Fully dynamic all-pairs shortest paths with worst-case update-time revisited

Ittai Abraham[1] Shiri Chechik[2] Sebastian Krinninger[3]

[1]Hebrew University of Jerusalem

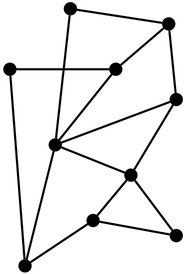
[2]Tel-Aviv University

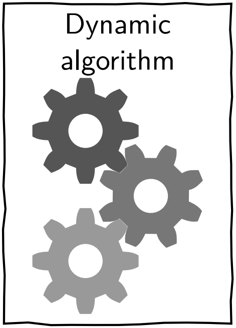
[3]Max Planck Institute for Informatics
Saarland Informatics Campus
since Jan: University of Vienna

SODA 2017

Dynamic model

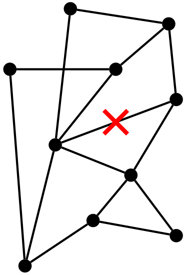
G undergoing updates:

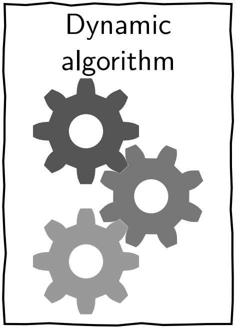




Dynamic model

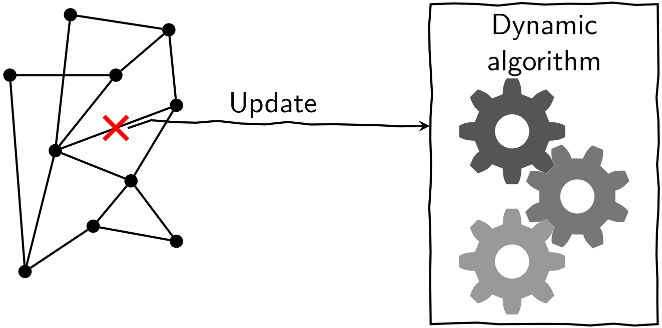
G undergoing updates:





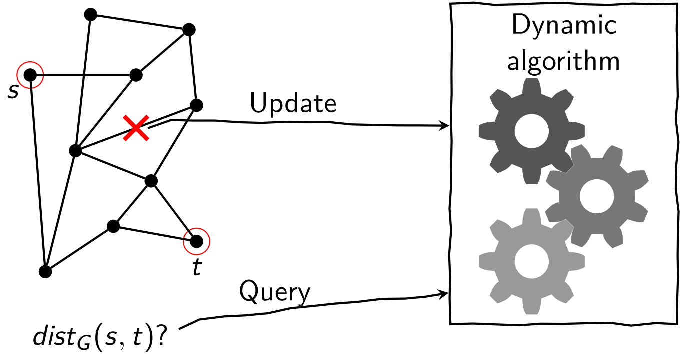
Dynamic model

G undergoing updates:



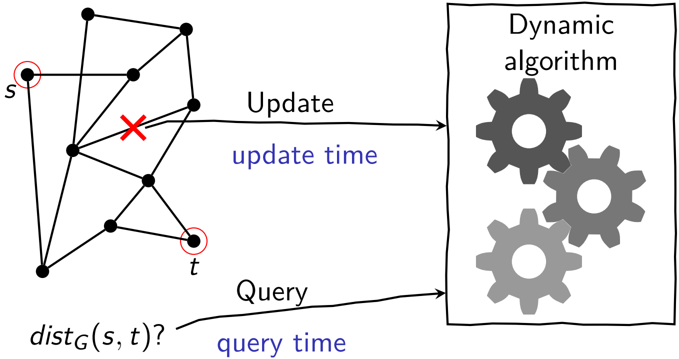
Dynamic model

G undergoing updates:



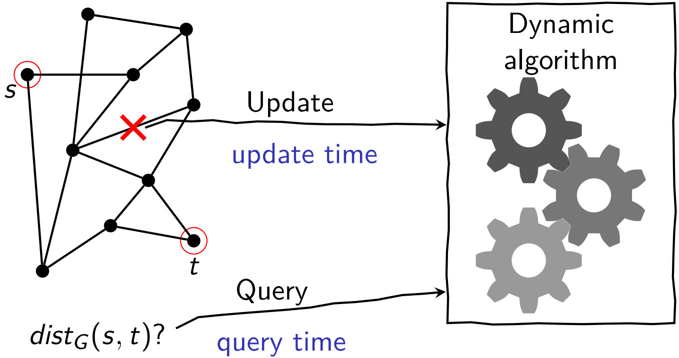
Dynamic model

G undergoing updates:



