Even More on Dynamic Programming

Lecture 15

Thursday, October 17, 2024

LaTeXed: August 25, 2024 14:22

Part I

Longest Common Subsequence Problem

The LCS Problem

Definition 15.1.

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

Example 15.2.

LCS between `ABAZDC` and `BACBAD` is 4 via `ABAD`

Derive a dynamic programming algorithm for the problem.

The LCS Problem

Definition 15.1.

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

Example 15.2.

LCS between `ABAZDC` and `BACBAD` is4 via `ABAD`

Derive a dynamic programming algorithm for the problem.

The LCS Problem

Definition 15.1.

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

Example 15.2.

LCS between `ABAZDC` and `BACBAD` is4 via `ABAD`

Derive a dynamic programming algorithm for the problem.

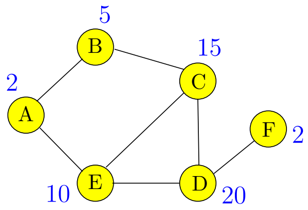
Part II

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

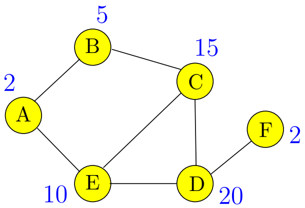


Maximum weight independent set in above graph: $\{B, D\}$

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

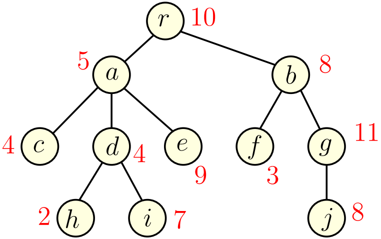


Maximum weight independent set in above graph: $\{B, D\}$

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_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

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

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 $r \notin \mathcal{O}$: 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? $O(n)$

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? $O(n)$

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? $O(n)$

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? $O(n)$

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.

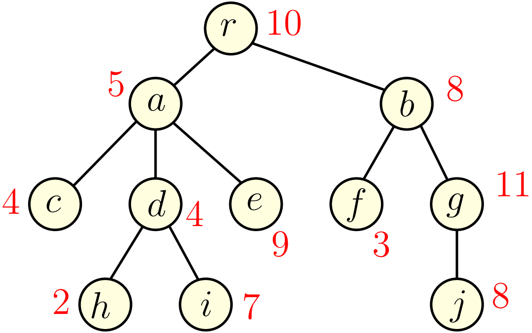
Case $r \notin \mathcal{O}$: Then $\mathcal{O}$ contains an optimum solution for each subtree of T hanging at a child of r.

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 $T(u)$

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

A Recursive Solution

$\boldsymbol{T(u)}$: subtree of $\boldsymbol{T}$ hanging at node $\boldsymbol{u}$
$\boldsymbol{OPT(u)}$: max weighted independent set value in $\boldsymbol{T(u)}$

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

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? Post-order traversal of a tree.

Iterative Algorithm

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

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.

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: $O(n^2)$ 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.

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: $O(n^2)$ 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.

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: $O(n^2)$ 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.

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( sum_{v_j child of v_i} M[v_j],
                      w(v_i) + sum_{v_j grandchild of v_i} M[v_j] )
    return M[v_n] (* Note: v_n is the root of T *)
```

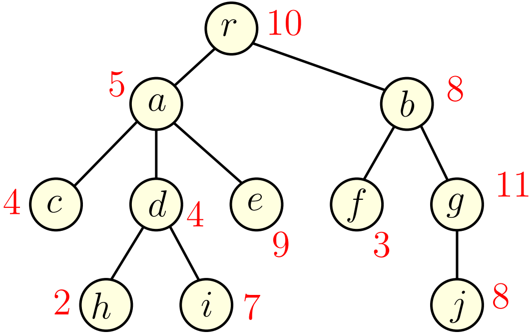
$$M[v_i] = \max\begin{pmatrix} \sum_{v_j \text{ child of } v_i} M[v_j], \\ w(v_i) + \sum_{v_j \text{ grandchild of } v_i} M[v_j] \end{pmatrix}$$

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



Part III

Context free grammars: The CYK Algorithm

Parsing

We saw regular languages and context free languages.

Most programming languages are specified via context-free grammars. Why?

- CFLs are sufficiently expressive to support what is needed.
- At the same time one can "efficiently" solve the parsing problem: given a string/program w, is it a valid program according to the CFG specification of the programming language?

CFG specification for C

