More DP: LCS and MIS in Trees

Lecture 15

March 18, 2021

Recipe for Dynamic Programming

1. Develop a recursive backtracking style algorithm $\mathcal{A}$ for given problem.
2. Identify *structure* of subproblems generated by $\mathcal{A}$ on an instance I of size n
    1. Estimate number of different subproblems generated as a function of n. Is it polynomial or exponential in n?
    2. If the number of problems is "small" (polynomial) then they typically have some "clean" structure.
3. Rewrite subproblems in a compact fashion.
4. Rewrite recursive algorithm in terms of notation for subproblems.
5. Convert to iterative algorithm by bottom up evaluation in an appropriate order.
6. Optimize further with data structures and/or additional ideas.

Part I

Longest Common Subsequence Problem

LCS Problem

Definition

LCS between two sequences X and Y is the length of longest common *subsequence* of X and Y.

Example

LCS between A, B, A, Z, D, C and B, A, C, B, A, D is

LCS Problem

Definition

LCS between two sequences X and Y is the length of longest common *subsequence* of X and Y.

Example

LCS between A, B, A, Z, D, C and B, A, C, B, A, D is 4 via A, B, A, D

LCS Problem

Definition

LCS between two sequences X and Y is the length of longest common *subsequence* of X and Y.

Example

LCS between A, B, A, Z, D, C and B, A, C, B, A, D is 4 via A, B, A, D

Question: Derive an efficient polynomial time algorithm to compute LCS of two given sequences $X[1..m]$ and $Y[1..n]$

Recursive Solution/Algorithm

Express **LCS**($X[1..m]$, $Y[1..n]$) in terms of smaller instances. How do we decompose? Case analysis.

Any common subsequence of X, Y is one of the following types

- Case 0: empty if X or Y is empty sequence
- Case 1: does not include $X[1]$ the first character of X
- Case 2: does not include $Y[1]$ the first character of Y
- Case 3: $X[1] = Y[1]$ and includes $X[1]$ as first in seq.

Recursive Solution/Algorithm

Express $\mathsf{LCS}(X[1..m], Y[1..n])$ in terms of smaller instances. How do we decompose? Case analysis.

Any common subsequence of X, Y is one of the following types

- Case 0: empty if X or Y is empty sequence
- Case 1: does not include $X[1]$ the first character of X
- Case 2: does not include $Y[1]$ the first character of Y
- Case 3: $X[1] = Y[1]$ and includes $X[1]$

Find longest common subsequence of each type recursively and take the max.

Recursive Solution/Algorithm

Express **LCS**($X[1..m]$, $Y[1..n]$) in terms of smaller instances. How do we decompose? Case analysis.

Any common subsequence of X, Y is one of the following types

- Case 0: empty if X or Y is empty sequence
- Case 1: does not include $X[1]$ the first character of X
- Case 2: does not include $Y[1]$ the first character of Y
- Case 3: $X[1] = Y[1]$ and includes $X[1]$

Recursive Solution/Algorithm

Express **LCS**$(X[1..m], Y[1..n])$ in terms of smaller instances. How do we decompose? Case analysis.

Any common subsequence of X, Y is one of the following types

- Case 0: empty if X or Y is empty sequence
- Case 1: does not include $X[1]$ the first character of X
- Case 2: does not include $Y[1]$ the first character of Y
- Case 3: $X[1] = Y[1]$ and includes $X[1]$

Find longest common subsequence of each type recursively and take the max.

Recursive Algorithm

LCS($X[1..m]$, $Y[1..n]$)
  If $(m = 0$ or $n = 0)$ return 0
  $m_1 =$ **LCS**$(X[2..m], Y[1..n])$
  $m_2 =$ **LCS**$(X[1..m], Y[2..n)]))$
  $m_3 = 0$
  If $(X[1] = Y[1])$ $m_3 = 1 +$ **LCS**$(X[2..m], Y[2..n])$
  return $\max(m_1, m_2, m_3)$

$\boxed{\begin{matrix} A \\ A \end{matrix}}$ $\begin{matrix} B\,\underbrace{C\,D} \\ Z\,\underbrace{C\,D} \end{matrix}$

