Algorithms for Min-Cost Flow and Dynamic Shortest Paths

Aaron Bernstein – Rutgers university
Maximilar Probst Gutenberg – ETH, Switzerland
Thatchaphol Saranurak – University of Michigan

January 17, 2021

Outline

Outline

Shortest Paths

Single-Source Shortest Paths (SSSP)

- **Input:** Undirected graph $G = (V, E)$, source $s \in V$
- **Output:** $\mathrm{dist}(s, v)$ for every $v \in V$
  - Can also output shortest path tree

Shortest Paths

Single-Source Shortest Paths (SSSP)

- **Input:** Undirected graph $G = (V, E)$, source $s \in V$
- **Output:** $\mathrm{dist}(s, v)$ for every $v \in V$
  - Can also output shortest path tree

Classic Algorithm: Can solve SSSP in $\sim O(m)$ time.

- e.g. BFS, Dijkstra, Thorup 97

m is the number of edges in the graph, n the number of vertices.

Dynamic Shortest Paths

Dynamic Algorithms: Maintain information in a graph that is changing over time.

Dynamic Shortest Paths

Dynamic Algorithms: Maintain information in a graph that is changing over time.

Fully Dynamic SSSP

data structure that handles adversarial update and query operations

- Update: insert or delete a single edge, or change an edge weight.
- Query(v): return $\mathrm{dist}(s, v)$ or corresponding path $\pi(s, v)$.
- **Goal:** Minimize update time while keeping small query time.

Dynamic Shortest Paths

Dynamic Algorithms: Maintain information in a graph that is changing over time.

Fully Dynamic SSSP

data structure that handles adversarial update and query operations

- Update: insert or delete a single edge, or change an edge weight.
- Query(v): return $\mathrm{dist}(s, v)$ or corresponding path $\pi(s, v)$.
- **Goal:** Minimize update time while keeping small query time.

Trivial Upper Bound: $O(m)$ update time, $O(1)$ query time.

- Compute SSSP from scratch after every update.

Dynamic Shortest Paths

Dynamic Algorithms: Maintain information in a graph that is changing over time.

Fully Dynamic SSSP

data structure that handles adversarial update and query operations

- Update: insert or delete a single edge, or change an edge weight.
- Query(v): return $\mathrm{dist}(s, v)$ or corresponding path $\pi(s, v)$.
- **Goal:** Minimize update time while keeping small query time.

Trivial Upper Bound: $O(m)$ update time, $O(1)$ query time.

- Compute SSSP from scratch after every update.

Conditional Lower Bound: $O(m)$ is best possible update time, even with $(1 + \epsilon)$ approximation.

Decremental Shortest Paths

Decremental SSSP: Each update only *deletes* an edge in G or *increases* an edge weight.

- So distances monotonically increasing.

Decremental Shortest Paths

Decremental SSSP: Each update only *deletes* an edge in G or *increases* an edge weight.

- So distances monotonically increasing.

Motivations for Decremental SSSP:

- Natural relaxation of fully dynamic SSSP
- Can hope for non-trivial results (unlike fully dynamic SSSP)
- Used as a subroutine in many dynamic algorithms (both decremental and fully dynamic)

Decremental Shortest Paths

Decremental SSSP: Each update only *deletes* an edge in G or *increases* an edge weight.

- So distances monotonically increasing.

Motivations for Decremental SSSP:

- Natural relaxation of fully dynamic SSSP
- Can hope for non-trivial results (unlike fully dynamic SSSP)
- Used as a subroutine in many dynamic algorithms (both decremental and fully dynamic)
- Powerful data structure for static algorithms. This Talk!

Existing Result for Decremental SSSP

Simplifying Assumption: G is unweighted

- So each update deletes an edge.
- Start with graph G, end with empty graph.

Existing Result for Decremental SSSP

Simplifying Assumption: G is unweighted

- So each update deletes an edge.
- Start with graph G, end with empty graph.

Existing work on Decremental SSSP

- **Trivial:** $O(m^2)$ *total* update time over all deletions.
  - $O(m)$ amortized update time (reconstruction from scratch).
- **Classic:** $O(mn)$ total update time ($O(n)$ amortized).
  Even and Shiloach, 1981

Condition Lower Bound: $O(mn)$ total update time is optimal

Existing Result for Decremental SSSP

Simplifying Assumption: G is unweighted

- So each update deletes an edge.
- Start with graph G, end with empty graph.

Existing work on Decremental SSSP

- **Trivial:** $O(m^2)$ *total* update time over all deletions.
  - $O(m)$ amortized update time (reconstruction from scratch).
- **Classic:** $O(mn)$ total update time ($O(n)$ amortized).
  Even and Shiloach, 1981

Condition Lower Bound: $O(mn)$ total update time is optimal

All recent work seeks to break through $O(mn)$ barrier by allowing $(1+\epsilon)$ approximation.

Adaptive vs. non-adaptive adversaries.

Decremental SSSP:

- Update: delete an edge
- Query(v): return shortest distance/path from s to v.

Adaptive vs. non-adaptive adversaries.

Decremental SSSP:

- Update: delete an edge
- Query(v): return shortest distance/path from s to v.

Stronger Model: Adaptive Adversary

Adversary can choose next update based on response to earlier queries

- Example: adversary does query(v) and then deletes every edge on the returned $s - v$ path.
- Deterministic algorithms always work against adaptive adversary.

Adaptive vs. non-adaptive adversaries.

Decemental SSSP:

- Update: delete an edge
- Query(v): return shortest distance/path from s to v.

Stronger Model: Adaptive Adversary

Adversary can choose next update based on response to earlier queries

- Example: adversary does query(v) and then deletes every edge on the returned $s - v$ path.
- Deterministic algorithms always work against adaptive adversary.

Weaker Model: Non-Adaptive Adversary (aka oblivious adversary)

Entire sequence of updates and queries is fixed in advance.

Adaptive vs. non-adaptive adversaries.

Decremental SSSP:

- Update: delete an edge
- Query(v): return shortest distance/path from s to v.

Stronger Model: Adaptive Adversary

Adversary can choose next update based on response to earlier queries

- Example: adversary does query(v) and then deletes every edge on the returned $s - v$ path.
- Deterministic algorithms always work against adaptive adversary.

Weaker Model: Non-Adaptive Adversary (aka oblivious adversary)

Entire sequence of updates and queries is fixed in advance.

- Many randomized algorithms only work against non-adaptive.
- Adaptive adversary can figure out algorithm's random choices.
- non-adaptive algorithm has zero information about random choices.

Adaptivity and Randomness

The Two Adversarial Models

- Adaptive Adversary (stronger): can typically figure out algorithm's random choices.
- Non-Adaptive Adversary (weaker): has zero information about random choices.

Adaptivity and Randomness

The Two Adversarial Models

- Adaptive Adversary (stronger): can typically figure out algorithm's random choices.
- Non-Adaptive Adversary (weaker): has zero information about random choices.
- We say that an algorithm is adaptive if it works against adaptive adversary
- Deterministic algorithms automatically adaptive.
- Some randomized algorithms also adaptive.

Adaptivity and Randomness

The Two Adversarial Models

- Adaptive Adversary (stronger): can typically figure out algorithm's random choices.
- Non-Adaptive Adversary (weaker): has zero information about random choices.
- We say that an algorithm is adaptive if it works against adaptive adversary
- Deterministic algorithms automatically adaptive.
- Some randomized algorithms also adaptive.

Non-adaptive algorithms are generally much easier to design because they can use randomness to "hide" information from the adversary.

Limitations of Non-adaptive Adversaries

First Limiation

In many natural applications, the adversary is adaptive.

- Examples: traffic control, wear and tear.

Limitations of Non-adaptive Adversaries

First Limiation

In many natural applications, the adversary is adaptive.

- Examples: traffic control, wear and tear.

Second Limitation – Crucial For This Talk

Non-adaptive algorithms cannot be used as black-box data structures.

- Example: user might want to query a path and then delete every edge on that path.

Limitations of Non-adaptive Adversaries

First Limiation

In many natural applications, the adversary is adaptive.

- Examples: traffic control, wear and tear.

Second Limitation – Crucial For This Talk

Non-adaptive algorithms cannot be used as black-box data structures.

- Example: user might want to query a path and then delete every edge on that path.

Bridging the gap between adaptive and non-adaptive algorithms is a central focus of dynamic algorithms over the past decade.

Back to Decremental SSSP

Decremental SSSP:

- Each update deletes an edge
- Query(v): return $(1+\epsilon)$-approximation to $\mathrm{dist}(s,v)$ or shortest-path(s,v)
- $O(mn)$ total update time optimal for exact version.

Back to Decremental SSSP

Decremental SSSP:

- Each update deletes an edge
- Query(v): return $(1+\epsilon)$-approximation to $\mathrm{dist}(s, v)$ or shortest-path(s, v)
- $O(mn)$ total update time optimal for exact version.

Non-Adaptive algorithm: Can solve in $\hat{O}(m)$ total update time

$\tilde{O}(\cdot)$ hides polylog factors; $\hat{O}(\cdot)$ hides $n^{o(1)}$ factors.

Back to Decremental SSSP

Decremental SSSP:

- Each update deletes an edge
- Query(v): return $(1+\epsilon)$-approximation to $\mathrm{dist}(s, v)$ or shortest-path(s, v)
- $O(mn)$ total update time optimal for exact version.

Non-Adaptive algorithm: Can solve in $\hat{O}(m)$ total update time

- [Forster, Henzinger, Nanongkai, 2014]
- Optimal up to sub-polynomial factors.
- Concludes long line of research

$\tilde{O}(\cdot)$ hides polylog factors; $\hat{O}(\cdot)$ hides $n^{o(1)}$ factors.

Back to Decremental SSSP

Decremental SSSP:

- Each update deletes an edge
- Query(v): return $(1+\epsilon)$-approximation to $\mathrm{dist}(s, v)$ or shortest-path(s, v)
- $O(mn)$ total update time optimal for exact version.

