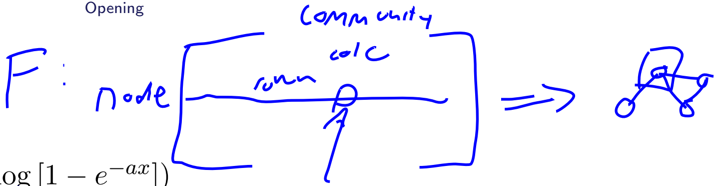
CSCI 4022 Fall 2021
Community Detection

Relevant **Calculus**: Compute $\frac{d}{dx}\left(\log\left[1-e^{-ax}\right]\right)$

AY



GOAL: push this element closer to the correct value

$$= \frac{1}{1-e^{-ax}} \frac{d}{dx}\left(1-e^{-ax}\right)$$

$$= \frac{1}{1-e^{-ax}}\left(a e^{-ax}\right)$$

$$= a\left[\frac{e^{-ax}}{1-e^{-ax}}\right] = -a\left[-1 + \boxed{\frac{1-e^{-ax}}{1-e^{-ax}}}\right]$$

- if many neighbors of u are in C, increase F(u,c).
- if most/many members of C are not neighbors of u, ⇒ − F(u,c)

CSCI 4022 Fall 2021
Community Detection

Relevant **Calculus**: Compute $\frac{d}{dx}\left(\log\left[1-e^{-ax}\right]\right)$

$$
\begin{aligned}
\frac{d}{dx}\left(\log\left[1-e^{-ax}\right]\right) &= \frac{1}{1-e^{-ax}}\left(ae^{-ax}\right) \\
&= a\frac{e^{-ax}}{1-e^{-ax}} \\
&= -a\frac{-e^{-ax}+1-1}{1-e^{-ax}} \\
&= -a\left(\frac{1}{1-e^{-ax}}-1\right)
\end{aligned}
$$

Via 2-Chainz rule

Then optional to rewrite as:

BigClam: Model review

$F_{u,A}$: the membership strength of node u in community a. Set $F_{u,a} = 0$ if and only if u is absolutely **not** a member of A.

Each community creates an edge between two of its members via
$P_A(u,v) = 1 - \exp\left(-F_{u,A} \times F_{v,A}\right).$

Result: if members share *multiple* communities, we get

$$P(u,v) = 1 - \exp\left(\sum_c -F_{\mathbf{u},c} \times F_{\mathbf{v},c}\right) = 1 - e^{-F_u \cdot F_v}$$

BigCLAM Implementation

Since every edge between members is created independently, the *joint* probability of a given graph: the set of all the edges E **and** the non-edges, given F, is:

$$L(F) = \prod_{(u,v)\in E} P(u,v) \prod_{(u,v)\notin E} (1 - P(u,v))$$

no edges

This is a big product and numerically unstable, so we logarithm...

$$l(F) = \sum_{(u,v)\in E} \log\left(1 - \exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})\right) - \underbrace{\sum_{(u,v)\notin E} F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}}}_{\log e^{-F_u \cdot F_v}}$$

and differentiate. The multi-variate derivative or *gradient* tells us which direction to **update** our iterative guesses of F and bring them closer to the maximum.

Gradient Ascent/Descent

Gradient Ascent is an algorithm of the form:

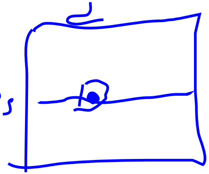
$$\underbrace{F^{(k+1)}}_{\text{new guess}} = \underbrace{F^{(k)}}_{\text{old guess}} + \underbrace{\nu}_{\text{step size}} \underbrace{F'(z^{(k)})}_{\text{step direction}}$$

In our case, we're going to *update* our estimates for F by taking small steps in the direction of the community affiliations for node u. In other words: A step is an update to u to be more closely affiliated with it's neighbors. Then repeat for *every* node u.

The *gradient* is the multivariate direction we're supposed to take steps in! We step "up" the slope or "down" the slope depending on whether we want a *max* or a *min*.

Gradient Ascent/Descent

F nodes



In practice, we're differentiating

$$l(F) = \sum_{\substack{(u,v)\in E \\ \text{edges}}} \log\left(1-\exp(-F_{\boldsymbol{u}}\cdot F_{\boldsymbol{v}})\right) - \sum_{\substack{(u,v)\notin E \\ \text{non-edges}}} F_{\boldsymbol{u}}\cdot F_{\boldsymbol{v}}$$

but we'll go at it *one specific node* at a time, so we're looking at

$$l(F_u) = \sum_{\substack{v\in N(u) \\ \text{neighbors}}} \log\left(1-\exp(-F_{\boldsymbol{u}}\cdot F_{\boldsymbol{v}})\right) - \sum_{\substack{v\notin N(u) \\ \text{not}}} F_{\boldsymbol{u}}\cdot F_{\boldsymbol{v}}$$

and differentiating with respect to row u
(In other words "how should we update our knowledge of person u").

Calculus friends: $\frac{d}{dx}\log(1-f(x)) =$,
$\frac{d}{dx}e^{f(x)} =$

Gradient Ascent/Descent

In practice, we're differentiating

$$l(F) = \sum_{(u,v)\in E} \log\left(1 - \exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})\right) - \sum_{(u,v)\notin E} F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}}$$

but we'll go at it *one specific node* at a time, so we're looking at

$$l(F_u) = \sum_{v\in N(u)} \log\left(1 - \exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})\right) - \sum_{v\notin N(u)} F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}}$$

and differentiating with respect to row u
(In other words "how should we update our knowledge of person u").
Calculus friends: $\frac{d}{dx}\log(1 - f(x)) = \frac{f'(x)}{1-f(x)}$,
$\frac{d}{dx}e^{f(x)} = f'(x)e^{f(x)}$

The BigCLAM gradient

$$\nabla l(F_u) = \frac{d}{dF_{\boldsymbol{u}}} \sum_{v \in N(u)} \log\left(1 - \exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})\right) - \sum_{v \notin N(u)} F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}}$$

$\frac{d}{dx}\left(\ln(1 - e^{-ax})\right) = a\left[\frac{e^{-ax}}{1-e^{-ax}}\right]$

Each term in the first sum is a derivative of $\log\left(1 - \exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})\right)$, which gives $F_v \frac{\exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})}{1 - \exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})}$.

