Parallel Algorithms for Trees

Ahmad Khayyat
Department of Electrical & Computer Engineering
ahmad.khayyat@ece.queensu.ca

Parallel Computation: Models and Methods
Selim G. Akl — Chapter 6

CISC 879 — Algorithms and Applications
Queen's University

October 29, 2008

Outline

Outline

1. Introduction
   - Graphs
   - Euler Tours
2. Building Euler Tours
3. Applications of Euler Tours
4. Conclusion

Graphs

- A Graph $G = (V, E)$ is a set of vertices V connected by a set of edges E.
- If the edges have orientation, the graph is directed.
- A path is an ordered list of edges in the form $(v_i, v_j), (v_j, v_k), (v_k, v_l)$.
- A cycle is a path that begins and ends at the same vertex.
- The degree of a vertex in an undirected graph is the number of edges adjacent to the vertex.
- The in-degree of a vertex v in a directed graph is the number of edges entering v,
  The out-degree of a vertex v in a directed graph is the number of edges leaving v.

Outline

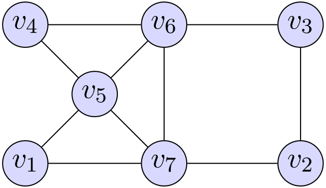
1. Introduction
   - Graphs
   - Euler Tours
2. Building Euler Tours
3. Applications of Euler Tours
4. Conclusion

Euler Tours

Euler Tour (ET)

A cycle where every edge of the graph appears in the cycle exactly once.

An undirected graph has an Euler tour if each vertex has an even degree.



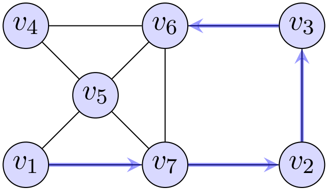
Euler Tours

Euler Tour (ET)

A cycle where every edge of the graph appears in the cycle exactly once.

An undirected graph has an Euler tour if each vertex has an even degree.

$(v_1, v_7), (v_7, v_2), (v_2, v_3), (v_3, v_6)$



Euler Tours

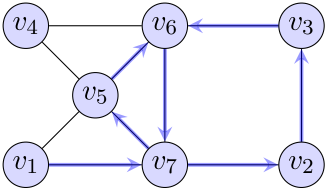
Euler Tour (ET)

A cycle where every edge of the graph appears in the cycle exactly once.

An undirected graph has an Euler tour if each vertex has an even degree.

$(v_1, v_7), (v_7, v_2), (v_2, v_3), (v_3, v_6)$

$(v_6, v_7), (v_7, v_5), (v_5, v_6)$



Euler Tours

Euler Tour (ET)

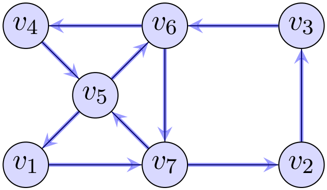
A cycle where every edge of the graph appears in the cycle exactly once.

An undirected graph has an Euler tour if each vertex has an even degree.

$(v_1, v_7), (v_7, v_2), (v_2, v_3), (v_3, v_6)$

$(v_6, v_7), (v_7, v_5), (v_5, v_6)$

$(v_6, v_4), (v_4, v_5), (v_5, v_1)$

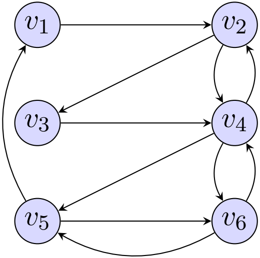


Euler Tours

Euler Tour (ET)

A cycle where every edge of the graph appears in the cycle exactly once.

A directed graph has an Euler tour if the in-degree of each vertex is equal to the out-degree of the vertex.



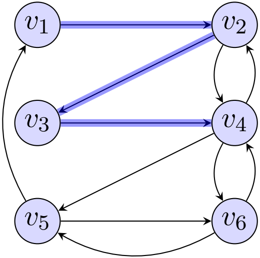
Euler Tours

Euler Tour (ET)

A cycle where every edge of the graph appears in the cycle exactly once.

A directed graph has an Euler tour if the in-degree of each vertex is equal to the out-degree of the vertex.

$(v_1, v_2), (v_2, v_3), (v_3, v_4)$



Euler Tours

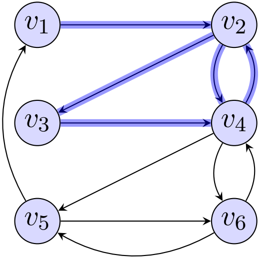
Euler Tour (ET)

A cycle where every edge of the graph appears in the cycle exactly once.

A directed graph has an Euler tour if the in-degree of each vertex is equal to the out-degree of the vertex.

$(v_1, v_2), (v_2, v_3), (v_3, v_4)$

$(v_4, v_2), (v_2, v_4)$



Euler Tours

Euler Tour (ET)

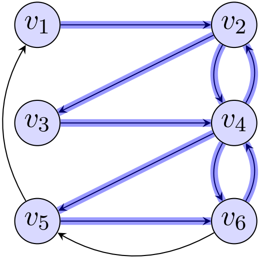
A cycle where every edge of the graph appears in the cycle exactly once.

A directed graph has an Euler tour if the in-degree of each vertex is equal to the out-degree of the vertex.

$(v_1, v_2), (v_2, v_3), (v_3, v_4)$

$(v_4, v_2), (v_2, v_4)$

$(v_4, v_5), (v_5, v_6), (v_6, v_4), (v_4, v_6)$



Euler Tours

Euler Tour (ET)

A cycle where every edge of the graph appears in the cycle exactly once.

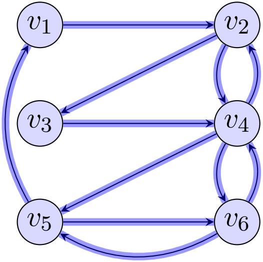
A directed graph has an Euler tour if the in-degree of each vertex is equal to the out-degree of the vertex.

$(v_1, v_2), (v_2, v_3), (v_3, v_4)$

$(v_4, v_2), (v_2, v_4)$

$(v_4, v_5), (v_5, v_6), (v_6, v_4), (v_4, v_6)$

$(v_6, v_5), (v_5, v_1)$

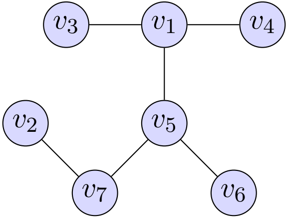


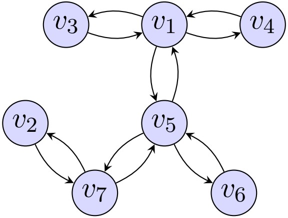
Trees

Tree

An undirected graph that is connected and contains no cycles.

- A tree with n vertices has exactly $n-1$ edges.
- A directed tree (DT) with n vertices has $2n-2$ edges.
    - ➩ Its ET is a sequence of $2n-2$ edges.





