parent supernode's portion of L can then be computed in its front, as can its update.

Multifrontal factorization

While the most popular sparse-direct Cholesky factorization implementation, CHOLMOD [Davis et al.], is *left-looking*, we will discuss the *right-looking multifrontal* equivalent because it is more amenable to parallelism.

Given a supernode with column indices $s_j = [j, ..., j + n_j)$ and structure $\mathcal{L}_j$, we initialize its **front** as a symmetric/Hermitian matrix of order $|s_j| + |\mathcal{L}_j|$:

$$F_j = \left(\begin{array}{c|c} A_{s_j} & 0 \\ \hline A_{\mathcal{L}_j, s_j} & 0 \end{array}\right).$$

The term **multifrontal** refers to each supernode's work being captured within their front, which will be left containing

$$\hat{F}_j = \left(\begin{array}{c|c} L_{s_j} & 0 \\ \hline L_{\mathcal{L}_j, s_j} & S_j \end{array}\right),$$

where S_j is the Schur complement update $-L_{\mathcal{L}_j, s_j} L'_{\mathcal{L}_j, s_j}$.

Once all of a node's children's Schur complement updates have been formed, the updates are added into the parent. The parent supernode's portion of L can then be computed in its front, as can its update.

Multifrontal factorization

While the most popular sparse-direct Cholesky factorization implementation, CHOLMOD [Davis et al.], is *left-looking*, we will discuss the *right-looking multifrontal* equivalent because it is more amenable to parallelism.

Given a supernode with column indices $s_j = [j, ..., j + n_j)$ and structure $\mathcal{L}_j$, we initialize its **front** as a symmetric/Hermitian matrix of order $|s_j| + |\mathcal{L}_j|$:

$$F_j = \left(\begin{array}{c|c} A_{s_j} & 0 \\ \hline A_{\mathcal{L}_j,s_j} & 0 \end{array}\right).$$

The term **multifrontal** refers to each supernode's work being captured within their front, which will be left containing

$$\hat{F}_j = \left(\begin{array}{c|c} L_{s_j} & 0 \\ \hline L_{\mathcal{L}_j,s_j} & S_j \end{array}\right),$$

where S_j is the Schur complement update $-L_{\mathcal{L}_j,s_j} L'_{\mathcal{L}_j,s_j}$.

Once all of a node's children's Schur complement updates have been formed, the updates are added into the parent. The parent supernode's portion of L can then be computed in its front, as can its update.

Right-looking sampling

Just as we could modify a dense LDL^H factorization to sample a DPP by decrementing pivots by one when their corresponding index was not kept, we need only make the same modification for sparse-direct DPP sampling.

In fact, we need only plug in our dense DPP sampler in place of the dense LDL^H factorizations of supernodal diagonal blocks!

An efficient interface for sparse-direct sampling from a DPP should therefore cache the symbolic analysis and simply reinitialize the fronts for each sample.

Right-looking sampling

Just as we could modify a dense LDL^H factorization to sample a DPP by decrementing pivots by one when their corresponding index was not kept, we need only make the same modification for sparse-direct DPP sampling.

In fact, we need only plug in our dense DPP sampler in place of the dense LDL^H factorizations of supernodal diagonal blocks!

An efficient interface for sparse-direct sampling from a DPP should therefore cache the symbolic analysis and simply reinitialize the fronts for each sample.

Right-looking sampling

Just as we could modify a dense LDL^H factorization to sample a DPP by decrementing pivots by one when their corresponding index was not kept, we need only make the same modification for sparse-direct DPP sampling.

In fact, we need only plug in our dense DPP sampler in place of the dense LDL^H factorizations of supernodal diagonal blocks!

An efficient interface for sparse-direct sampling from a DPP should therefore cache the symbolic analysis and simply reinitialize the fronts for each sample.

Parallelization

At one point, shared-memory parallelization of a sparse-direct solver was considered a heroic feat of software engineering.

But, with the introduction of OpenMP 4.0 task dependencies, one can easily blend dynamic task scheduling for the factorizations of the dense fronts with the embarrassing parallelism exposed by the supernodal elimination forest.

We refer to [Hogg/Reid/Scott-2009] for an early implementation of such an approach for sparse-direct factorization.

Due to our DPP factorization technique, the parallelization for sparse-direct DPP sampling is essentially identical.

Parallelization

At one point, shared-memory parallelization of a sparse-direct solver was considered a heroic feat of software engineering.

But, with the introduction of OpenMP 4.0 task dependencies, one can easily blend dynamic task scheduling for the factorizations of the dense fronts with the embarrassing parallelism exposed by the supernodal elimination forest.

We refer to [Hogg/Reid/Scott-2009] for an early implementation of such an approach for sparse-direct factorization.

Due to our DPP factorization technique, the parallelization for sparse-direct DPP sampling is essentially identical.

Parallelization

At one point, shared-memory parallelization of a sparse-direct solver was considered a heroic feat of software engineering.

But, with the introduction of OpenMP 4.0 task dependencies, one can easily blend dynamic task scheduling for the factorizations of the dense fronts with the embarrassing parallelism exposed by the supernodal elimination forest.

