Dynamic Programming

Outline

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

• Two key ingredients for an optimization problem to be suitable for a dynamic programming solution

1. optimal substructure

   Each substructure is optimal
   (principle of optimality)

2. overlapping sub-problems

   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 *optimal substructure* property (for the same reason as for the fractional)
- 0-1 knapsack problem however *cannot* be solved by a greedy strategy
- Can be solved (less) efficiently by *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)
     \[
     \begin{align*}
     \text{maximize} & \quad \sum_{i=2}^{n} b_i x_i & \quad \text{subject to} & \quad \sum_{i=2}^{n} w_i x_i \leq W - w_1 \\
     \end{align*}
     \]
  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:

\[
P[i,k] = \begin{cases} 
0 & i = n & w_i > k \\
\frac{b_n}{P[i+1,k]} & i = n & w_i \leq k \\
\max \{P[i+1,k], b_i + P[i+1,k-w_i]\} & i < n & w_i > k \\
\end{cases}
\]
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
  - $n=5$, $W=10$, $w=[2, 2, 6, 5, 4]$, $b=[6, 3, 5, 4, 6]$
\( w = [2, 2, 6, 5, 4] \quad b = [6, 3, 5, 4, 6] \)

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>\ldots</th>
<th>( w_{n-1} )</th>
<th>( w_n )</th>
<th>( w_{n+1} )</th>
<th>\ldots</th>
<th>( W )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P[n,k] )</td>
<td>0</td>
<td>0</td>
<td>\ldots</td>
<td>0</td>
<td>( b_n )</td>
<td>( b_n )</td>
<td>\ldots</td>
<td>( b_n )</td>
</tr>
</tbody>
</table>
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

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

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

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

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

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

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)

```python
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, -1, -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- and space-complexity

- Time complexity: $O(nW)$
- Technically, this is not a polynomial time algorithm
- These class of algorithms are called pseudo-polynomial
- Space complexity: $O(nW)$
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
  
  $X = B D C A B A$
  
  $Y = A B C B D A B$

• $Z=LCS(X,Y) = BCBA$

• $X = \begin{array}{cccccc}
  B & D & C & A & B & A \\
\end{array}$

• $Y = \begin{array}{cccccc}
  A & B & C & B & D & A & B \\
\end{array}$
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 that $Z$. (case 3 is symmetric to case 2)

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 $W$ of $X_{m-1}$ and $Y$ 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 $c[i, j] =$ length of LCS of $X_i$ and $Y_j$
- We want $c[m,n]$
- This gives a recursive algorithm and solves the problem
- But is it efficient?

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

Example

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

```
[ springtime, printing ]
  /             /
c[ springtim, printing ]  c[ springtime, printin ]

