Explicit Expanding Expanders

Michael Dinitz
Johns Hopkins University

Michael Schapira and Asaf Valadarsky
Hebrew University of Jerusalem

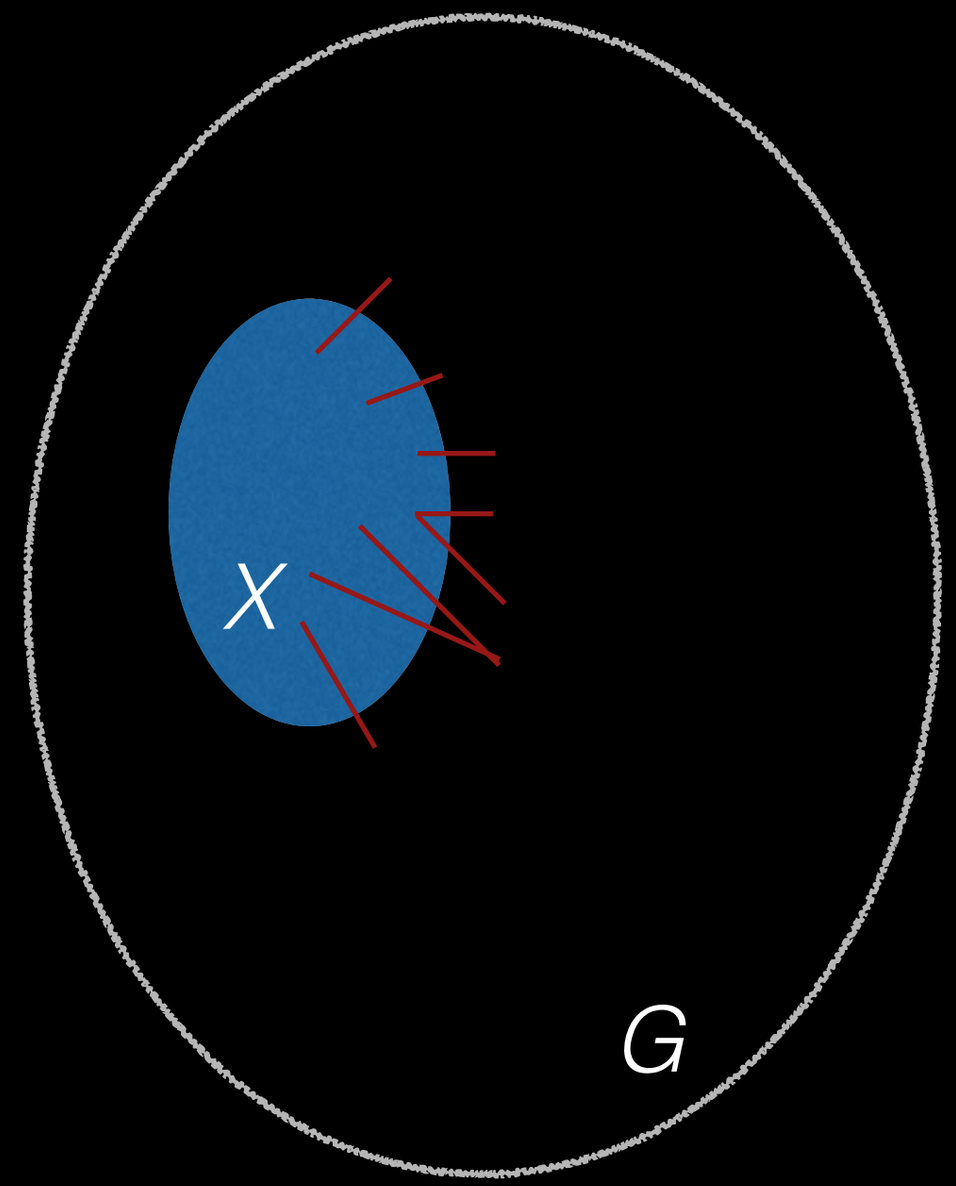
Expander Graphs

Expander Graphs

- Edge expansion:

Expander Graphs

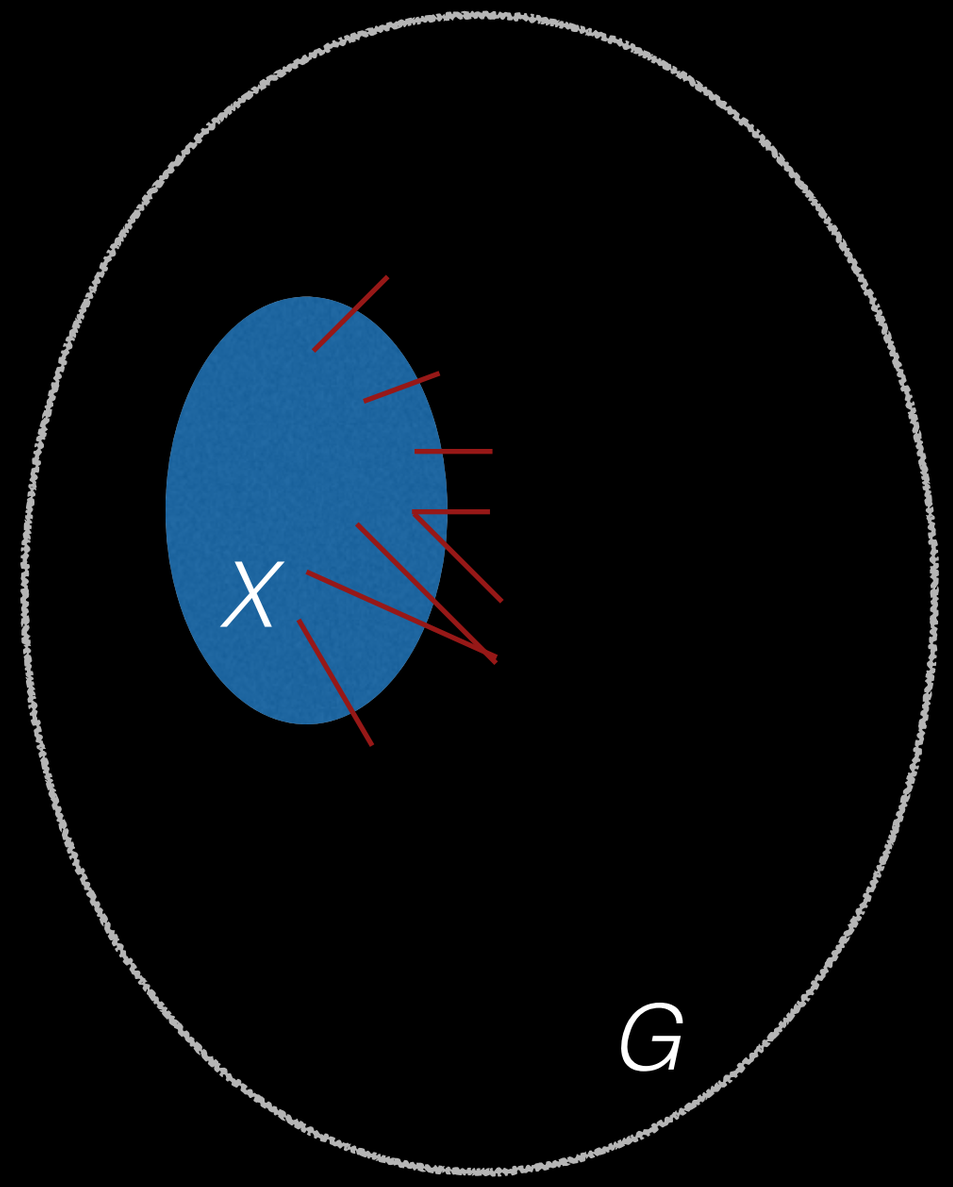
- Edge expansion:



Expander Graphs

- Edge expansion:

$$h(G) = \min_{X \subset V : |X| \leq n/2} \frac{|E(X, V \setminus X)|}{|X|}$$

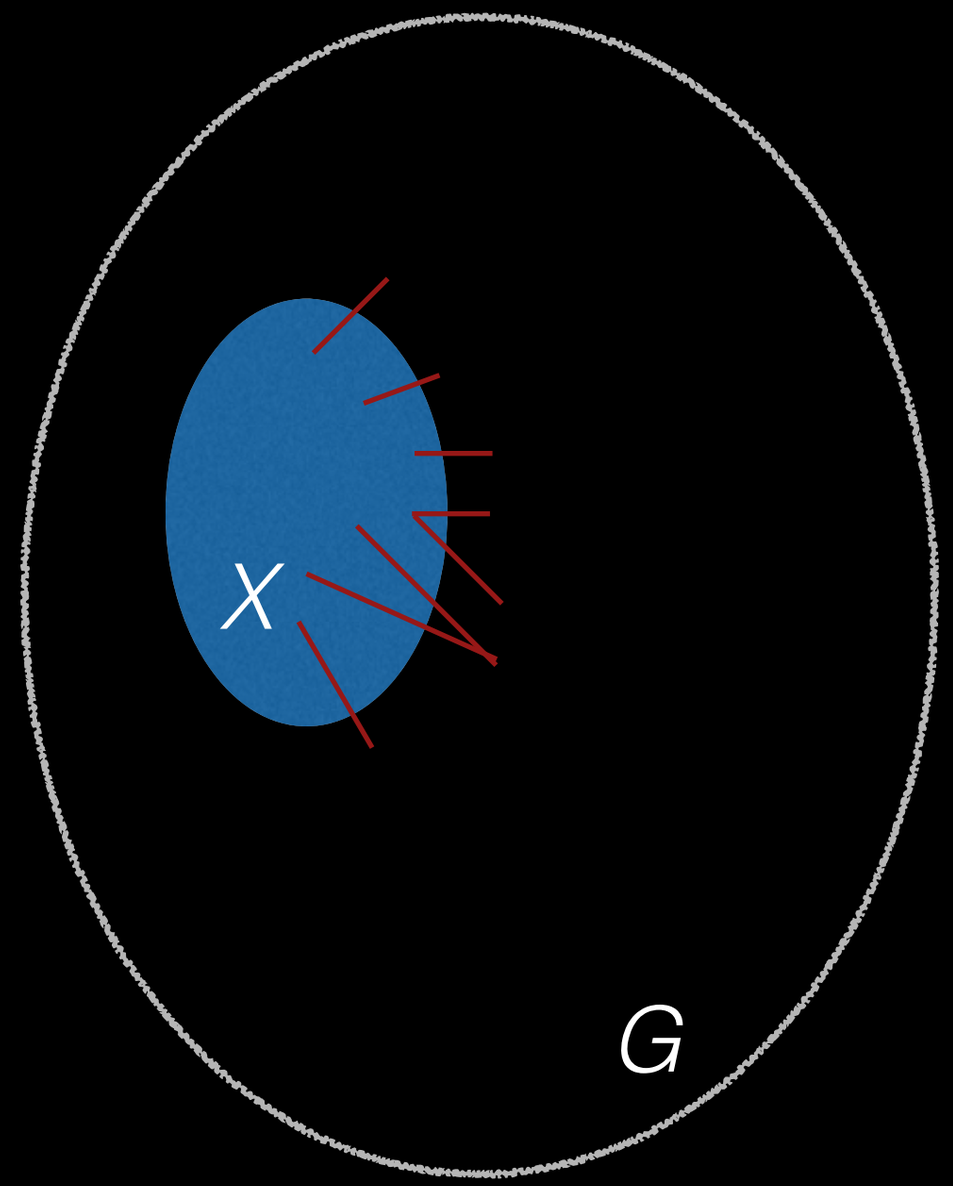


Expander Graphs

- Edge expansion:

