Algorithms for Graph Visualization

Summer Semester 2019
Lecture #1

Divide-and-Conquer Algorithms:
Trees and Series-Parallel Graphs

(based on slides from Martin Nöllenburg and Robert Görke, KIT)

Refs: [GD: AVG, Ch. 3] and [DG: M&M, Ch. 3]
Uses of Divide & Conquer

Well suited for inductively/recursively defined Graph Classes
Uses of Divide & Conquer

Well suited for inductively/recursively defined Graph Classes

Rooted Binary Trees:
1. draw the left subtree
2. draw the right subtree
3. combine together + draw root
Uses of Divide & Conquer

Well suited for inductively/recursively defined Graph Classes

Rooted Binary Trees:
1. draw the left subtree
2. draw the right subtree
3. combine together + draw root
Uses of Divide & Conquer

Well suited for inductively/recursively defined Graph Classes

Rooted Binary Trees:

1. draw the left subtree
2. draw the right subtree
3. combine together + draw root

Terminology

- depth($v$): distance from the root
- traversal
  - preorder
  - inorder
  - postorder
Overview

- balanced drawings of binary trees $O(nh)$
- radial drawings of trees $O(nh)$
- compact drawings of trees $O(n \log n)$
- upward drawings of series parallel graphs exp.
Overview

- balanced drawings of binary trees \( O(nh) \)
- radial drawings of trees \( O(nh) \)
- compact drawings of trees \( O(n \log n) \)
- upward drawings of series parallel graphs \( \exp \)
Algorithm of Reingold and Tilford ('81)

Trivial: If $T = \{v\}$, draw $v$ (e.g., as a small disk).
Algorithm of Reingold and Tilford ('81)

Trivial: If $T = \{v\}$, draw $v$ (e.g., as a small disk).

Divide: Run Alg. recursively on the left and right subtrees.
Algorithm of Reingold and Tilford ('81)

Trivial: If $T = \{v\}$, draw $v$ (e.g., as a small disk).
Divide: Run Alg. recursively on the left and right subtrees.
Conquer: shift the partial drawings
Algorithm of Reingold and Tilford (’81)

Trivial: If $T = \{v\}$, draw $v$ (e.g., as a small disk).

Divide: Run Alg. recursively on the left and right subtrees.

Conquer: shift the partial drawings to up to 2 units apart
Algorithm of Reingold and Tilford ('81)

Trivial: If $T = \{v\}$, draw $v$ (e.g., as a small disk).

Divide: Run Alg. recursively on the left and right subtrees.

Conquer: shift the partial drawings to up to 2 units apart
place the root $r$ one above and centrally btw. children.
Algorithm of Reingold and Tilford ('81)

Trivial: If $T = \{v\}$, draw $v$ (e.g., as a small disk).

Divide: Run Alg. recursively on the left and right subtrees.

Conquer: shift the partial drawings to up to 2 units apart
place the root $r$ one above and centrally btw. children.
Algorithm of Reingold and Tilford (’81)

Trivial: If $T = \{v\}$, draw $v$ (e.g., as a small disk).

Divide: Run Alg. recursively on the left and right subtrees.

Conquer: shift the partial drawings to up to 2 units apart
place the root $r$ one above and centrally btw. children.

Grid Drawing?
Algorithm of Reingold and Tilford ('81)

Implementation in 2 Phases:

1. postorder (bottom-up):
   contours and x-offsets
   gather the predecessors

2. preorder (top-down):
   calculate absolute coordinates
Algorithm of Reingold and Tilford ('81)

Implementation in 2 Phases:

1. postorder (bottom-up):
   contours and x-offsets
   gather the predecessors

2. preorder (top-down):
   calculate absolute coordinates

Contour: linked list of vertices (-coordinates)
Algorithm of Reingold and Tilford ('81)

Phase 1:
1. compute $T_\ell(v)$ und $T_r(v)$
2. trace the right contour of $T_\ell(v)$ and left of $T_r(v)$
3. Find $d_v = \min.$ horiz. distance between $v_\ell$ und $v_r$
4. $x$-offset($v_\ell$) = $-\lceil d_v/2 \rceil$, $x$-offset($v_r$) = $\lceil d_v/2 \rceil$
5. Build left contour of $T_v$ from:
   - $v$,
   - left contour of $T_\ell(v)$,
   - left contour of any low hanging part of $T_r(v)$
6. Symmetrically for right contour.
Algorithm of Reingold and Tilford ('81)

Phase 1:
1. compute $T_\ell(v)$ und $T_r(v)$
2. trace the right contour of $T_\ell(v)$ and left of $T_r(v)$
3. Find $d_v = \min.$ horiz. distance between $v_\ell$ und $v_r$
4. $x$-offset($v_\ell$) = $-\lceil d_v/2 \rceil$, $x$-offset($v_r$) = $\lceil d_v/2 \rceil$
5. Build left contour of $T_v$ from:
   - $v$,
   - left contour of $T_\ell(v)$,
   - left contour of any low hanging part of $T_r(v)$
6. Symmetrically for right contour.
Algorithm of Reingold and Tilford ('81)

Phase 1:
1. compute $T_\ell(v)$ und $T_r(v)$
2. trace the right contour of $T_\ell(v)$ and left of $T_r(v)$
3. Find $d_v = \min.$ horiz. distance between $v_\ell$ und $v_r$
4. $x$-offset($v_\ell$) = $-\lceil d_v/2 \rceil$, $x$-offset($v_r$) = $\lceil d_v/2 \rceil$
5. Build left contour of $T_v$ from:
   - $v$,
   - left contour of $T_\ell(v)$,
   - left contour of any low hanging part of $T_r(v)$
6. Symmetrically for right contour.
Algorithm of Reingold and Tilford ('81)

**Phase 1:**

1. compute $T_{\ell}(v)$ und $T_{r}(v)$
2. trace the right contour of $T_{\ell}(v)$ and left of $T_{r}(v)$
3. Find $d_{v} = \text{min. horiz. distance between } v_{\ell} \text{ and } v_{r}$
4. $x$-offset($v_{\ell}$) = $-\lceil d_{v}/2 \rceil$, $x$-offset($v_{r}$) = $\lceil d_{v}/2 \rceil$
5. Build left contour of $T_{v}$ from:
   - $v$,
   - left contour of $T_{\ell}(v)$,
   - left contour of any low hanging part of $T_{r}(v)$
6. Symmetrically for right contour.

