Dynamic Treewidth

Tuukka Korhonen

UNIVERSITY OF BERGEN

based on joint work with Konrad Majewski, Wojciech Nadara, Michał Pilipczuk, and Marek Sokołowski, University of Warsaw

Friday Seminar

12 May 2023

Dynamic graph algorithms

Dynamic graph algorithms

- Setting: Design a data structure that maintains a graph G and supports the following operations:

Dynamic graph algorithms

- Setting: Design a data structure that maintains a graph G and supports the following operations:
  1. Initialize(n): Create G as an empty n-vertex graph

Dynamic graph algorithms

- Setting: Design a data structure that maintains a graph G and supports the following operations:
  1. Initialize(n): Create G as an empty n-vertex graph
  2. Insert(u, v): Insert edge between u and v

Dynamic graph algorithms

- Setting: Design a data structure that maintains a graph G and supports the following operations:
  1. Initialize(n): Create G as an empty n-vertex graph
  2. Insert(u, v): Insert edge between u and v
  3. Delete(u, v): Delete edge between u and v

Dynamic graph algorithms

- Setting: Design a data structure that maintains a graph G and supports the following operations:
  1. Initialize(n): Create G as an empty n-vertex graph
  2. Insert(u, v): Insert edge between u and v
  3. Delete(u, v): Delete edge between u and v
  4. Query: Ask something about the graph G

Dynamic graph algorithms

- Setting: Design a data structure that maintains a graph G and supports the following operations:
  1. Initialize(n): Create G as an empty n-vertex graph
  2. Insert(u, v): Insert edge between u and v
  3. Delete(u, v): Delete edge between u and v
  4. Query: Ask something about the graph G

Question

Can we support the operations faster than by re-computing from scratch every time?

Dynamic graph algorithms

- Setting: Design a data structure that maintains a graph G and supports the following operations:
  1. Initialize(n): Create G as an empty n-vertex graph
  2. Insert(u, v): Insert edge between u and v
  3. Delete(u, v): Delete edge between u and v
  4. Query: Ask something about the graph G

Question

Can we support the operations faster than by re-computing from scratch every time?

Example: Connectivity (Query: Are s and t in the same component?)

Dynamic graph algorithms

- Setting: Design a data structure that maintains a graph G and supports the following operations:
  1. Initialize(n): Create G as an empty n-vertex graph
  2. Insert(u, v): Insert edge between u and v
  3. Delete(u, v): Delete edge between u and v
  4. Query: Ask something about the graph G

Question

Can we support the operations faster than by re-computing from scratch every time?

Example: Connectivity (Query: Are s and t in the same component?)

1. Naive: $\mathcal{O}(m)$ worst-case time per operation

Dynamic graph algorithms

- Setting: Design a data structure that maintains a graph G and supports the following operations:
  1. Initialize(n): Create G as an empty n-vertex graph
  2. Insert(u, v): Insert edge between u and v
  3. Delete(u, v): Delete edge between u and v
  4. Query: Ask something about the graph G

Question

Can we support the operations faster than by re-computing from scratch every time?

Example: Connectivity (Query: Are s and t in the same component?)

1. Naive: $\mathcal{O}(m)$ worst-case time per operation
2. Union-find: $\mathcal{O}(\alpha(n))$ worst-case time, but deletions not allowed [Tarjan'75]

Dynamic graph algorithms

- Setting: Design a data structure that maintains a graph G and supports the following operations:
  1. Initialize(n): Create G as an empty n-vertex graph
  2. Insert(u, v): Insert edge between u and v
  3. Delete(u, v): Delete edge between u and v
  4. Query: Ask something about the graph G

Question

Can we support the operations faster than by re-computing from scratch every time?

Example: Connectivity (Query: Are s and t in the same component?)

1. Naive: $\mathcal{O}(m)$ worst-case time per operation
2. Union-find: $\mathcal{O}(\alpha(n))$ worst-case time, but deletions not allowed [Tarjan'75]
3. Link/cut tree: $\mathcal{O}(\log n)$ *amortized* time when G is a forest [Sleator&Tarjan'81]

Dynamic graph algorithms

- Setting: Design a data structure that maintains a graph G and supports the following operations:
  1. Initialize(n): Create G as an empty n-vertex graph
  2. Insert(u, v): Insert edge between u and v
  3. Delete(u, v): Delete edge between u and v
  4. Query: Ask something about the graph G

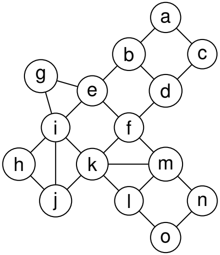
Question

Can we support the operations faster than by re-computing from scratch every time?

Example: Connectivity (Query: Are s and t in the same component?)

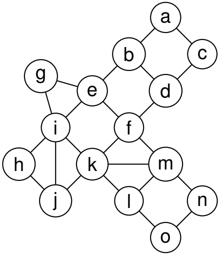
1. Naive: $\mathcal{O}(m)$ worst-case time per operation
2. Union-find: $\mathcal{O}(\alpha(n))$ worst-case time, but deletions not allowed [Tarjan'75]
3. Link/cut tree: $\mathcal{O}(\log n)$ *amortized* time when G is a forest [Sleator&Tarjan'81]
4. [Henzinger&King'99]: $\mathcal{O}(\log^3 n)$ amortized time

Treewidth

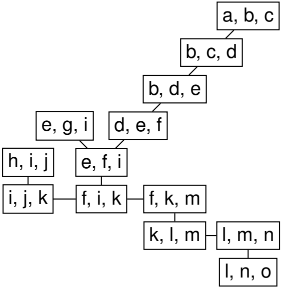


Graph G

Treewidth

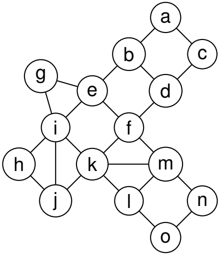


Graph G

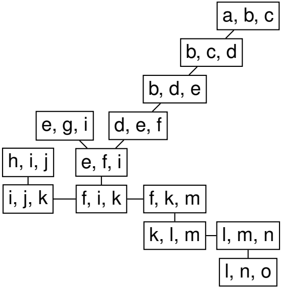


A tree decomposition of G

Treewidth



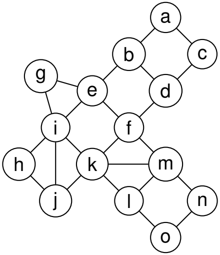
Graph G



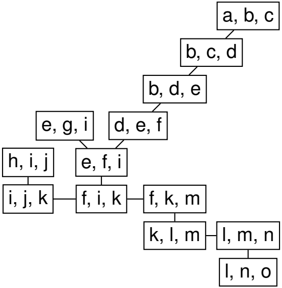
A tree decomposition of G

1. Every vertex should be in a bag

Treewidth



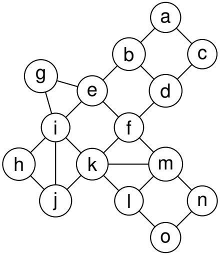
Graph G



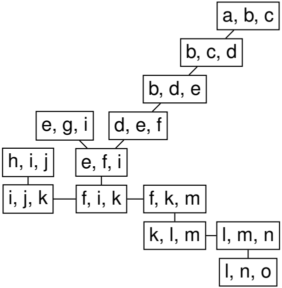
A tree decomposition of G

1. Every vertex should be in a bag
2. Every edge should be in a bag

Treewidth



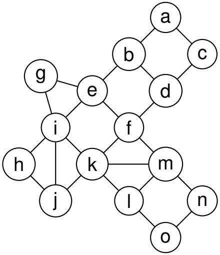
Graph G



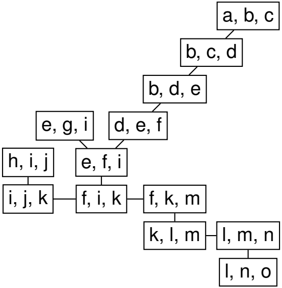
A tree decomposition of G

1. Every vertex should be in a bag
2. Every edge should be in a bag
3. Bags containing a vertex should form a connected subtree

