Dynamic programming

CS325
Shortest path in DAG

- **DAG**: directed acyclic graph

- A DAG can be linearized: all the nodes arranged in a line and the edges only goes from left to right (Chapter 3.3.2)
Consider the shortest path from node S to D, there are two ways of reaching D:
- Via \((B, D)\): \(\text{Dist}(B) + 1\)
- Via \((C, D)\): \(\text{Dist}(C) + 3\)

\[
\text{Dist}(D) = \min(\text{Dist}(B) + 1, \text{Dist}(C) + 3)
\]

Now let’s compute the \(\text{Dist}\) values in a left to right order:

\[
\begin{align*}
\text{Dist}(S) &= \\
\text{Dist}(C) &= \\
\text{Dist}(A) &= \\
\text{Dist}(B) &= \\
\text{Dist}(D) &= \\
\text{Dist}(E) &=
\end{align*}
\]
Dynamic Programming

• A general and powerful technique

• Given a problem, we identify a collection of subproblems (polynomially many) such that there is a recurrence relation that allows you to derive solution to a larger problem from solutions to smaller problems

• By solving the problems from the smallest to the largest, we achieve poly run time

• This process may require quite bit of practice to get comfortable
Longest Increasing Subsequences

Problem:
Given a sequence of numbers $\alpha_1, \alpha_2, \ldots, \alpha_n$, find the longest increasing subsequence (LIS)
Increasing Subsequences Represented as a DAG

- An increasing subsequence corresponds to a path
- Longest increasing subsequence = longest path in the graph

\[
\text{for } j = 1, 2, \ldots, n:
L(j) = 1 + \max\{L(i) : (i, j) \in E\}
\text{return } \max_j L(j)
\]
$L(i)$: the longest increasing subsequence ending at position $i$

for $j = 1, 2, \ldots, n$:

$L(j) = 1 + \max \{ L(i) : (i, j) \in E \}$

return $\max_j L(j)$

$L(1) = 1; L(2) = 1$

$L(3) = \max(L(1) + 1, L(2) + 1) = 2$

$L(4) = \max(L(1) + 1, L(2) + 1) = 2$

$L(5) = L(2) + 1 = 2$

$L(6) = \max(L(1) + 1, L(2) + 1, L(5) + 1) = 3$

$L(7) = \max(L(1) + 1, L(2) + 1, L(3) + 1, L(4) + 1, L(5) + 1, L(6) + 1) = 4$

$L(8) = \max(L(1) + 1, L(2) + 1, L(4) + 1, L(5) + 1, L(6) + 1) = 4$
Run time analysis

- n iterations
- Iteration i: worst case taking max over i-1 possibilities
- Total runtime: $O(n^2)$
Alternative way of arriving at the solution

• Given an input $a_1, a_2, ..., a_n$, let’s consider an optimal solution to this problem

• If a magical ball tells you that this solution ends with $a_j$, let’s denote its length as $L(j)$

• What can we say about $L(j)$?
  – $L(j) = \max(L(i) + 1: i < j \text{ and } a_i < a_j)$
  – By construction of the graph, this is the same as
    $$L(j) = \max(L(i) + 1: (i, j) \in E)$$

• Now since we don’t know what $j$ is, we just need to compute $L_j$ for all $j$’s and return the largest one
  – Gives us the same algorithm, without explicit reduction to a graph problem
Recursion vs. Iteration

- The presented algorithm uses iterations, alternatively we could use recursion to compute $L(j)$
  
  $$L(j) = \max(L(i) + 1: i < j \text{ and } a_i < a_j)$$

- In the worst case, this is what the computation may look like:

![Tree Diagram]

  * Exponentially many nodes in the tree
  * Each recursion, the problem size is reduced only slightly (unlike D&C)
  * Many repeated sub-problems
  * Dynamic programming obtain efficiency by explicitly enumerate all distinct sub-problems and solve them in the right order
Edit Distance

Background:
- spell checker uses edit distance to identify the word in the dictionary that is most similar to the one spelled
- Sequence alignment, widely used in computational biology

Problem:
Given two strings: $x[1, \ldots, m]$ and $y[1, \ldots, n]$ there are different ways of aligning them. The cost of an alignment is the total number of mismatched positions.

```
S - N O W Y
S U N N - Y
Cost: 3
```
```
- S N O W - Y
S U N - - N Y
Cost: 5
```

Goal: find the alignment with the smallest cost, which is called the **Edit Distance between the two strings** as it corresponds to the minimum # of edits (insertion, deletion, substitution) required to make two strings identical.
Developing a DP solution

• Given inputs $x[1, \ldots, m]$ and $y[1, \ldots, n]$, let’s think about the optimal solution, and how it treats $x[m]$ and $y[n]$

• There are three possibilities:

  1. $x[m]$ is matched with $y[n]$
  2. $x[m]$ is matched with a gap “—”
  3. $y[n]$ is matched with a gap “—”

– Case 1: $D(x[1, \ldots, m - 1], y[1, \ldots, n - 1]) + \text{diff}(x[m], y[n])$
– Case 2: $D(x[1, \ldots, m - 1], y[1, \ldots, n]) + 1$
– Case 3: $D(x[1, \ldots, m], y[1, \ldots, n - 1]) + 1$
– $D(x[1, \ldots, m], y[1, \ldots, n]) = \text{minimum over three cases}$
Generalize

Defining the subproblems:
• Let $E(i, j) = D(x[1, \ldots, i], y[1, \ldots, j])$ for $1 \leq i \leq m, 1 \leq j \leq n$