**Runtime?**
Algorithm of Reingold and Tilford (’81)

Phase 1:
1. compute $T_\ell(v)$ und $T_r(v)$
2. trace the right contour of $T_\ell(v)$ and left of $T_r(v)$
3. Find $d_v = \min$ horiz. distance between $v_\ell$ und $v_r$
4. $x$-offset($v_\ell$) = $-\lceil d_v / 2 \rceil$, $x$-offset($v_r$) = $\lceil d_v / 2 \rceil$
5. Build left contour of $T_v$ from:
   • $v$,
   • left contour of $T_\ell(v)$,
   • left contour of any low hanging part of $T_r(v)$
6. Symmetrically for right contour.

Runtime? $\sum_v (\cdot) =$
Algorithm of Reingold and Tilford (’81)

Phase 1:
1. compute $T_\ell(v)$ und $T_r(v)$
2. trace the right contour of $T_\ell(v)$ and left of $T_r(v)$
3. Find $d_v = \min$ horiz. distance between $v_\ell$ und $v_r$
4. $x$-offset($v_\ell$) = $\lceil d_v / 2 \rceil$, $x$-offset($v_r$) = $\lfloor d_v / 2 \rfloor$
5. Build left contour of $T_v$ from:
   • $v$,
   • left contour of $T_\ell(v)$,
   • left contour of any low hanging part of $T_r(v)$
6. Symmetrically for right contour.

Runtime? $\sum_v (1 + \min\{h_\ell(v), h_r(v)\}) = \ldots$
Phase 1:
1. compute $T_\ell(v)$ und $T_r(v)$
2. trace the right contour of $T_\ell(v)$ and left of $T_r(v)$
3. Find $d_v = \min \text{ horiz. distance between } v_\ell \text{ und } v_r$
4. $x\text{-offset}(v_\ell) = -\lceil d_v / 2 \rceil$, $x\text{-offset}(v_r) = \lceil d_v / 2 \rceil$
5. Build left contour of $T_v$ from:
   - $v$,
   - left contour of $T_\ell(v)$,
   - left contour of any low hanging part of $T_r(v)$
6. Symmetrically for right contour.

Runtime? $\sum_v (1 + \min\{h_\ell(v), h_r(v)\}) = n + \sum_v \min\{\ldots\} \leq$
Algorithm of Reingold and Tilford ('81)

Phase 1:
1. compute $T_\ell(v)$ und $T_r(v)$
2. trace the right contour of $T_\ell(v)$ and left of $T_r(v)$
3. Find $d_v = \min$ horiz. distance between $v_\ell$ und $v_r$
4. $x$-offset($v_\ell$) = $-\lceil d_v/2 \rceil$, $x$-offset($v_r$) = $\lceil d_v/2 \rceil$
5. Build left contour of $T_v$ from:
   • $v$,
   • left contour of $T_\ell(v)$,
   • left contour of any low hanging part of $T_r(v)$
6. Symmetrically for right contour.

Runtime? $\sum_v (1 + \min \{h_\ell(v), h_r(v)\}) = n + \sum_v \min \{\ldots\} \leq$
Algorithm of Reingold and Tilford ('81)

Phase 1:
1. compute $T_\ell(v)$ und $T_r(v)$
2. trace the right contour of $T_\ell(v)$ and left of $T_r(v)$
3. Find $d_v = \min.$ horiz. distance between $v_\ell$ und $v_r$
4. $x$-offset$(v_\ell) = -\lceil d_v/2 \rceil$, $x$-offset$(v_r) = \lceil d_v/2 \rceil$
5. Build left contour of $T_v$ from:
   • $v$
   • left contour of $T_\ell(v)$
   • left contour of any low hanging part of $T_r(v)$
6. Symmetrically for right contour.

Runtime? $\sum_v (1 + \min\{h_\ell(v), h_r(v)\}) = n + \sum_v \min\{\ldots \} \leq$
Algorithm of Reingold and Tilford ('81)

Phase 1:
1. compute $T_\ell(v)$ und $T_r(v)$
2. trace the right contour of $T_\ell(v)$ and left of $T_r(v)$
3. Find $d_v = \text{min. horiz. distance between } v_\ell \text{ und } v_r$
4. $x\text{-offset}(v_\ell) = -\lceil d_v / 2 \rceil$, $x\text{-offset}(v_r) = \lceil d_v / 2 \rceil$
5. Build left contour of $T_v$ from:
   - $v$,
   - left contour of $T_\ell(v)$,
   - left contour of any low hanging part of $T_r(v)$
6. Symmetrically for right contour.

Runtime? $\sum_v (1 + \min\{h_\ell(v), h_r(v)\}) = n + \sum_v \min\{\ldots\} \leq n + n$
Algorithm of Reingold und Tilford ('81)

Phase 2:

• Set \( y \)-coordinate \( y(v) = -\text{depth}(v) \) for each vertex \( v \).
Algorithm of Reingold und Tilford ('81)

Phase 2:

- Set $y$-coordinate $y(v) = -\text{depth}(v)$ for each vertex $v$.
- Set $x(w) := 0$ for the root $w$, then in preorder for $v \in V$:
  \[
  x(v_\ell) := x(v) + x\text{-offset}(v_\ell) \quad \text{and} \quad
  x(v_r) := x(v) + x\text{-offset}(v_r).
  \]
Algorithm of Reingold und Tilford ('81)

Phase 2:

- Set $y$-coordinate $y(v) = -\text{depth}(v)$ for each vertex $v$.
- Set $x(w) := 0$ for the root $w$, then in preorder for $v \in V$:
  - $x(v_\ell) := x(v) + x\text{-offset}(v_\ell)$ and
  - $x(v_r) := x(v) + x\text{-offset}(v_r)$.