Treewidth



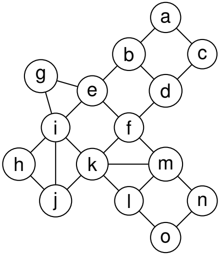
Graph G



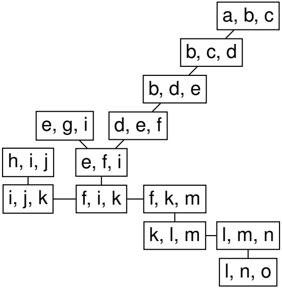
A tree decomposition of G

1. Every vertex should be in a bag
2. Every edge should be in a bag
3. Bags containing a vertex should form a connected subtree
4. Width = maximum bag size -1

Treewidth



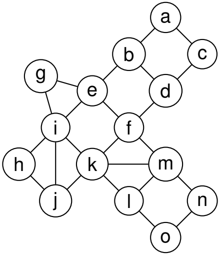
Graph G



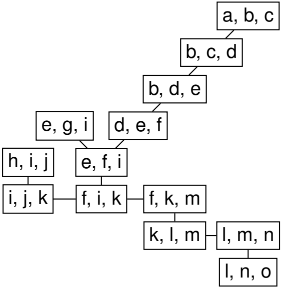
A tree decomposition of G
Width = 2

1. Every vertex should be in a bag
2. Every edge should be in a bag
3. Bags containing a vertex should form a connected subtree
4. Width = maximum bag size -1

Treewidth



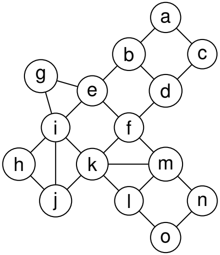
Graph G



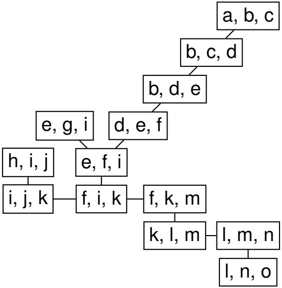
A tree decomposition of G
Width = 2

1. Every vertex should be in a bag
2. Every edge should be in a bag
3. Bags containing a vertex should form a connected subtree
4. Width = maximum bag size -1
5. Treewidth of G = minimum width of tree decomposition of G

Treewidth



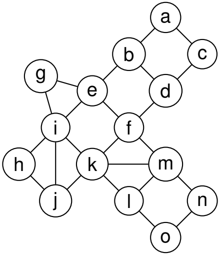
Graph G
Treewidth 2



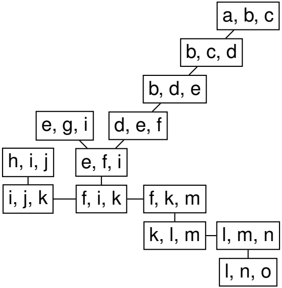
A tree decomposition of G
Width = 2

1. Every vertex should be in a bag
2. Every edge should be in a bag
3. Bags containing a vertex should form a connected subtree
4. Width = maximum bag size -1
5. Treewidth of G = minimum width of tree decomposition of G

Treewidth



Graph G
Treewidth 2



A tree decomposition of G
Width = 2

1. Every vertex should be in a bag
2. Every edge should be in a bag
3. Bags containing a vertex should form a connected subtree
4. Width = maximum bag size -1
5. Treewidth of G = minimum width of tree decomposition of G

[Robertson & Seymour'84, Bertele & Brioschi'72, Halin'76]

Dynamic treewidth

Question

Can we maintain a bounded-width tree decomposition of a bounded treewidth graph in the dynamic setting?

Dynamic treewidth

Question

Can we maintain a bounded-width tree decomposition of a bounded treewidth graph in the dynamic setting?

- Also, we would like to maintain any "finite-state" dynamic programming scheme on the tree decomposition

Dynamic treewidth

Question

Can we maintain a bounded-width tree decomposition of a bounded treewidth graph in the dynamic setting?

- Also, we would like to maintain any “finite-state” dynamic programming scheme on the tree decomposition (dynamic Courcelle’s theorem)

Dynamic treewidth

Question

Can we maintain a bounded-width tree decomposition of a bounded treewidth graph in the dynamic setting?

- Also, we would like to maintain any "finite-state" dynamic programming scheme on the tree decomposition (dynamic Courcelle's theorem)

Previous results:

Dynamic treewidth

Question

Can we maintain a bounded-width tree decomposition of a bounded treewidth graph in the dynamic setting?

- Also, we would like to maintain any “finite-state” dynamic programming scheme on the tree decomposition (dynamic Courcelle’s theorem)

Previous results:

- “Naive”: $\mathcal{O}_k(n)$ worst-case time per operation [Bodlaender’96]

Dynamic treewidth

Question

Can we maintain a bounded-width tree decomposition of a bounded treewidth graph in the dynamic setting?

- Also, we would like to maintain any "finite-state" dynamic programming scheme on the tree decomposition (dynamic Courcelle's theorem)

Previous results:

- "Naive": $\mathcal{O}_k(n)$ worst-case time per operation [Bodlaender'96]
- [Bodlaender'93]: $\mathcal{O}(\log n)$ worst-case time for treewidth-2 graphs

Dynamic treewidth

Question

Can we maintain a bounded-width tree decomposition of a bounded treewidth graph in the dynamic setting?

- Also, we would like to maintain any "finite-state" dynamic programming scheme on the tree decomposition (dynamic Courcelle's theorem)

Previous results:

- "Naive": $\mathcal{O}_k(n)$ worst-case time per operation [Bodlaender'96]
- [Bodlaender'93]: $\mathcal{O}(\log n)$ worst-case time for treewidth-2 graphs
- [Cohen,Sairam,Tamassia,Vitter'93]: $\mathcal{O}(\log n)$ worst-case time for treewidth-3, no edge deletions allowed

Dynamic treewidth

Question

Can we maintain a bounded-width tree decomposition of a bounded treewidth graph in the dynamic setting?

- Also, we would like to maintain any "finite-state" dynamic programming scheme on the tree decomposition (dynamic Courcelle's theorem)

Previous results:

- "Naive": $\mathcal{O}_k(n)$ worst-case time per operation [Bodlaender'96]
- [Bodlaender'93]: $\mathcal{O}(\log n)$ worst-case time for treewidth-2 graphs
- [Cohen,Sairam,Tamassia,Vitter'93]: $\mathcal{O}(\log n)$ worst-case time for treewidth-3, no edge deletions allowed
- [Dvořák,Kupec,Tůma'14]: $\mathcal{O}_d(1)$ worst-case time for treedepth-d

Dynamic treewidth

Question

Can we maintain a bounded-width tree decomposition of a bounded treewidth graph in the dynamic setting?

