

CS 140:
***Sparse Matrix-Vector Multiplication
and Graph Partitioning***

Parallel sparse matrix-vector product

- Lay out matrix and vectors by rows
- $y(i) = \text{sum}(A(i,j)*x(j))$
- Only compute terms with $A(i,j) \neq 0$

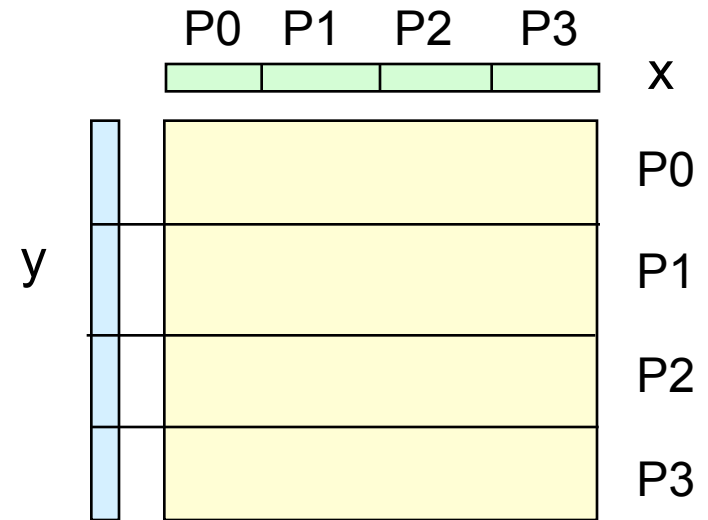
- Algorithm

Each processor i :

Broadcast $x(i)$

Compute $y(i) = A(i,:) * x$

Comm volume $v = p^*n$ (way too much!)



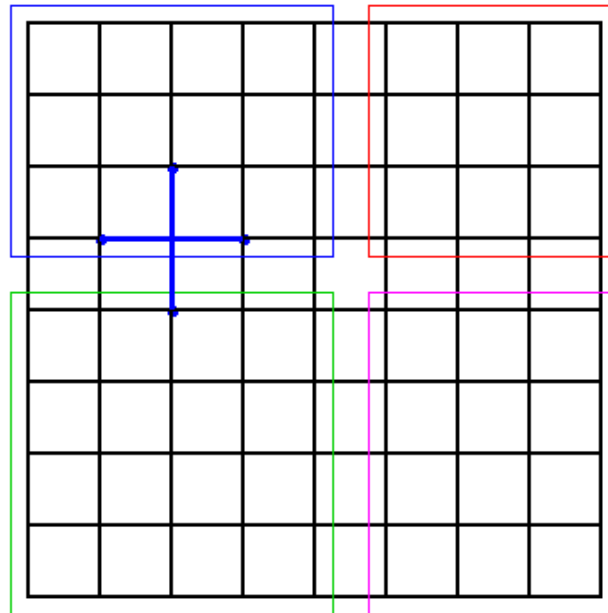
- Reducing communication volume

1. Only send each proc the parts of x it needs
2. Reorder matrix for better locality by graph partitioning

2D block decomposition for 5-point stencil

- n stencil cells, p processors
- Each processor has a patch of n/p cells
- Block row (or block col) layout: $v = 2 * p * \text{sqrt}(n)$
- 2-dimensional block layout: $v = 4 * \text{sqrt}(p) * \text{sqrt}(n)$

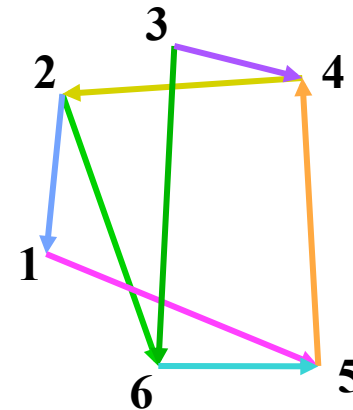
Partitioning of the 2D Heat Equation



Graphs and Sparse Matrices

- Sparse matrix is a representation of a (sparse) graph

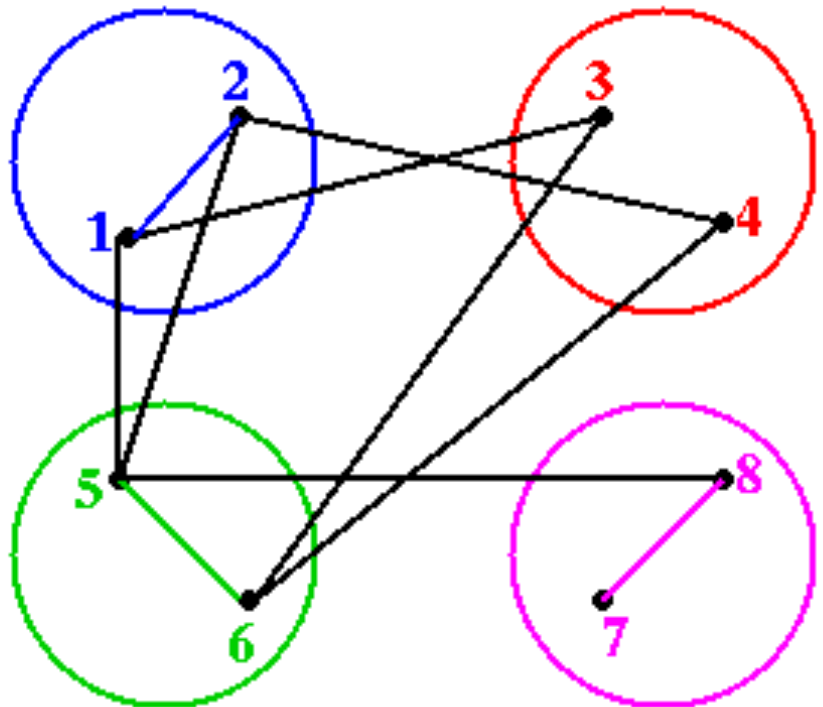
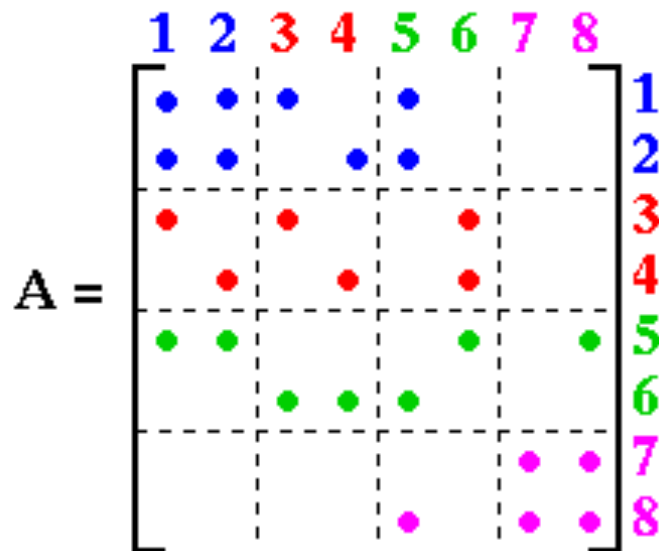
	1	2	3	4	5	6
1	1				1	
2	1	1				1
3			1	1		1
4		1		1		
5				1	1	
6					1	1



- Matrix entries are edge weights
- Number of nonzeros per row is the vertex degree
- Edges represent data dependencies in matrix-vector multiplication

