Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG
Chair for
INFORMATICS I
Efficient Algorithms and
Knowledge-Based Systems
Institute for Informatics

Advanced Algorithms

Winter term 2019/20

Lecture 13. Linear Time Planarity Testing via PQ-trees

(based on slides of Ignaz Rutter)

Steven Chaplick

Chair for Computer Science I

An Efficient Certifying Planarity Test

Thm
Given an n-vertex m-edge graph $G = (V, E)$, testing whether G is planar can be done in $O(n + m)$ time.

When G is planar, a planar embedding can be produced in the same time. Otherwise, a K_5 or $K_{3,3}$ minor can be found in the same time.

An Efficient Certifying Planarity Test

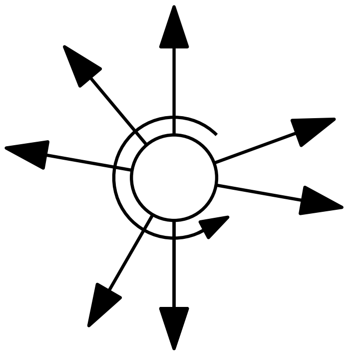
Thm
Given an n-vertex m-edge graph $G = (V, E)$, testing whether G is planar can be done in $O(n + m)$ time.

When G is planar, a planar embedding can be produced in the same time. Otherwise, a K_5 or $K_{3,3}$ minor can be found in the same time.

we will skip the details of the certifying part

Edge Ordering and Embeddings

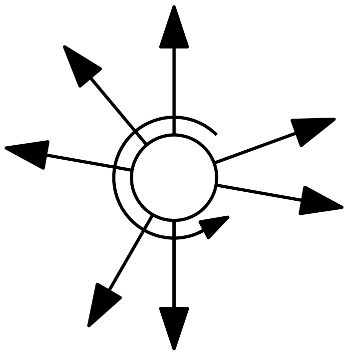
Embeddings are encoded as an edge ordering for each vertex.



$\rightsquigarrow$ Rotation-System

Edge Ordering and Embeddings

Embeddings are encoded as an edge ordering for each vertex.

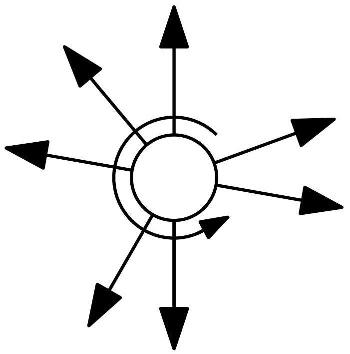


⇝ Rotation-System

Does a given rotation system descibe a planar embedding?

Edge Ordering and Embeddings

Embeddings are encoded as an edge ordering for each vertex.



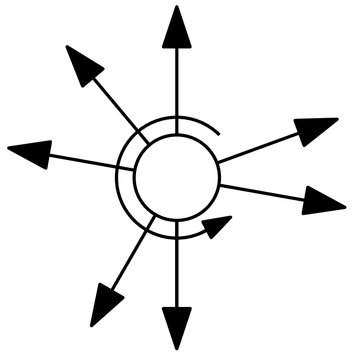
⇝ Rotation-System

Does a given rotation system descibe a planar embedding?

How can you test this?

Edge Ordering and Embeddings

Embeddings are encoded as an edge ordering for each vertex.



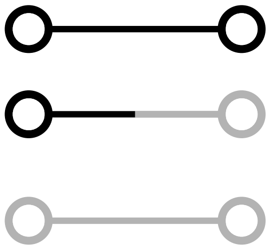
⇝ Rotation-System

Does a given rotation system descibe a planar embedding?

How can you test this?
(hint: Euler's formula)

Planarity Testing, 1st idea

Idea: Iteratively add nodes $\rightsquigarrow$ three types of edges:



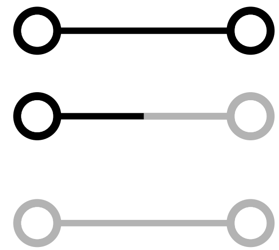
- embedded
- half-embedded
- absent

Objective: Save all possible partial embeddings (i.e., positions of the embedded and half-embedded edges).

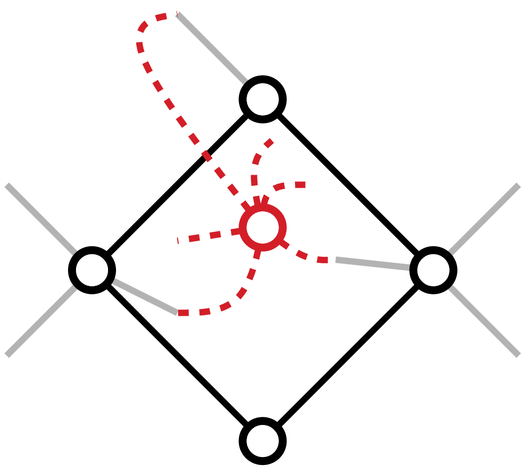
Planarity Testing, 1st idea

Idea: Iteratively add nodes $\rightsquigarrow$ three types of edges:

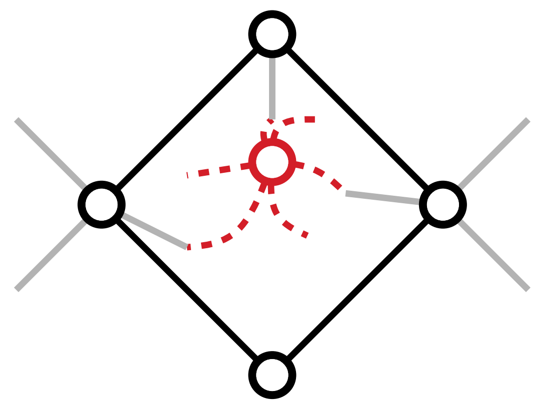
- embedded
- half-embedded
- absent



Objective: Save all possible partial embeddings (i.e., positions of the embedded and half-embedded edges).



No planar full embedding possible

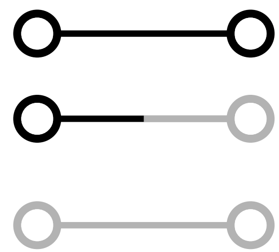


Full embedding possible.

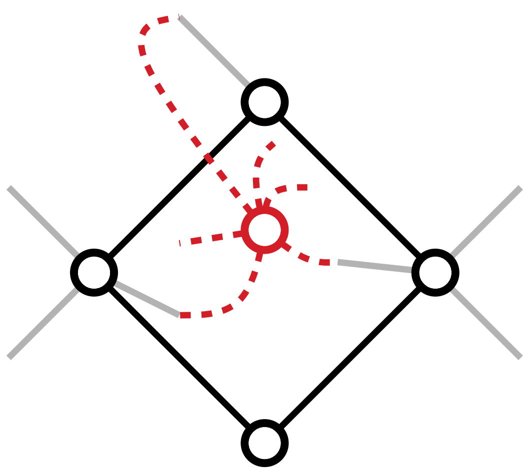
Planarity Testing, 1st idea

Idea: Iteratively add nodes $\rightsquigarrow$ three types of edges:

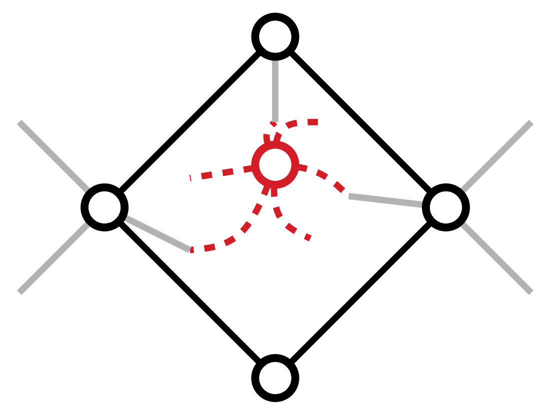
- embedded
- half-embedded
- absent



Objective: Save all possible partial embeddings (i.e., positions of the embedded and half-embedded edges).



No planar full embedding possible



Full embedding possible.

$\rightsquigarrow$ Exponential runtime & space 

Planarity Testing, refined

Idea to reduce options for insertions:

Force insertions always on the outerface.

Is this possible?

What do we get from it?

Planarity Testing, refined

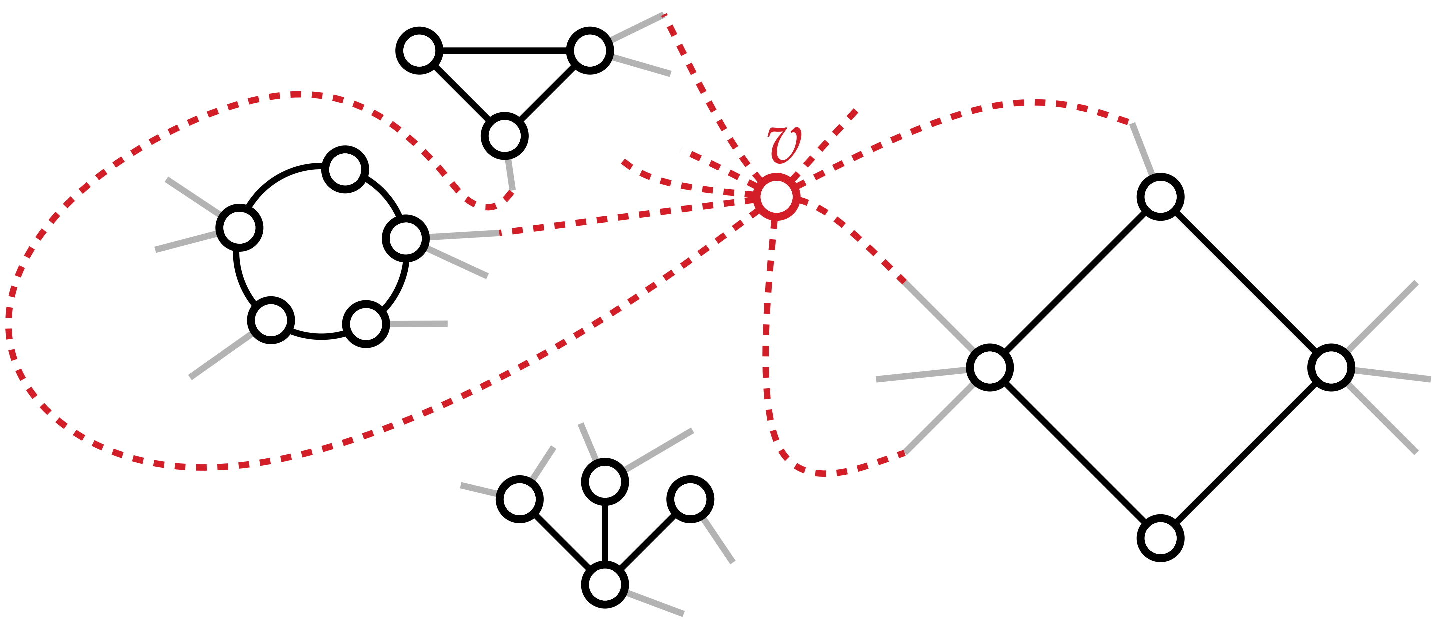
Idea to reduce options for insertions:

Force insertions always on the outerface.

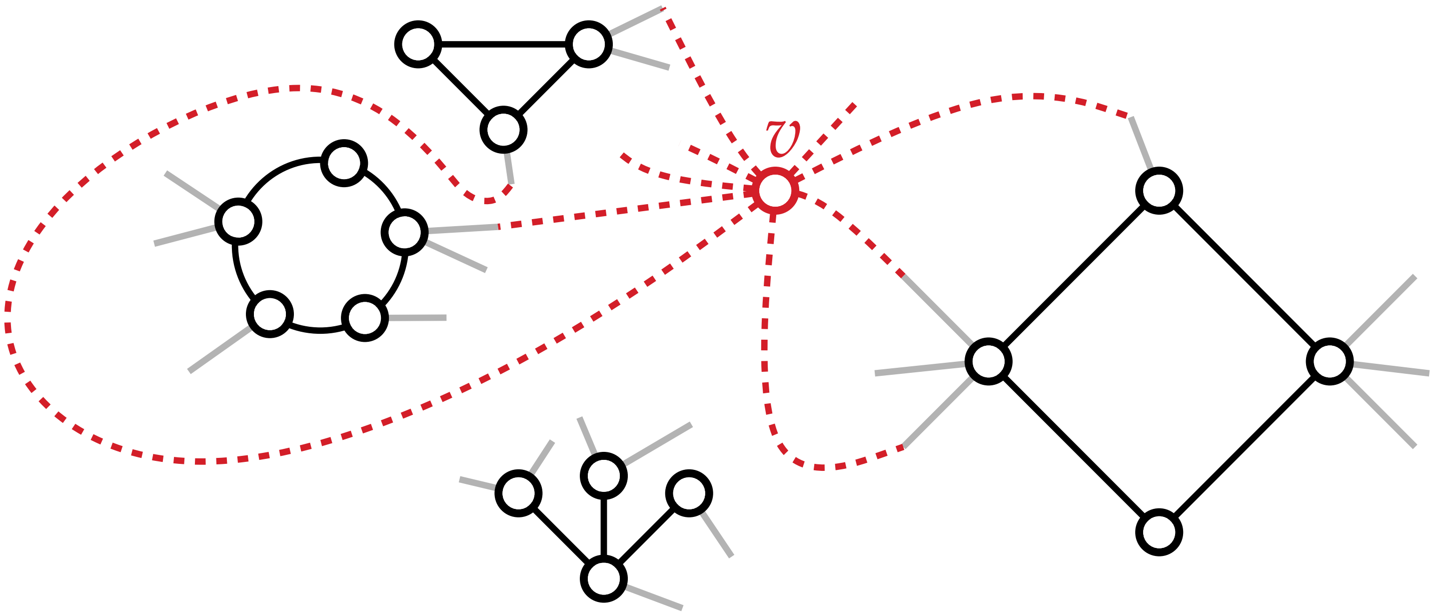
Is this possible?