Each term in the second sum is a derivative of $F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}}$, so we are left with just $F_{\boldsymbol{v}}$.

neighbor score

degree to move in that direction

Result:

$$\nabla l(F_u) = \left\langle \underbrace{\sum_{v \in N(u)} F_{v,A} \frac{\exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})}{1 - \exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})} - \sum_{v \notin N(u)} F_{\boldsymbol{v},A}}_{\nabla_A l(F_u)}, \ldots \right\rangle$$

u, B

A. B C, ...

The BigCLAM gradient

The full update:

positive

negative

$u \; \boxed{F_{u,A}, F_{u,B}, \dots}$

$$\nabla l(F_u) = \langle \overbrace{\sum_{v \in N(u)} F_{v,A} \frac{\exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})}{1 - \exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})}}^{\text{positive}} - \overbrace{\sum_{v \notin N(u)} F_{\boldsymbol{v},A}}^{\text{negative}}, \quad A$$

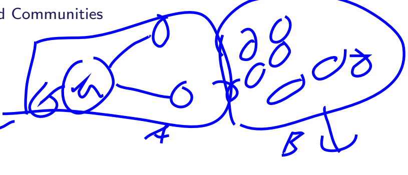
$$\sum_{v \in N(u)} F_{v,B} \frac{\exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})}{1 - \exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})} - \sum_{v \notin N(u)} F_{\boldsymbol{v},B}, \quad B$$

$$\sum_{v \in N(u)} F_{v,C} \frac{\exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})}{1 - \exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})} - \sum_{v \notin N(u)} F_{\boldsymbol{v},C}, \quad C$$

$$\dots, \rangle$$

Or: *for each community*, the corresponding entry to the vector $\nabla l(F_u)$ is the one that pushes u closer to the communities in it's neighbor set $N(u)$ and further from the communities not in its neighbor set.

The BigCLAM Iteration



each node is in A or

The full update:

$$\nabla l(F_u) = \langle \sum_{v \in N(u)} F_{v,A} \frac{\exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})}{1 - \exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})} - \sum_{v \notin N(u)} F_{\boldsymbol{v},A},$$

$$\sum_{v \in N(u)} F_{v,B} \frac{\exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})}{1 - \exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})} - \sum_{v \notin N(u)} F_{\boldsymbol{v},B}, \ldots, \rangle$$

all nodes = $N(u) \cup \neg N(u)$

Or **Iterate:**

1. Compute gradient of $l(F)$ with respect to (vector) Fu: $\nabla l(F_u)$ (keeping others fixed)
2. Update the row F_u as: $F_u^{new} = F_u^{old} + \nu \cdot \nabla l(F_u)$. ($\nu$ is a step size (usually small))
3. If any component c of F_u is negative $(F_{u,c} < 0)$, reset $F_{u,c} = 0$. (*Reflect:* why might this happen?)

The BigCLAM Iteration

1. Compute gradient of $l(F)$ with respect to (vector) Fu: $\nabla l(F_u)$ (keeping others fixed)
2. Update the row F_u as: $F_u^{new} = F_u^{old} + \nu \cdot \nabla l(F_u)$.
3. If any component c of F_u is negative $(F_{u,c} < 0)$, reset $F_{u,c} = 0$.

As written, this happens to be pretty slow! We can spruce it up a little, though! The steps in vector shorthand:

$$\nabla l(F_u) = \sum_{v \in N(u)} F_v \frac{\exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})}{1 - \exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})} - \boxed{\sum_{v \notin N(u)} F_{\boldsymbol{v}}}$$

Cleanup: F is sparse, since $N(u)$ is usually much smaller than all nodes. This means most of the additions are in the $\sum_{v \notin N(u)}$ sum. But we could rewrite:

all nodes — neighbors

non-neighbors

$$\sum_{v \notin N(u)} F_{\boldsymbol{v}} = \boxed{\sum_{v} F_{\boldsymbol{v}}} - F_u - \sum_{v \in N(u)} F_{\boldsymbol{v}}$$

— itself

The BigCLAM

$$\nabla l(F_u) = \sum_{v \in N(u)} F_v \frac{\exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})}{1 - \exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})} - \boxed{\sum_{v \notin N(u)} F_{\boldsymbol{v}}}$$

$$= \sum_{v \in N(u)} F_v \frac{\exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})}{1 - \exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})} - \left(\sum_{v} F_{\boldsymbol{v}} - F_u - \sum_{v \in N(u)} F_{\boldsymbol{v}} \right)$$

The BigCLAM

$$\nabla l(F_u) = \sum_{v \in N(u)} F_v \frac{\exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})}{1 - \exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})} - \sum_{v \notin N(u)} F_{\boldsymbol{v}}$$

$$= \sum_{v \in N(u)} F_v \frac{\exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})}{1 - \exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})} - \left(\sum_v F_{\boldsymbol{v}} - F_u - \sum_{v \in N(u)} F_{\boldsymbol{v}} \right)$$

$$= \sum_{v \in N(u)} F_v \left(\frac{\exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})}{1 - \exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})} + 1 \right) + F_u - \sum_v F_{\boldsymbol{v}}$$

The BigCLAM

$$
\begin{aligned}
\nabla l(F_u) &= \sum_{v \in N(u)} F_v \frac{\exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})}{1 - \exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})} - \sum_{v \notin N(u)} F_{\boldsymbol{v}} \\
&= \sum_{v \in N(u)} F_v \frac{\exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})}{1 - \exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})} - \left(\sum_{v} F_{\boldsymbol{v}} - F_u - \sum_{v \in N(u)} F_{\boldsymbol{v}} \right) \\
&= \sum_{v \in N(u)} F_v \left(\frac{\exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})}{1 - \exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})} + 1 \right) + F_u - \sum_{v} F_{\boldsymbol{v}} \\
&= \boxed{\sum_{v \in N(u)} F_v \left(\frac{1}{1 - \exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})} \right)} + F_u - \underbrace{\sum_{v} F_{\boldsymbol{v}}}
\end{aligned}
$$

The BigCLAM

$$\nabla l(F_u) = \sum_{v \in N(u)} F_v \frac{\exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})}{1 - \exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})} - \sum_{v \notin N(u)} F_{\boldsymbol{v}}$$

$$= \sum_{v \in N(u)} F_v \frac{\exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})}{1 - \exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})} - \left(\sum_v F_{\boldsymbol{v}} - F_u - \sum_{v \in N(u)} F_{\boldsymbol{v}} \right)$$

$$= \sum_{v \in N(u)} F_v \left(\frac{\exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})}{1 - \exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})} + 1 \right) + F_u - \sum_v F_{\boldsymbol{v}}$$

for each time step
for each node u
for each nbr of u v

$$= \sum_{v \in N(u)} F_v \left(\frac{1}{1 - \exp(-F_{\boldsymbol{u}} \cdot F_{\boldsymbol{v}})} \right) + F_u - \sum_v F_{\boldsymbol{v}}$$

What did we win?? Original RH sum: $v \notin N(u)$ was linear in total # of nodes. Now we have just $|N(u)|$ size updates! We can also cache/re-use the sum-over-people community scores in $\sum_v F_{\boldsymbol{v}}$!

