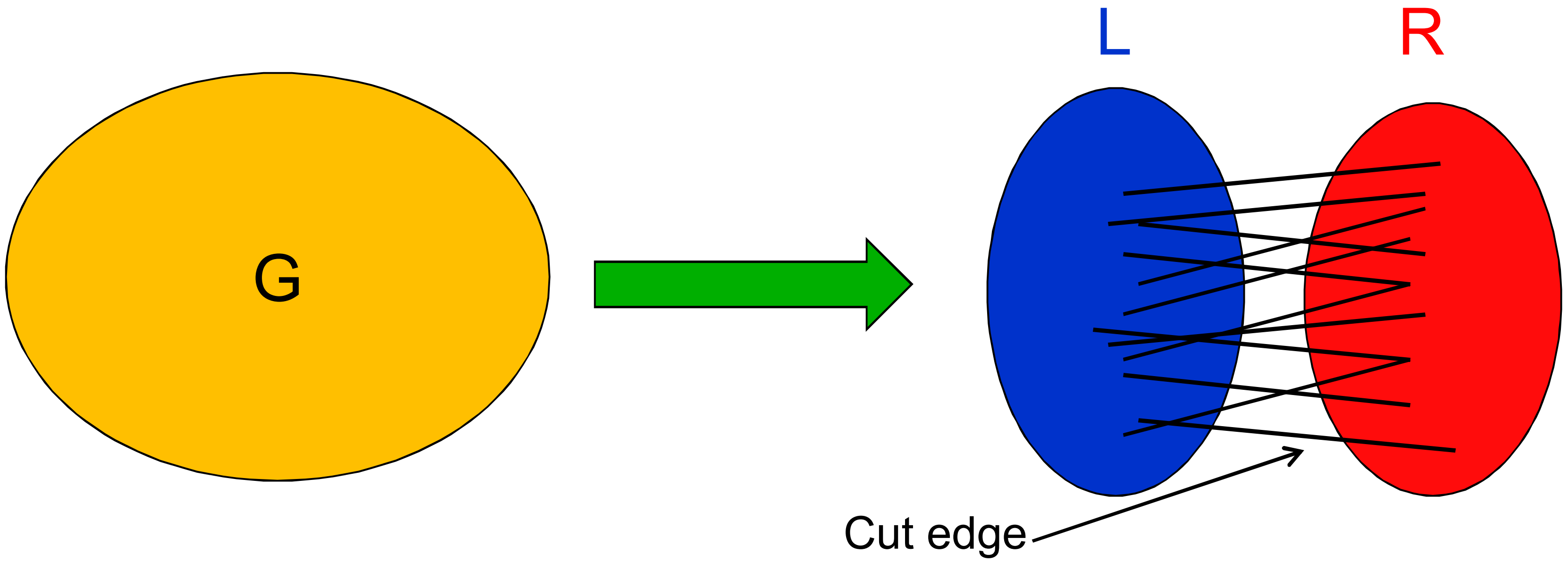
SAND2011-0095C

# Combinatorial Approximation Algorithms for MAXCUT using Random Walks

C. Seshadhri (Sandia National Labs, Livermore)

Joint work with
Satyen Kale (Yahoo! Research, Santa Clara)

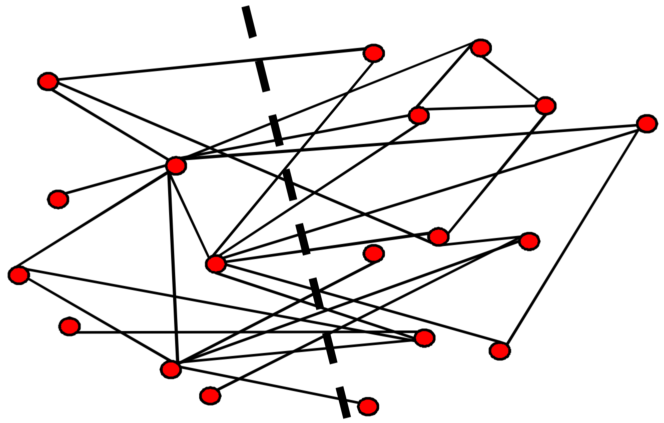
# The MAXCUT problem



Given graph G = (V, E), find a partition (or cut) V = L U R maximizing number of edges cut i.e.
(i, j) 2 E such that i 2 L and j 2 R

Fraction of cut edges is MaxCut value

# The MAXCUT problem

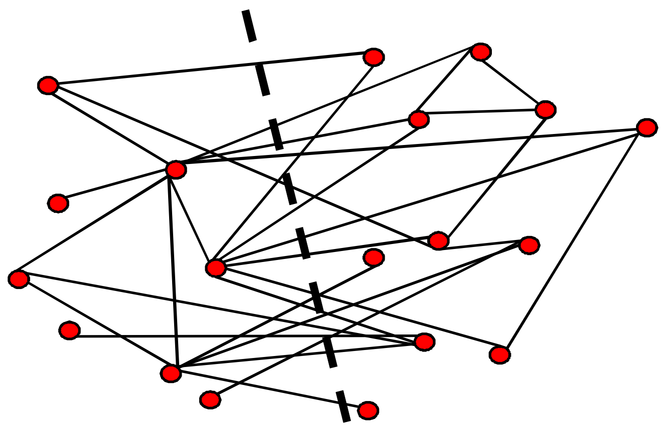


- [Karp ‘72] NP-hard
- So aim is to develop **approximation algorithms**
- Algo is β-approx, if cut given by algo satisfies:

  Algo(G) > β MaxCut(G)

  (Note that β < 1)

# The MAXCUT problem



- [Karp '72] NP-hard
- Greedy/LP relaxation give 0.5 approximation
- [Goemans-Williamson '95]: SDP → 0.878.. approximation
- [Arora, Kale '07]: GW approx alg implementation in $O^*(m)$ time
- [Trevisan, Soto '09] Eigenvalue approach gives 0.61 approx

# Combinatorial Algorithms

- No known *combinatorial* algo with > 0.5 approx
  - [VK07, STT07] Even LP hierarchies have integrality gap of 0.5
- Combinatorial: using graph structure, simple primitive operations, no matrix algebra

- Why combinatorial?
  - Reveals more insight into structure of problem
  - Less numerical issues
  - Likely better running time in practice?

