



# TOPICS IN ALGORITHMS

## ASSIGNMENT REPORT

---

*Vertex Cover on Trees (Dynamic Programming Approach)*

---

Prepared By :->

RISHAD C [B160346CS]

Instructor: DR. SUBASHINI

COURSE: CS4027

DATE: 25 / 04 /2019

# VERTEX COVER ON TREES

## **Problem:**

Given a graph which is a tree and we want to find the minimum no. of vertices needed to cover all edges in it.

## **Input:**

A tree represented in parenthesis form is given as input.

## **Output:**

Minimum no. of vertices need to cover whole tree is displayed.

## **Abbreviations used:**

VC : Vertex Cover

min : minimum

minVC : minimum Vertex Cover

## i.ALGORITHM(with DP)

```
//program start here
```

```
main()
```

```
{
```

```
//taking user input
```

```
1 Input >> tree;
```

```
//constructing tree structure and returning root vertex
```

```
2 rootTree = construct(tree);
```

```
//function VCTree calculate min vertex cover and returns
```

```
3 vc = VCTree(rootTree);
```

```
//displays min Vertex cover
```

```
4 Output << vc;
```

```
}
```

```
//maintaining 1D array for memoization purpose
```

```
DP[MAX];
```

```
//VCTree Function
```

```
VCTree ( v )
```

```
{
```

```
/* base condition {if no children(i.e leaf) , then minVC of tree rooted @ v return as ZERO}*/
```

```
1 if (v->childCount == 0)
```

```
2 return 0;
```

```
//return minVC value if minVC of this vertex already calculated
```

```
3 If (DP[ v ] != NULL)
```

```
4 return DP[v] ;
```

```

//calculating minVC
//consider 2 cases : v in cover , v not in cover
//case1: calculating VC for v in cover
5 Vin = 1 ; //including vertex v in cover
//calculating recursively & including VC of children subtrees(of v) in cover
6 for i=1 to (v->childcount)
7     Vin = Vin + VCTree (v->child[i]) ;

// case2: calculating VC for v not in cover
8 Vout = v->childcount ; //including children vertices(of v) in cover
//calculating recursively & including VC of grandchildren subtrees(of v) in cover
9 for i=1 to (v->childcount)
10     for j=1 to [ ( v->child[i] )->childcount ]
11         Vout = Vout + VCTree ( ( v->child[i] )-> child[j] ) ;

//taking min among the 2 cases
12 vc = min {Vin,Vout} ;
//updating DP array with calculated VC for subtree rooted at v
13 DP[ v ] = vc ;

//returns VC
14 return vc ;
}

```

## ii.ALGORITHM(without DP)

```
//program start here
```

```
main()
```

```
{
```

```
//taking user input
```

```
1 Input >> tree;
```

```
//constructing tree structure and returning root vertex
```

```
2 rootTree = construct(tree);
```

```
//function VCTree calculate min vertex cover and returns
```

```
3 vc = VCTree(rootTree);
```

```
//displays min Vertex cover
```

```
4 Output << vc;
```

```
}
```

```
//VCTree Function
```

```
VCTree ( v )
```

```
{
```

```
/* base condition {if no children(i.e leaf) , then minVC of tree  
rooted @ v is returned as ZERO}*/
```

```
1 if (v->childCount == 0)
```

```
2     return 0;
```

```

//calculating minVC
//consider 2 cases : v in cover , v not in cover
//case1: calculating VC for v in cover
3 Vin = 1 ; //including vertex v in cover
//calculating recursively & including VC of children subtrees(of v) in cover
4 for i=1 to (v->childcount)
5     Vin = Vin + VCTree (v->child[i]) ;

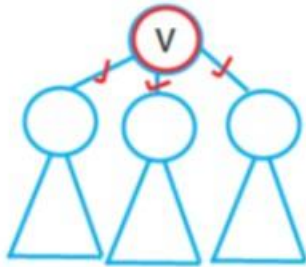
// case2: calculating VC for v not in cover
6 Vout = v->childcount ; //including children vertices(of v) in cover
//calculating recursively & including VC of grandchildren subtrees(of v) in cover
7 for i=1 to (v->childcount)
8     for j=1 to [ ( v->child[i] )->childcount ]
9         Vout = Vout + VCTree ( ( v->child[i] )-> child[j] ) ;

//taking min among the 2 cases
10 vc = min {Vin,Vout} ;
//returns VC
11 return vc ;
}

