

Lesson 2-7 Graph Partitioning

The Graph Partitioning Problem

Look at the problem from a different angle:

Let's multiply a sparse matrix 'A' by a vector 'X'.

Recall the duality between matrices and graphs:

Rows and columns are vertices

The nonzeros are edges

One way to do a BFS is to do a computation that looks like a linear algebra problem:

$$y \leftarrow A \cdot x \quad (A \text{ multiplied by } x)$$

To distribute the work by rows:

Assign rows to processes. This is equivalent to partitioning the graph.

When you partition the matrix, this implies that you are also partitioning the vectors x and y . This occurs because there is a one-to-one mapping of vector entries to graph vertices.

Work = $O(\text{non-zero})$ = the work is proportional to the number of nonzeros.

(If n is the number of nonzeros in the row, then the depth of the computation is the depth of the sum, which is $O(\log n)$, and the work is the sum of the work across the elements, which is $O(n)$.)

How to choose partitions:

Goal 1: Divide up the rows to balance the number of nonzeros per partition.

Goal 2: Minimize the communication volume by reducing the edge cuts.

The classic graph partitioning problem:

Given: Graph $G = (V, E)$ & number of partitions P

Output: Compute a (vertex) partition

$$V = V_0 \cup V_1 \cup V_2 \cup \dots \cup V_{p-1}$$

such that:

1. The partitions should cover all the vertices, but be disjoint.

$$\{V_i\} \text{ are disjoint} \rightarrow V_i \cap V_j = \text{empty set}$$

2. The partitions should all be about equal in size.

$$\{V_i\} \text{ are roughly balanced} \rightarrow |V_i| \sim |V_j|$$

3. The number of cut edges should be minimized.

$$\text{Let } E_{\text{cut}} = \{(u,v) \mid u \in V_i, v \in V_j, i \neq j\}$$

$$\text{Minimize } |E_{\text{cut}}|$$

Do You Really Want a Graph Partition?

For a sparse matrix multiply:

Recall the two computation goals: balance work, minimize communication

Are the goals the same for graph partitioning and sparse matrix multiply?

No, they are not the same.

Consider a three way partitioning of a graph:

Each partition has 2 vertices each

There are nine cut edges

If we translate this graph partitioning to the matrix partition we see:

The matrix partitions are not equal with regards to the nonzeros. One partition has 10 and the other two have 7 nonzeros.

This means the vertex counts are the same, but the WORK is NOT.

Find the partition that minimizes the number of edge cuts and balances number of nonzeros. (24 nonzeros total)

Graph Bisection and Planar Separators

Graph Partitioning is NP-Complete so:

Need heuristics

Need to exploit structure

(NP Complete explanation: <https://www.mathsisfun.com/sets/np-complete.html>)

(A divide and conquer [algorithm](#) works by recursively breaking down a problem into two or more sub-problems of the same (or related) type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.)

A Simple Heuristic: Bisection (based on divide and conquer)

Give a graph G , divide it into P partitions.

Step 1: divide the graph into two partitions

Step 2: Divide each half into two

Step 3: Continue to divide each partition into half until P partitions are reached

This works, but how do we get two way partitions? TBD

Planar Graphs

Planar graphs are ones that can be drawn in the plane with no edge crossing

Planar Graph Theorem:

Lipton and Tarjan Theorem: A planar graph $G = (V, E)$ with $|V| = n$ vertices has a disjoint

partition $V = A \cup S \cup B$ such that:

1. S separates A and B (this means there are no edges that directly connect A and B)
2. $|A|, |B| \leq 2/3n$ this means that no partition is more than twice the size of the other $|A|/|B| \leq 2$. This means the partitions are balanced.
3. $|S| = O(\sqrt{n})$.

In a planar graph s might be the a row or column.

The existence of S DOES NOT mean you can minimize the edge cuts efficiently. But any algorithm that can find the separator should be able to find a good partition.

(Breadth First Search:

1. start at some point, give it two values (distance, predecessor). Distance is its distance from the starting point, predecessor on the shortest path from the starting point)
2. Then visit the next level, those with distance 1, and set the (distance, predecessor) for those vertex.
3. Continue until all vertices have been visited.

https://en.wikipedia.org/wiki/Breadth-first_search)

Quiz: Partitioning via Breadth-First Search

An algorithm the uses BFS to bisect a graph:

1. Pick any vertex as a starting point
2. Run a level synchronous BFS from this vertex
3. You will notice that every level serves as a separator.
4. Because of this, you can stop when you have visited about half of the vertices.
5. Assign all visited vertices to one partition and all unvisited vertices to the other partition.

