CSE 421

Dynamic Programming

RNA, Sequence Alignment

Yin Tat Lee

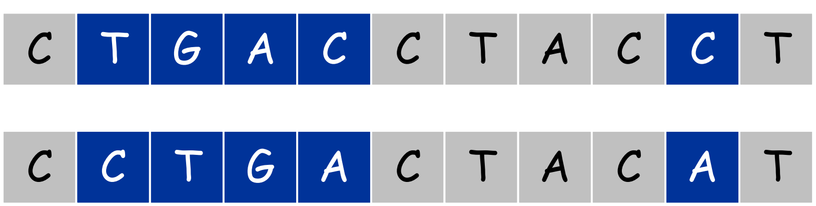
Edit Distance

Edit distance. [Levenshtein 1966, Needleman-Wunsch 1970]

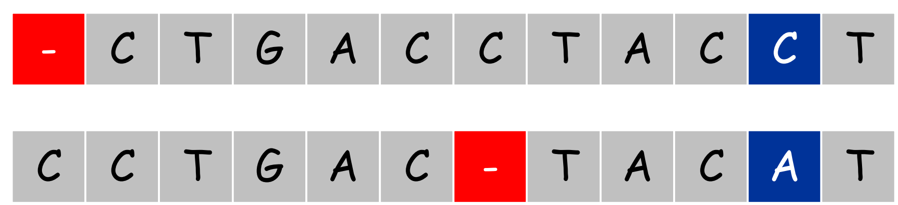
Cost = # of gaps + #mismatches.

Applications.

- Basis for Unix diff and Word correct in editors.
- Speech recognition.
- Computational biology.



Cost: 5



Cost: 3

DP for Sequence Alignment

Let $OPT(i, j)$ be min cost of aligning $x_1, \ldots, x_i$ and $y_1, \ldots, y_j$

Case 1: OPT matches x_i, y_j

- Then, pay mis-match cost if $x_i \neq y_j$ + min cost of aligning $x_1, \ldots, x_{i-1}$ and $y_1, \ldots, y_{j-1}$ i.e., $OPT(i-1, j-1)$

Case 2: OPT leaves x_i unmatched

- Then, pay gap cost for x_i + $OPT(i-1, j)$

Case 3: OPT leaves y_j unmatched

- Then, pay gap cost for y_j + $OPT(i, j-1)$

Bottom-up DP

```
Sequence-Alignment(m, n, x1x2...xm, y1y2...yn) {
   for i = 0 to m
      M[0, i] = i
   for j = 0 to n
      M[j, 0] = j

   for i = 1 to m
      for j = 1 to n
         M[i, j] = min( (xi=yj ? 0:1) + M[i-1, j-1],
                        1 + M[i-1, j],
                        1 + M[i, j-1])
   return M[m, n]
}
```

Analysis: $\Theta(mn)$ time and space.

Computational biology: m = n = 1,000,000. 1000 billions ops OK, but 1TB array?

```
M[i, j] = min( (xi=yj ? 0:1) + M[i-1, j-1],
               1 + M[i-1, j],
               1 + M[i, j-1])
```

Shortest Path

Edit distance is the distance between $(0,0)$ and (m,n) of the following graph.

- All horizontal edges has cost 1.
- All vertical edges has cost 1.
- The cost of edges from $(i-1,j-1)$ to (i,j) is $1_{x_i \neq y_j}$

The graph is a DAG.

Question:
How to recover the alignment
(or how to find the shortest path)
without using mn space?