What do we get from it?

Process vertices bottom-up via a DFS spanning tree.
All halfedges must be embedded on the same (outer) face.

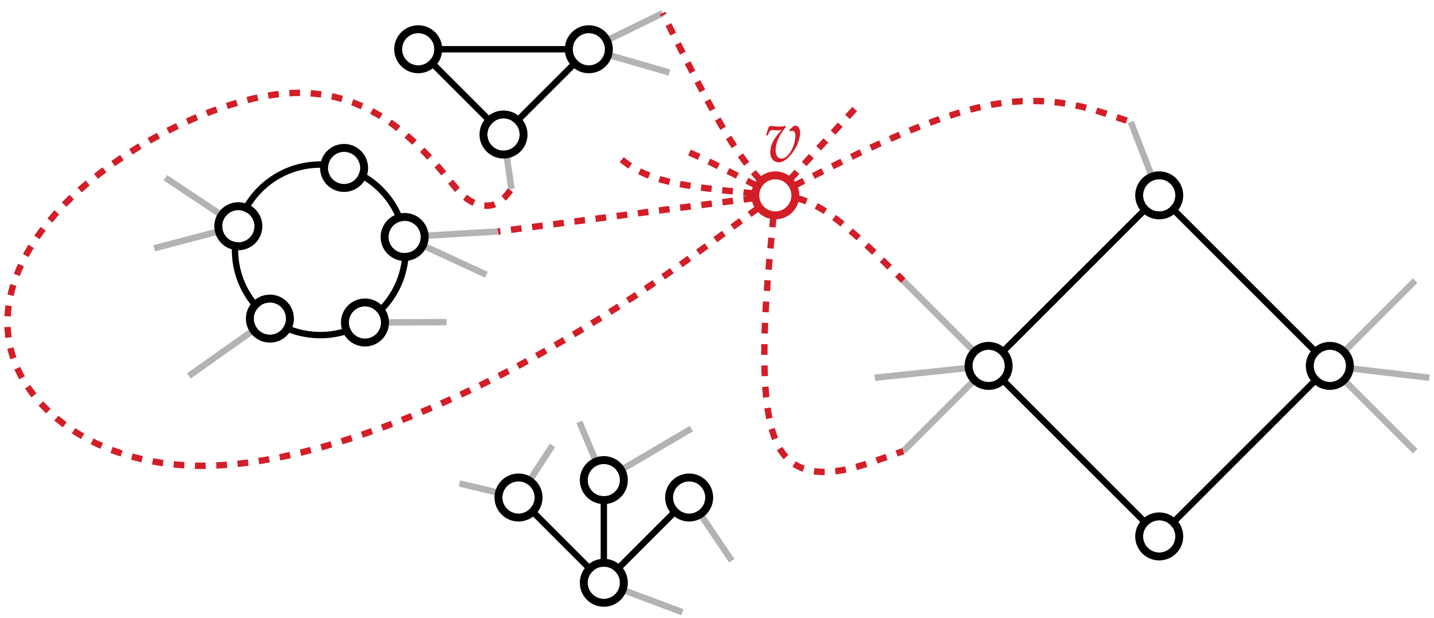


Planarity Testing, Vertex Insertion



1. **Restriction:** Half-embedded edges incident to v that belong to the same component must be consecutive.
2. **Combination:** Components and half-embedded edges hanging from v can be ordered arbitrarily.

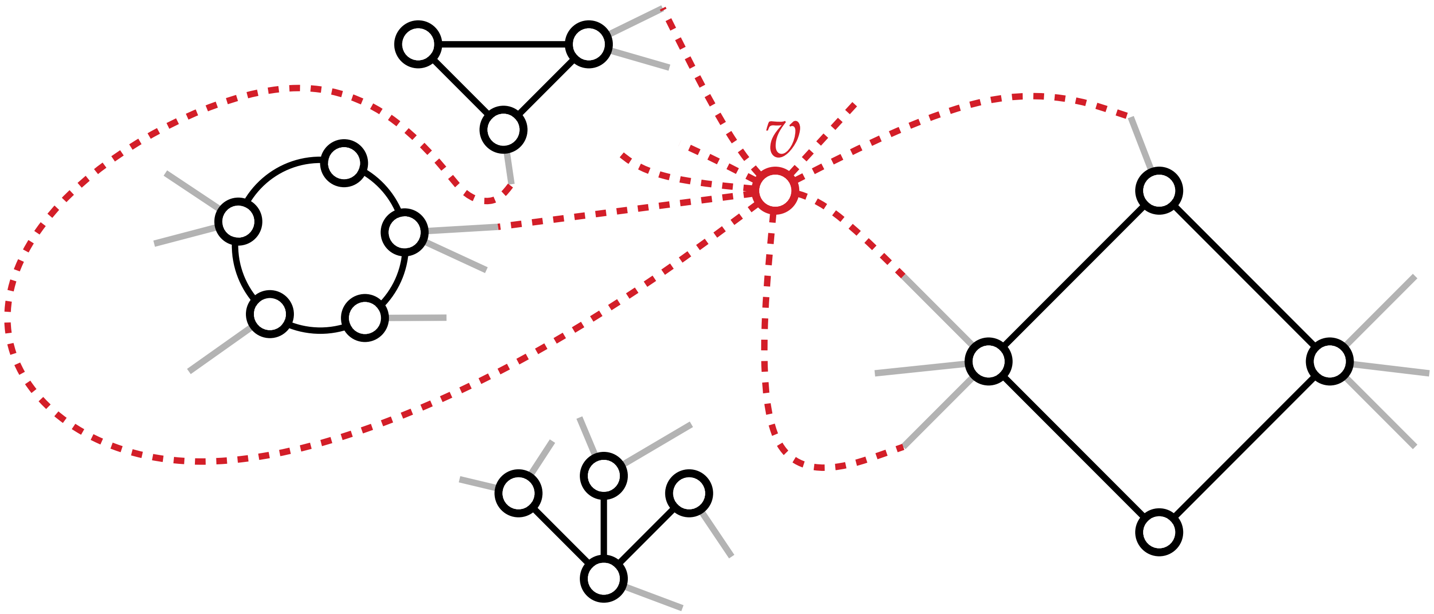
Planarity Testing, Vertex Insertion



1. **Restriction:** Half-embedded edges incident to v that belong to the same component must be consecutive.
2. **Combination:** Components and half-embedded edges hanging from v can be ordered arbitrarily.

Still too many options to store all of them.

Planarity Testing, Vertex Insertion



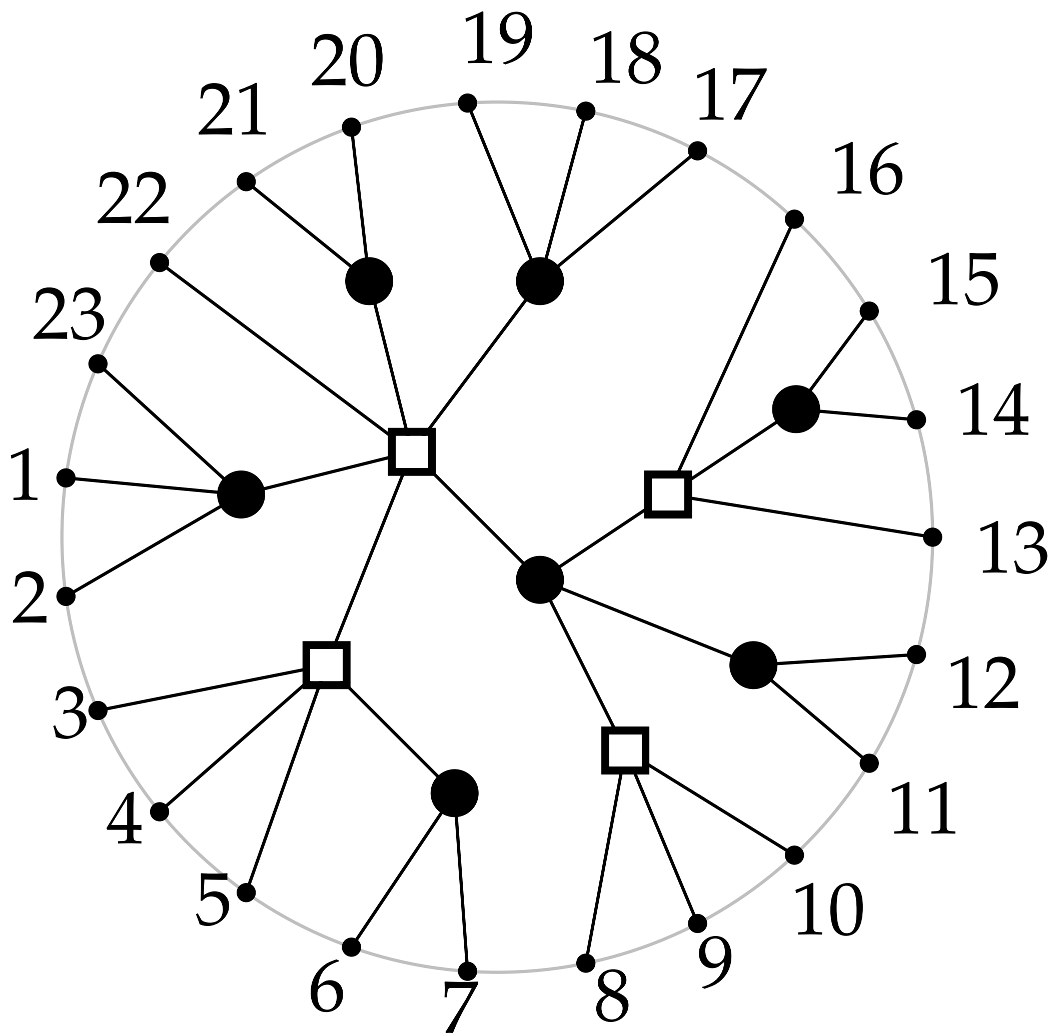
1. **Restriction:** Half-embedded edges incident to v that belong to the same component must be consecutive.
2. **Combination:** Components and half-embedded edges hanging from v can be ordered arbitrarily.

Still too many options to store all of them.

Solution: data structure to compactly represent such constrained orders.

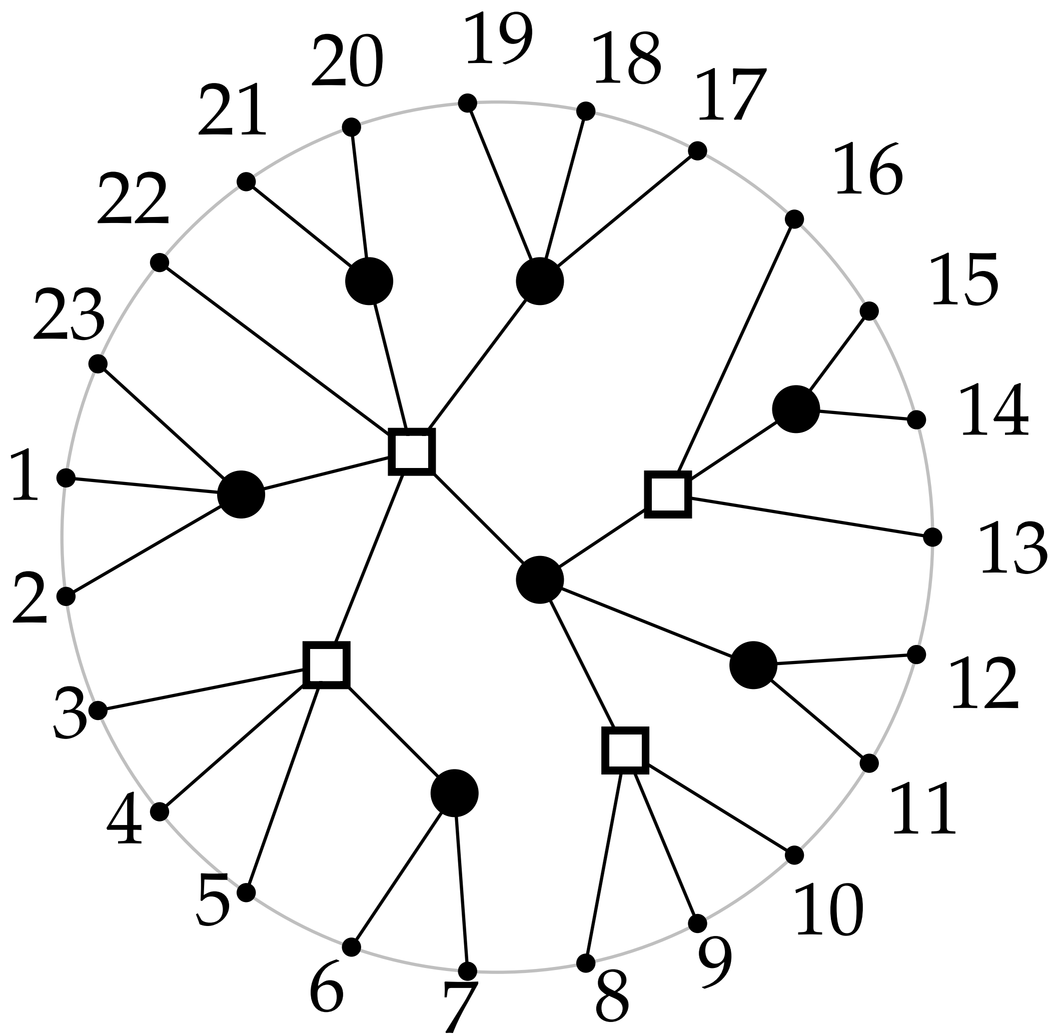
PQ-Tree

PQ-Tree



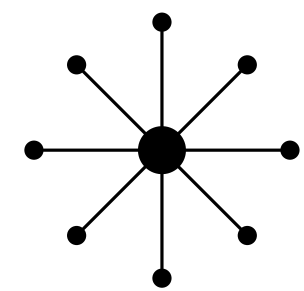
PQ-tree represents all circular orderings of its leaves by these operations