$$h(G) = \min_{X \subset V : |X| \leq n/2} \frac{|E(X, V \setminus X)|}{|X|}$$

- In a d-regular graph, would like expansion as close to $d/2$ as possible



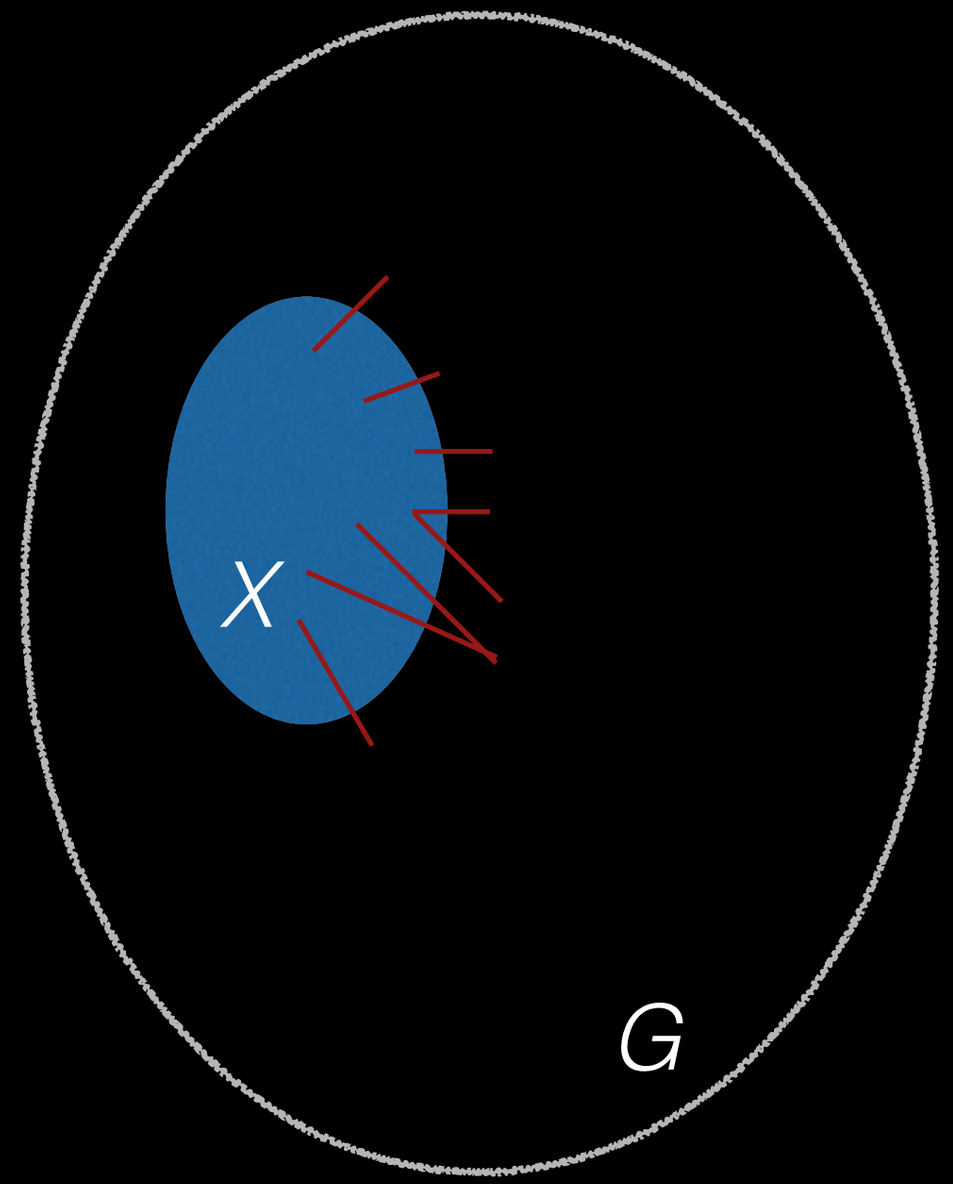
Expander Graphs

- Edge expansion:

$$h(G) = \min_{X \subset V : |X| \leq n/2} \frac{|E(X, V \setminus X)|}{|X|}$$

- In a *d*-regular graph, would like expansion as close to *d/2* as possible
- Expander: graph with $h(G) \geq \Omega(1)$
- Spectral gap: $d - \lambda_2$
- Related by Cheeger inequalities:

$$\frac{d - \lambda_2}{2} \leq h(G) \leq \sqrt{2d(d - \lambda_2)}$$



Expanders: History

Expanders: History

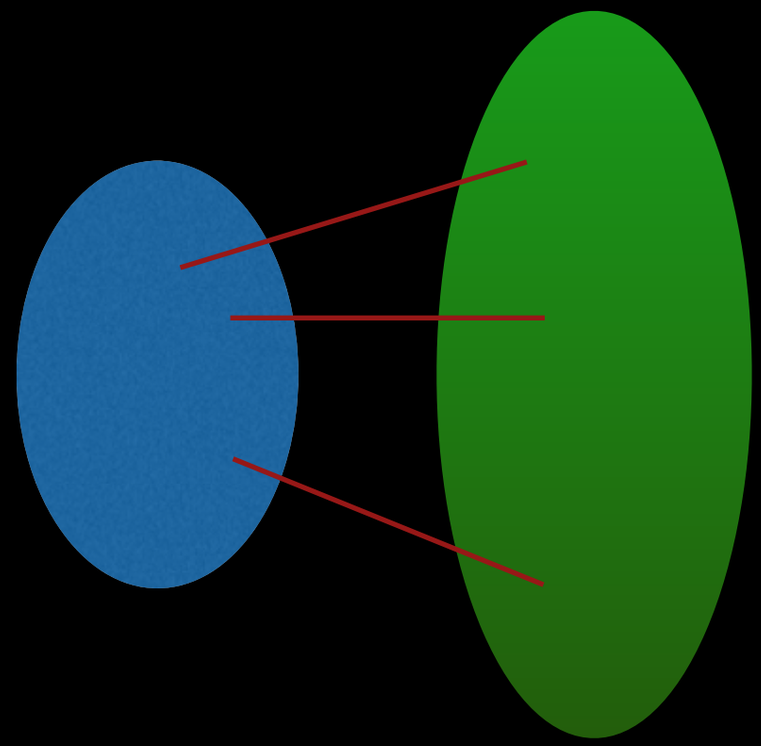
- Widely studied in graph theory / theoretical CS
- Many, many applications (mostly complexity theory)

Expanders: History

- Widely studied in graph theory / theoretical CS
- Many, many applications (mostly complexity theory)
- Random graphs are (w.h.p) very good expanders
- Surprisingly difficult to construct expanders deterministically

Data Centers

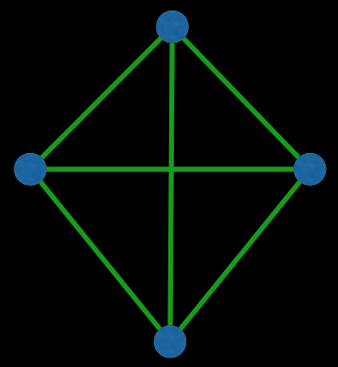
- Lots of traffic between nodes
- In a bad topology, might get “stuck”
- Problem if lots of traffic from one section to the rest, not much capacity
- Lots of traffic everywhere, so traffic proportional to # vertices
- Really: want large (edge) expansion!
- Regular graph (# ports at switches)



Incremental Expansion

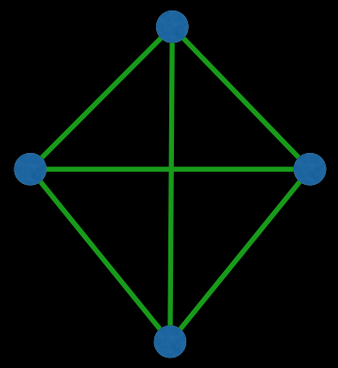
- So let's use expanders for our data centers!
- Data centers grow regularly: more servers and racks purchased and added
- Don't want to completely rewire network every time!
- Expander on *n* nodes should have approximately same edge set as expander on *n+1* nodes

Random
[Jellyfish SHPG NSDI'12]



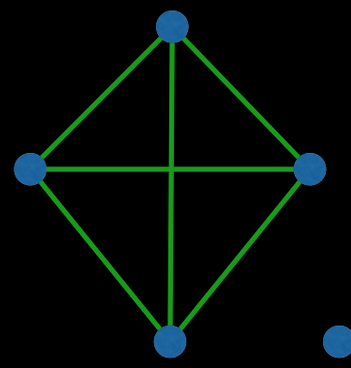
Random [Jellyfish SHPG NSDI'12]

- Construct expanders randomly
- To add node: choose random d/2 matching, remove, connect to new node



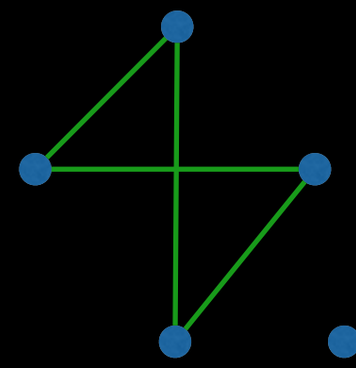
Random [Jellyfish SHPG NSDI'12]

- Construct expanders randomly
- To add node: choose random d/2 matching, remove, connect to new node



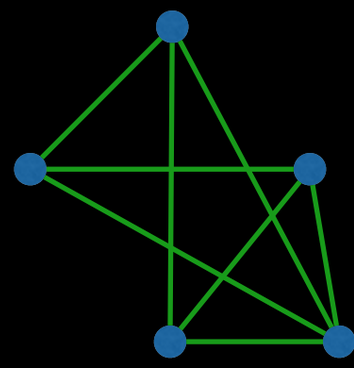
Random [Jellyfish SHPG NSDI'12]

- Construct expanders randomly
- To add node: choose random d/2 matching, remove, connect to new node



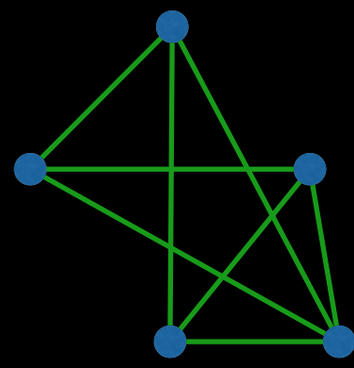
Random [Jellyfish SHPG NSDI'12]