Dynamic model

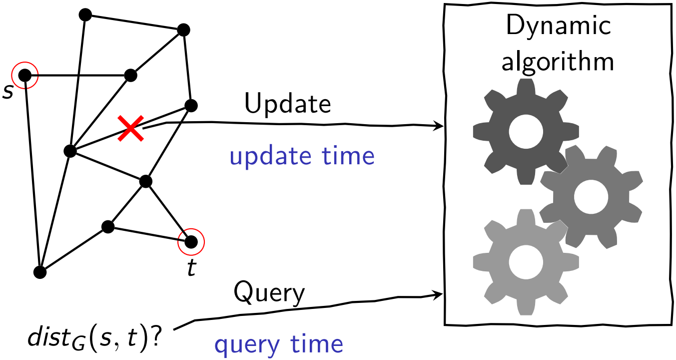
G undergoing updates:



Here: Small query time $\mathcal{O}(1)$ or $\mathcal{O}(\log n)$

Dynamic model

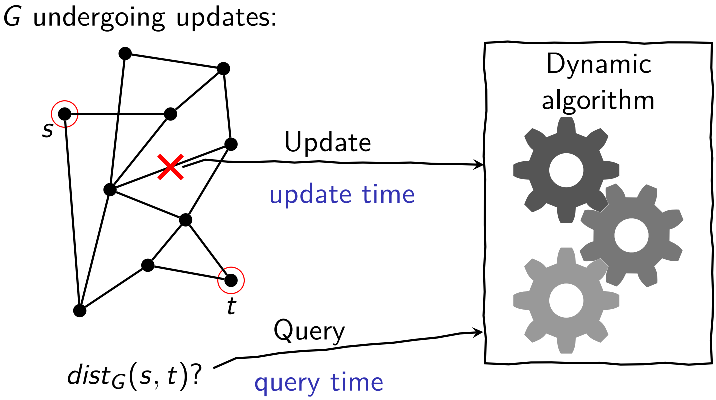
G undergoing updates:



Here: Small query time $\mathcal{O}(1)$ or $\mathcal{O}(\log n)$

Goal: Minimize update time $T(n)$

Dynamic model



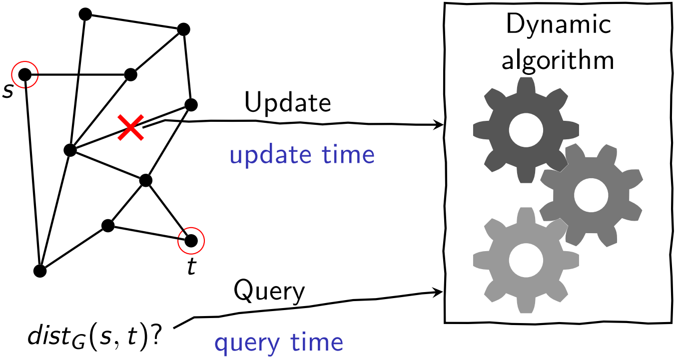
Here: Small query time $\mathcal{O}(1)$ or $\mathcal{O}(\log n)$

Goal: Minimize update time $T(n)$

- Worst-case: After each update, spend time $\leq T(n)$

Dynamic model

G undergoing updates:



Here: Small query time $\mathcal{O}(1)$ or $\mathcal{O}(\log n)$

Goal: Minimize update time $T(n)$

- Worst-case: After each update, spend time $\leq T(n)$
- Amortized: For a sequence of k updates, spend time $\leq kT(n)$

Question:
Can worst-case bounds match amortized bounds?

Prior work on dynamic APSP

approx.	update time	type of graphs	reference
exact	$\tilde{\mathcal{O}}(mn)$	weighted directed	[Dijkstra]
exact	$\tilde{\mathcal{O}}(n^{2.5}\sqrt{W})$	weighted directed	[King '99]
$1+\epsilon$	$\tilde{\mathcal{O}}(n^2 \log W)$	weighted directed	[King '99]
$2+\epsilon$	$\tilde{\mathcal{O}}(n^2)$	weighted directed	[King '99]
exact	$\tilde{\mathcal{O}}(n^{2.5}\sqrt{W})$	weighted directed	[Demetrescu/Italiano '01]
exact	$\tilde{\mathcal{O}}(n^2)$	weighted directed	[Demetrescu/Italiano '03]
exact	$\tilde{\mathcal{O}}(n^{2.75})$ (*)	weighted directed	[Thorup '05]
$2+\epsilon$	$\tilde{\mathcal{O}}(m \log W)$	weighted undirected	[Bernstein '09]
$2^{\mathcal{O}(k)}$	$\tilde{\mathcal{O}}(\sqrt{m}n^{1/k})$	unweighted undirected	[Abr./Chechik/Talwar '14]

(*) worst case

$\tilde{\mathcal{O}}$: ignores $\log n$-factors
n: number of nodes
m: number of edges
W: largest edge weight

Our result

Theorem (for this talk)

There is an algorithm for maintaining a distance matrix under insertions and deletions of nodes in unweighted undirected graphs with a worst-case update time of $\tilde{\mathcal{O}}(n^{2.75})$.