- Also, we would like to maintain any "finite-state" dynamic programming scheme on the tree decomposition (dynamic Courcelle's theorem)

Previous results:

- "Naive": $\mathcal{O}_k(n)$ worst-case time per operation [Bodlaender'96]
- [Bodlaender'93]: $\mathcal{O}(\log n)$ worst-case time for treewidth-2 graphs
- [Cohen,Sairam,Tamassia,Vitter'93]: $\mathcal{O}(\log n)$ worst-case time for treewidth-3, no edge deletions allowed
- [Dvořák,Kupec,Tůma'14]: $\mathcal{O}_d(1)$ worst-case time for treedepth-d
- [Majewski,Pilipczuk,Sokołowski'23]: $\mathcal{O}_\ell(\log n)$ amortized time for feedback vertex number ℓ

Dynamic treewidth

Question

Can we maintain a bounded-width tree decomposition of a bounded treewidth graph in the dynamic setting?

- Also, we would like to maintain any “finite-state” dynamic programming scheme on the tree decomposition (dynamic Courcelle’s theorem)

Previous results:

- “Naive”: $\mathcal{O}_k(n)$ worst-case time per operation [Bodlaender’96]
- [Bodlaender’93]: $\mathcal{O}(\log n)$ worst-case time for treewidth-2 graphs
- [Cohen,Sairam,Tamassia,Vitter’93]: $\mathcal{O}(\log n)$ worst-case time for treewidth-3, no edge deletions allowed
- [Dvořák,Kupec,Tůma’14]: $\mathcal{O}_d(1)$ worst-case time for treedepth-d
- [Majewski,Pilipczuk,Sokołowski’23]: $\mathcal{O}_\ell(\log n)$ amortized time for feedback vertex number ℓ
- [Goranci,Saranurak,Tan’21]: $n^{o(1)}$ amortized time $n^{o(1)}$-approximate tree decomposition. Not suitable for dynamic programming.

Our result

Summary of previous results

No non-trivial algorithms for maintaining tree decompositions of width $f(k)$ for fully dynamic graphs of treewidth $k > 3$.

Our result

Summary of previous results

No non-trivial algorithms for maintaining tree decompositions of width $f(k)$ for fully dynamic graphs of treewidth $k > 3$.

Theorem (This work)

There is a data structure that is initialized with an integer k and an empty n-vertex graph G, and maintains a tree decomposition of G of width at most $6k + 5$ under edge additions and deletions in amortized update time $\mathcal{O}_k(2^{\sqrt{\log n} \log \log n})$, under the promise that the treewidth of G never exceeds k.

Our result

Summary of previous results

No non-trivial algorithms for maintaining tree decompositions of width $f(k)$ for fully dynamic graphs of treewidth $k > 3$.

Theorem (This work)

There is a data structure that is initialized with an integer k and an empty n-vertex graph G, and maintains a tree decomposition of G of width at most $6k + 5$ under edge additions and deletions in amortized update time $\mathcal{O}_k(2^{\sqrt{\log n} \log \log n})$, under the promise that the treewidth of G never exceeds k.

Moreover

Our result

Summary of previous results

No non-trivial algorithms for maintaining tree decompositions of width $f(k)$ for fully dynamic graphs of treewidth $k > 3$.

Theorem (This work)

There is a data structure that is initialized with an integer k and an empty n-vertex graph G, and maintains a tree decomposition of G of width at most $6k + 5$ under edge additions and deletions in amortized update time $\mathcal{O}_k(2^{\sqrt{\log n} \log \log n})$, under the promise that the treewidth of G never exceeds k.

Moreover

- the data structure can maintain the run of any tree automaton with evaluation time $\mathcal{O}_k(1)$ within the same running time

Our result

Summary of previous results

No non-trivial algorithms for maintaining tree decompositions of width $f(k)$ for fully dynamic graphs of treewidth $k > 3$.

Theorem (This work)

There is a data structure that is initialized with an integer k and an empty n-vertex graph G, and maintains a tree decomposition of G of width at most $6k + 5$ under edge additions and deletions in amortized update time $\mathcal{O}_k(2^{\sqrt{\log n} \log \log n})$, under the promise that the treewidth of G never exceeds k.

Moreover

- the data structure can maintain the run of any tree automaton with evaluation time $\mathcal{O}_k(1)$ within the same running time
- the data structure persists even when the treewidth of G exceeds k, in that case returning a marker “Treewidth too large” instead of maintaining the automaton

Example application

Corollary

Let H be fixed planar graph. There is a dynamic algorithm with $\mathcal{O}_H(2^{\sqrt{\log n}\log\log n})$ amortized update time for maintaining whether G contains H as a minor.

Example application

Corollary

Let H be fixed planar graph. There is a dynamic algorithm with $\mathcal{O}_H(2^{\sqrt{\log n} \log \log n})$ amortized update time for maintaining whether G contains H as a minor.

Proof:

Example application

Corollary

Let H be fixed planar graph. There is a dynamic algorithm with $\mathcal{O}_H(2^{\sqrt{\log n} \log \log n})$ amortized update time for maintaining whether G contains H as a minor.

Proof:

- By the Grid Minor Theorem [Robertson&Seymour'85], there exists k so that every graph of treewidth $> k$ contains H as a minor

Example application

Corollary

Let H be fixed planar graph. There is a dynamic algorithm with $\mathcal{O}_H(2^{\sqrt{\log n}\log\log n})$ amortized update time for maintaining whether G contains H as a minor.

Proof:

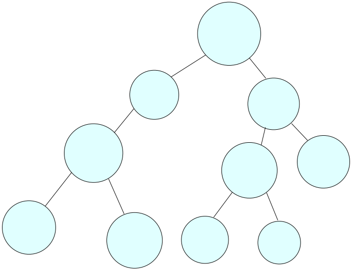
- By the Grid Minor Theorem [Robertson&Seymour'85], there exists k so that every graph of treewidth $> k$ contains H as a minor
- Use dynamic treewidth data structure with this k and a tree automaton that tests for H as a minor by dynamic programming □

The algorithm

General plan

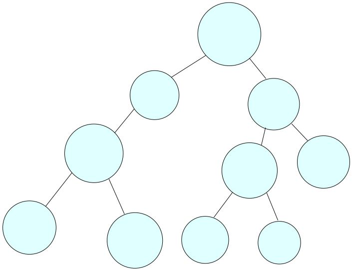
General plan

- Goal: Maintain a rooted binary tree decomposition of width $6k + 5$ and depth $d = 2^{\mathcal{O}_k(\sqrt{\log n \log \log n})}$



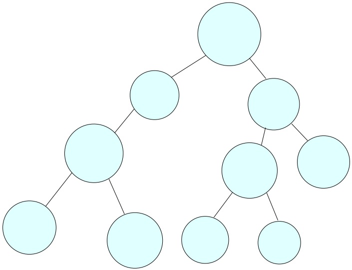
General plan

- Goal: Maintain a rooted binary tree decomposition of width $6k+5$ and depth $d = 2^{\mathcal{O}_k(\sqrt{\log n \log\log n})}$
- [Bodlaender&Hagerup'98]: Any tree decomposition of width k can be turned into rooted binary tree decomposition of depth $\mathcal{O}(\log n)$ and width $3k+2$