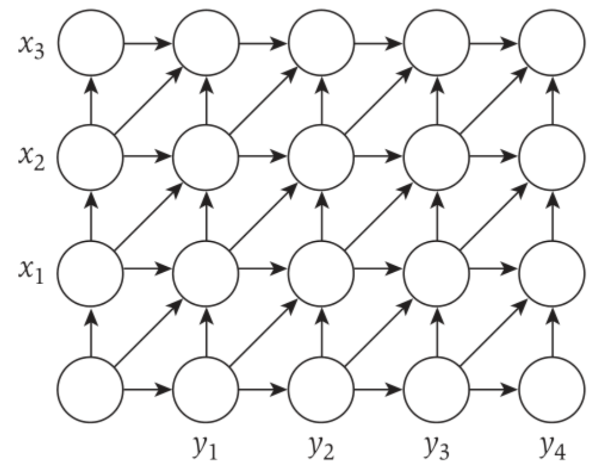
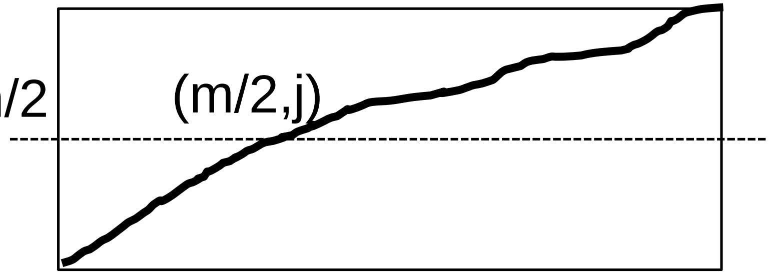


Figure 6.17 A graph-based picture of sequence alignment.

How to recover the alignment?

Idea 1: Suffices to find the point a shortest path pass on the middle row.

Why?
Divide and Conquer!



Idea 2: $d_{(0,0)\to(m,n)} = \min_j d_{(0,0)\to(m/2,j)} + d_{(m/2,j)\to(m,n)}$

```
Find(i₁,j₁,i₂,j₂) { // Due to spacing, ignored boundary cases
    Let k = ⌊(i₁ + i₂)/2⌋
    Compute d_{(i₁,j₁)→(k,j₂)} via Dijkstra at (i₁,j₁).
    Compute d_{(k,j)→(i₂,j₂)} via Dijkstra at (i₂,j₂) on reversed graph.
    Let k = argmin_k d_{(i₁,j₁)→(k,j₂)} + d_{(k,j₂)→(i₂,j₂)}
    p₁ = Find(i₁,j₁,k,j)
    p₂ = Find(k,j,i₂,j₂)

    return p₁,p₂
}
```

Lesson

Advantage of a bottom-up DP:

It is much easier to optimize the space.

By the way, edit distance

- can be computed in $O(s \times \min(m, n))$ if edit distance $\leq s$
- can be computed in $O(\frac{n^2}{\log^2 n})$ (1980).
- can be approximated by log factor in $O(n^{1+\varepsilon})$ (~2010).
- cannot be solved in $O(n^{2-\delta})$ exactly (2015).
- can be approximated by O(1) factor in $O(n^{2-2/7})$ (~2018).
- can be approximated by O(1) factor in $O(n^{1+\epsilon})$ (~2020).

Longest Path in a DAG

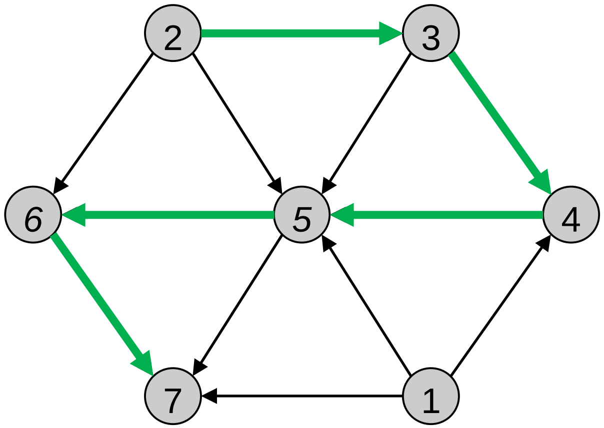
Longest Path in a DAG

Goal: Given a DAG G, find the longest path.

Recall: A directed graph G is a DAG if it has no cycle.

