Dynamic Programming on Trees

Lecture 4

Jan 25, 2018

Most slides are courtesy Prof. Chekuri

What is Dynamic Programming?

Every recursion can be memoized. Automatic memoization does not help us understand whether the resulting algorithm is efficient or not.

Dynamic Programming:

A recursion that when memoized leads to an *efficient* algorithm.

Key Questions:

- Given a recursive algorithm, how do we analyze the complexity when it is memoized?
- How do we recognize whether a problem admits a (recursive) dynamic programming based efficient algorithm?
- How do we further optimize time and space of a dynamic programming based algorithm?

Dynamic Programming Template

1. Come up with a recursive algorithm to solve problem
2. Understand the structure/number of the subproblems generated by recursion
3. Memoize the recursion
   - set up compact notation for subproblems
   - set up a data structure for storing subproblems

Dynamic Programming Template

1. Come up with a recursive algorithm to solve problem
2. Understand the structure/number of the subproblems generated by recursion
3. Memoize the recursion
   - set up compact notation for subproblems
   - set up a data structure for storing subproblems
4. Iterative algorithm
   - Understand dependency graph on subproblems
   - Pick an evaluation order (any topological sort of the dependency DAG)
5. Analyze time and space
6. Optimize

Dynamic Programming on Trees

Fact: Many graph optimization problems are **NP-Hard**

Fact: The same graph optimization problems are in P on trees.

Why?

Dynamic Programming on Trees

Fact: Many graph optimization problems are **NP-Hard**

Fact: The same graph optimization problems are in $\boldsymbol{P}$ on trees.

Why?

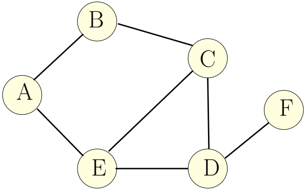
A significant reason: DP algorithm based on *decomposability*

Powerful methodology for graph algorithms via a formal notion of decomposability called **treewidth** (beyond the scope of this class)

Maximum Independent Set in a Graph

Definition

Given undirected graph $G = (V, E)$ a subset of nodes $S \subseteq V$ is an independent set (also called a stable set) if for there are no edges between nodes in S. That is, if $u, v \in S$ then $(u, v) \notin E$.

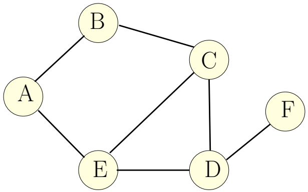


Some independent sets in graph above: $\{D\}, \{A, C\}, \{B, E, F\}$

Maximum Independent Set Problem

Input Graph $G = (V, E)$

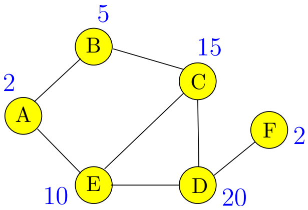
Goal Find maximum sized independent set in G



Maximum Weight Independent Set Problem

Input Graph $G = (V, E)$, weights $w(v) \geq 0$ for $v \in V$

Goal Find maximum weight independent set in G



Maximum Weight Independent Set Problem

1. No one knows an *efficient* (polynomial time) algorithm for this problem
2. Problem is **NP-Hard** and it is *believed* that there is no polynomial time algorithm

Brute-force algorithm:

Maximum Weight Independent Set Problem

1. No one knows an *efficient* (polynomial time) algorithm for this problem
2. Problem is **NP-Hard** and it is *believed* that there is no polynomial time algorithm

Brute-force algorithm:

Try all subsets of vertices.

A Recursive Algorithm

Let $V = \{v_1, v_2, \ldots, v_n\}$.
For a vertex u let $N(u)$ be its neighbors.

A Recursive Algorithm

Let $V = \{v_1, v_2, \ldots, v_n\}$.
For a vertex u let $N(u)$ be its neighbors.

Observation

v_1: *vertex in the graph.*
One of the following two cases is true

Case 1 v_1 *is in* some *maximum independent set.*

Case 2 v_1 *is in* no *maximum independent set.*

We can try both cases *to "reduce" the size of the problem*

A Recursive Algorithm

Let $V = \{v_1, v_2, \ldots, v_n\}$.
For a vertex u let $N(u)$ be its neighbors.

Observation

v_1: *vertex in the graph.*
One of the following two cases is true

Case 1 v_1 *is in* some *maximum independent set.*

Case 2 v_1 *is in* no *maximum independent set.*

We can try both cases *to "reduce" the size of the problem*

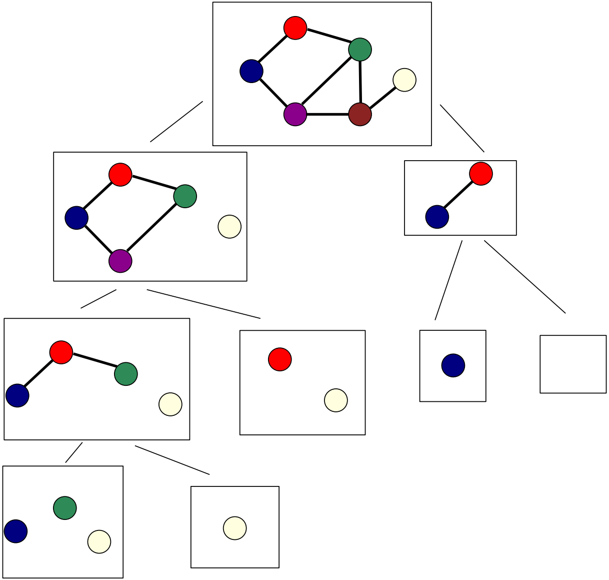
$G_1 = G - v_1$ obtained by removing v_1 and incident edges from G
$G_2 = G - v_1 - N(v_1)$ obtained by removing $N(v_1) \cup v_1$ from G