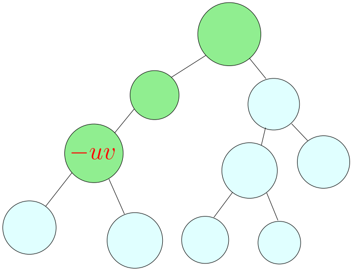
General plan

- Goal: Maintain a rooted binary tree decomposition of width $6k + 5$ and depth $d = 2^{\mathcal{O}_k(\sqrt{\log n \log \log n})}$
- [Bodlaender&Hagerup'98]: Any tree decomposition of width k can be turned into rooted binary tree decomposition of depth $\mathcal{O}(\log n)$ and width $3k + 2$
- Maintain also dynamic programming tables directed towards the root



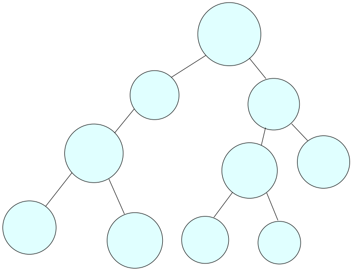
General plan

- Goal: Maintain a rooted binary tree decomposition of width $6k + 5$ and depth $d = 2^{\mathcal{O}_k(\sqrt{\log n \log \log n})}$
- [Bodlaender&Hagerup'98]: Any tree decomposition of width k can be turned into rooted binary tree decomposition of depth $\mathcal{O}(\log n)$ and width $3k + 2$
- Maintain also dynamic programming tables directed towards the root
- Edge deletion: Re-compute dynamic programming tables in time $\mathcal{O}_k(d)$



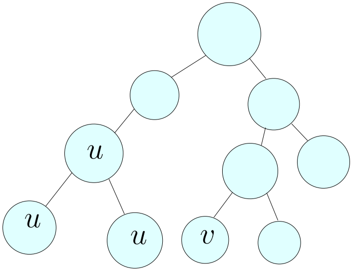
General plan

- Goal: Maintain a rooted binary tree decomposition of width $6k + 5$ and depth $d = 2^{\mathcal{O}_k(\sqrt{\log n \log \log n})}$
- [Bodlaender&Hagerup'98]: Any tree decomposition of width k can be turned into rooted binary tree decomposition of depth $\mathcal{O}(\log n)$ and width $3k + 2$
- Maintain also dynamic programming tables directed towards the root
- Edge deletion: Re-compute dynamic programming tables in time $\mathcal{O}_k(d)$



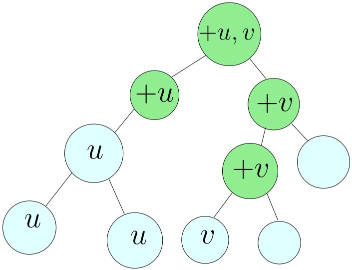
General plan

- Goal: Maintain a rooted binary tree decomposition of width $6k+5$ and depth $d = 2^{\mathcal{O}_k(\sqrt{\log n \log\log n})}$
- [Bodlaender&Hagerup'98]: Any tree decomposition of width k can be turned into rooted binary tree decomposition of depth $\mathcal{O}(\log n)$ and width $3k+2$
- Maintain also dynamic programming tables directed towards the root
- Edge deletion: Re-compute dynamic programming tables in time $\mathcal{O}_k(d)$
- Edge addition: Add u and v to all bags on the path from their subtrees to the root, and re-compute dynamic programming tables in time $\mathcal{O}_k(d)$

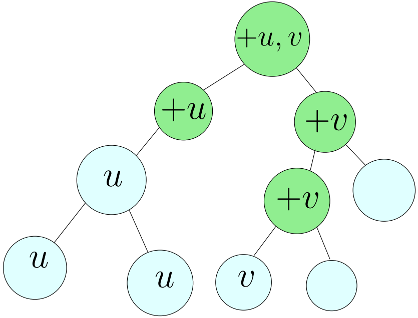


General plan

- Goal: Maintain a rooted binary tree decomposition of width $6k+5$ and depth $d = 2^{\mathcal{O}_k(\sqrt{\log n \log \log n})}$
- [Bodlaender&Hagerup'98]: Any tree decomposition of width k can be turned into rooted binary tree decomposition of depth $\mathcal{O}(\log n)$ and width $3k+2$
- Maintain also dynamic programming tables directed towards the root
- Edge deletion: Re-compute dynamic programming tables in time $\mathcal{O}_k(d)$
- Edge addition: Add u and v to all bags on the path from their subtrees to the root, and re-compute dynamic programming tables in time $\mathcal{O}_k(d)$

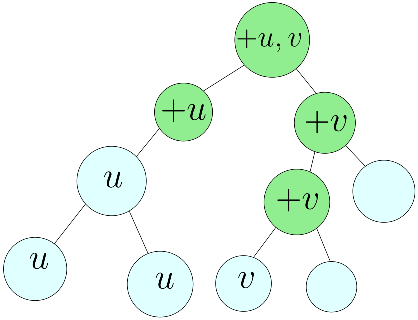


What can go wrong?



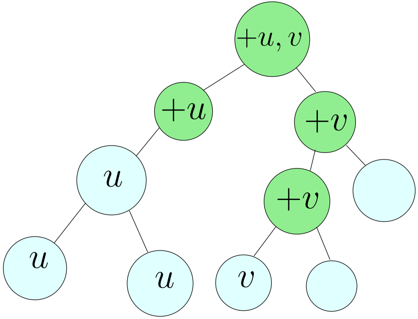
What can go wrong?

- The width can become more than $6k + 5$ on the green bags!



What can go wrong?

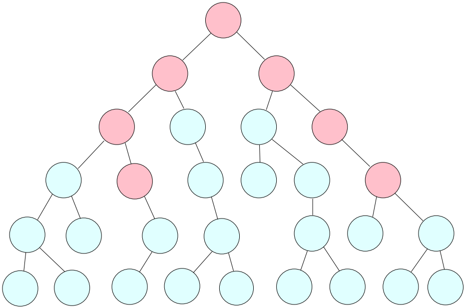
- The width can become more than $6k + 5$ on the green bags!
- Solution: a *Refinement operation* to re-compute the tree decomposition on these bags



Refinement operation

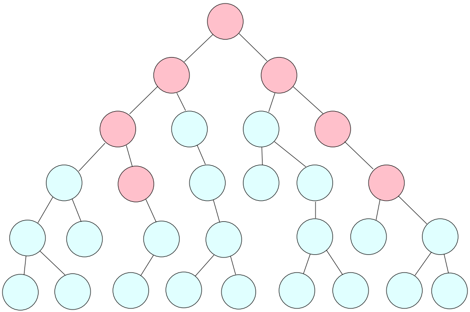
Refinement operation

- Refinement operation is given a *prefix* T_{pref} of the tree decomposition that contains all bags of width $> 6k + 5$



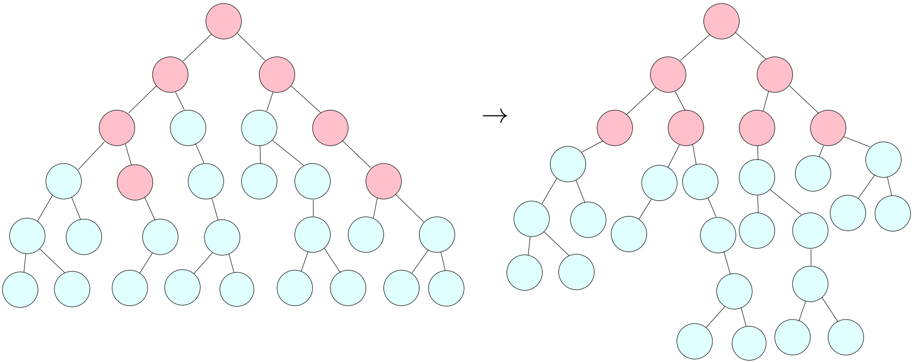
Refinement operation

- Refinement operation is given a *prefix* T_{pref} of the tree decomposition that contains all bags of width $> 6k + 5$
- Re-arranges the prefix into new prefix of width $\leq 6k + 5$ and depth $\leq \mathcal{O}(\log n)$