[ springt, printing ] [ springti, printin ] [ springt, printi ] [ springtime, printi ]
```

Citations: 0

References: 0

Journals: 0

Books: 0

Conference: 0

Proceedings: 0
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: $O(m+n)$
Longest Common Subsequence

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?
LCS in linear space

We calculate the optimal LCS path from $(0,0)$ to $(n,m)$ that crosses through $(i,m/2)$ where $i$ ranges from $[0,n]$.

Define $length(i)$ as the length of the LCS path from $(0,0)$ to $(n,m)$ that passes through cell $(i, m/2)$, for all choices of $i$.

- $prefix(i) = |LCS(x[1...m/2],y[1...i])|$.
- $suffix(i) = |LCS(x[m/2+1...m],y[i+1...n])| = |LCS(x^R_{[1..m/2]},y^R_{[1...n-i]})|$.
- $length(i) = prefix(i) + suffix(i)$ is the length of the LCS path that passes through cell $(i, m/2)$. 

LCS in linear space

- $prefix(i) = |LCS(x[1...m/2],y[1...i])|$.
- $suffix(i) = |LCS(x[m/2+1...m],y[i+1...n])| = |LCS(x^R_{[1..m/2]},y^R_{[1...n-i]})|$.
- $length(i) = prefix(i) + suffix(i)$ is the length of the LCS path that passes through cell $(i, m/2)$.
LCS in linear space

Define \((mid, m/2)\) as the vertex that contains the optimal LCS path (assume for simplicity there is only one), that is \(mid = \arg\max_{0 \leq i \leq n} length(i)\)

Computing Prefix\((i)\)

Compute \(prefix(i)\) from \(0 \rightarrow m/2\) where \(prefix(i)\) is the length of the LCS path from \((0,0)\) to \((i,m/2)\)
Computing Suffix(i)
Compute \(\text{suffix}(i)\) from \(m/2 \rightarrow m\) where \(\text{suffix}(i)\) is the length of the LCS path from \((n,m)\) to \((i,m/2)\)

\[
\begin{array}{ccc}
0 & m/2 & m \\
\end{array}
\]

Finding the middle point
- Find the value \(\text{mid}\) that maximizes \(\{\text{prefix}(i) + \text{suffix}(i)\}\) that is \(\text{mid}=\arg\max_{0 \leq i \leq n} \{\text{prefix}(i) + \text{suffix}(i)\}\)
- You now have a middle vertex of the maximum path \((\text{mid},m/2)\)
Time = Area: First Pass

- On first pass, the algorithm covers the entire area

Area = mn
Time = Area: Second Pass

• On second pass, the algorithm covers only 1/2 of the area

\[ \text{Area} = \frac{mn}{2} \]

Time = Area: Third Pass

• On third pass, only 1/4th is covered

\[ \text{Area} = \frac{mn}{4} \]
Time/space complexity

- \( nm(1 + \frac{1}{2} + \frac{1}{4} + \ldots) \leq 2nm \)

- Time complexity \( O(nm) \)

- Space complexity \( O(n+m) \)

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 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\( (G(V,E),s)\)

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

Iteration 0

[Diagram of a graph with labeled vertices and edges, showing the first iteration of the Bellman-Ford algorithm.]
Iteration 1

Iteration 2
Iteration 3

Iteration 4
Bellman-Ford is a dynamic programming algorithm. 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”

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

Let $d(s,v)$ be the length of the (correct) shortest path from $s$ to $v$.

**Lemma:** Assuming there are no negative-weight cycles reachable from $s$, $d[v] = d(s,v)$ holds upon termination of Bellman-Ford for all vertices $v$ reachable from $s$.

**Proof:**
Consider an (acyclic) shortest path $p$, where $p = \langle v_0, v_1, ..., v_k \rangle$, $v_0 = s$ and $v_k = v$. The path $p$ has $k \leq |V| - 1$ edges, otherwise $p$ has a cycle. We prove by induction that for $i=0,1,...,k$ we have $d[v_i] = d(s,v_i)$ after the $i$-th pass over the edges of $G$ and that equality is maintained thereafter (path-relaxation property).

**Basis:** $d[v_0] = d(s,v_0) = 0$.

**Inductive step:** assume $d[v_{i-1}] = d(s,v_{i-1})$ after $(i-1)$-st pass. Edge $(v_{i-1}, v_i)$ is relaxed at iteration $i$, and therefore $d[v_i] = d(s,v_i)$ and the equality is maintained thereafter.
Correctness

**Theorem:** Algorithm BF returns the correct TRUE/FALSE value (depending whether a negative cycle exists or not in the graph).

**Case 1:** There is no reachable negative-weight cycle from $s$.

Upon termination of BF, we have for all $(u, v)$:
\[
d[v] = d(s, v) \quad \text{by previous Lemma if } v \text{ is reachable}
\]
\[
d[v] = d(s, v) = \infty \quad \text{otherwise}
\]
\[
\leq d(s, u) + w(u, v)
\]
\[
= d[u] + w(u, v)
\]

So, algorithm returns TRUE.

**Case 2:** There exists a $s$-reachable negative-weight cycle $c = \langle v_0, v_1, ..., v_k \rangle$, where $v_0 = v_k$. Proof by contradiction.

We have $\sum_{i=1,...,k} w(v_{i-1}, v_i) < 0$. \((*)\)

Suppose algorithm returns TRUE. Then, $d[v_i] \leq d[v_{i-1}] + w(v_{i-1}, v_i)$ for $i = 1, ..., k$. Summing the inequality around the cycle, we get
\[
\sum_{i=1,...,k} d[v_i] \leq \sum_{i=1,...,k} d[v_{i-1}] + \sum_{i=1,...,k} w(v_{i-1}, v_i)
\]

But, $\sum_{i=1,...,k} d[v_i] = \sum_{i=1,...,k} d[v_{i-1}]$ because $v_0 = v_k$ and each vertex in $c$ appears exactly once.

We can show no $d[v_i]$ is infinite. Hence, $0 \leq \sum_{i=1,...,k} w(v_{i-1}, v_i)$.

Contradicts \((*)\). Thus, algorithm returns FALSE.
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 $D_{i,j}^k =$ "the shortest distance from $v_i$ to $v_j$ using only vertices $\{v_1,v_2,...,v_k\}$"
A dynamic programming shortest-path

Initially we set

\[ D_{i,j}^0 = \begin{cases} 
0 & \text{if } i = j \\
\infty & \text{otherwise} \\
w((v_i, v_j)) & \text{if } (v_i, v_j) \in E
\end{cases} \]

A dynamic programming shortest-path
A dynamic programming shortest-path

- The cost of going from $v_i$ to $v_j$ using vertices $1,\ldots,k$ is the shorter between
  - (do not 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($\bar{G}$):

**Input:** A weighted directed graph $\bar{G}$ with $n$ vertices numbered $v_1,v_2,\ldots,v_n$

**Output:** A matrix $D$ such that $D[i,j]$ is distance from $v_i$ to $v_j$ in $\bar{G}$

for $i$ ← 1 to $n$ do
  for $j$ ← 1 to $n$ do
    if $i = j$ then
      Set $D[i,i]$ ← 0 and continue looping
    if $(v_i,v_j)$ is an edge in $\bar{G}$ then
      Set $D[i,j] ← w((v_i,v_j))$
    else
      Set $D[i,j] ← +\infty$
  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 $D^n$
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”