This problem is NP-hard for general directed graphs:

- It has the Hamiltonian Path as a special case



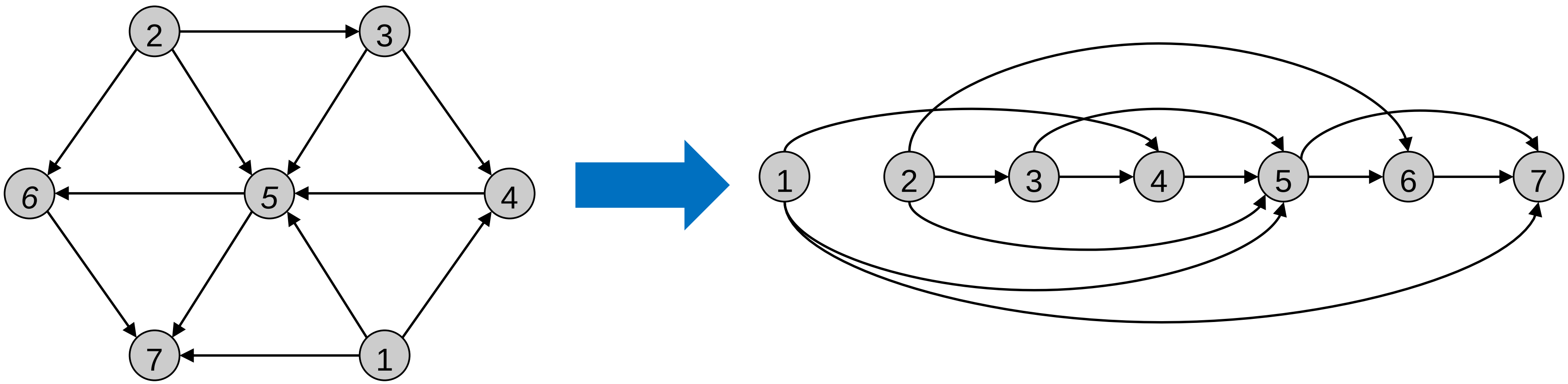
DP for Longest Path in a DAG

Q: What is the right ordering?

Remember, we have to use that G is a DAG, ideally in defining the ordering

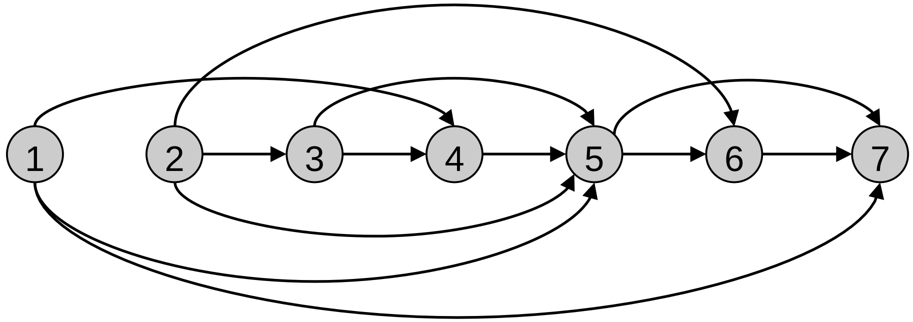
We saw that every DAG has a topological sorting

So, let's use that as an ordering.



DP for Longest Path in a DAG

Suppose we have labelled the vertices such that (i, j) is a directed edge only if $i < j$.



Let $OPT(j)$ = length of the longest path ending at j

Suppose OPT(j) is $(i_1, i_2), (i_2, i_3), \dots, (i_{k-1}, i_k), (i_k, j)$, then

Obs 1: $i_1 \le i_2 \le \cdots \le i_k \le j$.

Obs 2: $(i_1, i_2), (i_2, i_3), \dots, (i_{k-1}, i_k)$ is the longest path ending at i_k.

$$OPT(j) = 1 + OPT(i_k).$$

DP for Longest Path in a DAG

Suppose we have labelled the vertices such that (i, j) is a directed edge only if $i < j$.