Runtime?
Algorithm of Reingold und Tilford ('81)

Phase 2:

- Set $y$-coordinate $y(v) = -\text{depth}(v)$ for each vertex $v$.
- Set $x(w) := 0$ for the root $w$, then in preorder for $v \in V$:
  \[ x(v_\ell) := x(v) + x\text{-offset}(v_\ell) \] and
  \[ x(v_r) := x(v) + x\text{-offset}(v_r). \]

Runtime? $O(n)$
## Summary for Balanced Drawings of Binary Trees

**Theorem** [Reingold & Tilford ’81]

For a binary tree with $n$ vertices, in $O(n)$ time we can produce a drawing $\Gamma$ such that:
Summary for Balanced Drawings of Binary Trees

**Theorem** [Reingold & Tilford ’81]

For a binary tree with \( n \) vertices, in \( O(n) \) time we can produce a drawing \( \Gamma \) such that:

- \( \Gamma \) is layered, i.e., \( y \equiv - \text{depth} \),
- \( \Gamma \) is planar, straightline, and strictly downward,
- \( \Gamma \) matches the embedding (i.e., right children on the right),
- all vertices: horiz. & vert. distances \( \geq 1 \), and on the grid,
- the area is \( O(n^2) \),
- parent always centered above children.
Summary for Balanced Drawings of Binary Trees

**Theorem** [Reingold & Tilford ’81]

For a binary tree with $n$ vertices, in $O(n)$ time we can produce a drawing $\Gamma$ such that:

- $\Gamma$ is layered, i.e., $y \equiv -$ depth,
- $\Gamma$ is planar, straightline, and strictly downward,
- $\Gamma$ matches the embedding (i.e., right children on the right),
- all vertices: horiz. & vert. distances $\geq 1$, and on the grid,
- the area is $O(n^2)$,
- parent always centered above children.
Summary for Balanced Drawings of Binary Trees

Theorem [Reingold & Tilford ’81]

For a binary tree with \( n \) vertices, in \( O(n) \) time we can produce a drawing \( \Gamma \) such that:

- \( \Gamma \) is layered, i.e., \( y \equiv - \) depth,
- \( \Gamma \) is planar, straightline, and strictly downward,
- \( \Gamma \) matches the embedding (i.e., right children on the right),
- all vertices: horiz. & vert. distances \( \geq 1 \), and on the grid,
- the area is \( O(n^2) \),
- parent always centered above children.

Easily generalizes to arbitrary trees!
Summary for Balanced Drawings of Binary Trees

**Theorem** [Reingold & Tilford ’81]

For a binary tree with \( n \) vertices, in \( O(n) \) time we can produce a drawing \( \Gamma \) such that:

- \( \Gamma \) is layered, i.e., \( y \equiv - \text{depth} \),
- \( \Gamma \) is planar, straightline, and strictly downward,
- \( \Gamma \) matches the embedding (i.e., right children on the right),
- all vertices: horiz. & vert. distances \( \geq 1 \), and on the grid,
- the area is \( O(n^2) \),
- parent always centered above children.

Min. width (but without the grid):

Easily generalizes to arbitrary trees!
Summary for Balanced Drawings of Binary Trees

[Reingold & Tilford '81]

**Theorem**

For a binary tree with \( n \) vertices, in \( O(n) \) time we can produce a drawing \( \Gamma \) such that:

- \( \Gamma \) is layered, i.e., \( y \equiv - \text{depth} \),
- \( \Gamma \) is planar, straightline, and strictly downward,
- \( \Gamma \) matches the embedding (i.e., right children on the right),
- all vertices: horiz. & vert. distances \( \geq 1 \), and on the grid,
- the area is \( O(n^2) \),
- parent always centered above children.

Min. width (but without the grid): by Linear Programming (LP)!

Easily generalizes to arbitrary trees!
Summary for Balanced Drawings of Binary Trees

Theorem [Reingold & Tilford ’81]
For a binary tree with \( n \) vertices, in \( O(n) \) time we can produce a drawing \( \Gamma \) such that:

- \( \Gamma \) is layered, i.e., \( y \equiv - \text{depth} \),
- \( \Gamma \) is planar, straightline, and strictly downward,
- \( \Gamma \) matches the embedding (i.e., right children on the right),
- all vertices: horiz. & vert. distances \( \geq 1 \), and on the grid,
- the area is \( O(n^2) \),
- parent always centered above children.

Min. width (but without the grid): by Linear Programming (LP)!

Min. width and on the grid:

Easily generalizes to arbitrary trees!

example?
## Summary for Balanced Drawings of Binary Trees

**Theorem** [Reingold & Tilford ’81]

For a binary tree with \( n \) vertices, in \( O(n) \) time we can produce a drawing \( \Gamma \) such that:

- \( \Gamma \) is layered, i.e., \( y \equiv - \) depth,
- \( \Gamma \) is planar, straightline, and strictly downward,
- \( \Gamma \) matches the embedding (i.e., right children on the right),
- all vertices: horiz. & vert. distances \( \geq 1 \), and on the grid,
- the area is \( O(n^2) \),
- parent always centered above children.

Easily generalizes to arbitrary trees!

```latex
\begin{itemize}
  \item \( \Gamma \) is layered, i.e., \( y \equiv - \) depth,
  \item \( \Gamma \) is planar, straightline, and strictly downward,
  \item \( \Gamma \) matches the embedding (i.e., right children on the right),
  \item all vertices: horiz. & vert. distances \( \geq 1 \), and on the grid,
  \item the area is \( O(n^2) \),
  \item parent always centered above children.
\end{itemize}
```

Min. width (but without the grid): by Linear Programming (LP)!