- Construct expanders randomly
- To add node: choose random d/2 matching, remove, connect to new node



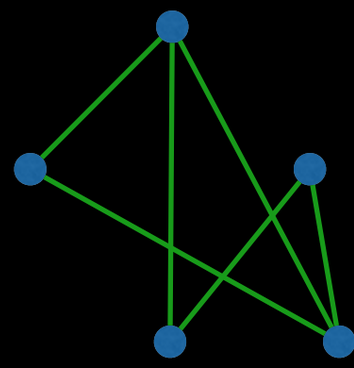
Random [Jellyfish SHPG NSDI'12]

- Construct expanders randomly
- To add node: choose random d/2 matching, remove, connect to new node



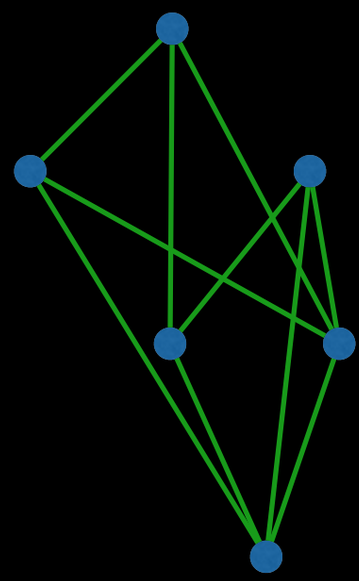
Random [Jellyfish SHPG NSDI'12]

- Construct expanders randomly
- To add node: choose random d/2 matching, remove, connect to new node



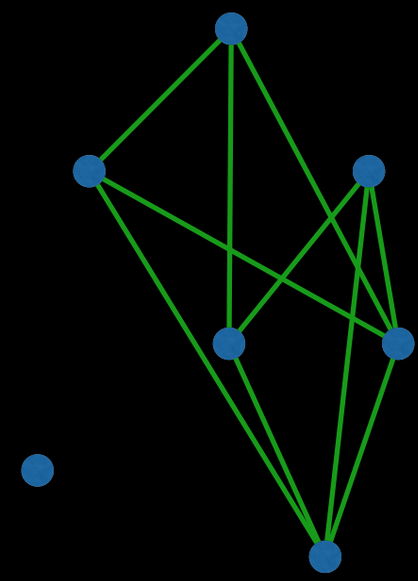
Random [Jellyfish SHPG NSDI'12]

- Construct expanders randomly
- To add node: choose random d/2 matching, remove, connect to new node



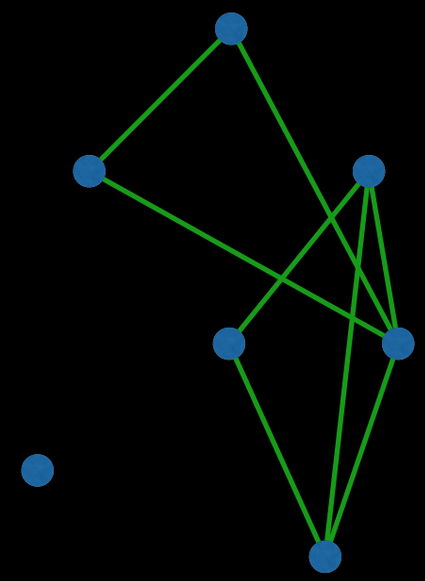
Random [Jellyfish SHPG NSDI'12]

- Construct expanders randomly
- To add node: choose random d/2 matching, remove, connect to new node



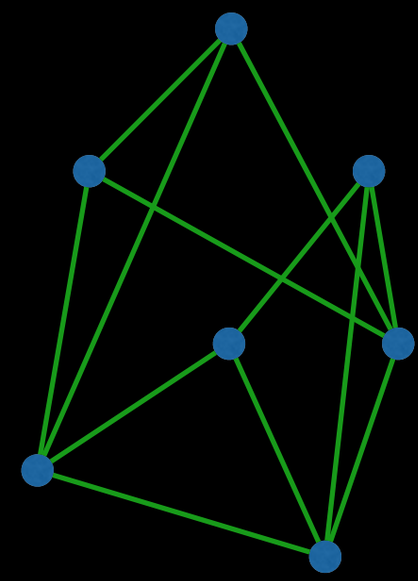
Random [Jellyfish SHPG NSDI'12]

- Construct expanders randomly
- To add node: choose random d/2 matching, remove, connect to new node



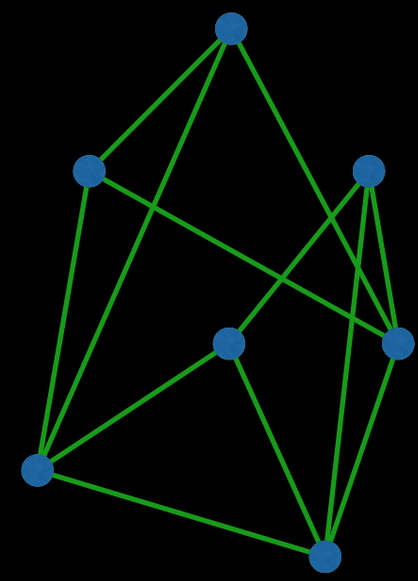
Random
[Jellyfish SHPG NSDI'12]

- Construct expanders randomly
- To add node: choose random d/2 matching, remove, connect to new node



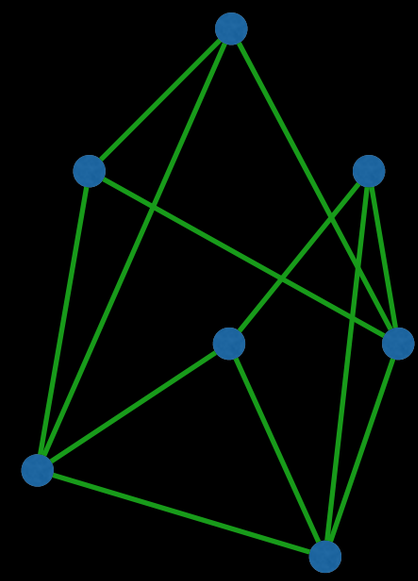
Random [Jellyfish SHPG NSDI'12]

- Construct expanders randomly
- To add node: choose random d/2 matching, remove, connect to new node
- Works great in simulation — only theory for uniform random regular graphs (Bollobas)



Random [Jellyfish SHPG NSDI'12]

- Construct expanders randomly
- To add node: choose random d/2 matching, remove, connect to new node



- Works great in simulation — only theory for uniform random regular graphs (Bollobas)
- Will companies actually use random datacenters?
- Can we get same guarantees with deterministic constructions?

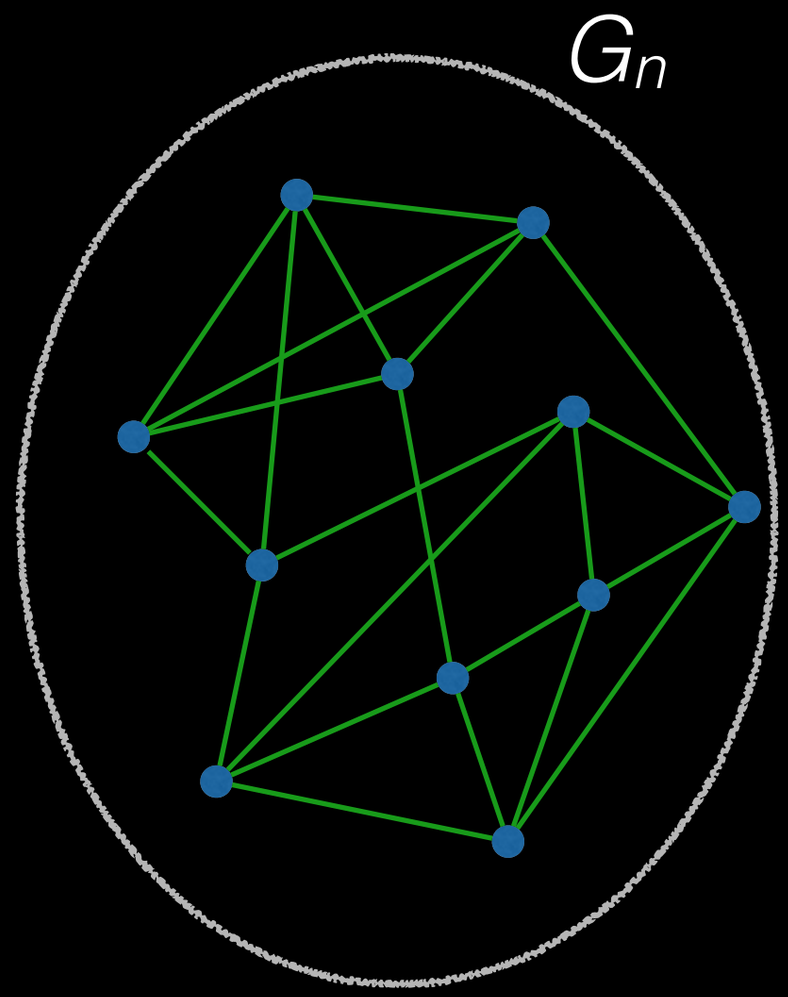
Expanding Expanders

Expanding Expanders

- Two problems with using existing deterministic expanders as data centers
    1. Need to exist for all *n* (not just primes, powers of 2, etc.)
    2. Need to handle **incremental expansion**

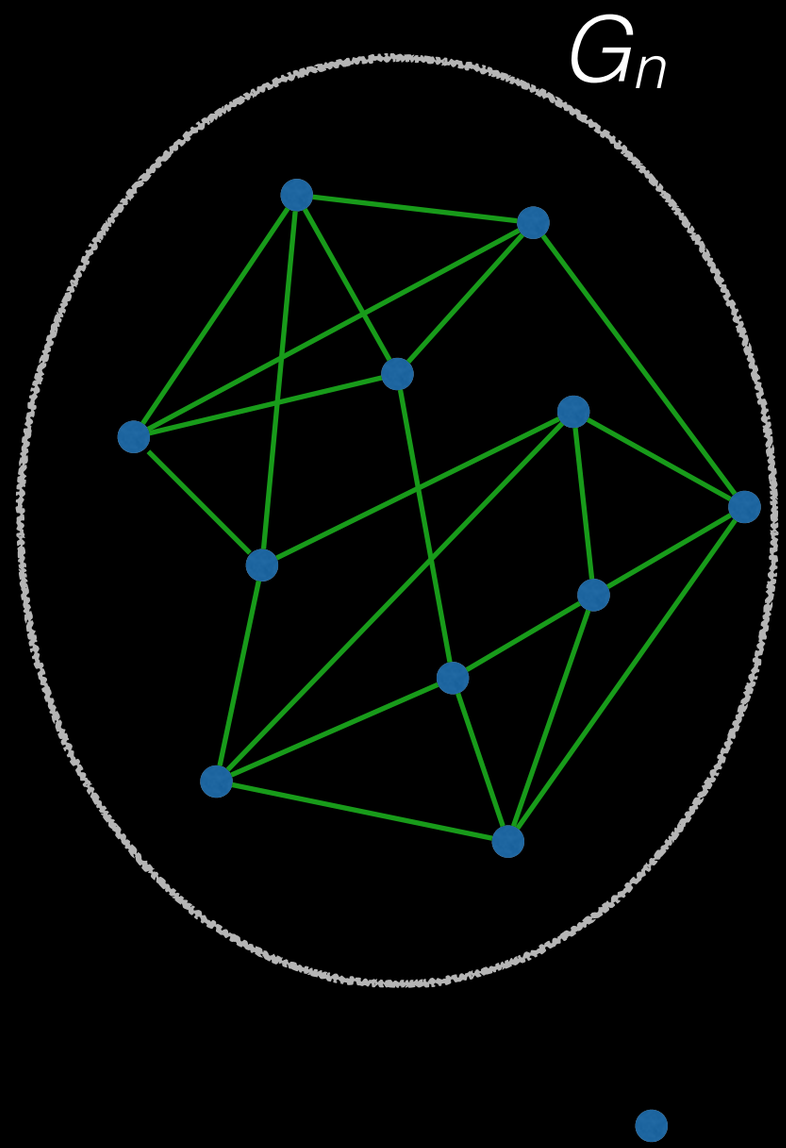
Expanding Expanders

- Two problems with using existing deterministic expanders as data centers
    1. Need to exist for all *n* (not just primes, powers of 2, etc.)
    2. Need to handle **incremental expansion**