Outline

1. Introduction
2. Building Euler Tours
   - Data Structure
   - The Algorithm
   - Analysis
3. Applications of Euler Tours
4. Conclusion

Directed Trees (DT) as Linked Lists

- n linked lists, one for each vertex.
- The nodes of a list are edges leaving that vertex.

head(v_1) $\longrightarrow$ $\boxed{(v_1, v_3)} \rightarrow \boxed{(v_1, v_4)} \rightarrow \boxed{(v_1, v_5)}$

head(v_2) $\longrightarrow$ $\boxed{(v_2, v_7)}$

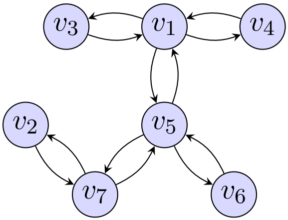
head(v_3) $\longrightarrow$ $\boxed{(v_3, v_1)}$

head(v_4) $\longrightarrow$ $\boxed{(v_4, v_1)}$

head(v_5) $\longrightarrow$ $\boxed{(v_5, v_1)} \rightarrow \boxed{(v_5, v_6)} \rightarrow \boxed{(v_5, v_7)}$

head(v_6) $\longrightarrow$ $\boxed{(v_6, v_5)}$

head(v_7) $\longrightarrow$ $\boxed{(v_7, v_2)} \rightarrow \boxed{(v_7, v_5)}$

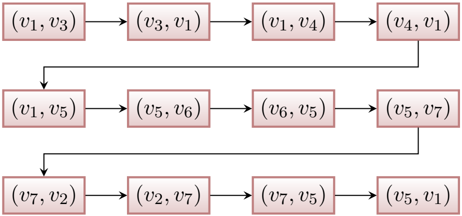


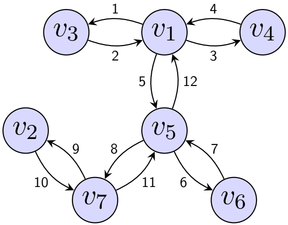
Outline

1. Introduction
2. Building Euler Tours
   - Data Structure
   - The Algorithm
   - Analysis
3. Applications of Euler Tours
4. Conclusion

Algorithm Input and Output

- **Input:**
  n linked lists representing a directed tree (DT).
- **Output:**
  Compute the Euler tour (ET) for the given DT:
  - Arrange all the edges of the DT in a single linked list.
  - Each edge (v_i, v_j) is followed by an edge (v_j, v_k).
  - The first edge leaves some vertex v_l, the last edge enters v_l.





Building the Euler Tour in Parallel

- Assuming the shared-memory model, with $n-1$ processors.
- Each processor $P_{ij}, i < j$, is in charge of two edges: (v_i, v_j) and (v_j, v_i).
- P_{ij} determines the position (actually, the successor) of the two nodes holding (v_i, v_j) and (v_j, v_i) (or ij and ji) — in ET.

Successor of (v_i, v_j)

if next$(ji) = jk$ **then**
  succ$(ij) \leftarrow jk$
else
  succ$(ij) \leftarrow$ head(v_j)
end if

Successor of (v_j, v_i)

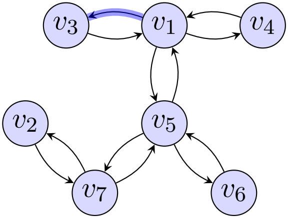
if next$(ij) = im$ **then**
  succ$(ji) \leftarrow im$
else
  succ$(ji) \leftarrow$ head(v_i)
end if

Example: $P_{ij} = P_{13}$

Successor of (v_1, v_3)

Successor of (v_i, v_j)

if next$(ji) = jk$ **then**
  succ$(ij) \leftarrow jk$
else
  succ$(ij) \leftarrow$ head(v_j)
end if



head$(v_1) \longrightarrow \boxed{(v_1, v_3)} \longrightarrow \boxed{(v_1, v_4)} \longrightarrow \boxed{(v_1, v_5)}$

$\vdots$

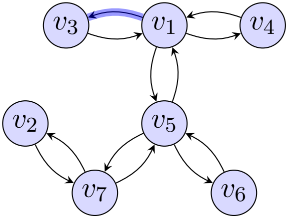
head$(v_3) \longrightarrow \boxed{(v_3, v_1)}$

Example: $P_{ij} = P_{13}$

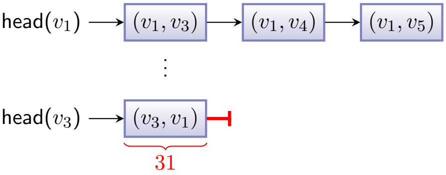
Successor of (v_1, v_3)

Successor of (v_i, v_j)

> **if** next(ji) $= jk$ **then**
> succ$(ij) \leftarrow jk$
> **else**
> succ$(ij) \leftarrow$ head(v_j)
> **end if**



next$(31) =$ `null`

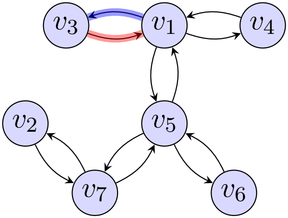


Example: $P_{ij} = P_{13}$

Successor of (v_1, v_3)

Successor of (v_i, v_j)

if next$(ji) = jk$ **then**
    succ$(ij) \leftarrow jk$
else
    succ$(ij) \leftarrow$ head(v_j)
end if



head$(v_1) \longrightarrow \overbrace{(v_1, v_3)}^{13} \longrightarrow (v_1, v_4) \longrightarrow (v_1, v_5)$

$\vdots$

head$(v_3) \longrightarrow (v_3, v_1)$

succ$(13) \leftarrow$ head(v_3)

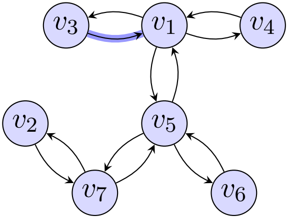
$(v_1, v_3) \longrightarrow (v_3, v_1)$

Example: $P_{ij} = P_{13}$

Successor of (v_3, v_1)

Successor of (v_j, v_i)

if next$(ij) = im$ **then**
  succ$(ji) \leftarrow im$
else
  succ$(ji) \leftarrow$ head(v_i)
end if



head$(v_1) \longrightarrow$ (v_1, v_3) $\longrightarrow$ (v_1, v_4) $\longrightarrow$ (v_1, v_5)

$\vdots$

head$(v_3) \longrightarrow$ (v_3, v_1)

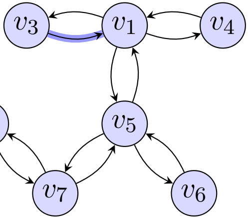
Example: $P_{ij} = P_{13}$

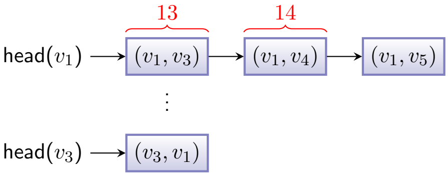
Successor of (v_3, v_1)

Successor of (v_j, v_i)

if next$(ij) = im$ **then**
  succ$(ji) \leftarrow im$
else
  succ$(ji) \leftarrow$ head(v_i)
end if

next$(13) = 14$



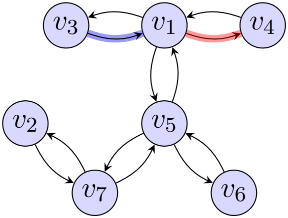


Example: $P_{ij} = P_{13}$

Successor of (v_3, v_1)

Successor of (v_j, v_i)

if next$(ij) = im$ **then**
  succ$(ji) \leftarrow im$
else
  succ$(ji) \leftarrow$ head(v_i)
end if



$$\text{succ}(31) \leftarrow 14$$

$(v_1, v_3) \rightarrow (v_3, v_1) \rightarrow (v_1, v_4)$

head$(v_1) \longrightarrow (v_1, v_3) \rightarrow \overbrace{(v_1, v_4)}^{14} \rightarrow (v_1, v_5)$

$\vdots$

head$(v_3) \longrightarrow \underbrace{(v_3, v_1)}_{31}$

Outline

1. Introduction
2. Building Euler Tours
   - Data Structure
   - The Algorithm
   - Analysis
3. Applications of Euler Tours
4. Conclusion

Correctness

- Visits all the edges:
  - Every pair of edges is assigned to a processor.
- Produces a single cycle, not several small cycles:

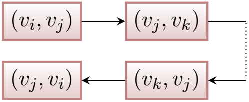
$\boxed{(v_i, v_j)} \longrightarrow \boxed{(v_j, v_k)}$

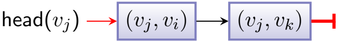
head(v_j) $\dashrightarrow \boxed{(v_j, v_i)} \longrightarrow \boxed{(v_j, v_k)} \dashrightarrow$

next(ji) $= jk$

Correctness

- Visits all the edges:
  - ➩ Every pair of edges is assigned to a processor.
- Produces a single cycle, not several small cycles:





next(ji) $= jk$

next$(jk) = ji$, or head(v_j) $= ji$

Complexity

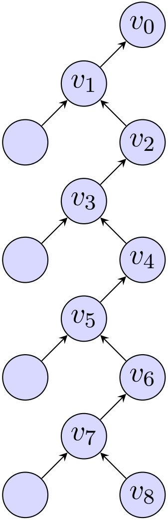
- $t(n) = O(1)$
- $p(n) = n - 1 = O(n)$

$$\Rightarrow \qquad c(n) = O(n)$$

Outline

Pointer Jumping $\text{parent}(v_i) \leftarrow \text{parent}(\text{parent}(v_i))$

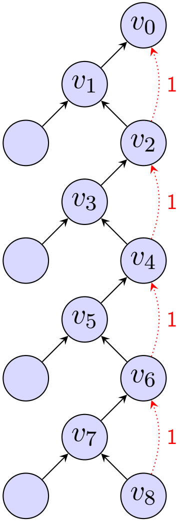
Find the root of v_8.



Pointer Jumping

$\text{parent}(v_i) \leftarrow \text{parent}(\text{parent}(v_i))$

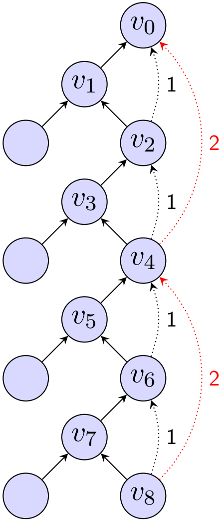
Find the root of v_8.



Pointer Jumping

$$\text{parent}(v_i) \leftarrow \text{parent}(\text{parent}(v_i))$$

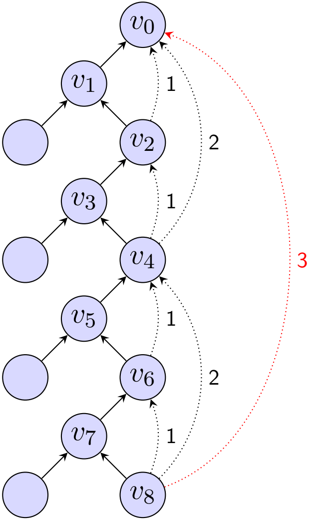
Find the root of v_8.



Pointer Jumping

parent(v_i) $\leftarrow$ parent(parent(v_i))

Find the root of v_8.



Time Complexity

Sequential: $O(n)$

Parallel: $O(\log n)$

Parallel Prefix Computation for Linked Lists

Step 1: **forall** i **do in parallel**
  next(i) $\leftarrow$ succ(i)

Step 2: $\mathit{finished} \leftarrow$ **false**

Step 3: **while not** $\mathit{finished}$ **do**
  (3.1) $\mathit{finished} \leftarrow$ **true**
  (3.2) **forall** i **do in parallel**
    (i) **if** next(i) $\neq \mathit{nil}$ **then**
      (a) val(next(i)) $\leftarrow$ val(i) $\circ$ val(next(i))
      (b) next(i) $\leftarrow$ next(next(i))
    (ii) **if** next(i) $\neq \mathit{nil}$ **then**
      $\mathit{finished} \xleftarrow{\text{common}}$ **false**

Input & Output

Input: A linked list
Output: Prefix computation

$$x_0 \to x_1 \to x_2 \to x_3 \to x_4 \to x_5 \to x_6 \to x_7$$

Parallel Prefix Computation for Linked Lists

Step 1: **forall** i **do in parallel**
next(i) $\leftarrow$ succ(i)

Step 2: *finished* $\leftarrow$ **false**

Step 3: **while not** *finished* **do**

(3.1) *finished* $\leftarrow$ **true**

(3.2) **forall** i **do in parallel**

(i) **if** next(i) $\neq$ *nil* **then**

(a) val(next(i)) $\leftarrow$ val(i) $\circ$ val(next(i))

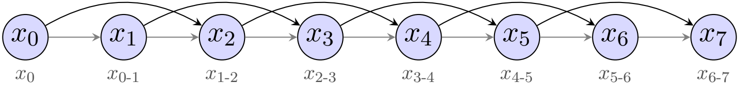
(b) next(i) $\leftarrow$ next(next(i))

(ii) **if** next(i) $\neq$ *nil* **then**

finished $\xleftarrow{\text{common}}$ **false**

Notation

$$x_{a\text{-}b} = x_a \circ x_{a+1} \circ \cdots \circ x_b$$



Parallel Prefix Computation for Linked Lists

Step 1: **forall** i **do in parallel**

next(i) $\leftarrow$ succ(i)

Step 2: *finished* $\leftarrow$ **false**

Step 3: **while not** *finished* **do**

(3.1) *finished* $\leftarrow$ **true**

(3.2) **forall** i **do in parallel**

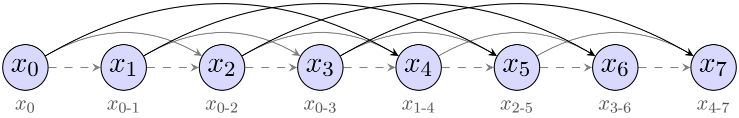
(i) **if** next(i) $\neq$ nil **then**

(a) val(next(i)) $\leftarrow$ val(i) $\circ$ val(next(i))

(b) next(i) $\leftarrow$ next(next(i))

(ii) **if** next(i) $\neq$ nil **then**

finished $\xleftarrow{\text{common}}$ **false**



Parallel Prefix Computation for Linked Lists

Step 1: **forall** i **do in parallel**
next(i) $\leftarrow$ succ(i)

Step 2: $finished \leftarrow$ **false**

Step 3: **while not** $finished$ **do**

(3.1) $finished \leftarrow$ **true**

(3.2) **forall** i **do in parallel**

(i) **if** next(i) $\neq nil$ **then**

(a) val(next(i)) $\leftarrow$ val(i) $\circ$ val(next(i))

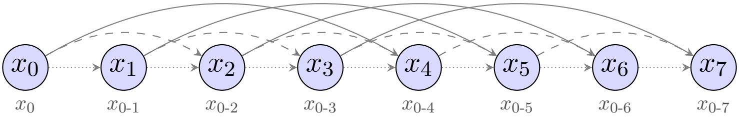
(b) next(i) $\leftarrow$ next(next(i))

(ii) **if** next(i) $\neq nil$ **then**

$finished \xleftarrow{\text{common}}$ **false**

Complexity

$$t(n) = O(\log n)$$
$$p(n) = n$$



Parallel Suffix Computation for Linked Lists

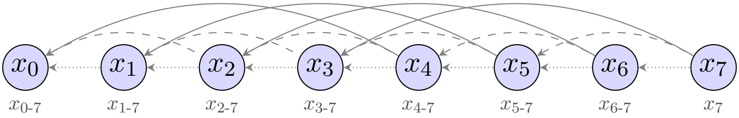
Step 1: **forall** i **do in parallel**
  next$(i) \leftarrow$ succ(i)

Step 2: $\mathit{finished} \leftarrow$ **false**

Step 3: **while not** $\mathit{finished}$ **do**
  (3.1) $\mathit{finished} \leftarrow$ **true**
  (3.2) **forall** i **do in parallel**
    (i) **if** next$(i) \neq \mathit{nil}$ **then**
      (a) val$(i) \leftarrow$ val$(i) \circ$ val(next(i))
      (b) next$(i) \leftarrow$ next(next(i))
    (ii) **if** next$(i) \neq \mathit{nil}$ **then**
      $\mathit{finished} \xleftarrow{\text{common}}$ **false**

Complexity

$$t(n) = O(\log n)$$
$$p(n) = n$$



List Sequencing

List Sequencing

Computing the distance of each node from the *beginning* of the list (node position).