Our result

Theorem (for this talk)

There is an algorithm for maintaining a distance matrix under insertions and deletions of nodes in unweighted undirected graphs with a worst-case update time of $\tilde{\mathcal{O}}(n^{2.75})$.

Toy example! ($\mathcal{O}(n^{\omega})$ in unweighted graphs)

Our result

Theorem (for this talk)

There is an algorithm for maintaining a distance matrix under insertions and deletions of nodes in unweighted undirected graphs with a worst-case update time of $\tilde{\mathcal{O}}(n^{2.75})$.

Toy example! ($\mathcal{O}(n^{\omega})$ in unweighted graphs)

More sophisticated use of our technique:

- $\tilde{\mathcal{O}}(n^{2.67})$ in weighted directed graphs (randomized)
- Improves $\tilde{\mathcal{O}}(n^{2.75})$ of [Thorup '05]
- (Arguably) simpler than [Thorup '05]
  (which is a deamortization of [Demetrescu/Italiano '03])

Batch deletion problem

Preprocessing phase:

- Preprocess a graph G in time $P(n)$

Deletion phase:

- A (single) set D of $\leq \Delta$ nodes is deleted from the graph
- Compute APSP in $G \setminus D$ in time $D(n)$

Batch deletion problem

Preprocessing phase:

- Preprocess a graph G in time $P(n)$

Deletion phase:

- A (single) set D of $\leq \Delta$ nodes is deleted from the graph
- Compute APSP in $G \setminus D$ in time $D(n)$

Lemma (Thorup '05)

If there is a batch deletion APSP algorithm supporting up to Δ deletions with proprocessing time $P(n)$ and batch deletion time $D(n)$, then there is a fully dynamic APSP algorithm with worst-case update time $\tilde{O}(P(n)/\Delta + D(n) + \Delta n^2)$.

Batch deletion problem

Preprocessing phase:

- Preprocess a graph G in time $P(n)$

Deletion phase:

- A (single) set D of $\leq \Delta$ nodes is deleted from the graph
- Compute APSP in $G \setminus D$ in time $D(n)$

Lemma (Thorup '05)

If there is a batch deletion APSP algorithm supporting up to Δ deletions with proprocessing time $P(n)$ and batch deletion time $D(n)$, then there is a fully dynamic APSP algorithm with worst-case update time $\tilde{O}(P(n)/\Delta + D(n) + \Delta n^2)$.

- Restart batch deletion algorithm periodically, spread out preprocessing time over Δ updates

Batch deletion problem

Preprocessing phase:

- Preprocess a graph G in time $P(n)$

Deletion phase:

- A (single) set D of $\leq \Delta$ nodes is deleted from the graph
- Compute APSP in $G \setminus D$ in time $D(n)$

Lemma (Thorup '05)

If there is a batch deletion APSP algorithm supporting up to Δ deletions with proprocessing time $P(n)$ and batch deletion time $D(n)$, then there is a fully dynamic APSP algorithm with worst-case update time $\tilde{\mathcal{O}}(P(n)/\Delta + D(n) + \Delta n^2)$.

- Restart batch deletion algorithm periodically, spread out preprocessing time over Δ updates
- Insertions are easy: $\mathcal{O}(\Delta n^2)$ (Floyd-Warshall)

Batch deletion problem

Preprocessing phase:

- Preprocess a graph G in time $P(n)$

Deletion phase:

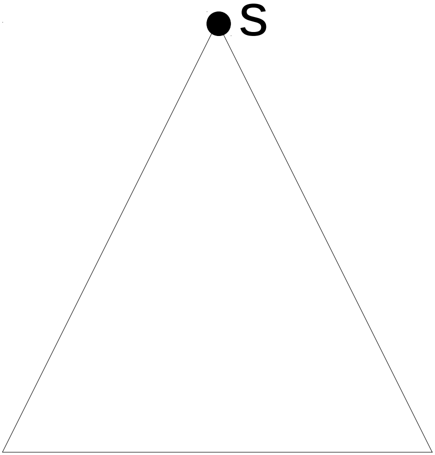
- A (single) set D of $\leq \Delta$ nodes is deleted from the graph
- Compute APSP in $G \setminus D$ in time $D(n)$
- Suffices to compute shortest paths consisting of $\leq h$ nodes

Lemma (Thorup '05)

If there is a batch deletion APSP algorithm supporting up to Δ deletions with proprocessing time $P(n)$ and batch deletion time $D(n)$, then there is a fully dynamic APSP algorithm with worst-case update time $\tilde{\mathcal{O}}(P(n)/\Delta + D(n) + \Delta n^2 + hn^2 + n^3/h)$.

