Faster Deterministic Worst-Case Dynamic All-Pairs Shortest Paths

Shiri Chechik               Tianyi Zhang
Dynamic All-Pairs Shortest Paths

• Given a weighted digraph $G = (V, E, \omega)$

• A sequence of vertex updates, maintain pairwise exact distances

More specifically, want a data structure:

• $\text{Ins}(v, \text{adj}(v)) / \text{Del}(v)$
  Insert/delete vertex $v$ in $G$ with adjacency list $\text{adj}(v)$; want $n^2$ runtime

• $\text{Query}(u, v)$
  Return the shortest distance from $u$ to $v$ in $G$; want $O(1)$ runtime
Dynamic All-Pairs Shortest Paths

- Given a weighted digraph $G = (V, E, \omega)$
- A sequence of vertex updates, maintain pairwise exact distances

Why vertex updates, not edge updates?
<table>
<thead>
<tr>
<th>reference</th>
<th>vertex update time</th>
<th>deterministic / randomized</th>
<th>worst-case / amortized</th>
</tr>
</thead>
<tbody>
<tr>
<td>King, 1999</td>
<td>$\tilde{O}(n^{2.5}\sqrt{W})$</td>
<td>deterministic</td>
<td>amortized</td>
</tr>
<tr>
<td>Demetrescu, Italiano, 2004</td>
<td>$\tilde{O}(n^2)$</td>
<td>deterministic</td>
<td>amortized</td>
</tr>
<tr>
<td>Thorup, 2005</td>
<td>$\tilde{O}(n^{3-1/4})$</td>
<td>deterministic</td>
<td>worst-case</td>
</tr>
<tr>
<td>Abraham, Chechik, Krinninger, 2017</td>
<td>$\tilde{O}(n^{3-1/3})$</td>
<td>randomized</td>
<td>worst-case</td>
</tr>
<tr>
<td>Probst, Wulff-Nilsen, 2020</td>
<td>$\tilde{O}(n^{3-2/7})$</td>
<td>deterministic</td>
<td>worst-case</td>
</tr>
<tr>
<td><strong>New</strong></td>
<td>$\tilde{O}(n^{3-20/61})$</td>
<td>deterministic</td>
<td>worst-case</td>
</tr>
</tbody>
</table>

$n$ is the number of vertices in the graph, $W$ refers to the maximum edge weight.
Previous approaches
Reduction to batch deletion

Batch deletion data structure:

- **Prep**($G$)
  Preprocess the graph $G$ and be ready for **one batch deletion** and queries

- **Batch**($B$)
  Remove a subset $B \subseteq V$ of vertices from graph $G$

- **Query**($u$, $v$)
  Return the shortest distance from $u$ to $v$ in $G \setminus B$ in $O(1)$ time
Reduction to batch deletion

Theorem [Thorup, 2005]

• Given a batch deletion algorithm, dynamic APSP can be solved with worst-case update time $T_{\text{prep}}/|B| + T_{\text{batch}} + |B|n^2$

• **Batch**(B)
  Remove a subset $B \subseteq V$ of vertices from graph $G$

• **Query**(u, v)
  Return the shortest distance from $u$ to $v$ in $G\setminus B$ in $O(1)$ time
Batch Deletion

Main difficulty:
Precompute shortest paths in $G$
A single deletion can destroy a lot
Batch Deletion

Main difficulty:
Precompute shortest paths in G
A single deletion can destroy a lot

Two basic ideas [Thorup, 2005]

• Shortest paths with small #hops
  Long-hop paths can be handled using hitting sets

• Prepare low-congestion shortest paths
Batch Deletion

Main difficulty:
Precompute shortest paths in G
A single deletion can destroy a lot

Two basic ideas [Thorup, 2005]

- Shortest paths with small #hops
  Long-hop paths can be handled using hitting sets

- Prepare low-congestion shortest paths

Identify a highly congested vertex
**Batch Deletion**

**Main difficulty:**
Precompute shortest paths in G
A single deletion can **destroy a lot**

Two basic ideas [Thorup, 2005]

- Shortest paths with small #hops
  Long-hop paths can be handled using hitting sets