Min. width and on the grid: NP-hard! [Supowit & Reingold ’83]
Example of width variation

Output of the Algorithm:
Example of width variation

Output of the Algorithm:
Example of width variation

Output of the Algorithm:
Example of width variation

Output of the Algorithm:

Optimal Drawing:
Example of width variation

Output of the Algorithm:

Optimal Drawing:
2. Radial Drawings of Trees

- balanced drawings of binary trees $O(nh)$
- radial drawings of trees $O(nh)$
- compact drawings of trees $O(n \log n)$
- upward drawings of series parallel graphs $\exp.$
Example: Radial Tree Layouts
An Algorithm for Radial Layout?
An Algorithm for Radial Layout?
An Algorithm for Radial Layout?
An Algorithm for Radial Layout?
An Algorithm for Radial Layout?
An Algorithm for Radial Layout?
An Algorithm for Radial Layout?
An Algorithm for Radial Layout?
Restricting to Smaller Sectors
Restricting to Smaller Sectors
Restricting to Smaller Sectors
Restricting to Smaller Sectors

\[ \rho_i + 1 \]

\[ \rho_i \]

\[ \nu \]
Restricting to Smaller Sectors

\[ \cos \tau = \frac{\rho_i}{\rho_{i+1}} \]
Restricting to Smaller Sectors

\[ \cos \tau = \frac{\rho_i}{\rho_{i+1}} \]
Restricting to Smaller Sectors

\[ \cos \tau = \frac{\rho_i}{\rho_{i+1}} \]

\[ \alpha_{\text{min}} = \alpha_v - \arccos \left( \frac{\rho_i}{\rho_{i+1}} \right) \]
Restricting to Smaller Sectors

\[ \cos \tau = \frac{\rho_i}{\rho_{i+1}} \]

\[ \Rightarrow \quad \alpha_{\text{min}} = \alpha_v - \arccos \frac{\rho_i}{\rho_{i+1}} \]

\[ \alpha_{\text{max}} = \alpha_v + \arccos \frac{\rho_i}{\rho_{i+1}} \]
Pseudocode for radial tree layout

RadialTreeLayout(tree $T$, root $r \in T$, radii $\rho_1 < \cdots < \rho_k$)

begin
  postorder($r$)
  preorder($r, 0, 0, 2\pi$)
  return $(d_v, \alpha_v)_{v \in V(T)}$

  {vertex pos./ polar coord.}

postorder(vertex $v$)

  calculate the size of the subtree recursively
Pseudocode for radial tree layout

RadialTreeLayout(tree $T$, root $r \in T$, radii $\rho_1 < \cdots < \rho_k$)

\begin{align*}
\text{begin} & \\
\quad & \text{postorder}(r) \\
\quad & \text{preorder}(r, 0, 0, 2\pi) \\
\quad & \text{return } (d_v, \alpha_v)_{v \in V(T)} \\
\end{align*}

\{vertex pos./ polar coord.\}

\text{postorder}(\text{vertex } v) \begin{align*}
\quad & n_v \leftarrow 1 \\
\quad & \text{foreach child } w \text{ of } v \text{ do} \\
\quad & \quad \text{postorder}(w) \\
\quad & \quad n_v \leftarrow n_v + n_w
\end{align*}
Pseudocode for radial tree layout

RadialTreeLayout(tree $T$, root $r \in T$, radii $\rho_1 < \cdots < \rho_k$)

begin
    \text{postorder}(r)
    \text{preorder}(r, 0, 0, 2\pi)
    \text{return } (d_v, \alpha_v)_{v \in V(T)}$
    \{vertex pos./ polar coord.\}
\end{align*}

\text{postorder}(vertex $v$)

\begin{align*}
    n_v &\leftarrow 1 \\
    \text{foreach } \text{child } w \text{ of } v \text{ do} \\
    &\quad \text{postorder}(w) \\
    n_v &\leftarrow n_v + n_w
\end{align*}
Pseudocode for radial tree layout

RadialTreeLayout(tree $T$, root $r \in T$, radii $\rho_1 < \cdots < \rho_k$)

begin

postorder($r$)

preorder($r$, 0, 0, $2\pi$)

return $(d_v, \alpha_v)_{v \in V(T)}$

{vertex pos./ polar coord.}

postorder(vertex $v$)

$n_v \leftarrow 1$

foreach child $w$ of $v$ do

postorder($w$)

$n_v \leftarrow n_v + n_w$

end

size of the subtree $T(v)$
Pseudocode for radial tree layout

RadialTreeLayout(tree $T$, root $r \in T$, radii $\rho_1 < \cdots < \rho_k$)

\[
\begin{align*}
\text{begin} & \\
    & \text{postorder}(r) \\
    & \text{preorder}(r, 0, 0, 2\pi) \\
    & \text{return } (d_v, \alpha_v)_{v \in V(T)} \\
    & \{ \text{vertex pos./ polar coord.} \}
\end{align*}
\]

\[
\begin{align*}
\text{postorder(vertex } v) & \\
    & n_v \leftarrow 1 \\
    & \text{foreach child } w \text{ of } v \text{ do} \\
    & \quad \text{postorder}(w) \\
    & \quad n_v \leftarrow n_v + n_w
\end{align*}
\]

size of the subtree $T(v)$

\[
\begin{align*}
\text{preorder(vertex } v, t, \alpha_{\text{min}}, \alpha_{\text{max}}) & \\
    & d_v \leftarrow \rho_t \\
    & \alpha_v \leftarrow (\alpha_{\text{min}} + \alpha_{\text{max}})/2 \\
    & \text{if } t > 0 \text{ then} \\
    & \quad \alpha_{\text{min}} \leftarrow \max\{\alpha_{\text{min}}, \alpha_v - \arccos \frac{\rho_t}{\rho_{t+1}} \} \\
    & \quad \alpha_{\text{max}} \leftarrow \min\{\alpha_{\text{max}}, \alpha_v + \arccos \frac{\rho_t}{\rho_{t+1}} \} \\
    & \quad \text{left} \leftarrow \alpha_{\text{min}} \\
    & \quad \text{foreach child } w \text{ of } v \text{ do} \\
    & \quad \quad \text{right} \leftarrow \text{left} + \frac{n_w}{n_v - 1} \cdot (\alpha_{\text{max}} - \alpha_{\text{min}}) \\
    & \quad \quad \text{preorder}(w, t + 1, \text{left}, \text{right}) \\
    & \quad \quad \text{left} \leftarrow \text{right}
\end{align*}
\]
Pseudocode for radial tree layout