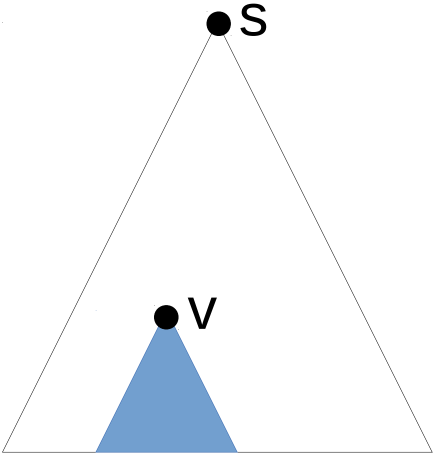
- Restart batch deletion algorithm periodically, spread out preprocessing time over Δ updates
- Insertions are easy: $\mathcal{O}(\Delta n^2)$ (Floyd-Warshall)
- Hitting set of size $\tilde{\mathcal{O}}(n/h)$ for all shortest paths with $\leq h$ nodes

Repairing a shortest path tree



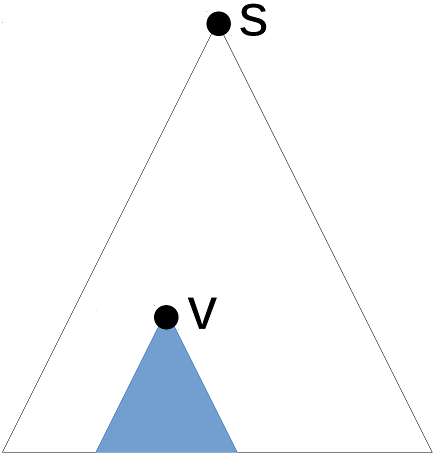
- Given: shortest path tree from s

Repairing a shortest path tree



- Given: shortest path tree from s
- Node v is deleted
- Shortest path destroyed only for nodes in subtree of v

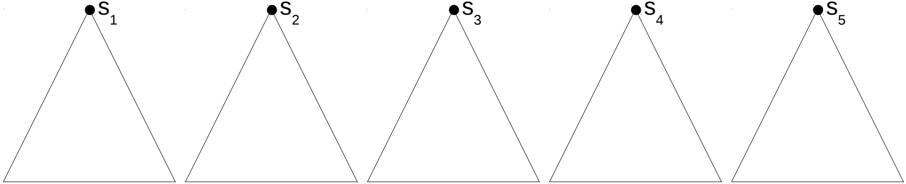
Repairing a shortest path tree



- Given: shortest path tree from s
- Node v is deleted
- Shortest path destroyed only for nodes in subtree of v
- Run Dijkstra's algorithm to reattach these nodes to the tree
- Charge time $\mathcal{O}(deg(u)) \leq \mathcal{O}(n)$ to every node u in subtree of v

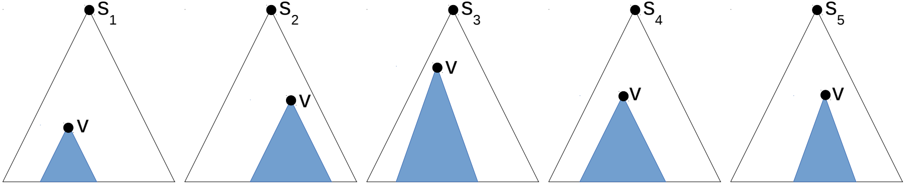
Multiple shortest path trees

Goal: shortest paths from a set of source nodes S



Multiple shortest path trees

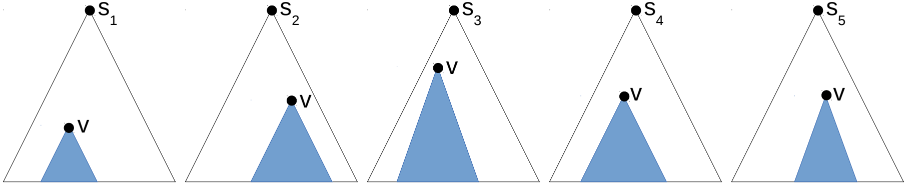
Goal: shortest paths from a set of source nodes S



Deletion of v

Multiple shortest path trees

Goal: shortest paths from a set of source nodes S

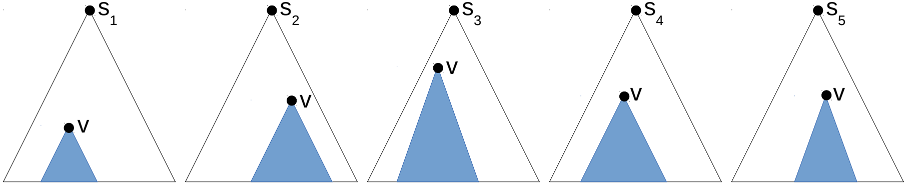


Deletion of v

Total work: (number of nodes in subtrees of v) $\times n$

Multiple shortest path trees

Goal: shortest paths from a set of source nodes S



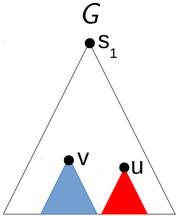
Deletion of v

Total work: (number of nodes in subtrees of v) $\times n$

Goal: limit sizes of subtrees of each node

Preprocessing

Construct shortest path tree up to depth h for all sources one by one:



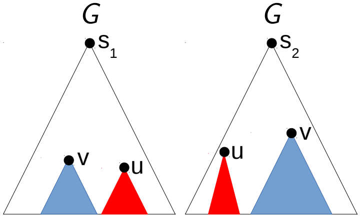
Count size of subtrees for every node

Rule: If number of nodes in subtrees of v exceeds λ:

- v is added to set of **heavy** nodes H
- v is deleted from graph, i.e., not considered in future trees

Preprocessing

Construct shortest path tree up to depth h for all sources one by one:



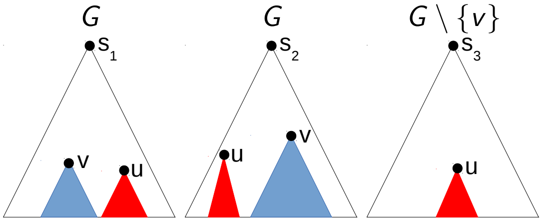
Count size of subtrees for every node

Rule: If number of nodes in subtrees of v exceeds λ:

- v is added to set of **heavy** nodes H
- v is deleted from graph, i.e., not considered in future trees

Preprocessing

Construct shortest path tree up to depth h for all sources one by one:



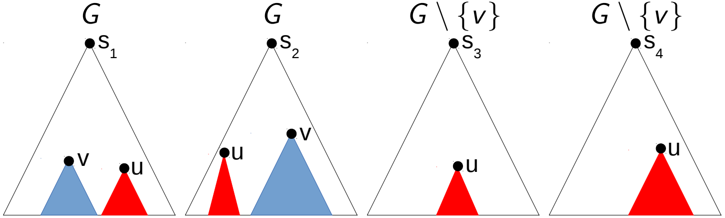
Count size of subtrees for every node

Rule: If number of nodes in subtrees of v exceeds λ:

- v is added to set of **heavy** nodes H
- v is deleted from graph, i.e., not considered in future trees

Preprocessing

Construct shortest path tree up to depth h for all sources one by one:



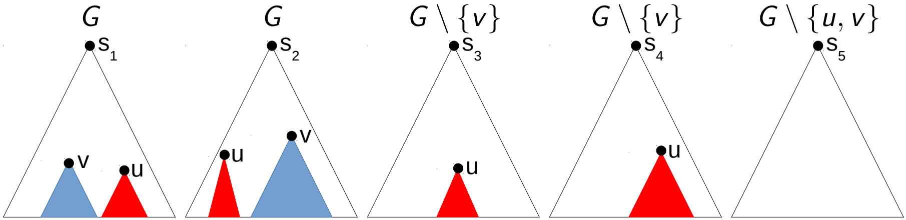
Count size of subtrees for every node

Rule: If number of nodes in subtrees of v exceeds λ:

- v is added to set of **heavy** nodes H
- v is deleted from graph, i.e., not considered in future trees

Preprocessing

Construct shortest path tree up to depth h for all sources one by one:



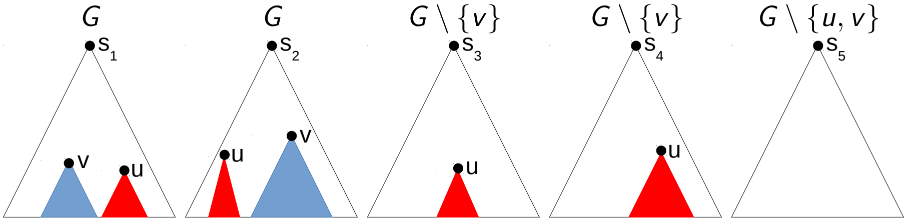
Count size of subtrees for every node

Rule: If number of nodes in subtrees of v exceeds λ:

- v is added to set of **heavy** nodes H
- v is deleted from graph, i.e., not considered in future trees

Preprocessing

Construct shortest path tree up to depth h for all sources one by one:



Count size of subtrees for every node

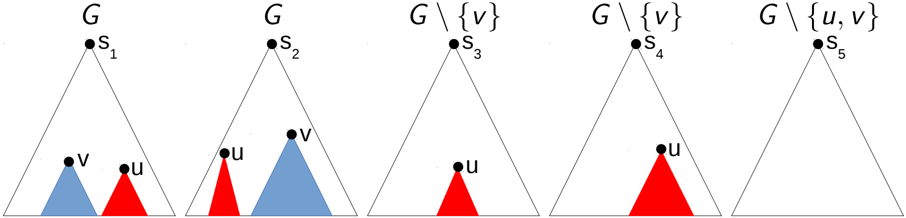
Rule: If number of nodes in subtrees of v exceeds λ:

- v is added to set of **heavy** nodes H
- v is deleted from graph, i.e., not considered in future trees

Observations:

- All shortest paths not using heavy nodes included in trees
- Number of heavy nodes: $|H| \leq \mathcal{O}(\frac{|S|nh}{\lambda}) \leq \mathcal{O}(\frac{n^2h}{\lambda})$
- Preprocessing time: $\mathcal{O}(|S|n^2) \leq \mathcal{O}(n^3)$

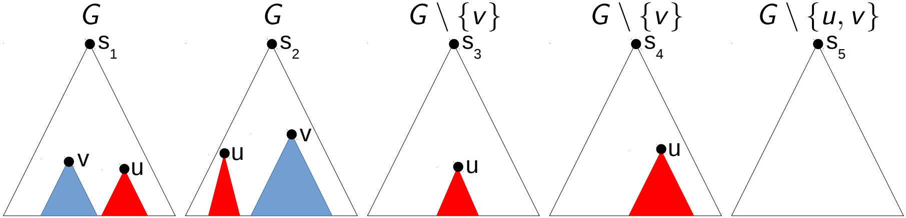
Computing distances after batch of Δ deletions



1. For all deleted nodes: Reattach children to tree using Dijkstra
   Running time: $\mathcal{O}(\Delta\lambda n)$ per deletion
   - Subtree size at most λ per node
   - Number of deleted nodes at most Δ

   Correct for all shortest paths not containing heavy nodes

Computing distances after batch of Δ deletions



1. For all deleted nodes: Reattach children to tree using Dijkstra
   Running time: $\mathcal{O}(\Delta\lambda n)$ per deletion
   - Subtree size at most λ per node
   - Number of deleted nodes at most Δ

   Correct for all shortest paths not containing heavy nodes
2. Special treatment of heavy nodes: shortest paths via heavy nodes

   $$\text{Compute } \min_{v \in H}(dist(s, v) + dist(v, t)) \text{ for all } s \text{ and } t$$

   Time per deletion: $\mathcal{O}(|H|n^2) = \mathcal{O}(\frac{n^4 h}{\lambda})$

Running time wrapped up

- $\mathcal{O}(\Delta\lambda n)$ Repair shortest path trees
- $\mathcal{O}(\frac{n^4 h}{\lambda})$ Shortest paths via heavy nodes

Running time wrapped up

- $\mathcal{O}(\Delta\lambda n)$ Repair shortest path trees
- $\mathcal{O}(\frac{n^4 h}{\lambda})$ Shortest paths via heavy nodes
- $\mathcal{O}(\frac{n^3}{\Delta})$ Preprocessing of $\mathcal{O}(n^3)$ spread over Δ updates

Running time wrapped up

- $\mathcal{O}(\Delta\lambda n)$ Repair shortest path trees
- $\mathcal{O}(\frac{n^4 h}{\lambda})$ Shortest paths via heavy nodes
- $\mathcal{O}(\frac{n^3}{\Delta})$ Preprocessing of $\mathcal{O}(n^3)$ spread over Δ updates
- $\mathcal{O}(\Delta n^2)$ Shortest paths via inserted nodes
- $\tilde{\mathcal{O}}(n^2 h + \frac{n^3}{h})$ Shortest paths of length more than h

Running time wrapped up

- $\mathcal{O}(\Delta\lambda n)$ Repair shortest path trees
- $\mathcal{O}(\frac{n^4 h}{\lambda})$ Shortest paths via heavy nodes
- $\mathcal{O}(\frac{n^3}{\Delta})$ Preprocessing of $\mathcal{O}(n^3)$ spread over Δ updates
- $\mathcal{O}(\Delta n^2)$ Shortest paths via inserted nodes
- $\tilde{\mathcal{O}}(n^2 h + \frac{n^3}{h})$ Shortest paths of length more than h

$$\Delta = n^{0.25},\ \lambda = n^{1.5},\ h = n^{0.25}$$
$$\Rightarrow \tilde{\mathcal{O}}(n^{2.75})$$

Improvements

Directed graphs:

Two types of shortest path trees: incoming and outgoing

Improvements

Directed graphs:
Two types of shortest path trees: incoming and outgoing

Weighted graphs:
Requires Bellman-Ford in preprocessing: $\mathcal{O}(n^2 h)$ per node

Improvements

Directed graphs:
Two types of shortest path trees: incoming and outgoing

Weighted graphs:
Requires Bellman-Ford in preprocessing: $\mathcal{O}(n^2 h)$ per node

Increased efficiency:
Multiple instances of algorithm to cover all hop ranges (+randomization)
Load balancing trick

Open problems

Is $\tilde{\mathcal{O}}(n^{2.5})$ the right answer?