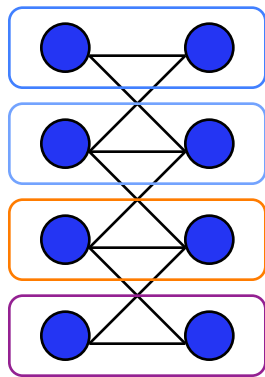
Sparse Matrix Vector Multiplication

Partitioning a Sparse Symmetric Matrix

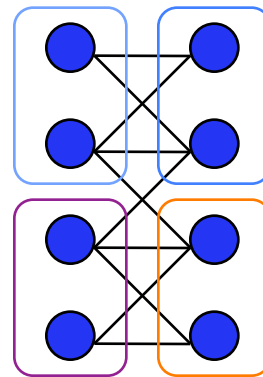


Graph partitioning

- Assigns subgraphs to processors
- Determines parallelism and locality.
- Tries to make subgraphs all same size (load balance)
- Tries to minimize edge crossings (communication).
- Exact minimization is NP-complete.



edge crossings = 6



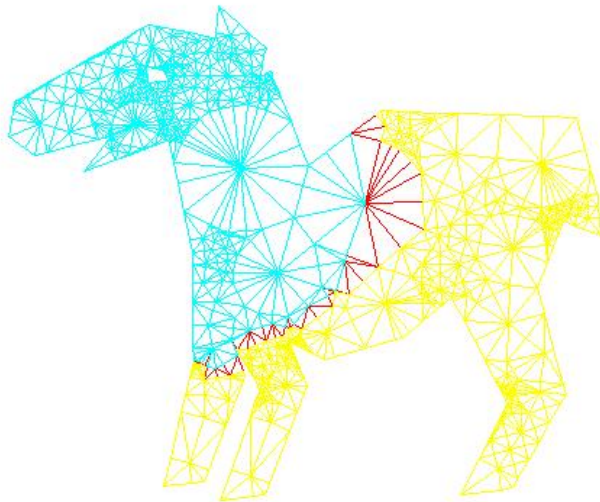
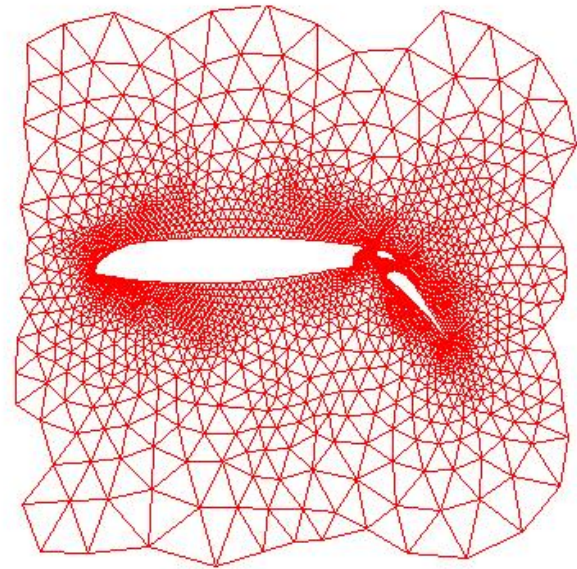
edge crossings = 10

Applications of graph partitioning

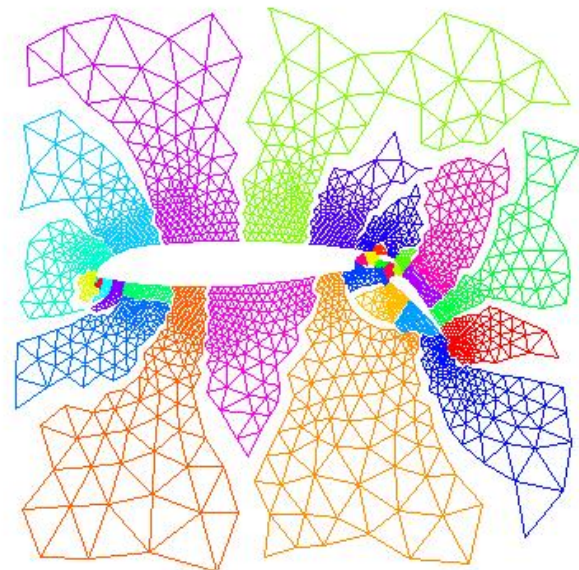
- Telephone network design
 - The original application! 1970 algorithm due to Kernighan & Lin
- Sparse Matrix times Vector Multiplication
 - To solve $Ax = b$, partial differential equations, eigenvalue problems, ...
 - $N = \{1, \dots, n\}$, $(j, k) \in E$ if $A(j, k)$ nonzero,
 - $W_N(j) = \# \text{nonzeros in row } j$, $W_E(j, k) = 1$
- Data mining and clustering
- Physical Mapping of DNA
- VLSI Layout
- Sparse Gaussian Elimination
 - Reorder matrix rows and columns to decrease “fill” in factors
- Load Balancing while Minimizing Communication
- . . .

