

11 Trees

11.1 Introduction to Trees

Def 1 A tree is a connected undirected graph with no simple circuits.

forest a set of trees.

Thm 1 An undirected graph is a tree, if and only if,
there is a **unique simple path** between **any** two of its vertices.

Def 1 A tree is an acyclic connected graph.

Defn 2 A rooted tree is a tree one vertex has designated as the **root**
and every edge is **directed away** from the root.

parent, children, siblings

ancestor, decendents

leaves, internal vertices

subtree

Recursive definition of rooted tree(see 5.3 Def. 3) in digraph

Definition 3 A rooted tree is called ***n*-ary tree**,
if every (**internal**) vertex has **no more than *n* children**.
The tree is called **full *n*-ary tree**,
if every **internal**(non-leaf) vertex has **exactly *n* children**.
A 2-ary tree is call **binary tree**.

An ordered rooted tree
children are ordered from left to right
left/right child
left/right subtree

Computer file system
Hierarchical structure
Family tree

Properties of Trees

Theorem 2 A tree with n vertices has $n-1$ edges.
minimal connections of vertices

Theorem 3 A full m -ary tree with i internal vertices has $n = mi+1$ nodes
 and $l = (m-1)i + 1$ leaves.

proof Every vertices except **root** is a child of internal vertices.

$\therefore mi + 1$ nodes.

Since $n = i + l$, $l = (m-1)i + 1$ leaves.

Theorem 4 A full m -ary tree with
 Consider a full m -ary tree.

$$e = n-1 \qquad n = i + l \qquad n = mi+1$$

Four variable e, i, n, l . If you know any one of them you can compute the other three.

See **Theorem 4** and note that $e = n-1$.

The **level** of a **node** is the **length** of the **path** from **root** to the node.

The **height** of a **tree** is **maximum** node level.

The m -ary tree with height h is called **balanced**,
if every **leaves** are **level** h or $h-1$.

Theorem 5 There are at most m^h leaves in m -ary tree of **height** h .

proof Consider a **full, balanced** m -ary tree
and every leaves are of level h .

It has m^h leaves.

Corollary 1 If an m -ary tree with l leaves has height h , then $h \geq \lceil \log_m l \rceil$.

If the tree is **full** and **balanced** $h = \lceil \log_m l \rceil$.

11.2 Application of Trees

Binary Search Tree

procedure search&insert($T = (V, E, r)$: binary Tree, x : item)

if $r \neq \text{null} \rightarrow v = r$;

do $v \neq \text{null} \wedge \text{label}(v) \neq x \rightarrow$

if $x < \text{label}(v) \rightarrow$

if $\text{left child}(v) \neq \text{null} \rightarrow v := \text{left child}(v)$

| $\text{left child}(v) = \text{null} \rightarrow \text{add new vertex } v(\text{label}(x)) \text{ as a left child}$

fi

| $x > \text{label}(v) \rightarrow$

if $\text{right child}(v) \neq \text{null} \rightarrow v := \text{right child}(v)$

| $\text{right child}(v) = \text{null} \rightarrow \text{add new vertex } v(\text{label}(x)) \text{ as a right child}$

od fi fi

| $r = \text{null} \rightarrow \text{add new vertex } v(\text{label}(x)) \text{ as root of the tree}$

fi

label(v) = x .

procedure *search&insert*($T = (V, E, r)$: *binary Tree*, x : *item*)

$v := r$;

if $v \neq \text{null}$ \rightarrow

if $x < \text{label}(v)$ \rightarrow **return** *search&insert*(*leftSubtree*(T), x)

 | $x > \text{label}(v)$ \rightarrow **return** *search&insert*(*rightSubtree*(T), x)

 | $x = \text{label}(v)$ \rightarrow **return** “*arleady present*”

fi

 | $v = \text{null}$ \rightarrow

$r :=$ *add a new vertex with label* x ;

return “*Not present and inserted*”

fi

Decision Trees

*One counterfeit coin in n coins
balance*

Three groups of $\lceil n/3 \rceil$ coins

if first group < second group \rightarrow counterfeit in the first group

| first group > second group \rightarrow counterfeit in the second group

| first group = second group \rightarrow counterfeit in the third group

fi

$\lceil \log_3 n \rceil$ comparisons

The complexity of sorting algorithms

n elements

n! permutations

binary comparisons

n! leaves in the binary decision tree

Height of the decision tree is $\log n!$

$O(n \log_2 n)$

*The **lower bound** for sorting based on binary search is $\Omega(n \log_2 n)$*

*The **average bound** for sorting based on binary search is $\Theta(n \log_2 n)$*

Prefix code

26 letters 5 bits ($2^4 < 26 \leq 2^5$)

But more **frequent** letters have **shorter** sequence.

e 00

t 01

w 0001

0001 *w* or *et*

e is a prefix of *w*

A set of sequences (codes) is a **prefix code**, if no sequence in the set is the **prefix** of other sequences in the set.