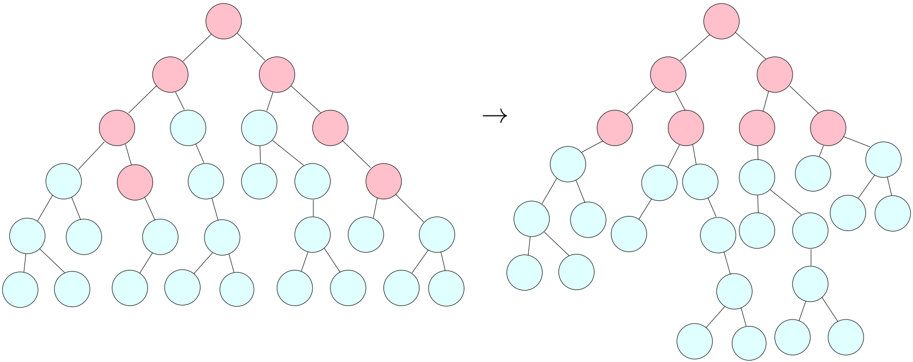
Refinement operation

- Refinement operation is given a *prefix* T_{pref} of the tree decomposition that contains all bags of width $> 6k + 5$
- Re-arranges the prefix into new prefix of width $\leq 6k + 5$ and depth $\leq \mathcal{O}(\log n)$



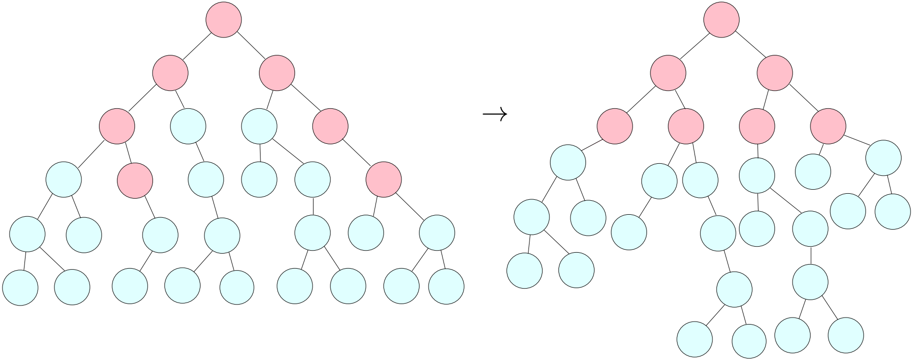
Refinement operation

- Refinement operation is given a *prefix* T_{pref} of the tree decomposition that contains all bags of width $> 6k + 5$
- Re-arranges the prefix into new prefix of width $\leq 6k + 5$ and depth $\leq \mathcal{O}(\log n)$
- Changes also other parts of the decomposition, but only improves the width, and the amortized amount of bags changed and the amortized complexity of the operation is $\mathcal{O}_k(|T_{\mathrm{pref}}|)$

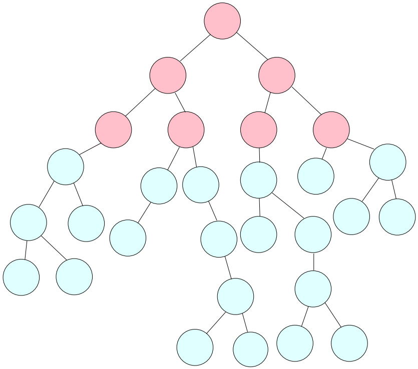


Refinement operation

- Refinement operation is given a *prefix* T_{pref} of the tree decomposition that contains all bags of width $> 6k + 5$
- Re-arranges the prefix into new prefix of width $\leq 6k + 5$ and depth $\leq \mathcal{O}(\log n)$
- Changes also other parts of the decomposition, but only improves the width, and the amortized amount of bags changed and the amortized complexity of the operation is $\mathcal{O}_k(|T_{\text{pref}}|)$
- Builds on the improvement operation of [K & Lokshtanov'23], also uses the dealternation lemma of [Bojańczyk&Pilipczuk'22] and Bodlaender-Hagerup-lemma

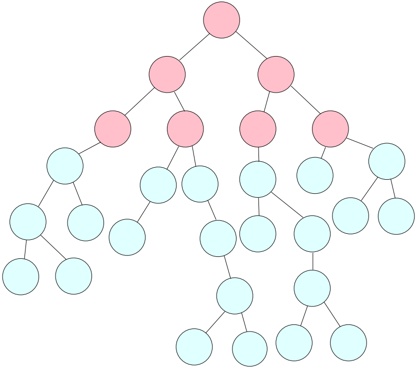


What can go wrong?



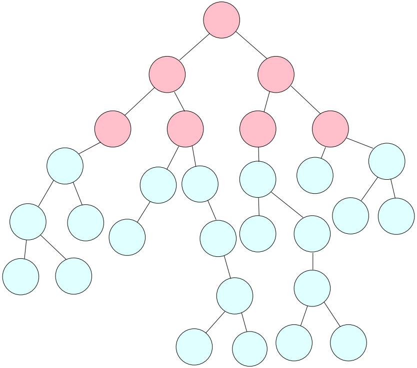
What can go wrong?

- Refinement operation can increase the depth by $\mathcal{O}(\log n)$



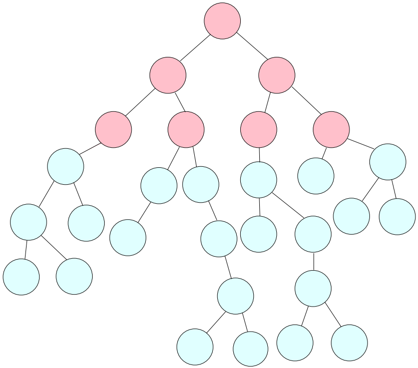
What can go wrong?

- Refinement operation can increase the depth by $\mathcal{O}(\log n)$
- Once depth becomes more than $2^{\mathcal{O}_k(\sqrt{\log n \log \log n})}$, need to reduce it



What can go wrong?

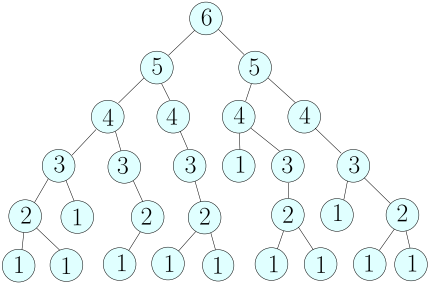
- Refinement operation can increase the depth by $\mathcal{O}(\log n)$
- Once depth becomes more than $2^{\mathcal{O}_k(\sqrt{\log n \log \log n})}$, need to reduce it
- Solution: A depth-reduction scheme by using the refinement operation and a potential function



Potential function

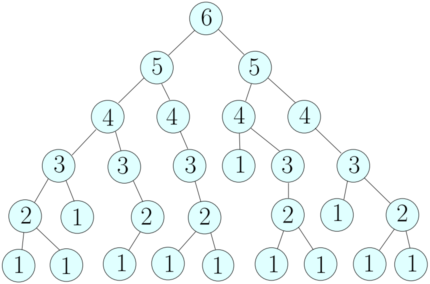
Potential function

- Potential function of form $\phi(T) = \sum_{t \in V(T)} k^{10 \cdot |B_t|} \cdot \text{height}(t)$