$$MIS(G) = \max\{MIS(G_1), MIS(G_2) + w(v_1)\}$$

A Recursive Algorithm

```
RecursiveMIS(G):
    if G is empty then Output 0
    v ← a vertex of G
    a = RecursiveMIS(G − v)
    b = w(v) + RecursiveMIS(G − v − N(v))
    Output max(a, b)
```

Example



Running time:

$$T(n) = T(n-1) + T\Big(n - 1 - deg(v)\Big) + O(1 + deg(v))$$

where $deg(v)$ is the degree of v. $T(0) = T(1) = 1$ is base case.

Running time:

$$T(n) = T(n-1) + T\Big(n - 1 - deg(v)\Big) + O(1 + deg(v))$$

where $deg(v)$ is the degree of v. $T(0) = T(1) = 1$ is base case.

Worst case is when $deg(v) = 0$ when the recurrence becomes

$$T(n) = 2T(n-1) + O(1)$$

Solution to this is $T(n) = O(2^n)$.

Memoization

We can memoize the recursive algorithm.

Question: Does it lead to an efficient algorithm?

Memoization

We can memoize the recursive algorithm.

Question: Does it lead to an efficient algorithm?

What are the sub-problems?

Memoization

We can memoize the recursive algorithm.

Question: Does it lead to an efficient algorithm?

What are the sub-problems? Ans.: Subgraphs (subsets of nodes).

How many are they if G has n nodes to start with?

Memoization

We can memoize the recursive algorithm.

Question: Does it lead to an efficient algorithm?

What are the sub-problems? Ans.: Subgraphs (subsets of nodes).

How many are they if G has n nodes to start with? A.: Exponential.

Exercise: Show that even when G is a cycle the number of subproblems is exponential in n.

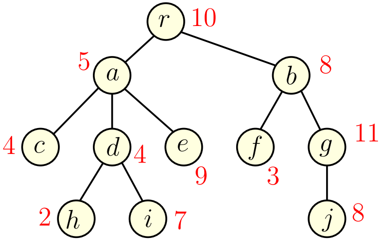
Part I

Maximum Weighted Independent Set in Trees

Maximum Weight Independent Set in a Tree

Input Tree $T = (V, E)$ and weights $w(v) \geq 0$ for each $v \in V$

Goal Find maximum weight independent set in T



Maximum weight independent set in above tree: ??

A Recursive Algorithm

For an arbitrary graph G:

1. Number vertices as $v_1, v_2, \ldots, v_n$
2. Find recursively optimum solutions without v_n (recurse on $G - v_n$) and with v_n (recurse on $G - v_n - N(v_n)$ & include v_n).
3. Saw that if graph G is arbitrary there was no good ordering that resulted in a small number of subproblems.

What about a tree?

A Recursive Algorithm

For an arbitrary graph G:

1. Number vertices as $v_1, v_2, \ldots, v_n$
2. Find recursively optimum solutions without v_n (recurse on $G - v_n$) and with v_n (recurse on $G - v_n - N(v_n)$ & include v_n).
3. Saw that if graph G is arbitrary there was no good ordering that resulted in a small number of subproblems.

What about a tree? Natural candidate for v_n is root r of T?

Towards a Recursive Solution

Natural candidate for v_n is root r of T? Let $\mathcal{O}$ be an optimum solution to the whole problem.

Case $r \notin \mathcal{O}$:

Towards a Recursive Solution

Natural candidate for v_n is root r of T? Let $\mathcal{O}$ be an optimum solution to the whole problem.

Case $r \notin \mathcal{O}$:Then $\mathcal{O}$ contains an optimum solution for each subtree of T hanging at a child of r.

Towards a Recursive Solution

Natural candidate for v_n is root r of T? Let $\mathcal{O}$ be an optimum solution to the whole problem.

Case $r \notin \mathcal{O}$:Then $\mathcal{O}$ contains an optimum solution for each subtree of T hanging at a child of r.

Case $r \in \mathcal{O}$:

Towards a Recursive Solution

Natural candidate for v_n is root r of T? Let $\mathcal{O}$ be an optimum solution to the whole problem.

Case $r \not\in \mathcal{O}$:Then $\mathcal{O}$ contains an optimum solution for each subtree of T hanging at a child of r.

Case $r \in \mathcal{O}$:None of the children of r can be in $\mathcal{O}$. $\mathcal{O} - \{r\}$ contains an optimum solution for each subtree of T hanging at a grandchild of r.