Write out the recurrence relation:
for $i = 1, \ldots, m, j = 1, \ldots, n$

$$E(i, j) = \min\{1 + E(i - 1, j), 1 + E(i, j - 1), E(i - 1, j - 1) + \text{diff}(i, j)\}$$
• Add the initialization and final output to complete the algorithm

\[
\begin{align*}
\text{for } i = 0, 1, 2, \ldots, m: \\
E(i, 0) &= i \\
\text{for } j = 1, 2, \ldots, n: \\
E(0, j) &= j \\
\text{for } i = 1, 2, \ldots, m: \\
\quad \text{for } j = 1, 2, \ldots, n: \\
E(i, j) &= \min\{E(i - 1, j) + 1, E(i, j - 1) + 1, E(i - 1, j - 1) + \text{diff}(i, j)\} \\
\text{return } E(m, n)
\end{align*}
\]
Example: EXPONENTIAL vs. POLYNOMIAL

<table>
<thead>
<tr>
<th>E</th>
<th>−</th>
<th>P</th>
<th>O</th>
<th>L</th>
<th>Y</th>
<th>N</th>
<th>O</th>
<th>M</th>
<th>I</th>
<th>A</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>O</td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N</td>
<td></td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td></td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I</td>
<td></td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td></td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Weighted Interval Scheduling

- Recall the interval scheduling problem
- Now requests have added weights (representing value)

$v_1 = 1$
$v_2 = 3$
$v_3 = 1$

Greedy algorithms do not work for this problem
Developing a DP Solution

- Given a set of requests $R = \{(s_1, f_1, v_1), \ldots, (s_n, f_n, v_n)\}$, let’s assume that they are sorted: $f_1 \leq f_2 \leq \cdots \leq f_n$

- Let’s think about an optimal solution $O$, whether it contains the $n$-th request:
  - Case 1: $O$ contains request $n$

  $O - \{n\}$ must be the optimal solution to the following set of requests $R' = \{1, \ldots, p(n)\}$, where $p(n)$ is the last request that does not intersect with $n$

  - Case 2: $O$ does not contain request $n$

  $O$ must be the optimal solution to $R - \{n\}$
Generalize

Defining the subproblems (assuming $f_1 \leq f_2 \leq \cdots \leq f_n$):

• Let $O(i)$ denote the optimal solution for request $1, 2, \ldots, i$
• Let $OPT(i)$ be its value

(Let $p(i)$ denote the last request that does not intersect with request $i$)

Write out the recurrence relation:

for $i = 1, \ldots, n$

$$OPT(i) = \max(OPT(i - 1), OPT(p(i)) + v_i)$$
• Add the initialization and final output to complete the algorithm

\[ OPT(0) = 0 \]

for \( i = 1, \ldots, n \)

\[ OPT(i) = \max(OPT(i - 1), OPT(p(i)) + v_i) \]

end for
\[ v_1 = 2 \]
\[ v_2 = 4 \]
\[ v_3 = 4 \]
\[ v_4 = 7 \]
\[ v_5 = 2 \]
\[ v_6 = 1 \]

- \( OPT(0) = 0(0) = \)
- \( OPT(1) = 0(1) = \)
- \( OPT(2) = 0(2) = \)
- \( OPT(3) = 0(3) = \)
- \( OPT(4) = 0(4) = \)
- \( OPT(5) = 0(5) = \)
- \( OPT(6) = 0(6) = \)
• Add the initialization and final output to complete the algorithm

\[ OPT(0) = 0 \]

for \( i = 1, \ldots, n \)

if \( OPT(i - 1) > OPT(p(i)) + v_i \)

\[ OPT(i) = OPT(i - 1) \]
\[ O(i) = \text{__________} \]

else

\[ OPT(i) = OPT(p(i)) + v_i \]
\[ O(i) = \text{__________} \]
end for

Return \( OPT(n) \) and \( O(n) \)
Recursive vs. Iterative

\[ OPT(0) = 0 \]

for \( i = 1, \ldots, n \)

\[ OPT(i) = \max(OPT(i - 1), OPT(p(i)) + v_i) \]

end for

\[ OPT(n) \]

if \( n = 0 \)

return 0

else

return \( \max(OPT(i - 1), OPT(p(i)) + v_i) \)

end if
Knapsack problem (with repetitions)

• In a robbery, a burglar has a bag that can hold a total of $W$ pounds. There are $n$ items to choose from: $(v_1, w_1), \ldots, (v_n, w_n)$, each item could be repeated selected

• What is the most valuable set of items that can fit into the bag?

<table>
<thead>
<tr>
<th>Item</th>
<th>Weight</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>$30</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>$14</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>$16</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>$9</td>
</tr>
</tbody>
</table>

• Does greedy work?
Developing a DP Solution

• Given a set of items \((v_1, w_1), \ldots, (v_n, w_n)\), and a weight limit \(W\), how can we make the problem smaller?

• Possibility 1: make the set of items smaller

• Possibility 2: make the weight limit smaller
Iterative vs. Memoization

• Iterative:

\[
\begin{align*}
K(0) &= 0 \\
\text{for } w = 1 \text{ to } W: & \\
K(w) &= \max\{K(w - w_i) + v_i : w_i \leq w\} \\
\text{return } K(W)
\end{align*}
\]

• Recursive with memoization

```python
function knapsack(w):
    if w is in hash table: return K(w)
    K(w) = \max\{knapsack(w - w_i) + v_i : w_i \leq w\}
    insert K(w) into hash table, with key w
    return K(w)
```

• Runtime? Trade-off?