PQ-Tree



● P-node
permute children freely

□ Q-node
only "flipping" allowed

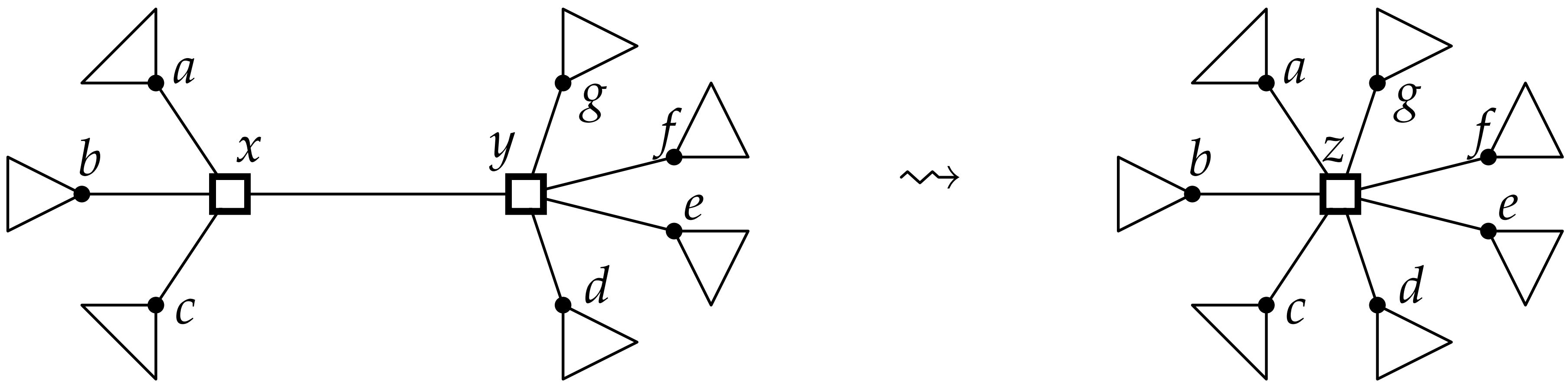


Single P-node ⇝ all possible circular orderings of its leaves.

PQ-tree represents all circular orderings of its leaves by these operations

Order-Preserving Contraction, Null-Tree

Order-preserving contraction

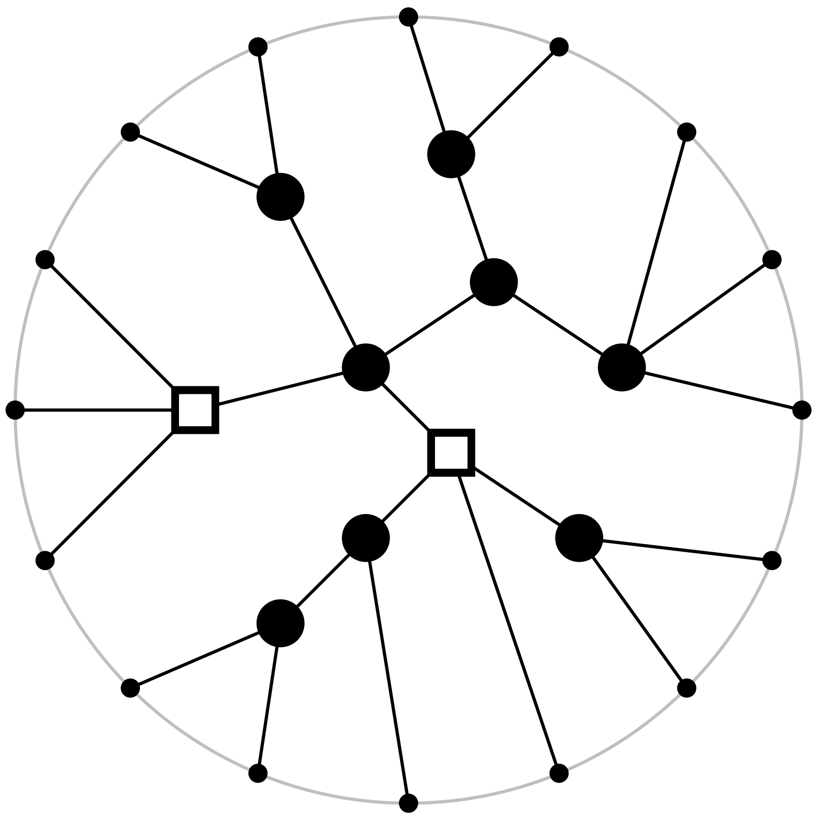


NOTE this restricts the representable orderings!

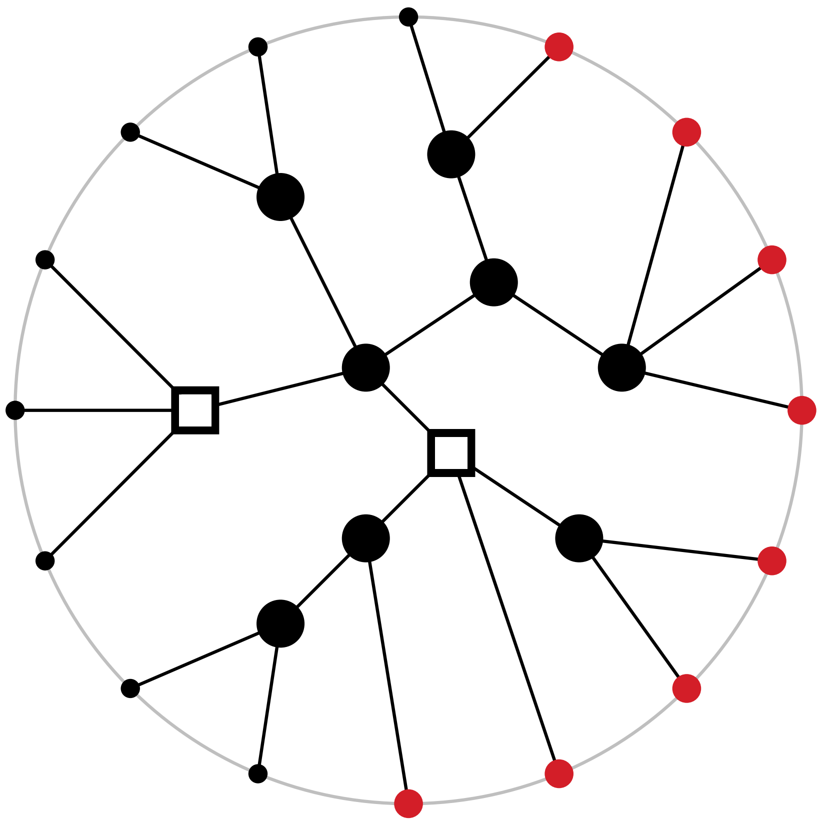
Null-tree: represents the empty set of permutations

NOTE Null-tree$\neq$ empty tree (represents permutations of the empty set)

Consecutivity of Subsets



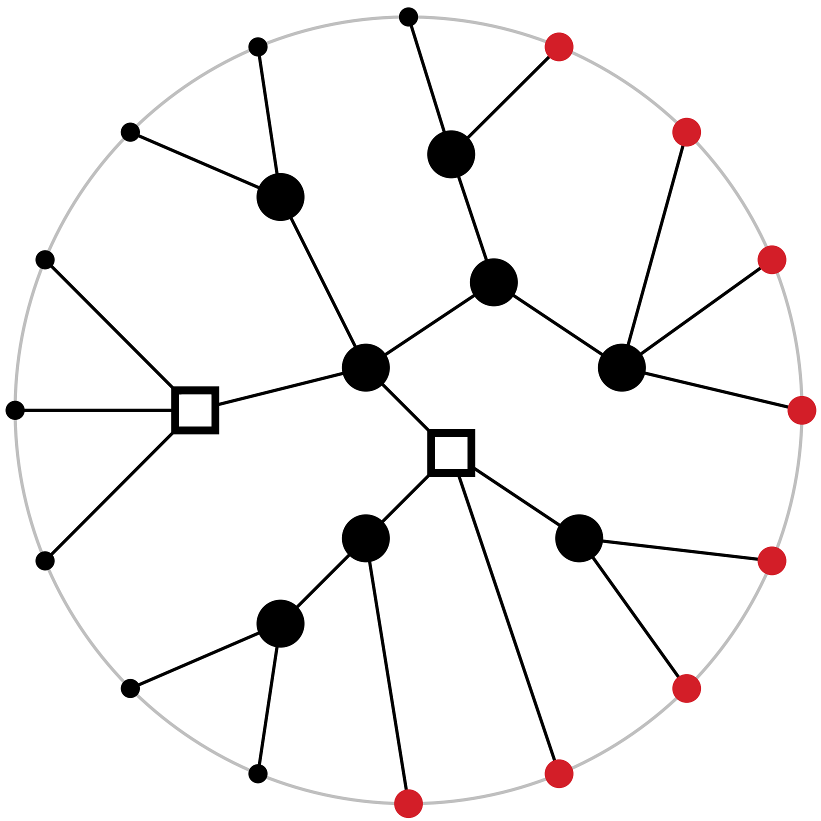
Consecutivity of Subsets



Find new PQ-tree, representing exactly the orderings which :

- are admitted by the current tree and
- have the red leaves consecutive

Consecutivity of Subsets



Find new PQ-tree, representing exactly the orderings which :

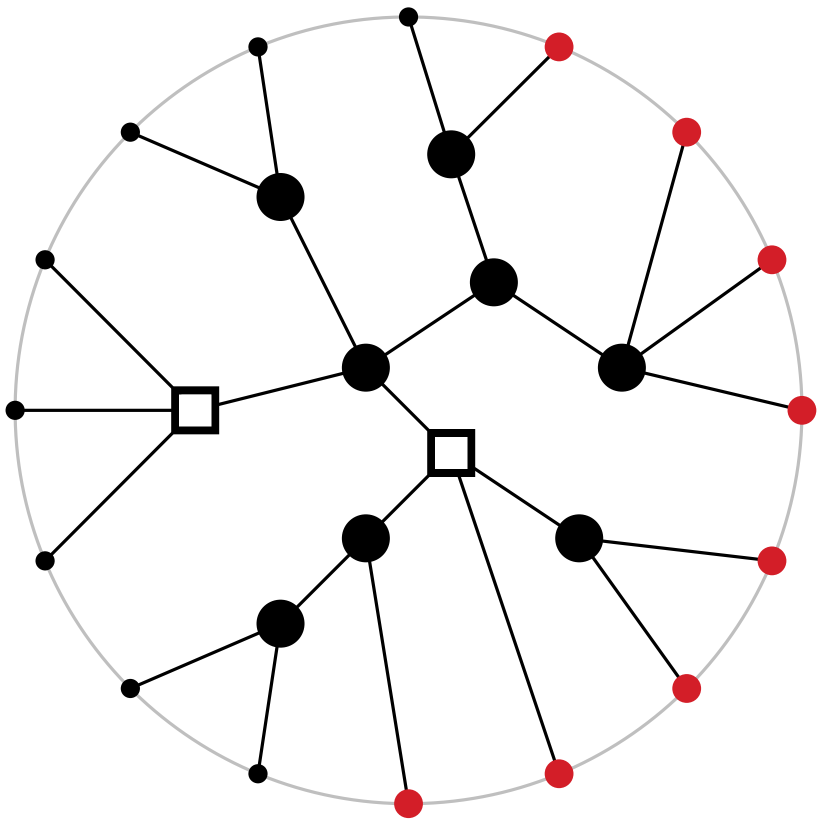
- are admitted by the current tree and
- have the red leaves consecutive