Graph partitioning demo

- > load meshes
- > gplotg(Airfoil,Axy)
- > specdice(Airfoil,Axy,5)
- > meshdemo



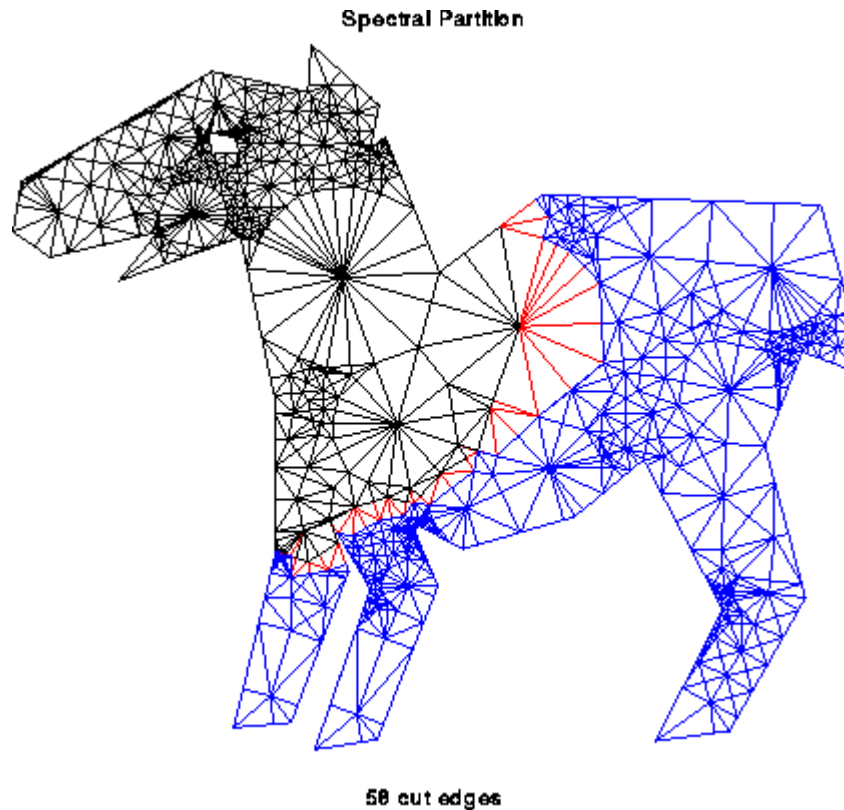
spectral bisection



32-way partition

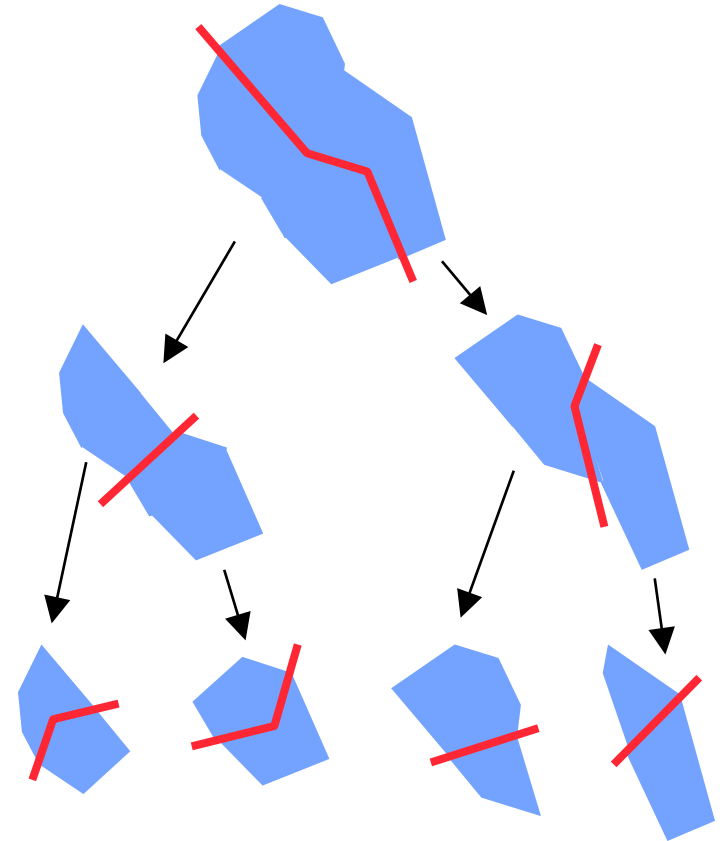
Partitioning by Repeated Bisection

- To partition into 2^k parts, bisect graph recursively k times



Recursive Bisection

- Recursive bisection approach:
 - Partition data into two sets.
 - Recursively subdivide each set into two sets.
 - Only minor modifications needed to allow $P \neq 2^n$.



CS 240A: Graph and hypergraph partitioning (excerpts)

Thanks to Aydin Buluc, Umit Catalyurek,
Alan Edelman, and Kathy Yelick
for some of these slides.

The whole CS240A partitioning lecture is at
<http://cs.ucsb.edu/~gilbert/cs240a/slides/cs240a-partitioning.pdf>

Graph partitioning in practice

- Graph partitioning heuristics have been an active research area for many years, often motivated by partitioning for parallel computation.
- Some techniques:
 - Iterative-swapping (Kernighan-Lin, Fiduccia-Matheysses)
 - Spectral partitioning (uses eigenvectors of Laplacian matrix of graph)
 - Geometric partitioning (for meshes with specified vertex coordinates)
 - Breadth-first search (fast but dated)
- Many popular modern codes (e.g. Metis, Chaco, Zoltan) use multilevel iterative swapping

Iterative swapping:

Kernighan/Lin, Fiduccia/Mattheyses

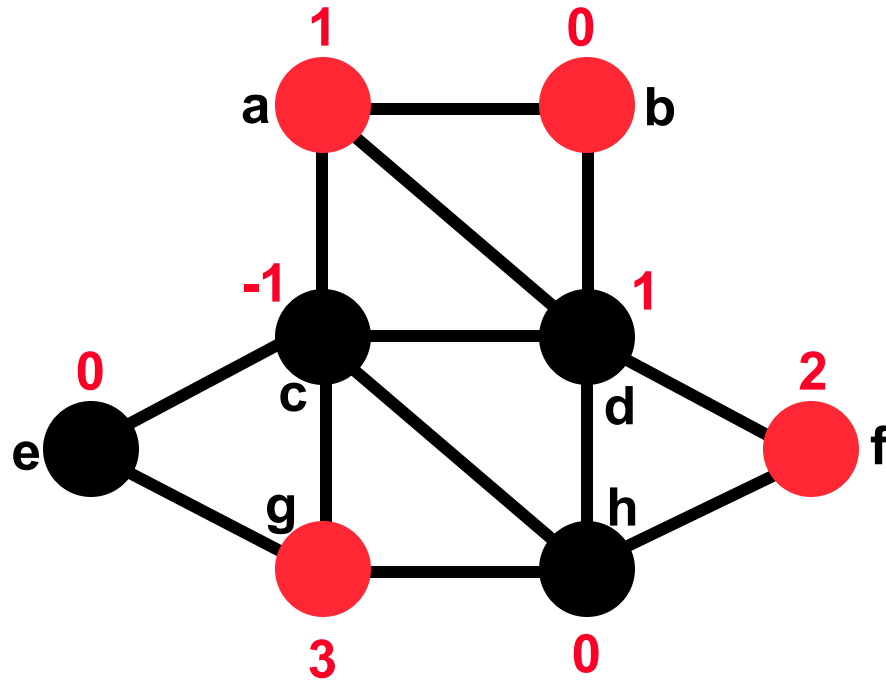
- Take a initial partition and iteratively improve it
 - Kernighan/Lin (1970), cost = $O(|N|^3)$ but simple
 - Fiduccia/Mattheyses (1982), cost = $O(|E|)$ but more complicated
- Start with a weighted graph and a partition $A \cup B$, where $|A| = |B|$
 - $T = \text{cost}(A, B) = \sum \{\text{weight}(e) : e \text{ connects nodes in } A \text{ and } B\}$
 - Find subsets X of A and Y of B with $|X| = |Y|$
 - Swapping X and Y should decrease cost:
 - $\text{newA} = A - X \cup Y$ and $\text{newB} = B - Y \cup X$
 - $\text{newT} = \text{cost}(\text{newA}, \text{newB}) < \text{cost}(A, B)$
- Compute newT efficiently for **many** possible X and Y , (not time to do all possible), then choose smallest

Simplified Fiduccia-Mattheyses: Example (1)

Red nodes are in Part1;
black nodes are in Part2.

The initial partition into two parts is arbitrary. In this case it cuts 8 edges.

The initial **node gains** are shown in red.



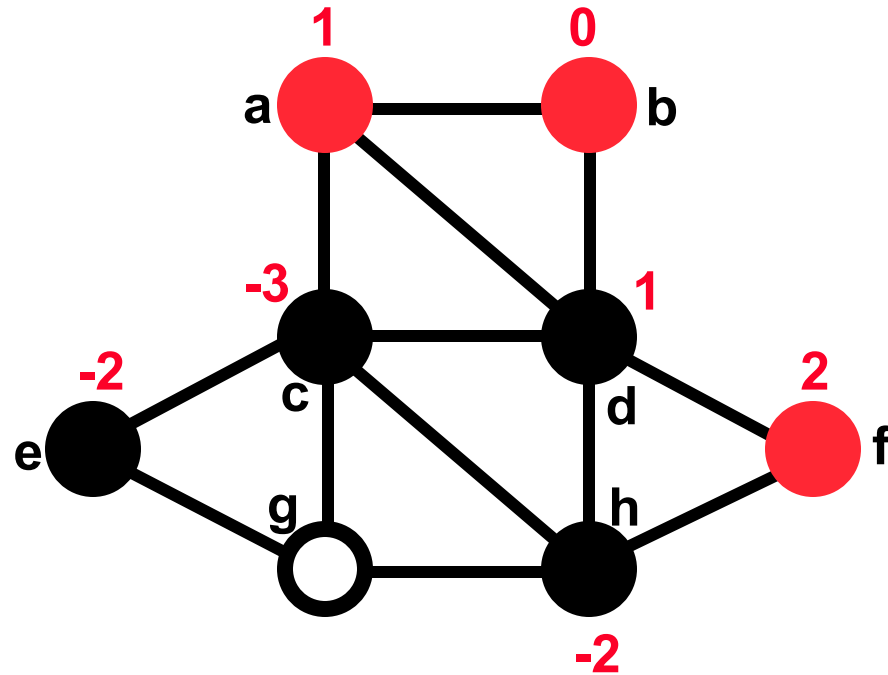
Nodes tentatively moved (and cut size after each pair):

none (8);

Simplified Fiduccia-Mattheyses: Example (2)

The node in Part1 with largest gain is g. We tentatively move it to Part2 and recompute the gains of its neighbors.