Let $OPT(j)$ = length of the longest path ending at j

$$OPT(j) = \begin{cases} 0 & \text{If } j \text{ is a source} \\ 1 + \max_{i:(i,j)\text{ an edge}} OPT(i) & \text{o.w.} \end{cases}$$

Outputting the Longest Path

```
Let G be a DAG given with a topological sorting:
   For all edges (i,j) we have i < j.
Initialize Parent[j]=-1 for all j.
Compute-OPT(j){
   if (in-degree(j) == 0)
     return 0
   if (M[j] == empty)
     M[j] = 0;
     for all edges (i,j)
       if (M[j] < 1+Compute-OPT(i))
         M[j] = 1 + Compute-OPT(i)
         Parent[j] = i
   return M[j]
}
Let k be the maximizer of Compute-OPT(1),…,Compute-OPT(n)
While (Parent[k]!=-1)
   Print k
   k = Parent[k]
```

Record the entry that we used to compute OPT(j)

Exercise:
Longest Increasing Subsequence

Longest Increasing Subsequence

Given a sequence of numbers

Find the longest increasing subsequence in $O(n^2)$ time

41, 22, 9, 15, 23, 39, 21, 56, 24, 34, 59, 23, 60, 39, 87, 23, 90

41, 22, **9**, **15**, **23**, 39, 21, 56, **24**, **34**, **59**, 23, **60**, 39, **87**, 23, **90**

Find the longest increasing subsequence in $O(n^2)$ time.

I can do it in $O(n \log n)$

Total Results: 0

DP for LIS

Let OPT(j) be the longest increasing subsequence ending at j.

Observation: Suppose the OPT(j) is the sequence

$$x_{i_1}, x_{i_2}, \dots, x_{i_k}, x_j$$

Then, $x_{i_1}, x_{i_2}, \dots, x_{i_k}$ is the longest increasing subsequence ending at x_{i_k}, i.e., $OPT(j) = 1 + OPT(i_k)$

How to make it faster?

$$OPT(j) = \begin{cases} 1 & \text{If } x_j < x_i \text{ for all } i < j \\ 1 + \max_{i: x_i < x_j} OPT(i) & \text{o.w.} \end{cases}$$

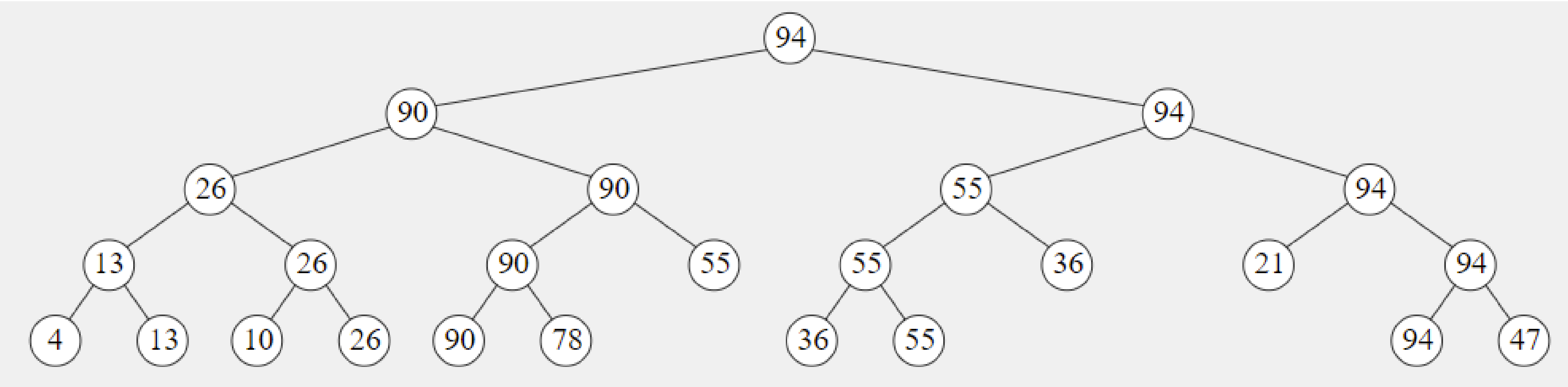
Alternative Soln: This is a special case of Longest path in a DAG: Construct a graph 1,…n where (i, j) is an edge if $i < j$ and $x_i < x_j$.