Potential function

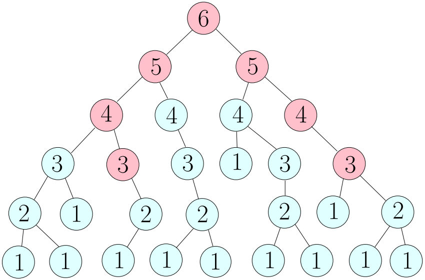
- Potential function of form $\phi(T) = \sum_{t \in V(T)} k^{10 \cdot |B_t|} \cdot \text{height}(t)$
- Inserting edge increases potential by $\mathcal{O}_k(d^2) = 2^{\mathcal{O}_k(\sqrt{\log n \log \log n})}$



Potential function

- Potential function of form $\phi(T) = \sum_{t \in V(T)} k^{10 \cdot |B_t|} \cdot \text{height}(t)$
- Inserting edge increases potential by $\mathcal{O}_k(d^2) = 2^{\mathcal{O}_k(\sqrt{\log n \log \log n})}$

How does refinement change potential?

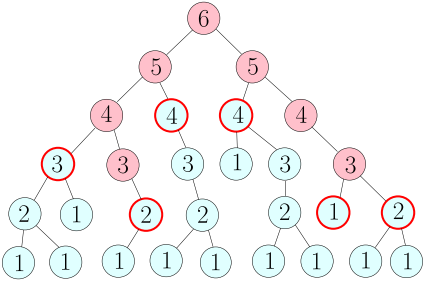


Potential function

- Potential function of form $\phi(T) = \sum_{t \in V(T)} k^{10 \cdot |B_t|} \cdot \text{height}(t)$
- Inserting edge increases potential by $\mathcal{O}_k(d^2) = 2^{\mathcal{O}_k(\sqrt{\log n \log \log n})}$

How does refinement change potential?

- Let $A \subseteq V(T)$ be the set of *appendices* of T_{pref}



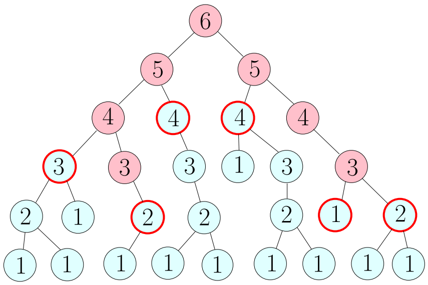
Potential function

- Potential function of form $\phi(T) = \sum_{t \in V(T)} k^{10 \cdot |B_t|} \cdot \text{height}(t)$
- Inserting edge increases potential by $\mathcal{O}_k(d^2) = 2^{\mathcal{O}_k(\sqrt{\log n \log \log n})}$

How does refinement change potential?

- Let $A \subseteq V(T)$ be the set of *appendices* of T_{pref}

$$\phi(T') \leq \phi(T) - \phi(T_{\text{pref}}) + \mathcal{O}_k(|T_{\text{pref}}| \cdot \log n) + \sum_{t \in A} \mathcal{O}_k(\text{height}(t) \cdot \log n)$$



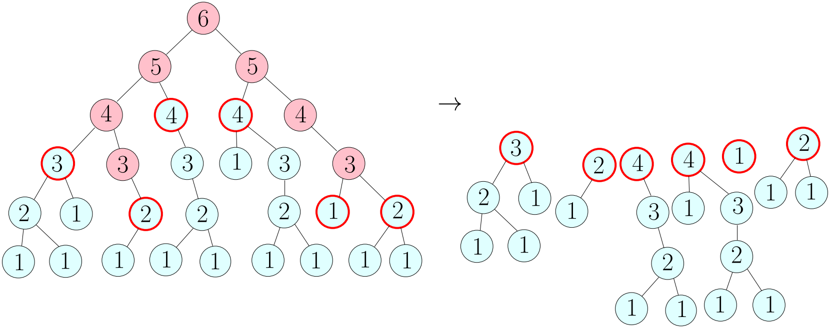
Potential function

- Potential function of form $\phi(T) = \sum_{t \in V(T)} k^{10 \cdot |B_t|} \cdot \text{height}(t)$
- Inserting edge increases potential by $\mathcal{O}_k(d^2) = 2^{\mathcal{O}_k(\sqrt{\log n \log \log n})}$

How does refinement change potential?

- Let $A \subseteq V(T)$ be the set of *appendices* of T_{pref}

$$\phi(T') \leq \phi(T) - \phi(T_{\text{pref}}) + \mathcal{O}_k(|T_{\text{pref}}| \cdot \log n) + \sum_{t \in A} \mathcal{O}_k(\text{height}(t) \cdot \log n)$$



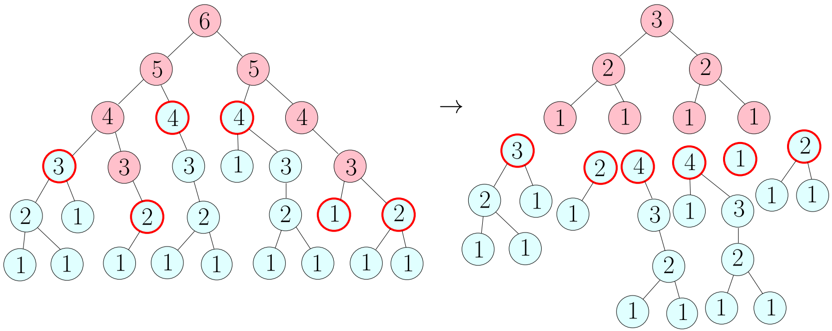
Potential function

- Potential function of form $\phi(T) = \sum_{t \in V(T)} k^{10 \cdot |B_t|} \cdot \text{height}(t)$
- Inserting edge increases potential by $\mathcal{O}_k(d^2) = 2^{\mathcal{O}_k(\sqrt{\log n \log \log n})}$

How does refinement change potential?

- Let $A \subseteq V(T)$ be the set of *appendices* of T_{pref}

$$\phi(T') \leq \phi(T) - \phi(T_{\text{pref}}) + \mathcal{O}_k(|T_{\text{pref}}| \cdot \log n) + \sum_{t \in A} \mathcal{O}_k(\text{height}(t) \cdot \log n)$$



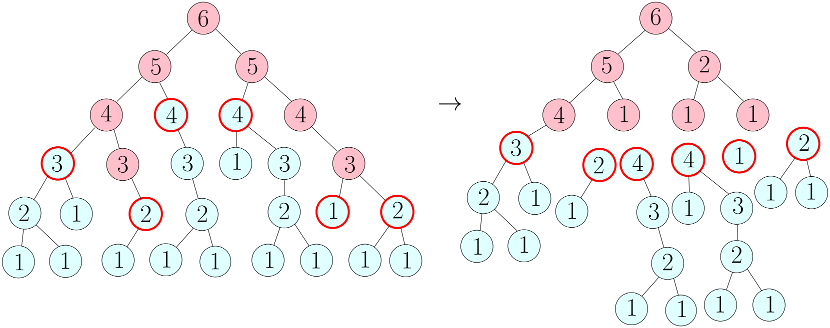
Potential function

- Potential function of form $\phi(T) = \sum_{t \in V(T)} k^{10 \cdot |B_t|} \cdot \text{height}(t)$
- Inserting edge increases potential by $\mathcal{O}_k(d^2) = 2^{\mathcal{O}_k(\sqrt{\log n \log \log n})}$

How does refinement change potential?