Towards a Recursive Solution

Natural candidate for v_n is root r of T? Let $\mathcal{O}$ be an optimum solution to the whole problem.

Case $r \notin \mathcal{O}$:Then $\mathcal{O}$ contains an optimum solution for each subtree of T hanging at a child of r.

Case $r \in \mathcal{O}$:None of the children of r can be in $\mathcal{O}$. $\mathcal{O} - \{r\}$ contains an optimum solution for each subtree of T hanging at a grandchild of r.

Subproblems? Subtrees of T rooted at nodes in T.

Towards a Recursive Solution

Natural candidate for v_n is root r of T? Let $\mathcal{O}$ be an optimum solution to the whole problem.

Case $r \notin \mathcal{O}$:Then $\mathcal{O}$ contains an optimum solution for each subtree of T hanging at a child of r.

Case $r \in \mathcal{O}$:None of the children of r can be in $\mathcal{O}$. $\mathcal{O} - \{r\}$ contains an optimum solution for each subtree of T hanging at a grandchild of r.

Subproblems? Subtrees of T rooted at nodes in T.

How many of them?

Towards a Recursive Solution

Natural candidate for v_n is root r of T? Let $\mathcal{O}$ be an optimum solution to the whole problem.

Case $r \not\in \mathcal{O}$:Then $\mathcal{O}$ contains an optimum solution for each subtree of T hanging at a child of r.

Case $r \in \mathcal{O}$:None of the children of r can be in $\mathcal{O}$. $\mathcal{O} - \{r\}$ contains an optimum solution for each subtree of T hanging at a grandchild of r.

Subproblems? Subtrees of T rooted at nodes in T.

How many of them?

Towards a Recursive Solution

Natural candidate for v_n is root r of T? Let $\mathcal{O}$ be an optimum solution to the whole problem.

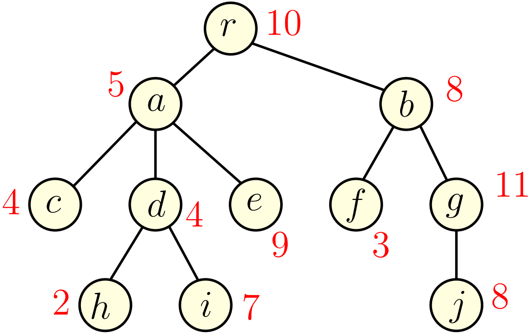
Case $r \notin \mathcal{O}$:Then $\mathcal{O}$ contains an optimum solution for each subtree of T hanging at a child of r.

Case $r \in \mathcal{O}$:None of the children of r can be in $\mathcal{O}$. $\mathcal{O} - \{r\}$ contains an optimum solution for each subtree of T hanging at a grandchild of r.

Subproblems? Subtrees of T rooted at nodes in T.

How many of them? $O(n)$

Example



A Recursive Solution

$T(u)$: subtree of T hanging at node u

$OPT(u)$: max weighted independent set value in $T(u)$

$$OPT(u) =$$

A Recursive Solution

$T(u)$: subtree of T hanging at node u

$OPT(u)$: max weighted independent set value in $T(u)$

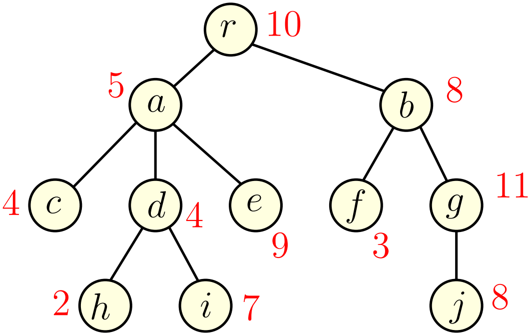
$$OPT(u) = \max \begin{cases} \sum_{v \text{ child of } u} OPT(v), \\ w(u) + \sum_{v \text{ grandchild of } u} OPT(v) \end{cases}$$

Iterative Algorithm

1. To evaluate $OPT(u)$ need to have computed values of all children and grandchildren of u. Compute $OPT(u)$ bottom up.

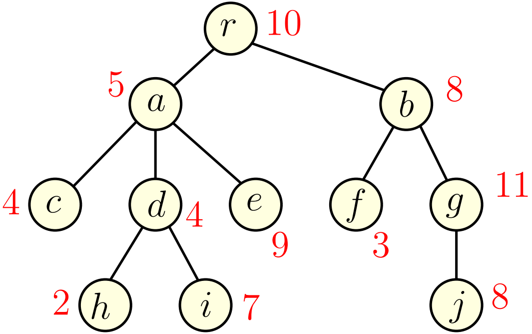
Iterative Algorithm

1. To evaluate $OPT(u)$ need to have computed values of all children and grandchildren of u. Compute $OPT(u)$ bottom up.
2. What is an ordering of nodes of a tree T to achieve above?



Iterative Algorithm

1. To evaluate $OPT(u)$ need to have computed values of all children and grandchildren of u. Compute $OPT(u)$ bottom up.
2. What is an ordering of nodes of a tree T to achieve above?



Ans.: Post-order traversal of a tree.