This is not the only option for a stopping criteria - can you come up with others?

BFS schemes work well on planar graphs and they are cheap - but we are using BFS to solve a BFS problem.

Kernighan-Lin

Kernighan-Lin algorithm is the most famous heuristic for graph partitioning.

Given a graph, divide the vertices into equal or nearly equal size. Any split will work.

$$V \equiv V_1 \cup V_2, |V_1| = |V_2|$$

Define cost to be: the number of edges that go between V_1 and V_2

Now take a subset of V_1 and V_2 , call them X_1 and X_2

let $X_1 \subseteq V_1$ and $X_2 \subseteq V_2$, with $|X_1| = |X_2|$

Now change the partitions so that $X_2 \subseteq V_1$ and $X_1 \subseteq V_2$.

You expect the cut size to change, but by how much?

To answer this question:

Pick a vertex V_1 in partition 1 and a vertex V_2 in partition 2.

The external costs are:

(The edges that cross the partitions)

$$E_1(a \in V_1) \equiv \# \text{ of edges } (a, b \in V_2)$$

$$E_2(b \in V_2) \equiv \# \text{ of edges } (b, a \in V_1)$$

The internal costs are:

(The edges that DO NOT cross partitions)

$$I_1(a \in V_1) \equiv \# \text{ of edges } (a, i \in V_1)$$

$$I_2(b \in V_2) \equiv \# \text{ of edges } (b, j \in V_2)$$

The Cost of the Partition: is the cost of the partition ignoring a and b plus the external cost of a and b minus some constant.

$$\text{Cost}(V_1, V_2) = \text{Cost}(V_1 - \{a\}, V_2 - \{b\}) + E_1(a) + E_2(b) - c_{a,b}$$

The constant is necessary to account for an edge between a and b.

$$c_{a,b} = 1 : \text{ if there is an edge}$$

$$c_{a,b} = 0 : \text{ if there is no edge}$$

BUT why subtract this constant?

But now swap a and b. What is the cost of the swap?

1. Any edge that was external is now internal
2. Any edge that was internal is now external

$$\text{Cost}(V_1^{\wedge}, V_2^{\wedge}) = \text{Cost}(V_1 - \{a\}, V_2 - \{b\}) + I_1(a) + I_2(b) + c_{a,b}$$

What is the change in cost?

$$\text{gain}(a \in V_1, b \in V_2) = \text{Cost}(V_1, V_2) - \text{Cost}(V_1^{\wedge}, V_2^{\wedge}) = E_1(a) + E_2(b) - (I_1(a) + I_2(b)) - 2c_{a,b}$$

The larger the change in cost the better because this means a larger decrease in the cost.

Kernighan-Lin Algorithm Quiz

Assume:

Every vertex has a partition label, the label can be accessed in constant time $O(1)$

The maximum degree of any vertex is given. Max degree $\equiv d$

Question: What is the sequential running time to compute $\text{gain}(a,b)$ in terms of

$$d, n_1 = |V_1|, n_2 = |V_2| ?$$

Answer: $O(d)$

To get the answer:

Sweep over the neighbors, there will be at most d neighbors. To determine if a neighbor is internal or external, check its partition label.

Kernighan-Lin Algorithm

$G \equiv (V,E)$

A graph is partitioned. To improve the partition, try swapping the elements, X_1, X_2 .

How are X_1, X_2 chosen?

The K-L Procedure

1. Compute the internal and external cost for every vertex.
2. Mark all the nodes as unvisited.
3. Then carry out an iterative procedure:

Go through every pair of unmarked vertices. Choose the pair with the largest gain and mark that pair as visited.

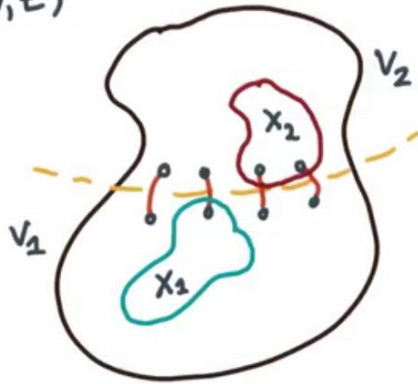
Go through and update every internal and external costs as if a and b had been swapped. You are not swapping a and b , just updating the costs.

Repeat until all the vertices are visited.