RadialTreeLayout(tree $T$, root $r \in T$, radii $\rho_1 < \cdots < \rho_k$)

begin
    postorder($r$)
    preorder($r$, 0, 0, $2\pi$)
    return $(d_v, \alpha_v)_{v \in V(T)}$
end

{vertex pos./ polar coord.}

postorder(vertex $v$)

$n_v \leftarrow 1$

foreach child $w$ of $v$ do
    postorder($w$)

$n_v \leftarrow n_v + n_w$

size of the subtree $T(v)$

preorder(vertex $v$, $t$, $\alpha_{\text{min}}$, $\alpha_{\text{max}}$)

d_v \leftarrow \rho_t

$\alpha_v \leftarrow (\alpha_{\text{min}} + \alpha_{\text{max}})/2$

if $t > 0$ then
    $\alpha_{\text{min}} \leftarrow \max\{\alpha_{\text{min}}, \alpha_v - \arccos \frac{\rho_t}{\rho_{t+1}} \}$

    $\alpha_{\text{max}} \leftarrow \min\{\alpha_{\text{max}}, \alpha_v + \arccos \frac{\rho_t}{\rho_{t+1}} \}$

left $\leftarrow \alpha_{\text{min}}$

foreach child $w$ of $v$ do
    right $\leftarrow$ left $+ \frac{n_w}{n_v-1}(\alpha_{\text{max}} - \alpha_{\text{min}})$

preorder($w$, $t + 1$, left, right)

left $\leftarrow$ right
Pseudocode for radial tree layout

RadialTreeLayout(tree $T$, root $r \in T$, radii $\rho_1 < \cdots < \rho_k$)

begin
  postorder($r$)
  preorder($r$, 0, 0, $2\pi$)
  return $(d_v, \alpha_v)v \in V(T)$
end

{vertex pos./ polar coord.}

postorder(vertex $v$)

$n_v \leftarrow 1$

foreach child $w$ of $v$ do
  postorder($w$)
  $n_v \leftarrow n_v + n_w$

{output}

preorder(vertex $v$, $t$, $\alpha_{\text{min}}$, $\alpha_{\text{max}}$)

$d_v \leftarrow \rho_t$

$\alpha_v \leftarrow (\alpha_{\text{min}} + \alpha_{\text{max}})/2$

if $t > 0$ then
  $\alpha_{\text{min}} \leftarrow$
  max{$\alpha_{\text{min}}$, $\alpha_v - \arccos \frac{\rho_t}{\rho_{t+1}}$}

  $\alpha_{\text{max}} \leftarrow$
  min{$\alpha_{\text{max}}$, $\alpha_v + \arccos \frac{\rho_t}{\rho_{t+1}}$}

  $\leftarrow \alpha_{\text{min}}$

  foreach child $w$ of $v$ do
    right $\leftarrow$ left $+ \frac{n_w}{n_v-1} \cdot (\alpha_{\text{max}} - \alpha_{\text{min}})$
  preorder($w$, $t + 1$, left, right)
  left $\leftarrow$ right

{size of the subtree $T(v)$}
Pseudocode for radial tree layout

RadialTreeLayout(tree $T$, root $r \in T$, radii $\rho_1 < \cdots < \rho_k$)

begin

postorder$(r)$
preorder$(r, 0, 0, 2\pi)$
return $(d_v, \alpha_v)_{v \in V(T)}$

{vertex pos./ polar coord.}

postorder(vertex $v$)

$n_v \leftarrow 1$

foreach child $w$ of $v$ do

postorder$(w)$

$n_v \leftarrow n_v + n_w$

{output}

preorder(vertex $v$, $t$, $\alpha_{\min}$, $\alpha_{\max}$)

d_v \leftarrow \rho_t
\alpha_v \leftarrow (\alpha_{\min} + \alpha_{\max})/2

if $t > 0$ then

$\alpha_{\min} \leftarrow$

$\max\{\alpha_{\min}, \alpha_v - \arccos \frac{\rho_t}{\rho_{t+1}}\}$

$\alpha_{\max} \leftarrow$

$\min\{\alpha_{\max}, \alpha_v + \arccos \frac{\rho_t}{\rho_{t+1}}\}$

left $\leftarrow \alpha_{\min}$

foreach child $w$ of $v$ do

right $\leftarrow left + \frac{n_w}{n_v - 1} \cdot (\alpha_{\max} - \alpha_{\min})$

preorder$(w, t + 1, left, right)$

left $\leftarrow right$

foreach child $w$ of $v$ do

left $\leftarrow \alpha_{\min}$

right $\leftarrow left + \frac{n_w}{n_v - 1} \cdot (\alpha_{\max} - \alpha_{\min})$

preorder$(w, t + 1, left, right)$

left $\leftarrow right$

Runtime?
Pseudocode for radial tree layout

```
RadialTreeLayout(tree T, root r ∈ T, radii ρ1 < · · · < ρk)
begin
postorder(r)
preorder(r, 0, 0, 2π)
return (dv, αv)v∈V(T)
{vertex pos./ polar coord.}
postorder(vertex v)
nv ← 1
foreach child w of v do
  postorder(w)
nv ← nv + nw

size of the subtree T(v)
```

```
preorder(vertex v, t, αmin, αmax)

| dv ← ρt | {output } |
| αv ← (αmin + αmax)/2 |

if t > 0 then
  αmin ← max{αmin, αv − arccos ρt/ρt+1 }
  αmax ← min{αmax, αv + arccos ρt/ρt+1 }
left ← αmin
foreach child w of v do
  right ← left + nw/nv−1 · (αmax − αmin)
  preorder(w, t + 1, left, right)
left ← right
```

```
Runtime? O(n).
```
Pseudocode for radial tree layout