A

Recursive Algorithm

```
LCS(X[1..m], Y[1..n])
    If (m = 0 or n = 0) return 0
    m_1 = LCS(X[2..m], Y[1..n])
    m_2 = LCS(X[1..m], Y[2..n)]))
    m_3 = 0
    If (X[1] = Y[1]) m_3 = 1 + LCS(X[2..m], Y[2..n])
    return max(m_1, m_2, m_3)
```

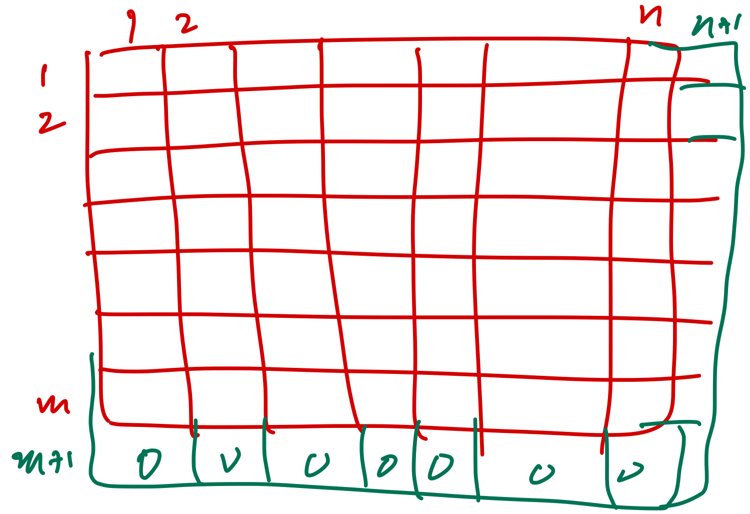
Observation: Each subproblem is of the form $\mathrm{LCS}(X[i..m], Y[j..n])$ for some $1 \leq i \leq m, 1 \leq j \leq n$ and hence only $O(nm)$ of them.

Memoizing the Recursive Algorithm

```
int M[1..m+1][1..n+1]
Initialize all entries of M[i][j] to −1
return LCS(X[1..m], Y[1..n])
```

```
LCS(X[i..m], Y[j..n])
    If (M[i][j] ≥ 0) return M[i][j]    (* return stored value *)

    If (i > m) M[i][j] = 0
    ElseIf (j > n) M[i][j] = 0
    Else
        m_1 = LCS(X[i+1..m], Y[j..n])
        m_2 = LCS(X[i..m], Y[j+1..n]))
        m_3 = 0
        If (X[i] = Y[j]) m_3 = 1 + LCS(X[i+1..m], Y[j+1..n])
        M[i,j] = max(m_1, m_2, m_3)
    return M[i][j]
```



$LCS(i,j)$ = longest common subseq of $x_i, \ldots, x_m$, and $y_j, \ldots, y_n$.

Subproblems and Recurrence

Optimal LCS

Let $\mathbf{LCS}(i, j)$ be length of longest common subsequence of $x_i, \ldots, x_m$ and $y_j, \ldots, y_n$. Then

$$\mathbf{LCS}(i, j) = \max \begin{cases} \mathbf{LCS}(i+1, n) \\ \mathbf{LCS}(i, j+1), \\ (1 + \mathbf{LCS}(i+1, j+1))[x_i = y_j] \end{cases}$$

Subproblems and Recurrence

Optimal LCS

Let $\mathbf{LCS}(i, j)$ be length of longest common subsequence of $x_i, \ldots, x_m$ and $y_j, \ldots, y_n$. Then

$$\mathbf{LCS}(i, j) = \max \begin{cases} \mathbf{LCS}(i+1, n) \\ \mathbf{LCS}(i, j+1), \\ (1 + \mathbf{LCS}(i+1, j+1))[x_i = y_j] \end{cases}$$

Base Cases: $\mathbf{LCS}(i, n+1) = 0$ for $i \geq 1$ and $\mathbf{LCS}(m+1, j) = 0$ for $j \geq 1$.

Return $LCS(1,1)$.

