Nodal Coordinates: Inertial Partitioning

- For a graph in 2D, choose line with half the nodes on one side and half on the other
  - In 3D, choose a plane, but consider 2D for simplicity
- Determine a line L, and then choose a line $L^{\perp}$ perpendicular to it, with half the nodes on either side

1. **Determine a line L through the points**

   L given by a*(x-xbar)+b*(y-ybar)=0,

   with $a^2+b^2=1$; (a,b) is unit vector $\perp$ to L

2. **Project each point to the line**

   For each $n_j = (x_j, y_j)$, compute coordinate

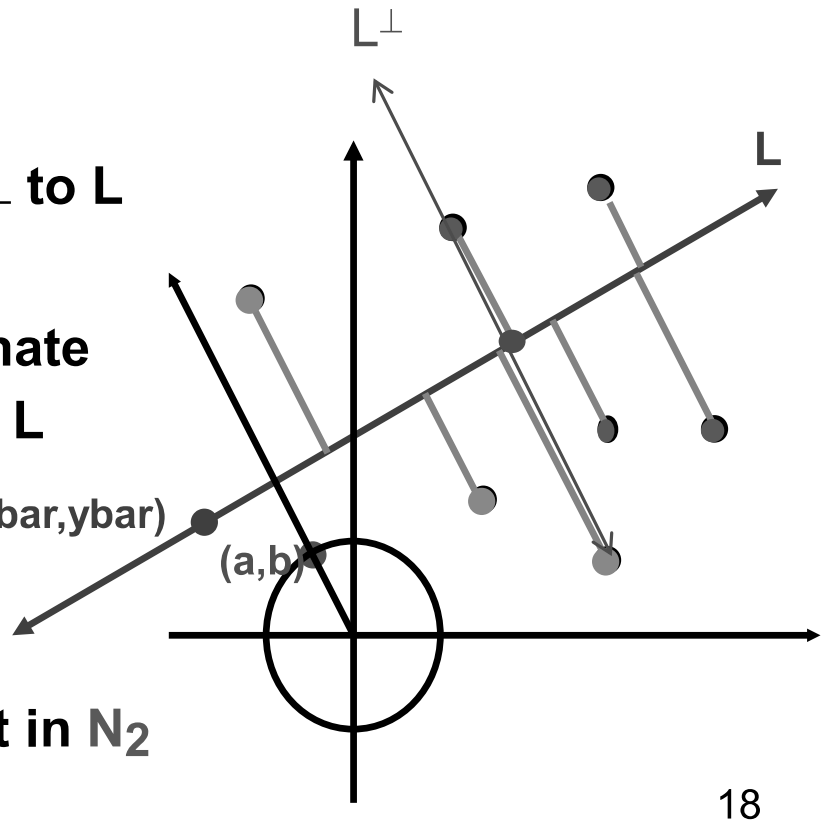
   $S_j = -b*(x_j-xbar) + a*(y_j-ybar)$ along L

3. **Compute the median**

   Let Sbar = median(S_1,…,S_n)

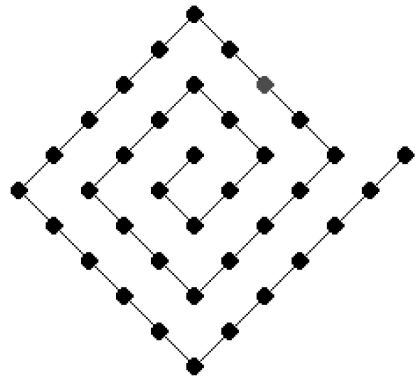
4. **Use median to partition the nodes**

   Let nodes with S_j < Sbar be in N_1, rest in N_2



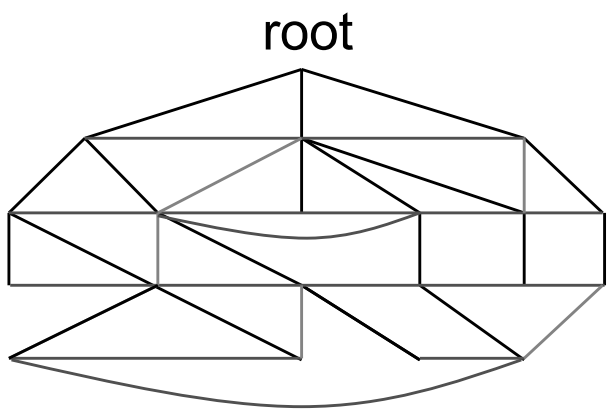
Nodal Coordinates: Summary

- Other variations on these algorithms
- Algorithms are efficient (i.e., fast)
- Rely on graphs having nodes connected (mostly) to "nearest neighbors" in space
  - algorithm does not depend on where actual edges are!
- Common when graph arises from physical model
- Ignores edges, but can be used as good starting guess for subsequent partitioning that does examine edges
- Can do poorly if graph connection is not spatial:



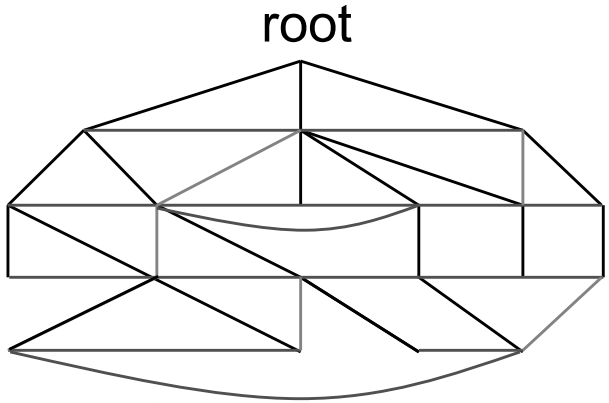
Breadth First Search (details)

- Queue (First In First Out, or FIFO)
  - Enqueue(x,Q) adds x to back of Q
  - x = Dequeue(Q) removes x from front of Q
- Compute Tree $T(N_T,E_T)$

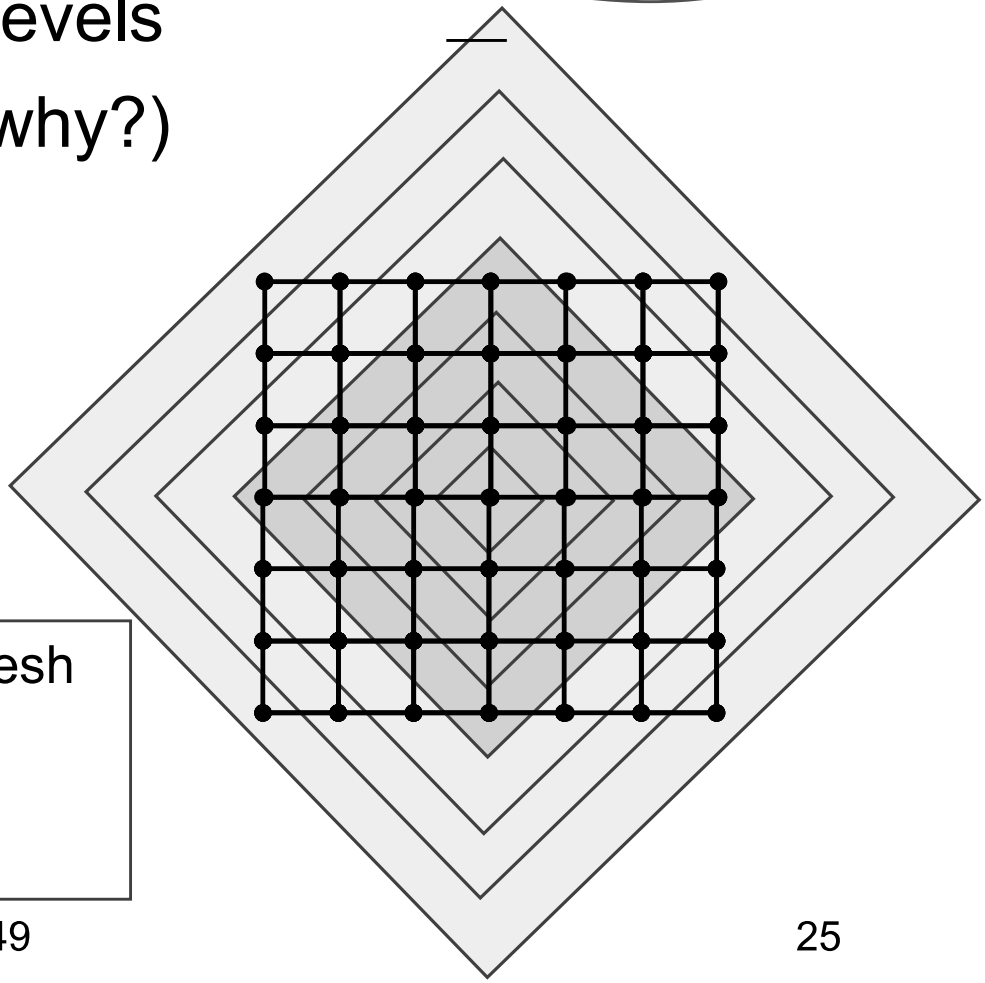


N_T = {(r,0)}, E_T = empty set … Initially T = root r, which is at level 0
Enqueue((r,0),Q) … Put root on initially empty Queue Q
Mark r … Mark root as having been processed
While Q not empty … While nodes remain to be processed
    (n,level) = Dequeue(Q) … Get a node to process
    For all unmarked children c of n
        $N_T = N_T$ U (c,level+1) … Add child c to N_T
        $E_T = E_T$ U (n,c) … Add edge (n,c) to E_T
        Enqueue((c,level+1),Q)) … Add child c to Q for processing
        Mark c … Mark c as processed
    Endfor
Endwhile

Partitioning via Breadth First Search



- BFS identifies 3 kinds of edges
  - Tree Edges - part of T
  - Horizontal Edges - connect nodes at same level
  - Interlevel Edges - connect nodes at adjacent levels