```

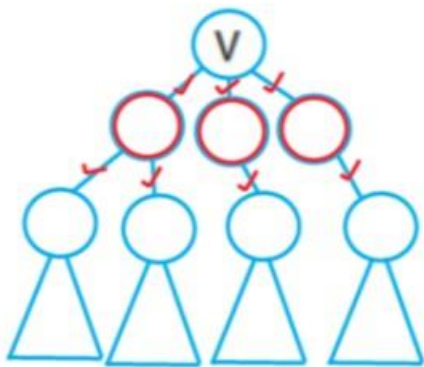
## REPRESENTATION 2 CASES IN ALGORITHM PICTORICALLY

Case 1:  $v$  is in cover



We will include  $v$  in cover and compute VC of children subtrees(of  $v$ ).

Case 2:  $v$  not in cover



We include children of  $v$  in cover and compute VC of grandchildren subtrees(of  $v$ ).

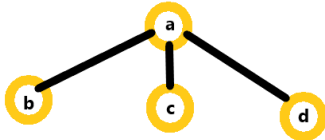
# IMPLIMENTATION

Programming language used: C++

Compiler used: g++ (Ubuntu 5.4.0-6ubuntu1~16.04.6) 5.4.0 20160609

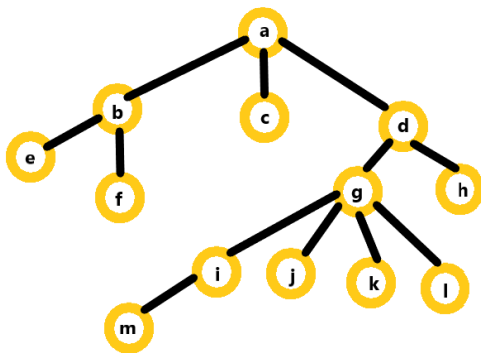
Input format: →

Example 1:



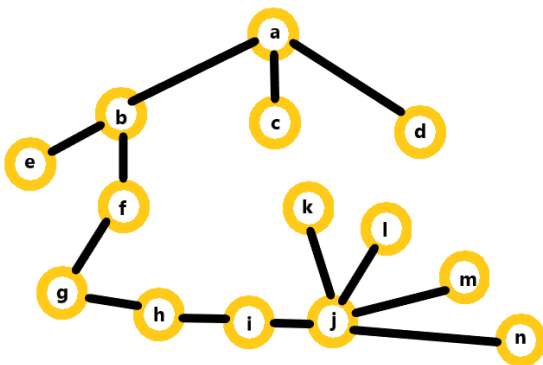
Input should be given as: **a(b)(c)(d)**

Example 2:



Input should be given as: **a(b(e)(f))(c)(d(g(i(m))(j)(k)(l))(h))**

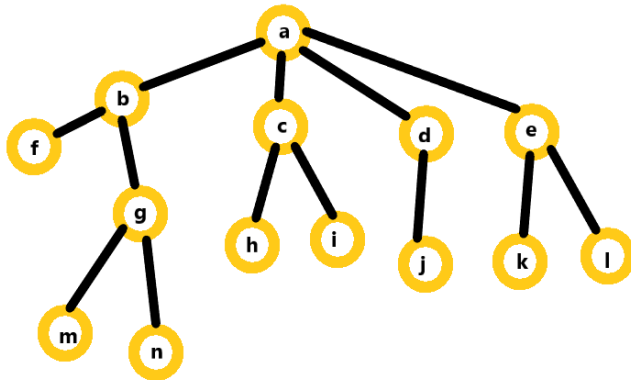
Example 3:



Input should be given as: **a(b(e)(f(g(h(i(j(k)(l)(m)(n)))))))(c)(d)**



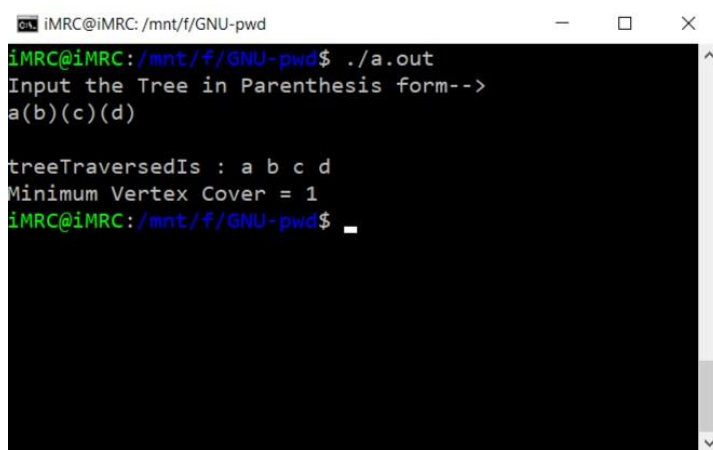
**Example 4:**



Input should be given as: **a(b(f)(g(m)(n)))(c(h)(i))(d(j))(e(k)(l))**

## OUTPUT SCREENSHOTS OF EXAMPLE INPUTS

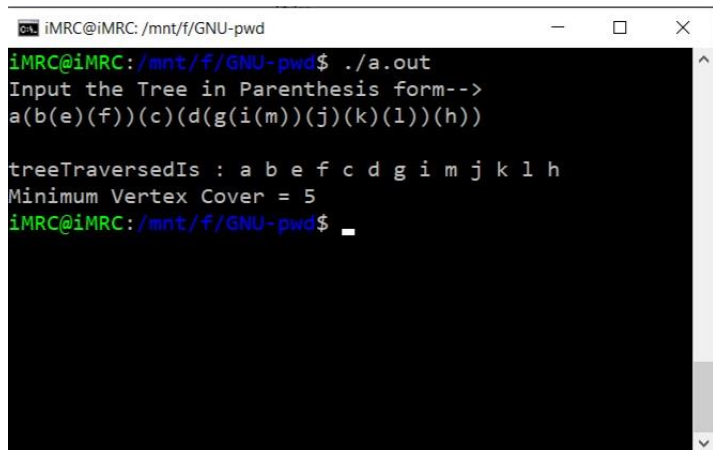
Example 1:

A terminal window titled 'iMRC@iMRC: /mnt/f/GNU-pwd' showing the execution of a program. The user runs './a.out', and the program prompts 'Input the Tree in Parenthesis form-->'. The user enters 'a(b)(c)(d)'. The program outputs 'treeTraversedIs : a b c d' and 'Minimum Vertex Cover = 1'.

```
iMRC@iMRC: /mnt/f/GNU-pwd$ ./a.out
Input the Tree in Parenthesis form-->
a(b)(c)(d)

treeTraversedIs : a b c d
Minimum Vertex Cover = 1
iMRC@iMRC: /mnt/f/GNU-pwd$
```

Example 2:

A terminal window titled 'iMRC@iMRC: /mnt/f/GNU-pwd' showing the execution of a program. The user runs './a.out', and the program prompts 'Input the Tree in Parenthesis form-->'. The user enters 'a(b(e)(f))(c)(d(g(i(m))(j)(k)(l))(h))'. The program outputs 'treeTraversedIs : a b e f c d g i m j k l h' and 'Minimum Vertex Cover = 5'.

```
iMRC@iMRC: /mnt/f/GNU-pwd$ ./a.out
Input the Tree in Parenthesis form-->
a(b(e)(f))(c)(d(g(i(m))(j)(k)(l))(h))

treeTraversedIs : a b e f c d g i m j k l h
Minimum Vertex Cover = 5
iMRC@iMRC: /mnt/f/GNU-pwd$
```

### Example 3:

```
iMRC@iMRC: /mnt/f/GNU-pwd
iMRC@iMRC: /mnt/f/GNU-pwd$ ./a.out
Input the Tree in Parenthesis form-->
a(b(e)(f(g(h(i(j(k)(l)(m)(n)))))))(c)(d)

treeTraversedIs : a b e f g h i j k l m n c d
Minimum Vertex Cover = 5
iMRC@iMRC: /mnt/f/GNU-pwd$
```

### Example 4:

```
iMRC@iMRC: /mnt/f/GNU-pwd
iMRC@iMRC: /mnt/f/GNU-pwd$ ./a.out
Input the Tree in Parenthesis form-->
a(b(f)(g(m)(n)))(c(h)(i))(d(j))(e(k)(l))

treeTraversedIs : a b f g m n c h i d j e k l
Minimum Vertex Cover = 5
iMRC@iMRC: /mnt/f/GNU-pwd$
```

# Dynamic Programming and Correctness

## Sub Problems:

For each  $v$  in  $V$ , size of smallest vertex cover in subtree rooted at  $v$

## base condition:

if no children for  $v$  (i.e  $v$  reached leaf), then minVC returned is ZERO, since it has no children vertices to consider and its participation in VC is computed already on previous subproblem from which this is called, So returns 0.

So base condition is correct

## Overlapping Sub Problem:

Since same subproblems are called again, this problem has Overlapping Subproblem property.

