Chapter 6
Dynamic Programming

6.4 Knapsack Problem
Knapsack Problem

Knapsack problem.
- Given \( n \) objects and a "knapsack."
- Item \( i \) weighs \( w_i > 0 \) kilograms and has value \( v_i > 0 \).
- Knapsack has capacity of \( W \) kilograms.
- Goal: fill knapsack so as to maximize total value.

Ex: \( \{ 3, 4 \} \) has value 40.

<table>
<thead>
<tr>
<th>#</th>
<th>value</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

Knapsack Problem

Knapsack problem.
- Knapsack has capacity of \( W \) kilograms.
- Goal: fill knapsack so as to maximize total value.

Greedy Solution?
- Sort by value (big to small)
- Sort by weight (small to big)
- Sort by the ratio of value/weight (big to small)

Counterexamples:
- \( \{ 5, 2, 1 \} \) has value 39
- \( \{ 1, 2, 3 \} \) has value 29
- \( \{ 1, 5, 2 \} \) has value 39

<table>
<thead>
<tr>
<th>#</th>
<th>value</th>
<th>weight</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
<td>3.6</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
<td>3.66</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
<td>4</td>
</tr>
</tbody>
</table>

Opt = \( \{ 3, 4 \} \)
Value = 40
Dynamic Programming: A Start

**Def.** $OPT(i) = \text{max profit subset of items } 1, ..., i.$

- **Case 1:** $OPT$ does not select item $i$.
  - $OPT$ selects best of $\{1, 2, ..., i-1\}$

- **Case 2:** $OPT$ selects item $i$.
  - Do we have room for $w_i$ now?
  - How does this affect later choices?

**Conclusion.** Need more sub-problems!

Dynamic Programming: Adding a New Variable

**Def.** $OPT(i, w) = \text{max profit subset of items } 1, ..., i \text{ with weight limit } w.$

- **Case 1:** $OPT$ does not select item $i$.
  - $OPT$ selects best of $\{1, 2, ..., i-1\}$ using weight limit $w$

- **Case 2:** $OPT$ selects item $i$.
  - $OPT$ selects best of $\{1, 2, ..., i-1\}$ using weight limit $w - w_i$

$$
OPT(i, w) = \begin{cases} 
0 & \text{if } i = 0 \\
OPT(i-1, w) & \text{if } w_i > w \\
\max\{OPT(i-1, w), v_i + OPT(i-1, w - w_i)\} & \text{otherwise}
\end{cases}
$$
Knapsack Problem: Bottom-Up Algorithm

Input: \( n, W, w_1, \ldots, w_n, v_1, \ldots, v_n \)

for \( w = 0 \) to \( W \)
\[ M[0, w] = 0 \]

for \( i = 1 \) to \( n \)
  for \( w = 1 \) to \( W \)
    if \( (w_i > w) \)
      \[ M[i, w] = M[i-1, w] \]
    else
      \[ M[i, w] = \max\{M[i-1, w], v_i + M[i-1, w-w_i]\} \]

return \( M[n, W] \)

Knapsack Table

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

OPT: \{ 4, 3 \}
value = 22 + 18 = 40

W = 11
Knapsack Problem: Runtime

Fill up an n-by-W array.

```
Input: n, W, w_1,…,w_N, v_1,…,v_N
for w = 0 to W
    M[0, w] = 0
for i = 1 to n
    for w = 1 to W
        if (w_i > w)
            M[i, w] = M[i-1, w]
        else
            M[i, w] = max {M[i-1, w], v_i + M[i-1, w-w_i]}
return M[n, W]
```

Runtime? O(nW)

Knapsack Problem: Running Time and Beyond

Run time. Θ(nW).
- Not polynomial in input size!
- "Pseudo-polynomial."

**Decision Version of Knapsack.** "Is there a solution of value at least V"? This problem is NP-complete. [Chapter 8]

**Greedy approximation algorithm.** Works with "unlimited" supply, and sorts based on ratio (v_i/w_i). Gives solution no worse than OPT/2.

**DP approximation scheme.** There exists a fully poly-time approximation scheme (FPTAS) that produces a solution within ε% of optimum in time O(poly(n,ε)). [Chapter 11]
Longest Decreasing Subsequence

Subsequence problem.
- Given a sequence $S_1 S_2 \ldots S_n$.
- Find a decreasing subsequence of maximum length.