- No edges connect nodes in levels differing by more than 1 (why?)
- BFS partioning heuristic
  - N = N_1 U N_2, where
    - N_1 = {nodes at level <= L},
    - N_2 = {nodes at level > L}
  - Choose L so $|N_1|$ close to $|N_2|$



BFS partition of a 2D Mesh using center as root:
N1 = levels 0, 1, 2, 3
N2 = levels 4, 5, 6

Kernighan/Lin: Preliminary Definitions

- T = cost(A, B), newT = cost(newA, newB)
- Need an efficient formula for newT; will use
  - E(a) = external cost of a in A = Σ {W(a,b) for b in B}
  - I(a) = internal cost of a in A = Σ {W(a,a') for other a' in A}
  - D(a) = cost of a in A = E(a) - I(a)
  - E(b), I(b) and D(b) defined analogously for b in B
- Consider swapping X = {a} and Y = {b}
  - newA = (A - {a}) U {b}, newB = (B - {b}) U {a}
- newT = T - (D(a) + D(b) - 2*w(a,b)) ≡ T - gain(a,b)
  - gain(a,b) measures improvement gotten by swapping a and b
- Update formulas (cost changes only when (a',a) or (a',b) exist)
  - newD(a') = D(a') + 2*w(a',a) - 2*w(a',b) for a' in A, a' ≠ a
  - newD(b') = D(b') + 2*w(b',b) - 2*w(b',a) for b' in B, b' ≠ b