Non-Adaptive algorithm: Can solve in $\hat{O}(m)$ total update time

- [Forster, Henzinger, Nanongkai, 2014]
- Optimal up to sub-polynomial factors.
- Concludes long line of research

Deterministic (and hence adaptive) algorithms

- $\tilde{O}(n^2)$ total update time [BC16,B17,CK19,CS20]
- $\tilde{O}(mn^{3/4})$ [BC17]
- $\hat{O}(m\sqrt{n})$ [GW20]

$\tilde{O}(\cdot)$ hides polylog factors; $\hat{O}(\cdot)$ hides $n^{o(1)}$ factors.

Our Result

Previous Work (undirected graph)

- **Non-Adaptive:** $\hat{O}(m)$ total update time.
- **Deterministic (and so adaptive):** $\hat{O}(\min(n^2, m\sqrt{n})$ total update time.

Our Result

Previous Work (undirected graph)

- **Non-Adaptive:** $\hat{O}(m)$ total update time.
- **Deterministic (and so adaptive):** $\hat{O}(\min(n^2, m\sqrt{n})$ total update time.

Our Result (undirected graph)

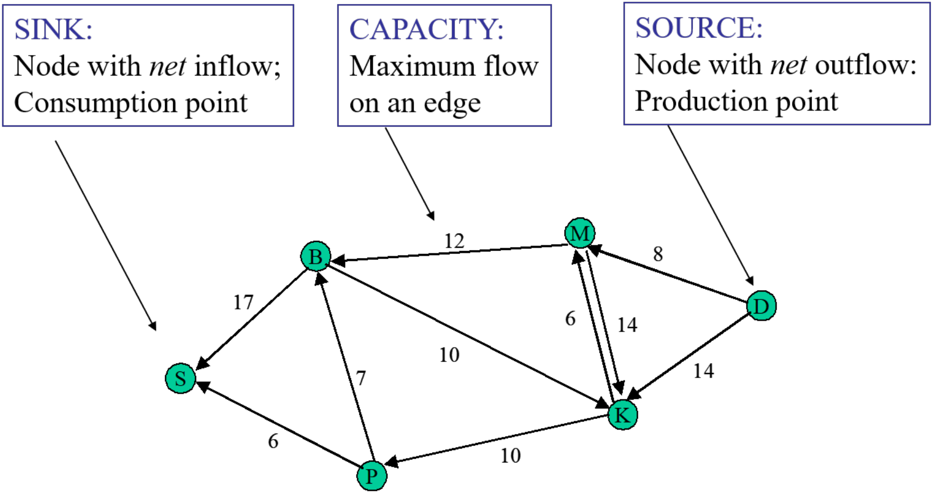
Adaptive decremental SSSP in total update time $\hat{O}(m)$

- Closes the adaptive / non-adaptive gap.
- Optimal update time up to sub-polynomial factors.
- Generalizes to weighted graphs.
- Concludes long line of research.

Outline

Outline

Maximum Flow: Figure



Edge-Capacitated Maximum Flow

Input:

- Undirected graph $G = (V, E)$
- Fixed source s, sink t.
  - Can also handle arbitrary demand vector
- Capacity function $u : E \to \mathbb{R}_{\geq 0}$

Output: maximum flow f from s to t such that $f(e) \leq u(e)\ \forall e \in E$.

Applications of Maximum Flow

Network	Nodes	Arcs	Flow
communication	telephone exchanges, computers, satellites	cables, fiber optics, microwave relays	voice, video, packets
circuits	gates, registers, processors	wires	current
mechanical	joints	rods, beams, springs	heat, energy
hydraulic	reservoirs, pumping stations, lakes	pipelines	fluid, oil
financial	stocks, companies	transactions	money
transportation	airports, rail yards, street intersections	highways, railbeds, airway routes	freight, vehicles, passengers
chemical	sites	bonds	energy

Common Flow Variants

Edge-Capacitated Max Flow (standard)

- Every edge has capacity $u(e) \geq 0$
- Flow f must satisfy $f(e) \leq u(e)$

Common Flow Variants

Edge-Capacitated Max Flow (standard)

- Every edge has capacity $u(e) \geq 0$
- Flow f must satisfy $f(e) \leq u(e)$

Vertex-Capacitated Max Flow

- Every *vertex* has capacity $u(v) \geq 0$
- Flow through any vertex must satisfy in-flow$(v) \leq u(v)$.

Common Flow Variants

Edge-Capacitated Max Flow (standard)

- Every edge has capacity $u(e) \geq 0$
- Flow f must satisfy $f(e) \leq u(e)$

Vertex-Capacitated Max Flow

- Every *vertex* has capacity $u(v) \geq 0$
- Flow through any vertex must satisfy in-flow$(v) \leq u(v)$.

Minimum Cost Flow

- Every edge also has cost $c(e)$
- Also given budget B as input
- Cost of flow f is $\sum_{e \in E} f(e) \cdot c(e)$
- Goal is to compute maximum $s - t$ flow with cost at most B.

Existing Work on Maximum Flow

Exact Max-Flow: State-of-the-art

- $\hat{O}(m + n^{1.5})$
  [van den Brand, Lee, Liu, Saranurak, Sidford, Song, Wang,2020]
- $\hat{O}(m^{4/3})$ for unit capacities [Axiotis, Madri, Vlaud, 2020]

Existing Work on Maximum Flow

Exact Max-Flow: State-of-the-art

- $\hat{O}(m + n^{1.5})$
  [van den Brand, Lee, Liu, Saranurak, Sidford, Song, Wang,2020]
- $\hat{O}(m^{4/3})$ for unit capacities [Axiotis, Madri, Vlaud, 2020]
- both extend to min-cost flow and vertex capacities.
- Based on interior-point methods.

Existing Work on Maximum Flow

Exact Max-Flow: State-of-the-art

- $\hat{O}(m + n^{1.5})$ [van den Brand, Lee, Liu, Saranurak, Sidford, Song, Wang,2020]
- $\hat{O}(m^{4/3})$ for unit capacities [Axiotis, Madri, Vlaud, 2020]
- both extend to min-cost flow and vertex capacities.
- Based on interior-point methods.

Approximate Max-Flow: State-of-the-art

$(1 + \epsilon)$-approximation, limited to undirected graphs

Existing Work on Maximum Flow

Exact Max-Flow: State-of-the-art

- $\hat{O}(m + n^{1.5})$
  [van den Brand, Lee, Liu, Saranurak, Sidford, Song, Wang,2020]
- $\hat{O}(m^{4/3})$ for unit capacities [Axiotis, Madri, Vlaud, 2020]
- both extend to min-cost flow and vertex capacities.
- Based on interior-point methods.

Approximate Max-Flow: State-of-the-art

$(1 + \epsilon)$-approximation, limited to undirected graphs

- Edge-Capacitated Max Flow: $\tilde{O}(m)$ [Sherman13, KLOS14,Peng16]
  - Does not extend to costs or vertex capacities

Existing Work on Maximum Flow

Exact Max-Flow: State-of-the-art

- $\hat{O}(m + n^{1.5})$
  [van den Brand, Lee, Liu, Saranurak, Sidford, Song, Wang,2020]
- $\hat{O}(m^{4/3})$ for unit capacities [Axiotis, Madri, Vlaud, 2020]
- both extend to min-cost flow and vertex capacities.
- Based on interior-point methods.

Approximate Max-Flow: State-of-the-art

$(1 + \epsilon)$-approximation, limited to undirected graphs

- Edge-Capacitated Max Flow: $\tilde{O}(m)$ [Sherman13, KLOS14,Peng16]
  - Does not extend to costs or vertex capacities
- Special Case – Transshipment (costs but no capacities): $\tilde{O}(m)$ [Sherman17,Li20,ASZ20]

Previous Work Summary

Exact Max Flow in Directed Graphs: $\hat{O}(m + n^{1.5})$

- Works for edge capacities, vertex capacities, costs.

Previous Work Summary

Exact Max Flow in Directed Graphs: $\hat{O}(m + n^{1.5})$

- Works for edge capacities, vertex capacities, costs.

$(1 + \epsilon)$-Approximate Max Flow in Undirected Graphs

- Edge-Capacitated Max Flow: $\tilde{O}(m)$
- Vertex-Capacitated Max Flow: $\hat{O}(m + n^{1.5})$
- Min-Cost flow: $\hat{O}(m + n^{1.5})$

Previous Work Summary

Exact Max Flow in Directed Graphs: $\hat{O}(m + n^{1.5})$

- Works for edge capacities, vertex capacities, costs.

$(1+\epsilon)$-Approximate Max Flow in Undirected Graphs

- Edge-Capacitated Max Flow: $\tilde{O}(m)$
- Vertex-Capacitated Max Flow: $\hat{O}(m + n^{1.5})$
- Min-Cost flow: $\hat{O}(m + n^{1.5})$

Open Problem: Can we solve approximate min-cost flow in time $\hat{O}(m)$?

Previous Work Summary

Exact Max Flow in Directed Graphs: $\hat{O}(m + n^{1.5})$

- Works for edge capacities, vertex capacities, costs.

$(1+\epsilon)$-Approximate Max Flow in Undirected Graphs

- Edge-Capacitated Max Flow: $\tilde{O}(m)$
- Vertex-Capacitated Max Flow: $\hat{O}(m + n^{1.5})$
- Min-Cost flow: $\hat{O}(m + n^{1.5})$

Open Problem: Can we solve approximate min-cost flow in time $\hat{O}(m)$?

our result: yes!

Outline

Our Result

Previous $(1+\epsilon)$-Approximate Max Flow in Undirected Graphs

- Edge-Capacitated Max Flow: $\tilde{O}(m)$
- Vertex-Capacitated Max Flow: $\hat{O}(m + n^{1.5})$
- Min-Cost flow: $\hat{O}(m + n^{1.5})$

Our Result

$(1+\epsilon)$-approximation min-cost flow in $\hat{O}(m)$ time.