- **Prepare low-congestion shortest paths**

Remove this vertex in advance
**Main difficulty:**
Precompute shortest paths in G
A single deletion can **destroy a lot**

Two basic ideas [Thorup, 2005]

- Shortest paths with small #hops
  Long-hop paths can be handled using hitting sets

- **Prepare low-congestion shortest paths**
Hop-Restricted Shortest Paths

- An $h$-hop shortest path $\pi_{s,t}$ is the **shortest path with at most $h$ edges**

- Single-source $h$-hop paths $\{\pi_{s,t}\}_{t \in V}$ can be computed using the **Bellman-Ford** algorithm in time $n^2h$

- Ordinary shortest path might contain $\gg h$ edges
The congestion technique

**Low-congestion shortest paths** [Thorup’05]

1. Pick a vertex $v$ that maximizes $cg(v)$
2. Compute $h$-hop shortest paths at $v$ using Bellman-Ford
3. Add $h$-hop paths to $\Pi$, update $cg(.)$
4. Remove $v$ from graph, go to Step 1

$\Pi = \{ \pi_{s,t} \mid s, t \in V \}$ a set of short paths

$cg(v) = \# \text{paths in } \Pi \text{ containing } v$
The congestion technique

**Low-congestion shortest paths** [Thorup’05]

1. Pick a vertex $v$ that maximizes $cg(v)$

2. Compute $h$-hop shortest paths at $v$ using Bellman-Ford

3. Add $h$-hop paths to $\Pi$, update $cg(.)$

4. Remove $v$ from graph, go to Step 1

$\Pi = \{ \pi_{s,t} \mid s, t \in V \}$ a set of short paths

$cg(v) = \#\text{paths in } \Pi \text{ containing } v$
The congestion technique

**Low-congestion shortest paths** [Thorup’05]

1. Pick a vertex $v$ that maximizes $cg(v)$
2. Compute $h$-hop shortest paths at $v$ using Bellman-Ford
3. Add $h$-hop paths to $\Pi$, update $cg(.)$
4. Remove $v$ from graph, go to Step 1

$$\Pi = \{ \pi_{s,t} \mid s, t \in V \}$$ a set of short paths

$$cg(v) = \# \text{paths in } \Pi \text{ containing } v$$
**The congestion technique**

**Low-congestion shortest paths** [Thorup’05]

1. Pick a vertex $v$ that maximizes $c_g(v)$
2. Compute $h$-hop shortest paths at $v$ using Bellman-Ford
3. Add $h$-hop paths to $\Pi$, update $c_g(.)$
4. Remove $v$ from graph, go to Step 1

$$\Pi = \{ \pi_{s,t} \mid s, t \in V \} \text{ a set of short paths}$$

$$c_g(v) = \# \text{paths in } \Pi \text{ containing } v$$
The congestion technique

**Low-congestion shortest paths** [Thorup’05]

1. Pick a vertex $v$ that maximizes $c_g(v)$
2. Compute $h$-hop shortest paths at $v$ using Bellman-Ford
3. Add $h$-hop paths to $\Pi$, update $c_g(.)$
4. Remove $v$ from graph, go to Step 1

$\Pi = \{ \pi_{s,t} \mid s, t \in V \}$ a set of short paths

$c_g(v) = \# \text{paths in } \Pi \text{ containing } v$
The congestion technique

**Low-congestion shortest paths** [Thorup’05]

1. Pick a vertex $v$ that maximizes $cg(v)$
2. Compute $h$-hop shortest paths at $v$ using Bellman-Ford
3. Add $h$-hop paths to $\Pi$, update $cg(.)$
4. Remove $v$ from graph, go to Step 1

$\Pi = \{ \pi_{s,t} \mid s, t \in V \}$ a set of short paths

$cg(v) = \text{#paths in } \Pi \text{ containing } v$
Recovery from batch deletion

Recovery by Dijkstra’s algorithm [ACK’17]
Recovery from batch deletion

Recovery by Dijkstra’s algorithm [ACK’17]
Recovery from batch deletion

Recovery by Dijkstra’s algorithm [ACK’17]

**Recovery algorithm:**
1. View red paths as shortcuts
2. Run Dijkstra on red / black edges

Runtime = $n \cdot \#\text{destroyed}$ h-hop paths

$\#\text{destroyed}$ is small by the congestion technique
Recovery from batch deletion