Iterative Algorithm

MIS-Tree(T):

    Let $v_1, v_2, \ldots, v_n$ be a post-order traversal of nodes of T
    for $i = 1$ to n **do**
$$M[v_i] = \max\left(\begin{array}{c}\sum_{v_j \text{ child of } v_i} M[v_j], \\ w(v_i) + \sum_{v_j \text{ grandchild of } v_i} M[v_j]\end{array}\right)$$
    return $M[v_n]$ (* Note: v_n is the root of T *)

Iterative Algorithm

MIS-Tree(T):
    Let $v_1, v_2, \ldots, v_n$ be a post-order traversal of nodes of T
    for $i = 1$ to n **do**

$$M[v_i] = \max\begin{pmatrix} \sum_{v_j \text{ child of } v_i} M[v_j], \\ w(v_i) + \sum_{v_j \text{ grandchild of } v_i} M[v_j] \end{pmatrix}$$

    return $M[v_n]$ (* Note: v_n is the root of T *)

Space:

Iterative Algorithm

MIS-Tree(T):
    Let $v_1, v_2, \ldots, v_n$ be a post-order traversal of nodes of T
    for $i = 1$ to n **do**

$$M[v_i] = \max\begin{pmatrix} \sum_{v_j \text{ child of } v_i} M[v_j], \\ w(v_i) + \sum_{v_j \text{ grandchild of } v_i} M[v_j] \end{pmatrix}$$

    return $M[v_n]$ (* Note: v_n is the root of T *)

Space: $O(n)$ to store the value at each node of T
Running time:

Iterative Algorithm

MIS-Tree(T):
    Let $v_1, v_2, \ldots, v_n$ be a post-order traversal of nodes of T
    for $i = 1$ to n **do**

$$M[v_i] = \max\left(\begin{array}{c} \sum_{v_j \text{ child of } v_i} M[v_j], \\ w(v_i) + \sum_{v_j \text{ grandchild of } v_i} M[v_j] \end{array}\right)$$

    return $M[v_n]$ (* Note: v_n is the root of T *)

Space: $O(n)$ to store the value at each node of T

Running time:

1. Naive bound: Each $M[V_i]$ evaluation may take

Iterative Algorithm

MIS-Tree(T):

    Let $v_1, v_2, \ldots, v_n$ be a post-order traversal of nodes of T
    for $i = 1$ to n **do**

$$M[v_i] = \max\left(\begin{array}{c} \sum_{v_j \text{ child of } v_i} M[v_j], \\ w(v_i) + \sum_{v_j \text{ grandchild of } v_i} M[v_j] \end{array}\right)$$

    return $M[v_n]$ (* Note: v_n is the root of T *)

Space: $O(n)$ to store the value at each node of T

Running time:

1. Naive bound: Each $M[V_i]$ evaluation may take $O(n)$.

Iterative Algorithm

```
MIS-Tree(T):
    Let v_1, v_2, ..., v_n be a post-order traversal of nodes of T
    for i = 1 to n do
```

$$M[v_i] = \max\left(\begin{array}{c} \sum_{v_j \text{ child of } v_i} M[v_j], \\ w(v_i) + \sum_{v_j \text{ grandchild of } v_i} M[v_j] \end{array}\right)$$

```
    return M[v_n] (* Note: v_n is the root of T *)
```

Space: $O(n)$ to store the value at each node of T

Running time:

1. Naive bound: Each $M[V_i]$ evaluation may take $O(n)$.There are n such evaluations – $O(n^2)$.

Iterative Algorithm

MIS-Tree(T):

Let $v_1, v_2, \ldots, v_n$ be a post-order traversal of nodes of T

for $i = 1$ to n **do**

$$M[v_i] = \max\left(\begin{array}{c} \sum_{v_j \text{ child of } v_i} M[v_j], \\ w(v_i) + \sum_{v_j \text{ grandchild of } v_i} M[v_j] \end{array}\right)$$

return $M[v_n]$ (* Note: v_n is the root of T *)

Space: $O(n)$ to store the value at each node of T

Running time:

1. Naive bound: Each $M[V_i]$ evaluation may take $O(n)$.There are n such evaluations – $O(n^2)$.
2. Better bound: Value $M[v_j]$ is accessed by who all?

Iterative Algorithm

MIS-Tree(T):

Let $v_1, v_2, \ldots, v_n$ be a post-order traversal of nodes of T

for $i = 1$ to n **do**

$$M[v_i] = \max\left(\begin{array}{c} \sum_{v_j \text{ child of } v_i} M[v_j], \\ w(v_i) + \sum_{v_j \text{ grandchild of } v_i} M[v_j] \end{array}\right)$$

return $M[v_n]$ (* Note: v_n is the root of T *)

Space: $O(n)$ to store the value at each node of T

Running time:

1. Naive bound: Each $M[V_i]$ evaluation may take $O(n)$.There are n such evaluations – $O(n^2)$.
2. Better bound: Value $M[v_j]$ is accessed by who all? Parent and grand-parent. So in total

Iterative Algorithm

MIS-Tree(T):

```
    Let v_1, v_2, ..., v_n be a post-order traversal of nodes of T
    for i = 1 to n do
```

$$M[v_i] = \max \left(\begin{array}{c} \sum_{v_j \text{ child of } v_i} M[v_j], \\ w(v_i) + \sum_{v_j \text{ grandchild of } v_i} M[v_j] \end{array} \right)$$

```
    return M[v_n] (* Note: v_n is the root of T *)
```

Space: $O(n)$ to store the value at each node of T

Running time:

1. Naive bound: Each $M[V_i]$ evaluation may take $O(n)$.There are n such evaluations – $O(n^2)$.
2. Better bound: Value $M[v_j]$ is accessed by who all? Parent and grand-parent. So in total $O(n)$.