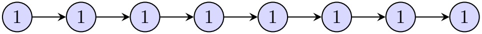
Input: Linked list L

Output: Sequence numbers

forall i **do in parallel**

  val(i) $\leftarrow$ 1

Parallel-Prefix(L, $+$)



List Sequencing

List Sequencing

Computing the distance of each node from the *beginning* of the list (node position).

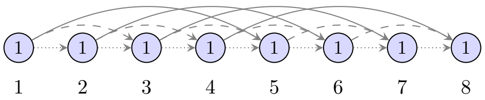
Input: Linked list L

Output: Sequence numbers

forall i **do in parallel**

val(i) $\leftarrow$ 1

Parallel-Prefix(L, $+$)



List Ranking

List Ranking

Computing the distance of each node from the *end* of the list.

Input: Linked list L **Output:** Node ranks

forall i **do in parallel**

(3.1) val(i) $\leftarrow$ 1

(3.2) **if** succ(i) $\neq$ nil **then**

succ(succ(i)) $\leftarrow$ i

Parallel-Prefix(L, $+$)

$$1 \to 1 \to 1 \to 1 \to 1 \to 1 \to 1 \to 1$$

List Ranking

List Ranking

Computing the distance of each node from the *end* of the list.

Input: Linked list L **Output:** Node ranks

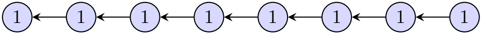
forall i **do in parallel**

(3.1) val$(i) \leftarrow 1$

(3.2) **if** succ$(i) \neq nil$ **then**

succ(succ$(i)) \leftarrow i$

Parallel-Prefix$(L, +)$



List Ranking

List Ranking

Computing the distance of each node from the *end* of the list.

Input: Linked list L

Output: Node ranks

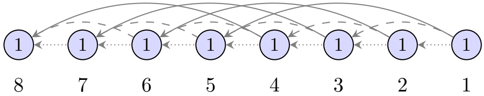
forall i **do in parallel**

(3.1) val(i) $\leftarrow$ 1

(3.2) **if** succ(i) $\neq nil$ **then**

succ(succ(i)) $\leftarrow i$

Parallel-Prefix(L, $+$)

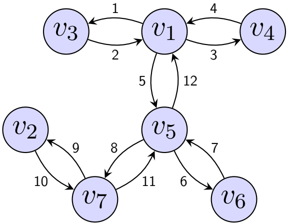


Complexity

$$t(n) = O(\log n), \qquad p(n) = O(n)$$

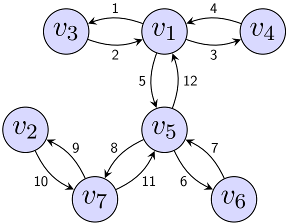
Outline

An Euler Tour is a . . .



An Euler Tour returns to the root of a subtree only after all vertices in the subtree have been visited.

Depth-First Traversal

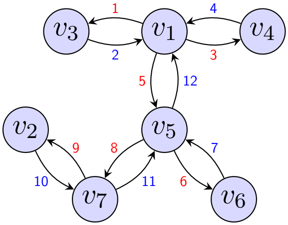


An Euler Tour returns to the root of a subtree only after all vertices in the subtree have been visited.

Finding Parents

- Recall:
  - pos(i) is node i's sequence number.
  - In an Euler Tour, a node looks like (v_i, v_j).
  - If (v_i, v_j) is a node, then (v_j, v_i) is also a node.
- pos(v_i, v_j) $<$ pos(v_j, v_i) $\Rightarrow$ (v_i, v_j) is an advance edge
  otherwise (v_i, v_j) is a retreat edge.

parent(root) $\leftarrow$ *nil*
forall $(v_i, v_j) \in$ ET **do in parallel**
    if pos(v_i, v_j) $<$ pos(v_j, v_i) **then**
        parent(v_j) $\leftarrow v_i$

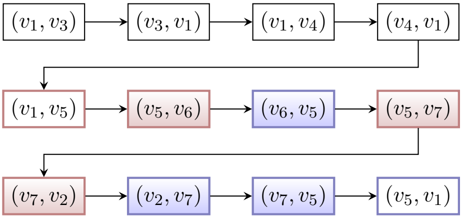


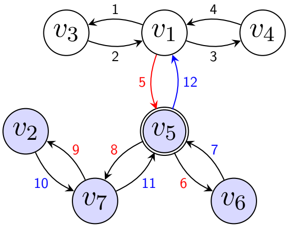
Counting Descendants

$\text{des}(v_i)$

- The subtree of v_i edges appear between the advance edge $(\text{parent}(v_i),\ v_i)$ and the retreat edge $(v_i,\ \text{parent}(v_i))$ in ET.
- Half of these edges are advance edges and half are retreat.
- $\text{des}(v_i)$ is the number of advance edges in the subtree $+\,1$.

$$\text{des}(v_i) = \frac{\text{pos}(v_i,\ \text{parent}(v_i)) - 1 - \text{pos}(\text{parent}(v_i),\ v_i)}{2} + 1$$

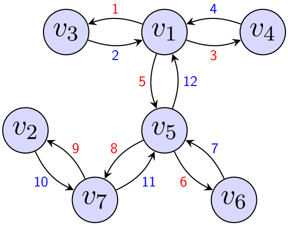




Numbering Vertices Preorder (top-down)

The preorder number of a vertex v_j is the number of advance edges in ET traversed before reaching v_j for the first time $+ 1$.

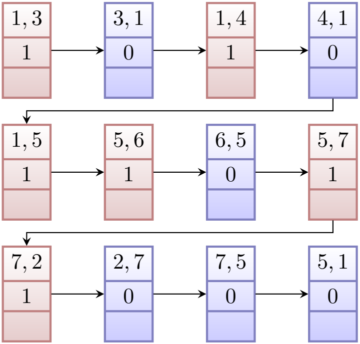
preorder(root) $\leftarrow 1$
forall $(v_i, v_j) \in$ ET **do in parallel**
  if pos(v_i, v_j) $<$ pos(v_j, v_i) **then**
    val(ij) $\leftarrow$ 1
  else
    val(ij) $\leftarrow$ 0
Parallel-Prefix(ET, $+$)
forall $(v_i, v_j) \in$ ET **do in parallel**
  if pos(v_i, v_j) $<$ pos(v_j, v_i) **then**
    preorder(v_j) $\leftarrow$ val(ij) $+$ 1



Numbering Vertices — Preorder (top-down)

The preorder number of a vertex v_j is the number of advance edges in ET traversed before reaching v_j for the first time $+ 1$.