Full binary tree \leftrightarrow prefix code

Algorithm 2 Huffman coding

procedure *Huffman*(V : set of symbols, $w: V \rightarrow \mathbf{R}^+$ is a weigh of a symbol)

$F :=$ **for** $a \in V$ **do** make a **tree** with symbol a and weight $w(a)$ **od**

do F is not a tree \rightarrow

 Find two **minimun** weighted trees T_1 and T_2 ;

 Make a **new tree** T with weight $w(T) = w(T_1) + w(T_2)$

 and T_1 and T_2 as **subtrees**.

od

<i>internal vertices</i>	<i>weight of the subtree</i>
<i>leaf</i>	<i>symbol</i>

11.3 Tree Traversal

Universal Address System in a Rooted Ordered Tree

Label root as ε .

If Subtree T is labeled as x and n children

then n children of x are labeled as $x.1, x.2, \dots, x.n$.

We may use 0 instead of ε .

We may use n instead of $\varepsilon.n = .n$.

Label root as 1 .

If Subtree T is labeled as x and n children

then n children of x are labeled as $x.1, x.2, \dots, x.n$.

Prefix code

Lexicographic ordering

$x_1.x_2.\dots.x_n < y_1.y_2.\dots.y_m$, if $\exists i, 0 \leq i \leq n, 1 \leq \forall j < i, x_j = y_j, x_i < y_i$ or
 $n < m, 1 \leq \forall j \leq n, x_j = y_j$.

procedure preorder(T with root r)

visit r; preorder(left child(r)); preorder(right child(r))

end preorder

procedure inorder(T with root r)

inorder(left child(r)) visit r; inorder(right child(r))

end inorder

procedure postorder(T with root r)

posrorder(left child(r)) postorder(right child(r)); visit(r)

end postorder

Infix, Prefix(Polish), and Postfix(Reverse Polish) notation

11.4 Spanning Trees

Definition 1 Let G be a ugraph. A **spanning tree** of G is a subgraph of G that is tree containing every vertex of G .

Algorithm 1 Depth-First Search

procedure DFS($G = (V, E)$: connected graph)

$T :=$ Tree with vertex $v \in V$ only;

 visit(v);

end DFS;

procedure visit($v \in V$)

for $\{v, w\} \in E \wedge w \notin T$ **do** add v and $\{v, w\}$ to T ; visit(w) **od**

end visit

Algorithm 2 Breadth-First Search

procedure *BFS*($G = (V, E)$: connected graph)

$T :=$ Tree with vertex $v \in V$ only; $L :=$ list with vertex $v \in V$;

do L is not empty \rightarrow

 remove the first vertex, v , from L ;

for $\{v, w\} \in E \wedge w \notin T \wedge w \notin L$ **do**

 add v and $\{v, w\}$ to T ; add w to the end of L ;

end *BFS*;

11.5 Minimum Spanning Trees

Definition 1 A *minimum spanning tree* in a connected weighted graph is a spanning tree that has the *smallest sum of weight of its edges*.

Algorithm 1 Prim's Algorithm

procedure Prim($G = (V, E)$: connected weighted graph)

$T :=$ a minimum weighted edge;

for $i := 1$ to $|V| - 2$ **do**

$e :=$ a *minimum* weighted edge *incident* to a vertex in T
and **not** forming a simple circuit in T , if added to T ;

$T := T$ with e added

od

end Prim

Algorithm 2 Kruskal's Algorithm

procedure *Kruskal*($G = (V, E)$: connected weighted graph)

$T :=$ empty graph;

for $i := 1$ to $|V| - 1$ **do**

$e :=$ any edge in G with **smallest** weight

that does **not** form a simple circuit in T , if added to T ;

$T := T$ with e added

od