- Can handle costs/capacities on both vertices/edges.
- Completes the picture for approximate flow in undirected graphs.

Outline

Faster Flow Algorithms via Dynamic Shortest Paths

MWU framework for maximum flow

1. Given: source s, sink t.
2. Algorithm introduces a weight function $w : E \to \mathbb{R}_{\geq 0}$
3. Start with initial $w(e)$ (simple)

Faster Flow Algorithms via Dynamic Shortest Paths

MWU framework for maximum flow

1. Given: source s, sink t.
2. Algorithm introduces a weight function $w : E \to \mathbb{R}_{\geq 0}$
3. Start with initial $w(e)$ (simple)
4. Repeat Many Times:
   - Compute a $(1+\epsilon)$-approximate shortest path $\pi(s,t)$ w.r.t w.
   - Send flow on $\pi(s,t)$.
   - Increase $w(e)$ $\forall e \in \pi(s,t)$.

Faster Flow Algorithms via Dynamic Shortest Paths

MWU framework for maximum flow

1. Given: source s, sink t.
2. Algorithm introduces a weight function $w : E \to \mathbb{R}_{\geq 0}$
3. Start with initial $w(e)$ (simple)
4. Repeat Many Times:
   - Compute a $(1+\epsilon)$-approximate shortest path $\pi(s,t)$ w.r.t w.
   - Send flow on $\pi(s,t)$.
   - Increase $w(e)$ $\forall e \in \pi(s,t)$.

Lemma: Above algorithm returns $(1+\epsilon)$-approximation to max flow

- Easily generalizes to min-cost flow and vertex capacities.
- [Garg and Koenneman, 1998]

MWU and Dynamic SSSP

MWU framework for maximum flow

1. Given: source s, sink t.
2. Initialize weight function $w : E \to \mathbb{R}_{\geq 0}$

MWU and Dynamic SSSP

MWU framework for maximum flow

1. Given: source s, sink t.
2. Initialize weight function $w : E \to \mathbb{R}_{\geq 0}$
3. Repeat Many Times:
   - Compute a $(1+\epsilon)$-approximate shortest path $\pi(s,t)$ w.r.t to w.
   - Send flow on $\pi(s,t)$
   - Increase $w(e) \forall e \in \pi(s,t)$.

MWU and Dynamic SSSP

MWU framework for maximum flow

1. Given: source s, sink t.
2. Initialize weight function $w : E \to \mathbb{R}_{\geq 0}$
3. Repeat Many Times:
   - Compute a $(1+\epsilon)$-approximate shortest path $\pi(s,t)$ w.r.t to w.
   - Send flow on $\pi(s,t)$
   - Increase $w(e) \forall e \in \pi(s,t)$.

Using Dynamic SSSP to speed up MWU [Madry 2010]

- Must compute a new shortest path in every iteration of step 3.

MWU and Dynamic SSSP

MWU framework for maximum flow

1. Given: source s, sink t.
2. Initialize weight function $w : E \to \mathbb{R}_{\geq 0}$
3. Repeat Many Times:
   - Compute a $(1+\epsilon)$-approximate shortest path $\pi(s,t)$ w.r.t to w.
   - Send flow on $\pi(s,t)$
   - Increase $w(e) \forall e \in \pi(s,t)$.

Using Dynamic SSSP to speed up MWU [Madry 2010]

- Must compute a new shortest path in every iteration of step 3.
- Weights $w(e)$ only increase between iterations.
- Find the paths using decremental SSSP.

MWU and Dynamic SSSP

MWU framework for maximum flow

1. Given: source s, sink t.
2. Initialize weight function $w : E \to \mathbb{R}_{\geq 0}$
3. Repeat Many Times:
   - Compute a $(1 + \epsilon)$-approximate shortest path $\pi(s, t)$ w.r.t to w.
   - Send flow on $\pi(s, t)$
   - Increase $w(e) \forall e \in \pi(s, t)$.

Using Dynamic SSSP to speed up MWU [Madry 2010]

- Must compute a new shortest path in every iteration of step 3.
- Weights $w(e)$ only increase between iterations.
- Find the paths using decremental SSSP.
- Total running time of MWU depends on total update time of decremental SSSP

Two Challenges of MWU

MWU framework: Repeatedly compute shortest path $\pi(s, t)$ and update every edge on the path.

Goal: Execute MWU framework in $\hat{O}(m)$ time.

Two Challenges of MWU

MWU framework: Repeatedly compute shortest path $\pi(s,t)$ and update every edge on the path.

Goal: Execute MWU framework in $\hat{O}(m)$ time.

First Challenge: need an *adaptive* decremental SSSP algorithm with total update time $\hat{O}(m)$

- Our first result
- Our MWU algorithm uses our decremental SSSP algorithm as black box.

Two Challenges of MWU

MWU framework: Repeatedly compute shortest path $\pi(s,t)$ and update every edge on the path.

Goal: Execute MWU framework in $\hat{O}(m)$ time.

First Challenge: need an *adaptive* decremental SSSP algorithm with total update time $\hat{O}(m)$

- Our first result
- Our MWU algorithm uses our decremental SSSP algorithm as black box.

Second Challenge: Total length of all the paths $\pi(s,t)$ may be too long.

- Known as flow decomposition barrier

Flow Decomposition Barrier

Question: say we are given a $s-t$ flow f and we decompose f into many $s-t$ paths: $f = \sum p(s,t)$. What is the maximum value of $\sum |p(s,t)|$?

Flow Decomposition Barrier

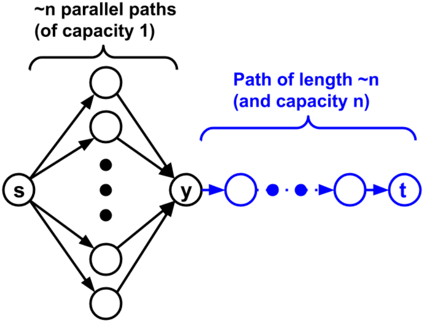
Question: say we are given a $s-t$ flow f and we decompose f into many $s-t$ paths: $f = \sum p(s,t)$. What is the maximum value of $\sum |p(s,t)|$?

- unit-capacity edges: $\sum |p(s,t)| = \Theta(m)$

Flow Decomposition Barrier

Question: say we are given a $s - t$ flow f and we decompose f into many $s - t$ paths: $f = \sum p(s,t)$. What is the maximum value of $\sum |p(s,t)|$?

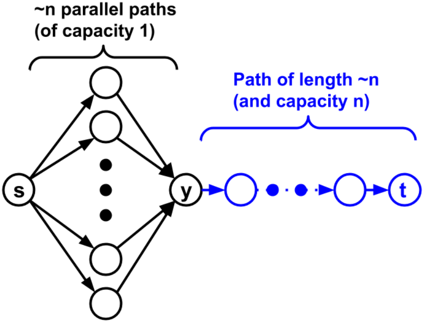
- unit-capacity edges: $\sum |p(s,t)| = \Theta(m)$



Flow Decomposition Barrier

Question: say we are given a $s - t$ flow f and we decompose f into many $s - t$ paths: $f = \sum p(s,t)$. What is the maximum value of $\sum |p(s,t)|$?

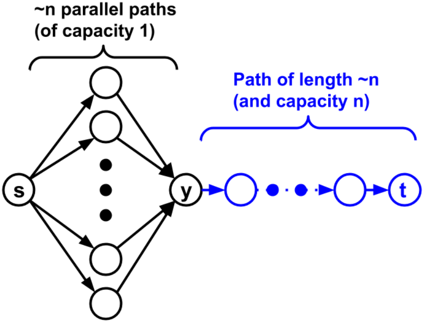
- unit-capacity edges: $\sum |p(s,t)| = \Theta(m)$
- general edge capacities: $\sum |p(s,t)| = \Theta(mn)$
- general vertex capacities: $\sum |p(s,t)| = \Theta(n^2)$



Flow Decomposition Barrier

Question: say we are given a $s-t$ flow f and we decompose f into many $s-t$ paths: $f = \sum p(s,t)$. What is the maximum value of $\sum |p(s,t)|$?

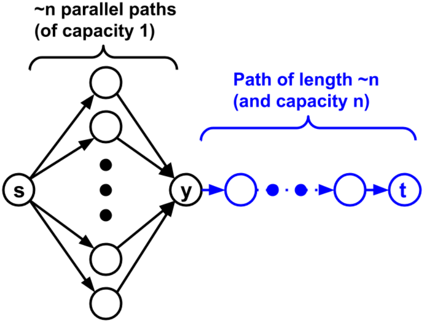
- unit-capacity edges: $\sum |p(s,t)| = \Theta(m)$
- general edge capacities: $\sum |p(s,t)| = \Theta(mn)$
- general vertex capacities: $\sum |p(s,t)| = \Theta(n^2)$
- Known as *flow decomposition barrier*



Flow Decomposition Barrier

Question: say we are given a $s - t$ flow f and we decompose f into many $s - t$ paths: $f = \sum p(s,t)$. What is the maximum value of $\sum |p(s,t)|$?

- unit-capacity edges: $\sum |p(s,t)| = \Theta(m)$
- general edge capacities: $\sum |p(s,t)| = \Theta(mn)$
- general vertex capacities: $\sum |p(s,t)| = \Theta(n^2)$
- Known as *flow decomposition barrier*



No previous MWU-based flow algorithm went beyond flow decomposition barrier.

Two Challenges of MWU

Goal: Execute MWU framework in $\hat{O}(m)$ time.

Two Challenges of MWU

Goal: Execute MWU framework in $\hat{O}(m)$ time.

First Challenge: need an *adaptive* decremental SSSP algorithm with total update time $\hat{O}(m)$.

- long-standing open problem in dynamic shortest paths
- Focus of this talk.

Two Challenges of MWU

Goal: Execute MWU framework in $\hat{O}(m)$ time.

First Challenge: need an *adaptive* decremental SSSP algorithm with total update time $\hat{O}(m)$.

- long-standing open problem in dynamic shortest paths
- Focus of this talk.