RadialTreeLayout(tree $T$, root $r \in T$, radii $\rho_1 < \cdots < \rho_k$)

\begin{align*}
\text{begin} & \quad \text{postorder}(r) \\
& \quad \text{preorder}(r, 0, 0, 2\pi) \\
& \quad \text{return} \ (d_v, \alpha_v)_{v \in V(T)} \\
& \quad \{ \text{vertex pos./ polar coord.} \}
\end{align*}

\begin{align*}
\text{postorder}(\text{vertex } v) & \quad n_v \leftarrow 1 \\
& \quad \text{foreach child } w \text{ of } v \text{ do} \\
& \quad \quad \text{postorder}(w) \\
& \quad \quad n_v \leftarrow n_v + n_w \\
& \quad \{ \text{size of the subtree } T(v) \}
\end{align*}

\begin{align*}
\text{preorder}(\text{vertex } v, t, \alpha_{\text{min}}, \alpha_{\text{max}}) & \quad d_v \leftarrow \rho_t \\
& \quad \alpha_v \leftarrow (\alpha_{\text{min}} + \alpha_{\text{max}})/2 \\
& \quad \{ \text{output } \}
\end{align*}

\begin{align*}
& \quad \text{if } t > 0 \text{ then} \\
& \quad \quad \alpha_{\text{min}} \leftarrow \max\{\alpha_{\text{min}}, \alpha_v - \arccos \frac{\rho_t}{\rho_{t+1}} \} \\
& \quad \quad \alpha_{\text{max}} \leftarrow \min\{\alpha_{\text{max}}, \alpha_v + \arccos \frac{\rho_t}{\rho_{t+1}} \} \\
& \quad \quad \leftarrow \alpha_{\text{min}} \\
& \quad \quad \text{foreach child } w \text{ of } v \text{ do} \\
& \quad \quad \quad \text{right} \leftarrow \text{left} + \frac{n_w}{n_v-1} \cdot (\alpha_{\text{max}} - \alpha_{\text{min}}) \\
& \quad \quad \quad \text{preorder}(w, t + 1, \text{left}, \text{right}) \\
& \quad \quad \text{left} \leftarrow \text{right}
\end{align*}

\text{Runtime? } O(n). \quad \text{Correctness?}
Pseudocode for radial tree layout

RadialTreeLayout(tree $T$, root $r \in T$, radii $\rho_1 < \cdots < \rho_k$)

begin
  postorder($r$)
  preorder($r$, 0, 0, $2\pi$)
  return $(d_v, \alpha_v)_{v \in V(T)}$

  {vertex pos./ polar coord.}

postorder(vertex $v$)

  $n_v \leftarrow 1$
  foreach child $w$ of $v$ do
    postorder($w$)
    $n_v \leftarrow n_v + n_w$

preorder(vertex $v$, $t$, $\alpha_{\text{min}}$, $\alpha_{\text{max}}$)

  $d_v \leftarrow \rho_t$
  $\alpha_v \leftarrow (\alpha_{\text{min}} + \alpha_{\text{max}})/2$

  {output}

  if $t > 0$ then
    $\alpha_{\text{min}} \leftarrow$
    \[\max\{\alpha_{\text{min}}, \alpha_v - \arccos \frac{\rho_t}{\rho_{t+1}}\}\]
    $\alpha_{\text{max}} \leftarrow$
    \[\min\{\alpha_{\text{max}}, \alpha_v + \arccos \frac{\rho_t}{\rho_{t+1}}\}\]

  left $\leftarrow \alpha_{\text{min}}$
  foreach child $w$ of $v$ do
    right $\leftarrow$ left + $\frac{n_w}{n_v-1} \cdot (\alpha_{\text{max}} - \alpha_{\text{min}})$
    preorder($w$, $t + 1$, left, right)
  left $\leftarrow$ right

size of the subtree $T(v)$

Runtime? $O(n)$. Correctness? ✓
Overview

- balanced drawings of binary trees $O(nh)$
- radial drawings of trees $O(nh)$
- compact drawings of trees $O(n \log n)$
- upward drawings of series parallel graphs exp.
Definition.

An *hv-drawing* of a binary tree is a straight line drawing, so that for each vertex $v$: 
**Definition.**

An *hv-drawing* of a binary tree is a straight line drawing, so that for each vertex $v$:

- each child of $v$ is either directly right or directly below $v$. 

![Diagram of an hv-drawing](image)
Definition.

An *hv-drawing* of a binary tree is a straight line drawing, so that for each vertex $v$:

- each child of $v$ is either directly right or directly below $v$.
- the smallest axis-parallel rectangle enclosing the subtrees of the children of $v$ are disjoint.
Definition.

An *hv-drawing* of a binary tree is a straight line drawing, so that for each vertex \( v \):

- each child of \( v \) is either directly right or directly below \( v \).
- the smallest axis-parallel rectangle enclosing the subtrees of the children of \( v \) are disjoint.

*horizontal combination*
hv-Drawings

Definition.
An hv-drawing of a binary tree is a straight line drawing, so that for each vertex $v$:

- each child of $v$ is either directly right or directly below $v$.
- the smallest axis-parallel rectangle enclosing the subtrees of the children of $v$ are disjoint.

horizontal combination  vertical combination
Definition.

An *hv-drawing* of a binary tree is a straight line drawing, so that for each vertex $v$:

- each child of $v$ is either directly right or directly below $v$.
- the smallest axis-parallel rectangle enclosing the subtrees of the children of $v$ are disjoint.

*horizontal combination*  
*vertical combination*
Algorithm \textit{RightHeavyHVTreeDraw}

- Recursively construct drawings of the left and right subtrees from the root.
Algorithm \textit{RightHeavyHVTreeDraw}

- Recursively construct drawings of the left and right subtrees from the root.
- Place the larger subtree on the right using the horizontal combination, and the smaller on the left.
Algorithm *RightHeavyHVTTreeDraw*

- Recursively construct drawings of the left and right subtrees from the root.