Kernighan/Lin Algorithm

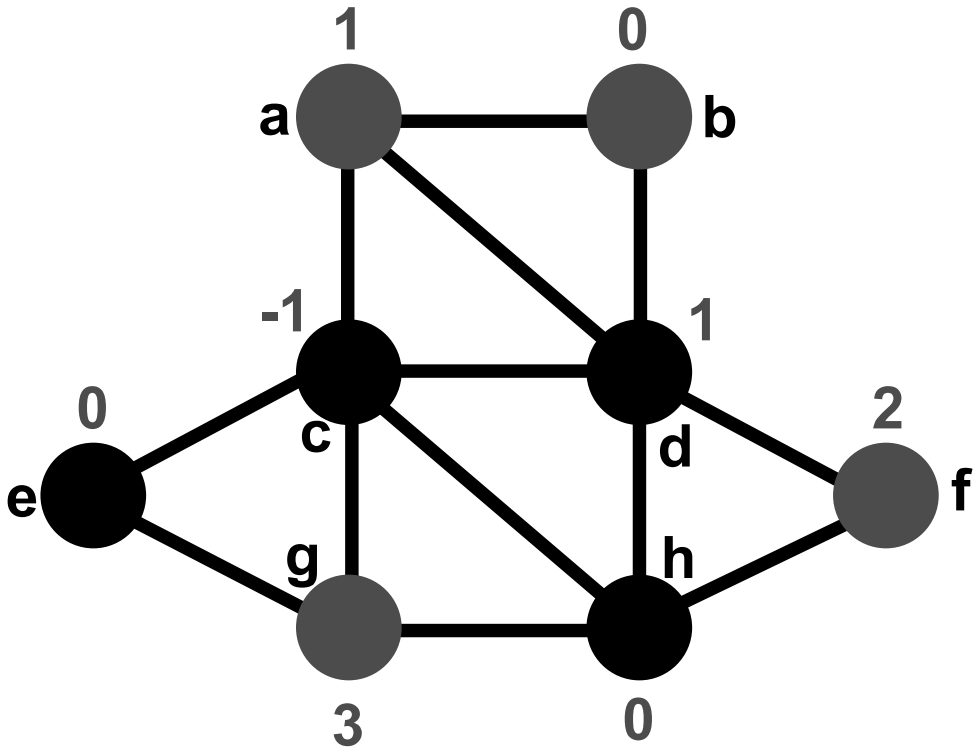
```
Compute  T = cost(A,B) for initial A, B                    … cost = O(|N|^2)
Repeat
    … One pass greedily computes |N|/2 possible X,Y to swap, picks best
    Compute costs D(n) for all n in N                      … cost = O(|N|^2)
    Unmark all nodes in N                                  … cost = O(|N|)
    While there are unmarked nodes                         … |N|/2 iterations
        Find an unmarked pair (a,b) maximizing gain(a,b)   … cost = O(|N|^2)
        Mark a and b (but do not swap them)                … cost = O(1)
        Update D(n) for all unmarked n,
            as though a and b had been swapped             … cost = O(|N|)
    Endwhile
        … At this point we have computed a sequence of pairs
        …  (a1,b1), … , (ak,bk)   and gains gain(1),…., gain(k)
        … where k = |N|/2, numbered in the order in which we marked them
    Pick m maximizing Gain = Σ_{k=1 to m} gain(k)          … cost = O(|N|)
        … Gain is reduction in cost from swapping (a1,b1) through (am,bm)
    If Gain > 0 then   … it is worth swapping
        Update newA = A - { a1,…,am } U { b1,…,bm }        … cost = O(|N|)
        Update newB = B - { b1,…,bm } U { a1,…,am }        … cost = O(|N|)
        Update T = T - Gain                                … cost = O(1)
    endif
Until Gain <= 0
```

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 by changing membership Part are shown in red.



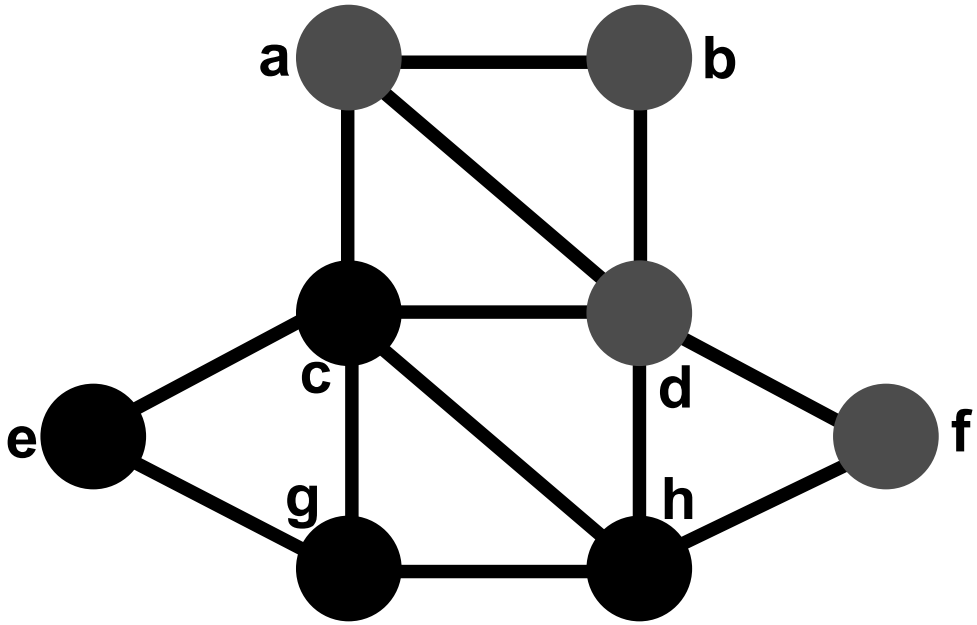
Nodes tentatively moved (and cut size after each pair):

none (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):