We refer to [Hogg/Reid/Scott-2009] for an early implementation of such an approach for sparse-direct factorization.

Due to our DPP factorization technique, the parallelization for sparse-direct DPP sampling is essentially identical.

Parallelization

At one point, shared-memory parallelization of a sparse-direct solver was considered a heroic feat of software engineering.

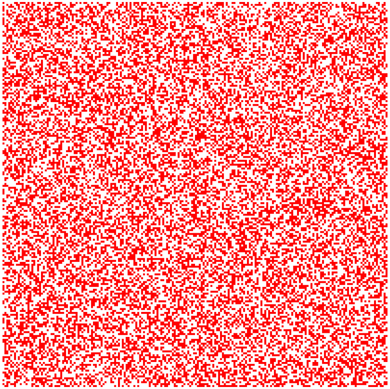
But, with the introduction of OpenMP 4.0 task dependencies, one can easily blend dynamic task scheduling for the factorizations of the dense fronts with the embarrassing parallelism exposed by the supernodal elimination forest.

We refer to [Hogg/Reid/Scott-2009] for an early implementation of such an approach for sparse-direct factorization.

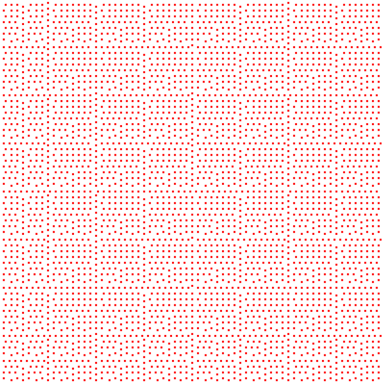
Due to our DPP factorization technique, the parallelization for sparse-direct DPP sampling is essentially identical.

(MAP) Sampling from 2D $-\sigma\Delta$

```
$ ./dpp_shifted_2d_negative_laplacian \
    --x_size=200 --y_size=200 --scale=0.72
```



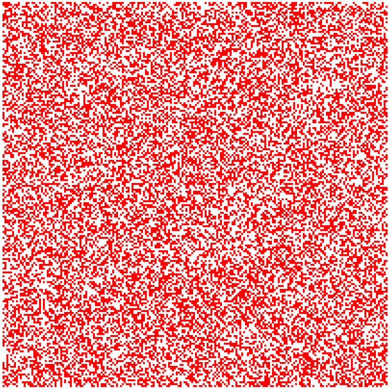
Log-likelihood: -27472.2
Sample time: 0.0107 seconds



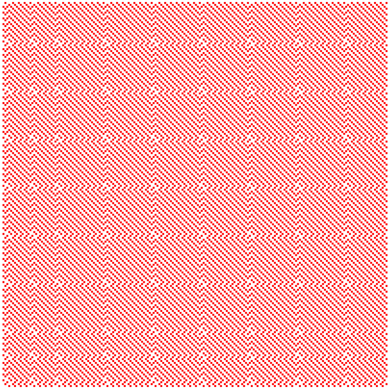
Log-likelihood: -26058
Sample time: 0.0112 seconds

(MAP) Sampling from 2D $-\sigma\Delta$

```
$ ./dpp_shifted_2d_negative_laplacian \
    --x_size=200 --y_size=200 --scale=0.75
```



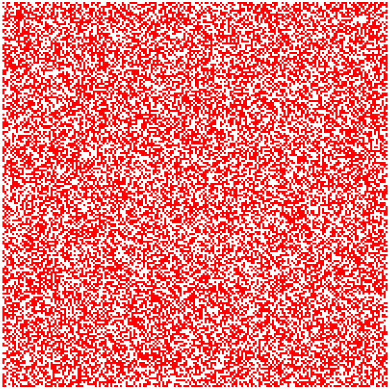
Log-likelihood: -27612.6
Sample time: 0.0124 seconds



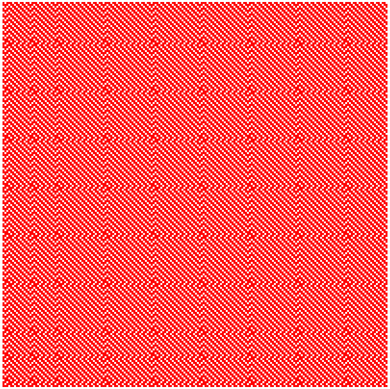
Log-likelihood: -26009
Sample time: 0.0114 seconds

(MAP) Sampling from 2D $-\sigma\Delta$

```
$ ./dpp_shifted_2d_negative_laplacian \
    --x_size=200 --y_size=200 --scale=0.85
```



Log-likelihood: -27581.7
Sample time: 0.0114 seconds



Log-likelihood: -25765
Sample time: 0.0118 seconds

Discussion

Availability:
Quotient is available under the Mozilla Public License 2.0 at hodgestar.com/quotient/ and gitlab.com/hodge_star/quotient.
This talk is based on version 0.3.

Catamari is available under the Mozilla Public License 2.0 at hodgestar.com/catamari/ and gitlab.com/hodge_star/catamari.
This talk is based on version 0.3.

These slides are available at:
hodgestar.com/G2S3/

Questions/comments?
Chatroom at:

`https://gitter.im/hodge_star/G2S3`