Removing Recursion to obtain Iterative Algorithm

Name subproblems and write recurrence relation
LCS(i, j): LCS of $X[i..m], Y[j..n]$

Removing Recursion to obtain Iterative Algorithm

LCS$(X[1..m], Y[1..n])$
  int $M[1..m+1][1..n+1]$
  for $i = 1$ to $m+1$ **do** $M[i, n+1] = 0$
  for $j = 1$ to $n+1$ **do** $M[m+1, j] = 0$

  for $i = m$ down to 1 **do**
    for $j = n$ down to 1 **do**

$$M[i][j] = \max \begin{cases} (X[i] =? Y[j])(1 + M[i+1][j+1]), \\ M[i+1][j], \\ M[i][j+1] \end{cases}$$

Removing Recursion to obtain Iterative Algorithm

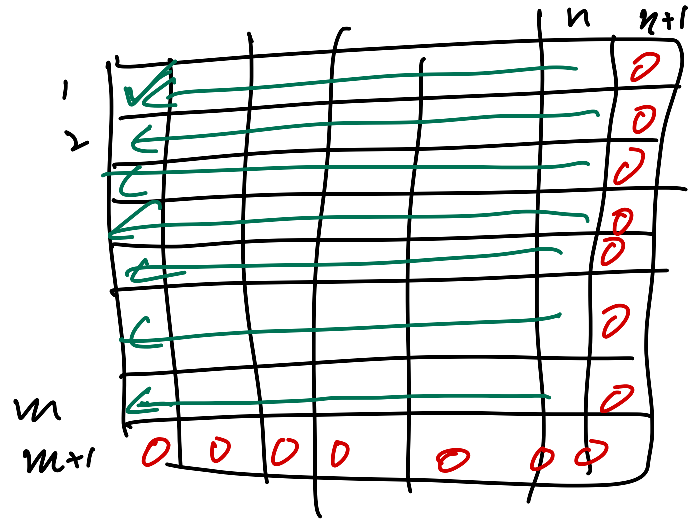
LCS($X[1..m]$, $Y[1..n]$)
    int $M[1..m+1][1..n+1]$
    for $i = 1$ `to` $m+1$ **do** $M[i, n+1] = 0$
    for $j = 1$ `to` $n+1$ **do** $M[m+1, j] = 0$

    for $i = m$ `down to` 1 **do**
        for $j = n$ `down to` 1 **do**

$$M[i][j] = \max \begin{cases} (X[i] =? Y[j])(1 + M[i+1][j+1]), \\ M[i+1][j], \\ M[i][j+1] \end{cases}$$

Analysis

1. Running time is $O(mn)$.
2. Space used is $O(mn)$. Can be reduced to $O(m+n)$.



LCS(1,1) $x_1, \ldots, x_m$ $y_1, \ldots, y_n$.

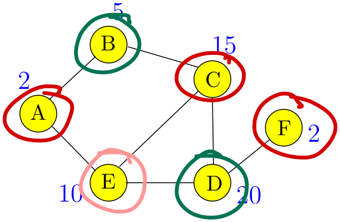
Part II

Maximum Weighted Independent Set in Trees

Maximum Weight Independent Set Problem

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

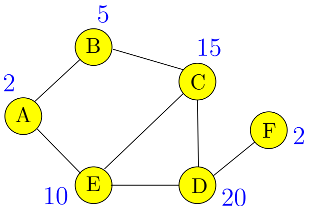
Goal Find maximum weight independent set in G



Maximum Weight Independent Set Problem

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

Goal Find maximum weight independent set in G

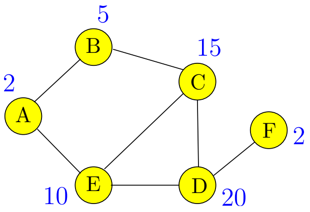


Maximum weight independent set in above graph: $\{B, D\}$

Maximum Weight Independent Set Problem

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

Goal Find maximum weight independent set in G



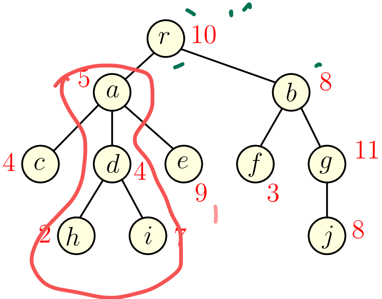
Maximum weight independent set in above graph: $\{B, D\}$

NP-Hard problem in general graphs.

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: ??

Towards a Recursive Solution

For an arbitrary graph G:

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

Towards a Recursive Solution

For an arbitrary graph G:

1. Number vertices as $v_1, v_2, \ldots, v_n$
2. Find recursively optimum solutions without v_1 (recurse on $G - v_1$) and with v_1 (recurse on $G - v_1 - N(v_1)$ & include v_1).
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?