## Our results

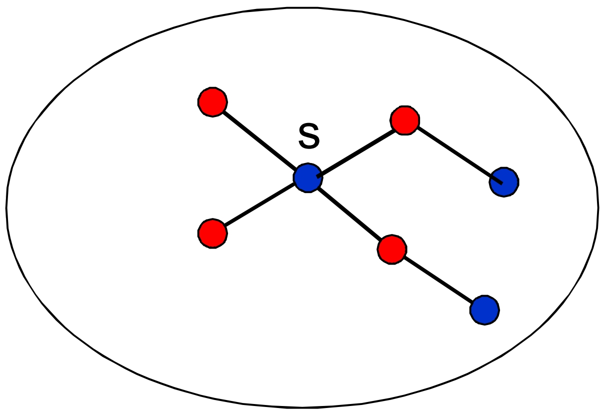
- A really really neat combinatorial algo beating the 0.5 approx factor
  - Really. It's neat.
  - For any $\mu > 0$, a combinatorial alg running in $O(m + n^{1.5 + \mu})$ time with approx factor $= 0.5 + g(\mu)$

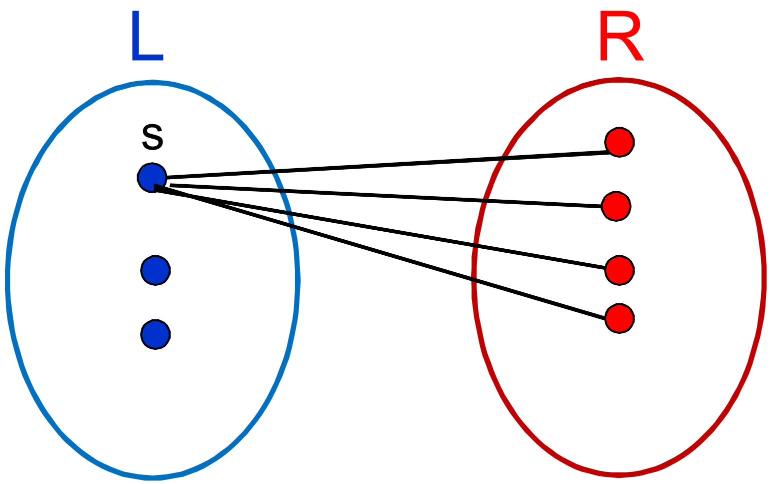
Increasing with $\mu$

- Analyze natural combinatorial heuristic via spectral methods
  - Trevisan/Soto analysis used

- Basic version of algorithm takes $O(n^{2 + \mu})$ time for $0.5 + h(\mu)$ approximation
  - For faster running time, devise new local graph partitioning algorithm

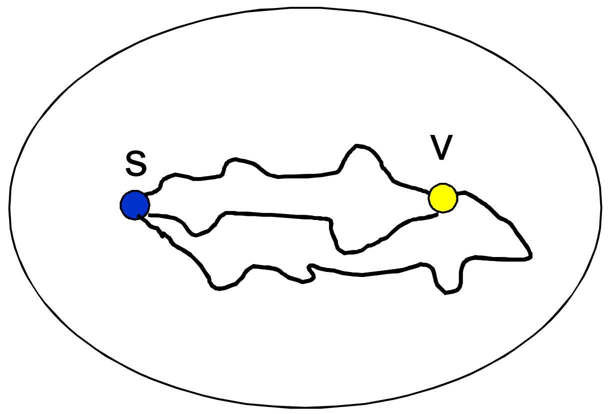
# A triviality

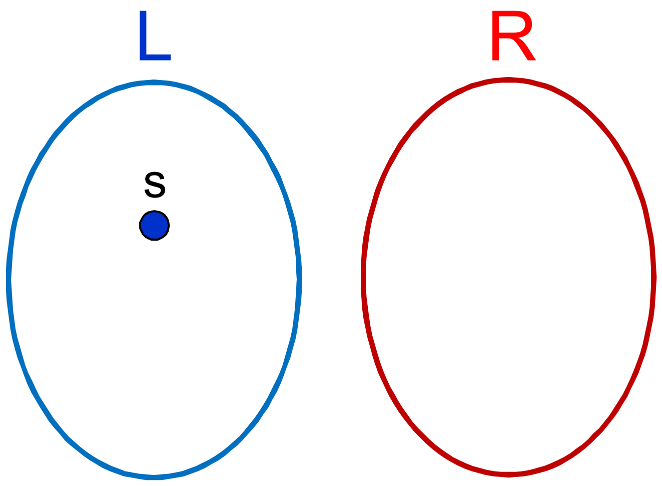




- Suppose MaxCut = 1 (a.k.a bipartite)
- Find optimal cut
- Look at path from s to v
  - If even length, v in L. If odd length, v in R.

# Generalize!

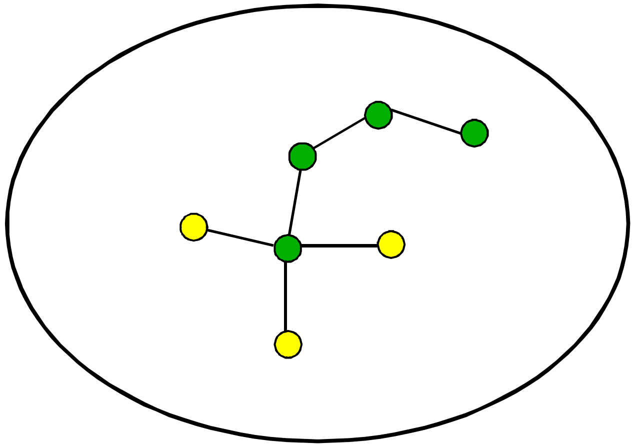




- Paths from s to v not of same parity
- If most paths (of length “roughly” r) are even, v in L. Else, v in R
- How to count this?
  - Random walks!

