Dynamic Programming

Outline

• Intro
• 0-1 Knapsack
• Longest common subsequence
• Bellman-Ford (single source shortest path)
• Floyd-Warshall (all pairs shortest path)
Intro

Two key ingredients

- Two key ingredients for an optimization problem to be suitable for a dynamic programming solution
  
  1. optimal substructure
  2. overlapping sub-problems

Each substructure is optimal (principle of optimality)

Sub-problems are dependent
Three basic components

• The development of a dynamic programming algorithm has three basic components
  – a recurrence relation (for defining the value/cost of an optimal solution)
  – a tabular computation (for computing the value of an optimal solution)
  – a trace-back procedure (for delivering an optimal solution)

0-1 Knapsack
The Knapsack Problem

• A thief robbing a store finds \( n \) items
• The \( i \)th item is worth \( b_i \) and weighs \( w_i \) pounds
• Thief’s knapsack can carry at most \( W \) pounds
• \( b_i, w_i \) and \( W \) are integers
• Problem: What items to select to maximize profit?

The 0-1 Knapsack Problem

• Each item must be either taken or left behind (a binary choice of 0 or 1)
• Exhibits \textit{optimal substructure} property (for the same reason as for the fractional)
• 0-1 knapsack problem however \textit{cannot} be solved by a greedy strategy
• Can be solved (less) efficiently by \textit{dynamic programming}
0-1 Knapsack Problem

- Let $x_i = 1$ denote item $i$ is in the knapsack, $x_i = 0$ denote item $i$ is not in the knapsack
- Problem stated formally as follows

\[
\begin{align*}
\text{maximize} & \quad \sum_{i=1}^{n} b_i x_i \quad \text{(total profit)} \\
\text{subject to} & \quad \sum_{i=1}^{n} w_i x_i \leq W \quad \text{(weight constraint)}
\end{align*}
\]

Define the problem recursively ...

- Consider the first item $i=1$
  1. If it is selected (in the knapsack)

\[
\begin{align*}
\text{maximize} & \quad \sum_{i=2}^{n} b_i x_i \quad \text{subject to} \quad \sum_{i=2}^{n} w_i x_i \leq W - w_i
\end{align*}
\]
  2. If it is not selected (not in the knapsack)

\[
\begin{align*}
\text{maximize} & \quad \sum_{i=2}^{n} b_i x_i \quad \text{subject to} \quad \sum_{i=2}^{n} w_i x_i \leq W
\end{align*}
\]
- Compute both cases, select the better one
Recursive Solution

- Let us define $P[i,k]$ as the maximum profit possible using items $\{i, i+1, \ldots, n\}$ and residual (knapsack) capacity $k$
- We can define $P[i,k]$ recursively as follows

$$P[i,k] = \begin{cases} 
0 & i = n \land w_n > k \\
b_n & i = n \land w_n \leq k \\
P[i+1,k] & i < n \land w_i > k \\
\max\{P[i+1,k], b_i + P[i+1,k-w_i]\} & i < n \land w_i \leq k
\end{cases}$$
0-1 knapsack (recursive) in Python

```python
def knapsack(items, i, k):
    n = len(items)
    if i == n:
        return b(items[n-1]) if w(items[n-1]) <= k else 0
    if w(items[i-1]) > k:
        return knapsack(items, i+1, k)
    else:
        return max(knapsack(items, i+1, k),
                   b(items[i-1]) + knapsack(items, i+1, k-w(items[i-1])))
```

Remark: i < n

Recursive Solution

- We can write an algorithm for the recursive solution based on the four cases
- Recursive algorithm will take $O(2^n)$ time
- Inefficient because $P[i,k]$ for the same $i$ and $k$ will be computed many times
- Example
  - $n=5$, $W=10$, $w=[2, 2, 6, 5, 4]$, $b=[6, 3, 5, 4, 6]$
Dynamic Programming Solution

- The inefficiency could be overcome by computing each $P[i,k]$ once and storing the result in a table for future use.
- The table is filled for $i=n,n-1,\ldots,2,1$ in that order for $1 \leq k \leq W$.
- First row (initialization)

<table>
<thead>
<tr>
<th>$k$</th>
<th>1</th>
<th>2</th>
<th>$w_n-1$</th>
<th>$w_n$</th>
<th>$w_n+1$</th>
<th>...</th>
<th>$W$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P[n,k]$</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td>0</td>
<td>$b_n$</td>
<td>$b_n$</td>
<td>...</td>
</tr>
</tbody>
</table>

$$w = [2, 2, 6, 5, 4] \quad b = [6, 3, 5, 4, 6]$$
Example

\(n=5, \ W=10, \ w = [2, 2, 6, 5, 4], \ b = [2, 3, 5, 4, 6]\)