The algorithm:

Kernighan-Lin Algorithm

$$G \equiv (V, E)$$



$$\text{let } V \equiv V_1 \cup V_2$$

$$C \equiv \text{cost}(V_1, V_2)$$

forall $a \in V_1$ do compute $E_1(a), I_1(a)$

forall $b \in V_2$ do compute $E_2(b), I_2(b)$

forall $v \in V$ do $\text{visited}[v] \leftarrow \text{false}$

while \exists unvisited vertices do

Choose unmarked ($a \in V_1, b \in V_2$)
with largest **gain** (a, b)

$\text{visited}[a], \text{visited}[b] \leftarrow \text{true}$

Update all $E_1(\cdot), E_2(\cdot), I_1(\cdot), I_2(\cdot)$

At the completion of the algorithm: a sequence of gains has been computed:

$$\text{gain}(a_1, b_1), \text{gain}(a_2, b_2), \dots,$$

This is the end of the first part of the algorithm.

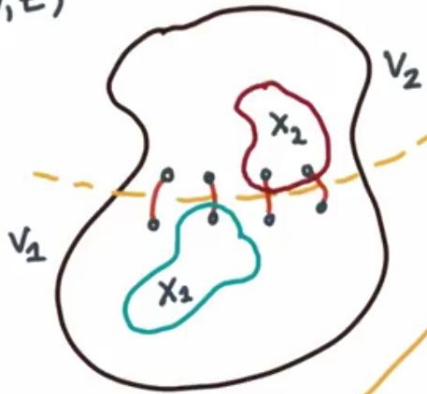
Now: Sum all the gains... let $\text{Gain}(j) \equiv \sum_{k=1}^j \text{gain}(a_k, b_k)$

Kernighan-Lin concept: keep all the swaps that maximize the gain. If the Gain is greater than zero, then this is candidate will improve the partition. Swap the two subsets and update the overall costs.

Repeat the above until there is no more Gain.

Kernighan-Lin Algorithm

$$G \equiv (V, E)$$



Repeat until no more Gain

$$\text{let } \underline{\text{Gain}}(j) \equiv \sum_{k=1}^j \text{gain}(a_k, b_k)$$

$$\text{Choose } j_{\max} \equiv \text{argmax}_j \text{Gain}(j)$$

if Gain(j_{\max}) > 0 then

$$X_1 \equiv \{a_1, a_2, \dots, a_{j_{\max}}\}$$

$$X_2 \equiv \{b_1, b_2, \dots, b_{j_{\max}}\}$$

$$\text{Update } C \leftarrow C - \underline{\text{Gain}}(j_{\max})$$

$$V_1 \leftarrow (V_1 - X_1) \cup X_2 \quad \left. \vphantom{V_1} \right\} \text{swap!}$$

$$V_2 \leftarrow (V_2 - X_2) \cup X_1$$

The main concern with this algorithm is the cost. The sequential running time is $O(|V|^2 d)$
 d is the maximum degree of any vertex.

Graph Coarsening

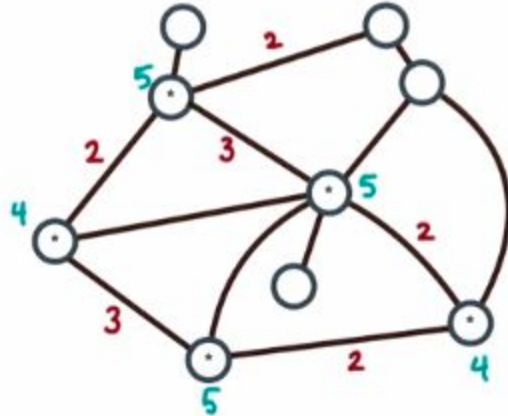
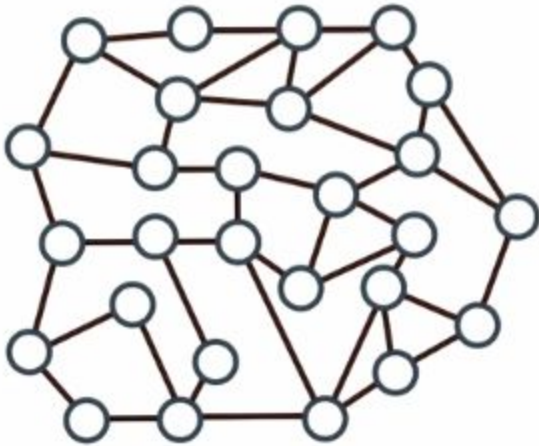
Graph coarsening is a different kind of graph partitioning, it is a form of divide and conquer. The goal of graph coarsening: take a graph, coarsen it so it looks similar to the original graph but with fewer nodes. Do this until you achieve a graph that is small enough to partition easily.

How to Coarsen a Graph:

1. Identify one subset of the vertices to collapse or merge
2. Replace the subset with a single super vertex.
3. Assign a weight to the super vertex that is equal to the number of vertices it replaced.
4. Assign a weight to the edges.