## Lazy Random Walks

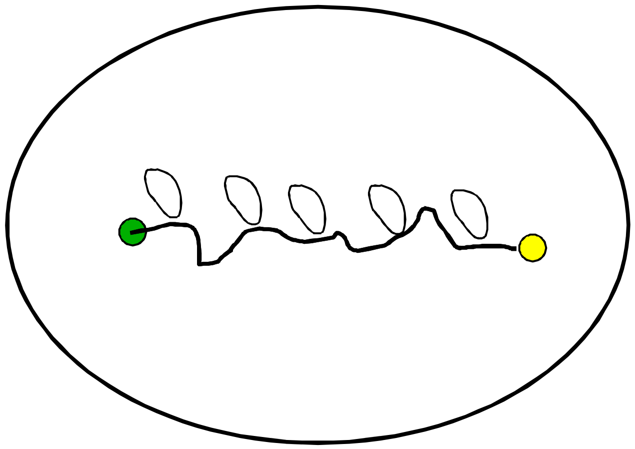
- Random process moving from node to node



- At each time step:
  - w.p. 0.5: stay at current node
  - w.p. 0.5: move to a uniform random neighbor

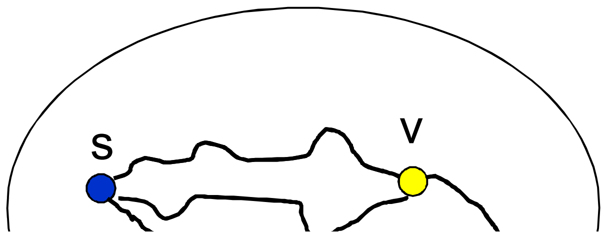
- Parity of walk = parity of length excluding self loops
  - Fixed walk *length* gives different parities

## Even and odd walks



- Even walk is one where parity of walk is…umm…even
- Odd walk has odd parity

## (Hopeful) heuristic



Our result: hope is not misplaced! Heuristic works!

- Take a random starting vertex s
- Run several random walks of some length r
- If v is hit often:

  if #(even walks to v) > #(odd walks to v), v in L

  else, v in R

# Approximation Algorithm

$d_s$: degree of s

- Take a random starting vertex s w.p. / $d_s$

- Run w random walks of some length r
  - w = O(n), r = O(log n)

t: threshold parameter

- For every vertex v:

  If #(even walks to v) - #(odd walks to v) > $td_v$, v in L

  If […] < $td_v$, v in R

- Recur on unclassified vertices

## That's it!

- Theorem: Can choose parameters s.t. approximation factor > 0.5.
  - We call this algorithm Simple
- Running time tradeoff: longer walk (larger r)
  - Better approx factor
  - [#(even walks) - #(odd walks)]/w gets smaller, so larger w needed to estimate accurately
- Better approx factor needs more time

# MaxCut and the Largest Eigenvalue

- Laplacian of graph
  - A = adjacency matrix, i.e. A(i, j) = 1 if {i, j} is an edge, 0 otherwise
  - For regular graph, $\mathcal{L}$ = I – A/d

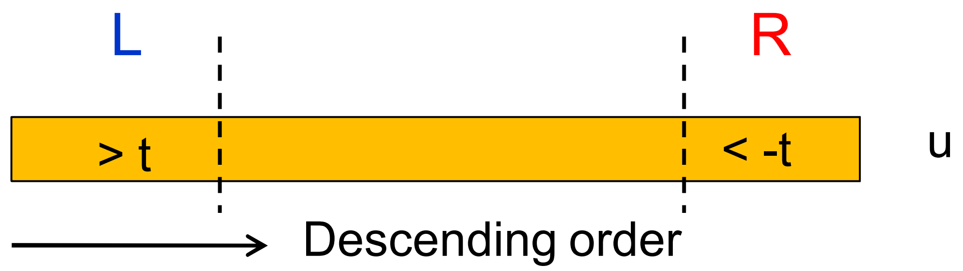
$$\max \sum_{(i,j)\in E} (x_i - x_j)^2 \qquad \forall i, x_i \in \{-1, +1\}$$

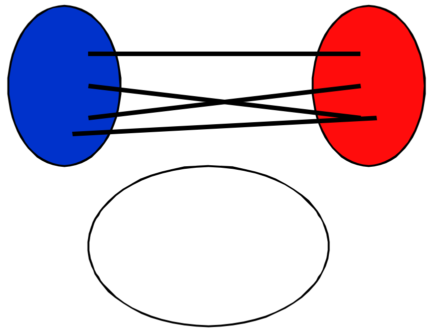
$$x^T \mathcal{L} x = \sum_{(i,j)\in E} (x_i - x_j)^2$$

$$\max x^T \mathcal{L} x \qquad x \in \mathbb{R}^n$$

But that's just top eigenvector of $\mathcal{L}$!

# Trevisan’s Spectral Algorithm





- Compute top eigenvector u of Laplacian $\mathcal{L}$
- Find  threshold t to cut most edges as follows:

  classify nodes i s.t. $u_i$ ¸ t as left, and $u_i$ · –t as right

- If no t cuts ¸ 0.5 fraction, output greedy cut and stop
- Else, classify using best t and recur on unclassified nodes

## Trevisan's Spectral Algorithm

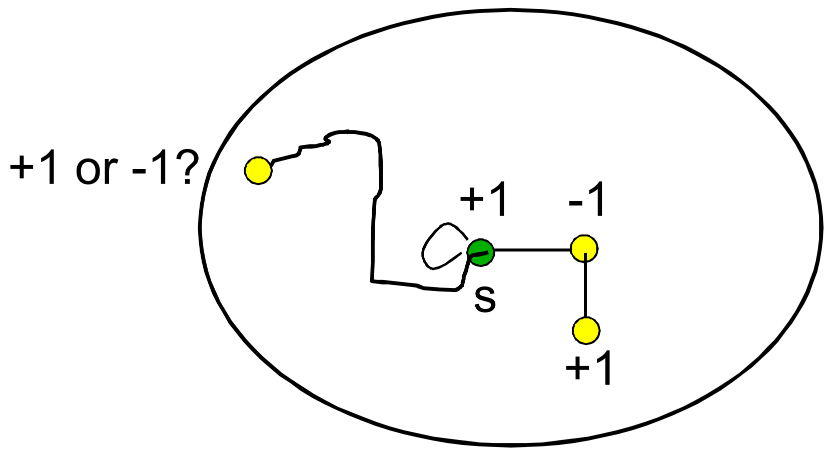
- Thm [Trevisan '09, Soto '09]: spectral algorithm has approx factor 0.6142.
- Running time: $O^*(mn)$
  - Computing approx top eigenvector: $O^*(m)$
  - Finding best threshold: $O^*(m)$
  - # of recursive levels: $O(n)$

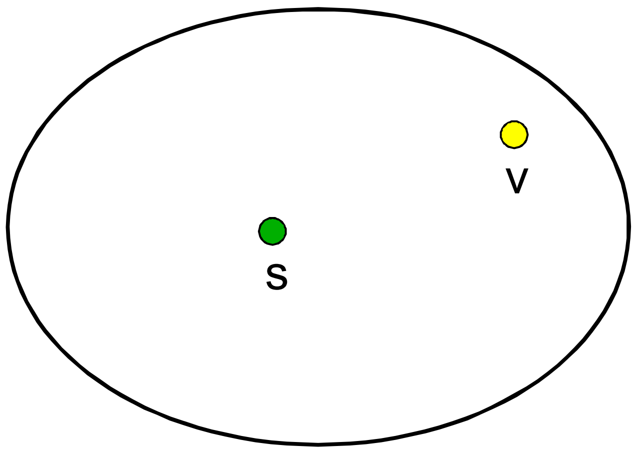
Easy sampling reduction makes all degrees O(log n) and hence m = $O^*(n)$