Second Challenge: Total length of all the paths $\pi(s, t)$ may be too long.

- Known as flow decomposition barrier
- We make significant changes to MWU-framework.
- Introduces randomization

Two Challenges of MWU

Goal: Execute MWU framework in $\hat{O}(m)$ time.

First Challenge: need an *adaptive* decremental SSSP algorithm with total update time $\hat{O}(m)$.

- long-standing open problem in dynamic shortest paths
- Focus of this talk.

Second Challenge: Total length of all the paths $\pi(s, t)$ may be too long.

- Known as flow decomposition barrier
- We make significant changes to MWU-framework.
- Introduces randomization

Our min-cost flow result introduces the first solution to both above challenges

Two Challenges of MWU

Goal: Execute MWU framework in $\hat{O}(m)$ time.

First Challenge: need an *adaptive* decremental SSSP algorithm with total update time $\hat{O}(m)$.

- long-standing open problem in dynamic shortest paths
- Focus of this talk.

Second Challenge: Total length of all the paths $\pi(s, t)$ may be too long.

- Known as flow decomposition barrier
- We make significant changes to MWU-framework.
- Introduces randomization

Our min-cost flow result introduces the first solution to both above challenges

Note: our solutions to the two challenges entirely unrelated

- This Talk: first challenge only (dynamic SSSP)

Outline

Decremental SSSP review

Reviewing the model

- Initial undirected graph G, fixed source s
- This Talk: assume G unweighted
- Each update deletes an edge (u, v) in G
- Goal: maintain $(1 + \epsilon)$-approximate shortest paths from s
- Goal: deterministic (and hence adaptive) algorithm.

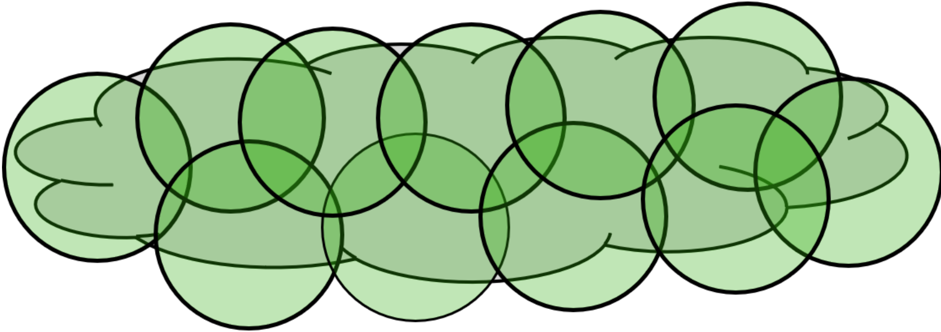
Decremental SSSP review

Reviewing the model

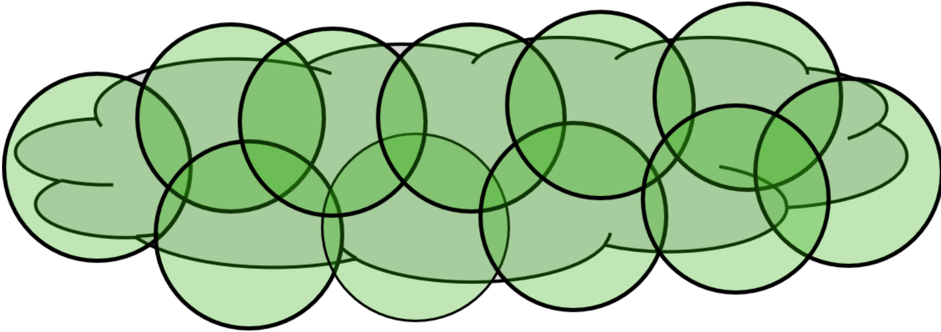
- Initial undirected graph G, fixed source s
- This Talk: assume G unweighted
- Each update deletes an edge (u, v) in G
- Goal: maintain $(1+\epsilon)$-approximate shortest paths from s
- Goal: deterministic (and hence adaptive) algorithm.

Our Result: Adaptive $(1+\epsilon)$-approximation in total update time $\hat{O}(m)$.

High-Level Approach: Maintain Low Diameter Balls

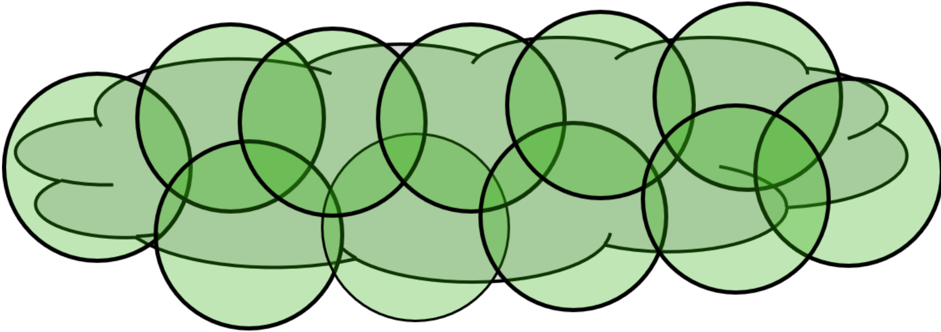


High-Level Approach: Maintain Low Diameter Balls



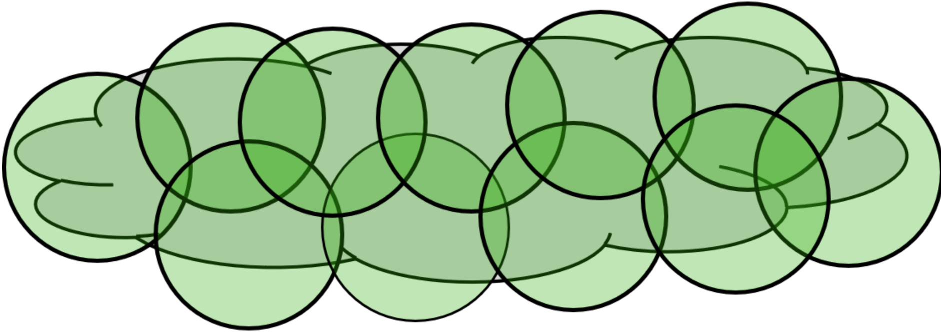
- **Static Problem (easy-ish)**: cover V with low-diameter balls

High-Level Approach: Maintain Low Diameter Balls



- **Static Problem (easy-ish)**: cover V with low-diameter balls
- **Dynamic Problem (hard):** maintain covering of low-diameter balls.

High-Level Approach: Maintain Low Diameter Balls



- **Static Problem (easy-ish)**: cover V with low-diameter balls
- **Dynamic Problem (hard):** maintain covering of low-diameter balls.
- **Key Dynamic Building Block:** Start with low-diameter ball K^{init}. As edges in G are deleted, detect vertices in K^{init} that are no longer close to the rest of the ball.

Definition

Recall: $\hat{O}(\cdot)$ and $\hat{\Omega}(\cdot)$ hide polynomial factors.

Recall: We assume that G is unweighted

- So each adversarial update deletes an edge in G

Definition: Weak Diameter Given graph G and set $K \subseteq V(G)$, define

$$\mathrm{diam}_G(K) \triangleq \min_{x,y \in K} \mathrm{dist}(x, y)$$

Defining Robust Core (Simplified Version)

Input: Graph G subject to edge deletions; initially $\mathrm{diam}(G) = d = n^{o(1)}$.
Define $|V(G) = n|$, $|E(G) = m|$.

Defining Robust Core (Simplified Version)

Input: Graph G subject to edge deletions; initially $\mathrm{diam}(G) = d = n^{o(1)}$.
Define $|V(G) = n|$, $|E(G) = m|$.

Simplified RobustCore(G)

Maintain a set $K \subseteq V(G)$ with the following properties:

Defining Robust Core (Simplified Version)

Input: Graph G subject to edge deletions; initially $\mathrm{diam}(G) = d = n^{o(1)}$.
Define $|V(G) = n|$, $|E(G) = m|$.

Simplified RobustCore(G)

Maintain a set $K \subseteq V(G)$ with the following properties:

- **Diameter Property:** $\mathrm{diam}_G(K) = \hat{O}(d) = n^{o(1)}$

Defining Robust Core (Simplified Version)

Input: Graph G subject to edge deletions; initially $\mathrm{diam}(G) = d = n^{o(1)}$.
Define $|V(G) = n|$, $|E(G) = m|$.

Simplified RobustCore(G)

Maintain a set $K \subseteq V(G)$ with the following properties:

- **Diameter Property:** $\mathrm{diam}_G(K) = \hat{O}(d) = n^{o(1)}$
- **Scattering Property:** For every $v \in V(G) \setminus K$ we have

$$|\mathrm{ball}(v, 2d)| \leq .99n$$

Defining Robust Core (Simplified Version)

Input: Graph G subject to edge deletions; initially $\mathrm{diam}(G) = d = n^{o(1)}$.
Define $|V(G) = n|$, $|E(G) = m|$.

Simplified RobustCore(G)

Maintain a set $K \subseteq V(G)$ with the following properties:

- **Diameter Property:** $\mathrm{diam}_G(K) = \hat{O}(d) = n^{o(1)}$
- **Scattering Property:** For every $v \in V(G) \setminus K$ we have
$$|\mathrm{ball}(v, 2d)| \leq .99n$$
- **Termination:** if at some point $|K| \leq n/2$, can set $K \leftarrow \emptyset$.

Robust Core and Decremental SSSP

First Task: Find a solution to RobustCore

- Focus of this talk

Robust Core and Decremental SSSP

First Task: Find a solution to RobustCore

- Focus of this talk

Second Task: Show that RobustCore $\rightarrow$ decremental SSSP

Robust Core and Decremental SSSP

First Task: Find a solution to RobustCore

- Focus of this talk

Second Task: Show that RobustCore $\rightarrow$ decremental SSSP

- Requires several new techniques
- Borrows many ideas from existing work on dynamic SSSP (hopsets, clustering, monotone even and Shiloach, etc.)