Tentatively moved nodes are hollow circles. After a node is tentatively moved its gain doesn't matter any more.



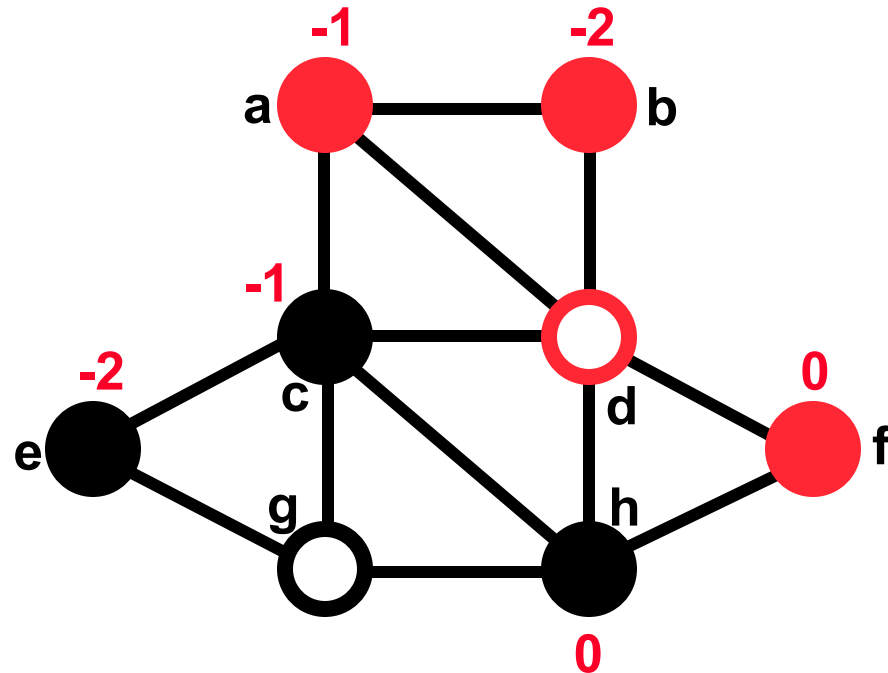
Nodes tentatively moved (and cut size after each pair):

none (8); g,

Simplified Fiduccia-Mattheyses: Example (3)

The node in Part2 with largest gain is d. We tentatively move it to Part1 and recompute the gains of its neighbors.

After this first tentative swap, the cut size is 4.

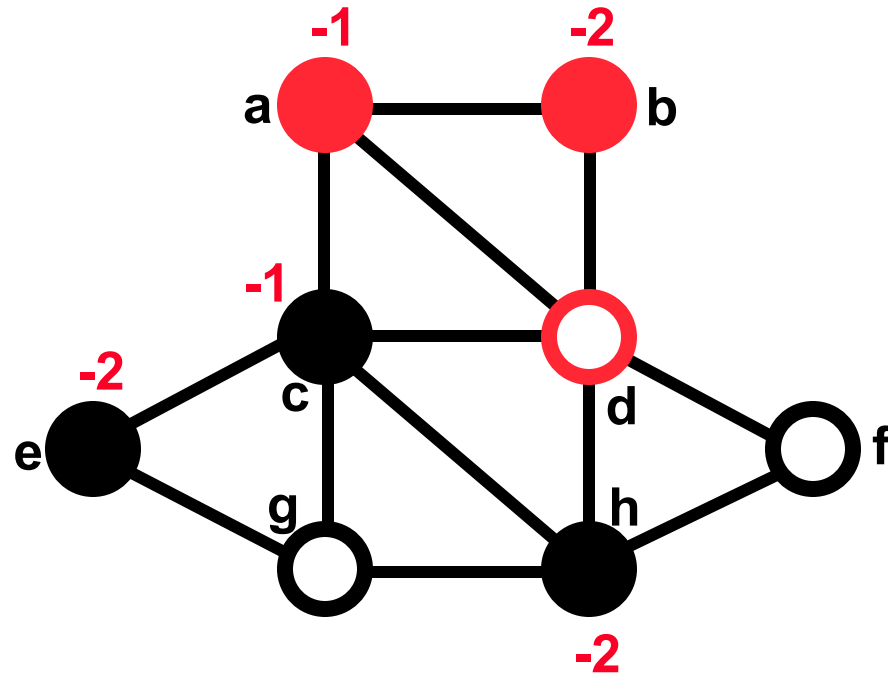


Nodes tentatively moved (and cut size after each pair):

none (8); g, d (4);

Simplified Fiduccia-Mattheyses: Example (4)

The unmoved node in Part1 with largest gain is f. We tentatively move it to Part2 and recompute the gains of its neighbors.



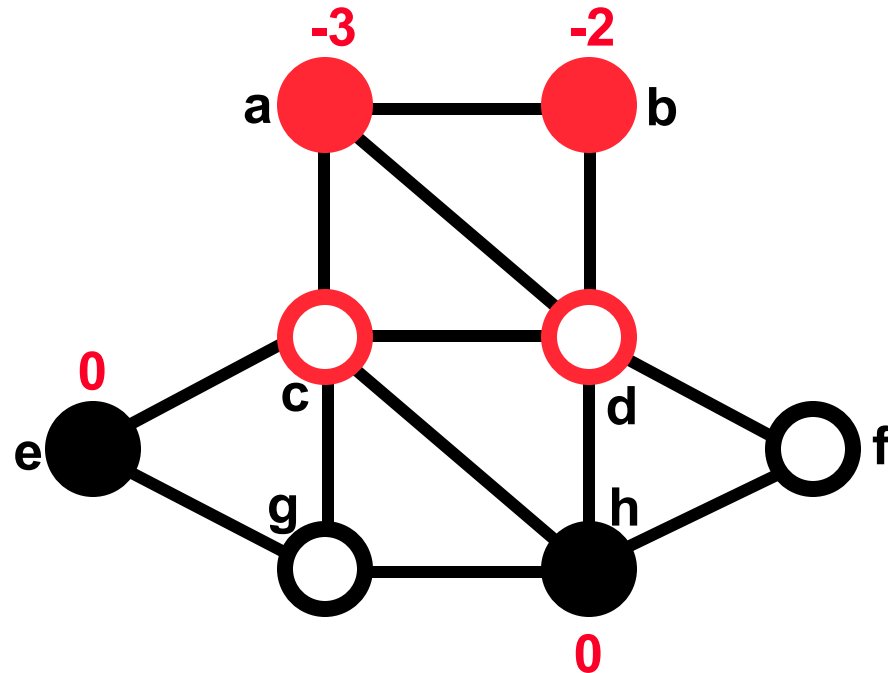
Nodes tentatively moved (and cut size after each pair):

none (8); g, d (4); f

Simplified Fiduccia-Mattheyses: Example (5)

The unmoved node in Part2 with largest gain is c.
We tentatively move it to Part1 and recompute the gains of its neighbors.

After this tentative swap,
the cut size is 5.