## Main Insight

- Random walk heuristic is combinatorial implementation of spectral algorithm!
- Random walk ¼ running power method on L
- With constant probability, random starting node is a good starting vector for power method
- Computing eigenvector → sampling to estimate probabilities accurately
  - Combinatorial

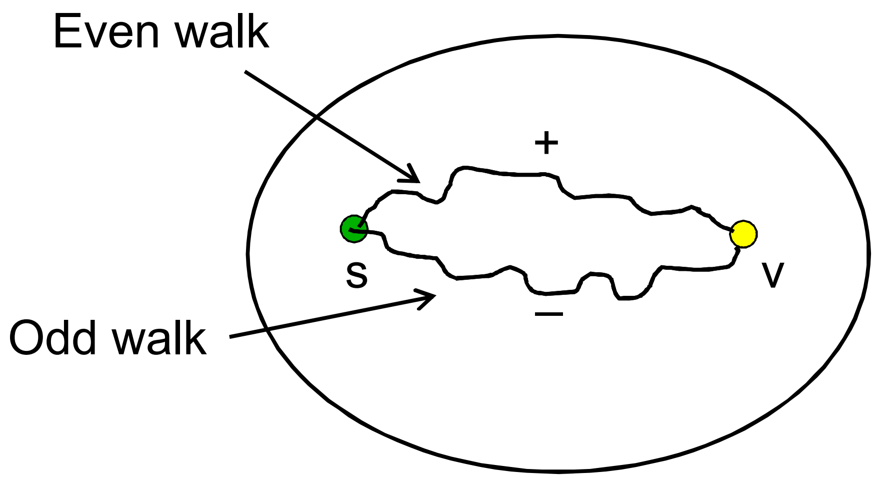
# The charged random walks





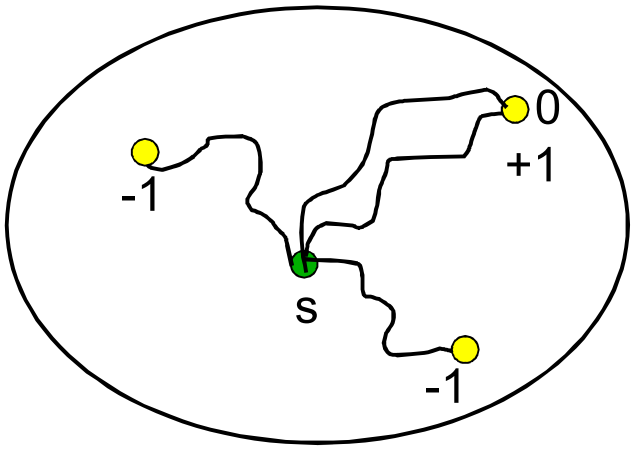
- Start with +1 charge as s
- Flip sign when we move to neighbor
- So walk deposits a unit charge at destination
- After r steps, what is expected charge at vertex v?

# The charged random walks



- Even walk contributes positive charge
- Odd walks contributes negative charge
- Expected charge at v = Pr[even walk] – Pr[odd walk]
    - Look familiar?

# The electric algorithm



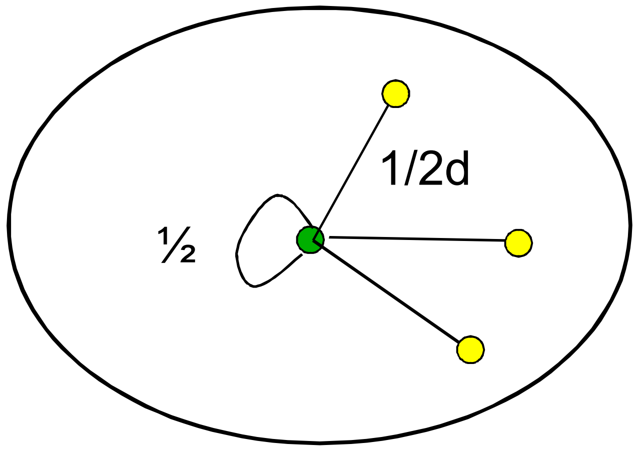
- Run w charged random walks of some length r

- Compute final charge on each vertex:

  If charge on v is very positive, v in L

  If charge on v is very negative, v in R

# A slightly different viewpoint



- Start with charge vector x. Perform one step of charged walk

$$\tilde{x}_i = \frac{x_i}{2} - \frac{1}{2d} \sum_{(i,j)\in E} x_j$$

$$\tilde{x} = \left( \frac{I}{2} - \frac{A}{2d} \right) x = \frac{\mathcal{L}x}{2}$$

After r steps of walk: $x^{(r)} = \left( \frac{\mathcal{L}}{2} \right)^r x$

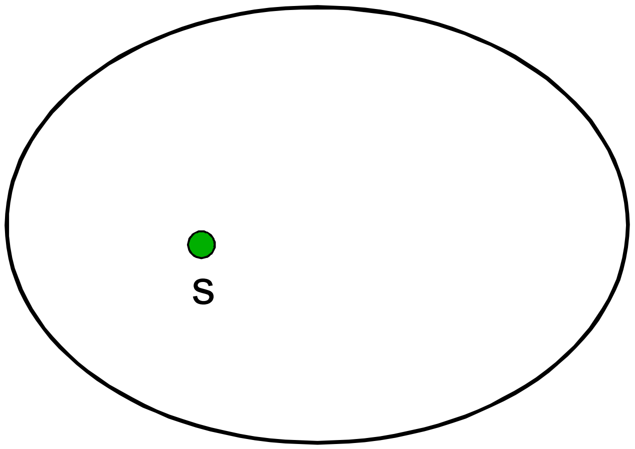
## The power method

$$x^{(r)} = \left(\frac{\mathcal{L}}{2}\right)^r x$$

- As r increases, this converges to top eigenvector of L
- Our random walk is implicitly using power method to compute top eigenvector!

- But how fast is convergence?