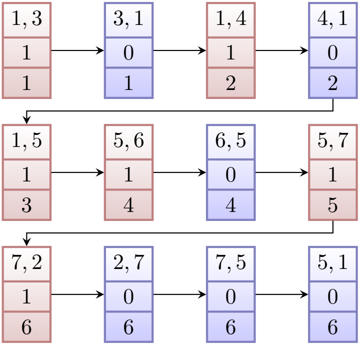
preorder(root) $\leftarrow 1$
forall $(v_i, v_j) \in$ ET **do in parallel**
  if pos(v_i, v_j) $<$ pos(v_j, v_i) **then**
    val(ij) $\leftarrow$ 1
  else
    val(ij) $\leftarrow$ 0
Parallel-Prefix(ET, $+$)
forall $(v_i, v_j) \in$ ET **do in parallel**
  if pos(v_i, v_j) $<$ pos(v_j, v_i) **then**
    preorder(v_j) $\leftarrow$ val(ij) $+$ 1



Numbering Vertices — Preorder (top-down)

The preorder number of a vertex v_j is the number of advance edges in ET traversed before reaching v_j for the first time $+ 1$.

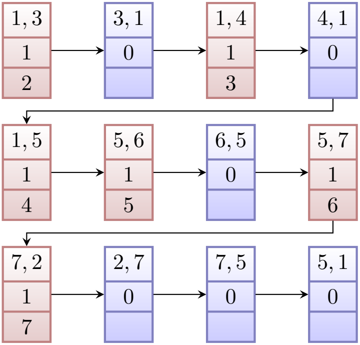
preorder(root) $\leftarrow 1$
forall $(v_i, v_j) \in$ ET **do in parallel**
  if pos$(v_i,\, v_j) <$ pos$(v_j,\, v_i)$ **then**
    val$(ij) \leftarrow 1$
  else
    val$(ij) \leftarrow 0$
Parallel-Prefix(ET, $+$)
forall $(v_i, v_j) \in$ ET **do in parallel**
  if pos$(v_i,\, v_j) <$ pos$(v_j,\, v_i)$ **then**
    preorder$(v_j) \leftarrow$ val$(ij) + 1$



Numbering Vertices — Preorder (top-down)

The preorder number of a vertex v_j is the number of advance edges in ET traversed before reaching v_j for the first time $+ 1$.

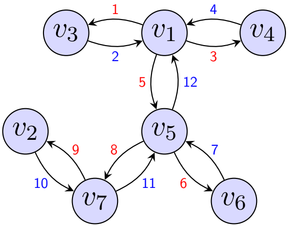
preorder(root) $\leftarrow 1$
forall $(v_i, v_j) \in$ ET **do in parallel**
  if pos(v_i, v_j) $<$ pos(v_j, v_i) **then**
    val(ij) $\leftarrow$ 1
  else
    val(ij) $\leftarrow$ 0
Parallel-Prefix(ET, $+$)
forall $(v_i, v_j) \in$ ET **do in parallel**
  if pos(v_i, v_j) $<$ pos(v_j, v_i) **then**
    preorder(v_j) $\leftarrow$ val(ij) $+$ 1



Numbering Vertices — Postorder (bottom-up)

The postorder number of a vertex v_i is the number of retreat edges in ET traversed before reaching v_i for the last time $+1$.

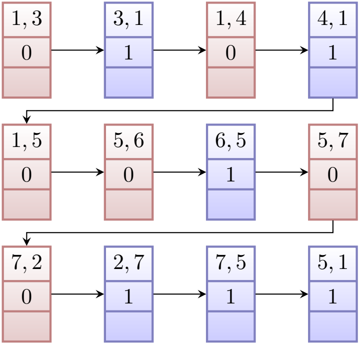
postorder(root) $\leftarrow n$
forall $(v_i, v_j) \in$ ET **do in parallel**
  if pos(v_i, v_j) $<$ pos(v_j, v_i) **then**
    val(ij) $\leftarrow$ 0
  else
    val(ij) $\leftarrow$ 1
Parallel-Prefix(ET, $+$)
forall $(v_i, v_j) \in$ ET **do in parallel**
  if pos(v_i, v_j) $>$ pos(v_j, v_i) **then**
    postorder(v_i) $\leftarrow$ val(ij)



Numbering Vertices — Postorder (bottom-up)

The postorder number of a vertex v_i is the number of retreat edges in ET traversed before reaching v_i for the last time $+1$.

postorder(root) $\leftarrow n$
forall $(v_i, v_j) \in$ ET **do in parallel**
  if pos(v_i, v_j) $<$ pos(v_j, v_i) **then**
    val(ij) $\leftarrow$ 0
  else
    val(ij) $\leftarrow$ 1
Parallel-Prefix(ET, $+$)
forall $(v_i, v_j) \in$ ET **do in parallel**
  if pos(v_i, v_j) $>$ pos(v_j, v_i) **then**
    postorder(v_i) $\leftarrow$ val(ij)



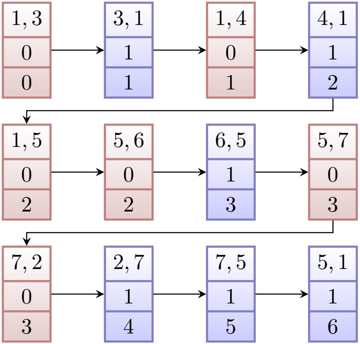
Numbering Vertices — Postorder (bottom-up)

The postorder number of a vertex v_i is the number of retreat edges in ET traversed before reaching v_i for the last time $+1$.

postorder(root) $\leftarrow n$
forall $(v_i, v_j) \in$ ET **do in parallel**
  if pos(v_i, v_j) $<$ pos(v_j, v_i) **then**
    val(ij) $\leftarrow$ 0
  else
    val(ij) $\leftarrow$ 1

Parallel-Prefix(ET, $+$)

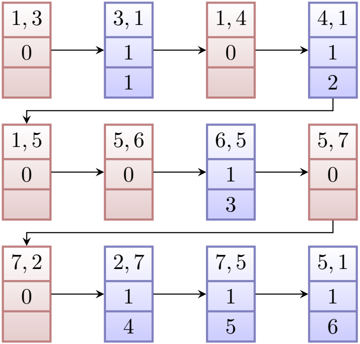
forall $(v_i, v_j) \in$ ET **do in parallel**
  if pos(v_i, v_j) $>$ pos(v_j, v_i) **then**
    postorder(v_i) $\leftarrow$ val(ij)



Numbering Vertices — Postorder (bottom-up)

The postorder number of a vertex v_i is the number of retreat edges in ET traversed before reaching v_i for the last time $+1$.

postorder(root) $\leftarrow n$
forall $(v_i, v_j) \in$ ET **do in parallel**
  if pos(v_i, v_j) $<$ pos(v_j, v_i) **then**
    val(ij) $\leftarrow$ 0
  else
    val(ij) $\leftarrow$ 1
Parallel-Prefix(ET, $+$)
forall $(v_i, v_j) \in$ ET **do in parallel**
  if pos(v_i, v_j) $>$ pos(v_j, v_i) **then**
    postorder(v_i) $\leftarrow$ val(ij)



Tree Binary Relations

- "An Euler Tour returns to the root of a subtree only after all vertices in the subtree have been visited."
- v_i is an ancestor of v_j (v_j is a descendant of v_i) iff:

$$\text{preorder}(v_i) \leq \text{preorder}(v_j) < \text{preorder}(v_i) + \text{des}(v_i)$$

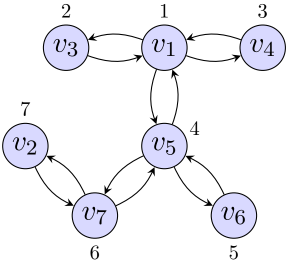
Examples:

v_7 is an ancestor of v_2:

$$6 \leq 7 < 6 + 2 = 8$$

v_6 is *not* an ancestor of v_2:

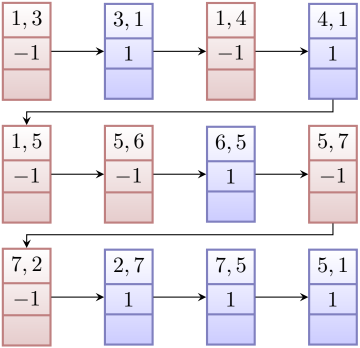
$$5 \leq 7 \not< 5 + 1 = 6$$



Levels of Vertices

level(v_j) $=$ retreat edges $-$ advance edges, following first v_j in ET

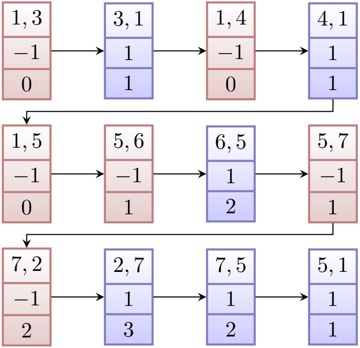
level(root) $\leftarrow$ 0
forall $(v_i, v_j) \in$ ET **do in parallel**
  if pos(v_i, v_j) $<$ pos(v_j, v_i) **then**
    val(ij) $\leftarrow -1$
  else
    val(ij) $\leftarrow 1$
Parallel-Suffix(ET, $+$)
forall $(v_i, v_j) \in$ ET **do in parallel**
  if pos(v_i, v_j) $<$ pos(v_j, v_i) **then**
    level(v_j) $\leftarrow$ val(ij) $+ 1$



Levels of Vertices

level(v_j) = retreat edges − advance edges, following first v_j in ET

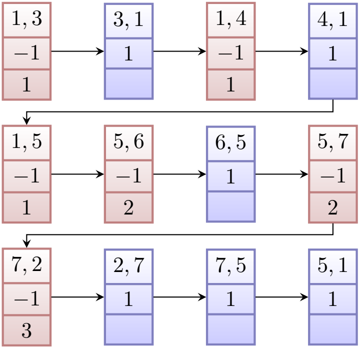
level(root) $\leftarrow$ 0
forall $(v_i, v_j) \in$ ET **do in parallel**
  if pos(v_i, v_j) $<$ pos(v_j, v_i) **then**
    val(ij) $\leftarrow -1$
  else
    val(ij) $\leftarrow 1$
Parallel-Suffix(ET, +)
forall $(v_i, v_j) \in$ ET **do in parallel**
  if pos(v_i, v_j) $<$ pos(v_j, v_i) **then**
    level(v_j) $\leftarrow$ val(ij) + 1



Levels of Vertices

level(v_j) = retreat edges − advance edges, following first v_j in ET

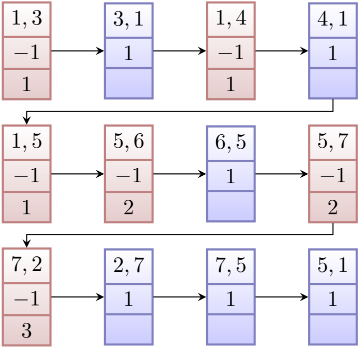
level(root) $\leftarrow$ 0
forall $(v_i, v_j) \in$ ET **do in parallel**
  if pos$(v_i, v_j) <$ pos(v_j, v_i) **then**
    val$(ij) \leftarrow -1$
  else
    val$(ij) \leftarrow 1$
Parallel-Suffix(ET, +)
forall $(v_i, v_j) \in$ ET **do in parallel**
  if pos$(v_i, v_j) <$ pos(v_j, v_i) **then**
    level$(v_j) \leftarrow$ val$(ij) + 1$



Levels of Vertices

$$\begin{aligned}\text{level}(v_j) &= {\color{blue}\text{retreat}} \text{ edges} - {\color{red}\text{advance}} \text{ edges, following first } v_j \text{ in ET} \\ &= {\color{red}\text{advance}} \text{ edges} - {\color{blue}\text{retreat}} \text{ edges, up to first } v_j \text{ in ET}\end{aligned}$$

level(root) $\leftarrow$ 0
forall $(v_i, v_j) \in$ ET **do in parallel**
  if pos(v_i, v_j) $<$ pos(v_j, v_i) **then**
    val(ij) $\leftarrow -1$
  else
    val(ij) $\leftarrow 1$
Parallel-Suffix(ET, $+$)
forall $(v_i, v_j) \in$ ET **do in parallel**
  if pos(v_i, v_j) $<$ pos(v_j, v_i) **then**
    level(v_j) $\leftarrow$ val(ij) $+ 1$



Outline

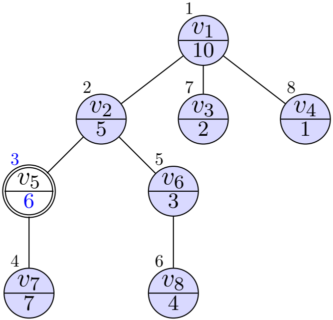
1. Introduction
2. Building Euler Tours
3. Applications of Euler Tours
   - Utility Computations
   - Simple Applications of Euler Tours
   - Computing Minima
4. Conclusion

The Interval Minima Problem

$$S = \{s_k : k = \text{preorder}(v_i) \Rightarrow s_k = \text{val}(v_i)\}$$

Example:

$$3 = \text{preorder}(v_5), s_3 = 6$$
$$S = \{10, 5, 6, 7, 3, 4, 2, 1\}$$



The Interval Minima Problem

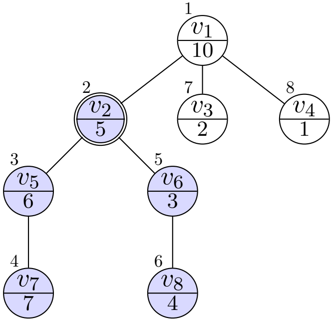
$$S = \{s_k : k = \text{preorder}(v_i) \Rightarrow s_k = \text{val}(v_i)\}$$

$$S_{v_i} = \{s_k, s_{k+1}, \ldots, s_\ell\}, \quad \ell = k + \text{des}(v_i) - 1$$

Example:

$$3 = \text{preorder}(v_5), s_3 = 6$$
$$S = \{10, 5, 6, 7, 3, 4, 2, 1\}$$

$$S_{v_2} : k = 2, \ell = 2 + 5 - 1 = 6$$
$$S_{v_2} = \{5, 6, 7, 3, 4\}$$



The Interval Minima Problem

$$S = \{s_k : k = \text{preorder}(v_i) \Rightarrow s_k = \text{val}(v_i)\}$$

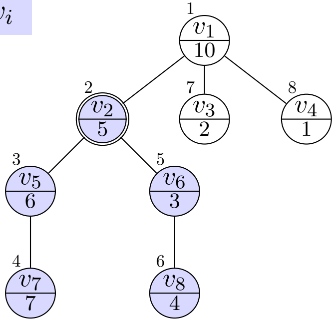
$$S_{v_i} = \{s_k, s_{k+1}, \ldots, s_\ell\}, \quad \ell = k + \text{des}(v_i) - 1$$

$$\boxed{\text{find } \min(S_{v_i}) \quad \forall\, v_i}$$

Example:

$$3 = \text{preorder}(v_5), s_3 = 6$$
$$S = \{10, 5, 6, 7, 3, 4, 2, 1\}$$

$$S_{v_2} : k = 2, \ell = 2 + 5 - 1 = 6$$
$$S_{v_2} = \{5, 6, 7, 3, 4\}$$



The Interval Minima Tree

Prefix Minima (PM)

The prefix minima of a sequence $\{a_1, a_2, \ldots, a_n\}$ are given by a sequence $\{b_1, b_2, \ldots, b_n\}$ where $b_i = \min(a_1, a_2, \ldots, a_i)$.