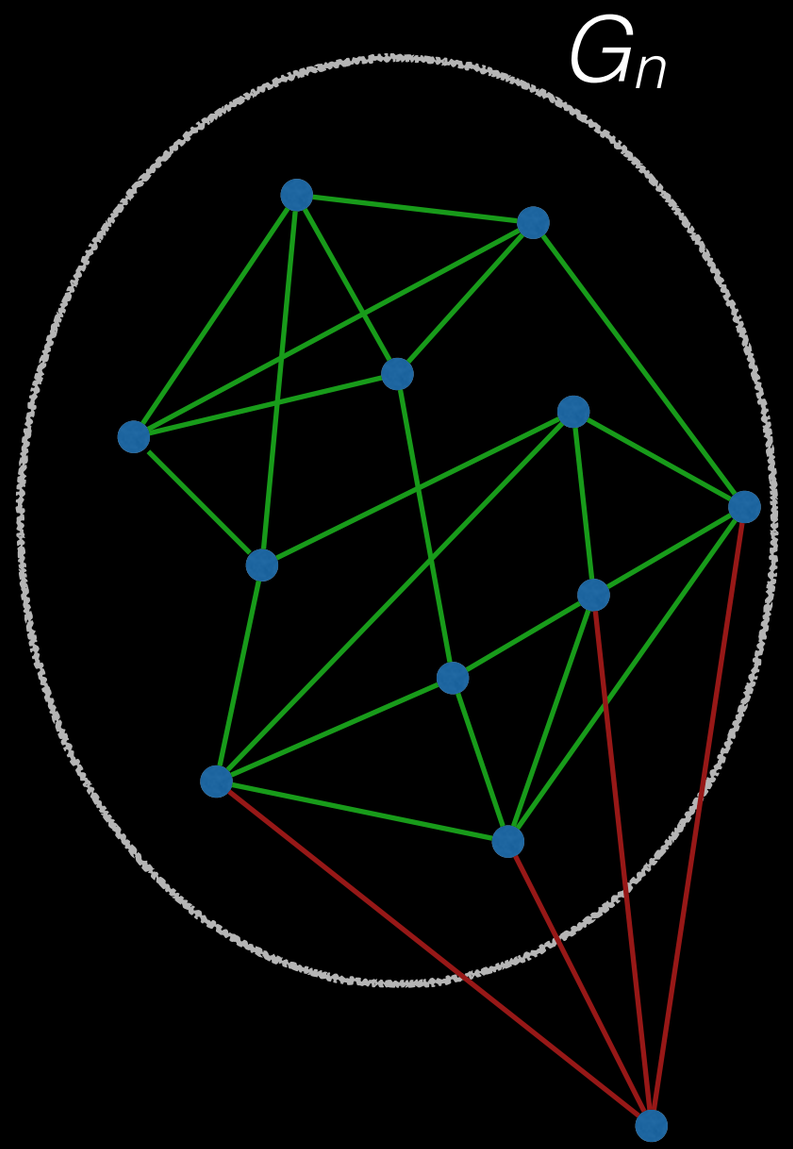
Expanding Expanders

- Two problems with using existing deterministic expanders as data centers
    1. Need to exist for all *n* (not just primes, powers of 2, etc.)
    2. Need to handle **incremental expansion**



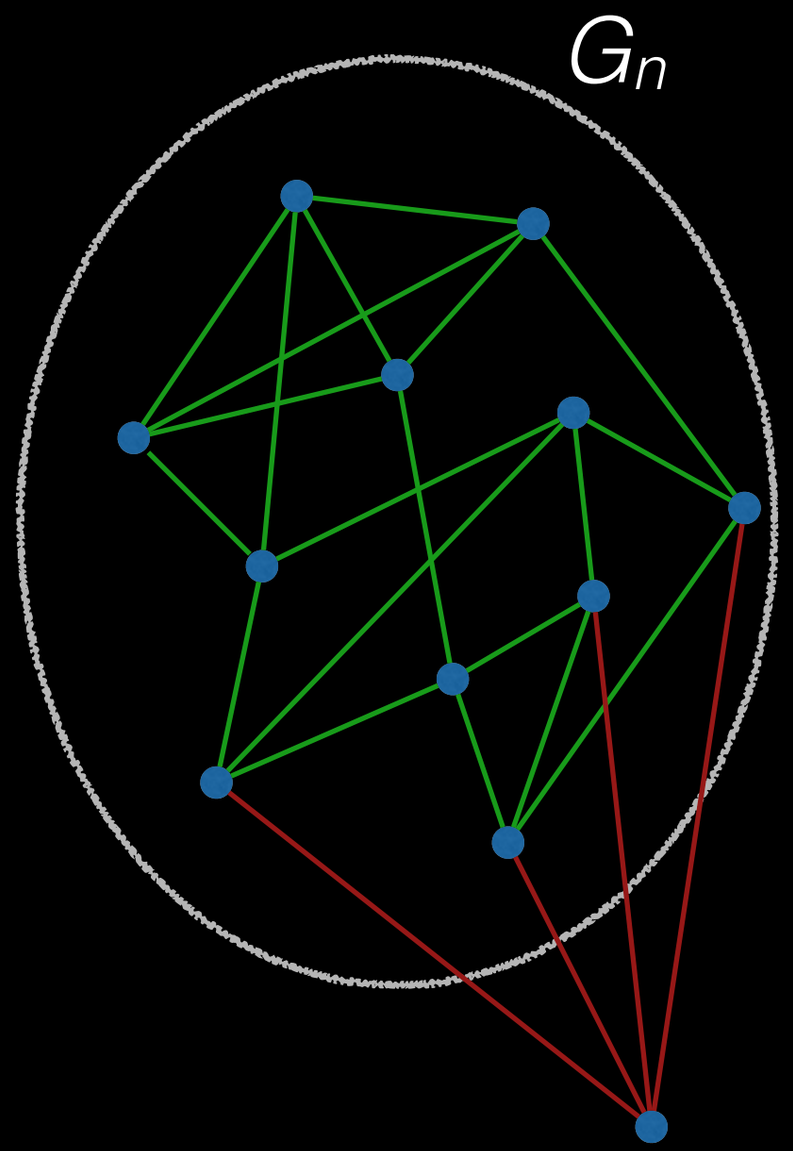
Expanding Expanders

- Two problems with using existing deterministic expanders as data centers
    1. Need to exist for all *n* (not just primes, powers of 2, etc.)
    2. Need to handle **incremental expansion**



Expanding Expanders

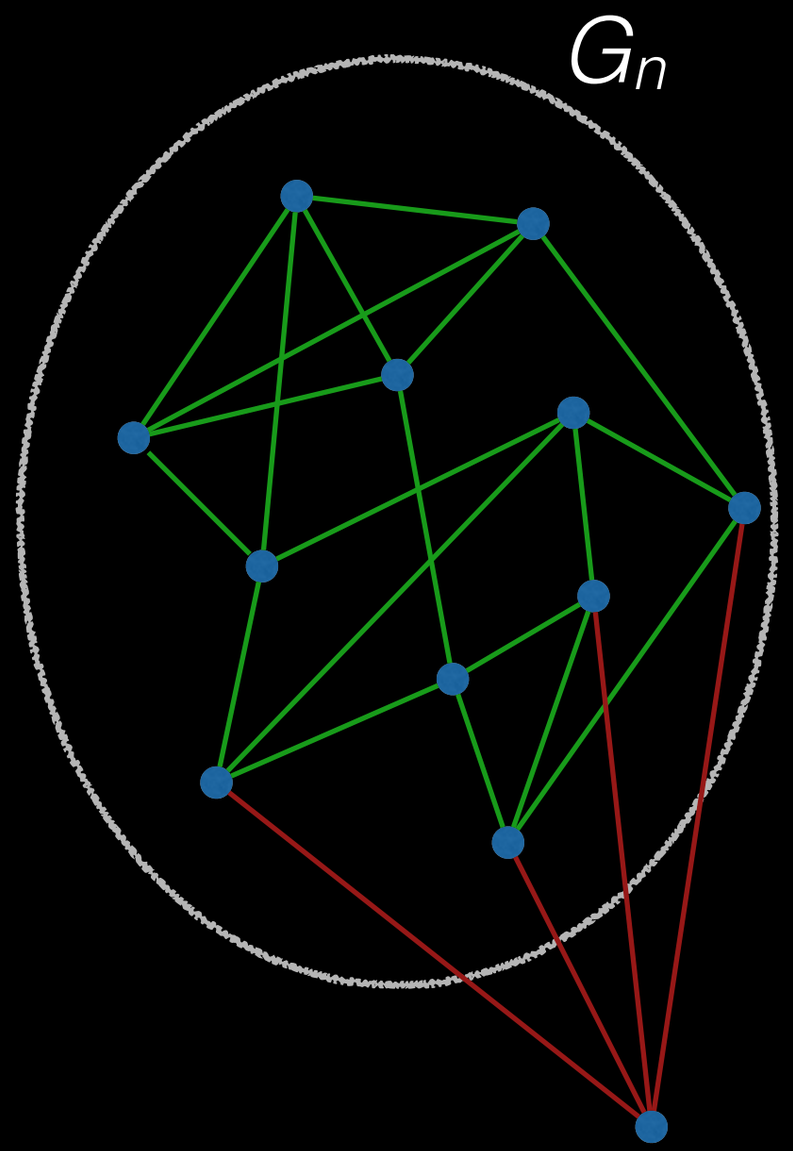
- Two problems with using existing deterministic expanders as data centers
    1. Need to exist for all *n* (not just primes, powers of 2, etc.)
    2. Need to handle **incremental expansion**