No rearrangment with the red leaves consecutive

$\Rightarrow$

The result is a Null-tree

Consecutivity of Subsets



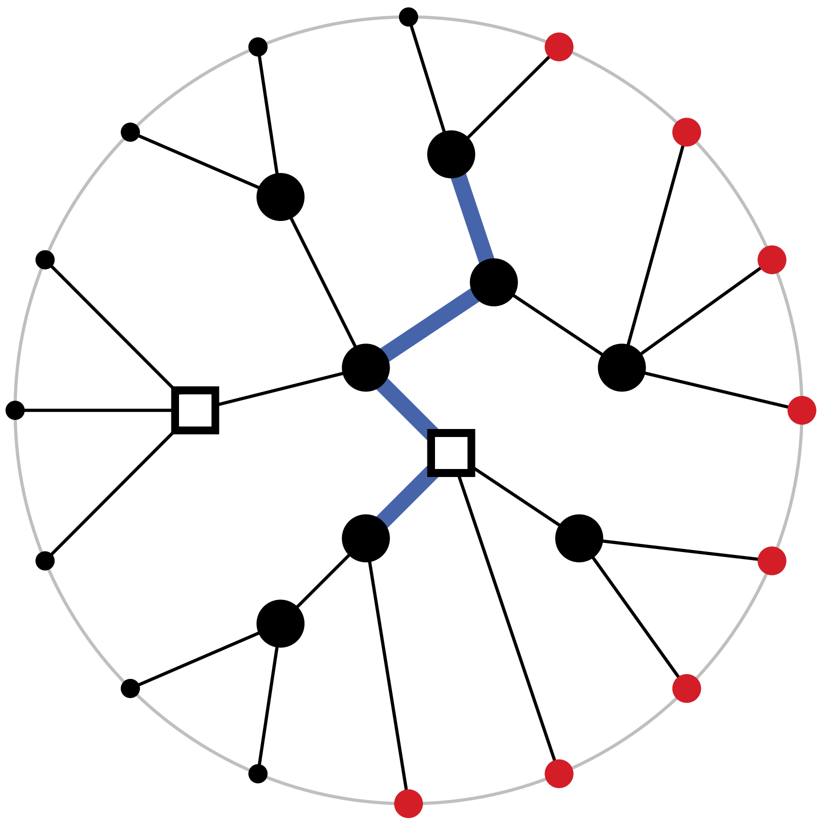
Find new PQ-tree, representing exactly
the orderings which :

- are admitted by the current tree and
- have the red leaves consecutive

No rearrangment with the
red leaves consecutive
$\Rightarrow$
The result is a Null-tree

An edge is **partial** when both subtrees have both red and black leaves $\rightsquigarrow$ partial edges allow forbidden orderings

Consecutivity of Subsets



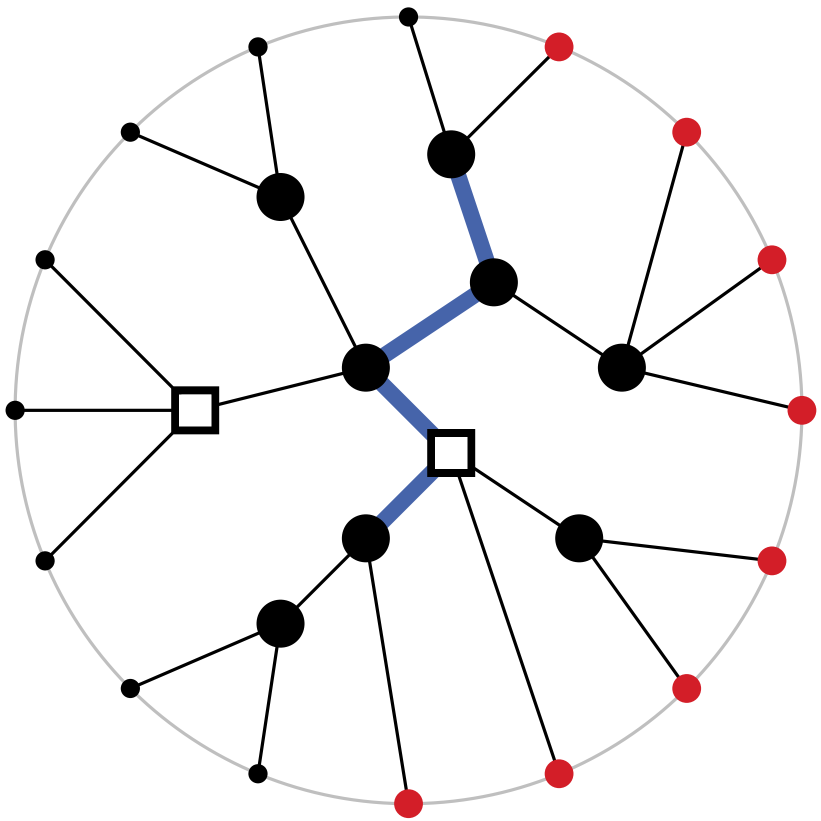
Find new PQ-tree, representing exactly the orderings which :

- are admitted by the current tree and
- have the red leaves consecutive

No rearrangment with the red leaves consecutive
$\Rightarrow$
The result is a Null-tree

An edge is **partial** when both subtrees have both red and black leaves $\rightsquigarrow$ partial edges allow forbidden orderings

Consecutivity of Subsets



Find new PQ-tree, representing exactly the orderings which :

- are admitted by the current tree and
- have the red leaves consecutive

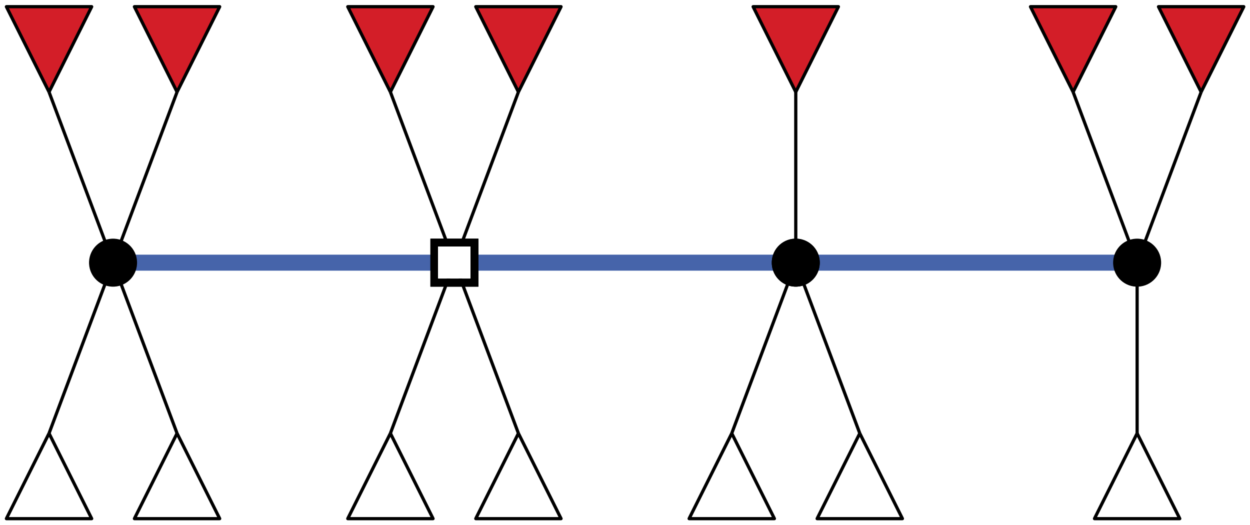
No rearrangment with the red leaves consecutive
$\Rightarrow$
The result is a Null-tree

An edge is **partial** when both subtrees have both red and black leaves $\rightsquigarrow$ partial edges allow forbidden orderings

Lemma: If an arrangement of the PQ-tree has the red leaves consecutive, the partial edges form a path.

Updating PQ-Trees

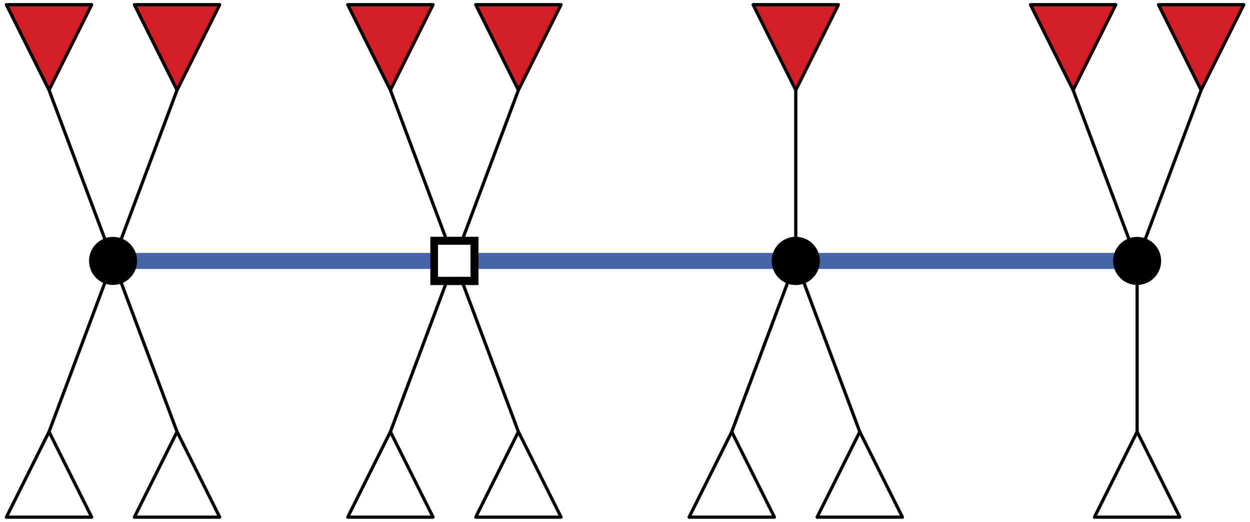
1. Find the path of partial edges and arrange the tree to split red and black leaves.



Updating PQ-Trees

1. Find the path of partial edges and arrange the tree to split red and black leaves.

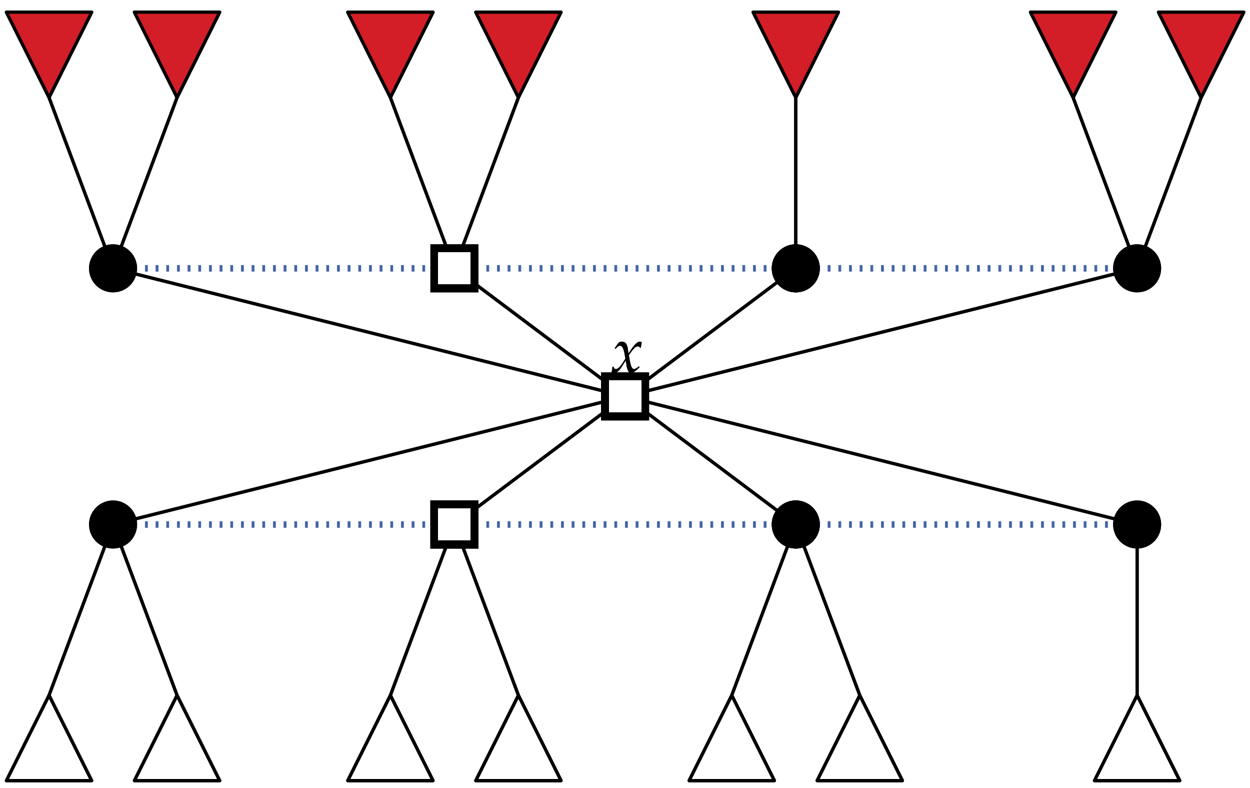
2. Split partial path (*terminal path*), insert a new Q-node x