Open problems

Is $\tilde{\mathcal{O}}(n^{2.5})$ the right answer?

- Pro: Natural barrier for algorithmic approaches
- Con: No scheme for a conditional lower bound applies

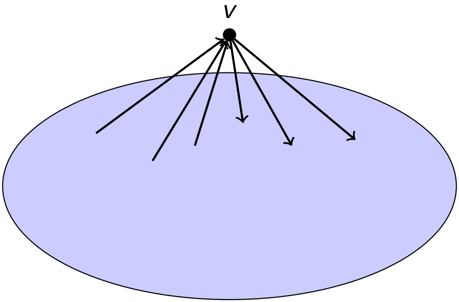
Thank you!

Thank you!

Questions?

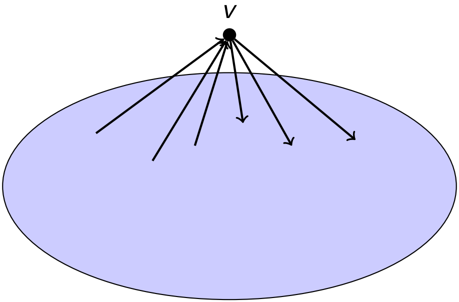
Insertions are easy

Inserting a node v:



Insertions are easy

Inserting a node v:



Floyd-Warshall algorithm

- For every node s: $dist'(s,v) = \min\limits_{(u,v)}(dist(s,u) + w(u,v))$
- For every node t: $dist'(v,t) = \min\limits_{(v,u)}(w(v,u) + dist(u,t))$
- For every pair s,t: $dist'(s,t) = \min(dist(s,t), dist'(s,v) + dist'(v,t))$
- Time per insertion: $\mathcal{O}(n^2)$

Handling deletions

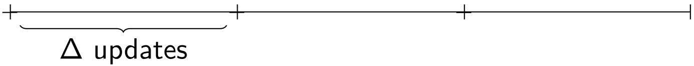
- Principle approach: deletions-only algorithm

Handling deletions

- Principle approach: deletions-only algorithm
- Preprocessing stage: Prepare data structure for handling a batch of $\leq \Delta$ deletions
- After every update:
  - Group updates since preprocessing into insertions and deletions
  - Perform $\leq \Delta$ deletions in data structure from preprocessing
  - Process $\leq \Delta$ deletions with Floyd Warshall $\mathcal{O}(\Delta n^2)$

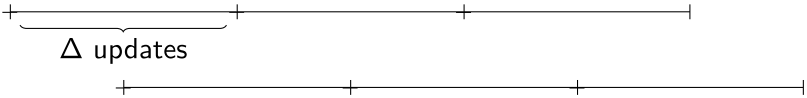
Handling deletions

- Principle approach: deletions-only algorithm
- Preprocessing stage: Prepare data structure for handling a batch of $\leq \Delta$ deletions
- After every update:
  - Group updates since preprocessing into insertions and deletions
  - Perform $\leq \Delta$ deletions in data structure from preprocessing
  - Process $\leq \Delta$ deletions with Floyd Warshall $\mathcal{O}(\Delta n^2)$
- Standard trick: preprocessing can be spread out over Δ updates



Handling deletions

- Principle approach: deletions-only algorithm
- Preprocessing stage: Prepare data structure for handling a batch of $\leq \Delta$ deletions
- After every update:
  - Group updates since preprocessing into insertions and deletions
  - Perform $\leq \Delta$ deletions in data structure from preprocessing
  - Process $\leq \Delta$ deletions with Floyd Warshall $\mathcal{O}(\Delta n^2)$
- Standard trick: preprocessing can be spread out over Δ updates



Restricted hop depth

Definition

shortest h-hop path: shortest among all paths with $\leq h$ edges

Restricted hop depth

Definition

shortest h-hop path: shortest among all paths with $\leq h$ edges

Suppose shortest h-hop path known for all pairs of nodes
$\Rightarrow$ Can compute all-pairs shortest paths in time $\mathcal{O}(n^3 \log n / h)$:

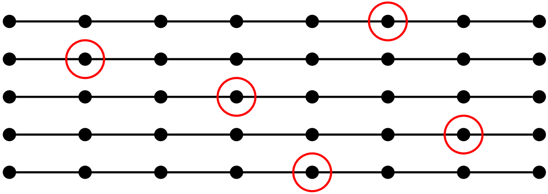
Restricted hop depth

Definition

shortest h-hop path: shortest among all paths with $\leq h$ edges

Suppose shortest h-hop path known for all pairs of nodes
$\Rightarrow$ Can compute all-pairs shortest paths in time $\mathcal{O}(n^3 \log n/h)$:

- Hitting set of size $\mathcal{O}(n/h)$ (probabilistic argument)