Why did DP work on trees?

Each node (including the root) is a *separator*!

Definition

Given a graph $G = (V, E)$ a set of nodes $S \subset V$ is a *separator* for G if $G - S$ has at least two connected components.

Why did DP work on trees?

Each node (including the root) is a *separator*!

Definition

Given a graph $G = (V, E)$ a set of nodes $S \subset V$ is a *separator* for G if $G - S$ has at least two connected components.

Definition

S is a *balanced* separator if each connected component of $G - S$ has at most $2|V(G)|/3$ nodes.

Why did DP work on trees?

Each node (including the root) is a *separator*!

Definition

Given a graph $G = (V, E)$ a set of nodes $S \subset V$ is a *separator* for G if $G - S$ has at least two connected components.

Definition

S is a *balanced* separator if each connected component of $G - S$ has at most $2|V(G)|/3$ nodes.

Exercise: Prove that every tree T has a balanced separator consisting of a single node.

Why did DP work on trees?

Each node (including the root) is a *separator*!

Definition

Given a graph $G = (V, E)$ a set of nodes $S \subset V$ is a *separator* for G if $G - S$ has at least two connected components.

Definition

S is a *balanced* separator if each connected component of $G - S$ has at most $2|V(G)|/3$ nodes.

Exercise: Prove that every tree T has a balanced separator consisting of a single node.
Aside: $O(2^{\sqrt{n}})$ algorithm to find MIS in planar graphs using, (i) balanced-separators, (ii) DP algorithm on trees.

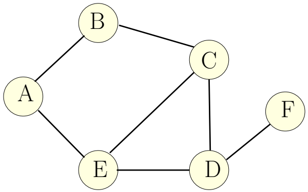
Part II

Minimum Dominating Set in Trees

Minimum Dominating Set in a Graph

Definition

Given undirected graph $G = (V, E)$ a subset of nodes $S \subseteq V$ is a dominating set if for all $v \in V$, either $v \in S$ or a neighbor of v is in S.

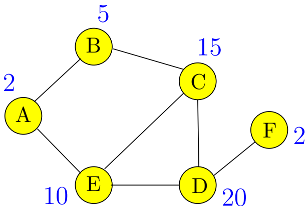


Some dominating sets in graph above: $\{A, B, C, D, E, F\}$,

Minimum Weight Dominating Set Problem

Input Graph $G = (V, E)$, weights $w(v) \geq 0$ for $v \in V$

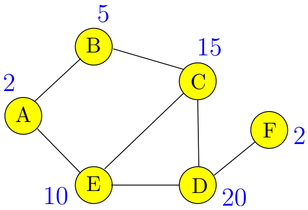
Goal Find minimum weight dominating set in G



Minimum Weight Dominating Set Problem

Input Graph $G = (V, E)$, weights $w(v) \geq 0$ for $v \in V$

Goal Find minimum weight dominating set in G

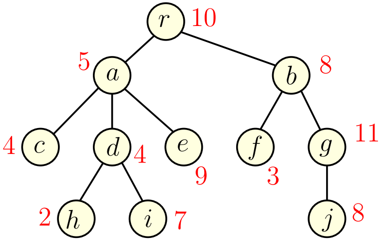


NP-Hard problem

Minimum Weight Dominating Set in a Tree

Input Tree $T = (V, E)$ and weights $w(v) \geq 0$ for each $v \in V$

Goal Find minimum weight dominating set in T



Minimum weight dominating set in above tree: ??

Recursive Algorithm

r is root of T. Let $\mathcal{O}$ be an optimum solution for T.

Case $r \notin \mathcal{O}$: Then $\mathcal{O}$ must contain some child of r. Which one?

Recursive Algorithm

r is root of T. Let $\mathcal{O}$ be an optimum solution for T.

Case $r \notin \mathcal{O}$: Then $\mathcal{O}$ must contain some child of r. Which one?

Case $r \in \mathcal{O}$: None of the children of r *need* to be in $\mathcal{O}$ because r can dominate them. However, they may have to be.

Recursive Algorithm

r is root of T. Let $\mathcal{O}$ be an optimum solution for T.

Case $r \notin \mathcal{O}$: Then $\mathcal{O}$ must contain some child of r. Which one?

Case $r \in \mathcal{O}$: None of the children of r *need* to be in $\mathcal{O}$ because r can dominate them. However, they may have to be.

Issue 1: In both cases it is not feasible to express $|\mathcal{O}|$ easily as optimum solution values of children or descendants of r.