Ex:  
$\begin{array}{cccccc}
2 & 3 & 2 & 4 & 1 & 5 & 4 \\
2 & 3 & 2 & 4 & 1 & 5 & 4 \\
\end{array}$

OPT = 3
Dynamic Programming: A Start

Def.  \( \text{OPT}(i) = \text{max length decreasing subsequence in substring } S_1, \ldots, S_i. \)

- Case 1:  \( \text{OPT} \) does not select item \( i \).
  - \( \text{OPT} \) selects best of \( \{1, 2, \ldots, i-1\} \)
- Case 2:  \( \text{OPT} \) selects item \( i \).
  - Where can we recurse to?
  - Try: \( \text{OPT} \) selects best among \( j < i \) where \( S_j > S_i \).
  - Problem: \( \text{OPT}(j) \) may not use \( S_j \).

Ex: Recall \( \text{OPT} = 3 \) below. This definition claims \( \text{OPT} = 4 \).
\[
\begin{array}{cccccccc}
S: & 2 & 3 & 2 & 4 & 1 & 5 & 4 \\
Opt: & 1 & 1 & 2 & 2 & 3 & 3 & 4 \\
\end{array}
\]

Conclusion. Need different sub-problems!

---

Dynamic Programming: Optimize Different Value

Def.  \( \text{New-OPT}(i) = \text{max length decreasing subsequence ending with } S_i. \)

- Case 1:  \( \text{New-OPT} \) does not select item \( i \).
  - Not an option!
- Case 2:  \( \text{New-OPT} \) selects item \( i \).
  - \( \text{New-OPT} \) selects best among \( j < i \) where \( S_j > S_i \).
  - Can now use this safely (\( \text{New-OPT}(j) \) does use \( S_j \))

\[
\text{NewOPT}(i) = \begin{cases} 
0 & \text{if } i = 0 \\
1 + \max_{0 < j < i \text{ such that } S_j > S_i} \{ \text{NewOPT}(j) \} & \text{otherwise}
\end{cases}
\]

Post-process to find \( \text{OPT} \)
- The opt subsequence ends somewhere, so \( \text{OPT} = \text{max} \{ \text{NewOPT}(i) \} \)
Longest Decreasing Subsequence: Runtime

Fill array of length n, then search for true OPT.

\[
\text{Input: } n, s_1, \ldots, s_n \\
M[0] = 0 \\
\text{for } i = 1 \text{ to } n \\
\quad M[i] = \max\{ 1, \max_{j:S_j > S_i} (1 + M[j]) \} \\
\text{OPT} = 0 \\
\text{for } i = 1 \text{ to } n \\
\quad \text{if } \text{Opt} < M[i] \\
\quad \text{OPT} = M[i] \\
\text{return OPT}
\]

Runtime? $O(n^2)$

Dynamic Programming Summary

Recipe.
- Characterize structure of problem.
- Recursively define value of optimal solution.
- Compute value of optimal solution.
- Construct optimal solution from computed information.

Dynamic programming techniques.
- Binary choice: weighted interval scheduling.
- Multi-way choice: segmented least squares.
- Dynamic programming over intervals: RNA secondary structure.
- Adding a new variable: knapsack.
- Optimizing a different value: max length increasing subsequence.

Top-down vs. bottom-up: different people have different intuitions.
Dynamic Programming In-Class Exercise

Top Tastiness. Curio has a dessert day i with tastiness \( t_i \) (\( t_i > 0 \)). Your new diet only lets you have dessert if you did not have dessert the day before. Maximize the total tastiness over \( n \) days.

Max Movie Magic. There are \( n \) movies coming out (only one is out at a time), but you only want to see a movie if it is better than the last one you saw. Maximize the number of movies you can watch.

Supreme Strings. Given a string \( S_1 \ldots S_n \), and a real valued function \( \text{quality}(x) \) on strings, determine how to break up the string in order to maximize the total quality of the pieces.

Choice Change. Given coins with values \( v_1 < \ldots < v_n \), make change for an amount of money \( C \) with as few coins as possible.