- Place the larger subtree on the right using the horizontal combination, and the smaller on the left

Size of a subtree := number of vertices
Algorithm *RightHeavyHVTreeDraw*

- Recursively construct drawings of the left and right subtrees from the root.
- Place the larger subtree on the right using the horizontal combination, and the smaller on the left

\[\text{Size of a subtree} \triangleq \text{number of vertices}\]
Algorithm RightHeavyHVTreeDraw

- Recursively construct drawings of the left and right subtrees from the root.
- Place the larger subtree on the right using the horizontal combination, and the smaller on the left

Size of a subtree := number of vertices
Algorithm *RightHeavyHVTreeDraw*

- Recursively construct drawings of the left and right subtrees from the root.
- Place the larger subtree on the right using the horizontal combination, and the smaller on the left.

Size of a subtree := number of vertices
Algorithm \textit{RightHeavyHVTreeDraw}

- Recursively construct drawings of the left and right subtrees from the root.
- Place the larger subtree on the right using the horizontal combination, and the smaller on the left

\text{Size of a subtree} := \text{number of vertices}
Algorithm \textit{RightHeavyHVTreeDraw}

- Recursively construct drawings of the left and right subtrees from the root.
- Place the larger subtree on the right using the horizontal combination, and the smaller on the left.

Size of a subtree := number of vertices
Algorithm \textit{RightHeavyHVTTreeDraw}

- Recursively construct drawings of the left and right subtrees from the root.
- Place the larger subtree on the right using the horizontal combination, and the smaller on the left.

Size of a subtree := number of vertices
Algorithm `RightHeavyHVTreeDraw`

- Recursively construct drawings of the left and right subtrees from the root.
- Place the larger subtree on the right using the horizontal combination, and the smaller on the left.

Size of a subtree := number of vertices

Obs. The drawing has width $\leq$
Algorithm *RightHeavyHVTreeDraw*

- Recursively construct drawings of the left and right subtrees from the root.
- Place the larger subtree on the right using the horizontal combination, and the smaller on the left.

*Size of a subtree := number of vertices*

*Obs.* The drawing has width $\leq n$, height $\leq \ldots$
Algorithm *RightHeavyHVTreeDraw*

- Recursively construct drawings of the left and right subtrees from the root.
- Place the larger subtree on the right using the horizontal combination, and the smaller on the left.

Size of a subtree := number of vertices

Obs. The drawing has width $\leq n$, height $\leq$
Algorithm \textit{RightHeavyHVTreeDraw}

- Recursively construct drawings of the left and right subtrees from the root.
- Place the larger subtree on the right using the horizontal combination, and the smaller on the left.

Size of a subtree := number of vertices

\textbf{Obs.} The drawing has width \( \leq n \), height \( \leq \)
Algorithm *RightHeavyHVTreeDraw*

- Recursively construct drawings of the left and right subtrees from the root.
- Place the larger subtree on the right using the horizontal combination, and the smaller on the left

Size of a subtree := number of vertices

**Obs.** The drawing has width $\leq n$, height $\leq$
Algorithm *RightHeavyHVTreeDraw*

- Recursively construct drawings of the left and right subtrees from the root.
- Place the larger subtree on the right using the horizontal combination, and the smaller on the left

Size of a subtree \(:=\) number of vertices

Obs. The drawing has width \(\leq n\), height \(\leq\) at least \(\cdot 2\)
Algorithm \textit{RightHeavyHVTreeDraw}

- Recursively construct drawings of the left and right subtrees from the root.
- Place the larger subtree on the right using the horizontal combination, and the smaller on the left

\text{Size of a subtree} := \text{number of vertices}

\textbf{Obs.} The drawing has width $\leq n$, height $\leq$
Algorithm *RightHeavyHVTreeDraw*

- Recursively construct drawings of the left and right subtrees from the root.
- Place the larger subtree on the right using the horizontal combination, and the smaller on the left.

Size of a subtree := number of vertices

*Obs.* The drawing has width $\leq n$, height $\leq$
Algorithm RightHeavyHVTreeDraw

- Recursively construct drawings of the left and right subtrees from the root.
- Place the larger subtree on the right using the horizontal combination, and the smaller on the left

Size of a subtree := number of vertices

Obs. The drawing has width $\leq n$, height $\leq$
Algorithm *RightHeavyHVTreeDraw*

- Recursively construct drawings of the left and right subtrees from the root.
- Place the larger subtree on the right using the horizontal combination, and the smaller on the left.

Size of a subtree := number of vertices

The drawing has width \( \leq n \), height \( \leq \)

**Obs.** The drawing has width \( \leq n \), height \( \leq \)
Algorithm *RightHeavyHVTreeDraw*

- Recursively construct drawings of the left and right subtrees from the root.
- Place the larger subtree on the right using the horizontal combination, and the smaller on the left.

Size of a subtree := number of vertices

Obs. The drawing has width ≤ \(n\), height ≤ \(n\)
Algorithm RightHeavyHVTreeDraw

- Recursively construct drawings of the left and right subtrees from the root.
- Place the larger subtree on the right using the horizontal combination, and the smaller on the left

Size of a subtree := number of vertices

Obs. The drawing has width \( \leq n \), height \( \leq \log_2 n \).
Overview

- balanced drawings of binary trees \( O(nh) \)
- radial drawings of trees \( O(nh) \)
- compact drawings of trees \( O(n \log n) \)
- upward drawings of series parallel graphs \( \text{exp.} \)

Grid Size

- \( O(nh) \)
- \( O(n \log n) \)
- \( \text{exp.} \)
Series Parallel Graphs

- simple series parallel graph
Series Parallel Graphs

- simple series parallel graph

- Induction: combining two series parallel graphs $G_1, G_2 \ldots$
Series Parallel Graphs

- simple series parallel graph

- Induction: combining two series parallel graphs $G_1, G_2 \ldots$

- \ldots series \ldots

\[
t_1 = s_2
\]
Series Parallel Graphs

• simple series parallel graph

• Induction: combining two series parallel graphs $G_1, G_2 \ldots$

- ... series ...