Here there is 2 cases while evaluating vertex cover on a subtree ,i.e include the root vertex in the solution or not include root vertex.

We are evaluating both cases and takes min .So there is chance of Already called subproblem will be called again for evaluation.

## Recurrence:

$$\begin{aligned} \text{VCTree}(v) = \min \{ & \\ & 1 + \sum(\text{VCTree}(c) \text{ for } c \text{ in } v.\text{children}) \\ & , \\ & v.\text{childcount} + \\ & \sum(\text{VCTree}(g) \text{ for } c \text{ in } v.\text{children} \\ & \quad \text{for } g \text{ in } c.\text{children}) \\ & \} \end{aligned}$$

## Proving Recurrence(correctness):

To prove this is correct, consider the optimal solution for  $\text{VCTree}(v)$ . There are two cases: either  $v$  is considered in the solution for  $\text{VCTree}(v)$ , or it is not.

Case 1: If  $v$  is considered for optimal solution for  $\text{VCTree}(v)$ , so  $v$  cover all edges from  $v$  to its children vertices, then smallest vertex cover is by considering  $v$  and the remaining vertex cover of children subtrees i.e.

By definition:  $1 + \sum(\text{VCTree}(c) \text{ for } c \text{ in } v.\text{children})$ .

Case 2: If  $v$  is not considered for optimal solution for  $VCTree(v)$ , then in order to cover edges from  $v$  to its children vertices, all  $v.children$  should be in cover, So this will cover all edges from  $v.children$  to  $v.grandchildren$ , then smallest vertex cover is by considering  $v.children$  and the remaining vertex cover of grand children subtrees

i.e. By definition:  $v.childcount + \sum(VCTree(g) \text{ for } c \text{ in } v.children, \text{for } g \text{ in } c.children)$

Finally, we will get  $VCTree(v)$  in its smallest, the min of these 2 cases is  $VCTree(v)$ .

### → Loop invariant in Case 1:

(consider  $VCTree$  in i.ALGORITHM line 5-7)

Initialization: before first iteration when  $i=1$ ,  $Vin=1$ , i.e. before considering children subtrees,  $Vin$  is 1 because  $v$  is considered for VC, which shows that the loop invariant holds prior to the 1<sup>st</sup> iteration of loop.

Maintenance: the body of for loop works by evaluating  $VCTree(v \rightarrow child[i])$  gets added with  $Vin$ . Incrementing  $i$  for the next iteration of the for loop then preserves the loop invariant.

Termination: the condition causing the for loop to terminate is that  $i > (v \rightarrow childcount) = n$ . because each loop iteration increases  $i$  by 1, we must have  $i=n+1$  at that time. Substituting  $n+1$  for  $i$  in the wording of loop invariant, then we have  $Vin$  consist of  $1 + \text{sum of All min VC of children subtrees}$  and it is the VC of  $v$  in Case 1.

SO for loop in line 3-4, loop invariant is true.

### → Outer Loop invariant in Case 2:

(consider  $VCTree$  in i.ALGORITHM line 8-11)

Initialization: before first iteration when  $i=1$ ,  $Vout=(v \rightarrow childcount)$ , i.e. before considering grandchildren subtrees,  $Vout$  is  $(v \rightarrow childcount)$  because  $v$  is not considered for VC, which shows that the loop invariant holds prior to the 1<sup>st</sup> iteration of loop.

Maintenance: the body of outer for loop works by moving  $VCTree((v \rightarrow child[1]) \rightarrow child[j])$ ,  $VCTree((v \rightarrow child[2]) \rightarrow child[j])$ , and so on which are function calls in the inner for loop. Incrementing  $i$  for the next iteration of the for loop then preserves the loop invariant.

Termination: the condition causing the for loop to terminate is that  $i > (v \rightarrow \text{childcount}) = n$ . because each loop iteration increases  $i$  by 1, we must have  $i = n + 1$  at that time. Substituting  $n + 1$  for  $i$  in the wording of loop invariant, then we have  $V_{out}$  consist of  $(v \rightarrow \text{childcount}) + \text{sum of All min VC of grandchildren subtrees}$  and it is the VC of  $v$  in Case 2. SO for loop in line 8-11, loop invariant is true.