Updating PQ-Trees

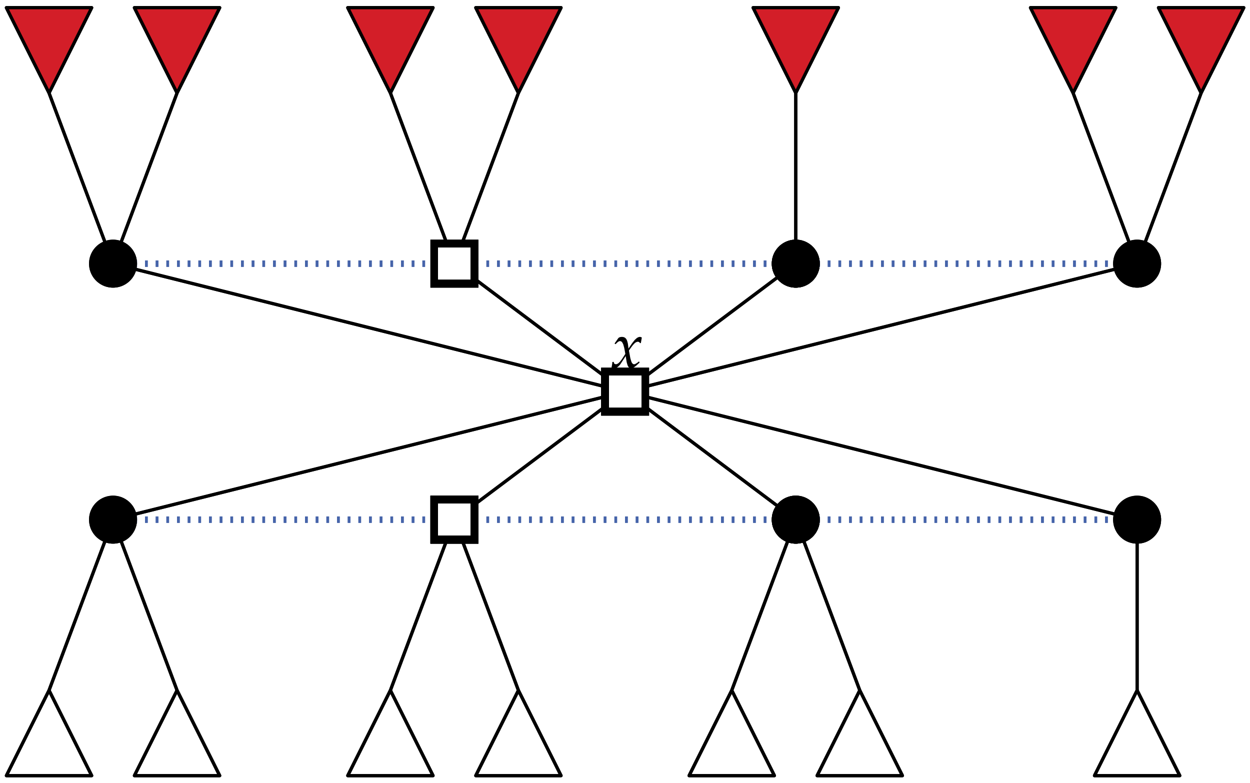
1. Find the path of partial edges and arrange the tree to split red and black leaves.

2. Split partial path (*terminal path*), insert a new Q-node x



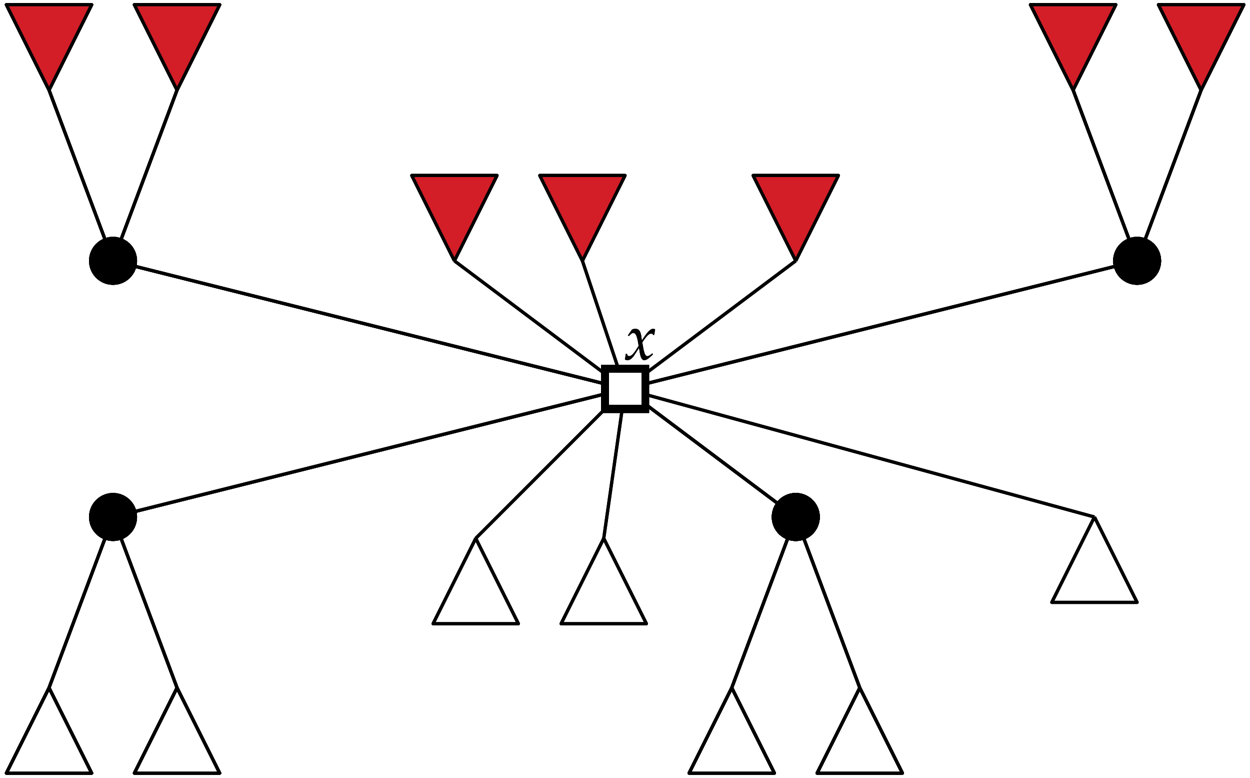
Updating PQ-Trees

1. Find the path of partial edges and arrange the tree to split red and black leaves.
2. Split partial path (*terminal path*), insert a new Q-node x
3. Contract edges from x to other Q-nodes, dissolve inner nodes of degree 2.



Updating PQ-Trees

1. Find the path of partial edges and arrange the tree to split red and black leaves.
2. Split partial path (*terminal path*), insert a new Q-node x
3. Contract edges from x to other Q-nodes, dissolve inner nodes of degree 2.



Correctness

Thm: The update algorithm is correct.

Proof:

- For any ordering represented before with the red leaves consecutive, the operation is the same and succeeds.

Correctness

Thm: The update algorithm is correct.

Proof:

- For any ordering represented before with the red leaves consecutive, the operation is the same and succeeds.
- Each ordering of the resulting PQ-tree is represented by the original.

Correctness

Thm: The update algorithm is correct.

Proof:

- For any ordering represented before with the red leaves consecutive, the operation is the same and succeeds.
- Each ordering of the resulting PQ-tree is represented by the original.

■

Correctness

Thm: The update algorithm is correct.

Proof:

- For any ordering represented before with the red leaves consecutive, the operation is the same and succeeds.
- Each ordering of the resulting PQ-tree is represented by the original. ■

Obs.: Update can be implemented in polynomial time.

Correctness

Thm: The update algorithm is correct.

Proof:

- For any ordering represented before with the red leaves consecutive, the operation is the same and succeeds.
- Each ordering of the resulting PQ-tree is represented by the original. ∎

Obs.: Update can be implemented in polynomial time.

What runtime can we easily guarantee?

Correctness

Thm: The update algorithm is correct.

Proof:

- For any ordering represented before with the red leaves consecutive, the operation is the same and succeeds.
- Each ordering of the resulting PQ-tree is represented by the original.

■

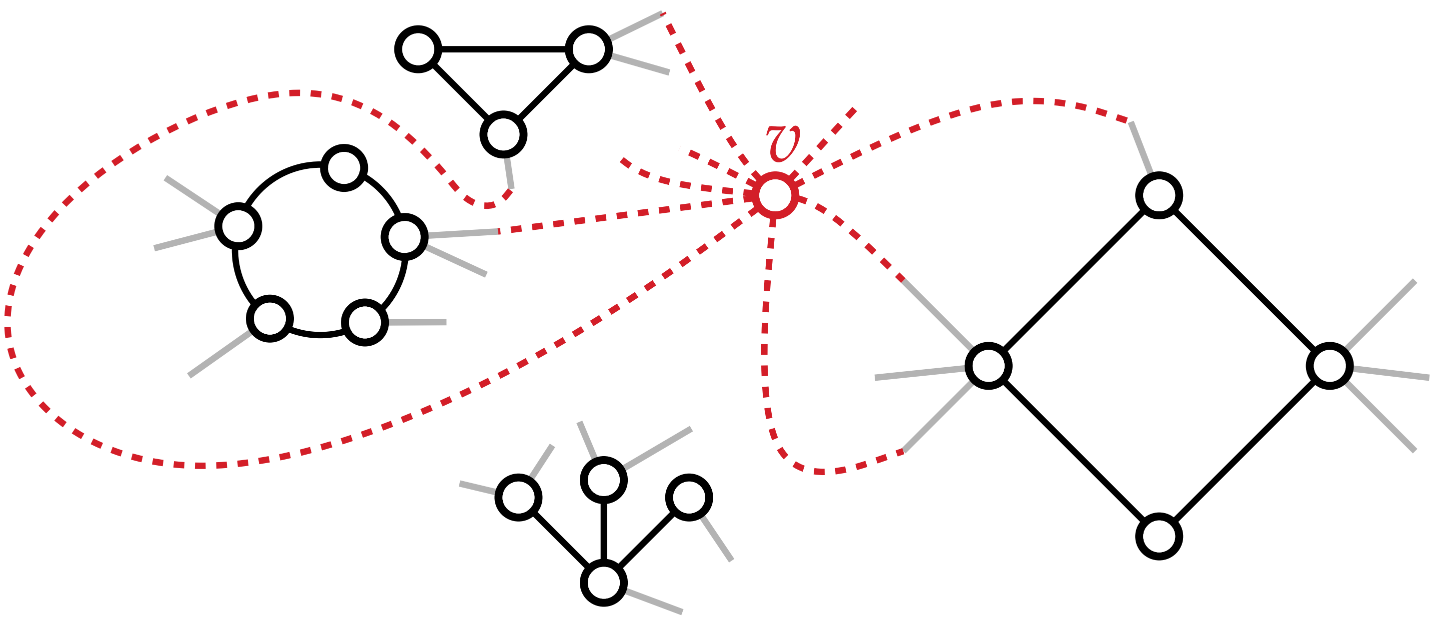
Obs.: Update can be implemented in polynomial time.

What runtime can we easily guarantee?

A linear time implementation needs more ideas . . .

Back to Planarity Testing

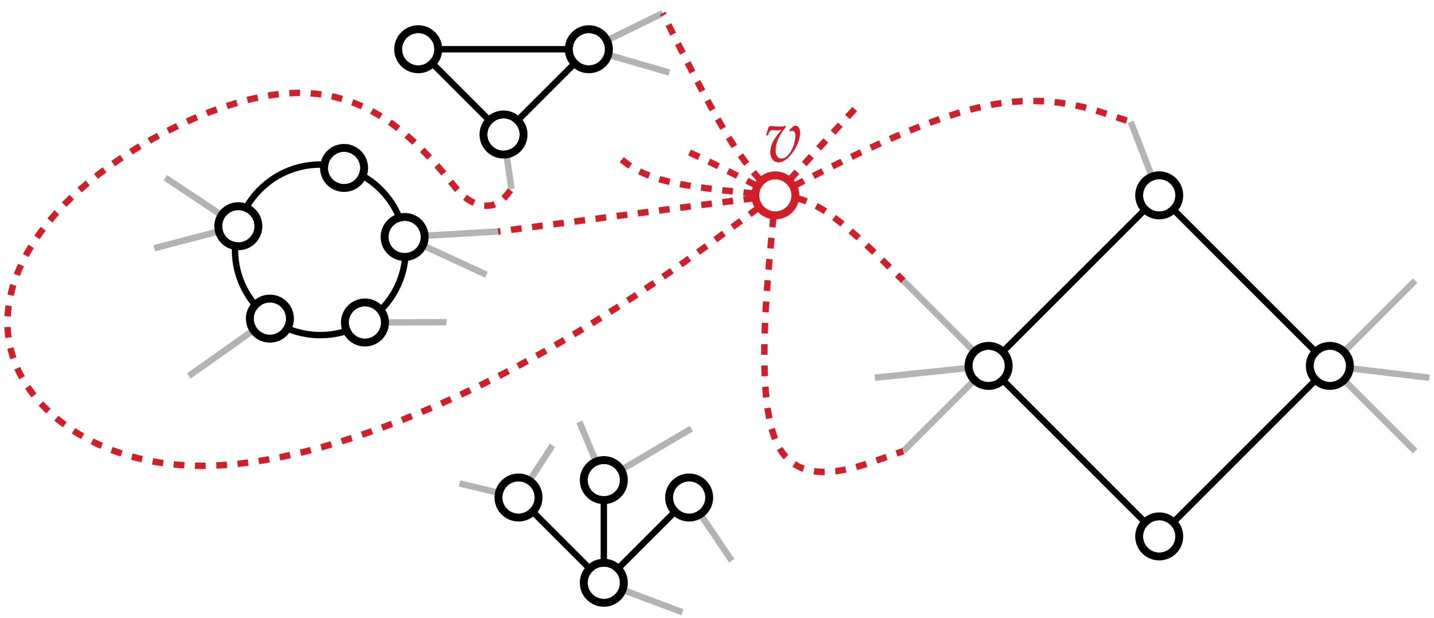
Planarity Testing, vertex insertion



1. **Restriction:** Half-embedded edges incident to v that belong to the same component must be consecutive.
2. **Combination:** Components and half-embedded edges hanging from v can be ordered arbitrarily.