Suffix Minima (SM)

The suffix minima of a sequence $\{a_1, a_2, \ldots, a_n\}$ are given by a sequence $\{d_1, d_2, \ldots, d_n\}$ where $d_i = \min(a_i, a_{i+1}, \ldots, a_n)$.

PM	2	1	1	1	1	1
Seq.	2	1	3	4	5	6
SM	1	1	3	4	5	6

The Interval Minima Tree

Prefix Minima (PM)

The prefix minima of a sequence $\{a_1, a_2, \ldots, a_n\}$ are given by a sequence $\{b_1, b_2, \ldots, b_n\}$ where $b_i = \min(a_1, a_2, \ldots, a_i)$.

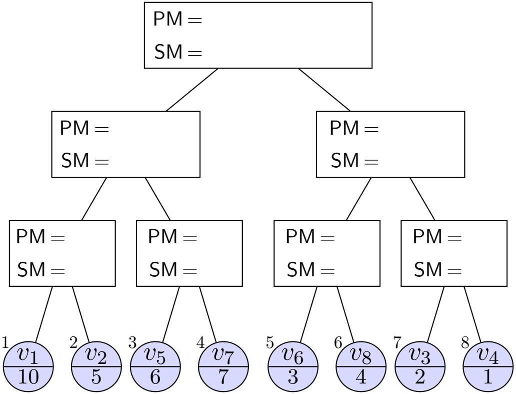
Suffix Minima (SM)

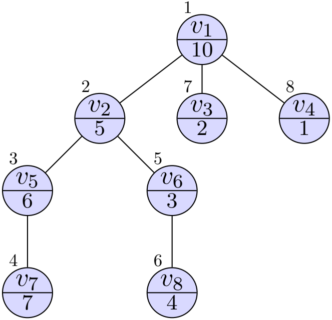
The suffix minima of a sequence $\{a_1, a_2, \ldots, a_n\}$ are given by a sequence $\{d_1, d_2, \ldots, d_n\}$ where $d_i = \min(a_i, a_{i+1}, \ldots, a_n\}$.

A complete binary tree such that:

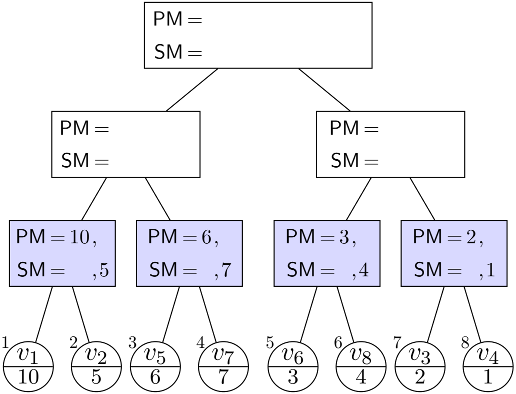
- Leaves are $s_1, s_2, \ldots, s_n$.
- Each internal node x contains two sequences: the prefix minima (PM) and suffix minima (SM) of the values held by leaves in the subtree rooted at x.

Constructing the Tree





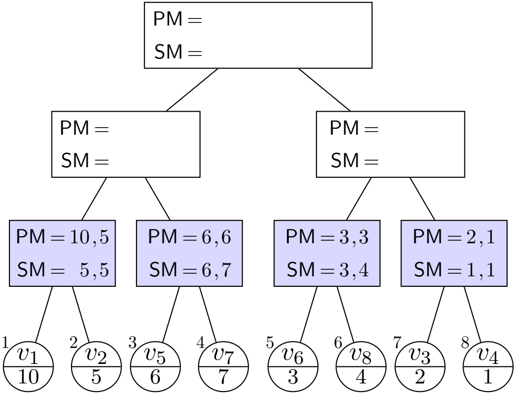
Constructing the Tree



PM:	L	$\{\min(l_p, r_i)\}$
SM:	$\{\min(r_1, l_i)\}$	R

L	R
$l_1, l_2, \ldots, l_p$	$r_1, r_2, \ldots, r_p$

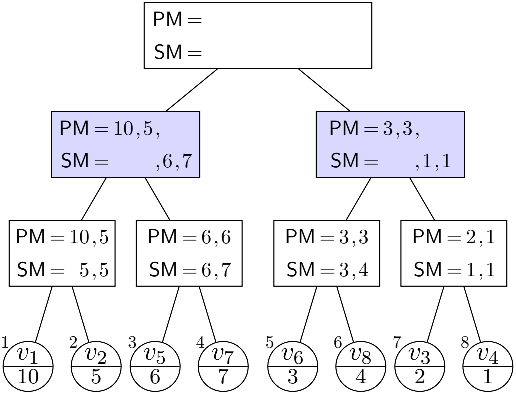
Constructing the Tree

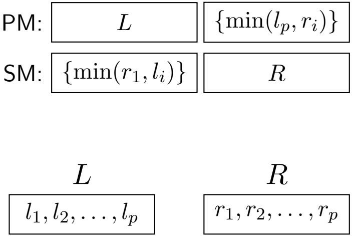


PM:	L	$\{\min(l_p, r_i)\}$
SM:	$\{\min(r_1, l_i)\}$	R

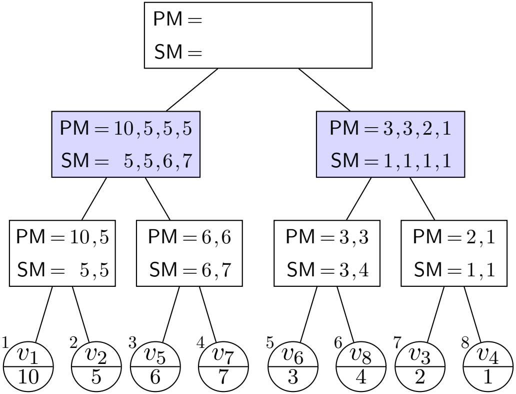
L	R
$l_1, l_2, \ldots, l_p$	$r_1, r_2, \ldots, r_p$

Constructing the Tree





Constructing the Tree

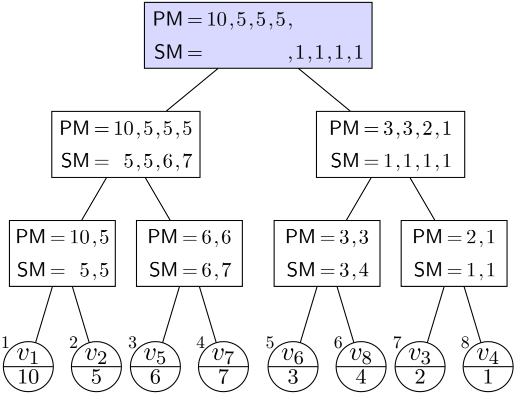


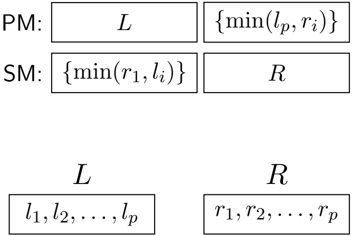
PM:	L	$\{\min(l_p, r_i)\}$
SM:	$\{\min(r_1, l_i)\}$	R