Data structure for LIS

$$OPT(j) = \begin{cases} 1 & \text{If } x_j < x_i \text{ for all } i < j \\ 1 + \max_{i: x_i < x_j} OPT(i) & \text{o.w.} \end{cases}$$

We need a data structure with following operations:

- Initialize(): Set $x_1, x_2, \cdots x_n$ to 0 in $O(n)$ time.
- Set(j,v): Set x_j to v in $O(\log n)$ time.
- Max(a,b): Output $\max_{a \le j \le b} x_j$ in $O(\log n)$ time.



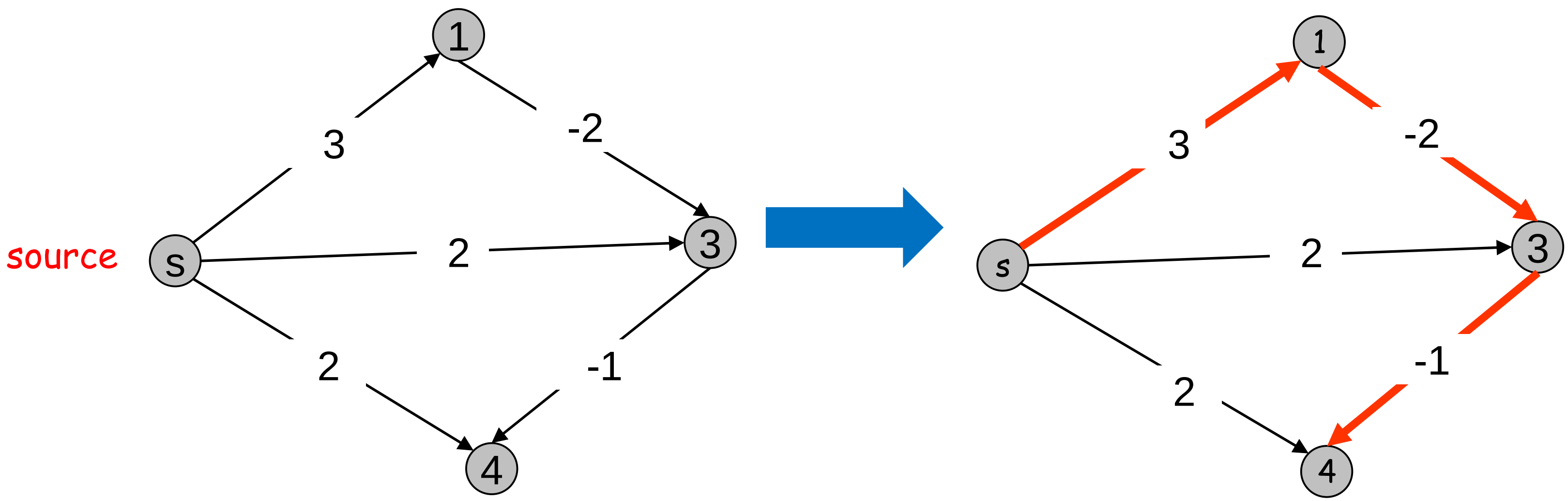
Shortest Paths with Negative Edge Weights

Shortest Paths with Neg Edge Weights

Given a weighted directed graph $G = (V, E)$ and a source vertex s, where the weight of edge (u,v) is $c_{u,v}$ **(that can be negative)**

Goal: Find the shortest path from s to all vertices of G.

Recall that Dikjstra's Algorithm fails when weights are negative



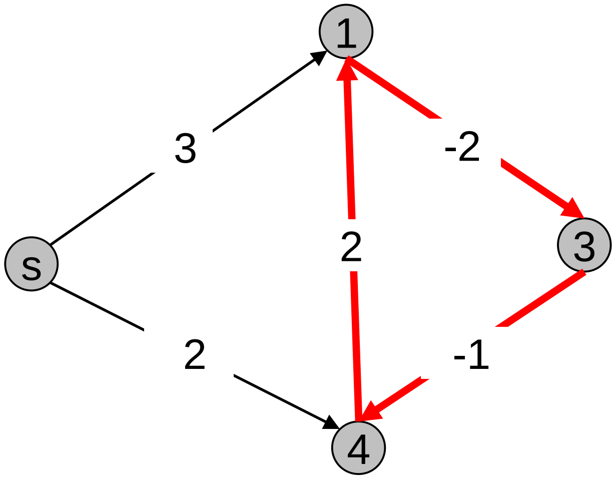
Why distance can be negative?
Think distance as cost instead.