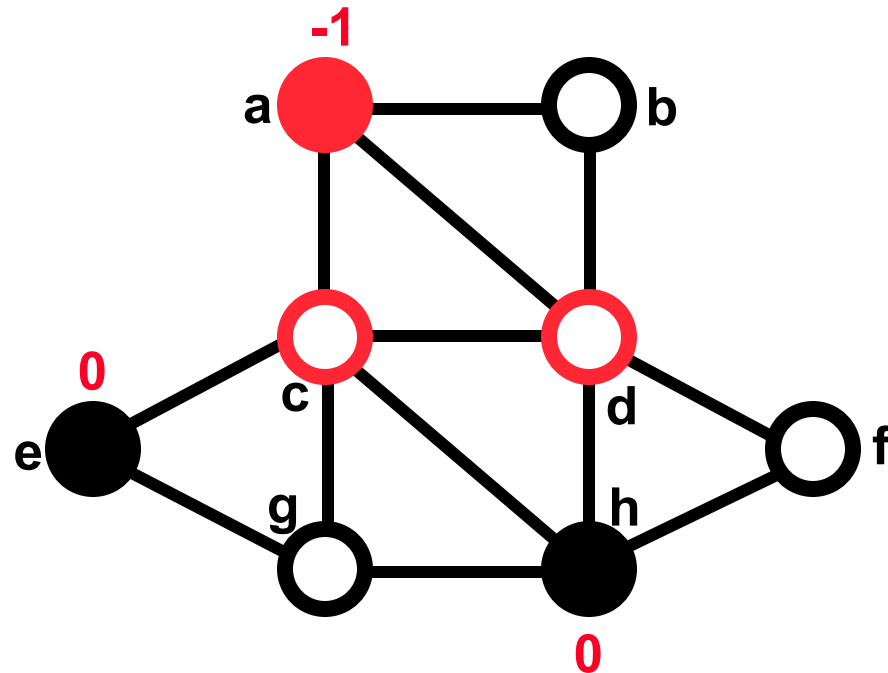


Nodes tentatively moved (and cut size after each pair):

none (8); g, d (4); f, c (5);

Simplified Fiduccia-Mattheyses: Example (6)

The unmoved node in Part1 with largest gain is b.
We tentatively move it to Part2 and recompute the gains of its neighbors.



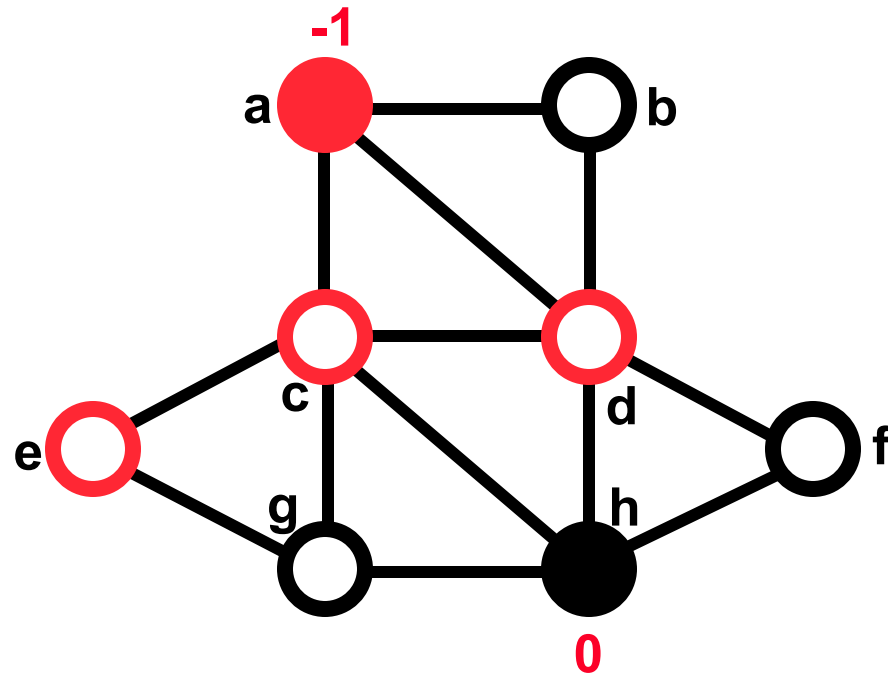
Nodes tentatively moved (and cut size after each pair):

none (8); g, d (4); f, c (5); b

Simplified Fiduccia-Mattheyses: Example (7)

There is a tie for largest gain between the two unmoved nodes in Part2. We choose one (say e) and tentatively move it to Part1. It has no unmoved neighbors so no gains are recomputed.

After this tentative swap the cut size is 7.

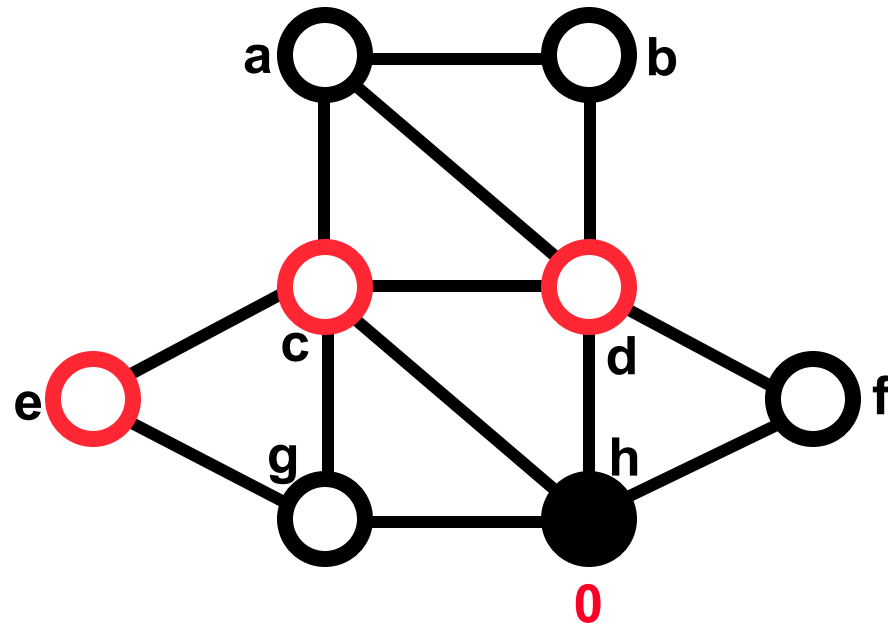


Nodes tentatively moved (and cut size after each pair):

none (8); g, d (4); f, c (5); b, e (7);

Simplified Fiduccia-Mattheyses: Example (8)

The unmoved node in Part1 with the largest gain (the only one) is a. We tentatively move it to Part2. It has no unmoved neighbors so no gains are recomputed.



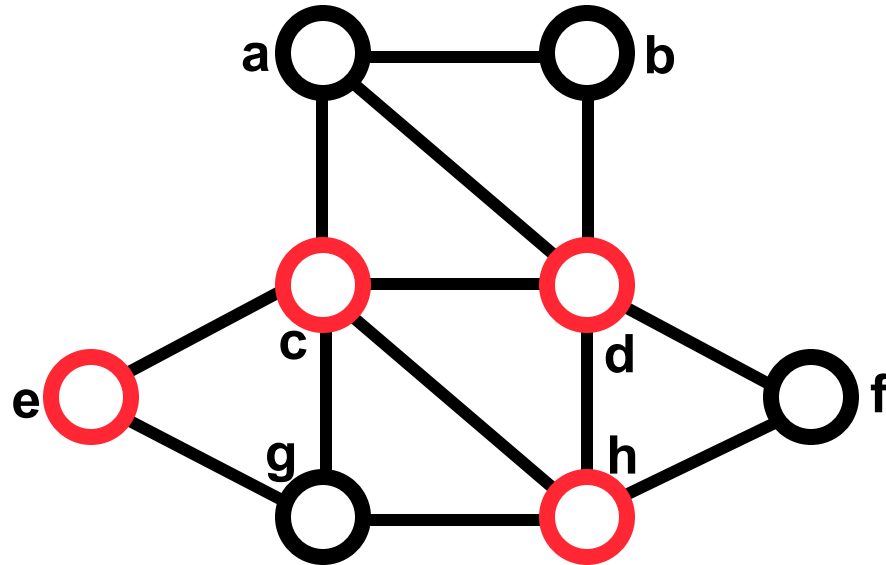
Nodes tentatively moved (and cut size after each pair):

none (8); g, d (4); f, c (5); b, e (7); a

Simplified Fiduccia-Mattheyses: Example (9)

The unmoved node in Part2 with the largest gain (the only one) is h. We tentatively move it to Part1.