BigCLAM Wrapup

We will implement BigCLAM in a course notebook. But there are a couple of major concerns with the algorithm

1. How do we initialize F for our gradient ascent?
2. How might we choose k?

When considering this algorithm, consider *why* a few things are important:

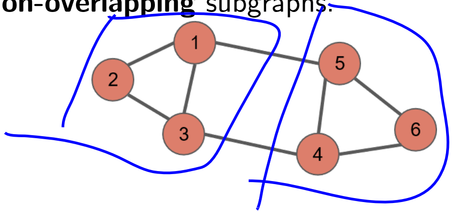
1. What happens if we have a large, sparse, graph and use a random initialization, e.g. where each node/community index get a $\text{NP.RANDOM.RAND}()$?
2. How would a background community with connection probability ε factor into the BigCLAM updates?

Graph Partitioning

BigCLAM and the AGM are models that allow for members to be in multiple communities at once, like a GMM-style soft clustering. There's a corresponding "hard clustering" approach to a graph that asks how we would break down a graph into **non-overlapping** subgraphs.

First question: What makes a "good" cluster in a graph G?

- Maximize the number of within-cluster connections?
- Minimize the number of between-cluster connections?



Definition: Given the undirected graph $G(V, E)$, with node set V and edge set E, the *bipartitioning task* is to divide the vertex set V into 2 disjoint sets A and B, such that $B = V - A$.

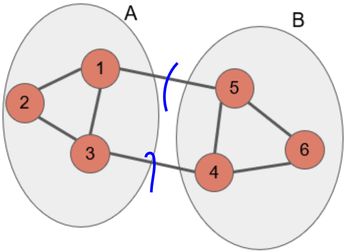
Graph Cuts

We need a measure for the “goodness” of a bipartition.

Definition: The *cut* of a partitioning of undirected graph $G(V, E)$ into A and A^C is the set of edges (or total edge weight of this set) with exactly one vertex in the set A and one in A^C.

$$cut(A) = \sum_{i \in A, j \notin A} w_{ij}$$

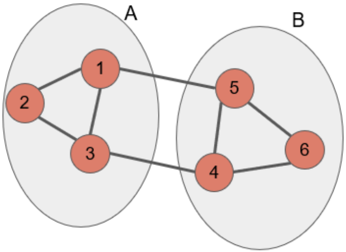
(where w_{ij} = weight of edge connecting nodes i and j; often 1 if the edge exists)



Example: On the G above with edge weights of 1, $cut(A) =$ 2.

Graph Cuts

We need a measure for the "goodness" of a bipartition.
Definition: The *cut* of a partitioning of undirected graph $G(V, E)$ into A and A^C is the set of edges (or total edge weight of this set) with exactly one vertex in the set A and one in A^C.



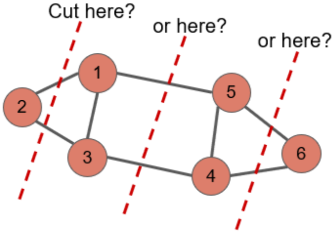
$$cut(A) = \sum_{i \in A, j \notin A} w_{ij}$$

(where w_{ij} = weight of edge connecting nodes i and j; often 1 if the edge exists)

Example: On the G above with edge weights of 1, $cut(A) = \underline{\quad\quad 2}$.

Graph Cuts

Goal: Over all possible bipartitions on a graph, find the one with the best cut.



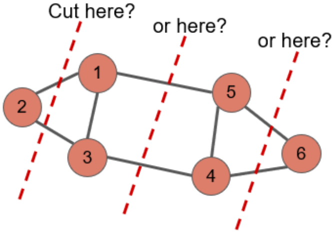
Preliminary Idea: Find the *minimum-cut*.

1. **Implementation?:** Cut in such a way as to minimize the weight of between-community edges
2. **Implementation?:** Choose communities A and B satisfying $\mathrm{argmin}_{A,\,B}\, cut(A)$.

Concerns:

Graph Cuts

Goal: Over all possible bipartitions on a graph, find the one with the best cut.



Preliminary Idea: Find the *minimum-cut*.

1. **Implementation?:** Cut in such a way as to minimize the weight of between-community edges
2. **Implementation?:** Choose communities A and B satisfying $\mathrm{argmin}_{A,\,B}\, cut(A)$.

Concerns:

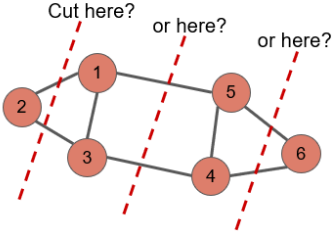
1. **Uniqueness:** If unweighted, where do we cut the above graph?

Graph Cuts

Goal: Over all possible bipartitions on a graph, find the one with the best cut.

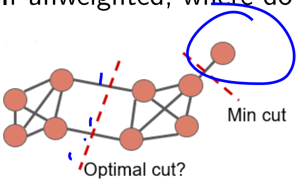
Preliminary Idea: Find the *minimum-cut*.

1. **Implementation?:** Cut in such a way as to minimize the weight of between-community edges
2. **Implementation?:** Choose communities A and B satisfying $\text{argmin}_{A,\,B}\, cut(A)$.



Concerns:

1. **Uniqueness:** If unweighted, where do we cut the above graph?
2. **Even worse:**



Normalized Cuts

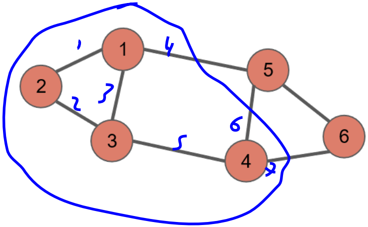
Goal: Over all possible bipartitions on a graph, find the one with the best cut.

To make sure the cuts split the graph *roughly* in half, we can use **normalized cuts**.

Definition: The *volume* of a vertex set S, denoted $vol(S)$, is the number of edges with at least one end in S.

Example: Find the volume of $A = \{1, 2, 3, 4\}$ on the graph at right.

size = 7

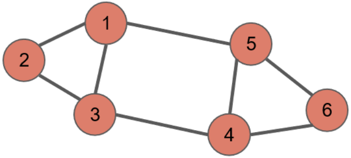


Normalized Cuts

Goal: Over all possible bipartitions on a graph, find the one with the best cut.

To make sure the cuts split the graph *roughly* in half, we can use **normalized cuts**.

Definition: The *volume* of a vertex set S, denoted $vol(S)$, is the number of edges with at least one end in S.