Outline

Simplified Robust Core

Input: Graph G subject to edge deletions; initially $\mathrm{diam}(G) = d = n^{o(1)}$. Define $|V(G) = n|$, $|E(G) = m|$.

Simplified RobustCore(G)

Maintain a set $K \subseteq V(G)$ with the following properties:

- **Diameter Property:** $\mathrm{diam}_G(K) = \hat{O}(d) = n^{o(1)}$
- **Scattering Property:** For every $v \in V(G) \setminus K$ we have

$$|\mathrm{ball}(v, 2d)| \leq .99n$$

- **Termination:** if at some point $|K| \leq |n/2|$, can set $K \leftarrow \emptyset$.

Simplified Robust Core

Input: Graph G subject to edge deletions; initially $\mathrm{diam}(G) = d = n^{o(1)}$.
Define $|V(G) = n|$, $|E(G) = m|$.

Simplified RobustCore(G)

Maintain a set $K \subseteq V(G)$ with the following properties:

- **Diameter Property:** $\mathrm{diam}_G(K) = \hat{O}(d) = n^{o(1)}$
- **Scattering Property:** For every $v \in V(G) \setminus K$ we have

$$|\mathrm{ball}(v, 2d)| \leq .99n$$

- **Termination:** if at some point $|K| \leq |n/2|$, can set $K \leftarrow \emptyset$.

Robust Core distills basic subroutine used by almost all previous algorithms for Decremental SSSP.

Simplified Robust Core

Input: Graph G subject to edge deletions; initially $\mathrm{diam}(G) = d = n^{o(1)}$.
Define $|V(G) = n|$, $|E(G) = m|$.

Simplified RobustCore(G)

Maintain a set $K \subseteq V(G)$ with the following properties:

- **Diameter Property:** $\mathrm{diam}_G(K) = \hat{O}(d) = n^{o(1)}$
- **Scattering Property:** For every $v \in V(G) \setminus K$ we have

$$|\mathrm{ball}(v, 2d)| \leq .99n$$

- **Termination:** if at some point $|K| \leq |n/2|$, can set $K \leftarrow \emptyset$.

Robust Core distills basic subroutine used by almost all previous algorithms for Decremental SSSP.

Our Result: Solve RobustCore(G) in *total* time $\hat{O}(m)$.

Non-Adaptive Algorithms: Random Source

Recall: [initial diameter of G] $= d = n^{o(1)}$
Scattering: If $v \in V(G) \setminus K$ then $|\mathrm{ball}(v, 2d)| \leq .99n$

Non-Adaptive Algorithms: Random Source

Recall: [initial diameter of G] $= d = n^{o(1)}$
Scattering: If $v \in V(G) \setminus K$ then $|\mathrm{ball}(v, 2d)| \leq .99n$

RobustCore(G) via Random Source

- Pick *random* source $s \in V$
- Maintain $\mathrm{ball}(s, 5d)$: can do in total time $O(md) = \hat{O}(m)$ (ES-tree).

Non-Adaptive Algorithms: Random Source

Recall: [initial diameter of G] $= d = n^{o(1)}$
Scattering: If $v \in V(G) \setminus K$ then $|\mathrm{ball}(v, 2d)| \leq .99n$

RobustCore(G) via Random Source

- Pick *random* source $s \in V$
- Maintain $\mathrm{ball}(s, 5d)$: can do in total time $O(md) = \hat{O}(m)$ (ES-tree).
- Whenever v leaves $\mathrm{ball}(s, 5d)$, remove v from K.

Non-Adaptive Algorithms: Random Source

Recall: [initial diameter of G] $= d = n^{o(1)}$
Scattering: If $v \in V(G) \setminus K$ then $|\mathrm{ball}(v, 2d)| \leq .99n$

RobustCore(G) via Random Source

- Pick *random* source $s \in V$
- Maintain $\mathrm{ball}(s, 5d)$: can do in total time $O(md) = \hat{O}(m)$ (ES-tree).
- Whenever v leaves $\mathrm{ball}(s, 5d)$, remove v from K.
- If at any point $|\mathrm{ball}(s, 2d)| \leq n/2$, restart with new source.

Non-Adaptive Algorithms: Random Source

Recall: [initial diameter of G] $= d = n^{o(1)}$
Scattering: If $v \in V(G) \setminus K$ then $|\mathrm{ball}(v, 2d)| \leq .99n$

RobustCore(G) via Random Source

- Pick *random* source $s \in V$
- Maintain $\mathrm{ball}(s, 5d)$: can do in total time $O(md) = \hat{O}(m)$ (ES-tree).
- Whenever v leaves $\mathrm{ball}(s, 5d)$, remove v from K.
- If at any point $|\mathrm{ball}(s, 2d)| \leq n/2$, restart with new source.

Analysis:

- If $|\mathrm{ball}(s, 2d)| \leq n/2$ then s is scattered.

Non-Adaptive Algorithms: Random Source

Recall: [initial diameter of G] $= d = n^{o(1)}$
Scattering: If $v \in V(G) \setminus K$ then $|\mathrm{ball}(v, 2d)| \leq .99n$

RobustCore(G) via Random Source

- Pick *random* source $s \in V$
- Maintain $\mathrm{ball}(s, 5d)$: can do in total time $O(md) = \hat{O}(m)$ (ES-tree).
- Whenever v leaves $\mathrm{ball}(s, 5d)$, remove v from K.
- If at any point $|\mathrm{ball}(s, 2d)| \leq n/2$, restart with new source.

Analysis:

- If $|\mathrm{ball}(s, 2d)| \leq n/2$ then s is scattered.
- s picked at random, so in expectation half of vertices scattered.
- So w.h.p only $O(\log(n))$ random sources before termination.

Random Source Useless Against Adaptive Adversaries

Random Source: Let s be a random source in G

Non-Adaptive Adversary:

- Adversary has no access to randomness of algorithm.
- To scatter s, must scatter (in expectation) half of $V(G)$
- So only $\log(n)$ random sources.

Random Source Useless Against Adaptive Adversaries

Random Source: Let s be a random source in G

Non-Adaptive Adversary:

- Adversary has no access to randomness of algorithm.
- To scatter s, must scatter (in expectation) half of $V(G)$
- So only $\log(n)$ random sources.

Adaptive Adversary

- Adversary can guess randomness of algorithm via queries
- Can Show: easy for adversary to detect s.

Random Source Useless Against Adaptive Adversaries

Random Source: Let s be a random source in G

Non-Adaptive Adversary:

- Adversary has no access to randomness of algorithm.
- To scatter s, must scatter (in expectation) half of $V(G)$
- So only $\log(n)$ random sources.

Adaptive Adversary

- Adversary can guess randomness of algorithm via queries
- Can Show: easy for adversary to detect s.
- Adversary can delete all edges of s while leaving rest of G intact.
- Will need $\Omega(n)$ sources.

The random-source technique accounts for much of the gap between adaptive and non-adaptive algorithms for dynamic shortest paths and related problems.

Previous Adaptive Approach: Many Sources

Non-adaptive Adversary: maintain shortest path tree from *random* $s \in G$

Previous Adaptive Approach: Many Sources

Non-adaptive Adversary: maintain shortest path tree from *random* $s \in G$

Adaptive Adversary: maintain shortest path tree from *all* $v \in V(G)$.

- Can somewhat limit sizes of trees with density arguments.
- State-of-the art with many-source approach:
  - $\hat{O}(mn^{3/4})$ Bernstein and Chechik, 2016
  - $\hat{O}(mn^{1/2})$ Gutenberg and Wulff-Nilsen, 2016
- Hard barrier to this approach: $O(mn^{1/2})$.

Outline

Outline

Defining Vertex Expanders

Definition: For any set $L \subset V(G)$, let $N(L)$ be the neighbors of L *not in* L.

- So $L \cap N(L) = \emptyset$

Defining Vertex Expanders

Definition: For any set $L \subset V(G)$, let $N(L)$ be the neighbors of L *not in* L.

- So $L \cap N(L) = \emptyset$

Vertex Expander

$G = (V, E)$ is a vertex expander if for *any* set $L \subset V$ with $|L| \leq |V|/2$:

$$|N(L)| \geq O(|L|/\log(n)).$$

Defining Vertex Expanders

Definition: For any set $L \subset V(G)$, let $N(L)$ be the neighbors of L *not in* L.

- So $L \cap N(L) = \emptyset$

Vertex Expander

$G = (V, E)$ is a vertex expander if for *any* set $L \subset V$ with $|L| \leq |V|/2$:

$$|N(L)| \geq O(|L|/\log(n)).$$

This Talk: Only vertex expanders, expansion factor always $1/\log(n)$.

Defining Vertex Expanders

Definition: For any set $L \subset V(G)$, let $N(L)$ be the neighbors of L *not in* L.

- So $L \cap N(L) = \emptyset$

Vertex Expander

$G = (V, E)$ is a vertex expander if for *any* set $L \subset V$ with $|L| \leq |V|/2$:

$$|N(L)| \geq O(|L|/\log(n)).$$

This Talk: Only vertex expanders, expansion factor always $1/\log(n)$.

Key Property: expanders have diameter polylog

Existing Technique: Expander Pruning

Key fact: expanders are robust to edge deletions.

Existing Technique: Expander Pruning

Expander Pruning (slightly informal) [Saranurak, Wang]

Let G be an expander subject to edge deletions. Algorithm $\textsc{Prune}(G)$ can process up to $O(n/\log(n))$ edge deletions while maintaining a set $X \subset V(G)$ such that

- $G[X]$ is an expander.
- $|X| \geq V(G)/2$

Idealized Scenario: G is initially an expander

Scattering Property: If $v \in V(G) \setminus K$ then $|\mathrm{ball}(v, 2d)| \leq .99n$

RobustCore(G)

- Initially: $K \leftarrow V(G)$

Idealized Scenario: G is initially an expander

Scattering Property: If $v \in V(G) \setminus K$ then $|\mathrm{ball}(v, 2d)| \leq .99n$

RobustCore(G)

- Initially: $K \leftarrow V(G)$
- Maintain $X \leftarrow \textsc{Prune}(G)$

Idealized Scenario: G is initially an expander

Scattering Property: If $v \in V(G) \setminus K$ then $|\mathrm{ball}(v, 2d)| \le .99n$

RobustCore(G)

- Initially: $K \leftarrow V(G)$
- Maintain $X \leftarrow \textsc{Prune}(G)$
- Maintain $\mathrm{ball}(X, 2d)$.
  Can do in $\hat{O}(m)$ time (ES-tree).