<table>
<thead>
<tr>
<th>(i/k)</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(P[i,k] = \max\{P[i+1,k], \ b_i + P[i+1,k-w_i]\}\)
Example

\(n=5, \ W=10, \ w = [2, 2, 6, 5, 4], \ b = [2, 3, 5, 4, 6]\)

\[\begin{array}{cccccccccc}
\text{i} \text{k} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\hline
5 & 0 & 0 & 0 & 0 & 6 & 6 & 6 & 6 & 6 & 6 & 6 \\
4 & 0 & 0 & 0 & 0 & 6 & 6 & 6 & 6 & \textcolor{red}{10} & \textcolor{red}{10} & \\
3 & 0 & 0 & 0 & 0 & 6 & 6 & 6 & 6 & 10 & \textcolor{red}{11} & \\
2 & \\
1 & \\
\end{array}\]

\(P[i,k] = \max\{P[i+1,k], \ b_i + P[i+1,k-w_i]\}\)

Example

\(n=5, \ W=10, \ w = [2, 2, 6, 5, 4], \ b = [2, 3, 5, 4, 6]\)

\[\begin{array}{cccccccccc}
\text{i} \text{k} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\hline
5 & 0 & 0 & 0 & 0 & 6 & 6 & 6 & 6 & 6 & 6 & 6 \\
4 & 0 & 0 & 0 & 0 & 6 & 6 & 6 & 6 & \textcolor{red}{10} & \textcolor{red}{10} & \\
3 & 0 & 0 & 0 & 0 & 6 & 6 & 6 & 6 & 10 & \textcolor{red}{11} & \\
2 & 0 & 0 & 3 & 3 & 6 & 6 & 9 & 9 & 9 & 10 & 11 \\
1 & \\
\end{array}\]

\(P[i,k] = \max\{P[i+1,k], \ b_i + P[i+1,k-w_i]\}\)
Example

\( n=5, \ W=10, \ w = [2, 2, 6, 5, 4], \ b = [2, 3, 5, 4, 6] \)

\[
P[i,k] = \max \{P[i+1,k], \ b_i + P[i+1,k-w_i]\}\]

Example

\( n=5, \ W=10, \ w = [2, 2, 6, 5, 4], \ b = [2, 3, 5, 4, 6] \)

\[
x = [0,0,1,0,1] \quad x = [1,1,0,0,1] \]
0-1 knapsack in Python (dyn prog)

def knapsack(items, w):
P, n = {}, len(items)
for j in range(w+1):
P[n, j] = b(items[n-1]) if w(items[n-1])<=j else 0
for i in range(len(items)-1, 0, -1):
    for j in range(w+1):
        if w(items[i-1])>j:
P[i, j] = P[i+1, j]
        else:
P[i, j] = max(P[i+1, j],
b(items[i-1])+P[i+1, j-w(items[i-1])])
return P

Time complexity

• Running time: $O(nW)$

• Technically, this is not a poly-time algorithm

• These class of algorithms are called *pseudo-polynomial*
Longest common subsequence

Longest Common Subsequence

A sequence $Z = \langle z_1, z_2, \ldots, z_k \rangle$ is a subsequence of a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$ if $Z$ can be generated by striking out some (or none) elements from $X$.

For example, $\langle b, c, d, b \rangle$ is a subsequence of $\langle a, b, c, a, d, c, a, b \rangle$. 
Longest Common Subsequence

The **longest common subsequence problem** is the problem of finding, for given two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$, a maximum-length common subsequence of $X$ and $Y$.

- For example, given
  $X = B D C A B A$
  $Y = A B C B D A B$
  $Z = \text{LCS}(X, Y) = BCBA$
  $X =$
  
  $Y =$

- $X =$
  $Y =$
Optimal Substructure

**Theorem.** Let $Z = <z_1, \ldots, z_k>$ be any LCS of $X$ and $Y$.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and $Z_{k-1}$ is an LCS of $X_{m-1}$ and $Y_{n-1}$
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that $Z$ is an LCS of $X_{m-1}$ and $Y$
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that $Z$ is an LCS of $X$ and $Y_{n-1}$

**Proof:** (case 1: $x_m = y_n$)

If $z_k \neq x_m$, we could append $x_m = y_n$ to $Z$ to obtain a CS of $X$ and $Y$ of length $k+1$, which contradicts the optimality of $Z$. Thus we must have that $z_k = x_m = y_n$.

Let $Z_{k-1}$ be a length-$(k-1)$ common subsequence of $X_{m-1}$ and $Y_{n-1}$. $Z_{k-1}$ must be an LCS of $X_{m-1}$ and $Y_{n-1}$. If $W$ is a common subsequence of $X_{m-1}$ and $Y_{n-1}$ longer than $k-1$, appending $x_m = y_n$ to $W$ would make $W$ longer than $Z$.