Expanding Expanders

- Two problems with using existing deterministic expanders as data centers
  1. Need to exist for all *n* (not just primes, powers of 2, etc.)
  2. Need to handle **incremental expansion**

- Goal: infinite series of *d*-regular graphs G_{d+1}, G_{d+2}, G_{d+3}, … where:
  1. G_i has *i* nodes
  2. Each G_i has large edge expansion (approx. *d/2*)
  3. Few edge changes to get from G_i to G_{i+1} (approx. *3d/2*)



Explicit Expanding Expanders

Main Result: graphs G_i where:

- G_i has i nodes
- Expansion approx. $d/3$
- At most $5d/2$ edge changes from G_i to G_{i+1}

Explicit Expanding Expanders

Main Result: graphs G_i where:

- G_i has i nodes
- Expansion approx. $d/3$
- At most $5d/2$ edge changes from G_i to G_{i+1}

- Still room for improvement!
- Technicality: use multiple edges / edge weights

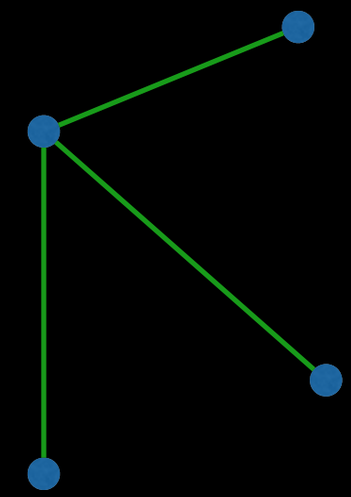
2-Lifts

2-Lifts

- Main tool: 2-Lifts [Bilu-Linial]
- 2-Lift of G: double every vertex, replace edge by matching

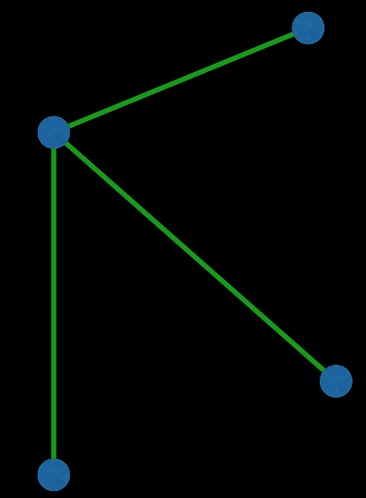
2-Lifts

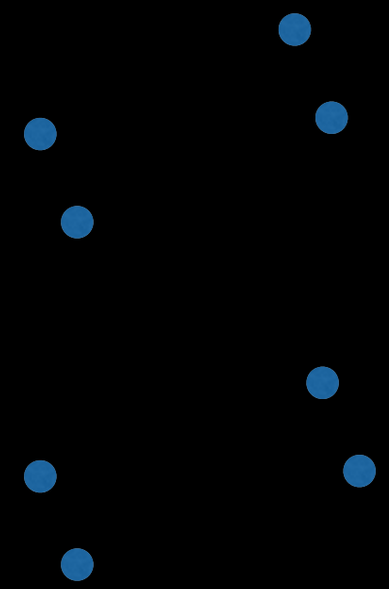
- Main tool: 2-Lifts [Bilu-Linial]
- 2-Lift of *G*: double every vertex, replace edge by matching



2-Lifts

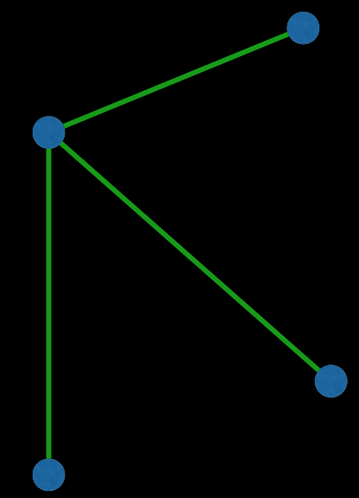
- Main tool: 2-Lifts [Bilu-Linial]
- 2-Lift of *G*: double every vertex, replace edge by matching





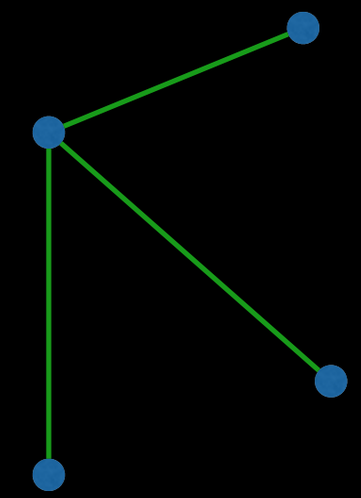
2-Lifts

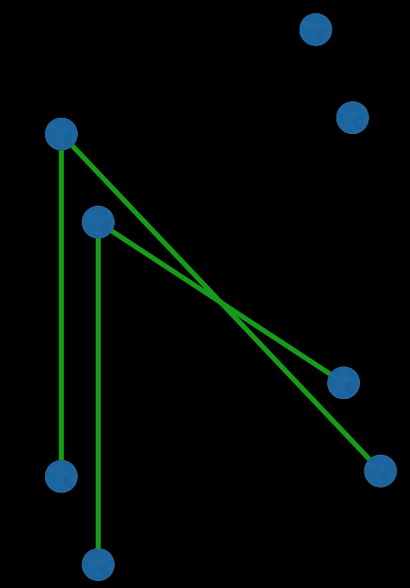
- Main tool: 2-Lifts [Bilu-Linial]
- 2-Lift of *G*: double every vertex, replace edge by matching



2-Lifts

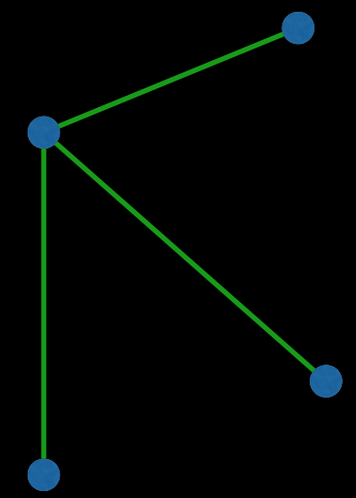
- Main tool: 2-Lifts [Bilu-Linial]
- 2-Lift of G: double every vertex, replace edge by matching

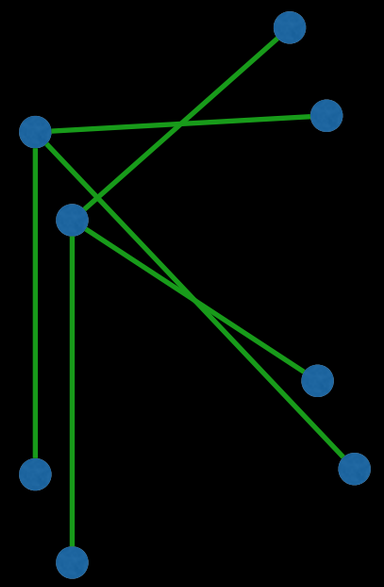




2-Lifts

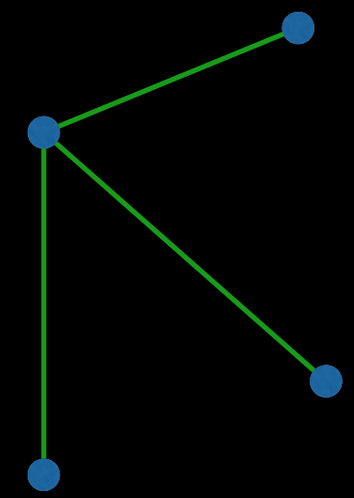
- Main tool: 2-Lifts [Bilu-Linial]
- 2-Lift of *G*: double every vertex, replace edge by matching

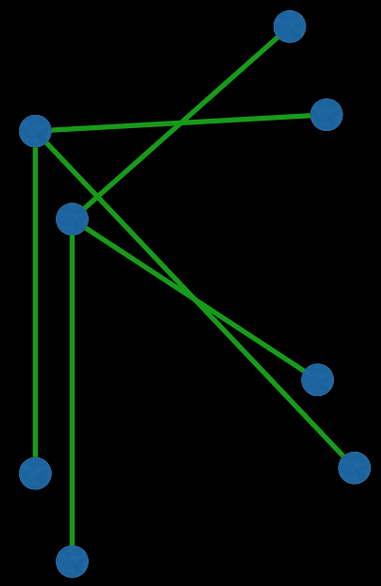




2-Lifts

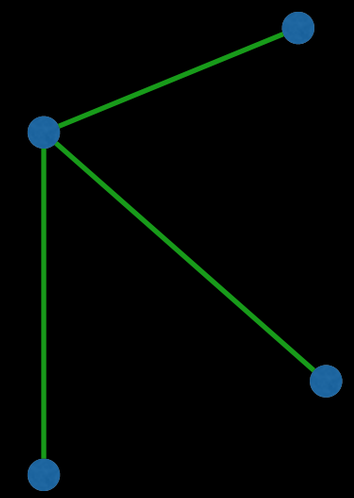
- Main tool: 2-Lifts [Bilu-Linial]
- 2-Lift of G: double every vertex, replace edge by matching
- Two options for each matching, so $2^{|E|}$ possible 2-lifts

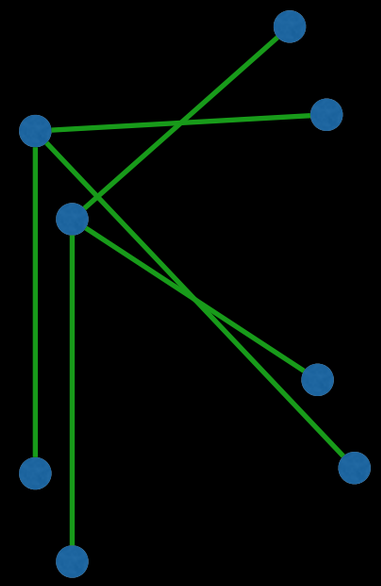




2-Lifts

- Main tool: 2-Lifts [Bilu-Linial]
- 2-Lift of G: double every vertex, replace edge by matching
- Two options for each matching, so $2^{|E|}$ possible 2-lifts
- Thm [BL]: If G an expander, random matchings gives good expander w.h.p.
- Can be derandomized!