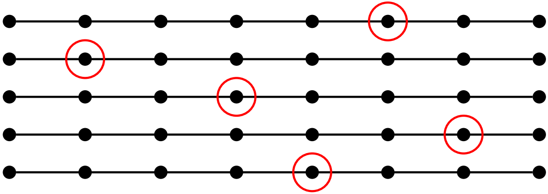
Restricted hop depth

Definition

shortest h-hop path: shortest among all paths with $\leq h$ edges

Suppose shortest h-hop path known for all pairs of nodes
$\Rightarrow$ Can compute all-pairs shortest paths in time $\mathcal{O}(n^3 \log n/h)$:

- Hitting set of size $\mathcal{O}(n/h)$ (probabilistic argument)



- Find hitting set C of size $\mathcal{O}(n \log n/h)$ in time $\mathcal{O}(n^2 h)$ (greedy)

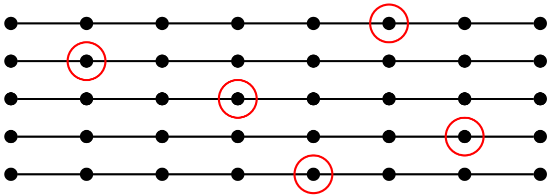
Restricted hop depth

Definition

shortest h-hop path: shortest among all paths with $\leq h$ edges

Suppose shortest h-hop path known for all pairs of nodes
$\Rightarrow$ Can compute all-pairs shortest paths in time $\mathcal{O}(n^3 \log n/h)$:

- Hitting set of size $\mathcal{O}(n/h)$ (probabilistic argument)



- Find hitting set C of size $\mathcal{O}(n \log n/h)$ in time $\mathcal{O}(n^2 h)$ (greedy)
- Compute shortest paths from nodes in C: $\mathcal{O}(n^3 \log n/h)$

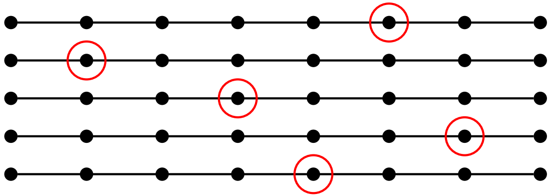
Restricted hop depth

Definition

shortest h-hop path: shortest among all paths with $\leq h$ edges

Suppose shortest h-hop path known for all pairs of nodes
$\Rightarrow$ Can compute all-pairs shortest paths in time $\mathcal{O}(n^3 \log n/h)$:

- Hitting set of size $\mathcal{O}(n/h)$ (probabilistic argument)



- Find hitting set C of size $\mathcal{O}(n \log n/h)$ in time $\mathcal{O}(n^2 h)$ (greedy)
- Compute shortest paths from nodes in C: $\mathcal{O}(n^3 \log n/h)$
- For all pairs s, t:
  $dist(s,t) = \min(dist^h(s,t), \min_{v \in C}(dist(s,v) + dist(v,t)))$

Summary

Known techniques allow the following restrictions:

1. Only necessary to maintain shortest h-hop paths up to length (for some parameter h)

Summary

Known techniques allow the following restrictions:

1. Only necessary to maintain shortest h-hop paths up to length (for some parameter h)
2. To obtain a fully dynamic algorithm it is sufficient to design a deletions-only algorithm that
   - can handle up to Δ deletions of nodes with worst-case guarantees
   - after **preprocessing** the graph

   Restart deletions-only algorithm each Δ updates

Barriers

Combinatorial approach [Thorup '05, Abraham/Chechik/Krinninger '17]

The best we can hope for:

- Preprocessing: $\mathcal{O}(n^3)$
- Spread preprocessing over Δ updates: $\mathcal{O}(n^3/k)$
- Deal with $\leq \Delta$ insertions after each update: $\mathcal{O}(n^2 k)$

$\Rightarrow \mathcal{O}(n^{2.5})$

Algebraic approach [Sankowski '04/'05]

Here: Intuition in DAGs

Algebraic approach [Sankowski '04/'05]

Here: Intuition in DAGs

Transitive closure:

- Count number of paths from s to t for all pairs
- Reachable iff $\#\text{paths} > 0$
- Perform operations for counting modulo random prime
- Update time $\mathcal{O}(n^2)$
- Avoids special treatment of insertions

Algebraic approach [Sankowski '04/'05]

Here: Intuition in DAGs

Transitive closure:

- Count number of paths from s to t for all pairs
- Reachable iff $\#\text{paths} > 0$
- Perform operations for counting modulo random prime
- Update time $\mathcal{O}(n^2)$
- Avoids special treatment of insertions

All-pairs shortest paths (distances):

- For every $1 \leq \ell \leq h$, count #paths of length exactly ℓ
- Additional trick: fast convolution
- Update time: $\tilde{\mathcal{O}}(n^2 h)$.
- Standard trick for hitting long paths: $h = \sqrt{n}$

$\Rightarrow \mathcal{O}(n^{2.5})$