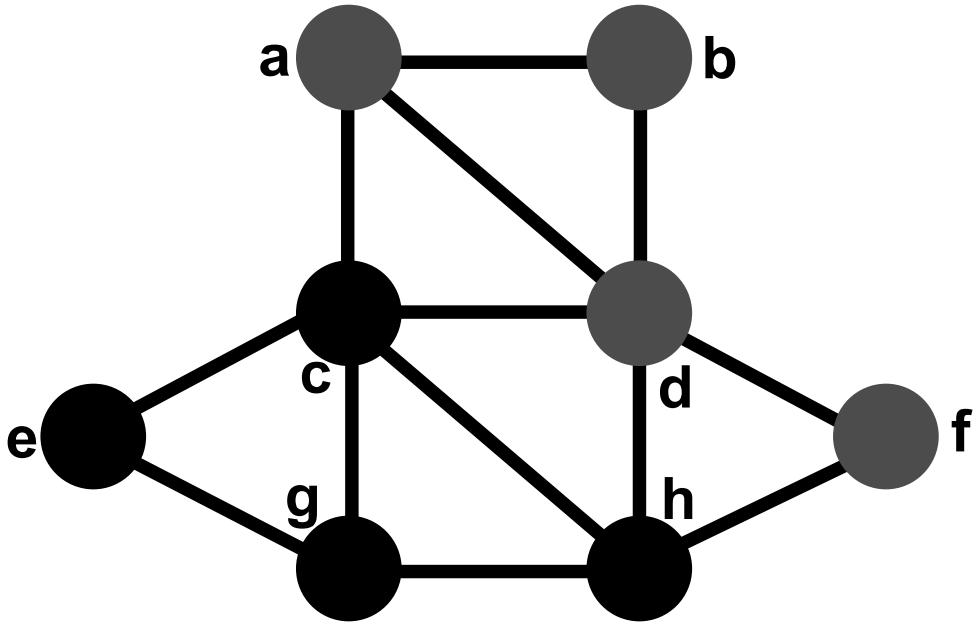
none (8); **g, d (4);** f, c (5); b, e (7); a, h (8)

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.



Basic Definitions

- *Definition*: The incidence matrix In(G) of a graph G(N,E) is an |N| by |E| matrix, with one row for each node and one column for each edge. For an edge e=(i,j), column e of In(G) is zero except for the i-th and j-th entries, which are +1 and -1, respectively.
- Slightly ambiguous definition because multiplying column e of In(G) by -1 still satisfies the definition, but this won't matter...
- *Definition*: The Laplacian matrix L(G) of a graph G(N,E) is an |N| by |N| symmetric matrix, with one row and column for each node. It is defined by
  - L(G) (i,i) = degree of node i (number of incident edges)
  - L(G) (i,j) = -1 if i ≠ j and there is an edge (i,j)
  - L(G) (i,j) = 0 otherwise

Properties of Laplacian Matrix

- *Theorem 1:* Given G, L(G) has the following properties
  (proof on 1996 CS267 web page)
  - L(G) is symmetric.
    - This means the eigenvalues of L(G) are real and its eigenvectors are real and orthogonal.
  - $In(G) * (In(G))^T = L(G)$
  - The eigenvalues of L(G) are nonnegative:
    - $0 = \lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$
  - The number of connected components of G is equal to the number of λ_i equal to 0.
  - *Definition*: $\lambda_2(L(G))$ is the algebraic connectivity of G
    - The magnitude of λ_2 measures connectivity
    - In particular, $\lambda_2 \neq 0$ if and only if G is connected.