Idealized Scenario: G is initially an expander

Scattering Property: If $v \in V(G) \setminus K$ then $|\mathrm{ball}(v, 2d)| \leq .99n$

RobustCore(G)

- Initially: $K \leftarrow V(G)$
- Maintain $X \leftarrow \textsc{Prune}(G)$
- Maintain $\mathrm{ball}(X, 2d)$.
  Can do in $\hat{O}(m)$ time (ES-tree).
- Whenever v leaves $\mathrm{ball}(X, 2d)$, remove v from K

Idealized Scenario: G is initially an expander

Scattering Property: If $v \in V(G) \setminus K$ then $|\mathrm{ball}(v, 2d)| \leq .99n$

RobustCore(G)

- Initially: $K \leftarrow V(G)$
- Maintain $X \leftarrow \textsc{Prune}(G)$
- Maintain $\mathrm{ball}(X, 2d)$.
  Can do in $\hat{O}(m)$ time (ES-tree).
- Whenever v leaves $\mathrm{ball}(X, 2d)$, remove v from K
- Terminate once $|X| < n/2$

Idealized Scenario: G is initially an expander

Scattering Property: If $v \in V(G) \setminus K$ then $|\mathrm{ball}(v, 2d)| \leq .99n$

RobustCore(G)

- Initially: $K \leftarrow V(G)$
- Maintain $X \leftarrow \textsc{Prune}(G)$
- Maintain $\mathrm{ball}(X, 2d)$.
  Can do in $\hat{O}(m)$ time (ES-tree).
- Whenever v leaves $\mathrm{ball}(X, 2d)$, remove v from K
- Terminate once $|X| < n/2$
  - Can only happen after $n/\log(n)$ deletions.

Idealized Scenario: G is initially an expander

Scattering Property: If $v \in V(G) \setminus K$ then $|\text{ball}(v, 2d)| \leq .99n$

RobustCore(G)

- Initially: $K \leftarrow V(G)$
- Maintain $X \leftarrow \textsc{Prune}(G)$
- Maintain $\text{ball}(X, 2d)$.
  Can do in $\hat{O}(m)$ time (ES-tree).
- Whenever v leaves $\text{ball}(X, 2d)$, remove v from K
- Terminate once $|X| < n/2$
  - Can only happen after $n/\log(n)$ deletions.

Intuition: shortest path tree rooted at expander instead of random source.

What if G is not an expander?
(Input to RobustCore only guarantees that G has small diameter.)

What if G is not an expander?
(Input to RobustCore only guarantees that G has small diameter.)

Note: unclear how to efficiently use expander decomposition

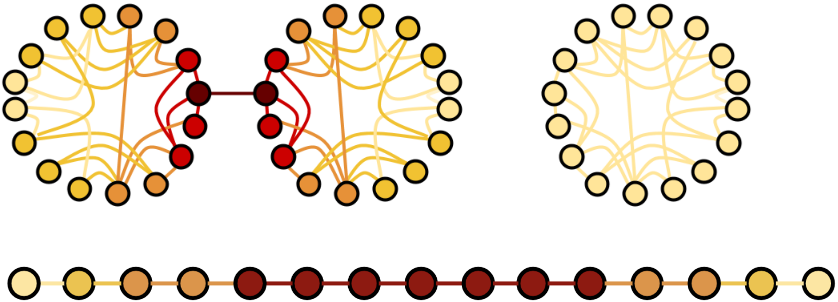
Our Result: expander tools without expander decomposition.

Outline

What if G not an expander?

Criticality: vertex v is critical if deleting edge (u, v) can scatter many vertices in G

1. G is expander: no vertices are critical
2. G is arbitrary graph: can have many very critical vertices
3. G has small diameter: total criticality is small.



Capacitated Expander

Regular Expander

$G = (V, E)$ is an expander if for *any* set $L \subset V$ with $|L| \leq |V|/2$:

$$|N(L)| \geq O(|L|/\log(n)).$$

Capacitated Expander

Regular Expander

$G = (V, E)$ is an expander if for *any* set $L \subset V$ with $|L| \le |V|/2$:

$$|N(L)| \geq O(|L|/\log(n)).$$

Capacitated Expander

G is a capacitated expander with respect to κ if for *any* set $L \subset V$ with $|L| \le |V|/2$:

$$\sum_{v \in N(L)} \kappa(v) \geq |L|/\log(n).$$

Capacity Function Examples

Capacitated Expander

G is a capacitated expander with respect to κ if for *any* set $L \subset V$ with $|L| \leq |V|/2$ we have: $\sum_{v \in N(L)} \kappa(v) \geq |L|/\log(n)$.

Capacity Function Examples

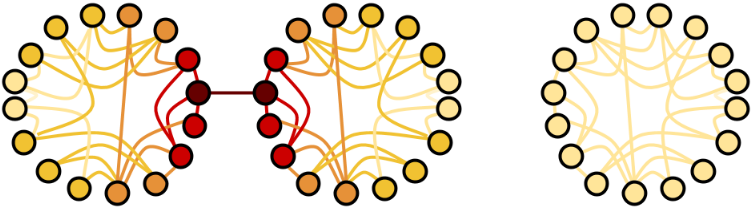
Capacitated Expander

G is a capacitated expander with respect to κ if for *any* set $L \subset V$ with $|L| \leq |V|/2$ we have: $\sum_{v \in N(L)} \kappa(v) \geq |L|/\log(n)$.

Capacity Function Examples

Capacitated Expander

G is a capacitated expander with respect to κ if for *any* set $L \subset V$ with $|L| \leq |V|/2$ we have: $\sum_{v \in N(L)} \kappa(v) \geq |L|/\log(n)$.



Goal: Low Total Capacity

Capacitated Expander

G is a capacitated expander with respect to κ if for *any* set $L \subset V$ with $|L| \leq |V|/2$ we have: $\sum_{v \in N(L)} \kappa(v) \geq |L|/\log(n)$.

Goal: Low Total Capacity

Capacitated Expander

G is a capacitated expander with respect to κ if for *any* set $L \subset V$ with $|L| \leq |V|/2$ we have: $\sum_{v \in N(L)} \kappa(v) \geq |L|/\log(n)$.

- Note: G automatically capacitated expander if $\kappa(v) = n \; \forall v \in V$

Goal: Low Total Capacity

Capacitated Expander

G is a capacitated expander with respect to κ if for *any* set $L \subset V$ with $|L| \leq |V|/2$ we have: $\sum_{v \in N(L)} \kappa(v) \geq |L| / \log(n)$.

- Note: G automatically capacitated expander if $\kappa(v) = n \; \forall v \in V$

High-Level Goal

Given a graph G, compute a capacity function κ such that G is a capacitated vertex expander and $\sum_{v \in V(G)} \kappa(v)$ is small.

Goal: Low Total Capacity

Capacitated Expander

G is a capacitated expander with respect to κ if for *any* set $L \subset V$ with $|L| \leq |V|/2$ we have: $\sum_{v \in N(L)} \kappa(v) \geq |L| / \log(n)$.

- Note: G automatically capacitated expander if $\kappa(v) = n \; \forall v \in V$

High-Level Goal

Given a graph G, compute a capacity function κ such that G is a capacitated vertex expander and $\sum_{v \in V(G)} \kappa(v)$ is small.

Question: Why do we want $\sum_{v \in V} \kappa(v)$ to be small.

Goal: Low Total Capacity

Capacitated Expander

G is a capacitated expander with respect to κ if for *any* set $L \subset V$ with $|L| \leq |V|/2$ we have: $\sum_{v \in N(L)} \kappa(v) \geq |L|/\log(n)$.

- Note: G automatically capacitated expander if $\kappa(v) = n \; \forall v \in V$

High-Level Goal

Given a graph G, compute a capacity function κ such that G is a capacitated vertex expander and $\sum_{v \in V(G)} \kappa(v)$ is small.

Question: Why do we want $\sum_{v \in V} \kappa(v)$ to be small.

Answer: vertices with low $\kappa(v)$ are not crucial, so adversarial deletions of edges incident to v are easy to process.

Capacitated Expander Pruning

Simplification: We assume that all vertices in G have constant degree.

Capacitated Expander Pruning

Simplification: We assume that all vertices in G have constant degree.

Definition: For edge $e = (u, v)$, let $\kappa(u, v) = \kappa(u) + \kappa(v)$.

- Note: $\sum_{v \in V(G)} \kappa(v) = \Theta(\sum_{e \in E(G)} \kappa(e))$

Capacitated Expander Pruning

Simplification: We assume that all vertices in G have constant degree.

Definition: For edge $e = (u, v)$, let $\kappa(u, v) = \kappa(u) + \kappa(v)$.

- Note: $\sum_{v \in V(G)} \kappa(v) = \Theta(\sum_{e \in E(G)} \kappa(e))$

$\textsc{Prune}(G, \kappa)$

Let G be a capacitated expander wr.t. to κ and say G subject to edge deletions. Algorithm $\textsc{Prune}(G, \kappa)$ can process edge deletions while maintaining a set $X \subset V(G)$ such that

- $G[X]$ is a capacitated expander.
- $|X| \geq V(G)/2$ as long as $\sum_{e \in E^{del}} \kappa(e) \leq O(n/\log(n))$, where E^{del} is the set of edges deleted by the adversary.