- ... or parallel.
Decomposition Tree for SP-graphs
Decomposition Tree for SP-graphs
Decomposition Tree for SP-graphs
Decomposition Tree for SP-graphs
Decomposition Tree for SP-graphs
Decomposition Tree for SP-graphs
Decomposition Tree for SP-graphs
Decomposition Tree for SP-graphs
Decomposition Tree for SP-graphs
Decomposition Tree for SP-graphs
Decomposition Tree for SP-graphs

Generalization: SPQR-Tree
SP-Graphs: applications

Flow Charts

PERT-Diagrams
(Program Evaluation and Review Technique)
SP-Graphs: applications

Flow Charts

Permits:

- Linear time algorithms for NP-complete problems (e.g., Maximum Independent Set)
Theorem [Bertolazzi et al. ’92]

There is a family \((G_n)_{n \in \mathbb{N}}\) of embedded SP-graphs where \(G_n\) has \(2^n\) vertices and every upward planar drawing of \(G_n\) requires \(\Omega(4^n)\) area.
Theorem [Bertolazzi et al. ’92]

There is a family \((G_n)_{n \in \mathbb{N}}\) of embedded SP-graphs where \(G_n\) has \(2^n\) vertices and every upward planar drawing of \(G_n\) requires \(\Omega(4^n)\) area.

Proof:

\[
\begin{align*}
G_0: & \quad t_0 \quad s_0 \\
G_{n+1}: & \quad t_{n+1} \quad s_{n+1} \\
G_n: & \quad t_n \quad s_n
\end{align*}
\]
Grid Size

**Theorem** [Bertolazzi et al. ’92]

There is a family $(G_n)_{n \in \mathbb{N}}$ of embedded SP-graphs where $G_n$ has $2^n$ vertices and every *upward planar drawing* of $G_n$ requires $\Omega(4^n)$ area.

**Proof:**

\[ \begin{align*}
G_0 & \quad G_{n+1} \\
\bullet \ s_0 & \quad \bullet \ s_{n+1}
\end{align*} \]

\[ \begin{align*}
\bullet \ t_0 & \quad \bullet \ t_n & \quad \bullet \ t_{n+1}
\end{align*} \]

\[ \begin{align*}
\bullet \ s_n & \quad G_n & \quad \bullet \ s_{n-1}
\end{align*} \]
There is a family \( (G_n)_{n \in \mathbb{N}} \) of embedded SP-graphs where \( G_n \) has \( 2^n \) vertices and every \textit{upward planar drawing} of \( G_n \) requires \( \Omega(4^n) \) area.

**Proof:**

The diagram illustrates the construction of the sequence of graphs \( G_0, G_1, \ldots \) where each \( G_n \) is built upon \( G_{n-1} \) by adding new vertices \( s_n, t_n, t_{n+1} \) and connecting them appropriately.
Grid Size

Theorem [Bertolazzi et al. ‘92]
There is a family $\left( G_n \right)_{n \in \mathbb{N}}$ of embedded SP-graphs where $G_n$ has $2^n$ vertices and every *upward planar drawing* of $G_n$ requires $\Omega(4^n)$ area.

Proof:

$G_0 \quad t_0 \quad s_0$

$G_n \quad t_n \quad s_n$

$G_{n+1} \quad t_{n+1} \quad s_{n+1}$
Theorem [Bertolazzi et al. ’92]

There is a family $(G_n)_{n \in \mathbb{N}}$ of embedded SP-graphs where $G_n$ has $2^n$ vertices and every upward planar drawing of $G_n$ requires $\Omega(4^n)$ area.

Proof:

$$G_0 \quad G_{n+1}$$

$G_n$
Grid Size

**Theorem** [Bertolazzi et al. ’92]

There is a family \( (G_n)_{n \in \mathbb{N}} \) of embedded SP-graphs where \( G_n \) has \( 2^n \) vertices and every **upward planar drawing** of \( G_n \) requires \( \Omega(4^n) \) area.

**Proof:**

- **Diagram:**
  - \( G_0 \) and \( G_{n+1} \) are shown with \( s_0 \) to \( s_n \) and \( t_n \) to \( t_{n+1} \) vertices.
  - The upward planar drawing requires \( \Omega(4^n) \) area.

---

Steven Chaplick · Lehrstuhl für Informatik I · Universität Würzburg
Theorem [Bertolazzi et al. ’92]

There is a family \((G_n)_{n \in \mathbb{N}}\) of embedded SP-graphs where \(G_n\) has \(2n\) vertices and every upward planar drawing of \(G_n\) requires \(\Omega(4^n)\) area.

Proof:
Grid Size

**Theorem** [Bertolazzi et al. ’92]

There is a family \((G_n)_{n \in \mathbb{N}}\) of embedded SP-graphs where \(G_n\) has \(2^n\) vertices and every *upward planar drawing* of \(G_n\) requires \(\Omega(4^n)\) area.

**Proof:**

\[
a(G_{n+1}) \geq a(\Pi) + a(\Delta_1) + a(\Delta_2)
\]
There is a family \((G_n)_{n \in \mathbb{N}}\) of embedded SP-graphs where \(G_n\) has \(2^n\) vertices and every upward planar drawing of \(G_n\) requires \(\Omega(4^n)\) area.

**Proof:**

\[
a(G_{n+1}) \geq a(\Pi) + a(\Delta_1) + a(\Delta_2)
\]
Grid Size

**Theorem** [Bertolazzi et al. '92]

There is a family \((G_n)_{n \in \mathbb{N}}\) of embedded SP-graphs where \(G_n\) has \(2^n\) vertices and every *upward planar drawing* of \(G_n\) requires \(\Omega(4^n)\) area.

**Proof:**

\[
a(G_{n+1}) \geq a(\Pi) + a(\Delta_1) + a(\Delta_2) \geq 2 \cdot a(\Pi)
\]
Grid Size

**Theorem** [Bertolazzi et al. '92]

There is a family \((G_n)_{n \in \mathbb{N}}\) of embedded SP-graphs where \(G_n\) has \(2^n\) vertices and every *upward planar drawing* of \(G_n\) requires \(\Omega(4^n)\) area.

**Proof:**

\[
a(G_{n+1}) \geq a(\Pi) + a(\Delta_1) + a(\Delta_2) \geq 2 \cdot a(\Pi) \geq 4 \cdot a(G_n)
\]