Recursive Algorithm

r is root of T. Let $\mathcal{O}$ be an optimum solution for T.

Case $r \notin \mathcal{O}$: Then $\mathcal{O}$ must contain some child of r. Which one?

Case $r \in \mathcal{O}$: None of the children of r *need* to be in $\mathcal{O}$ because r can dominate them. However, they may have to be.

Issue 1: In both cases it is not feasible to express $|\mathcal{O}|$ easily as optimum solution values of children or descendants of r.

Issue 2: Removing r decomposes T into subtrees rooted at children of r. However, not easy to decompose problem structure recursively. Problems at children of r are *dependent*.
Need to introduce additional variable(s).

Recursive Algorithm: Understanding Dependence

Let $u_1, u_2, \ldots, u_k$ be children of root r of T

What "information" do $T_{u_1}, \ldots, T_{u_k}$ need to know about r's status in an optimum solution in order to become "independent"

Recursive Algorithm: Understanding Dependence

Let $u_1, u_2, \ldots, u_k$ be children of root r of T

What "information" do $T_{u_1}, \ldots, T_{u_k}$ need to know about r's status in an optimum solution in order to become "independent"

- Whether r is included in the solution
- If r is not included then which of the children is going to cover it. Equivalently, T_{u_i} needs to know whether it should cover r or some other child will.

Recursive Algorithm: Introducing Variables

- u: node in tree
- pi: boolean variable to indicate whether parent is in solution. $pi = 0$ means parent is not included. $pi = 1$ means it is included.
- cp: boolean variable to indicate whether node needed to cover parent. $cp = 1$ means parent needs to be covered. $cp = 0$ means not needed.

Recursive Algorithm: Introducing Variables

- u: node in tree
- pi: boolean variable to indicate whether parent is in solution. $pi = 0$ means parent is not included. $pi = 1$ means it is included.
- cp: boolean variable to indicate whether node needed to cover parent. $cp = 1$ means parent needs to be covered. $cp = 0$ means not needed.

$OPT(u, pi, cp)$: value of minimum dominating set in T_u with booleans pi and cp with meaning above.

Recursive Algorithm: Sub-problem

- u: node in tree
- pi indicates if the parent is included or not.
- cp indicates if the parent needs to be covered or not.

$OPT(u, pi, cp)$: opt value in T_u with pi and cp as above.

$OPT(u, 0, 0)$: opt value in T_u when parent of u is not included and u need not cover it.

Recursive Algorithm: Sub-problem

- u: node in tree
- pi indicates if the parent is included or not.
- cp indicates if the parent needs to be covered or not.

$OPT(u, pi, cp)$: opt value in T_u with pi and cp as above.

$OPT(u, 0, 0)$: opt value in T_u when parent of u is not included and u need not cover it.
$OPT(u, 0, 1)$: opt value in T_u when parent of u is not included and u need to cover it.

Recursive Algorithm: Sub-problem

- u: node in tree
- pi indicates if the parent is included or not.
- cp indicates if the parent needs to be covered or not.

$OPT(u, pi, cp)$: opt value in T_u with pi and cp as above.

$OPT(u, 0, 0)$: opt value in T_u when parent of u is not included and u need not cover it.
$OPT(u, 0, 1)$: opt value in T_u when parent of u is not included and u need to cover it.
$OPT(u, 1, 0)$: opt value in T_u when parent of u is included and u need not cover it.

Recursive Algorithm: Sub-problem

- u: node in tree
- pi indicates if the parent is included or not.
- cp indicates if the parent needs to be covered or not.

$OPT(u, pi, cp)$: opt value in T_u with pi and cp as above.

$OPT(u, 0, 0)$: opt value in T_u when parent of u is not included and u need not cover it.
$OPT(u, 0, 1)$: opt value in T_u when parent of u is not included and u need to cover it.
$OPT(u, 1, 0)$: opt value in T_u when parent of u is included and u need not cover it.
$OPT(u, 1, 1)$: NOT NEEDED!

Recursive Algorithm: Sub-problem

- u: node in tree
- pi indicates if the parent is included or not.
- cp indicates if the parent needs to be covered or not.

$OPT(u, pi, cp)$: opt value in T_u with pi and cp as above.

$OPT(u, 0, 0)$: opt value in T_u when parent of u is not included and u need not cover it.
$OPT(u, 0, 1)$: opt value in T_u when parent of u is not included and u need to cover it.
$OPT(u, 1, 0)$: opt value in T_u when parent of u is included and u need not cover it.
$OPT(u, 1, 1)$: NOT NEEDED!

$OPT(r, 0, 0)$: value of minimum dominating set in T.

Recursive Solution

Can we express $OPT(u, pi, cp)$ recursively via children of u?

Recursive Solution

Can we express $OPT(u, pi, cp)$ recursively via children of u?
$OPT(u, 0, 0)$: Value of a minimum dominating set in T_u where we assume that u's parent is not included and u does not need to cover its parent.

Recursive Solution

Can we express $OPT(u, pi, cp)$ recursively via children of u?
$OPT(u, 0, 0)$: Value of a minimum dominating set in T_u where we assume that u's parent is not included and u does not need to cover its parent.Let C_u be children of u.