# The starting vector



$$x^{(r)} = \left(\frac{\mathcal{L}}{2}\right)^{r} x$$

- We start with uniform random unit vector $e_s$
- If top eigenvector is u, number of iterations to converge is

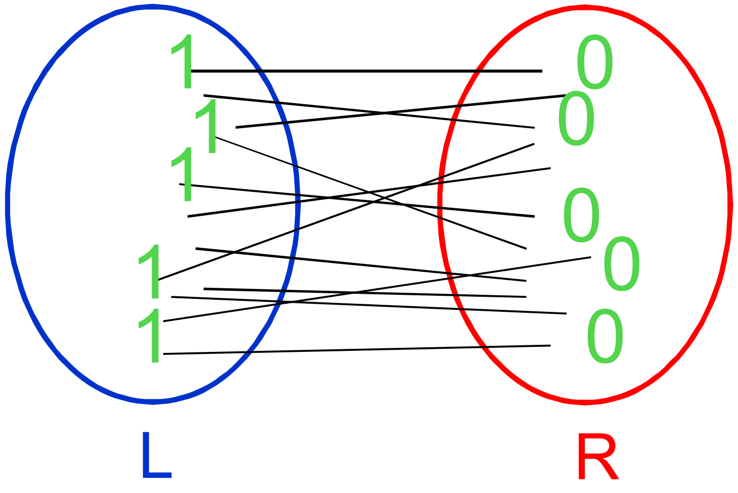
$$\log\left(\frac{1}{\langle e_s, u\rangle}\right) \longleftarrow \text{Projection of } e_s \text{ on u}$$

# Random Starting Vertex is Good

Lemma: With constant probability over s,

$$\langle e_s, u \rangle = \Omega(1/n)$$

MaxCut



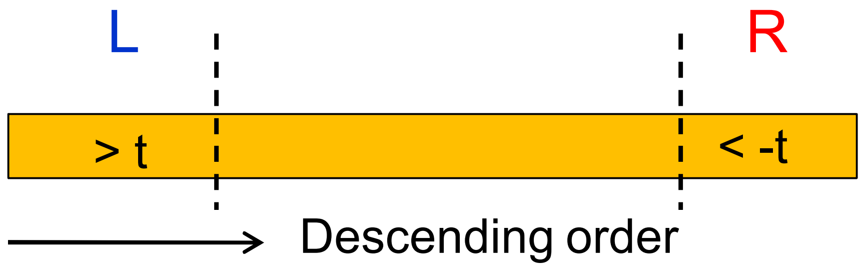
Define x = $1_L$

x has large projection on u.

By convexity, for many s in L, $e_s$ has large projection

Since |L| > n/2, we are done.

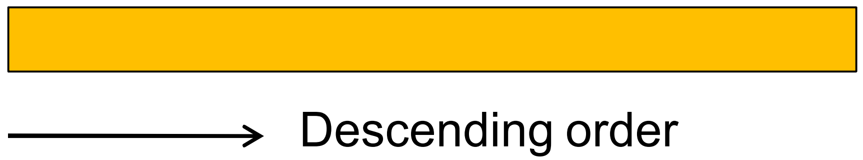
# Putting it together



$$x^{(r)} = \left(\frac{\mathcal{L}}{2}\right)^r x$$

- Charged random walk from uniform random s converges to u in O(log n) steps
- So threshold cut on final charge is like threshold on u
- But algorithm does not compute final charge vector, it *samples* from it

# Sampling from charge



$$x^{(r)} = \left(\frac{\mathcal{L}}{2}\right)^r x$$

- $x_v$ = Prob[even walk to v] – Prob[odd walk to v]
- To estimate $x_v$, need at least $1/x_v$ walks

- Lemma: If r = μ log n, then

$$\|x^{(r)}\| \geq \frac{1}{n^{1+\mu}}$$

- $n^{1+\mu}$ walks enough to get large coordinates

# The procedure Simple

- Take uniform random starting vertex s
- Run w random walks of some length r
  - $w = n^{1+\mu}$, $r = \mu \log n$
- For every vertex v:

  If [#(even walks) - #(odd walks)]/w > t, v in L

  If […]/w < t, v in R
- Recur on unclassified vertices

# Running time for Simple

- Approx. factor is $0.5 + h(\mu)$
  - Longer walks give better approx factor, but need more walks to estimate probs accurately

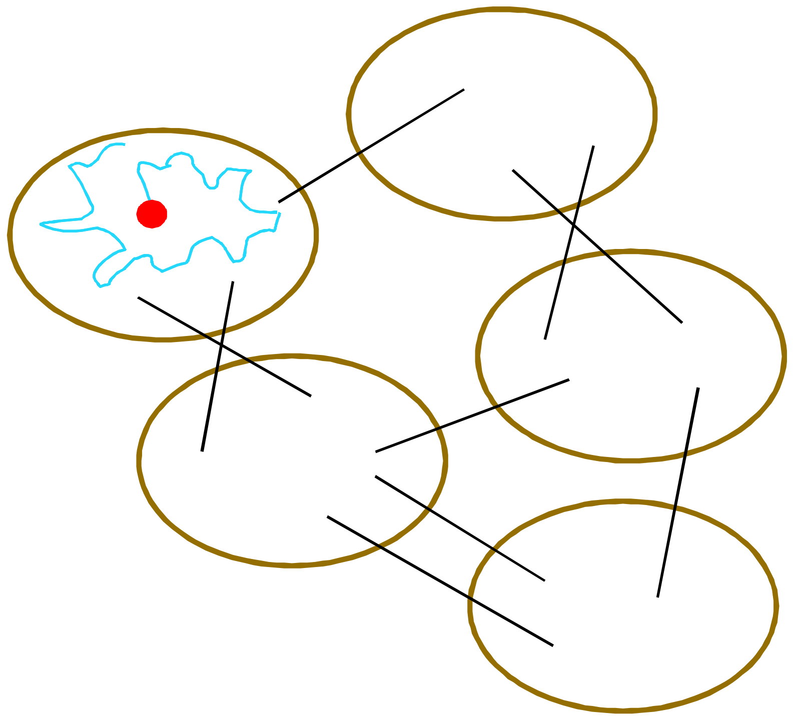
- work/output ratio for a recursive call
  = (total time)/(total nodes classified)

- In recursive call, worst case is O(1) classified vertices
  - Guaranteed to get at least constant
- Each call takes $O(n^{1+\mu})$
  - work/output ratio ¼ $n^{1+\mu}$ ) $O^*(n^{2+\mu})$ overall time

## The procedure Balance

# Improving Running Time

Bad case: G has many small connected components



Random walk gets stuck in small component

Too few nodes classified, so work/output ratio blows up

Same problem occurs if G has many components connected to each other by “sparse cuts”