Example: Find the volume of $A = \{1, 2, 3, 4\}$ on the graph at right.

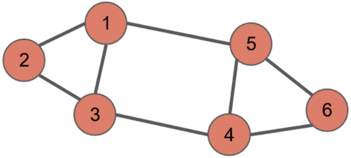
Solution: The volume of A is 7: it counts all of the 8 edges in the original graph except the [5,6] edge.

Normalized Cuts

Goal: Over all possible bipartitions on a graph, find the one with the best cut.

To make sure the cuts split the graph *roughly* in half, we can use **normalized cuts**.

Definition: The *volume* of a vertex set S, denoted $vol(S)$, is the number of edges with at least one end in S.

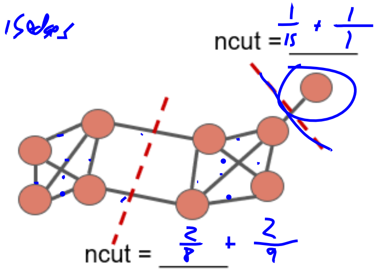


Example: Find the volume of $A = \{1, 2, 3, 4\}$ on the graph at right.

Solution: The volume of A is 7: it counts all of the 8 edges in the original graph except the [5,6] edge.

Definition: The *normalized cut value* of a cut, denoted $ncut(A, B)$, for a bipartition of node set V into disjoint sets A and B is given by:

$$ncut(B) = ncut(A) = \frac{cut(A,B)}{vol(A)} + \frac{cut(A,B)}{vol(B)}$$

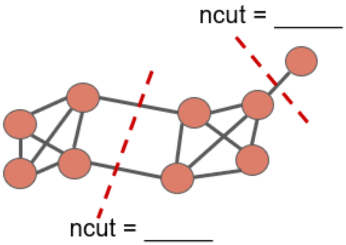


Normalized Cuts

Goal: Over all possible bipartitions on a graph, find the one with the best cut.
To make sure the cuts split the graph *roughly* in half, we can use **normalized cuts**.

Definition: The *normalized cut value* of a cut, denoted $ncut(A, B)$, for a bipartition of node set V into disjoint sets A and B is given by:

$$ncut(A) = \frac{cut(A,B)}{vol(A)} + \frac{cut(A,B)}{vol(B)}$$



Idea: Try to weakly encourage $vol(A) \approx vol(B)$.

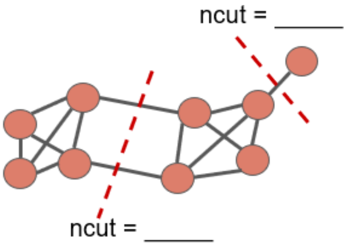
Example

Normalized Cuts

Goal: Over all possible bipartitions on a graph, find the one with the best cut.
To make sure the cuts split the graph *roughly* in half, we can use **normalized cuts**.

Definition: The *normalized cut value* of a cut, denoted $ncut(A, B)$, for a bipartition of node set V into disjoint sets A and B is given by:

$$ncut(A) = \frac{cut(A, B)}{vol(A)} + \frac{cut(A, B)}{vol(B)}$$



Idea: Try to weakly encourage $vol(A) \approx vol(B)$.

Example Solution: the cut in the middle cuts 2 edges, but leaves $vol(A) = 8$, $vol(B) = 9$ for a $ncut = \frac{2}{9} + \frac{2}{8}$. The cut at the right cuts only one edge, but leaves lopsided volumes of $vol(A) = 15$, $vol(B) = 1$ for a total $ncut$ of $\frac{16}{15}$... it's worse!

Finding Best Cuts

Definition: The *normalized cut value* of a cut, denoted $ncut(A, B)$, for a bipartition of node set V into disjoint sets A and B is given by:

$$ncut(A) = \frac{cut(A,B)}{vol(A)} + \frac{cut(A,B)}{vol(B)}$$

Goal: Find the optimal best cut for a given graph.

1. Examine all possible partitionings' normalized cut scores.
2. Pick the partitioning that minimizes ncut.
3. Easy, right?!

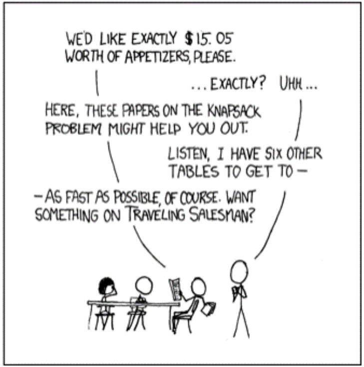
Finding Best Cuts

Definition: The *normalized cut value* of a cut, denoted $ncut(A, B)$, for a bipartition of node set V into disjoint sets A and B is given by:

$$ncut(A) = \frac{cut(A, B)}{vol(A)} + \frac{cut(A, B)}{vol(B)}$$

Goal: Find the optimal best cut for a given graph.

1. Examine all possible partitionings' normalized cut scores.
2. Pick the partitioning that minimizes ncut.
3. Easy, right?!
4. Wrong! **NP-hard**, actually



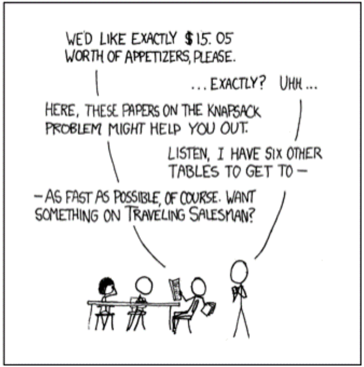
Finding Best Cuts

Definition: The *normalized cut value* of a cut, denoted $ncut(A, B)$, for a bipartition of node set V into disjoint sets A and B is given by:

$$ncut(A) = \frac{cut(A, B)}{vol(A)} + \frac{cut(A, B)}{vol(B)}$$

Goal: Approximate the optimal best cut for a given graph.

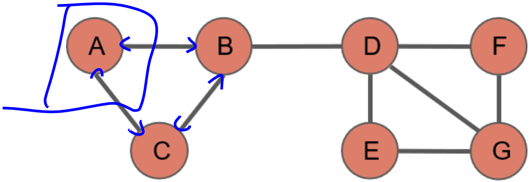
1. Examine all possible partitionings' normalized cut scores.
2. Pick the partitioning that minimizes ncut.
3. Easy, right?!
4. Wrong! **NP-hard**, actually



Finding Best Cuts

We can turn the "best normalized cut" problem into an eigenvalue problem for an **approximate** solution! But of what matrix, and why?