Towards a Recursive Solution

For an arbitrary graph G:

1. Number vertices as $v_1, v_2, \ldots, v_n$
2. Find recursively optimum solutions without v_1 (recurse on $G - v_1$) and with v_1 (recurse on $G - v_1 - N(v_1)$ & include v_1).
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_1 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 $r \notin \mathcal{O}$: 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 $r \not\in \mathcal{O}$: 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 $r \not\in \mathcal{O}$: 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.

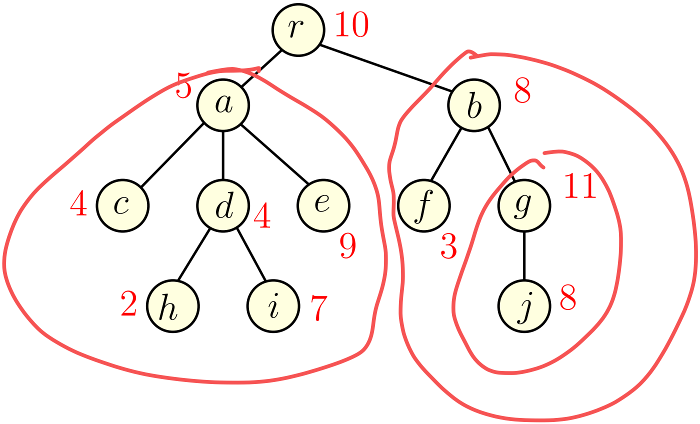
Case $r \notin \mathcal{O}$: Then $\mathcal{O}$ contains an optimum solution for each subtree of T hanging at a child of r.

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

Iterative Algorithm

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

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

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

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:

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:

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:

1. Naive bound: $O(n^2)$ since each $M[v_i]$ evaluation may take $O(n)$ time and there are n evaluations.

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
```

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

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

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

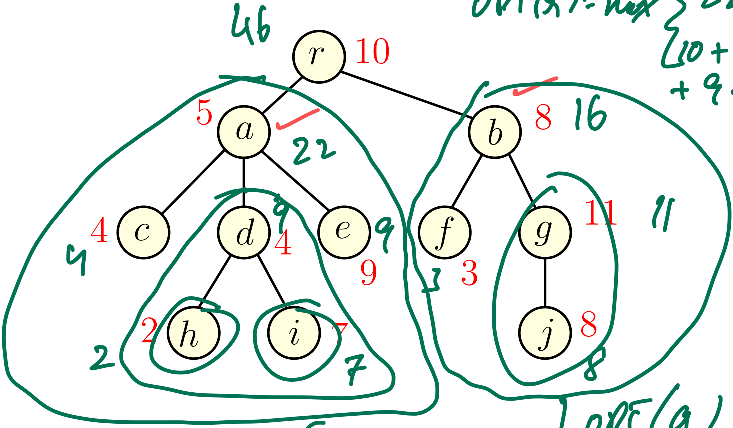
Running time:

1. Naive bound: $O(n^2)$ since each $M[v_i]$ evaluation may take $O(n)$ time and there are n evaluations.
2. Better bound: $O(n)$. A value $M[v_j]$ is accessed only by its parent and grand parent.

Example

$$OPT(r) = \max \begin{cases} 22+16 \quad (38) \\ 10+4+9+9+3+11 \end{cases}$$

46



$$OPT(d) = \max \begin{cases} 2+7 \\ 4+0 \end{cases}$$

$$OPT(a) = \max \begin{cases} 4+9+9 \\ 5+2+7 \end{cases}$$

Takeaway Points

1. Dynamic programming is based on finding a recursive way to solve the problem. Need a recursion that generates a small number of subproblems.
2. Given a recursive algorithm there is a natural DAG associated with the subproblems that are generated for given instance; this is the dependency graph. An iterative algorithm simply evaluates the subproblems in some topological sort of this DAG.
3. The space required can be reduced in some cases by a careful examination of the dependency DAG of the subproblems, and keeping only a subset of the DAG during the computation.
4. The time required can be reduced in some cases by a careful examination of the computation of the iterative algorithm and using data structures and other techniques.