Explicitly representing all options is too expensive.

Planarity Testing, vertex insertion

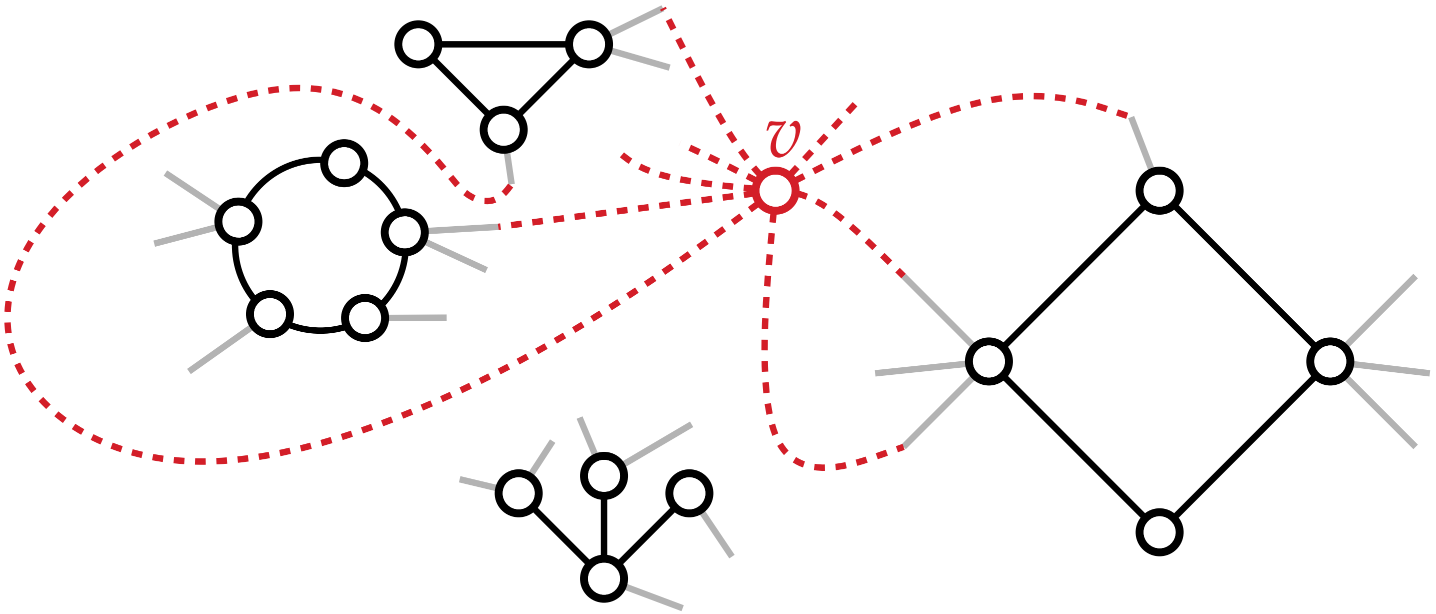


1. **Restriction:** Half-embedded edges incident to v that belong to the same component must be consecutive.
2. **Combination:** Components and half-embedded edges hanging from v can be ordered arbitrarily.

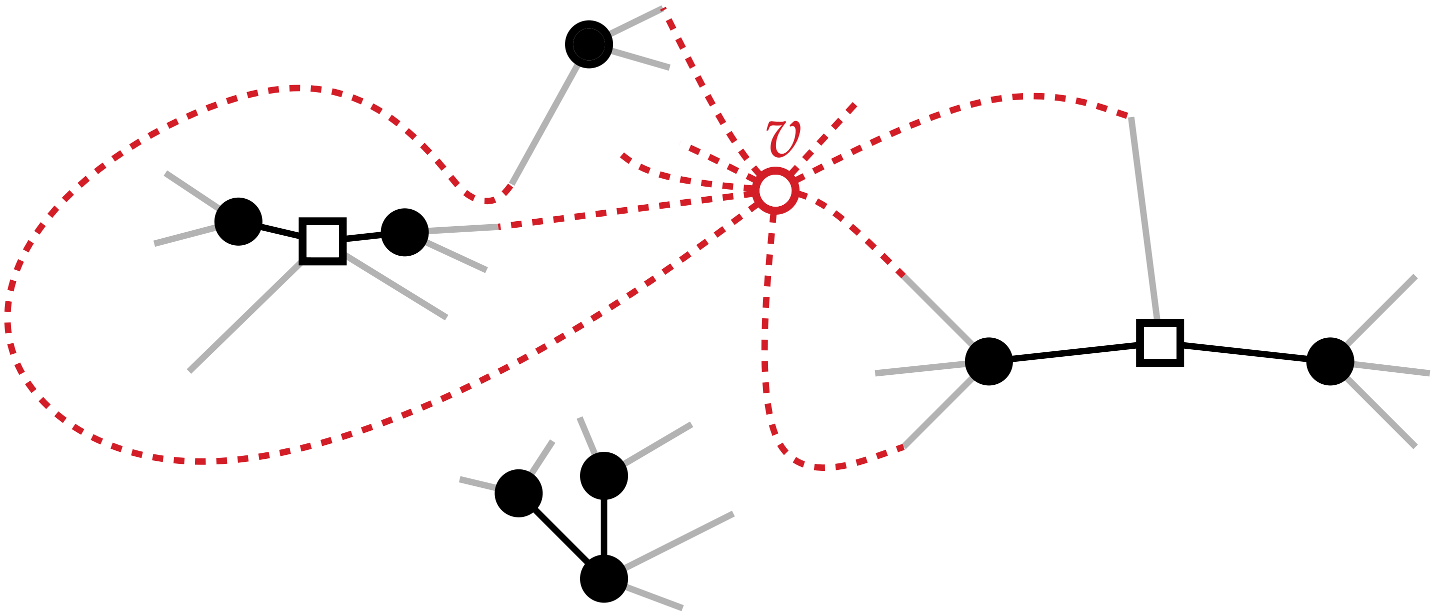
Explicitly representing all options is too expensive.

⇝ use PQ-trees!

Planarity Testing, vertex ins. (PQ-trees)

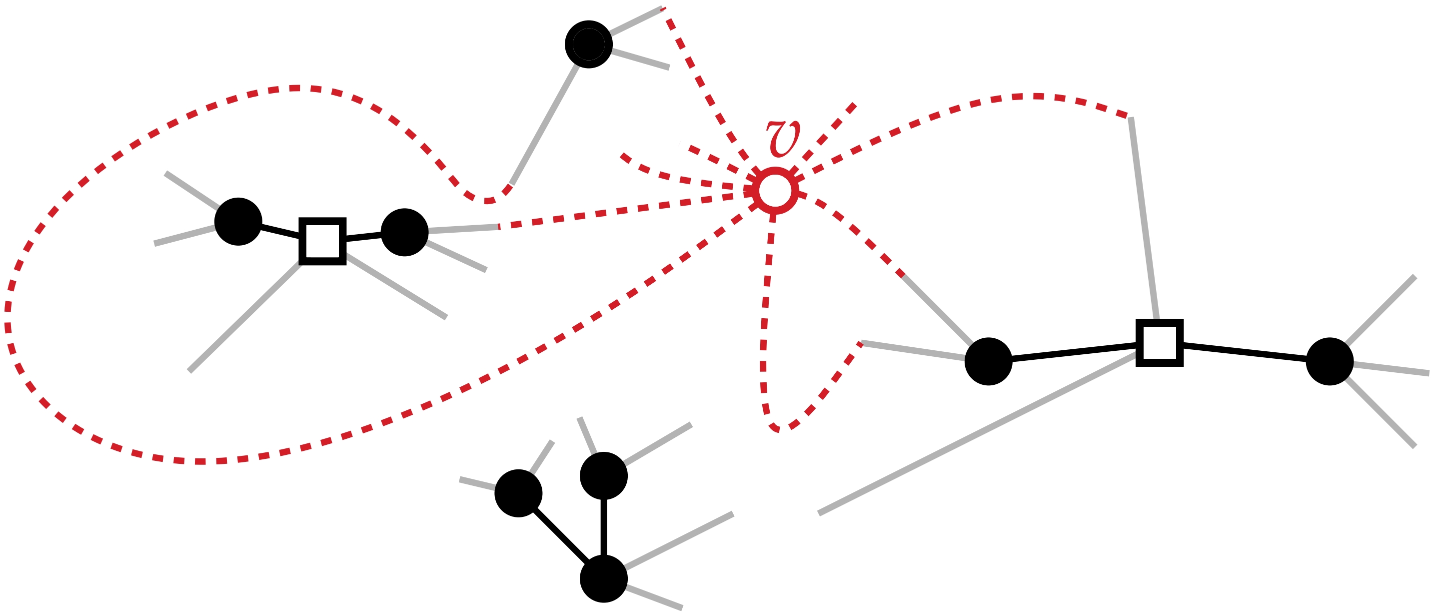


Planarity Testing, vertex ins. (PQ-trees)



One PQ-tree for each component

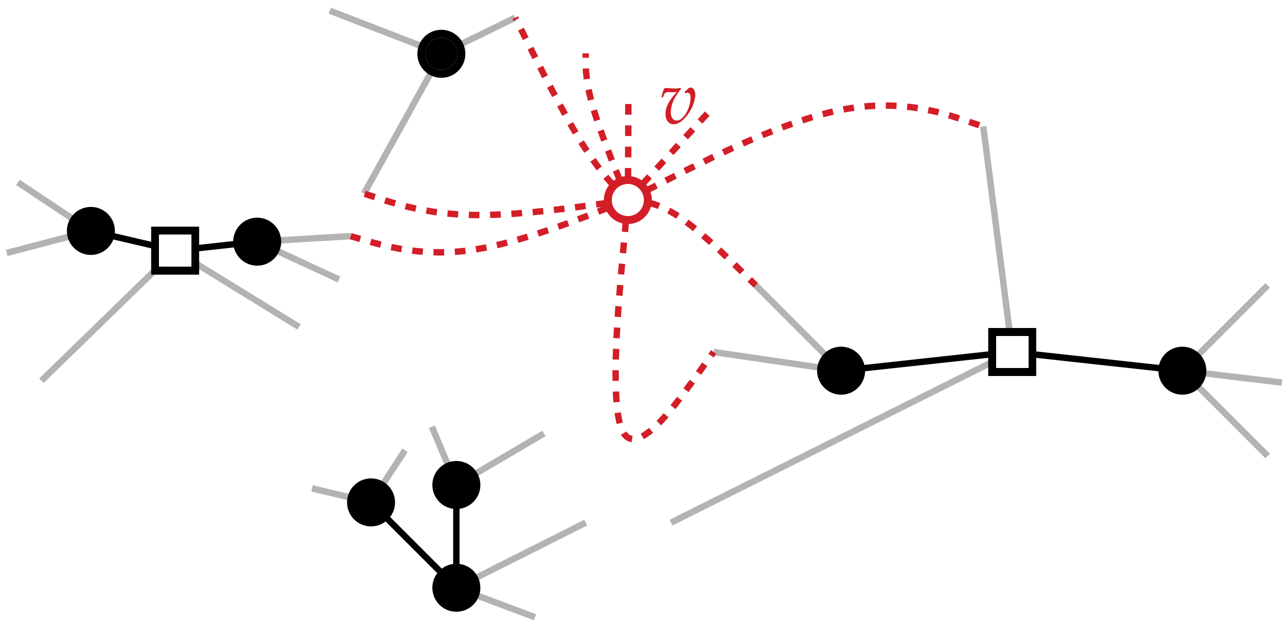
Planarity Testing, vertex ins. (PQ-trees)



One PQ-tree for each component

- **Restriction:** for each component C, make Cv edges consecutive in C's PQ-tree

