Faster Parallel Algorithm for Approximate Shortest Path

Jason Li (CMU)
STOC 2020

March 2, 2020

Introduction

Approximate single-source shortest path (SSSP)...

Approximate single-source shortest path (SSSP)...

- Input: undirected graph with nonnegative weights, and a source vertex s

Approximate single-source shortest path (SSSP)...

- Input: undirected graph with nonnegative weights, and a source vertex s
- Output (distances): approximations $\tilde{d}(v)$ for all $v \in V$ satisfying $d_G(s,v) \leq \tilde{d}(v) \leq (1+\epsilon)d_G(s,v)$

Approximate single-source shortest path (SSSP)...

- Input: undirected graph with nonnegative weights, and a source vertex s
- Output (distances): approximations $\tilde{d}(v)$ for all $v \in V$ satisfying $d_G(s, v) \leq \tilde{d}(v) \leq (1 + \epsilon)d_G(s, v)$
- Output (tree): a spanning tree T satisfying $d_G(s, v) \leq \tilde{d}_T(s, v) \leq (1 + \epsilon)d_G(s, v)$

Introduction

Approximate single-source shortest path (SSSP)...

- Input: undirected graph with nonnegative weights, and a source vertex s
- Output (distances): approximations $\tilde{d}(v)$ for all $v \in V$ satisfying $d_G(s,v) \leq \tilde{d}(v) \leq (1+\epsilon)d_G(s,v)$
- Output (tree): a spanning tree T satisfying $d_G(s,v) \leq \tilde{d}_T(s,v) \leq (1+\epsilon)d_G(s,v)$

...in parallel

Introduction

Approximate single-source shortest path (SSSP)...

- Input: undirected graph with nonnegative weights, and a source vertex s
- Output (distances): approximations $\tilde{d}(v)$ for all $v \in V$ satisfying $d_G(s, v) \leq \tilde{d}(v) \leq (1 + \epsilon) d_G(s, v)$
- Output (tree): a spanning tree T satisfying $d_G(s, v) \leq \tilde{d}_T(s, v) \leq (1 + \epsilon) d_G(s, v)$

...in parallel

- PRAM model: parallel **foreach**, runs each loop independently in parallel

Introduction

Approximate single-source shortest path (SSSP)...

- Input: undirected graph with nonnegative weights, and a source vertex s
- Output (distances): approximations $\tilde{d}(v)$ for all $v \in V$ satisfying $d_G(s,v) \leq \tilde{d}(v) \leq (1+\epsilon)d_G(s,v)$
- Output (tree): a spanning tree T satisfying $d_G(s,v) \leq \tilde{d}_T(s,v) \leq (1+\epsilon)d_G(s,v)$

...in parallel

- PRAM model: parallel **foreach**, runs each loop independently in parallel
- Work: sum of running times of each loop

Introduction

Approximate single-source shortest path (SSSP)...

- Input: undirected graph with nonnegative weights, and a source vertex s
- Output (distances): approximations $\tilde{d}(v)$ for all $v \in V$ satisfying $d_G(s,v) \le \tilde{d}(v) \le (1+\epsilon)d_G(s,v)$
- Output (tree): a spanning tree T satisfying $d_G(s,v) \le \tilde{d}_T(s,v) \le (1+\epsilon)d_G(s,v)$

...in parallel

- PRAM model: parallel **foreach**, runs each loop independently in parallel
- Work: sum of running times of each loop
- Time/Span: max of running times

Past work

Results

Past work

- Cohen ['94]: $m^{1+\delta}$ work and polylog(n) time for any constant $\delta > 0$

Results

Past work

- Cohen ['94]: $m^{1+\delta}$ work and polylog(n) time for any constant $\delta > 0$
  - Introduced the concept of **hopsets**, "shortcut edges"

Results

Past work

- Cohen ['94]: $m^{1+\delta}$ work and polylog(n) time for any constant $\delta > 0$
  - Introduced the concept of **hopsets**, "shortcut edges"
- polylog(n) factor improved by Elkin and Neiman ['18]