Definition:



The *adjacency matrix* of a graph G is given by:

$$A_{ij} = \begin{cases} 1 & \text{if nodes i and j share an edge} \\ 0 & \text{else.} \end{cases}$$

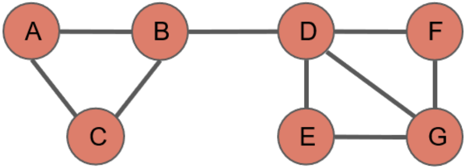
Consider: Suppose we have a vector x which has some value for each node. What does the vector Ax represent?

$$\underbrace{\begin{array}{c} \begin{array}{ccccccc} A & B & C & D & E & F & G \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ & & & & & & \\ & & & & & & \end{bmatrix} \end{array}}_{A} \underbrace{\begin{bmatrix} x_A \\ x_B \\ x_C \\ \vdots \\ x_G \end{bmatrix}}_{x} = \begin{bmatrix} x_B + x_C \\ \\ \\ \end{bmatrix}$$

Finding Best Cuts

We can turn the "best normalized cut" problem into an eigenvalue problem for an **approximate** solution! But of what matrix, and why?

Definition:



The *adjacency matrix* of a graph G is given by:

$$A_{ij} = \begin{cases} 1 & \text{if nodes i and j share an edge} \\ 0 & \text{else.} \end{cases}$$

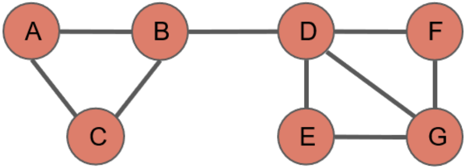
Consider: Suppose we have a vector x which has some value for each node. What does the vector Ax represent? **Solution:**

$$\begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & & \vdots \\ a_{n1} & \dots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}.$$

Finding Best Cuts

We can turn the "best normalized cut" problem into an eigenvalue problem for an **approximate** solution! But of what matrix, and why?

Definition:



The *adjacency matrix* of a graph G is given by:

$$A_{ij} = \begin{cases} 1 & \text{if nodes i and j share an edge} \\ 0 & \text{else.} \end{cases}$$

Consider: Suppose we have a vector x which has some value for each node. What does the vector Ax represent? **Solution:**

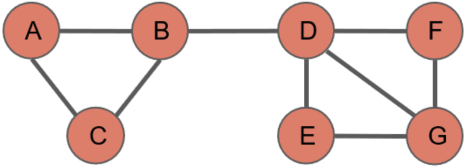
$$\begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & & \vdots \\ a_{n1} & \dots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}.$$

Then $y_i = \sum_{i=1}^n A_{ij}x_j = \boxed{\sum_{(i,j)\in E}} x_j$. In other words: y_i is the sum of the x values of all nodes *connected* to node i.

Spectral Analysis

Definition: The spectrum of a matrix consists of its eigenvectors x_i, ordered by the **magnitude** of their corresponding eigenvalues λ_i: $\Lambda = \lambda_1, \lambda_2, \ldots \lambda_n$, (where $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$)

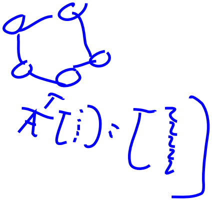
How do these help us? *Spectral graph theory* is using the "spectrum" of the matrix representing our graph G, and seeing what it tells us about the system G models!



So let's talk eigenvalues/eigenvectors. Suppose G is a d-regular connected graph, that each node has degree d (to simplify things at first; we'll back this off in a minute).

Goal: Seeking λ and x such that $Ax = \lambda x$

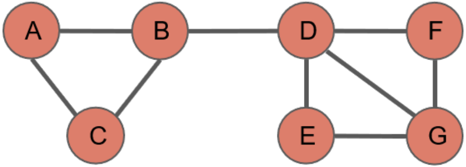
Example: For $x = [1, 1, \ldots, 1]$, what is Ax?



Spectral Analysis

Definition: The spectrum of a matrix consists of its eigenvectors x_i, ordered by the **magnitude** of their corresponding eigenvalues λ_i: $\Lambda = \lambda_1, \lambda_2, \ldots \lambda_n$, (where $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$)

How do these help us? *Spectral graph theory* is using the "spectrum" of the matrix representing our graph G, and seeing what it tells us about the system G models!

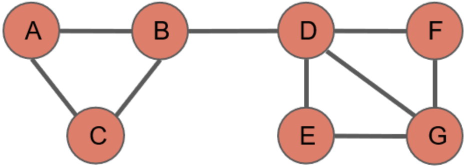


So let's talk eigenvalues/eigenvectors. Suppose G is a d-regular connected graph, that each node has degree d (to simplify things at first; we'll back this off in a minute).
Goal: Seeking λ and x such that $Ax = \lambda x$
Example: For $x = [1, 1, \ldots, 1]$, what is Ax?
Solution: $Ax = [d, d, \ldots, d] = dx$, or $\lambda = d$ is an eigenvalue associated with the 1's vector. Nice!

Spectral Analysis

Definition: The spectrum of a matrix consists of its eigenvectors x_i, ordered by the **magnitude** of their corresponding eigenvalues λ_i: $\Lambda = \lambda_1, \lambda_2, \ldots \lambda_n$, (where $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$)

How do these help us? *Spectral graph theory* is using the "spectrum" of the matrix representing our graph G, and seeing what it tells us about the system G models!



So let's talk eigenvalues/eigenvectors. Suppose G is a d-regular connected graph, that each node has degree d (to simplify things at first; we'll back this off in a minute).

Goal: Seeking λ and x such that $Ax = \lambda x$

Example: For $x = [1, 1, \ldots, 1]$, what is Ax?

Solution: $Ax = [d, d, \ldots, d] = dx$, or $\lambda = d$ is an eigenvalue associated with the 1's vector. Nice!

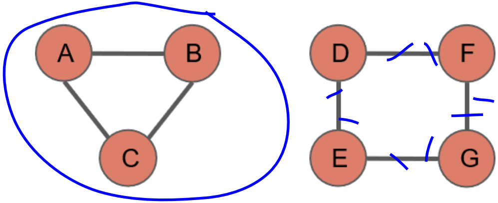
NB: We probably already knew this one, since this is the result of treating A like a stochastic (Markov) transition matrix!

Spectral Analysis

What if the graph G is **not-connected**? For example, consider the G below that consists of 2 components.

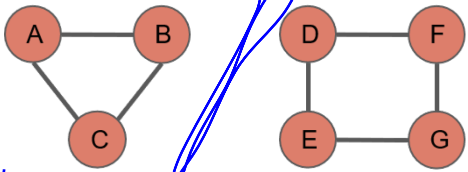
Goal: Seeking λ and x such that $Ax = \lambda x$

Examples: consider $x^A = 1$'s for components in subgraph A and 0's for components in subgraph B.



Spectral Analysis

What if the graph G is **not-connected**? For example, consider the G below that consists of 2 components.