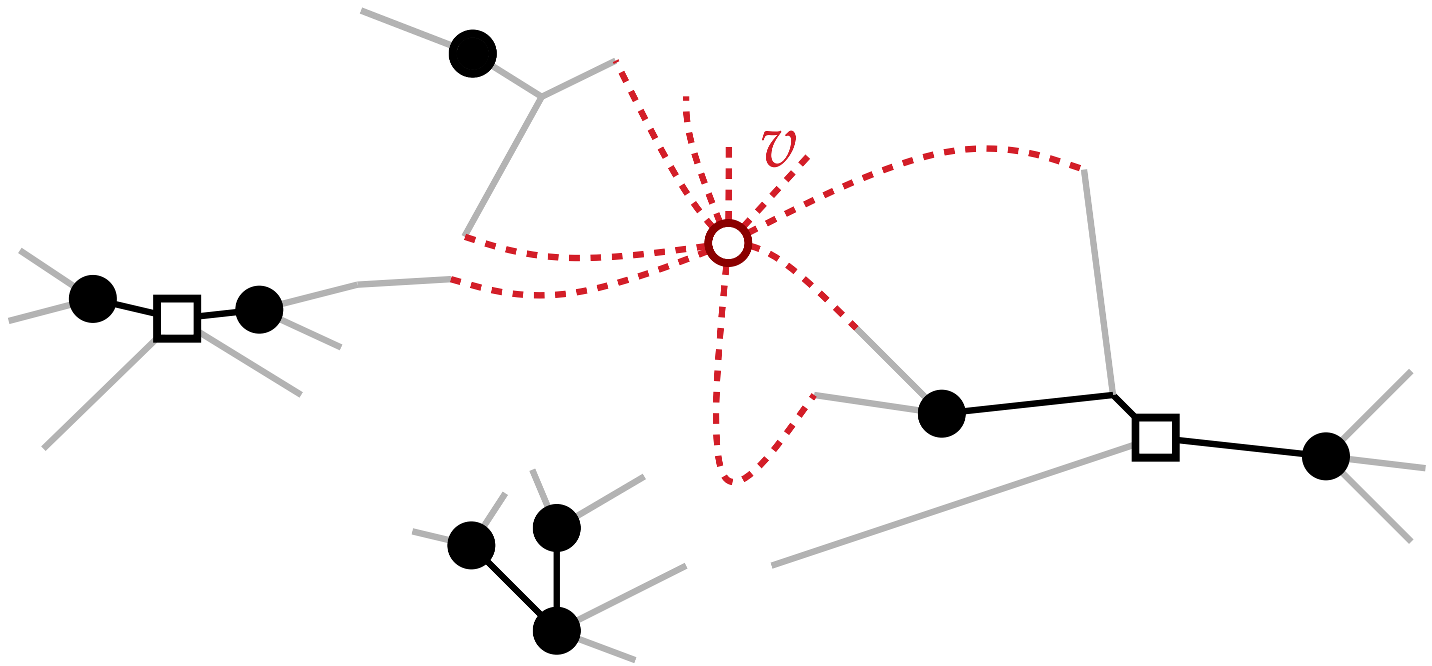
Planarity Testing, vertex ins. (PQ-trees)



One PQ-tree for each component

- **Restriction:** for each component C, make Cv edges consecutive in C's PQ-tree

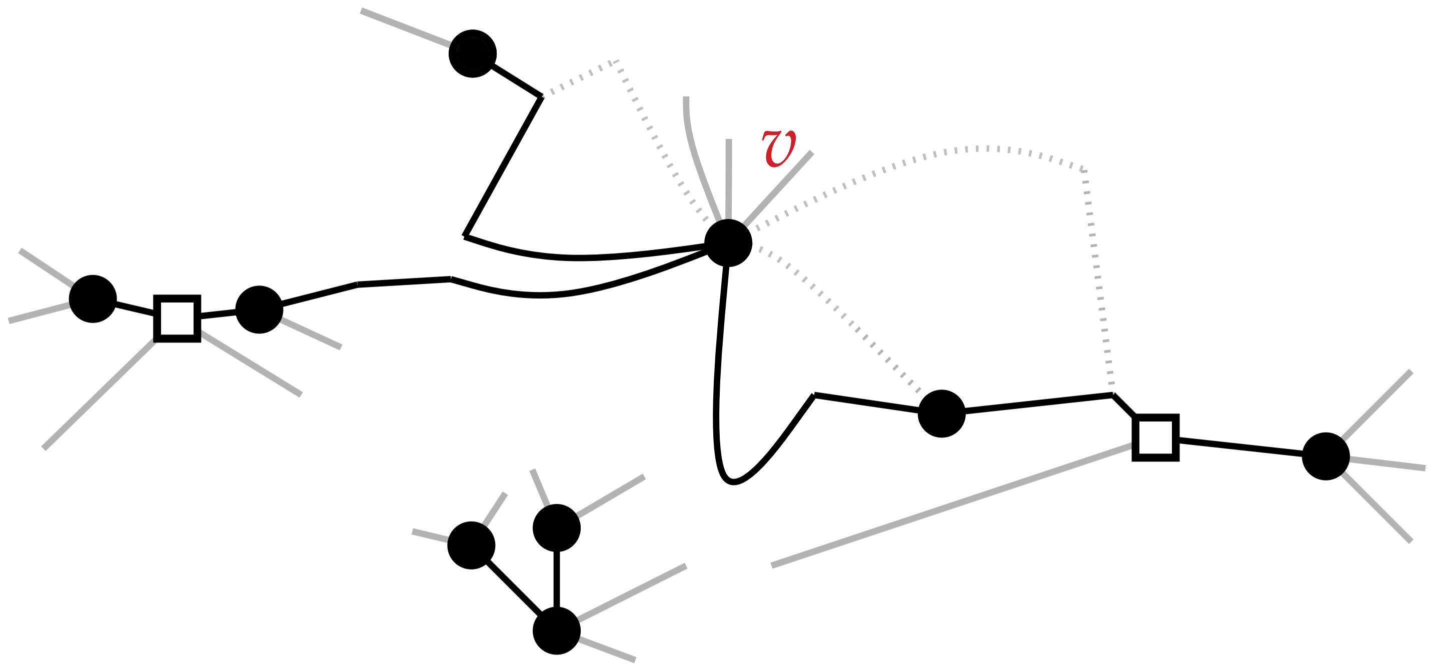
Planarity Testing, vertex ins. (PQ-trees)



One PQ-tree for each component

- **Restriction:** for each component C, make Cv edges consecutive in C's PQ-tree
- **Combination:** merge PQ-trees of v's components into one, and hang new half-embedded edges.
  $\rightsquigarrow$ single PQ-tree for the resulting component

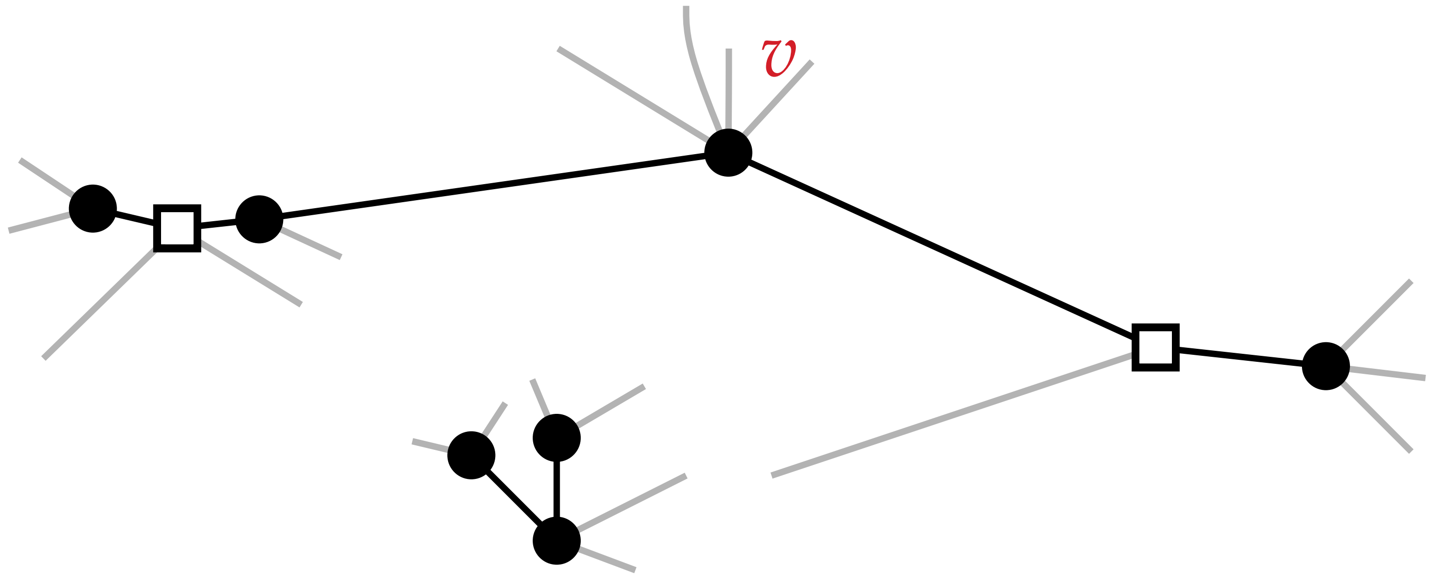
Planarity Testing, vertex ins. (PQ-trees)



One PQ-tree for each component

- **Restriction:** for each component C, make Cv edges consecutive in C's PQ-tree
- **Combination:** merge PQ-trees of v's components into one, and hang new half-embedded edges.
  $\rightsquigarrow$ single PQ-tree for the resulting component

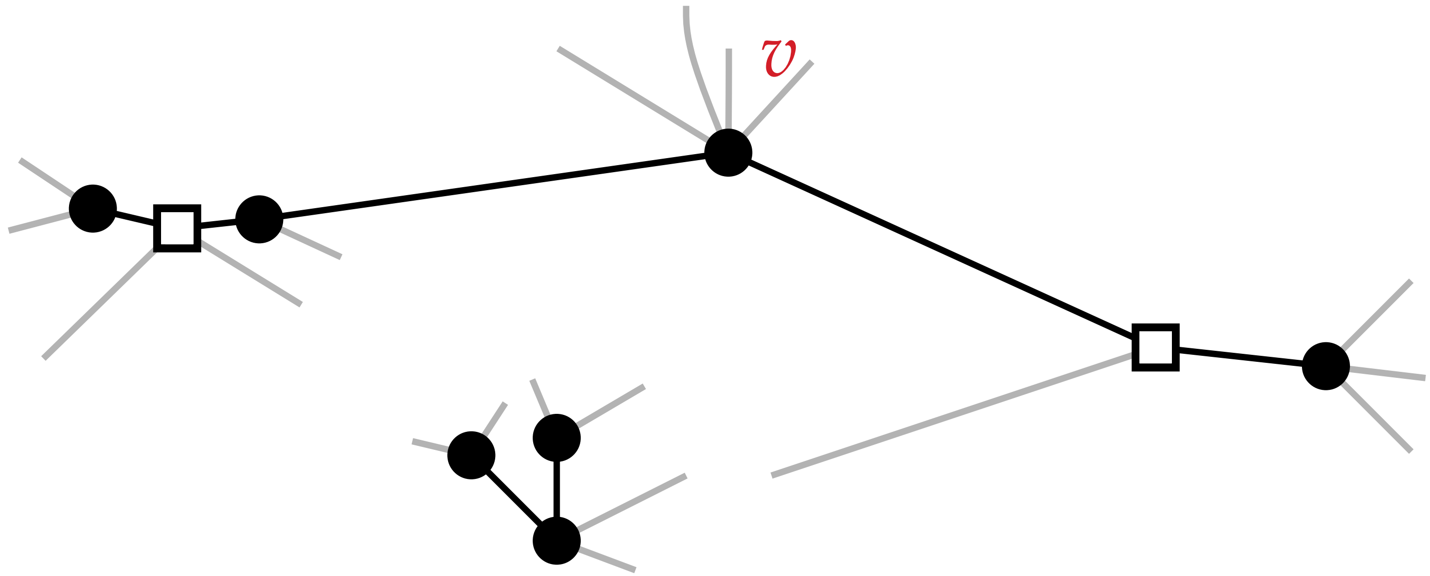
Planarity Testing, vertex ins. (PQ-trees)



One PQ-tree for each component

- **Restriction:** for each component C, make Cv edges consecutive in C's PQ-tree
- **Combination:** merge PQ-trees of v's components into one, and hang new half-embedded edges.
  $\rightsquigarrow$ single PQ-tree for the resulting component

Planarity Testing, vertex ins. (PQ-trees)



One PQ-tree for each component

- **Restriction:** for each component C, make Cv edges consecutive in C's PQ-tree
- **Combination:** merge PQ-trees of v's components into one, and hang new half-embedded edges.
  $\rightsquigarrow$ single PQ-tree for the resulting component

Cost per vertex: one consecutivity constraint $O(\deg v)$

Planarity Testing

Graph is planar if and only if all reduction steps succeed.

Embedding can be recovered by undoing steps.
Select/expand orders within the PQ-tree.

Planarity Testing

Graph is planar if and only if all reduction steps succeed.

Embedding can be recovered by undoing steps.
Select/expand orders within the PQ-tree.

Warning on linear runtime: finding a terminal path efficiently is tricky, involves rooting each PQ-tree. When merging, need to root consistently.

Planarity Testing

Graph is planar if and only if all reduction steps succeed.