Recovery by Dijkstra’s algorithm [ACK’17]

Recovery by path concat [P-WN’20]
Recovery from batch deletion

Recovery by Dijkstra’s algorithm [ACK’17]

Recovery by path concat [P-WN’20]
Recovery from batch deletion

Recovery by Dijkstra’s algorithm [ACK’17]

Recovery by path concat [P-WN’20]

h-hop paths

incident edges

incident edges

incident edges

A deterministic hitting set for h/2-hop paths
Recovery from batch deletion

Recovery by Dijkstra’s algorithm [ACK’17]

Recovery by path concat [P-WN’20]

h-hop paths

incident edges

incident edges

incident edges

A deterministic hitting set for h/2-hop paths

h/2-hop paths
Recovery from batch deletion

Recovery by Dijkstra’s algorithm [ACK’17]

Recovery by path concat [P-WN’20]

A deterministic hitting set for \( h/2 \)-hop paths

Runtime of concatenation:

\[
n/h \cdot \#\text{destroyed} \text{ } h\text{-hop paths}
\]
Our improvement
Outline

Dynamic APSP

standard reduction

Batch Deletion: the \textit{congestion} technique

[Thorup’05, ACK’17, PG-WN’20]

Batch Deletion: faster \textit{preprocessing}

new
Decremental hop-restricted shortest paths

Low-congestion shortest paths [Thorup’05]

1. Pick a vertex $v$ that maximizes $cg(v)$
2. Compute $h$-hop shortest paths at $v$ using Bellman-Ford
3. Add $h$-hop paths to $\Pi$, update $cg(.)$
4. Remove $v$ from graph, go to Step 1
Low-congestion shortest paths [Thorup’05]

1. Pick a vertex $v$ that maximizes $cg(v)$

2. Compute $h$-hop shortest paths at $v$ using Bellman-Ford

3. Add $h$-hop paths to $\Pi$, update $cg(.)$

4. Remove $v$ from graph, go to Step 1
Decremental hop-restricted shortest paths

Low-congestion shortest paths [Thorup’05]
1. Pick a vertex \( v \) that maximizes \( c_g(v) \)
2. Compute \( h \)-hop shortest paths at \( v \) using Bellman-Ford
3. Add \( h \)-hop paths to \( \Pi \), update \( c_g(.) \)
4. Remove \( v \) from graph, go to Step 1

Decremental \( h \)-hop shortest paths:
1. Adversary picks a vertex \( v \)
2. Compute \( h \)-hop SSSP at \( v \)
3. Adversary deletes an arbitrary vertex
4. Go to Step 1
Decremental hop-restricted shortest paths

Trivial algorithm:

• Apply Bellman-Ford for h-hop SSSP
• Total time = $n^2h \cdot \#\text{deletions}$

Faster runtime?

• Try to maintain all h-hop paths under vertex deletions

Decremental h-hop shortest paths:

1. Adversary picks a vertex $v$
2. Compute h-hop SSSP at $v$
3. Adversary deletes an arbitrary vertex
4. Go to Step 1
Locally h-hop shortest paths

- Adapt the idea of **locally shortest paths** in [Demetrescu and Italiano, 2004]
- A path $\langle u_0, u_1, \cdots, u_k \rangle$ is **locally h-hop shortest**, if both of the sub-paths $\langle u_0, u_1, \cdots, u_{k-1} \rangle$ and $\langle u_1, \cdots, u_k \rangle$ are (h-1)-hop shortest paths
Locally h-hop shortest paths

**Shortest** locally h-hop shortest paths = h-hop shortest paths

#(locally h-hop) can be bounded

- Each vertex \( v \) is on **at most** \( h \) different **locally h-hop paths** from \( s \) to \( t \)

- At most \( n^3 \log n \) (all-pairs locally h-hop) in total
Locally h-hop shortest paths

**Shortest** locally h-hop shortest paths = h-hop shortest paths

#(locally h-hop) can be bounded

- Each vertex \( v \) is on **at most** \( h \) different **locally h-hop paths** from \( s \) to \( t \)