Spectral Bisection Algorithm

- Spectral Bisection Algorithm:
  - Compute eigenvector v_2 corresponding to $\lambda_2(L(G))$
  - For each node n of G
    - if $v_2(n) < 0$ put node n in partition N-
    - else put node n in partition N+
- Why does this make sense? First reasons...
  - *Theorem 2 (Fiedler, 1975):* Let G be connected, and N- and N+ defined as above. Then N- is connected. If no $v_2(n) = 0$, then N+ is also connected.
  - Recall $\lambda_2(L(G))$ is the algebraic connectivity of G
  - *Theorem 3 (Fiedler):* Let $G_1(N,E_1)$ be a subgraph of G(N,E), so that G_1 is "less connected" than G. Then $\lambda_2(L(G_1)) \leq \lambda_2(L(G))$, i.e. the algebraic connectivity of G_1 is less than or equal to the algebraic connectivity of G.

Spectral Bisection Algorithm

- Spectral Bisection Algorithm:
  - Compute eigenvector v_2 corresponding to $\lambda_2(L(G))$
  - For each node n of G
    - if $v_2(n) < 0$ put node n in partition N-
    - else put node n in partition N+
- Why does this make sense? More reasons...
  - *Theorem 4 (Fiedler, 1975):* Let G be connected, and N1 and N2 be any partition into part of equal size |N|/2. Then the number of edges connecting N1 and N2 is at least .25 * |N| * $\lambda_2(L(G))$.

Computing v_2 and λ_2 of L(G) using Lanczos

- Given any n-by-n symmetric matrix A (such as L(G)) Lanczos computes a k-by-k "approximation" T by doing k matrix-vector products, k << n

```
Choose an arbitrary starting vector r
b(0) = ||r||
j=0
repeat
    j=j+1
    q(j) = r/b(j-1)          … scale a vector (BLAS1)
    r = A*q(j)               … matrix vector multiplication, the most expensive step
    r = r - b(j-1)*v(j-1)    … "axpy", or scalar*vector + vector (BLAS1)
   a(j) = v(j)^T * r         … dot product (BLAS1)
   r = r - a(j)*v(j)         … "axpy" (BLAS1)
   b(j) = ||r||              … compute vector norm (BLAS1)
until convergence            … details omitted
```

$$T = \begin{pmatrix} a(1) & b(1) & & & & \\ b(1) & a(2) & b(2) & & & \\ & b(2) & a(3) & b(3) & & \\ & & \ddots & \ddots & \ddots & \\ & & & b(k-2) & a(k-1) & b(k-1) \\ & & & & b(k-1) & a(k) \end{pmatrix}$$

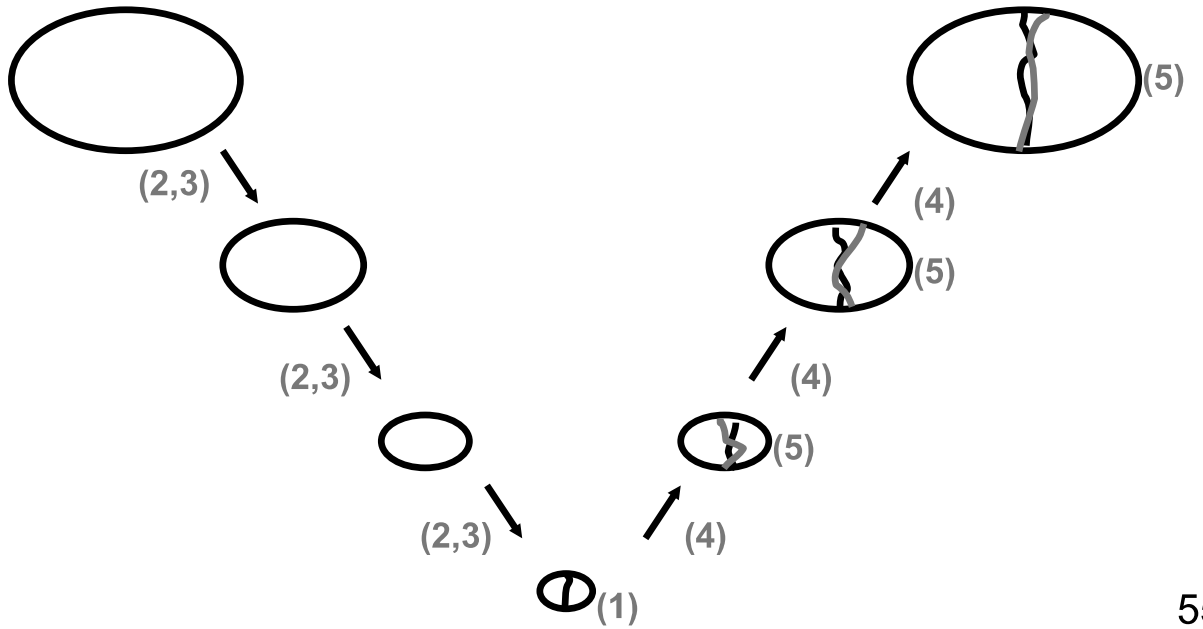
- Approximate A's eigenvalues/vectors using T's