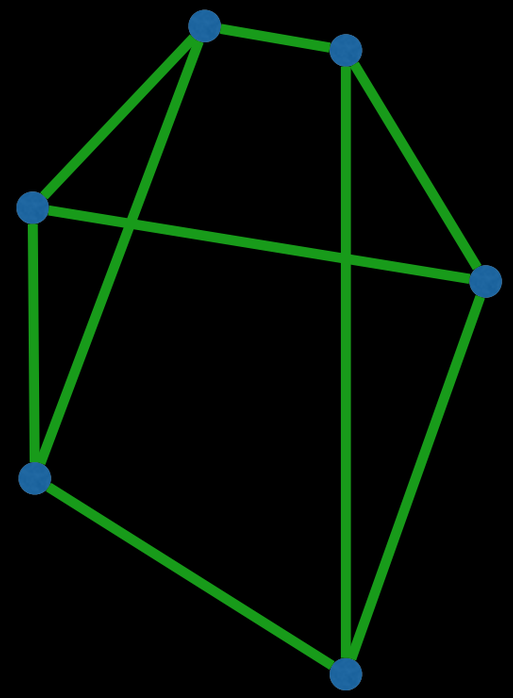
Our Approach

Our Approach

- "Split" each node one at a time, rather than all at once
- Start with *d/2*-regular expander, weights *2*

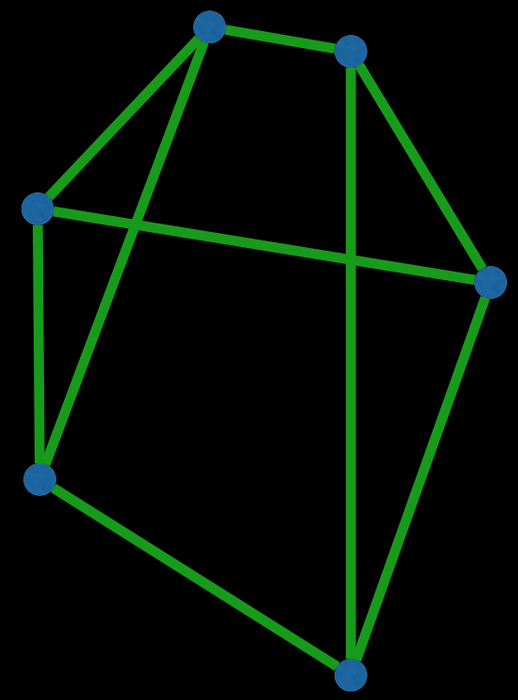
Our Approach

- “Split” each node one at a time, rather than all at once
- Start with *d/2*-regular expander, weights *2*



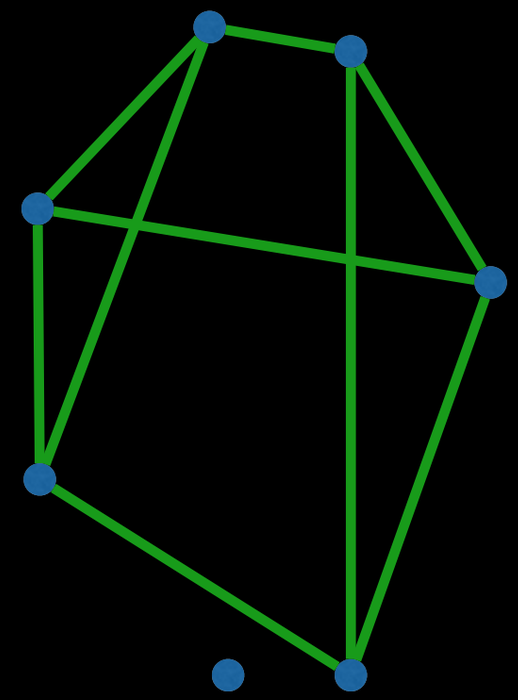
Our Approach

- "Split" each node one at a time, rather than all at once
- Start with *d/2*-regular expander, weights *2*
- Inserting new node: split currently unsplit node



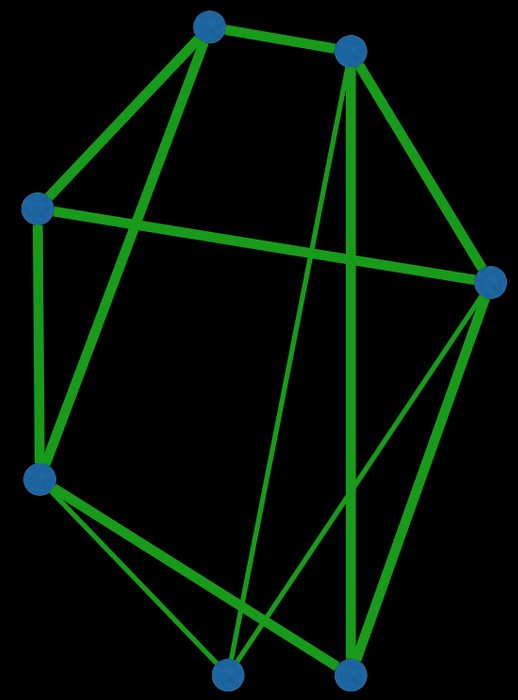
Our Approach

- "Split" each node one at a time, rather than all at once
- Start with *d/2*-regular expander, weights *2*
- Inserting new node: split currently unsplit node



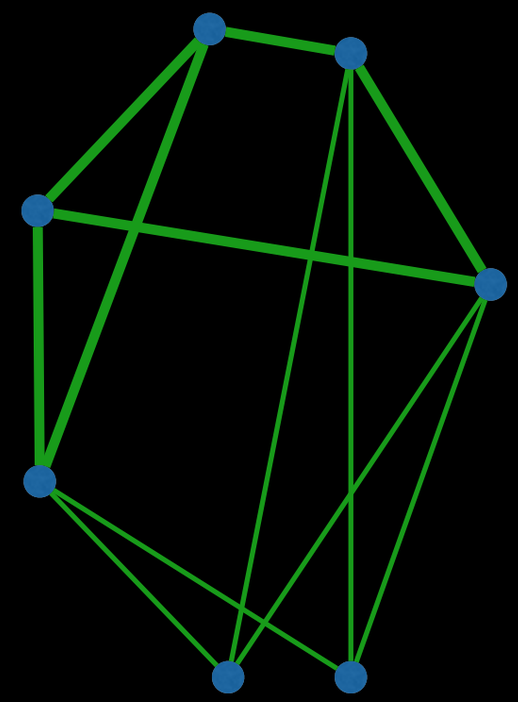
Our Approach

- "Split" each node one at a time, rather than all at once
- Start with *d/2*-regular expander, weights *2*
- Inserting new node: split currently unsplit node



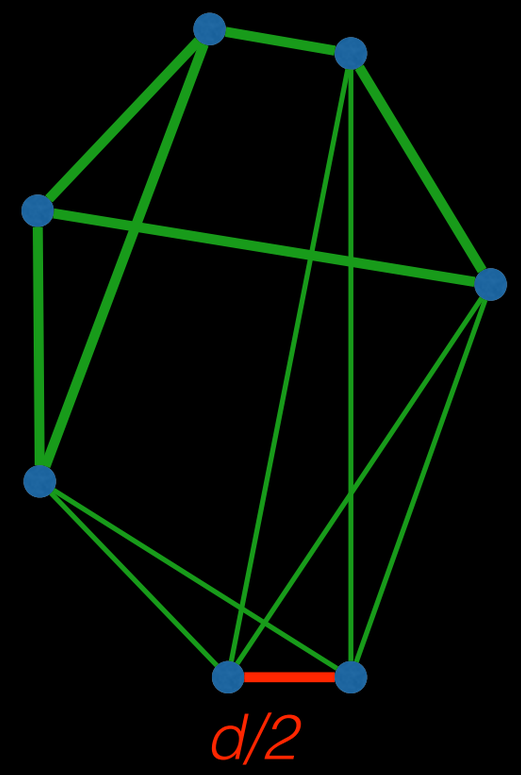
Our Approach

- "Split" each node one at a time, rather than all at once
- Start with *d/2*-regular expander, weights *2*
- Inserting new node: split currently unsplit node



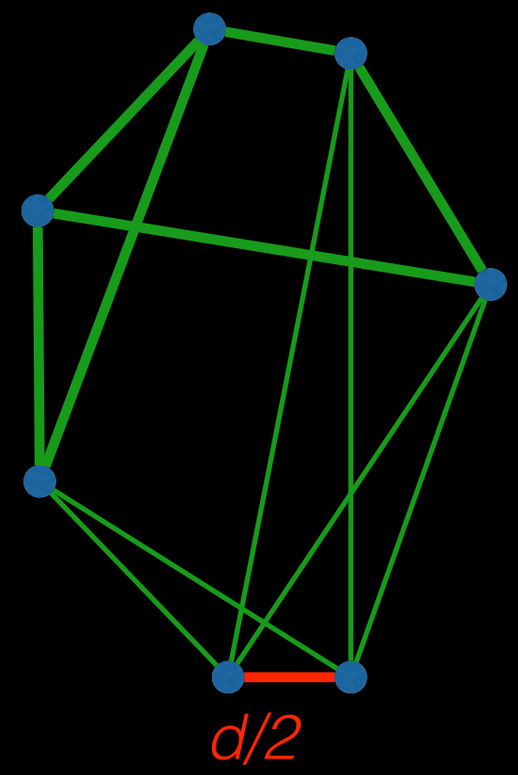
Our Approach

- "Split" each node one at a time, rather than all at once
- Start with *d/2*-regular expander, weights *2*
- Inserting new node: split currently unsplit node



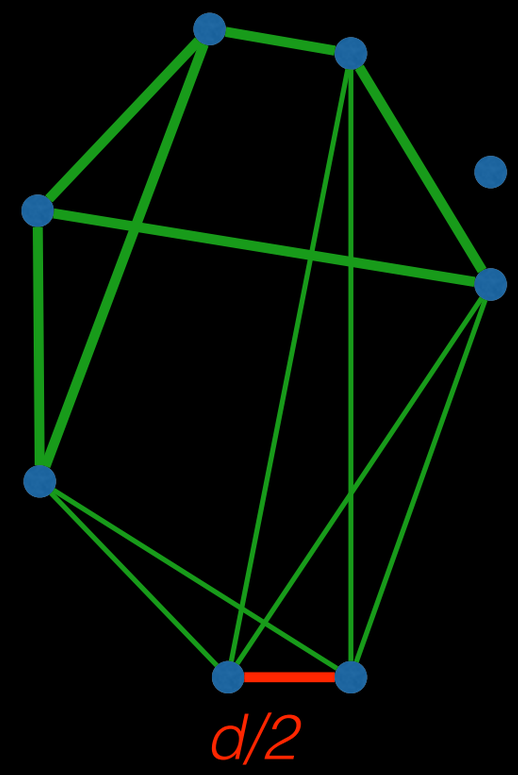
Our Approach

- "Split" each node one at a time, rather than all at once
- Start with *d/2*-regular expander, weights *2*
- Inserting new node: split currently unsplit node
  - Unsplit neighbors: replace weight *2* edge with two weight *1* edges