```
<relational-expression> ::= <shift-expression>
                          | <relational-expression> < <shift-expression>
                          | <relational-expression> > <shift-expression>
                          | <relational-expression> <= <shift-expression>
                          | <relational-expression> >= <shift-expression>

<shift-expression> ::= <additive-expression>
                     | <shift-expression> << <additive-expression>
                     | <shift-expression> >> <additive-expression>

<additive-expression> ::= <multiplicative-expression>
                        | <additive-expression> + <multiplicative-expression>
                        | <additive-expression> - <multiplicative-expression>

<multiplicative-expression> ::= <cast-expression>
                              | <multiplicative-expression> * <cast-expression>
                              | <multiplicative-expression> / <cast-expression>
                              | <multiplicative-expression> % <cast-expression>

<cast-expression> ::= <unary-expression>
                    | ( <type-name> ) <cast-expression>

<unary-expression> ::= <postfix-expression>
                     | ++ <unary-expression>
                     | -- <unary-expression>
                     | <unary-operator> <cast-expression>
                     | sizeof <unary-expression>
                     | sizeof <type-name>

<postfix-expression> ::= <primary-expression>
                       | <postfix-expression> [ <expression> ]
                       | <postfix-expression> ( {<assignment-expression>}* )
```

Algorithmic Problem

Given a CFG $G = (V, T, P, S)$ and a string $w \in T^*$, is $w \in L(G)$?

- That is, does S derive w?
- Equivalently, is there a parse tree for w?

Simplifying assumption: G is in Chomsky Normal Form (CNF)

- Productions are all of the form $A \to BC$ or $A \to a$.
  If $\epsilon \in L$ then $S \to \epsilon$ is also allowed.
  (This is the only place in the grammar that has an ε.)
- Every CFG G can be converted into CNF form via an efficient algorithm
- Advantage: parse tree of constant degree.

Algorithmic Problem

Given a CFG $G = (V, T, P, S)$ and a string $w \in T^*$, is $w \in L(G)$?

- That is, does S derive w?
- Equivalently, is there a parse tree for w?

Simplifying assumption: G is in Chomsky Normal Form (CNF)

- Productions are all of the form $A \to BC$ or $A \to a$.
  If $\epsilon \in L$ then $S \to \epsilon$ is also allowed.
  (This is the only place in the grammar that has an ε.)
- Every CFG G can be converted into CNF form via an efficient algorithm
- Advantage: parse tree of constant degree.

Example

$S \to \epsilon \mid AB \mid XB$
$Y \to AB \mid XB$
$X \to AY$
$A \to \mathbf{0}$
$B \to \mathbf{1}$

Question:

- Is $\mathbf{000111}$ in $L(G)$?
- Is $\mathbf{00011}$ in $L(G)$?

Towards Recursive Algorithm

Assume G is a CNF grammar.
S derives w iff one of the following holds:

- $|w| = 1$ and $S \to w$ is a rule in P
- $|w| > 1$ and there is a rule $S \to AB$ and a split $w = uv$ with $|u|, |v| \geq 1$ such that A derives u and B derives v

Observation: Subproblems generated require us to know if some non-terminal A will derive a substring of w.

Towards Recursive Algorithm

Assume G is a CNF grammar.