Case u is included: Then u does not need to be covered by any child.

Recursive Solution

Can we express $OPT(u, pi, cp)$ recursively via children of u?
$OPT(u, 0, 0)$: Value of a minimum dominating set in T_u where we assume that u's parent is not included and u does not need to cover its parent.Let C_u be children of u.

Case u is included: Then u does not need to be covered by any child. Include u and recurse.
$OPT(u, 0, 0) = w(u) + \sum_{v \in C_u}$

Recursive Solution

Can we express $OPT(u, pi, cp)$ recursively via children of u?
$OPT(u, 0, 0)$: Value of a minimum dominating set in T_u where we assume that u's parent is not included and u does not need to cover its parent.Let C_u be children of u.

Case u is included: Then u does not need to be covered by any child. Include u and recurse.

$$OPT(u, 0, 0) = w(u) + \sum_{v \in C_u} OPT(v, 1, 0)$$

Case u is not included: Then u needs to be covered by some child. We do a min over all children.

$$OPT(u, 0, 0) =$$
$$\min_{v \in C_u}$$

Recursive Solution

Can we express $OPT(u, pi, cp)$ recursively via children of u?
$OPT(u, 0, 0)$: Value of a minimum dominating set in T_u where we assume that u's parent is not included and u does not need to cover its parent.Let C_u be children of u.

Case u is included: Then u does not need to be covered by any child. Include u and recurse.
$OPT(u, 0, 0) = w(u) + \sum_{v \in C_u} OPT(v, 1, 0)$

Case u is not included: Then u needs to be covered by some child. We do a min over all children.
$OPT(u, 0, 0) =$
$\min_{v \in C_u} (OPT(v, 0, 1) + \sum_{v' \in C_u - v} OPT(v', 0, 0))$

Recursive Solution

Can we express $OPT(u, pi, cp)$ recursively via children of u?
$OPT(u, 0, 0)$: Value of a minimum dominating set in T_u where we assume that u's parent is not included and u does not need to cover its parent.Let C_u be children of u.

Case u is included: Then u does not need to be covered by any child. Include u and recurse.
$OPT(u, 0, 0) = w(u) + \sum_{v \in C_u} OPT(v, 1, 0)$

Case u is not included: Then u needs to be covered by some child. We do a min over all children.
$OPT(u, 0, 0) =$
$\min_{v \in C_u} \left(OPT(v, 0, 1) + \sum_{v' \in C_u - v} OPT(v', 0, 0)\right)$

Since one of these cases has to be true, we take the min of the values in the above two cases to compute $OPT(u, 0, 0)$.

Recursive Solution

$OPT(u, 0, 1)$ **:** Value of a minimum dominating set in T_u where we assume that u's parent is not included and u needs to cover its parent. Let C_u be children of u.

Recursive Solution

$OPT(u, 0, 1)$ **:** Value of a minimum dominating set in T_u where we assume that u's parent is not included and u needs to cover its parent. Let C_u be children of u.

Case u is included: Then u does not need to covered by any child. Include u and recurse.

$$OPT(u, 0, 1) = w(u) + \sum_{v \in C_u} OPT(v, 1, 0)$$

Recursive Solution

$OPT(u, 0, 1)$ **:** Value of a minimum dominating set in T_u where we assume that u's parent is not included and u needs to cover its parent. Let C_u be children of u.

Case u is included: Then u does not need to covered by any child.
Include u and recurse.
$OPT(u, 0, 1) = w(u) + \sum_{v \in C_u} OPT(v, 1, 0)$

Case u is not included:

Recursive Solution

$OPT(u, 0, 1)$ **:** Value of a minimum dominating set in T_u where we assume that u's parent is not included and u needs to cover its parent. Let C_u be children of u.

Case u is included: Then u does not need to covered by any child. Include u and recurse.
$OPT(u, 0, 1) = w(u) + \sum_{v \in C_u} OPT(v, 1, 0)$

Case u is not included: This does not arise because u has to cover its parent.

Recursive Solution

$OPT(u, 1, 0)$: Value of a minimum dominating set in T_u where we assume that u's parent is included and u does not need to cover its parent. Let C_u be children of u.

Recursive Solution

$OPT(u, 1, 0)$ **:** Value of a minimum dominating set in T_u where we assume that u's parent is included and u does not need to cover its parent. Let C_u be children of u.

Case u is included: Then u does not need to covered by any child.
Include u and recurse.
$OPT(u, 1, 0) = w(u) + \sum_{v \in C_u} OPT(v, 1, 0)$

Recursive Solution

$OPT(u, 1, 0)$ **:** Value of a minimum dominating set in T_u where we assume that u's parent is included and u does not need to cover its parent. Let C_u be children of u.

Case u is included: Then u does not need to covered by any child. Include u and recurse.

$$OPT(u, 1, 0) = w(u) + \sum_{v \in C_u} OPT(v, 1, 0)$$

Case u is not included: u's parent is included. Now, does u need to be covered by its children?

Recursive Solution

$OPT(u, 1, 0)$ **:** Value of a minimum dominating set in T_u where we assume that u's parent is included and u does not need to cover its parent. Let C_u be children of u.