- Let $A \subseteq V(T)$ be the set of *appendices* of T_{pref}

$$\phi(T') \leq \phi(T) - \phi(T_{\text{pref}}) + \mathcal{O}_k(|T_{\text{pref}}| \cdot \log n) + \sum_{t \in A} \mathcal{O}_k(\text{height}(t) \cdot \log n)$$



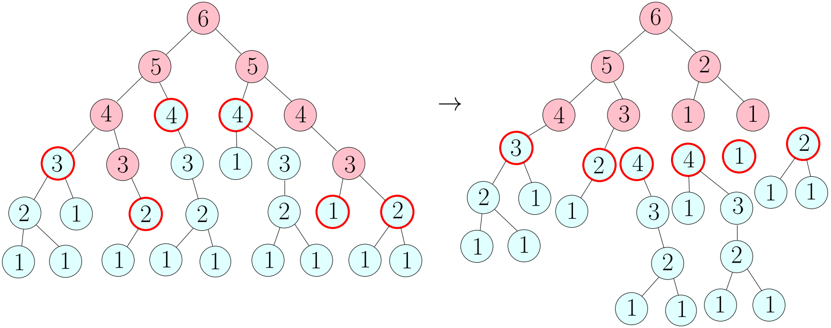
Potential function

- Potential function of form $\phi(T) = \sum_{t \in V(T)} k^{10 \cdot |B_t|} \cdot \text{height}(t)$
- Inserting edge increases potential by $\mathcal{O}_k(d^2) = 2^{\mathcal{O}_k(\sqrt{\log n \log \log n})}$

How does refinement change potential?

- Let $A \subseteq V(T)$ be the set of *appendices* of T_{pref}

$$\phi(T') \leq \phi(T) - \phi(T_{\text{pref}}) + \mathcal{O}_k(|T_{\text{pref}}| \cdot \log n) + \sum_{t \in A} \mathcal{O}_k(\text{height}(t) \cdot \log n)$$



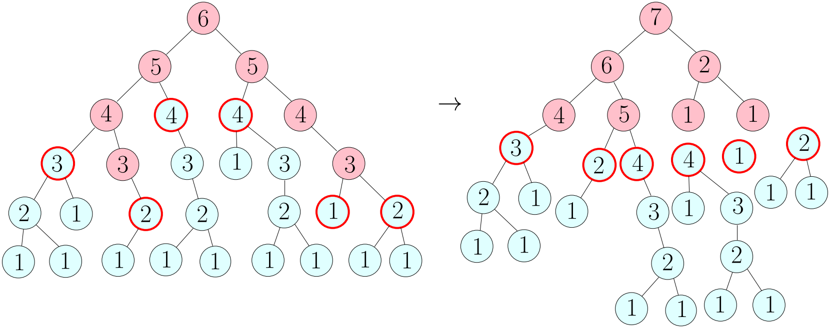
Potential function

- Potential function of form $\phi(T) = \sum_{t \in V(T)} k^{10 \cdot |B_t|} \cdot \text{height}(t)$
- Inserting edge increases potential by $\mathcal{O}_k(d^2) = 2^{\mathcal{O}_k(\sqrt{\log n \log \log n})}$

How does refinement change potential?

- Let $A \subseteq V(T)$ be the set of *appendices* of T_{pref}

$$\phi(T') \leq \phi(T) - \phi(T_{\text{pref}}) + \mathcal{O}_k(|T_{\text{pref}}| \cdot \log n) + \sum_{t \in A} \mathcal{O}_k(\text{height}(t) \cdot \log n)$$



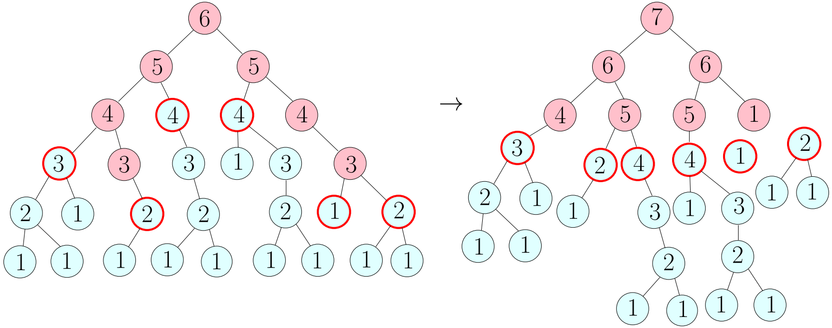
Potential function

- Potential function of form $\phi(T) = \sum_{t \in V(T)} k^{10 \cdot |B_t|} \cdot \text{height}(t)$
- Inserting edge increases potential by $\mathcal{O}_k(d^2) = 2^{\mathcal{O}_k(\sqrt{\log n \log \log n})}$

How does refinement change potential?

- Let $A \subseteq V(T)$ be the set of *appendices* of T_{pref}

$$\phi(T') \leq \phi(T) - \phi(T_{\text{pref}}) + \mathcal{O}_k(|T_{\text{pref}}| \cdot \log n) + \sum_{t \in A} \mathcal{O}_k(\text{height}(t) \cdot \log n)$$



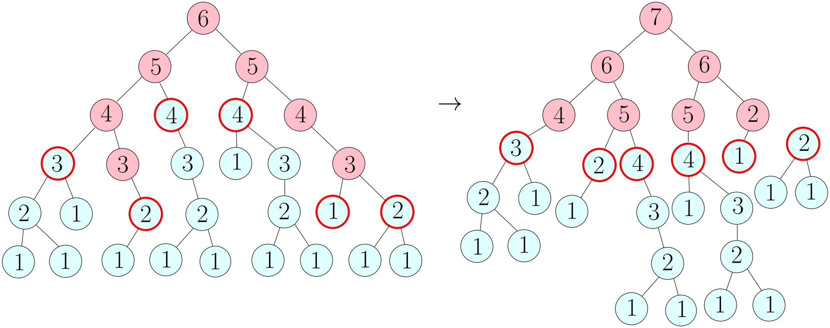
Potential function

- Potential function of form $\phi(T) = \sum_{t \in V(T)} k^{10 \cdot |B_t|} \cdot \text{height}(t)$
- Inserting edge increases potential by $\mathcal{O}_k(d^2) = 2^{\mathcal{O}_k(\sqrt{\log n \log \log n})}$

How does refinement change potential?

- Let $A \subseteq V(T)$ be the set of *appendices* of T_{pref}

$$\phi(T') \leq \phi(T) - \phi(T_{\text{pref}}) + \mathcal{O}_k(|T_{\text{pref}}| \cdot \log n) + \sum_{t \in A} \mathcal{O}_k(\text{height}(t) \cdot \log n)$$



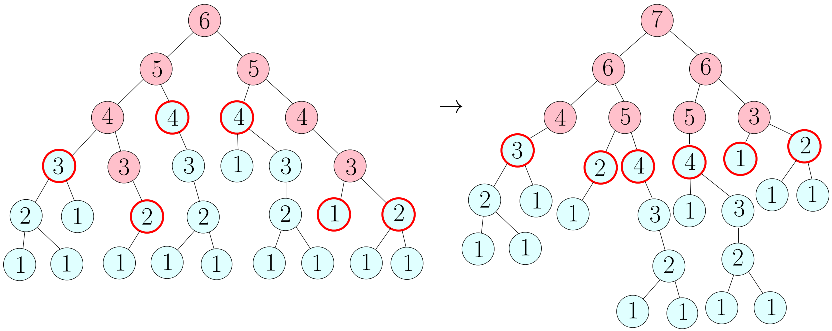
Potential function

- Potential function of form $\phi(T) = \sum_{t \in V(T)} k^{10 \cdot |B_t|} \cdot \text{height}(t)$
- Inserting edge increases potential by $\mathcal{O}_k(d^2) = 2^{\mathcal{O}_k(\sqrt{\log n \log \log n})}$

How does refinement change potential?