Results

Past work

- Cohen ['94]: $m^{1+\delta}$ work and polylog(n) time for any constant $\delta > 0$
  - Introduced the concept of **hopsets**, "shortcut edges"
- polylog(n) factor improved by Elkin and Neiman ['18]
- Open: m polylog(n) work and polylog(n) time

Results

Past work

- Cohen ['94]: $m^{1+\delta}$ work and polylog(n) time for any constant $\delta > 0$
    - Introduced the concept of **hopsets**, "shortcut edges"
- polylog(n) factor improved by Elkin and Neiman ['18]
- Open: m polylog(n) work and polylog(n) time
- Surprising **lower bound**: no hopset-based m polylog(n) work and polylog(n) time algorithm!

Results

Past work

- Cohen ['94]: $m^{1+\delta}$ work and polylog(n) time for any constant $\delta > 0$
  - Introduced the concept of **hopsets**, "shortcut edges"
- polylog(n) factor improved by Elkin and Neiman ['18]
- Open: m polylog(n) work and polylog(n) time
- Surprising **lower bound**: no hopset-based m polylog(n) work and polylog(n) time algorithm!

Our result

Results

Past work

- Cohen ['94]: $m^{1+\delta}$ work and polylog(n) time for any constant $\delta > 0$
  - Introduced the concept of **hopsets**, “shortcut edges”
- polylog(n) factor improved by Elkin and Neiman ['18]
- Open: m polylog(n) work and polylog(n) time
- Surprising **lower bound**: no hopset-based m polylog(n) work and polylog(n) time algorithm!

Our result

- m polylog(n) work and polylog(n) time via **continuous optimization**

Results

Past work

- Cohen ['94]: $m^{1+\delta}$ work and polylog(n) time for any constant $\delta > 0$
  - Introduced the concept of **hopsets**, "shortcut edges"
- polylog(n) factor improved by Elkin and Neiman ['18]
- Open: m polylog(n) work and polylog(n) time
- Surprising **lower bound**: no hopset-based m polylog(n) work and polylog(n) time algorithm!

Our result

- m polylog(n) work and polylog(n) time via **continuous optimization**
- Study a **continuous relaxation** of SSSP, the **minimum transshipment** problem

Results

Past work

- Cohen ['94]: $m^{1+\delta}$ work and polylog(n) time for any constant $\delta > 0$
  - Introduced the concept of **hopsets**, "shortcut edges"
- polylog(n) factor improved by Elkin and Neiman ['18]
- Open: m polylog(n) work and polylog(n) time
- Surprising **lower bound**: no hopset-based m polylog(n) work and polylog(n) time algorithm!

Our result

- m polylog(n) work and polylog(n) time via **continuous optimization**
- Study a **continuous relaxation** of SSSP, the **minimum transshipment** problem
- Concurrently: Andoni, Stein, Zhong [STOC'20] obtain the same result with similar techniques

Transshipment

Transshipment, a.k.a. uncapacitated min-cost flow

Transshipment, a.k.a. uncapacitated min-cost flow

- Input: graph with vertex-edge incidence matrix $A \in \mathbb{R}^{V \times E}$ and demand vector $b \in \mathbb{R}^V$ satisfying $\sum_v b_v = 0$

Transshipment, a.k.a. uncapacitated min-cost flow

- Input: graph with vertex-edge incidence matrix $A \in \mathbb{R}^{V \times E}$ and demand vector $b \in \mathbb{R}^V$ satisfying $\sum_v b_v = 0$
- Constraint: a flow vector $f \in \mathbb{R}^E$ satisfying the flow constraint $Af = b$

Transshipment

Transshipment, a.k.a. uncapacitated min-cost flow

- Input: graph with vertex-edge incidence matrix $A \in \mathbb{R}^{V \times E}$ and demand vector $b \in \mathbb{R}^V$ satisfying $\sum_v b_v = 0$
- Constraint: a flow vector $f \in \mathbb{R}^E$ satisfying the flow constraint $Af = b$
- Objective: minimize $\|Cf\|_1 = \sum_e c_e f_e$