Case u is included: Then u does not need to covered by any child. Include u and recurse.

$$OPT(u, 1, 0) = w(u) + \sum_{v \in C_u} OPT(v, 1, 0)$$

Case u is not included: u's parent is included. Now, does u need to be covered by its children? No. Thus we have,

$$OPT(u, 1, 0) = \sum_{v \in C_u} OPT(v, 0, 0)$$

Take the min of the values in the above two cases to compute $OPT(u, 1, 0)$.

Recursive Solution

$OPT(u, 1, 0)$ **:** Value of a minimum dominating set in T_u where we assume that u's parent is included and u does not need to cover its parent. Let C_u be children of u.

Case u is included: Then u does not need to covered by any child. Include u and recurse.

$$OPT(u, 1, 0) = w(u) + \sum_{v \in C_u} OPT(v, 1, 0)$$

Case u is not included: u's parent is included. Now, does u need to be covered by its children? No. Thus we have,

$$OPT(u, 1, 0) = \sum_{v \in C_u} OPT(v, 0, 0)$$

Take the min of the values in the above two cases to compute $OPT(u, 1, 0)$.

Caution: Not including u may appear to be always advantageous but it is not true.

Recursive Solution

$OPT(u, 1, 1)$ **:** Value of a minimum dominating set in T_u where we assume that u's parent is included and u needs to cover its parent.

This subproblem does not make sense since if u's parent is included then u does not need to cover it.

Base Cases

Leaves are base cases. If u is a leaf.

- $OPT(u, 0, 0) =$

Base Cases

Leaves are base cases. If u is a leaf.

- $OPT(u, 0, 0) = w(u)$
- $OPT(u, 0, 1) =$

Base Cases

Leaves are base cases. If u is a leaf.

- $OPT(u, 0, 0) = w(u)$
- $OPT(u, 0, 1) = w(u)$
- $OPT(u, 1, 0) =$

Base Cases

Leaves are base cases. If u is a leaf.

- $OPT(u, 0, 0) = w(u)$
- $OPT(u, 0, 1) = w(u)$
- $OPT(u, 1, 0) = 0$

DP Algorithm

- Minimum weight dominating set value in T is $OPT(r, \mathbf{0}, \mathbf{0})$

DP Algorithm

- Minimum weight dominating set value in T is $OPT(r, 0, 0)$
- To compute $OPT(r, 0, 0)$ we need to compute recursively $OPT(u, 0, 0), OPT(u, 0, 1), OPT(u, 1, 0)$ for all $u \in T$. Thus number of subproblems is $O(n)$.

DP Algorithm

- Minimum weight dominating set value in T is $OPT(r, 0, 0)$
- To compute $OPT(r, 0, 0)$ we need to compute recursively $OPT(u, 0, 0), OPT(u, 0, 1), OPT(u, 1, 0)$ for all $u \in T$. Thus number of subproblems is $O(n)$.
- Nodes should be traveresed in what order?

DP Algorithm

- Minimum weight dominating set value in T is $OPT(r, 0, 0)$
- To compute $OPT(r, 0, 0)$ we need to compute recursively $OPT(u, 0, 0), OPT(u, 0, 1), OPT(u, 1, 0)$ for all $u \in T$. Thus number of subproblems is $O(n)$.
- Nodes should be traveresed in what order? Ans.: bottom up from leaves to root.

DP Algorithm

- Minimum weight dominating set value in T is $OPT(r, 0, 0)$
- To compute $OPT(r, 0, 0)$ we need to compute recursively $OPT(u, 0, 0), OPT(u, 0, 1), OPT(u, 1, 0)$ for all $u \in T$. Thus number of subproblems is $O(n)$.
- Nodes should be traveresed in what order? Ans.: bottom up from leaves to root.
- In particular?

DP Algorithm

- Minimum weight dominating set value in T is $OPT(r, 0, 0)$
- To compute $OPT(r, 0, 0)$ we need to compute recursively $OPT(u, 0, 0), OPT(u, 0, 1), OPT(u, 1, 0)$ for all $u \in T$. Thus number of subproblems is $O(n)$.
- Nodes should be traveresed in what order? Ans.: bottom up from leaves to root.
- In particular? Ans.: post-order traversal.

Iterative Algorithm

```
DominatingSet-Tree(T):
    Let v_1, v_2, ..., v_n be a post-order traversal of nodes of T
    Allocate array M[1..n, 0..1, 0..1] to store OPT(v_i, pi, cp) values
    for i = 1 to n do
        Compute OPT(v_i, 0, 0), OPT(v_i, 1, 0) and OPT(v_i, 0, 1) using
            values of children of v_i stored in M,
            or via base cases if v_i is leaf

            Store computed values in M for use by parent of v_i.
    return OPT(v_n, 0, 0) (* Note: v_n is the root of T *)
```

Exercise: Work out details and prove an $O(n)$ time implementation.

Recap

- To obtain recursive solution we introduced additional variables based on "information" needed to decompose
- Decomposition depends both on structure (trees decompose via separators) and objective function
- Subproblems and recursion are almost defined hand in hand