Impossibility on Graphs with Neg Cycles

Condition: No solution exists if G has a negative cycle.

This is because we can minimize the length by going over the cycle again and again.

So, suppose G does not have a negative cycle.



DP for Shortest Path (First Attempt)

Def: Let $OPT(v)$ be the length of the shortest s - v path

$$OPT(v) = \begin{cases} 0 & \text{if } v = s \\ \min_{u:(u,v) \text{ an edge}} OPT(u) + c_{u,v} & \end{cases}$$

The formula is correct. But it is not clear how to compute it.

DP for Shortest Path

Def: Let $OPT(v, i)$ be the length of the shortest s - v path with at most i edges.

Let us characterize $OPT(v, i)$.

Case 1: $OPT(v, i)$ path has less than i edges.

- Then, $OPT(v, i) = OPT(v, i-1)$.

Case 2: $OPT(v, i)$ path has exactly i edges.

- Let $s, v_1, v_2, \ldots, v_{i-1}, v$ be the $OPT(v, i)$ path with i edges.
- Then, $s, v_1, \ldots, v_{i-1}$ must be the shortest s - v_{i-1} path with at most $i - 1$ edges. So,
$$OPT(v, i) = OPT(v_{i-1}, i - 1) + c_{v_{i-1}, v}$$

DP for Shortest Path

Def: Let $OPT(v, i)$ be the length of the shortest s - v path with at most i edges.

$$OPT(v,i) = \begin{cases} 0 & \text{if } v = s \\ \infty & \text{if } v \neq s, i = 0 \\ \min\left(OPT(v, i-1), \min_{u:(u,v) \text{ an edge}} OPT(u, i-1) + c_{u,v}\right) \end{cases}$$

So, for every v, $OPT(v, ?)$ is the shortest path from s to v.

But how long do we have to run?

Since G has no negative cycle, it has at most $n - 1$ edges. So, $OPT(v, n-1)$ is the answer.

Bellman Ford Algorithm

```
for v=1 to n
   if v ≠ s then
      M[v,0]=∞
M[s,0]=0.

for i=1 to n-1
   for v=1 to n
      M[v,i]=M[v,i-1]
      for every edge (u,v)
         M[v,i]=min(M[v,i], M[u,i-1]+c_{u,v})
```

	Complexity	Author
	$O(n^4)$	Shimbel (1955) [30]
	$O(Wn^2m)$	Ford (1956) [14]
*	$O(nm)$	Bellman (1958) [1], Moore (1959) [25]
	$O(n^{\frac{3}{4}}m \log W)$	Gabow (1983) [9]
	$O(\sqrt{n}m \log(nW))$	Gabow and Tarjan (1989) [10]
*	$O(\sqrt{n}m \log(W))$	Goldberg (1993) [12]
*	$\tilde{O}(Wn^{\omega})$	Sankowski (2005) [27] Yuster and Zwick (2005) [35]
*	$\tilde{O}(m^{10/7} \log W)$	Cohen, Madry, Sankowski, Vladu (2016)

Table 1: The complexity results for the SSSP problem with negative weights (* indicates asymptotically the best bound for some range of parameters).

Running Time: $O(nm)$

Can we test if G has negative cycles?

Yes, run for i=1…3n and see if the M[v,n-1] is different from M[v,3n]

Exercise:
Minimum Vertex Cover for Tree

Minimum Vertex Cover for Tree

Given an undirected tree $T = (V, E)$.

We call $S \subset V$ is a vertex cover if every edge touches some vertex in S.

Give a linear time algorithm to find the minimum vertex cover of tree.

Answer:

Let $F(v)$ be the size of minimum vertex cover of the subtree at v.

Then

$$F(v) = \min(\#children(v) + \sum_{g:\, grandchild\ of\ v} F(g)\, , 1 + \sum_{c:\, child\ of\ v} F(c))$$