Key Lemma

Small Diameter Implies Small Capacity Sum

Say that $\mathrm{diam}(G) = d$. Then, there exists a function κ such that:

- G is a capacitated expander w.r.t κ.
- $\sum_{v \in V} \kappa(v) = \hat{O}(nd)$

Key Lemma

Small Diameter Implies Small Capacity Sum

Say that $\mathrm{diam}(G) = d$. Then, there exists a function κ such that:

- G is a capacitated expander w.r.t κ.
- $\sum_{v \in V} \kappa(v) = \hat{O}(nd)$

Key Lemma

Small Diameter Implies Small Capacity Sum

Say that $\mathrm{diam}(G) = d$. Then, there exists a function κ such that:

- G is a capacitated expander w.r.t κ.
- $\sum_{v \in V} \kappa(v) = \hat{O}(nd)$

Issue: Cannot compute above function κ in almost-linear time.

Key Lemma

Small Diameter Implies Small Capacity Sum

Say that $\mathrm{diam}(G) = d$. Then, there exists a function κ such that:

- G is a capacitated expander w.r.t κ.
- $\sum_{v \in V} \kappa(v) = \hat{O}(nd)$

Issue: Cannot compute above function κ in almost-linear time.

- We actually compute slightly relaxed version of κ that only guarantees expansion for *balanced* cuts: $|L| \geq \epsilon n$

Algorithm RobustCore(G)

Goal: given G with $\mathrm{diam}(G) = d = n^{o(1)}$, maintain small-diam $K \subseteq V$.

Algorithm RobustCore(G)

Goal: given G with $\mathrm{diam}(G) = d = n^{o(1)}$, maintain small-diam $K \subseteq V$.

1. Compute capacity function κ such that G is a capacitated expander and $\sum_{v \in V(G)} \kappa(v) = \hat{O}(nd) = \hat{O}(n)$.

Algorithm RobustCore(G)

Goal: given G with $\mathrm{diam}(G) = d = n^{o(1)}$, maintain small-diam $K \subseteq V$.

1. Compute capacity function κ such that G is a capacitated expander and $\sum_{v \in V(G)} \kappa(v) = \hat{O}(nd) = \hat{O}(n)$.
2. $X \leftarrow \textsc{Prune}(G, \kappa)$

Algorithm RobustCore(G)

Goal: given G with $\mathrm{diam}(G) = d = n^{o(1)}$, maintain small-diam $K \subseteq V$.

1. Compute capacity function κ such that G is a capacitated expander and $\sum_{v \in V(G)} \kappa(v) = \hat{O}(nd) = \hat{O}(n)$.
2. $X \leftarrow \textsc{Prune}(G, \kappa)$
3. Maintain $\mathrm{ball}(X, 2d)$

Algorithm RobustCore(G)

Goal: given G with $\mathrm{diam}(G) = d = n^{o(1)}$, maintain small-diam $K \subseteq V$.

1. Compute capacity function κ such that G is a capacitated expander and $\sum_{v \in V(G)} \kappa(v) = \hat{O}(nd) = \hat{O}(n)$.
2. $X \leftarrow \textsc{Prune}(G, \kappa)$
3. Maintain $\mathrm{ball}(X, 2d)$
4. If a vertex v leaves $\mathrm{ball}(X, 2d)$, remove v from k

---

Algorithm RobustCore(G)

Goal: given G with $\mathrm{diam}(G) = d = n^{o(1)}$, maintain small-diam $K \subseteq V$.

1. Compute capacity function κ such that G is a capacitated expander and $\sum_{v \in V(G)} \kappa(v) = \hat{O}(nd) = \hat{O}(n)$.
2. $X \leftarrow \textsc{Prune}(G, \kappa)$
3. Maintain $\mathrm{ball}(X, 2d)$
4. If a vertex v leaves $\mathrm{ball}(X, 2d)$, remove v from k
5. If at ant point $|X| \leq n/2$: restart from scratch with current K
   - We call this a new *phase*

Algorithm RobustCore(G)

Goal: given G with $\mathrm{diam}(G) = d = n^{o(1)}$, maintain small-diam $K \subseteq V$.

1. Compute capacity function κ such that G is a capacitated expander and $\sum_{v \in V(G)} \kappa(v) = \hat{O}(nd) = \hat{O}(n)$.
2. $X \leftarrow \textsc{Prune}(G, \kappa)$
3. Maintain $\mathrm{ball}(X, 2d)$
4. If a vertex v leaves $\mathrm{ball}(X, 2d)$, remove v from k
5. If at ant point $|X| \leq n/2$: restart from scratch with current K
   - We call this a new *phase*

Analysis

Algorithm RobustCore(G)

Goal: given G with $\mathrm{diam}(G) = d = n^{o(1)}$, maintain small-diam $K \subseteq V$.

1. Compute capacity function κ such that G is a capacitated expander and $\sum_{v \in V(G)} \kappa(v) = \hat{O}(nd) = \hat{O}(n)$.
2. $X \leftarrow \textsc{Prune}(G, \kappa)$
3. Maintain $\mathrm{ball}(X, 2d)$
4. If a vertex v leaves $\mathrm{ball}(X, 2d)$, remove v from k
5. If at ant point $|X| \leq n/2$: restart from scratch with current K
   - We call this a new *phase*

Analysis

- Each phase requires time $\hat{O}(nd) = \hat{O}(n)$.

Algorithm RobustCore(G)

Goal: given G with $\mathrm{diam}(G) = d = n^{o(1)}$, maintain small-diam $K \subseteq V$.

1. Compute capacity function κ such that G is a capacitated expander and $\sum_{v \in V(G)} \kappa(v) = \hat{O}(nd) = \hat{O}(n)$.
2. $X \leftarrow \textsc{Prune}(G, \kappa)$
3. Maintain $\mathrm{ball}(X, 2d)$
4. If a vertex v leaves $\mathrm{ball}(X, 2d)$, remove v from k
5. If at ant point $|X| \leq n/2$: restart from scratch with current K
   - We call this a new *phase*

Analysis

- Each phase requires time $\hat{O}(nd) = \hat{O}(n)$.
- Each phase terminates only after adversary deletes $n/\log(n)$ capacity.

Algorithm RobustCore(G)

Goal: given G with $\mathrm{diam}(G) = d = n^{o(1)}$, maintain small-diam $K \subseteq V$.

1. Compute capacity function κ such that G is a capacitated expander and $\sum_{v \in V(G)} \kappa(v) = \hat{O}(nd) = \hat{O}(n)$.
2. $X \leftarrow \textsc{Prune}(G, \kappa)$
3. Maintain $\mathrm{ball}(X, 2d)$
4. If a vertex v leaves $\mathrm{ball}(X, 2d)$, remove v from k
5. If at ant point $|X| \leq n/2$: restart from scratch with current K
   - We call this a new *phase*

Analysis

- Each phase requires time $\hat{O}(nd) = \hat{O}(n)$.
- Each phase terminates only after adversary deletes $n/\log(n)$ capacity.
- Total capacity $\sum_{v \in V} \kappa(v) = \hat{O}(n)$

Algorithm RobustCore(G)

Goal: given G with $\mathrm{diam}(G) = d = n^{o(1)}$, maintain small-diam $K \subseteq V$.

1. Compute capacity function κ such that G is a capacitated expander and $\sum_{v \in V(G)} \kappa(v) = \hat{O}(nd) = \hat{O}(n)$.
2. $X \leftarrow \textsc{Prune}(G, \kappa)$
3. Maintain $\mathrm{ball}(X, 2d)$
4. If a vertex v leaves $\mathrm{ball}(X, 2d)$, remove v from k
5. If at ant point $|X| \leq n/2$: restart from scratch with current K
   - We call this a new *phase*

Analysis

- Each phase requires time $\hat{O}(nd) = \hat{O}(n)$.
- Each phase terminates only after adversary deletes $n/\log(n)$ capacity.
- Total capacity $\sum_{v \in V} \kappa(v) = \hat{O}(n)$
- So # phases $= \hat{O}(\frac{n}{n/\log(n)}) = n^{o(1)}$.

Algorithm RobustCore(G)

Goal: given G with $\mathrm{diam}(G) = d = n^{o(1)}$, maintain small-diam $K \subseteq V$.

1. Compute capacity function κ such that G is a capacitated expander and $\sum_{v \in V(G)} \kappa(v) = \hat{O}(nd) = \hat{O}(n)$.
2. $X \leftarrow \textsc{Prune}(G, \kappa)$
3. Maintain $\mathrm{ball}(X, 2d)$
4. If a vertex v leaves $\mathrm{ball}(X, 2d)$, remove v from k
5. If at ant point $|X| \leq n/2$: restart from scratch with current K
   - We call this a new *phase*

Analysis

- Each phase requires time $\hat{O}(nd) = \hat{O}(n)$.
- Each phase terminates only after adversary deletes $n/\log(n)$ capacity.
- Total capacity $\sum_{v \in V} \kappa(v) = \hat{O}(n)$
- So # phases $= \hat{O}(\frac{n}{n/\log(n)}) = n^{o(1)}$.

Technical Note: above analysis requires that $\kappa(v)$ is monotonically increasing between phases.

Takeaway

Key Lemma

Say that $\mathrm{diam}(G) = d$. Then, there exists a function κ such that:

- G is a capacitated expander w.r.t κ.
- $\sum_{v \in V} \kappa(v) = \hat{O}(nd)$

Takeaway

Key Lemma

Say that $\mathrm{diam}(G) = d$. Then, there exists a function κ such that:

- G is a capacitated expander w.r.t κ.
- $\sum_{v \in V} \kappa(v) = \hat{O}(nd)$

Takeaway: Can turn *any* low-diameter graph into an expander and apply expander tools.

Outline

Computing the Function κ via Congestion Balancing.

Capacitated Expander

G is a capacitated expander with respect to κ if for *any* set $L \subset V$ with $|L| \leq |V|/2$ we have: $\sum_{v \in N(L)} \kappa(v) \geq |L|/\log(n)$.