The cut size after the final tentative swap is 8, the same as it was before any tentative moves.



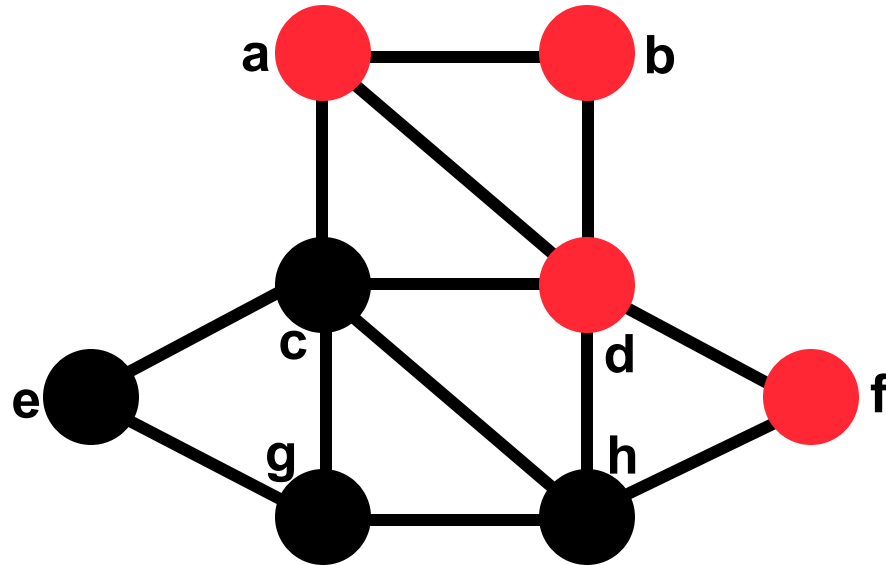
Nodes tentatively moved (and cut size after each pair):

none (8); g, d (4); f, c (5); b, e (7); a, h (8)

Simplified Fiduccia-Mattheyses: Example (10)

After every node has been tentatively moved, we look back at the sequence and see that the smallest cut was 4, after swapping g and d. We make that swap permanent and undo all the later tentative swaps.

This is the end of the first improvement step.



Nodes tentatively moved (and cut size after each pair):

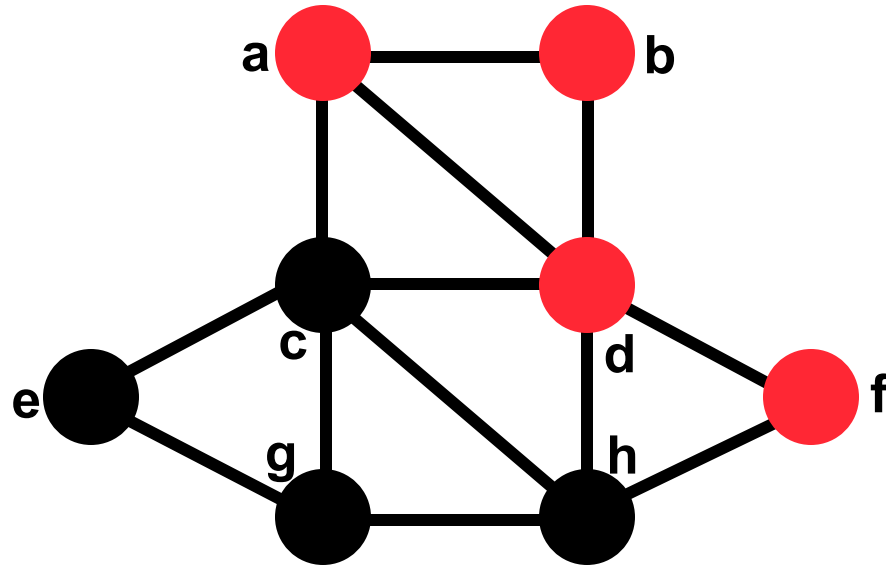
none (8); **g, d** (4); f, c (5); b, e (7); a, h (8)

Simplified Fiduccia-Mattheyses: Example (11)

Now we recompute the gains and do another improvement step starting from the new size-4 cut. The details are not shown.

The second improvement step doesn't change the cut size, so the algorithm ends with a cut of size 4.

In general, we keep doing improvement steps as long as the cut size keeps getting smaller.

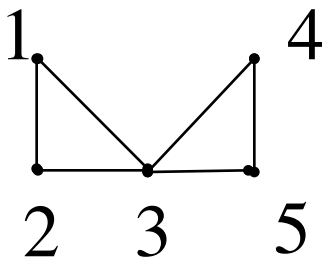


Spectral Bisection

- Based on theory of Fiedler (1970s), rediscovered several times in different communities
- Motivation I: analogy to a vibrating string
- Motivation II: continuous relaxation of discrete optimization problem
- Implementation: eigenvectors via Lanczos algorithm
 - To optimize sparse-matrix-vector multiply, we graph partition
 - To graph partition, we find an eigenvector of a matrix
 - To find an eigenvector, we do sparse-matrix-vector multiply
 - No free lunch ...

Laplacian Matrix

- *Definition:* The **Laplacian matrix** $L(G)$ of a graph G is a symmetric matrix, with one row and column for each node. It is defined by
 - $L(G)(i,i) = \text{degree of node } i \text{ (number of incident edges)}$
 - $L(G)(i,j) = -1$ if $i \neq j$ and there is an edge (i,j)
 - $L(G)(i,j) = 0$ otherwise

$G =$ 

$L(G) =$
$$\begin{pmatrix} 2 & -1 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ -1 & -1 & 4 & -1 & -1 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & -1 & 2 \end{pmatrix}$$

Properties of Laplacian Matrix

- *Theorem:* $L(G)$ has the following properties
 - $L(G)$ is symmetric.
 - This implies the eigenvalues of $L(G)$ are real, and its eigenvectors are real and orthogonal.
 - Rows of L sum to zero:
 - Let $e = [1, \dots, 1]^T$, i.e. the column vector of all ones. Then $L(G)e=0$.
 - The eigenvalues of $L(G)$ are nonnegative:
 - $0 = \lambda_1 \leq \lambda_2 \leq \dots \leq \lambda_n$
 - The number of connected components of G is equal to the number of λ_i that are 0.