### → Inner Loop invariant in Case 2:

(consider VCTree in i.ALGORITHM line 10-11)

Initialization: before first iteration when  $j = 1$ ,

$V_{out} = (v \rightarrow \text{childcount}) + (\text{sum of min VC of grandchildren subtrees of } 1^{\text{st}} \text{ to } (i-1)^{\text{th}} \text{ children of } v)$ ,

i.e. before considering children subtrees  $V_{out}$  is  $(v \rightarrow \text{childcount}) + (\text{sum of min VC of grandchildren subtrees of } 1^{\text{st}} \text{ to } (i-1)^{\text{th}} \text{ child of } v)$

because  $v$  is not considered for VC, and outer for loop reached only  $i^{\text{th}}$  iteration ( $i^{\text{th}}$  is just started), so only upto  $(i-1)^{\text{th}}$  child's minVC of grandchildren subtree is evaluated. which shows that the loop invariant holds prior to the  $1^{\text{st}}$  iteration of loop.

Maintenance: the body of for loop works by evaluating

VCTree ( (  $v \rightarrow \text{child}[i]$  )  $\rightarrow \text{child}[j]$  ) and gets added with  $V_{out}$ .

Incrementing  $j$  for the next iteration of the for loop then preserves the loop invariant.

Termination: the condition causing the for loop to terminate is that

$j > [ (v \rightarrow \text{child}[i]) \rightarrow \text{childcount} ] = n$ . because each loop iteration increases  $j$  by 1, we must have  $j = n + 1$  at that time. Substituting  $n + 1$  for  $j$  in the wording of loop invariant, then we have  $V_{out}$  consist of

$(v \rightarrow \text{childcount}) + (\text{sum of min VC of grandchildren subtrees of } 1^{\text{st}} \text{ to } i^{\text{th}} \text{ child of } v)$

because  $v$  is not considered for VC, and outer for loop reached only  $i^{\text{th}}$  iteration ( $i^{\text{th}}$  is completed), so only upto  $(i)^{\text{th}}$  child's minVC of grandchildren subtree is evaluated.

It means this evaluates all grandchildren subtrees of  $i^{\text{th}}$  child of  $v$

SO for loop in line 10-11, loop invariant is true.

### DP Optimizations

(consider lines 3-4,13 of VCTree in i.ALGORITHM)

In line 13 , we update DPArray with calculated minVC(which we get by line 12 ) in index as v of subproblem.

In lines 3-4 , we check wheather the subproblem is already evaluated,this is done by checking value in DPArray with index as v of subproblem is NULL or not.

If not NULL ,means already evaluated ,simply returns the minVC from the DPArray.

If NULL ,not evaluated .. to do evaluation algorithms continues from line 5

Hence the i.ALGORITHM is CORRECT.

## ANALYSIS OF ALGORITHM(i & ii)

i.ALGORITHM is with DP

ii.ALGORITHM is with out DP

In both algorithms there is Computation of minVC in 2 ways i.e considering root vertex in VC , not considering root vertex in VC

Time complexity of (algorithm ii ) is Exponential , because it computes same subproblems again and again.

Time complexity of (algorithm i ) is polynomial , because it avoid computation of same subproblems again. It made possible by memoization technique of DP , it stores the solution of subproblems and it simply returns the soln if it is already computed.

c: constant time;

For Algorithm ii (without DP)

Time for n vertices and m max children for each vertex in worst case→

$$T(n)=c + d*mT(n/m) + d*nT(n/m^2)$$

This will result in Exponential time complexity

For Algorithm I (with DP)

$$T(n)=\left\{ \begin{array}{ll} c , & \text{when already computed subproblem} \\ d*mT(n/m) + d*m^2T(n/m^2) , & \text{when not computed subproblem} \end{array} \right\}$$

$$T(n)= c+ mT(n/m);$$



**VCTree ( v )**

```
{
1  if (v->childCount == 0)          --- constant time
2    return 0;
3  If (DP[ v ] != NULL)             ---constant time
4    return DP[v] ;
5  Vin = 1 ;
6  for i=1 to (v->childcount)        ----|V|---
7    Vin = Vin + VCTree (v->child[i]) ;
8  Vout = v->childcount ;
9  for i=1 to (v->childcount)        -----|V|---
10   for j=1 to [ ( v->child[i] )->childcount ]
11     Vout = Vout + VCTree ( ( v->child[i] )-> child[j] ) ;
12  vc = min {Vin,Vout} ;
13  DP[ v ] = vc ;
14  return vc ;
}
```

So DPtime = #subproblems.(time/subproblem)

$$=|V|.O(V) = O(V^2)$$

$O(V)$  because each vertex visited twice as parent and grandparent

SO Algorithm i complexity is  $O(V^2)$