---

**Optimal Substructure**

**Theorem.** Let $Z = <z_1, \ldots, z_k>$ be any LCS of $X$ and $Y$.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and $Z_{k-1}$ is an LCS of $X_{m-1}$ and $Y_{n-1}$
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that $Z$ is an LCS of $X_{m-1}$ and $Y$
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that $Z$ is an LCS of $X$ and $Y_{n-1}$

**Proof:** (case 2: $x_m \neq y_n$ and $z_k \neq x_m$)

Since $Z$ does not end in $x_m$, then $Z$ is a common subsequence of $X_{m-1}$ and $Y$.

$Z$ is a longest common subsequence because if there was a common subsequence $W$ with length greater than $k$, $W$ would also be a common subsequence of $X_m$ and $Y$, contradicting the optimality of $Z$.

(case 3 is symmetric to case 2)
Recursive Formulation

- Define $c[i, j] = \text{length of LCS of } X_i \text{ and } Y_j$

$$c[i, j] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0, \\
(c[i - 1, j - 1] + 1) & \text{if } i, j > 0 \text{ and } x_i = y_j, \\
\max(c[i - 1, j], c[i, j - 1]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j.
\end{cases}$$

- We want $c[m, n]$
- This gives a recursive algorithm and solves the problem
- But is it efficient?

Example

$$c[\alpha, \beta] = \begin{cases} 
0 & \text{if } \alpha \text{ empty or } \beta \text{ empty}, \\
c[\text{prefix}\alpha, \text{prefix}\beta] + 1 & \text{if } \text{end}(\alpha) = \text{end}(\beta), \\
\max(c[\text{prefix}\alpha, \beta], c[\alpha, \text{prefix}\beta]) & \text{if } \text{end}(\alpha) \neq \text{end}(\beta).
\end{cases}$$

```
c[springtime, printing]
  /   \
[c[springtim, printing]   c[springtime, printin]
    /   \
[springti, printing] [springtim, printin] [springt, printing] [springti, printin]
```

```markdown
[c[springtime, printin]
  /   \
[springt, printing] [springti, printin] [springtim, printi] [springtime, print]
```
LCS in Python

```python
def LCS(X, Y):
c = {}
for i in range(len(X)+1):
    for j in range(len(Y)+1):
        if i == 0 or j == 0:
            c[i,j] = 0
        elif X[i-1] == Y[j-1]:
            c[i,j] = c[i-1,j-1] + 1
        else:
            c[i,j] = max(c[i-1,j], c[i,j-1])
#...continues
```

Remark: $c[i,j]$ contains the length of an LCS of $X[i]$ and $Y[j]$

Time: $O(mn)$

---

Reporting the LCS in Python

```python
#...continued
i, j = len(X), len(Y)
LCS = []
while c[i,j]:
    while c[i,j] == c[i-1,j]:
        i -= 1
    while c[i,j] == c[i,j-1]:
        j -= 1
    i -= 1
    j -= 1
    LCS.append(X[i])
LCS.reverse()
return LCS
```

Remark: append matches

Time: $O(m+n)$
LCS algorithm

- Time complexity $O(nm)$
- Space complexity $O(nm)$
- Space can be reduced to linear by observing that we just need the previous row to compute the current row
- The length of the LCS can be computed easily in linear space, but how to traceback?

Bellman-Ford
Bellman-Ford Algorithm

• Dijkstra’s algorithm does not work when the weighted graph contains negative edges
  – we cannot be greedy anymore on the assumption that the lengths of paths will not decrease in the future
• Bellman-Ford algorithm detects negative cycles (returns \textit{false}) or returns the shortest path-tree

Bellman-Ford Algorithm

• Use $d[\cdot]$ labels (like in Dijkstra and Prim)
• Initialize $d[s]=0$, $d[\cdot]=\infty$ otherwise
• Perform $|V|-1$ rounds
• In each round, attempt an edge relation for all the edges in the graph
• An extra round of edge relaxation can tell the presence of a negative cycle
Bellman-Ford Algorithm

Algorithm Bellman-Ford\((G(V,E), s)\)

\[
\begin{align*}
&\text{for each vertex } u \text{ in } V \\
&\quad d[u] \leftarrow \infty \\
&\quad d[s] \leftarrow 0 \\
&\text{for } i \leftarrow 1 \text{ to } |V|-1 \text{ do} \\
&\quad \text{for each edge } (u,v) \text{ in } E \text{ do} \\
&\quad\quad \text{if } d[v] > d[u] + w(u,v) \text{ then} \\
&\quad\quad\quad d[v] \leftarrow d[u] + w(u,v) \\
&\quad \text{for each edge } (u,v) \text{ in } E \text{ do} \\
&\quad\quad \text{if } d[v] > d[u] + w(u,v) \text{ then} \\
&\quad\quad\quad \text{return } FALSE \\
&\text{return } d[], TRUE
\end{align*}
\]