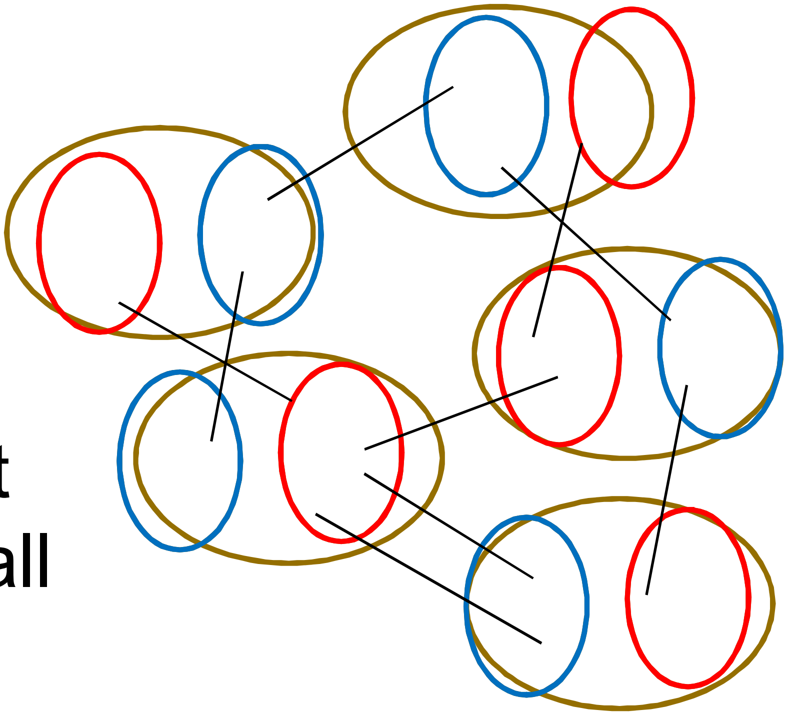
## Partitioning

Balance

Idea: Use partitioning alg to disconnect sparse cuts

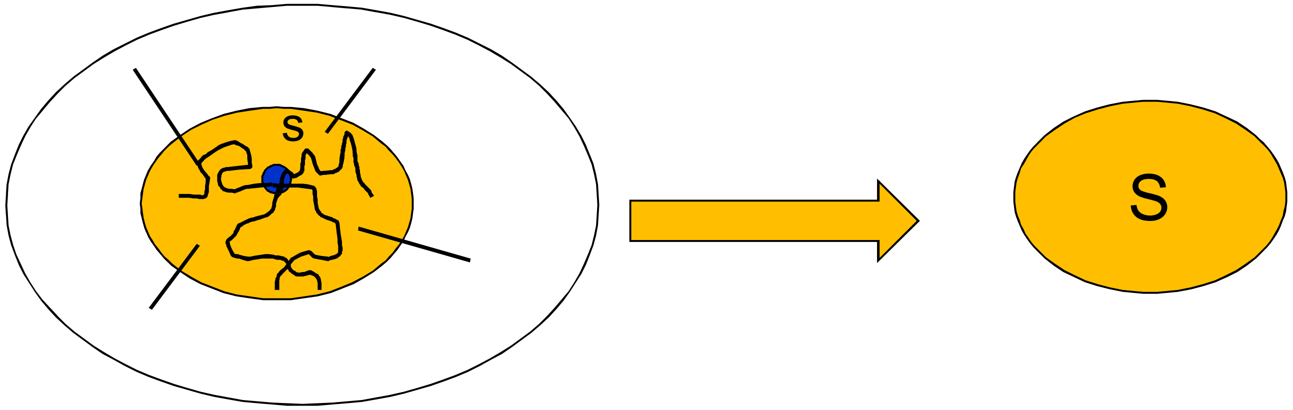
Recur on obtained pieces

Greedily merge left and right sides from each recursive call



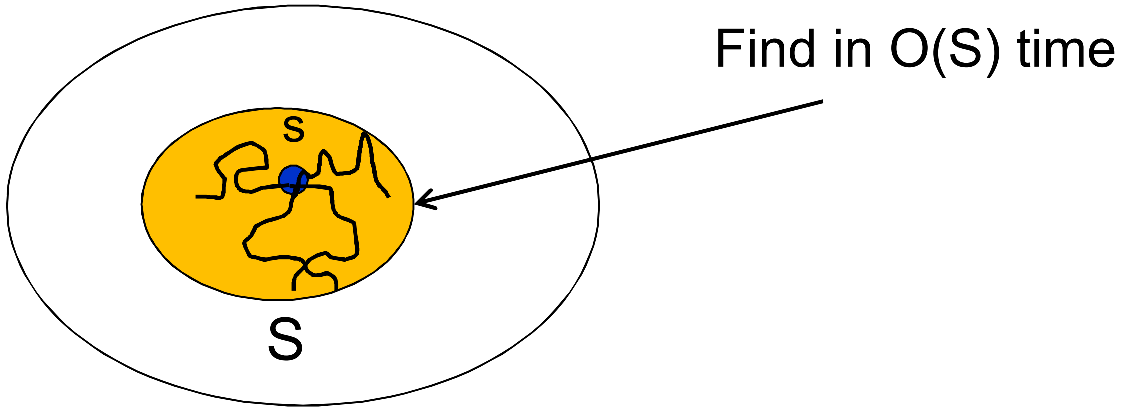
Greedy merging cuts at least half the edges in the sparse cuts ) degradation in approx factor

# Local partitioning



- Perform O(log n) length walks (as in Simple)
- [Local partitioning step] If walks are getting “stuck”, find low conductance set S around s
- Run Simple on $G_S$
  - Work/output is $S^{1+\mu}$
- If walks are not stuck, run Simple(G)

# More precisely



- Local partitioning: find a sparse cut quickly, if any exist
  - Sparse cut: #(edges in cut) < $\phi$¢#(edges in either side)
- [ST '04, ACL '06, AL '08]: (roughly) can find sparse cut if walk of length O(log$^2$(n)) gets stuck
- Our work: can find sparse cut if walk of length O(log(n)) gets stuck
  - Caveat: worse running time than previous work