Transshipment

Transshipment, a.k.a. uncapacitated min-cost flow

- Input: graph with vertex-edge incidence matrix $A \in \mathbb{R}^{V \times E}$ and demand vector $b \in \mathbb{R}^V$ satisfying $\sum_v b_v = 0$
- Constraint: a flow vector $f \in \mathbb{R}^E$ satisfying the flow constraint $Af = b$
- Objective: minimize $\|Cf\|_1 = \sum_e c_e f_e$
- ℓ_1 version of max-flow (which is minimize $\|Cf\|_\infty$)

Transshipment, a.k.a. uncapacitated min-cost flow

- Input: graph with vertex-edge incidence matrix $A \in \mathbb{R}^{V \times E}$ and demand vector $b \in \mathbb{R}^V$ satisfying $\sum_v b_v = 0$
- Constraint: a flow vector $f \in \mathbb{R}^E$ satisfying the flow constraint $Af = b$
- Objective: minimize $\|Cf\|_1 = \sum_e c_e f_e$
- ℓ_1 version of max-flow (which is minimize $\|Cf\|_\infty$)
- If $b = \sum_v (1_v - 1_s)$, then best flow sends 1 unit along shortest s–v path for each $v \neq s$

Transshipment, a.k.a. uncapacitated min-cost flow

- Input: graph with vertex-edge incidence matrix $A \in \mathbb{R}^{V \times E}$ and demand vector $b \in \mathbb{R}^V$ satisfying $\sum_v b_v = 0$
- Constraint: a flow vector $f \in \mathbb{R}^E$ satisfying the flow constraint $Af = b$
- Objective: minimize $\|Cf\|_1 = \sum_e c_e f_e$
- ℓ_1 version of max-flow (which is minimize $\|Cf\|_\infty$)
- If $b = \sum_v (1_v - 1_s)$, then best flow sends 1 unit along shortest s–v path for each $v \neq s$
  $\implies$ generalizes SSSP in exact case

Transshipment

Transshipment, a.k.a. uncapacitated min-cost flow

- Input: graph with vertex-edge incidence matrix $A \in \mathbb{R}^{V \times E}$ and demand vector $b \in \mathbb{R}^V$ satisfying $\sum_v b_v = 0$
- Constraint: a flow vector $f \in \mathbb{R}^E$ satisfying the flow constraint $Af = b$
- Objective: minimize $\|Cf\|_1 = \sum_e c_e f_e$
- ℓ_1 version of max-flow (which is minimize $\|Cf\|_\infty$)
- If $b = \sum_v (1_v - 1_s)$, then best flow sends 1 unit along shortest s–v path for each $v \neq s$
  $\implies$ generalizes SSSP in exact case
- **Approximate versions do not generalize!**

Transshipment, a.k.a. uncapacitated min-cost flow

- Input: graph with vertex-edge incidence matrix $A \in \mathbb{R}^{V \times E}$ and demand vector $b \in \mathbb{R}^V$ satisfying $\sum_v b_v = 0$
- Constraint: a flow vector $f \in \mathbb{R}^E$ satisfying the flow constraint $Af = b$
- Objective: minimize $\|Cf\|_1 = \sum_e c_e f_e$
- ℓ_1 version of max-flow (which is minimize $\|Cf\|_\infty$)
- If $b = \sum_v (1_v - 1_s)$, then best flow sends 1 unit along shortest s–v path for each $v \neq s$
  $\implies$ generalizes SSSP in exact case
- **Approximate versions do not generalize!**
- But can reduce approximate SSSP to polylog(n) many approximate transshipment calls

Sherman's framework

Sherman's framework

- Originally used by Sherman ['13] to solve $(1+\epsilon)$-approximate max flow

Sherman's framework

- Originally used by Sherman ['13] to solve $(1+\epsilon)$-approximate max flow
- Reduces $(1+\epsilon)$-approximate transshipment to computing a polylog(n)-approximate ℓ_1-**oblivious routing** scheme

Sherman's framework

- Originally used by Sherman ['13] to solve $(1+\epsilon)$-approximate max flow
- Reduces $(1+\epsilon)$-approximate transshipment to computing a polylog(n)-approximate ℓ_1-**oblivious routing** scheme
- Reduction can be interpreted as multiplicative weights update