Multilevel Partitioning - High Level Algorithm

```
(N+,N- ) = Multilevel_Partition( N, E )
        … recursive partitioning routine returns N+ and N- where N = N+ U N-
        if |N| is small
(1)         Partition G = (N,E)  directly to get N = N+ U N-
            Return (N+, N- )
        else
(2)         Coarsen G to get an approximation Gc = (Nc, Ec)
(3)         (Nc+ , Nc- ) = Multilevel_Partition( Nc, Ec )
(4)         Expand (Nc+ , Nc- ) to a partition  (N+ , N- ) of N
(5)         Improve the partition ( N+ , N- )
            Return ( N+ , N- )
        endif
```

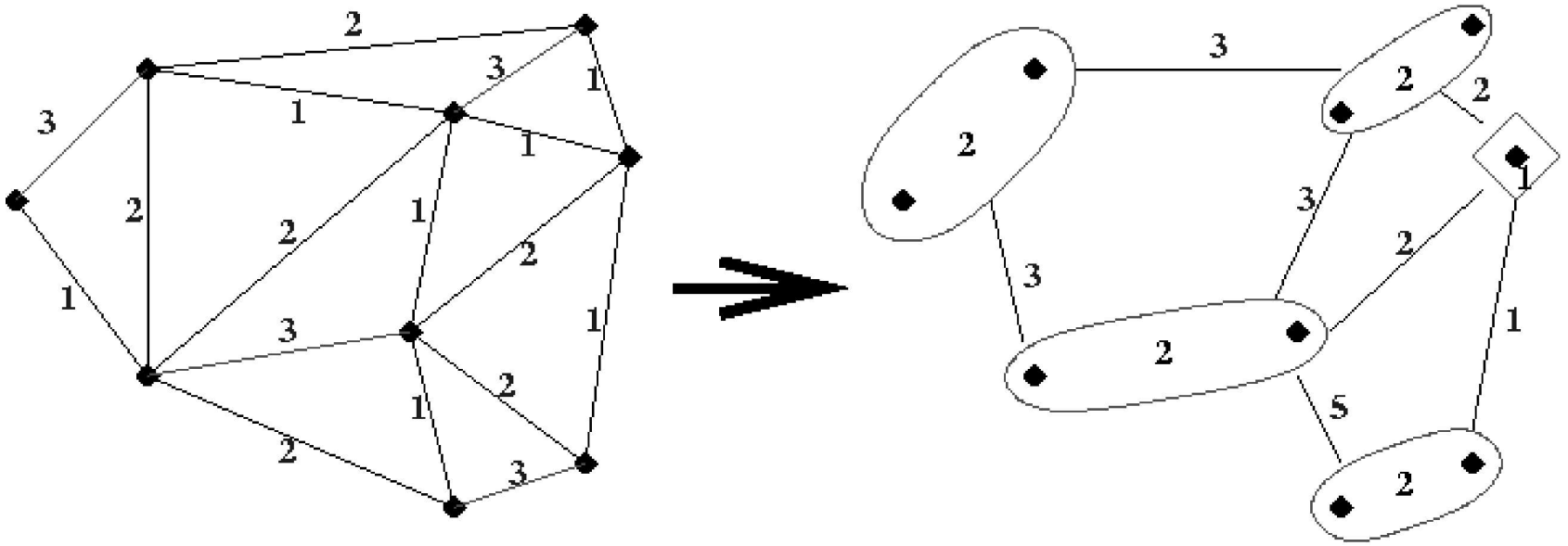
"V - cycle:"

How do we
- **Coarsen?**
- **Expand?**
- **Improve?**



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

$G_c = (N_c, E_c)$

N_c is shown in red

Edge weights shown in blue

Node weights shown in black

Expanding a partition of G_c to a partition of G