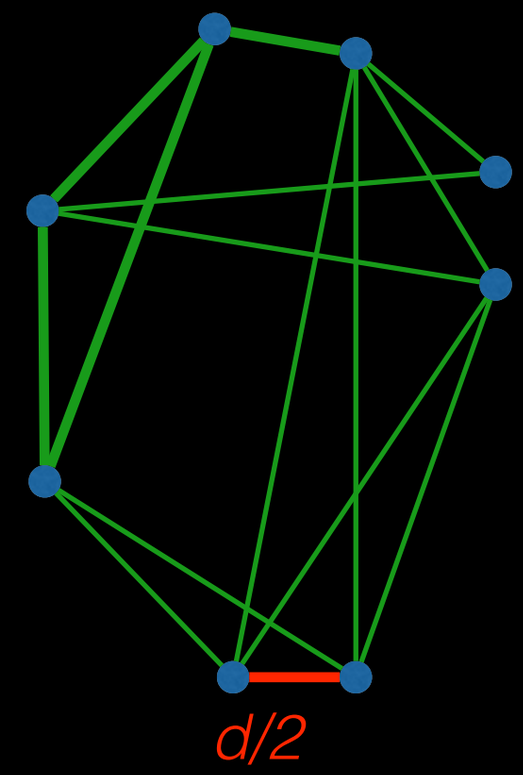
Our Approach

- "Split" each node one at a time, rather than all at once
- Start with *d/2*-regular expander, weights *2*
- Inserting new node: split currently unsplit node
  - Unsplit neighbors: replace weight *2* edge with two weight *1* edges



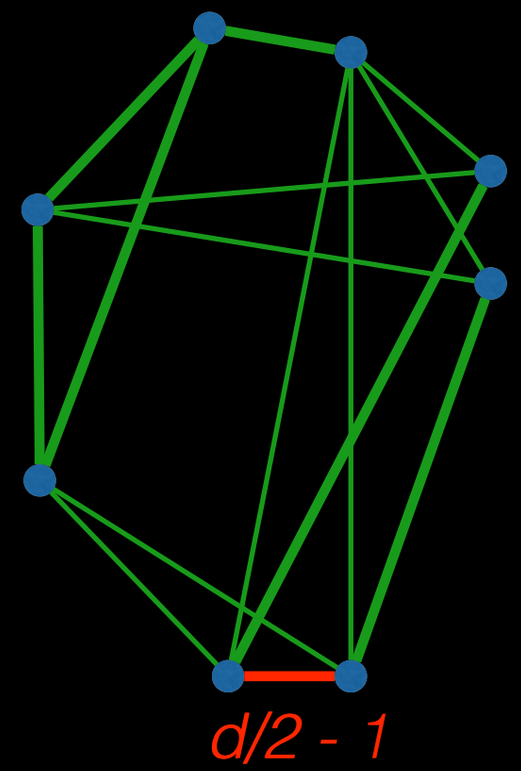
Our Approach

- "Split" each node one at a time, rather than all at once
- Start with *d/2*-regular expander, weights *2*
- Inserting new node: split currently unsplit node
  - Unsplit neighbors: replace weight *2* edge with two weight *1* edges



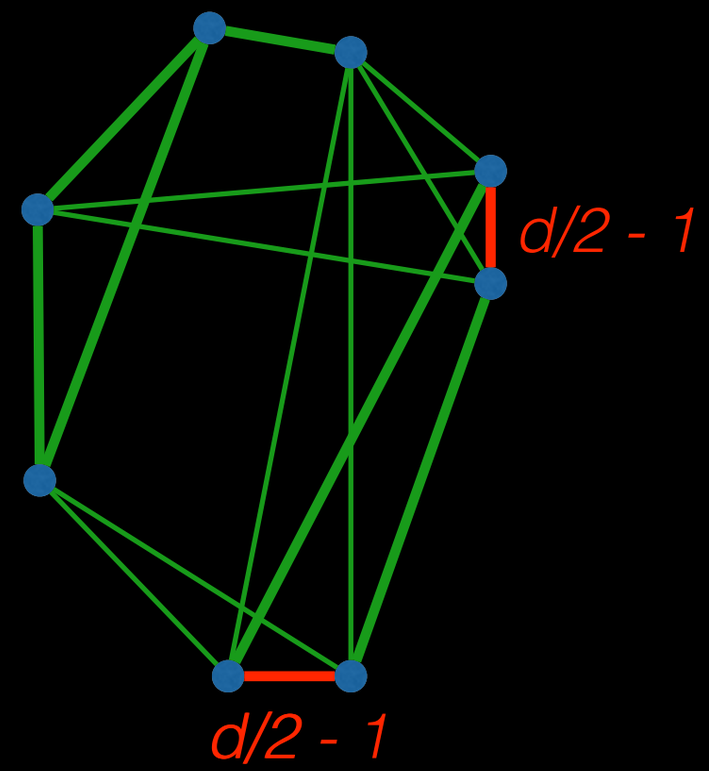
Our Approach

- "Split" each node one at a time, rather than all at once
- Start with *d/2*-regular expander, weights *2*
- Inserting new node: split currently unsplit node
  - Unsplit neighbors: replace weight *2* edge with two weight *1* edges



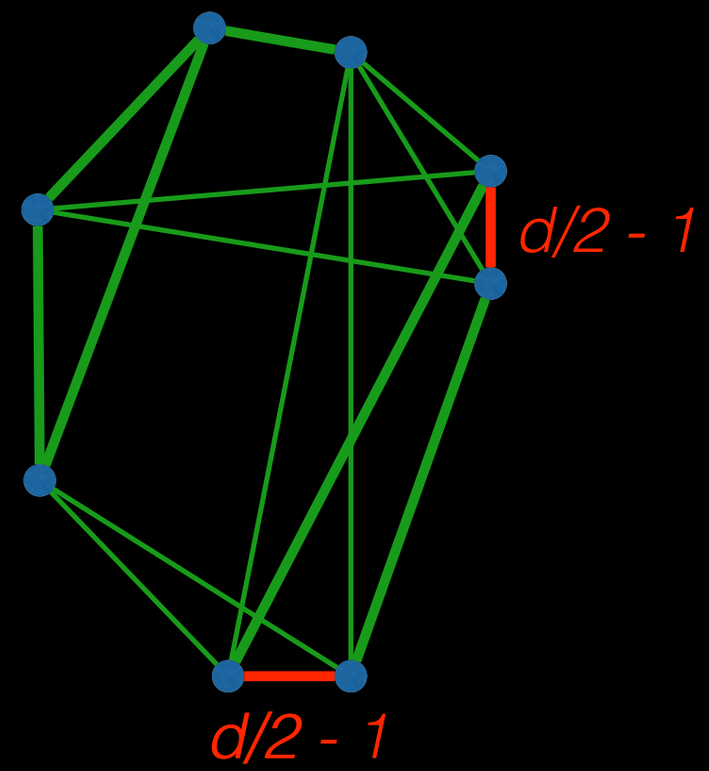
Our Approach

- "Split" each node one at a time, rather than all at once
- Start with *d/2*-regular expander, weights *2*
- Inserting new node: split currently unsplit node
  - Unsplit neighbors: replace weight *2* edge with two weight *1* edges



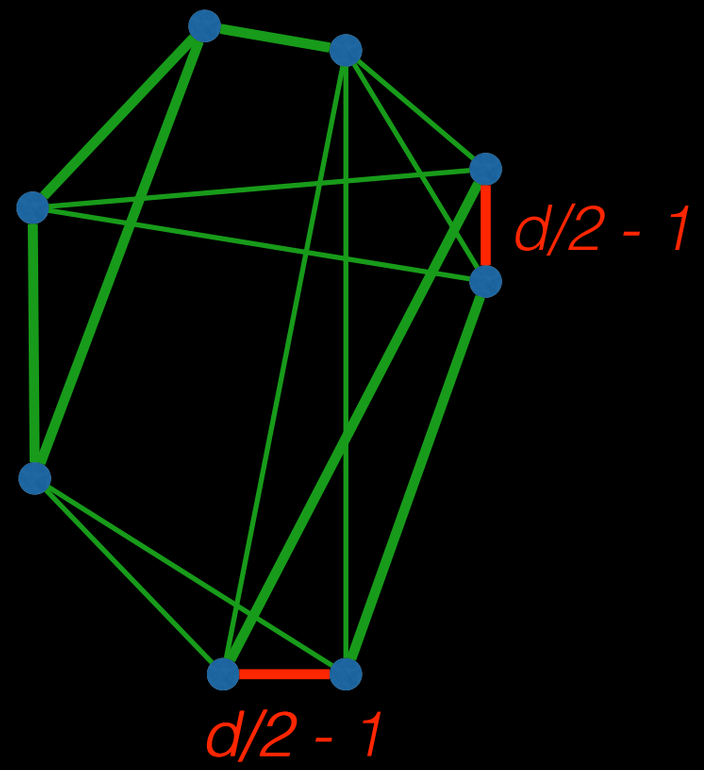
Our Approach

- "Split" each node one at a time, rather than all at once
- Start with *d/2*-regular expander, weights *2*
- Inserting new node: split currently unsplit node
  - Unsplit neighbors: replace weight *2* edge with two weight *1* edges
  - Split neighbors: replace two weight *1* edges with matching of weight *2* edges



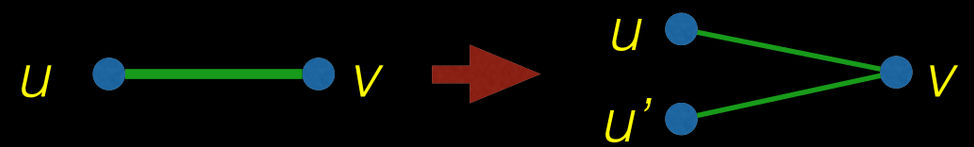
Our Approach

- "Split" each node one at a time, rather than all at once
- Start with *d/2*-regular expander, weights *2*
- Inserting new node: split currently unsplit node
  - Unsplit neighbors: replace weight *2* edge with two weight *1* edges
  - Split neighbors: replace two weight *1* edges with matching of weight *2* edges
- Nice property: after all nodes split, have precisely next BL expander

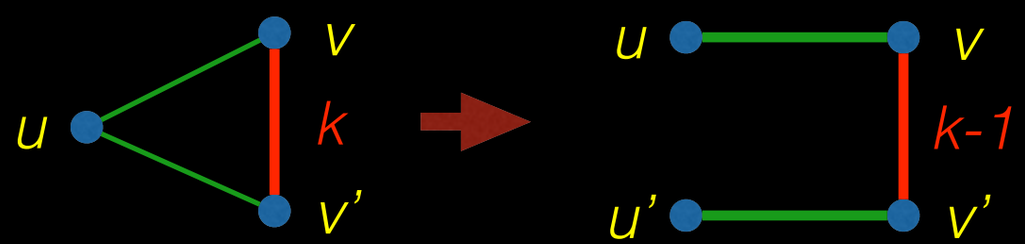


Analysis: Edge Changes

- Split *u* into *u, u'*:
  - Unsplit neighbor *v*: one edge of weight *2* → *2* edges of weight *1*. Cost *2*



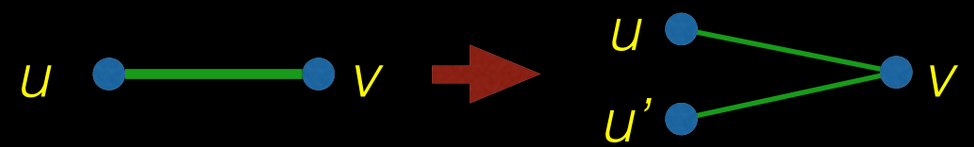
  - Split neighbors *v, v'*: two edges of weight 1 → two edges of weight 2, decrease *{v, v'}* by 1. Cost 5.