Example:

Initial graph to final result:



Maximal and Maximum Matchings

To coarsen a graph effectively, a scheme is necessary to determine which vertices to combine.

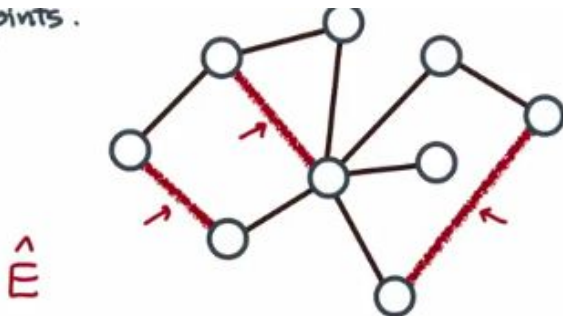
One idea: compute a matching

Matching: a matching of a graph $G = (V, E)$ is a subset of $E \wedge \subseteq E$ of with no common endpoints.

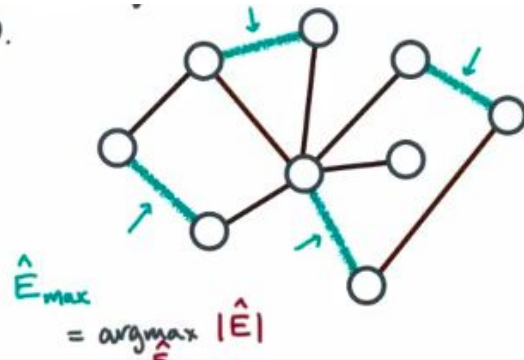
In the example below, the three edges are a matching because they don't share any endpoints. It is also a maximal matching because no more edges can be added to it.

A maximum matching is one that has the most number of matches. This graph has a matching that has more than three edges in it. The green matching is a maximum matching.

011115.



).



A Fact about Maximal Matchings

Given a graph with n vertices, you coarsen it k times so that it has s vertices.
How large must k be in terms of n and s ?

Answer: $\log_2(n/s)$

Why:

1. Imagine that there is a maximal that will match every vertex (meaning every vertex is part of a matched edge). This will result in a coarsened graph that has $\frac{1}{2}$ the number of vertices.
2. The k version of the graph must have $\frac{1}{2}$ as many vertices as the previous level.
3. Every level has to follow this pattern.
4. The final graph must have $n/(2^k)$ vertices.
5. This means $k \geq \log_2(n/s)$

Can you think of a worst case graph to coarsen?

Computing a Maximal Matching

At each stage of this scheme chose any unmatched vertex at random.

1. Pick any vertex
2. Match it to any of its unmatched neighbors.
3. The neighbor you want to chose is the one with the highest edge weight. The reason to do this is it will decrease the overall weight in the next level of the graph.

Fine to Coarse and Back Again Quiz

Projected Separator: the vertices and edges that will be combined in the next level of the graph.
There can be a situation where the next level graph maps ambiguously to the previous level.

Partition Refinement

A minimum balanced edge cut in a coarsened graph minimizes the balanced edge cut in the next finer graph. This is false. You need to remember that coarsening is based on a heuristic.
What if the coarsened graph had been based on a maximum rather than maximal matching?

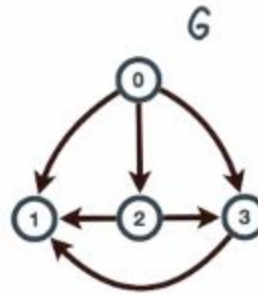
Spectral Partitioning Part 1: The Graph Laplacian

Consider an unweighted directed graph, G .

When represented by an incidence matrix, each row is an edge and each column is a vertex. Put a 1 at the source, and a -1 at the destination.

$C = C(G)$: Incidence matrix

	0	1	2	3	
0	1	-1			$0 \rightarrow 1$
1	1		-1		$0 \rightarrow 2$
2	1			-1	$0 \rightarrow 3$
3		-1	1		$2 \rightarrow 1$
4			1	-1	$2 \rightarrow 3$
5		-1		1	$3 \rightarrow 1$



Graph Laplacian $L(G) \equiv C^T C$

$L(G)$ will look like this ...

let $C \equiv \begin{bmatrix} e_0^T \\ e_1^T \\ \vdots \\ e_{m-1}^T \end{bmatrix}$

edge $(i, j) \rightarrow e_k$
consider $e_k e_k^T$

$$C^T C = \sum_{k=0}^{m-1} e_k e_k^T$$

i $\begin{bmatrix} \vdots \\ +1 \\ \vdots \\ -1 \\ \vdots \end{bmatrix}$ $\begin{bmatrix} \vdots \\ \dots +1 \dots -1 \dots \\ \vdots \end{bmatrix}$