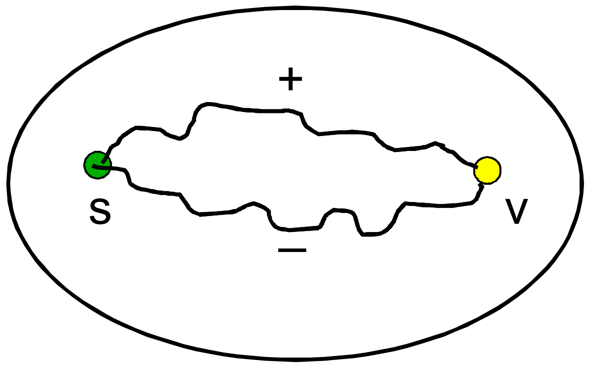
# Our contribution

- We do r-length random walks
- Let $p_v$ be prob. of reaching v
- Theorem: There is algorithm with running time $O(1/\lambda)$ that outputs either:
  - Cut of conductance $\phi$
  - Correctly declares that $\max_v p_v < \lambda$

- In time, we get $\sqrt{n}$ sized set of low conductance, or $\max p_v < 1/\sqrt{n}$
- Proven using Lovasz-Simonovits technique

# Benefit of small alpha

- $x_v$ = Pr[even walk to v] – Pr[odd walk to v]
- $p_v$ = Pr[even walk to v] + Pr[odd walk to v]
- $X_v$ is random var
  - +1 if walk is even and ends as v
  - -1 if walk is odd and ends at v
  - 0 else

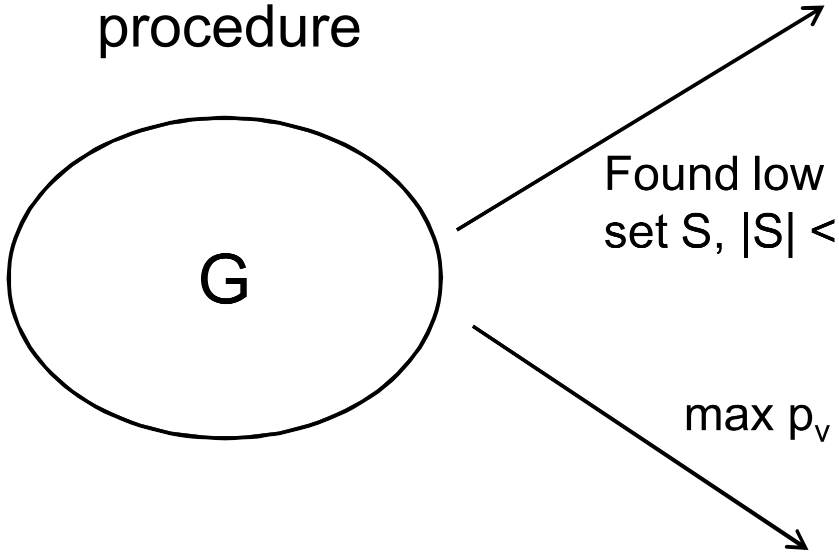


- $E[X_v] = x_v$ $\qquad E[X_v^2] = p_v$
- If $p_v$ is small, easier to get estimates for $x_v$

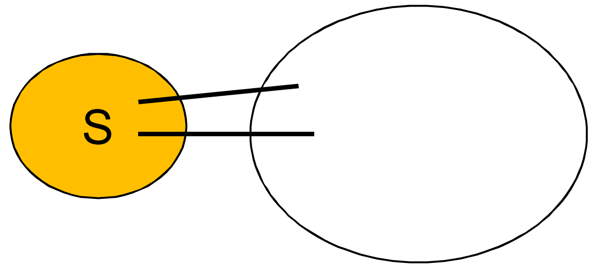
- Lemma: If max $p_v < \lambda$, work/output of Simple is $O(\lambda n)$