L: $l_1, l_2, \ldots, l_p$

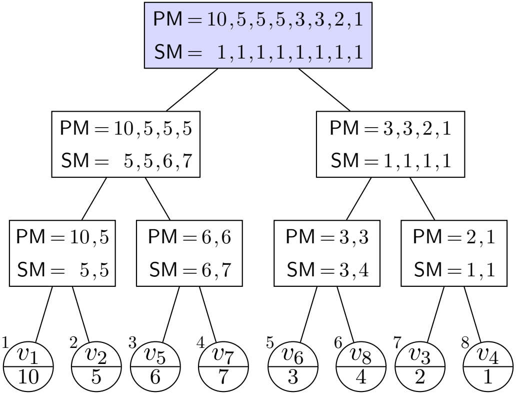
R: $r_1, r_2, \ldots, r_p$

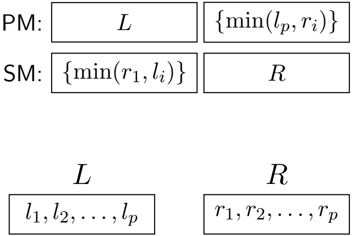
Constructing the Tree





Constructing the Tree

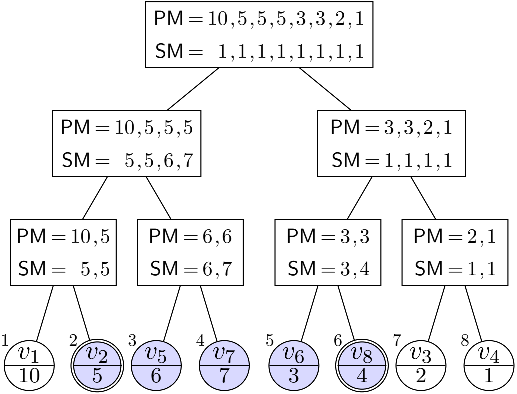


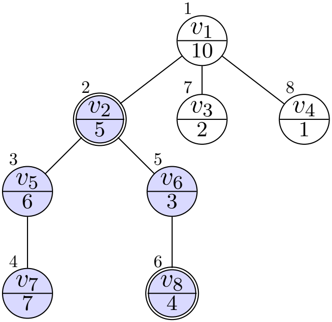


Finding the Minima Using the Tree

An Example: S_{v_2}

$$k = 2, \qquad \ell = k + 5 - 1 = 6 \qquad \Rightarrow \qquad S_{v_2} = \{s_2, \ldots, s_6\}$$

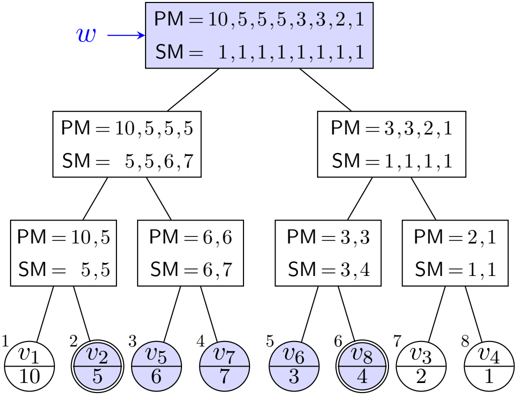


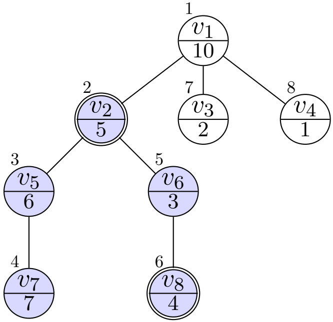


Finding the Minima Using the Tree

An Example: S_{v_2}

$$k = 2, \qquad \ell = k + 5 - 1 = 6 \qquad \Rightarrow \qquad S_{v_2} = \{s_2, \ldots, s_6\}$$

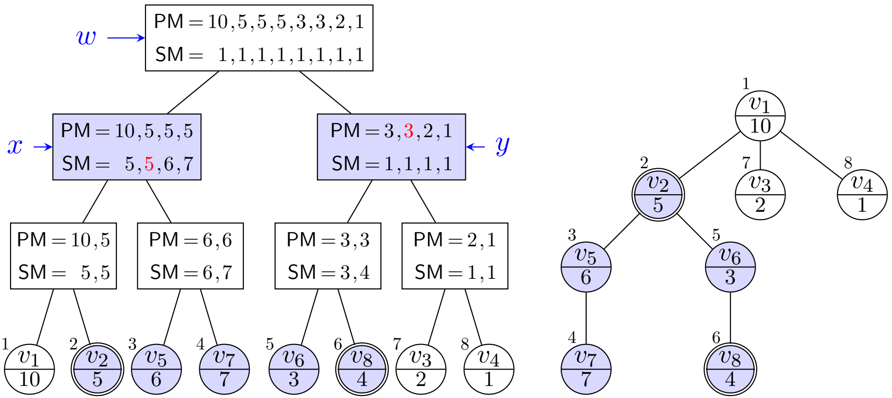




Finding the Minima Using the Tree

An Example: S_{v_2}

$$k = 2, \qquad \ell = k + 5 - 1 = 6 \qquad \Rightarrow \qquad S_{v_2} = \{s_2, \ldots, s_6\}$$



The Interval Minima Algorithm

Input: A tree T **Output:** $\min(S_{v_i})$ for all $v_i \in T$

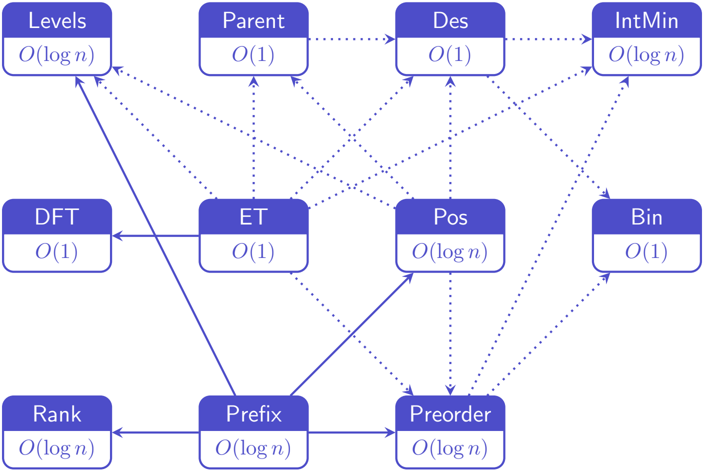
$O(1)$ ET $\leftarrow$ Parallel-Euler-Tour(T)
$O(\log n)$ Parallel-Preorder(ET)
$O(\log n)$ Parallel-Descendants(ET)
$O(\log n)$ $T' \leftarrow$ Interval-Minima-Tree(T)
forall $v_i \in T$ **do in parallel**
$O(1)$ $k \leftarrow$ preorder(v_i), $\ell \leftarrow k +$ des(v_i) $- 1$
$O(\log n)$ $w \leftarrow$ Lowest-Common-Ancestor(s_k, s_ℓ) in T'
$O(1)$ $x \leftarrow$ Left-Child(w), $y \leftarrow$ Right-Child(w);
$O(1)$ $\min(S_{v_i}) \leftarrow \min(\mathsf{SM}_k^{(x)}, \mathsf{PM}_\ell^{(y)})$

Complexity:

$$t(n) = O(\log n) \qquad p(n) = O(n) \qquad \Rightarrow \qquad c(n) = n\log(n)$$

Outline

Summary


Levels
$O(\log n)$
Parent
$O(1)$
Des
$O(1)$
IntMin
$O(\log n)$
DFT
$O(1)$
ET
$O(1)$
Pos
$O(\log n)$
Bin
$O(1)$
Rank
$O(\log n)$
Prefix
$O(\log n)$
Preorder
$O(\log n)$

Thank You

Thank You!

Outline

1. Introduction
2. Building Euler Tours
3. Applications of Euler Tours
4. Conclusion
   - Summary
   - Questions

Questions

???