Goal: Seeking λ and x such that $Ax = \lambda x$

Examples: consider $x^A = 1$'s for components in subgraph A and 0's for components in subgraph B.

Solution: Then:

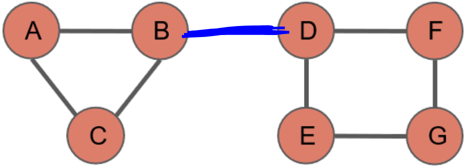
left rH +leA

$$Ax^A = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 2 \\ 2 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = 2x_A$$

Spectral Analysis

What if the graph G is **not-connected**? For example, consider the G below that consists of 2 components.

Goal: Seeking λ and x such that $Ax = \lambda x$



Examples: consider $x^A = 1$'s for components in subgraph A and 0's for components in subgraph B.

Result: x^A and x^B are **both** eigenvectors with associated eigenvalues of $d = 2$. They're also *linearly independent*, since $x^A \cdot x^B = 0$.

How does this help us? One measure that cutting the edge from B to D was a "good" cut might be that the original graph has eigenvectors *close to* x^A and x^B.

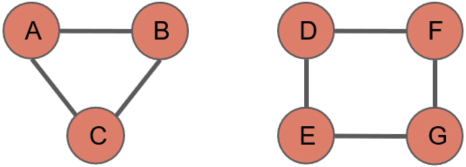
Spectral Analysis

Spectral Intuition If the graph G is **not-connected**:

1. If each component is degree d, we get eigenvalues of $\lambda_1 = \lambda_2 = d$ and eigenvectors of x^A and x^B, just bunches of 1s.

If the graph G is connected by only a few edges...

1. We should probably get eigenvalues that are similar $\lambda_1 \approx \lambda_2$ and eigenvectors that are *similar* to x^A and x^B.



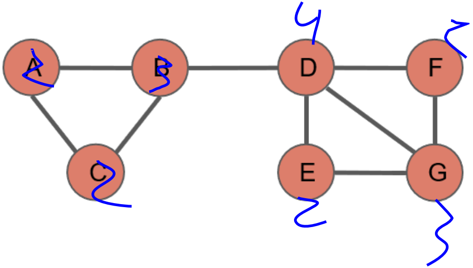
If we can directly solve for the eigenstuff of our system, that's great and could answer this. But we usually cant for very large matrices! So we find a way to approximate or only find *some* eigenvalues. We can simply this problem by asking about only a single matrix that combines both degree d and adjacency A.

Definition: The *degree matrix* D of a graph with n nodes is the $n \times n$ diagonal matrix with $D_{ii} = d_i$, where d_i is the degree of node i.

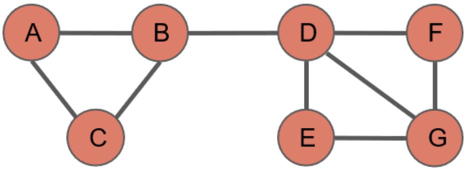
D_{ii}

Definition: The *Laplacian matrix* L of a graph is given by $L = D - A$.

Example: what are D, L, and A for the graph given?



Definition: The *degree matrix* D of a graph with n nodes is the $n \times n$ diagonal matrix with $D_i i = d_i$, where d_i is the degree of node i.



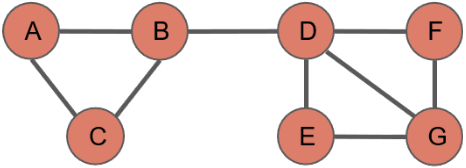
Definition: The *Laplacian matrix* L of a graph is given by $L = D - A$.

Example: what are D, L, and A for the graph given?

$$\begin{array}{c} \begin{array}{cccc} A & B & C & \cdots \end{array} \\ \begin{bmatrix} 2 & & & & \\ & 3 & & & \\ & & 2 & & \\ & & & \ddots & \\ & & & & 3 \end{bmatrix} \end{array} - \begin{bmatrix} & 1 & 1 & & & & \\ 1 & & 1 & 1 & & & \\ 1 & 1 & & & & & \\ & 1 & & & 1 & 1 & 1 \\ & & & 1 & & & 1 \\ & & & 1 & & & 1 \\ & & & 1 & 1 & 1 & \end{bmatrix} = \begin{bmatrix} 2 & -1 & -1 & & & & \\ -1 & 3 & -1 & -1 & & & \\ -1 & -1 & 2 & & & & \\ & -1 & & 4 & -1 & -1 & -1 \\ & & & -1 & 2 & & -1 \\ & & & -1 & & 2 & -1 \\ & & & -1 & -1 & -1 & 3 \end{bmatrix}$$

The Graph Laplacian

Definition: The *degree matrix* D of a graph with n nodes is the $n \times n$ diagonal matrix with $D_i i = d_i$, where d_i is the degree of node i.



Definition: The *Laplacian matrix* L of a graph is given by $L = D - A$.

Properties of L:

1. Row and column sums are all zero... so
2. There's a trivial eigenpair of $x = [1, 1, \ldots 1]$ with $\lambda = 0$.
3. All eigenvalues are non-negative and real. (Symmetry helps here!)
4. All eigenvectors are real *and orthogonal*, with dot products of 0 against one another.

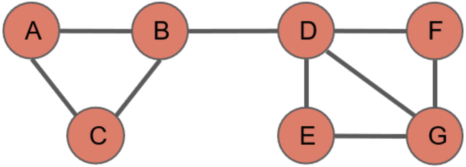
The Graph Laplacian

So what does L give us?
Theorem: For a symmetric matrix M, the second

smallest eigenvalue, with eigenvalue x, satisfies

$$\lambda_2 = \min_{\vec{x}} \frac{\vec{x}^T M \vec{x}}{\vec{x}^T \vec{x}}$$

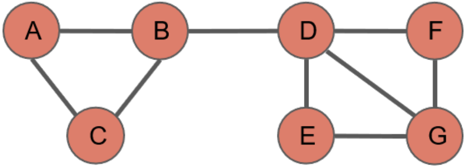
...why do we care about the *second-smallest* eigenvalue?



The Graph Laplacian

So what does L give us?
Theorem: For a symmetric matrix M, the second smallest eigenvalue, with eigenvalue x, satisfies

$$\lambda_2 = \min_{\vec{x}} \frac{\vec{x}^T M \vec{x}}{\vec{x}^T \vec{x}}$$



...why do we care about the *second-smallest* eigenvalue?

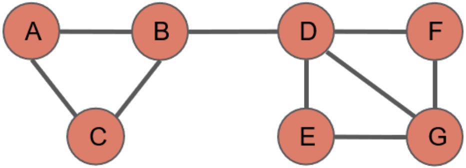
1. The smallest was the trivial one with $\lambda_1 = 0$
2. So this is the first/smallest one with any "interesting" information.