# Putting it together

Run partitioning procedure



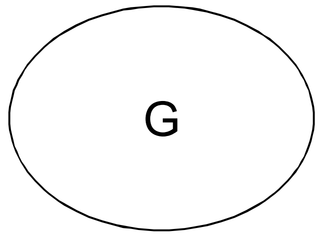
Found low cond set S, $|S| < 1/\lambda$



Remove cut
Run Simple(S)
Recur on remainder

Work/output = $|S| < 1/\lambda$

max $p_v < \lambda$



Run Simple(G)
Work/output = $\lambda n$

Balance at $\lambda = 1/\sqrt{n}$

Work/output = $\sqrt{n}$

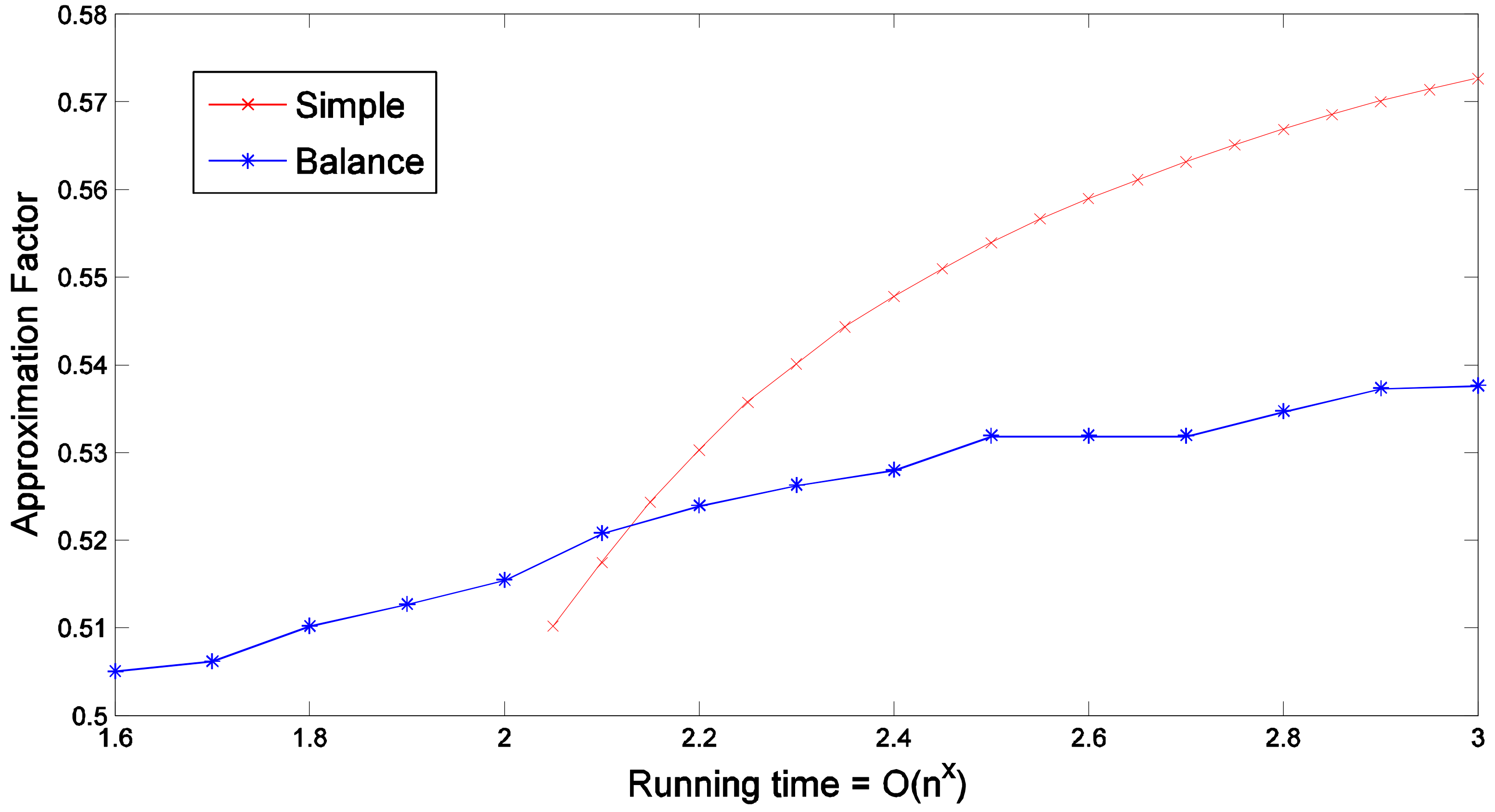
## The final algorithm

- For any $\mu > 0$, Balance runs in $O(n^{1.5+\mu})$ time with approx factor $= 0.5 + g(\mu)$

- Beats $O(n^2)$ time of eigenvalue approach (but gives much worse approx factor)

- Demonstrates power and insight of combinatorial viewpoint

# Running time/Approx Factor tradeoff



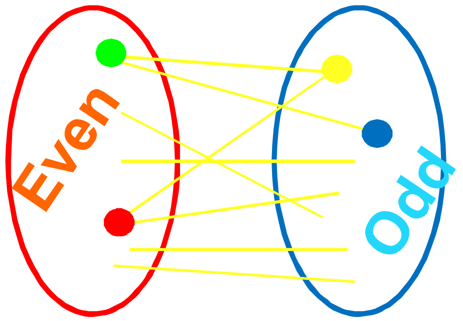
## Conclusions and Open Problems

- First combinatorial algorithm to approximate MaxCut to $> 0.5$ factor
- Open problems:
  - Near linear running time?
  - Better approx factor?
  - Similar algorithms for MaxCSP?
  - Better running time for local partitioning?

Thank you!

## Greedy++

- Natural extension: fix a walk length *l*. Count # even walks and odd walks and classify.
  - Perfectly classifies bipartite graphs
- Trouble: # walks can grow exponentially with length *l*



- Randomness to the rescue: use random walks!

## RW Heuristic ´ Spectral Algorithm

Assume: $G$ is $d$-regular. Then $L = I - (1/d)A$.

Apply power method to $(½)L$ with starting vector $e_i$

$$\left(\frac{1}{2}L\right)^{l} e_i = \left(\frac{1}{2}I - \frac{1}{2d}A\right)^{l} e_i$$

$$= \sum_{h=0}^{l} \binom{l}{l-h} (-1)^h \left(\frac{1}{2}\right)^{l-h} \left(\frac{1}{2d}A\right)^{h} e_i$$

# RW Heuristic ´ Spectral Algorithm

$$\left(\frac{1}{2}L\right)^{l} = \sum_{h=0}^{l} \overbrace{\binom{l}{l-h}(-1)^{h}\left(\frac{1}{2}\right)^{l-h}\left(\frac{1}{2d}A\right)^{h}}^{\text{Walk with } h \text{ move steps and } (l-h) \text{ self-loop steps}} e_i$$

Choice of self-loop steps

Parity of walk

Prob of *(l -h)* self-loop steps

Prob of *h* move steps

Coordinate j$= \mathbf{Pr}[i \xrightarrow{\text{even}} j] - \mathbf{Pr}[i \xrightarrow{\text{odd}} j]$

¼ [#(even walks) - #(odd walks)]/w

# Starting Vector

- Starting vector for power method influences convergence rate to top eigenvector

- Starting from unit vector $x_0$, vector after $k$ iterations of power method is $x_k = L^k x_0/kL^k x_0 k$

$\lambda_{max}$ = max eigenvalue

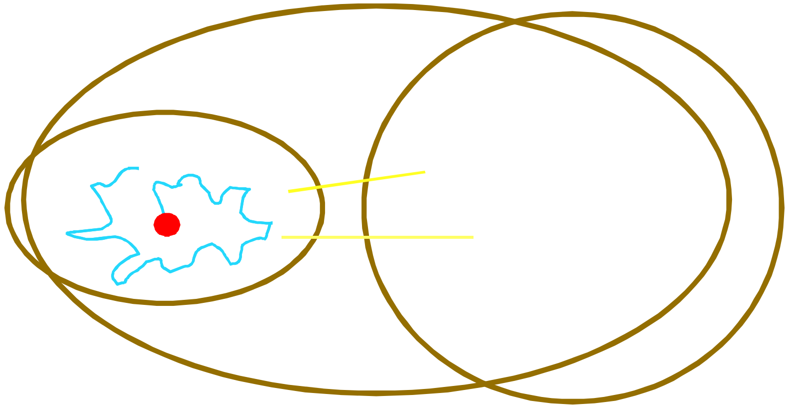
- To ensure $x_k^T L x_k$ , $(1-\varepsilon)\ \lambda_{max}$

  need $$k = O\left(\frac{\log(1/\|x_0^H\|)}{\epsilon}\right)$$

$x_0^H$ = projection of $x_0$ on subspace of top eigenvectors

# Local Partitioning

Thm: [JS '89] If a random walk from target vertex gets stuck, there must be a sparse cut close by



1. Run several random walks from target vertex

2. For each length up to *l*:
   2a. Sort vertices in decreasing order of hitting frequency
   2b. Check each prefix of for sparse cut

# Running time for Balance

- Local partitioning result: for any $\alpha > 0$, in $O^*(1/\alpha)$ time can find sparse cut or walk mixes sufficiently so that work/output $\cdot$ $O^*(\alpha n^{1+\mu})$
- Balancing point: $\alpha = n^{0.5+\mu/2}$, so that work/output $\cdot$ $O^*(n^{0.5+\mu/2})$
- Overall running time: $O^*(n^{1.5+\mu/2})$
- Approx factor = $0.5 + h'(\mu)$