Sherman's framework

- Originally used by Sherman ['13] to solve $(1+\epsilon)$-approximate max flow
- Reduces $(1+\epsilon)$-approximate transshipment to computing a polylog(n)-approximate ℓ_1**-oblivious routing** scheme
- Reduction can be interpreted as multiplicative weights update

ℓ_1-oblivious routing

Sherman's framework

- Originally used by Sherman ['13] to solve $(1+\epsilon)$-approximate max flow
- Reduces $(1+\epsilon)$-approximate transshipment to computing a polylog(n)-approximate ℓ_1**-oblivious routing** scheme
- Reduction can be interpreted as multiplicative weights update

ℓ_1-oblivious routing

- "ℓ_1" version of standard oblivious routing for max flow

Sherman's framework

Sherman's framework

- Originally used by Sherman ['13] to solve $(1+\epsilon)$-approximate max flow
- Reduces $(1+\epsilon)$-approximate transshipment to computing a polylog(n)-approximate ℓ_1**-oblivious routing** scheme
- Reduction can be interpreted as multiplicative weights update

ℓ_1-oblivious routing

- "ℓ_1" version of standard oblivious routing for max flow
- **Main technical contribution**: ℓ_1-oblivious routing in $\tilde{O}(m)$ work and polylog(n) time given an ℓ_1**-embedding** of the graph

Reducing to ℓ_1-metric

Reducing to ℓ_1-metric

- Bourgain's embedding: can embed a graph metric into $O(\log n)$ dimensions distortion $O(\log n)$ (under ℓ_2 metric)

Reducing to ℓ_1-metric

- Bourgain's embedding: can embed a graph metric into $O(\log n)$ dimensions distortion $O(\log n)$ (under ℓ_2 metric) $\implies O(\log^{1.5} n)$ distortion under ℓ_1 metric

Oblivious routing

Reducing to ℓ_1-metric

- Bourgain's embedding: can embed a graph metric into $O(\log n)$ dimensions distortion $O(\log n)$ (under ℓ_2 metric) $\implies O(\log^{1.5} n)$ distortion under ℓ_1 metric
- Not clear how to do in parallel! (more later)

Oblivious routing

Reducing to ℓ_1-metric

- Bourgain's embedding: can embed a graph metric into $O(\log n)$ dimensions distortion $O(\log n)$ (under ℓ_2 metric) $\implies O(\log^{1.5} n)$ distortion under ℓ_1 metric
- Not clear how to do in parallel! (more later)
- But if we can do this, then reduces to solving ℓ_1-oblivious routing on ℓ_1-metric

Reducing to ℓ_1-metric

- Bourgain's embedding: can embed a graph metric into $O(\log n)$ dimensions distortion $O(\log n)$ (under ℓ_2 metric) $\implies O(\log^{1.5} n)$ distortion under ℓ_1 metric
- Not clear how to do in parallel! (more later)
- But if we can do this, then reduces to solving ℓ_1-oblivious routing on ℓ_1-metric
- Purely **geometric** problem now: vertices are just points in ℓ_1-space

Oblivious routing

Oblivious routing on ℓ_1-metric

Oblivious routing on ℓ_1-metric

- Input: set of points $V \subseteq \mathbb{Z}^d$, and "demand" function $b : V \to \mathbb{R}$ ($\sum_{v \in V} b(v) = 0$) that is unknown to us.

Oblivious routing on ℓ_1-metric

- Input: set of points $V \subseteq \mathbb{Z}^d$, and "demand" function $b : V \to \mathbb{R}$ ($\sum_{v \in V} b(v) = 0$) that is unknown to us.
- On each step, choose any two points $x, y \in \mathbb{Z}^d$ and a scalar $c \in \mathbb{R}$, and "shift" c times the demand at x to location y. That is, we simultaneously update $b(x) \leftarrow b(x) - c \cdot b(x)$ and $b(y) \leftarrow b(y) + c \cdot b(x)$. Pay $c \cdot b(x) \cdot |x - y|$ total cost for this step. (We not know how much we pay!)