Key Lemma

Say that $\mathrm{diam}(G) = d$. Then there exists κ such that G is a capacitated expander w.r.t κ and $\sum_{v \in V} \kappa(v) = \hat{O}(nd)$.

Computing the Function κ via Congestion Balancing.

Capacitated Expander

G is a capacitated expander with respect to κ if for *any* set $L \subset V$ with $|L| \leq |V|/2$ we have: $\sum_{v \in N(L)} \kappa(v) \geq |L|/\log(n)$.

Key Lemma

Say that $\mathrm{diam}(G) = d$. Then there exists κ such that G is a capacitated expander w.r.t κ and $\sum_{v \in V} \kappa(v) = \hat{O}(nd)$.

Constructing Desired Function κ

1. Initially set $\kappa(v) = 1 \;\; \forall v \in V(G)$.

Computing the Function κ via Congestion Balancing.

Capacitated Expander

G is a capacitated expander with respect to κ if for *any* set $L \subset V$ with $|L| \leq |V|/2$ we have: $\sum_{v \in N(L)} \kappa(v) \geq |L|/\log(n)$.

Key Lemma

Say that $\mathrm{diam}(G) = d$. Then there exists κ such that G is a capacitated expander w.r.t κ and $\sum_{v \in V} \kappa(v) = \hat{O}(nd)$.

Constructing Desired Function κ

1. Initially set $\kappa(v) = 1 \;\; \forall v \in V(G)$.
2. While there exists L such that $\sum_{v \in N(L)} \kappa(v) < |L|/\log(n)$

Computing the Function κ via Congestion Balancing.

Capacitated Expander

G is a capacitated expander with respect to κ if for *any* set $L \subset V$ with $|L| \leq |V|/2$ we have: $\sum_{v \in N(L)} \kappa(v) \geq |L|/\log(n)$.

Key Lemma

Say that $\mathrm{diam}(G) = d$. Then there exists κ such that G is a capacitated expander w.r.t κ and $\sum_{v \in V} \kappa(v) = \hat{O}(nd)$.

Constructing Desired Function κ

1. Initially set $\kappa(v) = 1 \ \ \forall v \in V(G)$.
2. While there exists L such that $\sum_{v \in N(L)} \kappa(v) < |L|/\log(n)$
   - Do $\forall v \in N(L)$: $\kappa(v) \leftarrow 2\kappa(v)$

Computing the Function κ via Congestion Balancing.

Capacitated Expander

G is a capacitated expander with respect to κ if for *any* set $L \subset V$ with $|L| \leq |V|/2$ we have: $\sum_{v \in N(L)} \kappa(v) \geq |L| / \log(n)$.

Key Lemma

Say that $\mathrm{diam}(G) = d$. Then there exists κ such that G is a capacitated expander w.r.t κ and $\sum_{v \in V} \kappa(v) = \hat{O}(nd)$.

Constructing Desired Function κ

1. Initially set $\kappa(v) = 1 \;\; \forall v \in V(G)$.
2. While there exists L such that $\sum_{v \in N(L)} \kappa(v) < |L| / \log(n)$
   - Do $\forall v \in N(L)$: $\kappa(v) \leftarrow 2\kappa(v)$

Generalizes Congestion Balancing Technique from [BGS20]

Analysis of Congestion Balancing

Defining the Potential Function

Analysis of Congestion Balancing

Defining the Potential Function

- Define the *cost* of a vertex v to be $c(v) \triangleq \log(\kappa(v))$

Analysis of Congestion Balancing

Defining the Potential Function

- Define the *cost* of a vertex v to be $c(v) \triangleq \log(\kappa(v))$
- Let $\Pi(G, \kappa)$ be the cost of the min-cost embedding (unbounded capacities) of a constant-degree expander into G.

Analysis of Congestion Balancing

Defining the Potential Function

- Define the *cost* of a vertex v to be $c(v) \triangleq \log(\kappa(v))$
- Let $\Pi(G, \kappa)$ be the cost of the min-cost embedding (unbounded capacities) of a constant-degree expander into G.

Key Facts

- $\Pi(G, \kappa)$ increases monotonically from 0 to $\tilde{O}(nd)$

Analysis of Congestion Balancing

Defining the Potential Function

- Define the *cost* of a vertex v to be $c(v) \triangleq \log(\kappa(v))$
- Let $\Pi(G, \kappa)$ be the cost of the min-cost embedding (unbounded capacities) of a constant-degree expander into G.

Key Facts

- $\Pi(G, \kappa)$ increases monotonically from 0 to $\tilde{O}(nd)$
- Whenever $\sum_{v \in V} \kappa(v)$ increases by Δ, $\Pi(G, \kappa)$ increases by $\hat{\Omega}(\Delta)$

Analysis of Congestion Balancing

Defining the Potential Function

- Define the *cost* of a vertex v to be $c(v) \triangleq \log(\kappa(v))$
- Let $\Pi(G, \kappa)$ be the cost of the min-cost embedding (unbounded capacities) of a constant-degree expander into G.

Key Facts

- $\Pi(G, \kappa)$ increases monotonically from 0 to $\tilde{O}(nd)$
- Whenever $\sum_{v \in V} \kappa(v)$ increases by Δ, $\Pi(G, \kappa)$ increases by $\hat{\Omega}(\Delta)$
- Thus, at the end, $\sum_{v \in V} \kappa(v) = \hat{O}(nd)$

Outline

Summary: Simplified RobustCore

Input: Graph G subject to edge deletions; initially $\mathrm{diam}(G) = d = n^{o(1)}$.
Define $|V(G) = n|$, $|E(G) = m|$.
High-Level Goal: maintain small-diameter set K inside G.

Summary: Simplified RobustCore

Input: Graph G subject to edge deletions; initially $\mathrm{diam}(G) = d = n^{o(1)}$.
Define $|V(G) = n|$, $|E(G) = m|$.
High-Level Goal: maintain small-diameter set K inside G.

Simplified RobustCore(G)

Maintain a set $K \subseteq V(G)$ with the following properties:

- **Diameter Property:** $\mathrm{diam}_G(K) = \hat{O}(d) = n^{o(1)}$
- **Scattering Property:** For every $v \in V(G) \setminus K$ we have
$$|\mathrm{ball}(v, 2d)| \leq .99n$$
- **Termination:** if at some point $|K| \leq n/2$, can set $K \leftarrow \emptyset$.

Result: Can maintain RobustCore(G) in total time $\hat{O}(m)$ over all deletions.

Summary: Capacitated Expansion

Capacitated Expander

G is a capacitated expander with respect to κ is for every $L \subset V$ with $L \geq |V|/2$ we have:

$$\sum_{v \in S} \kappa(s) \geq |L| / \log(n)$$

.

Summary: Capacitated Expansion

Capacitated Expander

G is a capacitated expander with respect to κ is for every $L \subset V$ with $L \geq |V|/2$ we have:

$$\sum_{v \in S} \kappa(s) \geq |L|/\log(n)$$

.

Key Lemma

Say that $\mathrm{diam}(G) = d$. Then there exists κ such that G is a capacitated expander w.r.t κ and $\sum_{v \in V} \kappa(v) = \hat{O}(nd)$.

Summary: Dynamic Shortest Path Result

Previous Work (undirected graph)

- **Non-Adaptive:** $\hat{O}(m)$ total update time.
- **Deterministic (and so adaptive):** $\hat{O}(\min(n^2, m\sqrt{n})$ total update time.

Summary: Dynamic Shortest Path Result

Previous Work (undirected graph)

- **Non-Adaptive:** $\hat{O}(m)$ total update time.
- **Deterministic (and so adaptive):** $\hat{O}(\min(n^2, m\sqrt{n})$ total update time.

Our Result (undirected graph)

Adaptive decremental SSSP in total update time $\hat{O}(m)$

- Closes the adaptive / non-adaptive gap.
- Optimal update time up to sub-polynomial factors.
- Generalizes to weighted graphs
- Concludes long line of research.

Summary: Flow Results

Previous $(1+\epsilon)$-Approximate Max Flow in Undirected Graphs

- Edge-Capacitated Max Flow: $\tilde{O}(m)$
- Vertex-Capacitated Max Flow: $\hat{O}(m + n^{1.5})$
- Min-Cost flow: $\hat{O}(m + n^{1.5})$

Summary: Flow Results

Previous $(1+\epsilon)$-Approximate Max Flow in Undirected Graphs

- Edge-Capacitated Max Flow: $\tilde{O}(m)$
- Vertex-Capacitated Max Flow: $\hat{O}(m + n^{1.5})$
- Min-Cost flow: $\hat{O}(m + n^{1.5})$

Our Result

$(1+\epsilon)$-approximation min-cost flow in $\hat{O}(m)$ time.

- can handle costs/capacities on both vertices/edges.
- Completes the picture for approximate flow in undirected graphs.

Summary: Flow Results

Previous $(1+\epsilon)$-Approximate Max Flow in Undirected Graphs

- Edge-Capacitated Max Flow: $\tilde{O}(m)$
- Vertex-Capacitated Max Flow: $\hat{O}(m+n^{1.5})$
- Min-Cost flow: $\hat{O}(m+n^{1.5})$

Our Result

$(1+\epsilon)$-approximation min-cost flow in $\hat{O}(m)$ time.

- can handle costs/capacities on both vertices/edges.
- Completes the picture for approximate flow in undirected graphs.

Techniques:

- New version of MWU framework for max flow
- Uses decremental SSSP algorithm as black box.

Open Problems

1) Decremental SSSP for directed graphs

Open Problems

1) Decremental SSSP for directed graphs

2) Close adaptive/non-adaptive gaps for other dynamic algorithms.

Open Problems

1) Decremental SSSP for directed graphs

2) Close adaptive/non-adaptive gaps for other dynamic algorithms.

3) Combine dynamic algorithms with MWU to develop faster static algorithms.

Open Problems

1) Decremental SSSP for directed graphs

2) Close adaptive/non-adaptive gaps for other dynamic algorithms.

3) Combine dynamic algorithms with MWU to develop faster static algorithms.

Thanks!