- Let $A \subseteq V(T)$ be the set of *appendices* of T_{pref}

$$\phi(T') \leq \phi(T) - \phi(T_{\text{pref}}) + \mathcal{O}_k(|T_{\text{pref}}| \cdot \log n) + \sum_{t \in A} \mathcal{O}_k(\text{height}(t) \cdot \log n)$$



Potential function

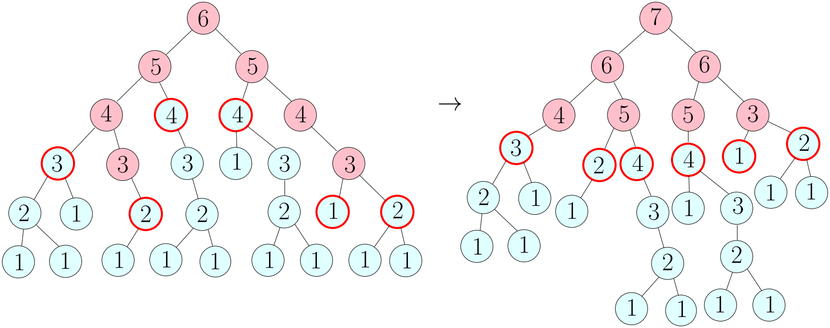
- Potential function of form $\phi(T) = \sum_{t \in V(T)} k^{10 \cdot |B_t|} \cdot \text{height}(t)$
- Inserting edge increases potential by $\mathcal{O}_k(d^2) = 2^{\mathcal{O}_k(\sqrt{\log n \log \log n})}$

How does refinement change potential?

- Let $A \subseteq V(T)$ be the set of *appendices* of T_{pref}

$$\phi(T') \leq \phi(T) - \phi(T_{\text{pref}}) + \mathcal{O}_k(|T_{\text{pref}}| \cdot \log n) + \sum_{t \in A} \mathcal{O}_k(\text{height}(t) \cdot \log n)$$

- Width-reduction increases potential by $\mathcal{O}_k(d^2 \log n) = 2^{\mathcal{O}_k(\sqrt{\log n \log \log n})}$



Depth-reduction scheme

Depth-reduction scheme

- Idea: If depth too large, can decrease potential "for free"

Depth-reduction scheme

- Idea: If depth too large, can decrease potential "for free"

Lemma

If depth $> 2^{\mathcal{O}_k(\sqrt{\log n \log \log n})}$, then exists prefix T_{pref} so that $\phi(T') < \phi(T) - \Omega(\phi(T_{\mathrm{pref}}))$.

Depth-reduction scheme

- Idea: If depth too large, can decrease potential “for free”

Lemma

If depth $> 2^{\mathcal{O}_k(\sqrt{\log n \log \log n})}$, then exists prefix T_{pref} so that $\phi(T') < \phi(T) - \Omega(\phi(T_{\text{pref}}))$.

$\Rightarrow$ We can decrease potential in time propotional to the decrease

Depth-reduction scheme

- Idea: If depth too large, can decrease potential “for free”

Lemma

If depth $> 2^{\mathcal{O}_k(\sqrt{\log n \log \log n})}$, then exists prefix T_{pref} so that $\phi(T') < \phi(T) - \Omega(\phi(T_{\text{pref}}))$.

$\Rightarrow$ We can decrease potential in time propotional to the decrease

$\Rightarrow$ Amortized time complexity bounded by the potential

Depth-reduction scheme

- Idea: If depth too large, can decrease potential "for free"

Lemma

If depth $> 2^{\mathcal{O}_k(\sqrt{\log n \log \log n})}$, then exists prefix T_{pref} so that $\phi(T') < \phi(T) - \Omega(\phi(T_{\text{pref}}))$.

$\Rightarrow$ We can decrease potential in time propotional to the decrease

$\Rightarrow$ Amortized time complexity bounded by the potential

$\Rightarrow$ Can keep depth at most $2^{\mathcal{O}_k(\sqrt{\log n \log \log n})}$ with amortized time complexity $2^{\mathcal{O}_k(\sqrt{\log n \log \log n})}$

Conclusion

- $\mathcal{O}_k(2^{\sqrt{\log n}\log\log n})$ amortized update time for maintaining a tree decomposition of width at most $6k+5$ of dynamic graph of treewidth $\leq k$

Conclusion

- $\mathcal{O}_k(2^{\sqrt{\log n} \log \log n})$ amortized update time for maintaining a tree decomposition of width at most $6k + 5$ of dynamic graph of treewidth $\leq k$
  - Can also maintain any dynamic programming on the tree decomposition

Conclusion

- $\mathcal{O}_k(2^{\sqrt{\log n} \log \log n})$ amortized update time for maintaining a tree decomposition of width at most $6k + 5$ of dynamic graph of treewidth $\leq k$
  - Can also maintain any dynamic programming on the tree decomposition

- Open problems and directions:

Conclusion

- $\mathcal{O}_k(2^{\sqrt{\log n}\log\log n})$ amortized update time for maintaining a tree decomposition of width at most $6k+5$ of dynamic graph of treewidth $\leq k$
  - Can also maintain any dynamic programming on the tree decomposition

- Open problems and directions:
  - Improve to $\mathcal{O}_k(\text{poly}\log n)$

Conclusion

- $\mathcal{O}_k(2^{\sqrt{\log n}\log\log n})$ amortized update time for maintaining a tree decomposition of width at most $6k + 5$ of dynamic graph of treewidth $\leq k$
  - Can also maintain any dynamic programming on the tree decomposition

- Open problems and directions:
  - Improve to $\mathcal{O}_k(\text{poly}\log n)$
    - Conjecture: Can be improved to $\mathcal{O}_k(\log n)$

Conclusion

- $\mathcal{O}_k(2^{\sqrt{\log n}\log\log n})$ amortized update time for maintaining a tree decomposition of width at most $6k+5$ of dynamic graph of treewidth $\leq k$
  - Can also maintain any dynamic programming on the tree decomposition

- Open problems and directions:
  - Improve to $\mathcal{O}_k(\text{poly}\log n)$
    - Conjecture: Can be improved to $\mathcal{O}_k(\log n)$
  - Dynamic version of Baker's technique: dynamic approximation scheme for DOMINATING SET on planar graphs?

Conclusion

- $\mathcal{O}_k(2^{\sqrt{\log n} \log \log n})$ amortized update time for maintaining a tree decomposition of width at most $6k + 5$ of dynamic graph of treewidth $\leq k$
  - Can also maintain any dynamic programming on the tree decomposition

- Open problems and directions:
  - Improve to $\mathcal{O}_k(\text{poly} \log n)$
    - Conjecture: Can be improved to $\mathcal{O}_k(\log n)$
  - Dynamic version of Baker's technique: dynamic approximation scheme for DOMINATING SET on planar graphs?
  - Dynamic k-DISJOINT PATHS on planar graphs?

Thank you!