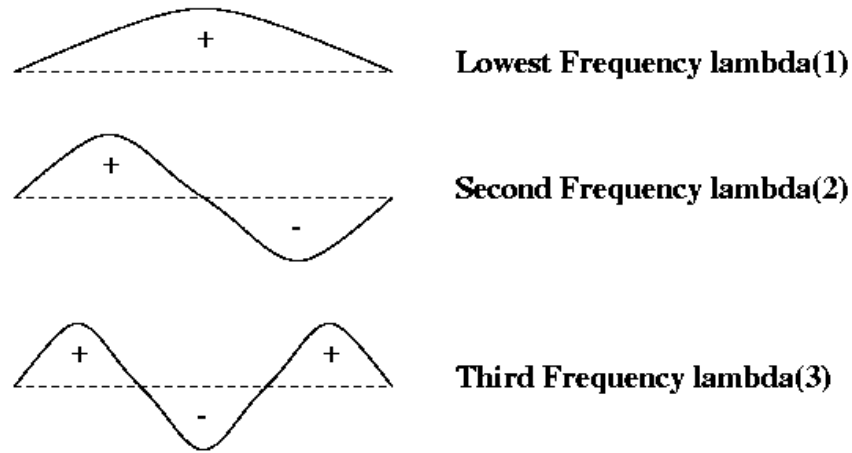
Spectral Bisection Algorithm

- Spectral Bisection Algorithm:
 - Compute eigenvector v_2 corresponding to $\lambda_2(L(G))$
 - Partition nodes around the median of $v_2(n)$
- Why in the world should this work?
- Intuition: vibrating string or membrane
- Heuristic: continuous relaxation of discrete optimization

Motivation for Spectral Bisection

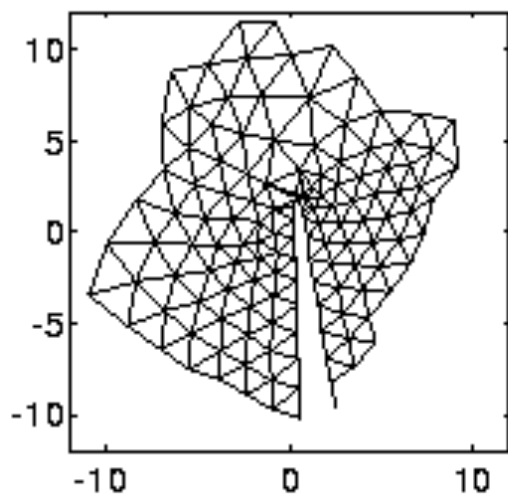
- Vibrating string
- Think of $G = 1D$ mesh as masses (nodes) connected by springs (edges), i.e. a string that can vibrate
- Vibrating string has **modes of vibration**, or **harmonics**
- Label nodes by whether mode - or + to partition into N_- and N_+
- Same idea for other graphs (eg planar graph ~ trampoline)

Modes of a Vibrating String

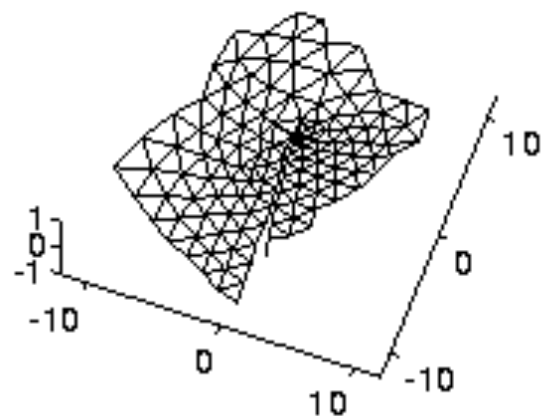


2nd eigenvector of L (planar mesh)

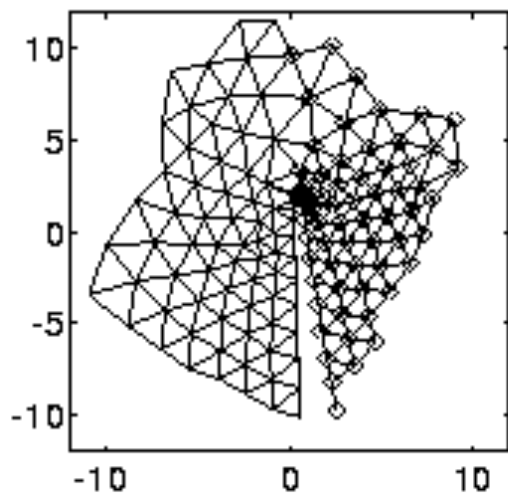
Original FE mesh



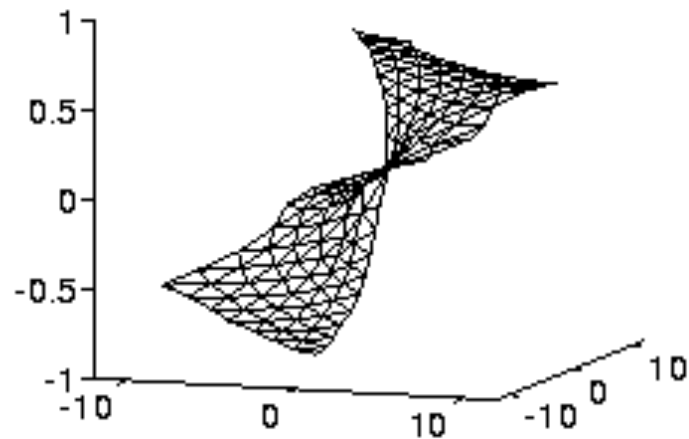
Plot of v_2 from above



Circle node i if $v_2(i) > 0$



Plot of v_2 head on



Multilevel Partitioning

- If G is too big for our algorithms, what can we do?
 - (1) Replace G by a **coarse approximation** G_C , and partition G_C instead
 - (2) Use partition of G_C to get a rough partitioning of G , and then iteratively improve it
- What if G_C is still too big?
 - Apply same idea recursively

Multilevel Partitioning - High Level Algorithm

$(N+, N-) = \text{Multilevel_Partition}(N, E)$

... recursive partitioning routine returns $N+$ and $N-$ where $N = N+ \cup N-$
if $|N|$ is small