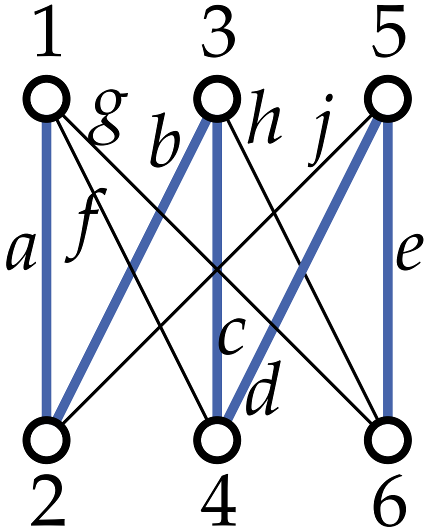
Embedding can be recovered by undoing steps.
Select/expand orders within the PQ-tree.

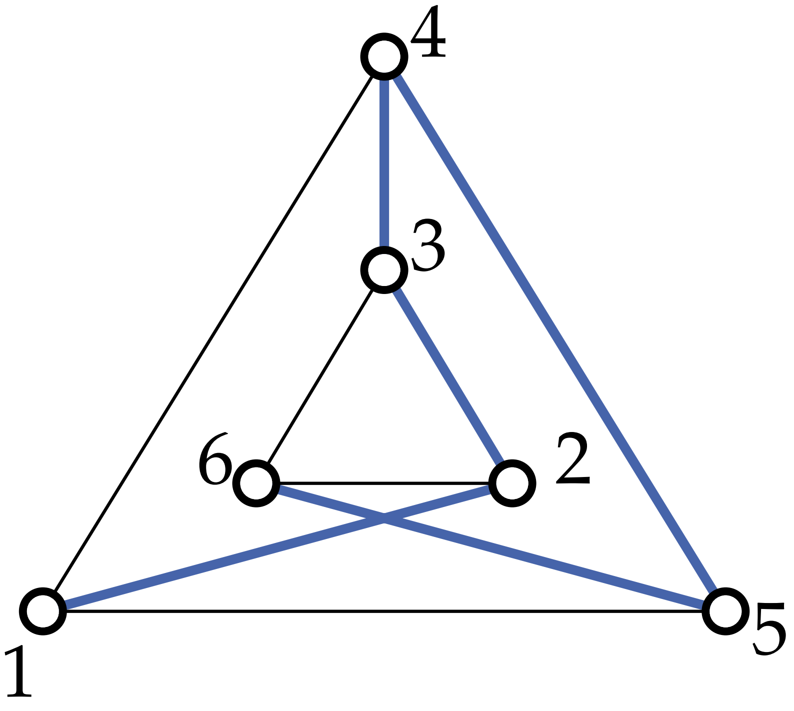
Warning on linear runtime: finding a terminal path efficiently is tricky, involves rooting each PQ-tree. When merging, need to root consistently.

What about the sequence to process vertices?

- (s, t)-ordering (two-connected graphs) (see *graph visualization* lecture) [Lempel, Even, Cederbaum '67]
- Depth-first search [Shih, Hsu '99, Boyer, Myrvold '04]

Examples

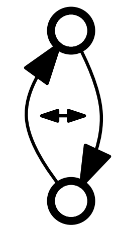




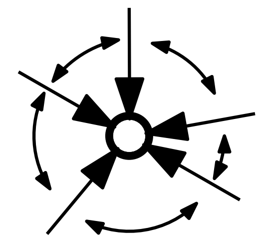
Linear-time PQ-Tree construction

Linear-time PQ-tree construction

Linear time implementation details



- choose root
- store each edge in both directions
- store incoming edges at each node in double-linked list



- mark each edge regarding orientation to root
- P-nodes have pointer to parent

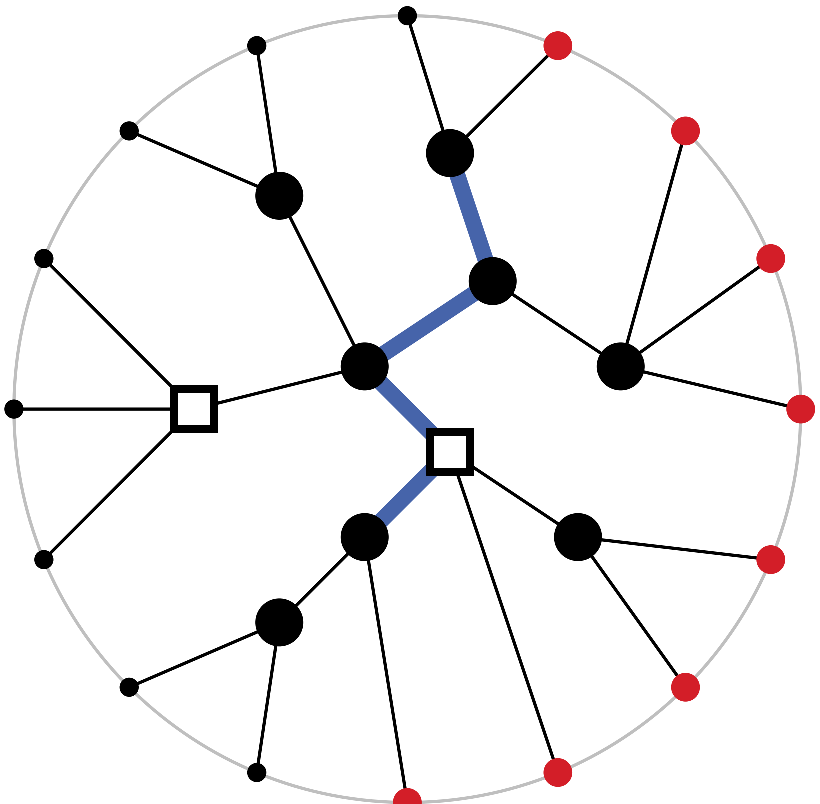
NOTE: Parent of a Q-node is “expensive” to determine, but this means we do not need to keep track of it when editing Q-nodes.

Computing the Terminal Path

length of terminal path

number of consecutive elements

Lemma: Terminal path can be found in $O(p + k)$ time.



Computing the Terminal Path

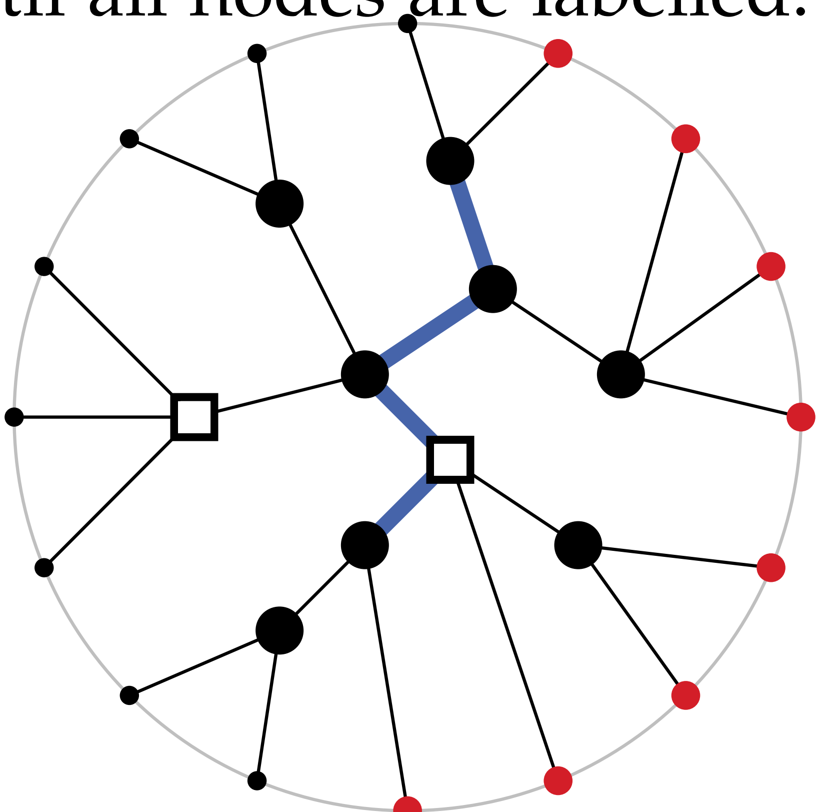
length of terminal path — number of consecutive elements

Lemma: Terminal path can be found in $O(p + k)$ time.

Classify the nodes of the PQ-tree:

- a leaf is full when it is a red element
- an inner node is full when all but one neighbor is full.
- a non-full node is partial at least one neighbor is full.

$\rightsquigarrow$ $O(k)$ time until all nodes are labelled.



Computing the Terminal Path

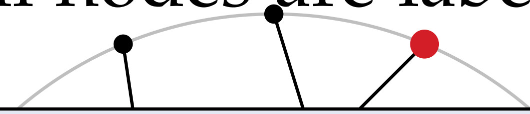
length of terminal path — number of consecutive elements

Lemma: Terminal path can be found in $O(p + k)$ time.

Classify the nodes of the PQ-tree:

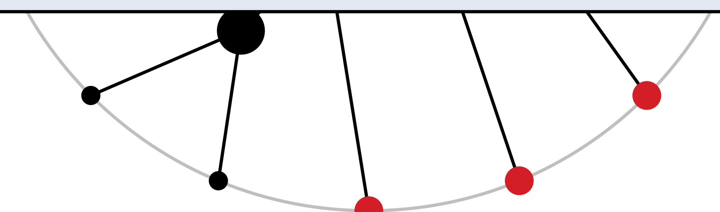
- a leaf is full when it is a red element
- an inner node is full when all but one neighbor is full.
- a non-full node is partial at least one neighbor is full.

$\rightsquigarrow$ $O(k)$ time until all nodes are labelled.



Key idea: Partial nodes must belong to the terminal path

- extend potential path from each partial node to parent
- stop extending when another path is hit, and join
- leads to a tree with at most one degree 3 node (o.w. reject)
- highest node found which is either partial or meeting of two extensions, is the high point of the terminal path.



Computing the Terminal Path

length of terminal path → p; number of consecutive elements → k

Lemma: Terminal path can be found in $O(p + k)$ time.

Classify the nodes of the PQ-tree:

- a leaf is full when it is a red element
- an inner node is full when all but one neighbor is full.
- a non-full node is partial at least one neighbor is full.

$\rightsquigarrow$ $O(k)$ time until all nodes are labelled.

Key idea: Partial nodes must belong to the terminal path

- extend potential path from each partial node to parent
- stop extending when another path is hit, and join
- leads to a tree with at most one degree 3 node (o.w. reject)
- highest node found which is either partial or meeting of two extensions, is the high point of the terminal path.

How do we find the parent nodes quickly?

Computing the Terminal Path

length of terminal path → p; number of consecutive elements → k

Lemma: Terminal path can be found in $O(p + k)$ time.

Classify the nodes of the PQ-tree:

- a leaf is full when it is a red element
- an inner node is full when all but one neighbor is full.
- a non-full node is partial at least one neighbor is full.

$\rightsquigarrow O(k)$ time until all nodes are labelled.

Key idea: Partial nodes must belong to the terminal path

- extend potential path from each partial node to parent
- stop extending when another path is hit, and join
- leads to a tree with at most one degree 3 node (o.w. reject)
- highest node found which is either partial or meeting of two extensions, is the high point of the terminal path.

How do we find the parent nodes quickly?

where is the parent-edge with respect to full child-edges?

Update Step

Split terminal path, new nodes will receive full neighbors.

Single Split, for each node of the terminal path:

- detach full neighbors F, and delete incident edges to neighbors on the path — $O(1)$
- make a copy, hang F from it. — $O(\#\text{full neighbors})$

$O(p+k)$

Create central Q-node, $O(p)$ time

Each contraction, $O(1)$ time

Update Step

Split terminal path, new nodes will receive full neighbors.

Single Split, for each node of the terminal path:

- detach full neighbors F, and delete incident edges to neighbors on the path — $O(1)$
- make a copy, hang F from it. — $O(\#\text{full neighbors})$

$$\overline{\overline{}}$$

$O(p+k)$

Create central Q-node, $O(p)$ time
Each contraction, $O(1)$ time

NOTE if terminal path contains $q \geq 2$ Q-nodes, then number of Q-nodes decreases by $q-1$, i.e., cost of processing this terminal path saves $q-1$ for us for later.

Runtime analysis

X ground set
$\mathcal{U} = \{U_1, \ldots, U_\ell\}$ collection of subsets of X.

Thm: PQ-tree representing all orderings where $U_1, \ldots, U_\ell$ are consecutive can be computed in $O(|X| + |U_1| + \cdots + |U_\ell|)$ time (amortized analysis).

Proof: Consider potential function
$\phi(\mathcal{U}, i) = 2u_i + |Q_i| + \sum_{x \in P_i}(\deg(x) - 1)$, where:

- $u_i = \sum_{j>i} |U_j|$
- Q_i = Q-nodes in PQ-tree T_i after processing $U_1, \ldots, U_i$.
- P_i = P-nodes in T_i

$\phi(\mathcal{U}, 0) = \Theta(|X| + \sum_i |U_i|)$ (budget to be used)
Inductively show budget $\phi(\mathcal{U}, i-1) -$ cost$(U_i) \geq \phi(\mathcal{U}, i)$
need: ϕ stay to stay ≥ 0
$\leadsto$ claimed runtime.