The diagonals are tallying the number of incident edges on each vertex. They count the degree of each vertex. (D)

The off-diagonals will indicate the presence of an edge. (W)

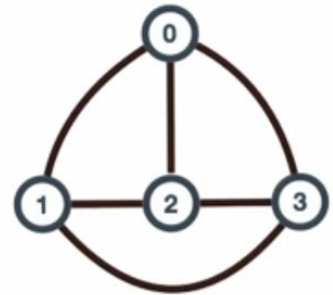
$$C^T C = D - W$$

The Graph Laplacian Example:

$$D \equiv \begin{bmatrix} 3 & & & \\ & 3 & & \\ & & 3 & \\ & & & 3 \end{bmatrix}$$

$$W \equiv \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

$$L(G) = D - W = \begin{bmatrix} 3 & -1 & -1 & -1 \\ -1 & 3 & -1 & -1 \\ -1 & -1 & 3 & -1 \\ -1 & -1 & -1 & 3 \end{bmatrix}$$



The $L(G)$ should be symmetric about the diagonal and every row sums to 0.

Spectral Partitioning, Part 3: Algebraic Connectivity

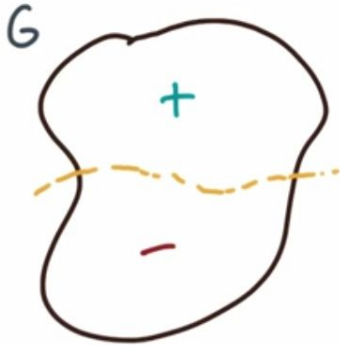
Handy Facts:

1. $L(G)$ is symmetric
2. $L(G)$ has real-valued, non-negative eigenvalues and real-valued, orthogonal eigenvectors.
Multiplying $L(G)$ by its eigenvector will give a scaled eigenvector. The scaling factor is the eigenvalue.
Orthogonal \rightarrow the dot product of any pair of eigenvectors will be 0 if they are different and 1 if they are the same.
3. G has k connected components if and only if k 's smallest eigenvalues are identical to 0.
4. The number of cut edges in a partition is: $\frac{1}{4} x^T L(G)x$. So if you want to minimize edge cuts, minimize the product.

Counting Edge Cuts

Summing the degrees of all the vertices is the same as counting the number of edges, twice.

Quiz! Counting Edge Cuts



Your task:
Fill in the boxes!

$$x^T L(G) x = \sum_{i,j} l_{ij} x_i x_j$$

$$= \sum_{i=j} l_{ij} x_i x_j \leftarrow = l_{ii} x_i^2$$

d_i +1

$$+ \sum_{\substack{i,j \in V_+ \\ i \neq j}} l_{ij} x_i x_j + \sum_{\substack{i,j \in V_- \\ i \neq j}} l_{ij} x_i x_j$$

-1 1 1 -1 1 1

$$+ \sum_{\substack{i \in V_+ \\ j \in V_-}} l_{ij} x_i x_j + \sum_{\substack{i \in V_- \\ j \in V_+}} l_{ij} x_i x_j$$

-1 1 -1 -1 -1 1

The first sum is the number of edges wholly contained in V_+ , it counts them twice and is negative. $(-2 * \# \text{ of edges in } V_+)$

The total is basically 4 times the number of cut edges.

Spectral Partitioning, Part 4: Putting it all Together

Start with a graph G

Construct its Laplacian $L(G)$

Now suppose there is a partition of G , $V = V_+ \cup V_-$

The vertices are separated in the two sections

Each vertex is assigned to one partition or the other

The cut edges can be found, and the number of them can even be minimized

The partition should be the following rules:

Every partition must be in one partition or the other

Assign +1 or -1 to each vertex

The partitions must have the same number of vertices

The problem is NP-Complete.

Therefore, as a work around,

Remove the requirement that each vertex must be assigned +1 or -1

Now we can say:

The second smallest eigenvector is proportional to the minimum value of x


The Algorithm for Spectral Partitioning

1. Create $L(G)$
2. Compute the second smallest eigenpair of $L(G)$
3. Determine the partition using the signs of the eigenpair

Spectral Partitioning Algorithm:

- 1) Create $L(G)$
- 2) Compute (λ_1, \vec{q}_1) eigenpair of $L(G)$
- 3) Choose

$$\chi(i) \leftarrow \text{sign}(\vec{q}_1(i))$$



± 1