(1) Partition $G = (N, E)$ directly to get $N = N+ \cup N-$
Return $(N+, N-)$

else

(2) Coarsen G to get an approximation $G_c = (N_c, E_c)$

(3) $(N_{c+}, N_{c-}) = \text{Multilevel_Partition}(N_c, E_c)$

(4) Expand (N_{c+}, N_{c-}) to a partition $(N+, N-)$ of N

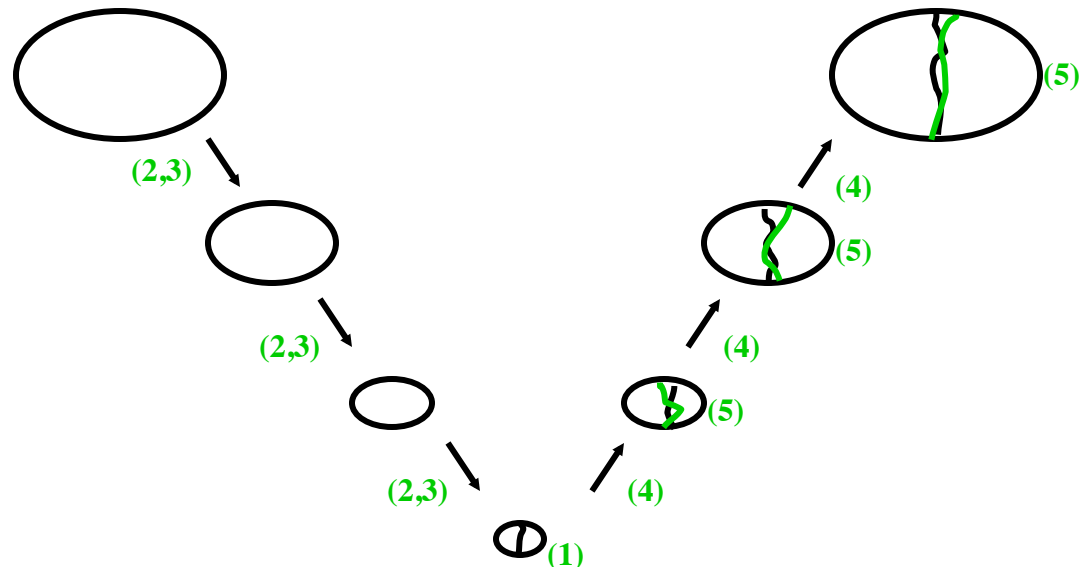
(5) Improve the partition $(N+, N-)$

Return $(N+, N-)$

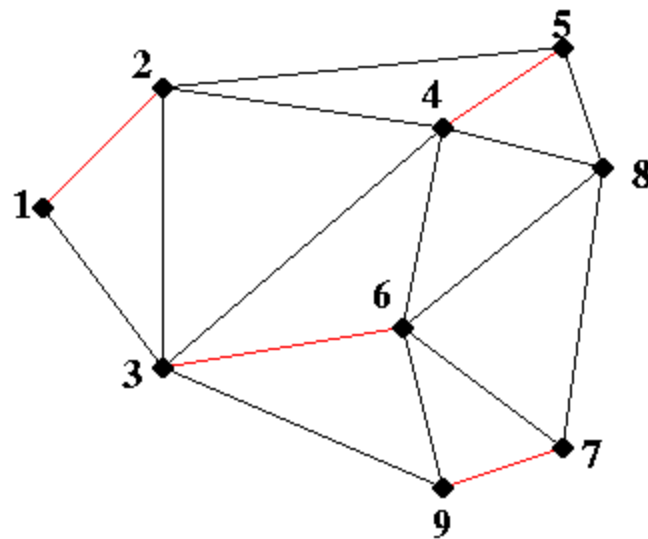
endif

“V - cycle:”

How do we
Coarsen?
Expand?
Improve?

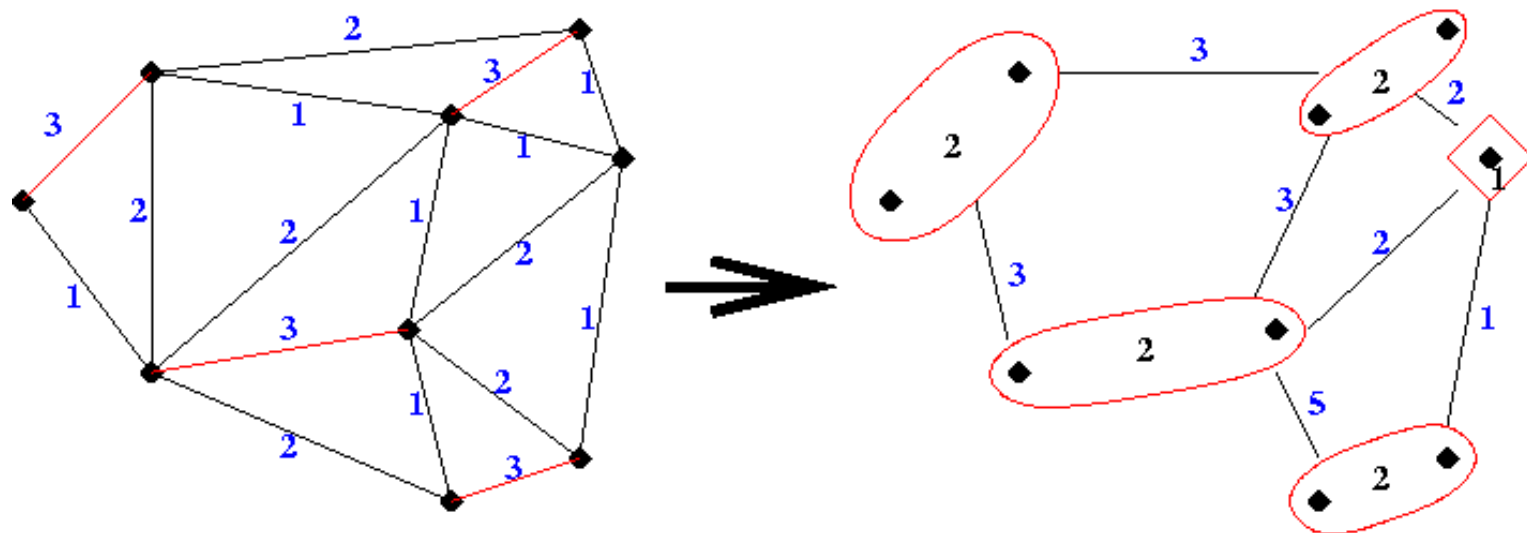


Coarsening by Maximal Matching



Example of Coarsening

How to coarsen a graph using a maximal matching



$$G = (N, E)$$

E_m is shown in red

Edge weights shown in blue

Node weights are all one

$$\mathbf{G}_{\mathbf{c}} = (\mathbf{N}_{\mathbf{c}}, \mathbf{E}_{\mathbf{c}})$$

N_c is shown in red

Edge weights shown in blue

Node weights shown in black

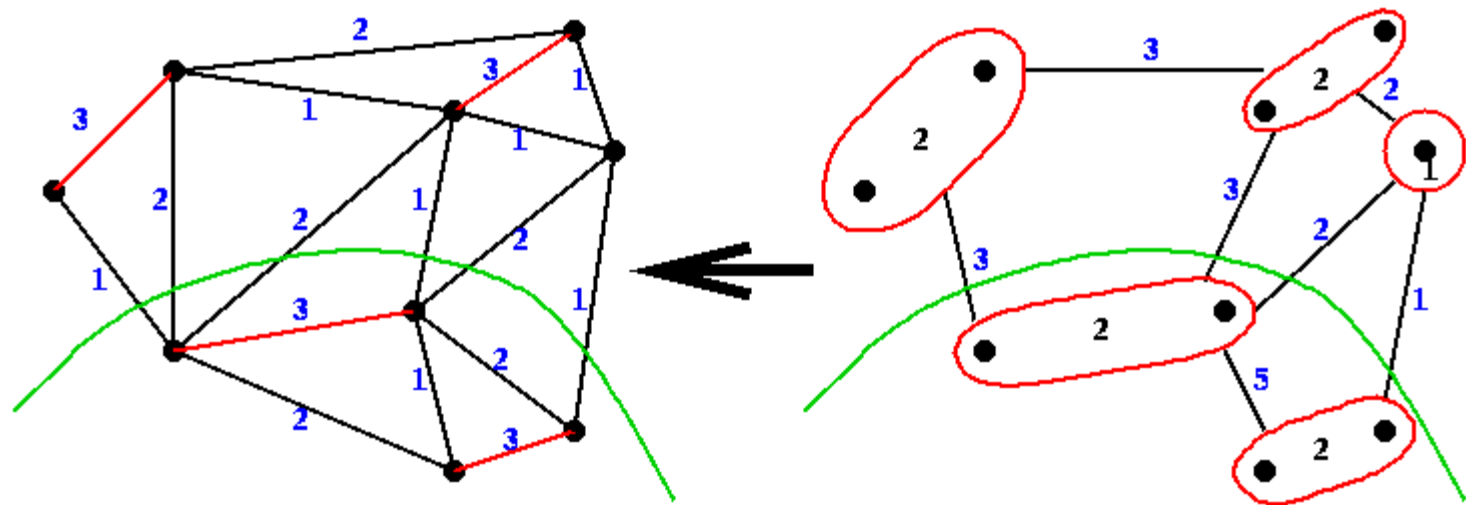
At bottom of recursion, Partition the coarsest graph

...Using spectral bisection, say.

$$\mathbf{G} = \begin{array}{ccccc} & 1 & & & 4 \\ & \bullet & & & \bullet \\ & | & \diagdown & & / \\ 2 & \bullet & & 3 & \bullet \\ & | & & & | \\ & \bullet & & & \bullet \\ & 2 & & 3 & 5 \end{array} \quad \mathbf{L}(\mathbf{G}) = \begin{pmatrix} 2 & -1 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ -1 & -1 & 4 & -1 & -1 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & -1 & 2 \end{pmatrix}$$

Expand a partition of G_C to a partition of G

Converting a coarse partition to a fine partition



Partition shown in green

*After each expansion, Improve the partition...
... by iterative swapping.*