S derives w iff one of the following holds:

- $|w| = 1$ and $S \to w$ is a rule in P
- $|w| > 1$ and there is a rule $S \to AB$ and a split $w = uv$ with $|u|, |v| \geq 1$ such that A derives u and B derives v

Observation: Subproblems generated require us to know if some non-terminal A will derive a substring of w.

Recursive solution

1. Input: $w = w_1 w_2 \ldots w_n$
2. Assume r non-terminals in G: $R_1, \ldots, R_r$.
3. R_1: Start symbol.
4. $f(\ell, s, b)$: TRUE $\iff$ $w_s w_{s+1} \ldots, w_{s+\ell-1} \in L(R_b)$.
   $=$ Substring w starting at pos ℓ of length s is deriveable by R_b.
5. **Recursive formula:** $f(1, s, a)$ is 1 iff $\left(R_a \to w_s\right) \in G$.
6. For $\ell > 1$:

$$f(\ell, s, a) = \bigvee_{p=1}^{\ell-1} \bigvee_{(R_a \to R_b R_c) \in G} \Big(f(p, s, b) \wedge f(\ell - p, s + p, c)\Big)$$

7. **Output:** $w \in L(G) \iff f(n, 1, 1) = 1$.

Recursive solution

1. Input: $w = w_1 w_2 \dots w_n$
2. Assume r non-terminals in G: $R_1, \dots, R_r$.
3. R_1: Start symbol.
4. $f(\ell, s, b)$: TRUE $\iff$ $w_s w_{s+1} \dots, w_{s+\ell-1} \in L(R_b)$.
   $=$ Substring w starting at pos ℓ of length s is deriveable by R_b.
5. **Recursive formula:** $f(1, s, a)$ is 1 iff $\left(R_a \to w_s\right) \in G$.
6. For $\ell > 1$:

$$f(\ell, s, a) = \bigvee_{p=1}^{\ell-1} \bigvee_{(R_a \to R_b R_c) \in G} \Big(f(p, s, b) \wedge f(\ell - p, s + p, c) \Big)$$

7. **Output:** $w \in L(G) \iff f(n, 1, 1) = 1$.

Analysis

Assume $G = \{R_1, R_2, \ldots, R_r\}$ with start symbol R_1

- Number of subproblems: $O(rn^2)$
- Space: $O(rn^2)$
- Time to evaluate a subproblem from previous ones: $O(|P|n)$ where P is set of rules
- Total time: $O(|P|rn^3)$ which is polynomial in both $|w|$ and $|G|$. For fixed G the run time is cubic in input string length.
- Running time can be improved to $O(n^3|P|)$.
- Not practical for most programming languages. Most languages assume restricted forms of CFGs that enable more efficient parsing algorithms.

CYK Algorithm

```
Input string: X = x_1 ... x_n.
Input grammar G: r nonterminal symbols R_1...R_r, R_1 start symbol.
```

```
P[n][n][r]: Array of booleans. Initialize all to FALSE
for s = 1 to n do
    for each unit production R_v → x_s do
        P[1][s][v] ← TRUE
for ℓ = 2 to n do    // Length of span
    for s = 1 to n − ℓ + 1 do    // Start of span
        for p = 1 to ℓ − 1 do    // Partition of span
            for all (R_a → R_b R_c) ∈ G do
                if P[p][s][b] and P[l − p][s + p][c] then
                    P[l][s][a] ← TRUE
if P[n][1][1] is TRUE then
    return ``X is member of language''
else
    return ``X is not member of language''
```

Example

$S \to \epsilon \mid AB \mid XB$
$Y \to AB \mid XB$
$X \to AY$
$A \to \mathbf{0}$
$B \to \mathbf{1}$

Question:

- Is $\mathbf{000111}$ in $L(G)$?
- Is $\mathbf{00011}$ in $L(G)$?

Order of evaluation for iterative algorithm: increasing order of substring length.

Example

$S \rightarrow \epsilon \mid AB \mid XB$
$Y \rightarrow AB \mid XB$
$X \rightarrow AY$
$A \rightarrow \mathbf{0}$
$B \rightarrow \mathbf{1}$

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 to evaluate the answer can be reduced in some cases by a careful examination of that dependency DAG of the subproblems and keeping only a subset of the DAG at any time.