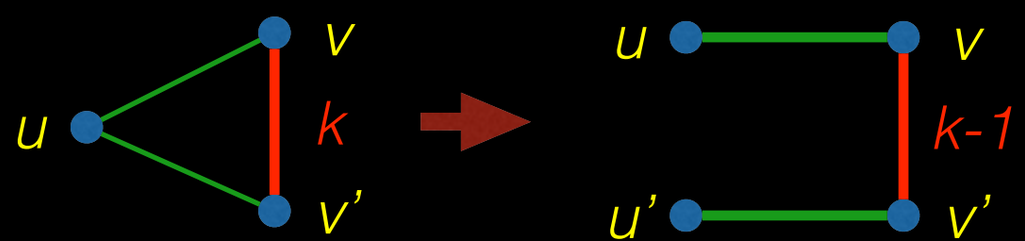


  - Add *{u,u'}* of weight *(# unsplit neighbors)*

Analysis: Edge Changes

- Split *u* into *u, u'*:
  - Unsplit neighbor *v*: one edge of weight *2* → *2* edges of weight *1*. Cost *2*
  - Split neighbors *v, v'*: two edges of weight 1 → two edges of weight 2, decrease *{v, v'}* by 1. Cost 5.
  - Add *{u,u'}* of weight *(# unsplit neighbors)*



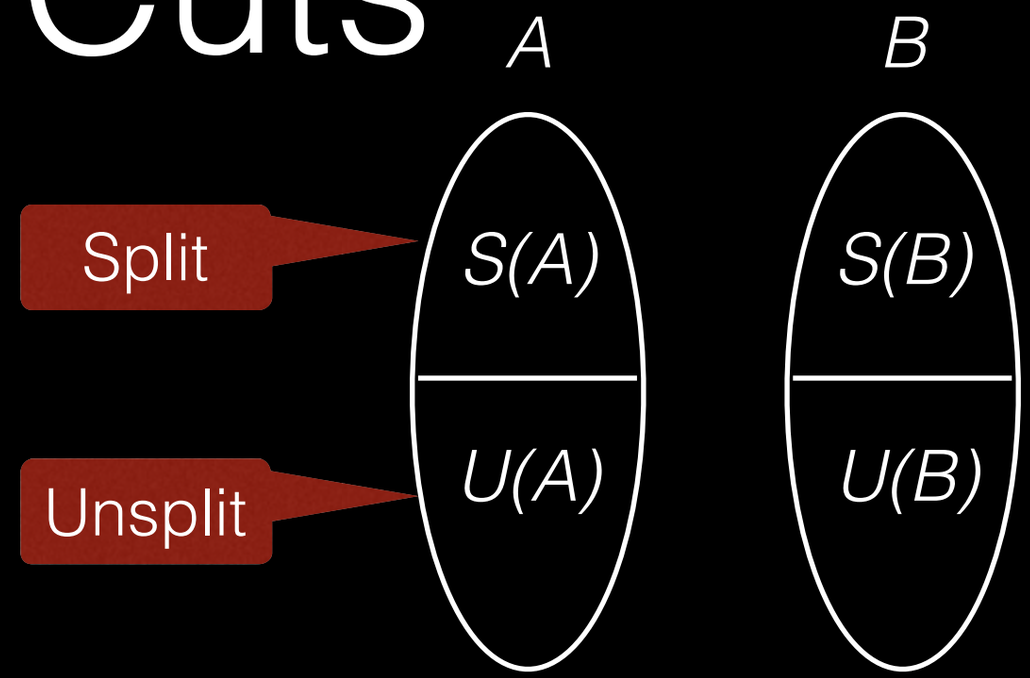


Total cost: *3*(unsplit neighbors) + 5*(split neighbors)*

Know *2*(unsplit) + 2*(split) = d*

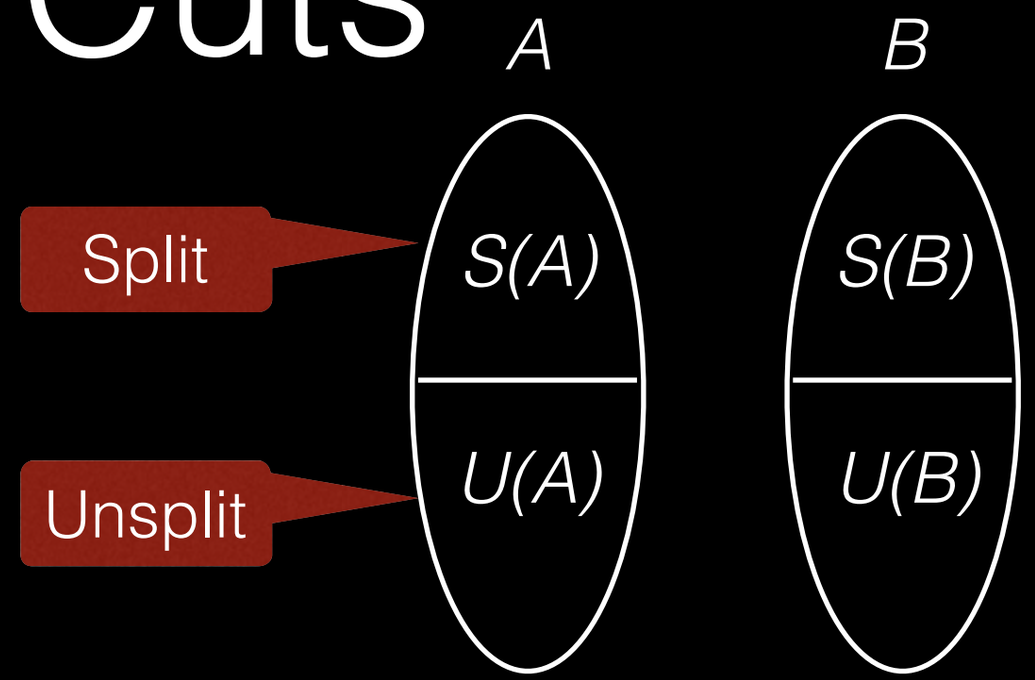
Future Cuts

- Cut *(A, B)*

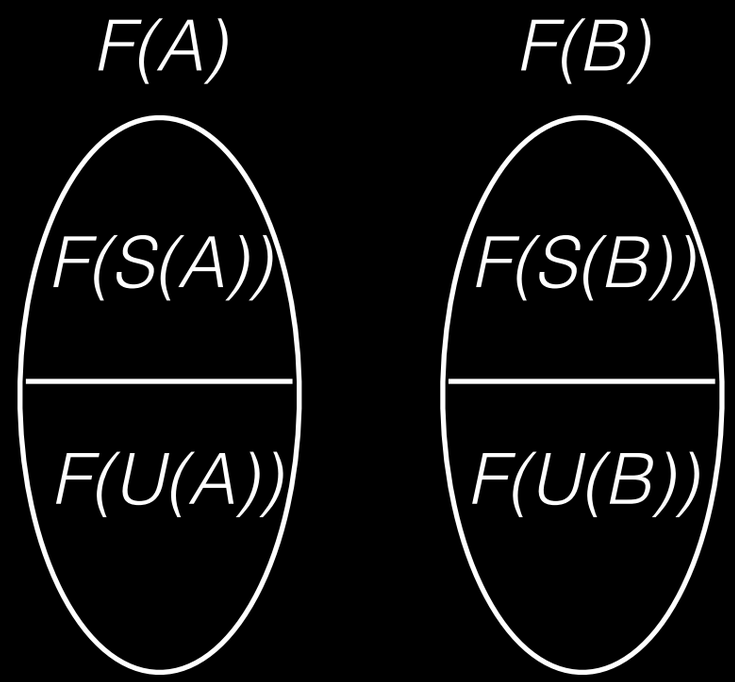


Future Cuts

- Cut *(A, B)*



- Future cut *(F(A), F(B))* of next BL expander:
  - *F(S(·)) = S(·)*
  - *F(U(·)) = U(·)* and splits of *U(·)*



Easy expansion

Easy expansion

- Know that *w(F(A), F(B))* large (≈*d/2*) — argue that *w(A, B)* close to it

Easy expansion

- Know that *w(F(A), F(B))* large (≈*d/2*) — argue that *w(A, B)* close to it
- 2 * *w(U(A), U(B))* = *w(F(U(A), F(U(B)))*

Easy expansion

- Know that *w(F(A), F(B))* large (≈*d/2*) — argue that *w(A, B)* close to it
- 2 * *w(U(A), U(B))* = *w(F(U(A), F(U(B)))*
- *w(S(A), S(B))* = *w(F(S(A)), F(S(B)))*

Easy expansion

- Know that *w(F(A), F(B))* large (≈*d/2*) — argue that *w(A, B)* close to it
- 2 * *w(U(A), U(B))* = *w(F(U(A), F(U(B)))*
- *w(S(A), S(B))* = *w(F(S(A)), F(S(B)))*
- *2 * w(S(A), U(B))* = *w(F(S(A)), F(U(B)))*

Easy expansion

- Know that *w(F(A), F(B))* large (≈*d/2*) — argue that *w(A, B)* close to it
- 2 * *w(U(A), U(B))* = *w(F(U(A), F(U(B)))*
- *w(S(A), S(B))* = *w(F(S(A)), F(S(B)))*
- *2 * w(S(A), U(B))* = *w(F(S(A)), F(U(B)))*

Half the weight, so at least half the expansion (*d/4*)!

Real expansion

- Use fact that fewer vertices than in future cut

- Combine with Expander Mixing Lemma, get expansion *d/3*
  - Need to use strong spectral expansion
  - Fact: there are graphs in sequence with $\lambda_2 \approx d/2$, so cannot get expansion bound directly from Cheeger

Self-Healing Expanders

- Side effect of construction: best distributed algorithms for “self-healing” expanders
  - Expanders which can (distributedly) fix themselves when nodes added, removed
- Deterministic construction makes everything simpler!
  - Essentially just need to know n
- Care about degree, expansion, edge changes, round complexity, message complexity

Self-Healing Expanders

- Previous best [Pandurangan, Robinson, Trehan ‘14]:
  - Max degree *O(1)*,
  - At most *O(1)* edge changes
  - Expansion Ω(1)≈3/20000
  - Message, round complexity *O(log n)* (with high probability)

Thm: for any *d ≥ 6*, self-healing expander with

- Degree *d*
- At most *O(d)* edge changes
- Expansion at least *d/6 - o(d)*
- Message, round complexity *O(log n)*

Future Work

- Analysis is tight — really are *5d/2* edge changes, expansion really is *d/3*
- Improved construction? Expansion *d/2-o(d)*, only *3d/2* edge changes?
- Simple graphs?
- Many interesting questions when trying to make expanders that really work in data centers
  - Wiring? Throughput? Clustering? Failures?

Thanks!