...but what does $\vec{x}^T M \vec{x}$ represent, and why might it be interesting?

The Graph Laplacian

So what does L give us?
Theorem: For a symmetric matrix M, the second

smallest eigenvalue, with eigenvalue x, satisfies



$$\lambda_2 = \min_{\vec{x}} \frac{\vec{x}^T M \vec{x}}{\vec{x}^T \vec{x}}$$

...why do we care about the *second-smallest* eigenvalue?

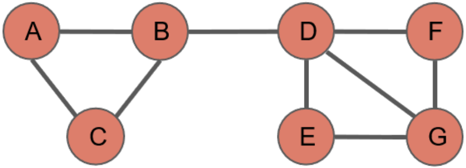
$\vec{x}^T M \vec{x}$ is called a *quadratic form* of M.
Example: compute

$$\begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$=$

The Graph Laplacian

So what does L give us?
Theorem: For a symmetric matrix M, the second smallest eigenvalue, with eigenvalue x, satisfies



$$\lambda_2 = \min_{\vec{x}} \frac{\vec{x}^T M \vec{x}}{\vec{x}^T \vec{x}}$$

...why do we care about the *second-smallest* eigenvalue?

$\vec{x}^T M \vec{x}$ is called a *quadratic form* of M.
Example: compute

$$\begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$= Ax_1^2 + Dx_2^2 + (B+C)x_1x_2$
Idea: Looks like FOILing!

The Graph Laplacian

Goal: Interpret $\vec{x}^T L\vec{x}$ for a graph G.

$$\vec{x}^T L\vec{x} = \underbrace{\sum_{i,j=1}^{n} L_{ij}x_ix_j}_{quad\,form} = \sum_{i,j=1}^{n}(D_{ij} - A_{i,j})x_ix_j = \sum_{i,j=1}^{n} D_{ij}x_ix_j - 2\sum_{i,j=1}^{n} A_{ij}x_ix_j$$

The Graph Laplacian

Goal: Interpret $\vec{x}^T L \vec{x}$ for a graph G.

$$\vec{x}^T L \vec{x} = \underbrace{\sum_{i,j=1}^{n} L_{ij} x_i x_j}_{quad\, form} = \sum_{i,j=1}^{n} (D_{ij} - A_{i,j}) x_i x_j = \sum_{i,j=1}^{n} D_{ij} x_i x_j - 2 \sum_{i,j=1}^{n} A_{ij} x_i x_j$$

$$= \sum_{i=1}^{n} D_{ii} x_i^2 - 2 \sum_{edges} x_i x_j = \sum_{edges} \left(x_i^2 + x_j^2 - 2 x_i x_j \right)$$

The Graph Laplacian

Goal: Interpret $\vec{x}^T L\vec{x}$ for a graph G.

$$\vec{x}^T L\vec{x} = \underbrace{\sum_{i,j=1}^{n} L_{ij}x_ix_j}_{quad\,form} = \sum_{i,j=1}^{n}(D_{ij} - A_{i,j})x_ix_j = \sum_{i,j=1}^{n} D_{ij}x_ix_j - 2\sum_{i,j=1}^{n} A_{ij}x_ix_j$$

$$= \sum_{i=1}^{n} D_{ii}x_i^2 - 2\sum_{edges} x_ix_j = \sum_{edges}\left(x_i^2 + x_j^2 - 2x_ix_j\right)$$

$$= \sum_{(i,j)\in E} (x_i - x_j)^2$$

The Graph Laplacian

Goal: Interpret $\vec{x}^T L\vec{x}$ for a graph G.

$$\vec{x}^T L\vec{x} = \underbrace{\sum_{i,j=1}^{n} L_{ij}x_i x_j}_{quad\, form} = \sum_{i,j=1}^{n}(D_{ij} - A_{i,j})x_i x_j = \sum_{i,j=1}^{n} D_{ij}x_i x_j - 2\sum_{i,j=1}^{n} A_{ij}x_i x_j$$

$$= \sum_{i=1}^{n} D_{ii}x_i^2 - 2\sum_{edges} x_i x_j = \sum_{edges}\left(x_i^2 + x_j^2 - 2x_i x_j\right)$$

$$= \sum_{(i,j)\in E}(x_i - x_j)^2$$

Interpretation: For a vector x, $\vec{x}^T L\vec{x}$ measures the (squared) distance between the components of x, *but only where G had edges*.

The Graph Laplacian

So: $\vec{x}^T L\vec{x} = \sum_{(i,j)\in E} (x_i - x_j)^2$ helps us with our theorem:

$\lambda_2 = \min_{\vec{x}} \frac{\vec{x}^T L\vec{x}}{\vec{x}^T \vec{x}}$

Interpretation: $\vec{x}^T L\vec{x}$ measures the (squared) distance between the components of x where G had edges.
Further, if we're trying to find the eigenvector x, we can:

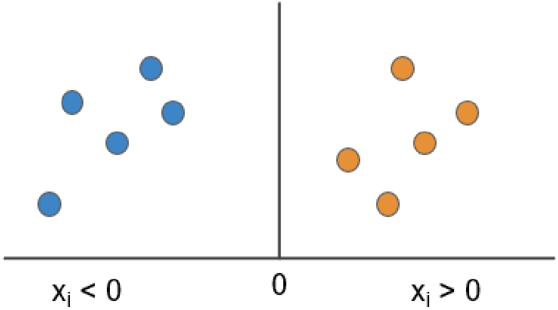
1. Define it as a unit vector, so $\sum_i x_i^2 = 1$
2. Note that it must be *orthogonal* to $x_1 = [1, 1, \ldots, 1]$, which means that $x_1 \cdot x_2 = 0$ or $\sum_i x_i = 0$

So $\lambda_2 = \min_{\vec{x}:\sum x_i = 0} \sum_{(i,j)\in E} (x_i - x_j)^2$.

Balance and the Laplacian

$$\lambda_2 = \min_{\vec{x}:\sum x_i = 0} \sum_{(i,j)\in E} (x_i - x_j)^2.$$

In other words, we're looking for an eigenpair that *balances* the node values x_i about 0. The values sum to 0 but are chosen to that x values on nodes that share an edge should be close together.