Oblivious routing on ℓ_1-metric

- Input: set of points $V \subseteq \mathbb{Z}^d$, and "demand" function $b : V \to \mathbb{R}$ ($\sum_{v \in V} b(v) = 0$) that is unknown to us.
- On each step, choose any two points $x, y \in \mathbb{Z}^d$ and a scalar $c \in \mathbb{R}$, and "shift" c times the demand at x to location y. That is, we simultaneously update $b(x) \leftarrow b(x) - c \cdot b(x)$ and $b(y) \leftarrow b(y) + c \cdot b(x)$. Pay $c \cdot b(x) \cdot |x - y|$ total cost for this step. (We not know how much we pay!)
- After some steps, declare that we are done. At this point, we must be certain that the demand is 0 everywhere: $b(x) = 0$ for all $x \in \mathbb{Z}^d$.

Oblivious routing on ℓ_1-metric

- Input: set of points $V \subseteq \mathbb{Z}^d$, and "demand" function $b : V \to \mathbb{R}$ ($\sum_{v \in V} b(v) = 0$) that is unknown to us.
- On each step, choose any two points $x, y \in \mathbb{Z}^d$ and a scalar $c \in \mathbb{R}$, and "shift" c times the demand at x to location y. That is, we simultaneously update $b(x) \leftarrow b(x) - c \cdot b(x)$ and $b(y) \leftarrow b(y) + c \cdot b(x)$. Pay $c \cdot b(x) \cdot |x - y|$ total cost for this step. (We not know how much we pay!)
- After some steps, declare that we are done. At this point, we must be certain that the demand is 0 everywhere: $b(x) = 0$ for all $x \in \mathbb{Z}^d$.
- Once we are done, learn the set of initial demands, sum up our total cost, and compare it to the optimal strategy we could have taken if we had known the demands beforehand. Want polylog(n)-approximation.

Algorithm intuition

Oblivious routing

Algorithm intuition

- Should be **unbiased**: demand from a given vertex should be spread **evenly**

Algorithm intuition

- Should be **unbiased**: demand from a given vertex should be spread **evenly**
- Sherman's algorithm ['17]: generalizes 1-d case, routes each point to all 2^d corners of the cube ($d = \sqrt{\log n}$, get $2^{\sqrt{\log n}}$ factor)

Oblivious routing

Algorithm intuition

- Should be **unbiased**: demand from a given vertex should be spread **evenly**
- Sherman's algorithm ['17]: generalizes 1-d case, routes each point to all 2^d corners of the cube ($d = \sqrt{\log n}$, get $2^{\sqrt{\log n}}$ factor)

Our algorithm

Oblivious routing

Algorithm intuition

- Should be **unbiased**: demand from a given vertex should be spread **evenly**
- Sherman's algorithm ['17]: generalizes 1-d case, routes each point to all 2^d corners of the cube ($d = \sqrt{\log n}$, get $2^{\sqrt{\log n}}$ factor)

Our algorithm

- Route each point to polylog(n) random points nearby (not 1 point to control the variance)

Oblivious routing

Algorithm intuition

- Should be **unbiased**: demand from a given vertex should be spread **evenly**
- Sherman's algorithm ['17]: generalizes 1-d case, routes each point to all 2^d corners of the cube ($d = \sqrt{\log n}$, get $2^{\sqrt{\log n}}$ factor)

Our algorithm

- Route each point to polylog(n) random points nearby (not 1 point to control the variance)
- Need to control number of new points (don't want polylog(n) blowup each level)

Oblivious routing

Algorithm intuition

- Should be **unbiased**: demand from a given vertex should be spread **evenly**
- Sherman's algorithm ['17]: generalizes 1-d case, routes each point to all 2^d corners of the cube ($d = \sqrt{\log n}$, get $2^{\sqrt{\log n}}$ factor)

Our algorithm

- Route each point to polylog(n) random points nearby (not 1 point to control the variance)
- Need to control number of new points (don't want polylog(n) blowup each level)
- Overlay a **randomly shifted grid**: each point sends to the center of the grid it's in; do this for polylog(n) many grids