- At most \( n^3 \log n \) (all-pairs locally h-hop) in total
Locally h-hop shortest paths

**Shortest** locally h-hop shortest paths = h-hop shortest paths

#(locally h-hop) can be bounded

- Each vertex \(v\) is on **at most** \(h\) different **locally h-hop paths** from \(s\) to \(t\)

- At most \(n^3 \log n\) (all-pairs locally h-hop) in total
Decremental locally h-hop shortest paths

Maintain all locally h-hop shortest paths under \textit{vertex deletions}
Decremental locally h-hop shortest paths

Maintain all locally h-hop shortest paths under \textit{vertex deletions}

1. A vertex deletion \textbf{hits} at most $n^2 h$
   old locally h-hop shortest paths

Runtime = #(destroyed) $\leq n^2 h$
Decremental locally h-hop shortest paths

Maintain all locally h-hop shortest paths under vertex deletions

1. A vertex deletion hits at most $n^2h$ old locally h-hop shortest paths
   
   Runtime = #(destroyed) $\leq n^2h$

2. A vertex deletion may generate new locally h-hop shortest paths
   
   Runtime = #(new locally h-hop)

Total #(locally h-hop paths)
Decremental locally h-hop shortest paths

Maintain all locally h-hop shortest paths under vertex deletions

1. A vertex deletion hits at most $n^2h$ old locally h-hop shortest paths
   Runtime = #(destroyed) $\leq n^2h$

2. A vertex deletion may generate new locally h-hop shortest paths
   Runtime = #(new locally h-hop)

3. Output-sensitive $\rightarrow$ total time $= n^3h$
Accelerating Bellman-Ford

Original goal:

1. Adversary picks a vertex v
2. Compute h-hop SSSP at v
3. Adversary deletes an arbitrary vertex
4. Go to Step 1

Trivial algorithm:

- Apply Bellman-Ford for h-hop SSSP
- Total time $= n^2h \cdot \#\text{deletions}$

Faster runtime?

- Decremental h-hop paths has total runtime $= n^3h$, no improvement
Accelerating Bellman-Ford

Original goal:

1. Adversary picks a vertex \( v \)

2. Compute \textcolor{red}{h-hop SSSP at} \( v \)

3. Adversary \textcolor{red}{deletes an arbitrary} vertex

4. Go to Step 1

Faster runtime?

• Decremental h-hop paths has total runtime = \( n^3h \), no improvement
Accelerating Bellman-Ford

Original goal:

1. Adversary picks a vertex $v$
2. Compute **h-hop SSSP at $v$**
3. Adversary **deletes an arbitrary** vertex
4. Go to Step 1

Faster runtime?

- Decremental h-hop paths has total runtime $= n^3h$, **no improvement**

Solution:

- Apply decremental g-hop paths
- Bellman-Ford runs in time $n^2h/g$
- Total time $= n^3(g + h/g) < n^3h$
Accelerating Bellman-Ford

Faster runtime?

• Decremental h-hop paths has total runtime = $n^3h$, no improvement

Solution:

• Apply decremental g-hop paths

• Bellman-Ford runs in time $n^2h/g$

• Total time = $n^3(g + h/g) < n^3h$

Original goal:

1. Adversary picks a vertex $v$
2. Compute **h-hop SSSP at $v$**
3. Adversary **deletes an arbitrary** vertex
4. Go to Step 1
Accelerating Bellman-Ford

Bellman-Ford runs in time $n^2h/g$

1. Standard Bellman-Ford = \textbf{h rounds} of dynamic programming

2. \textbf{Compress every g-rounds} into a single round using g-hop paths
Accelerating Bellman-Ford

Bellman-Ford runs in time $n^2 h/g$

1. Standard Bellman-Ford = h rounds of dynamic programming

2. **Compress every g-rounds** into a single round using g-hop paths
Accelerating Bellman-Ford

Bellman-Ford runs in time $n^2h/g$

1. Standard Bellman-Ford = \textbf{h rounds} of dynamic programming

2. **Compress every g-rounds** into a single round using g-hop paths
Further Questions

• Faster randomized worst-case update time $n^{3-1/3-\epsilon}$ ?

• Faster deterministic worst-case update time $n^{3-1/3}$ ?

Thank you !