Balance and the Laplacian

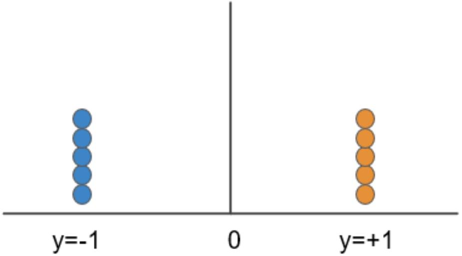
$$\lambda_2 = \min_{\vec{x}:\sum x_i=0} \sum_{(i,j)\in E} (x_i - x_j)^2.$$

This vector that assigns similar x_i values to nodes that are connected by an edge inherently will assign similar x_i values to *groups* of nodes that are heavily connected. This naturally lends it to a **partition** if we draw a cutoff based on x_i values. And $x_i = 0$ is **on average** right in the middle in the vector, since it sums to 0!

We can create a hard cluster or a *partition* (A, B) by

creating the vector y such that:

$$y = \begin{cases} +1 & \text{if node i is in A} \\ -1 & \text{if node i is in B} \end{cases}$$



Balance and the Laplacian

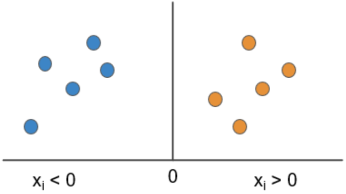
The best *partition* vector y is the one that cuts the least number of edges while balancing +1's and -1's. Or we're solving

$$\vec{x} = \min_{\vec{y} \in [-1,1]^n} \sum_{(i,j) \in E} (y_i - y_j)^2$$

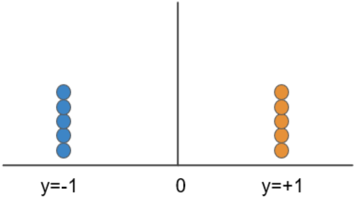
which is almost the same problem as our eigenvalue problem! So we create y **from** the eigenvector, and instead find

$$\vec{x} = \min_{\vec{y} \in \mathbb{R}^n} \sum_{(i,j) \in E} (y_i - y_j)^2$$

which is called the *Fiedler vector*, and is the optimal solution for the given minimization.



$\rightarrow$

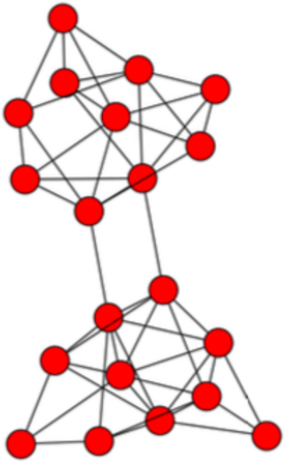


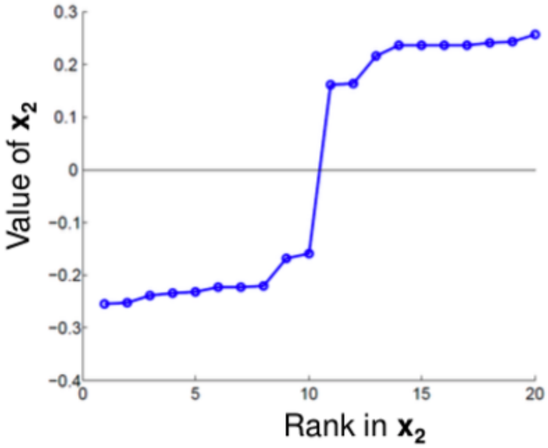
Spectral Graph Partitioning

Algorithm: 3 steps

1. Pre-processing – construct the matrix representation of our graph (A, D and L)
2. Decomposition – compute eigenvalues and eigenvectors of L
   – In other words, we're mapping each node to a lower-dimensional representation (one x_i value per node!), based on eigenvectors. We'll do more **dimension reduction** in coming weeks.
3. Grouping – look at second eigenvalue and its eigenvector x_2.
   – gives the "weights" / "values" / "labels" for each node
   – which are left of 0? Which are right?
4. Those grouping are the partition for the cut, so we're done... but try to plot/visualize the resulting graph.

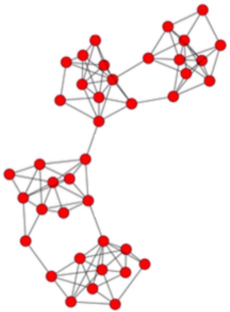
Fiedler Vector in Action: Finding Communities

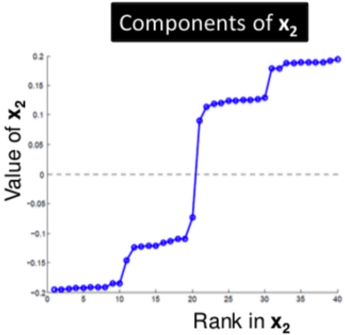




Fiedler Vector in Action: Finding *More* Communities

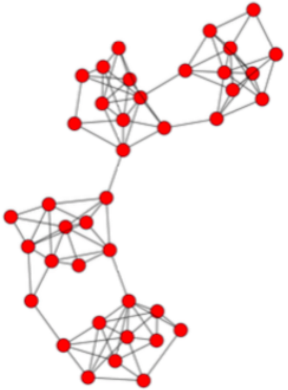
Three options to find multiple cuts:

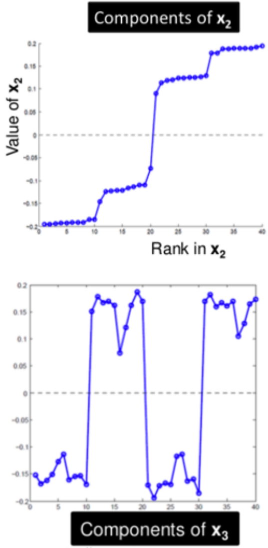




1. Find multiple places to cut the original x vector, not just zero. **Idea:** cut at places with *large jumps* in x-value.
2. Cut at $x = 0$ for the second eigenvector, then *also* at $x = 0$ for the third eigenvector. Or better: **cluster** nodes based on their values from *each* eigenvector (or as many as you compute)
3. Cut at $x = 0$ for the second eigenvector, then *recompute* the Fiedler vector for the new subgraphs from that bipartition. Repeat.

Finding *More* Communities





Using the zeros of k eigenvectors can result in up to 2^k partitions;

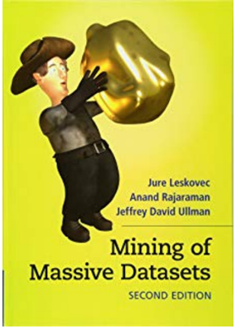
making a *bipartition* of the original *bipartitions* will result in up to 2^k partitions;

for a fixed k a single partition on x_2 is likely easiest.

Acknowledgments

Next time: On to recommendations!

Some material is adapted/adopted from Mining of Massive Data Sets, by Jure Leskovec, Anand Rajaraman, Jeff Ullman (Stanford University) `http://www.mmds.org`



Special thanks to Tony Wong for sharing his original adaptation and adoption of slide material.