Iteration 0

![Graph Diagram](image-url)
Iteration 1

Iteration 2
Iteration 3

Iteration 4
Observation: BF is dynamic programming
Subproblems: paths composed by increasing # of edges
Let \( d(i, j) \) = “cost of the shortest path from source \( s \) to vertex \( i \) that uses at most \( j \) edges/hops”

\[
d(i, j) = \begin{cases} 
0 & \text{if } i = s, j = 0 \\
\infty & \text{if } i \neq s, j = 0 \\
\min_{(k, i) \in E} \{d(k, j-1) + w((k, i)), d(i, j-1)\} & \text{if } j > 0 
\end{cases}
\]

All-pair shortest path
All-pairs shortest path

• We want to compute the shortest path distance between every pair of vertices in a directed graph $G$ (n vertices, m edges)

• We want to know $D[i,j]$ for all $i,j$, where $D[i,j]$ = shortest distance from $v_i$ to $v_j$

All-pairs shortest path

• If $G$ has no negative-weight edges, we could use Dijkstra repeatedly from each vertex
• Dijkstra runs in $O(m + n \log n)$ time
• It would take $O(n (m + n \log n))$ time, that is $O(n^2 \log n + nm)$ time, which could be as large as $O(n^3)$
All-pairs shortest path

• If $G$ has negative-weight edges (but no negative-weight cycles) we could use Bellman-Ford repeatedly from each vertex
• Bellman-Ford runs in $O(nm)$
• It would take $O(n^2m)$ time, which could be as large $O(n^4)$ time

All-pairs shortest path

• We now see an algorithm to solve the all-pairs shortest path in $O(n^3)$ time

• The graph can contain negative-weight edges (but no negative-weight cycles)
All-pairs shortest path

- Let $G = (V, E)$ a weighted directed graph
- Let $V = (v_1, v_2, ..., v_n)$
- Define cost function $D_{i,j}^k = \text{"the shortest distance from } v_i \text{ to } v_j \text{ using only vertices } \{v_1, v_2, ..., v_k\}\text{"}$

A dynamic programming shortest-path

Initially we set

$$D_{i,j}^0 = \begin{cases} 0 & \text{if } i = j \\ w((v_i, v_j)) & \text{if } (v_i, v_j) \in E \\ +\infty & \text{otherwise} \end{cases}$$
A dynamic programming shortest-path

\[ D_{i,j}^k = \min\left\{ D_{i,j}^{k-1}, D_{i,k}^{k-1} + D_{k,j}^{k-1} \right\} \]

- The cost of going from \( v_i \) to \( v_j \) using vertices \( 1, \ldots, k \) is the shorter between
  - (do not to use \( v_k \)) The shortest path from \( v_i \) to \( v_j \) using vertices \( 1, \ldots, k-1 \)
  - (use \( v_k \)) The shortest path from \( v_i \) to \( v_k \) using \( 1, \ldots, k-1 \) plus the cost of the shortest path from \( v_k \) to \( v_j \) using \( 1, \ldots, k-1 \)

Then \( D_{i,j}^k = \min\{D_{i,j}^{k-1}, D_{i,k}^{k-1} + D_{k,j}^{k-1}\} \).
All-pairs shortest path

Algorithm AllPairs(G):

Input: A weighted directed graph G with n vertices numbered v1, v2, ..., vn
Output: A matrix D such that D[i, j] is distance from vi to vj in G

for i ← 1 to n do
    for j ← 1 to n do
        if i = j then
            Set D[i, i] ← 0 and continue looping
        else
            if (vi, vj) is an edge in G then
                Set D[i, j] ← w((vi, vj))
            else
                Set D[i, j] ← ∞

for i ← 1 to n do
    for j ← 1 to n do
        for k ← 1 to n do
            Set D[i, j] ← min{D[i, j], D[i, k] + D[k, j]}

Return D

All-pairs shortest path

• Floyd-Warshall’s algorithm computes the shortest path distance between each pair of vertices of G in O(n^3) time

• FYI: when the graph is sparse consider Johnson’s algorithm, which has complexity O(n^2 log n + nm) even if there are negative weights
Reading assignment

• Chapter 15, “Dynamic Programming”
• Section 15.4, “Longest common subsequence”
• Section 15.2, “Matrix chain multiplication”
• Section 24.1, “The Bellman-Ford algorithm”
• Section 25.2, “All-pairs shortest path”