ℓ_1-embedding

Reducing to SSSP

Reducing to SSSP

- Bourgain's embedding: reduces to computing $O(\log^2 n)$ many exact SSSP's

Reducing to SSSP

- Bourgain's embedding: reduces to computing $O(\log^2 n)$ many exact SSSP's
- Turns out $(1 + \frac{1}{\log n})$-approximate SSSP(*) is sufficient

Reducing to SSSP

- Bourgain's embedding: reduces to computing $O(\log^2 n)$ many exact SSSP's
- Turns out $(1 + \frac{1}{\log n})$-approximate SSSP(*) is sufficient
- Want to apply recursion somehow

ℓ_1-embedding

Reducing to SSSP

- Bourgain's embedding: reduces to computing $O(\log^2 n)$ many exact SSSP's
- Turns out $(1 + \frac{1}{\log n})$-approximate SSSP(*) is sufficient
- Want to apply recursion somehow

Ultrasparsification and recursion

ℓ_1-embedding

Reducing to SSSP

- Bourgain's embedding: reduces to computing $O(\log^2 n)$ many exact SSSP's
- Turns out $(1 + \frac{1}{\log n})$-approximate SSSP(*) is sufficient
- Want to apply recursion somehow

Ultrasparsification and recursion

- Compute an **ultraspanner** of the graph: $(n-1) + \frac{m}{\log^4 n}$ edges, preserves distances up to polylog(n) factor

ℓ_1-embedding

Reducing to SSSP

- Bourgain's embedding: reduces to computing $O(\log^2 n)$ many exact SSSP's
- Turns out $(1 + \frac{1}{\log n})$-approximate SSSP(*) is sufficient
- Want to apply recursion somehow

Ultrasparsification and recursion

- Compute an **ultraspanner** of the graph: $(n-1) + \frac{m}{\log^4 n}$ edges, preserves distances up to polylog(n) factor
- Suffices to ℓ_1-embed this ultraspanner (pick up extra polylog(n) in the distortion)

Reducing to SSSP

- Bourgain's embedding: reduces to computing $O(\log^2 n)$ many exact SSSP's
- Turns out $(1 + \frac{1}{\log n})$-approximate SSSP(*) is sufficient
- Want to apply recursion somehow

Ultrasparsification and recursion

- Compute an **ultraspanner** of the graph: $(n-1) + \frac{m}{\log^4 n}$ edges, preserves distances up to polylog(n) factor
- Suffices to ℓ_1-embed this ultraspanner (pick up extra polylog(n) in the distortion)
- SSSP on an ultraspanner can be reduced to approximate SSSP on a graph of size $\frac{m}{\log^4 n}$

Reducing to SSSP

- Bourgain's embedding: reduces to computing $O(\log^2 n)$ many exact SSSP's
- Turns out $(1 + \frac{1}{\log n})$-approximate SSSP(*) is sufficient
- Want to apply recursion somehow

Ultrasparsification and recursion

- Compute an **ultraspanner** of the graph: $(n-1) + \frac{m}{\log^4 n}$ edges, preserves distances up to polylog(n) factor
- Suffices to ℓ_1-embed this ultraspanner (pick up extra polylog(n) in the distortion)
- SSSP on an ultraspanner can be reduced to approximate SSSP on a graph of size $\frac{m}{\log^4 n}$
- After all reductions, recursively call approximate SSSP on $\log^4 n$ many graphs of size $\frac{m}{\log^4 n}$

Open problems

Improve polylog(n) factor in running time

Open problems

Improve polylog(n) factor in running time

- Currently very high (at least $\log^{20} n$)

Open problems

Improve polylog(n) factor in running time

- Currently very high (at least $\log^{20} n$)
- Deeper connection between transshipment and SSSP?

Open problems

Improve polylog(n) factor in running time

- Currently very high (at least $\log^{20} n$)
- Deeper connection between transshipment and SSSP?
- **Exact** SSSP? Current best is $\tilde{O}(m)$ work, $n^{1/2+o(1)}$ time [Cao, Fineman, Russell STOC'20]