Optimal Olympic Outing. There are \( n \) events over \( n \) days, which you want to see with excitement \( P_1 \ldots P_n \) (\( P_i \) may be \( < 0 \), e.g. nordic skiing). Maximize the sum excitement over a set of contiguous days.
4.4 Shortest Paths in a Graph

Edsger W. Dijkstra

The question of whether computers can think is like the question of whether submarines can swim.

Do only what only you can do.

In their capacity as a tool, computers will be but a ripple on the surface of our culture. In their capacity as intellectual challenge, they are without precedent in the cultural history of mankind.
Shortest Path Problem

Shortest path network.
- Directed graph \( G = (V, E) \).
- Source \( s \), destination \( t \).
- Length \( \ell_e \) = length of edge \( e \).

Shortest path problem: find shortest directed path from \( s \) to \( t \).

\[
\text{cost of path} = \sum \text{edge costs in path}
\]

\[
\text{Cost}(s, 2, 3, 5, t) = 9 + 23 + 2 + 16 = 50.
\]

Dijkstra’s Algorithm

Dijkstra’s algorithm.
- Maintain a set of explored nodes \( S \) for which we have determined the shortest path distance \( d(u) \) from \( s \) to \( u \).
- Initialize \( S = \{s\} \), \( d(s) = 0 \).
- Repeatedly choose unexplored node \( v \) which minimizes
\[
\pi(v) = \min_{e = (u,v) : u \in S} d(u) + \ell_e,
\]
add \( v \) to \( S \), and set \( d(v) = \pi(v) \).

shortest path to some \( u \) in explored part, followed by a single edge \((u,v)\)
Dijkstra's Algorithm

Dijkstra's algorithm.
- Maintain a set of explored nodes $S$ for which we have determined the shortest path distance $d(u)$ from $s$ to $u$.
- Initialize $S = \{s\}$, $d(s) = 0$.
- Repeatedly choose unexplored node $v$ which minimizes
  \[ \pi(v) = \min_{e = (u, v), \; u \in S} d(u) + \ell_e, \]
  add $v$ to $S$, and set $d(v) = \pi(v)$.

Proof of Correctness

**Observation.** For all $u \in S$, $d(u)$ is the length of the shortest $s$-$u$ path.

**Pf.** (by induction on $|S|$)

**Base case:** $|S| = 1$ is trivial.

**Inductive hypothesis:** Assume true for $|S| = k \geq 1$.

- Let $v$ be next node added to $S$, and let $u$-$v$ be the chosen edge.
- The shortest $s$-$u$ path plus $(u, v)$ is an $s$-$v$ path of length $\pi(v)$.
- Consider any $s$-$v$ path $P$. We'll see that it's no shorter than $\pi(v)$.
- Let $x$-$y$ be the first edge in $P$ that leaves $S$, and let $P'$ be the subpath to $x$.
- $P$ is already too long as soon as it leaves $S$.

\[ \ell(P) \geq \ell(P') + \ell(x, y) \geq d(x) + \ell(x, y) \geq \pi(y) \geq \pi(v) \]

- nonnegative weights
- inductive hypothesis
- defn of $\pi(y)$
- Dijkstra chose $v$ instead of $y$
6.8 Shortest Paths

Shortest path problem. Given a directed graph $G = (V, E)$, with edge weights $c_{vw}$, find shortest path from node $s$ to node $t$.

Ex. Nodes represent agents in a financial setting and $c_{vw}$ is cost of transaction in which we buy from agent $v$ and sell immediately to $w$. 
Shortest Paths: Failed Attempts

Dijkstra. Can fail if negative edge costs.

\[ s - 2 - u - 3 - v \]
\[ s \quad 1 \quad t \quad -6 \]

Re-weighting. Adding a constant to every edge weight can fail.

\[ s - 5 - W - 5 - t \]
\[ s \quad 6 \quad W \quad 0 \quad t \quad 3 \]

Shortest Paths: Negative Cost Cycles

Negative cost cycle.

\[ -6 \quad -4 \]
\[ -7 \]

Observation. If some path from \( s \) to \( t \) contains a negative cost cycle, there does not exist a shortest \( s \)-\( t \) path; otherwise, there exists one that is simple.

\[ s - W - 1 \]
\[ c(W) < 0 \]
Shortest Paths: Dynamic Programming

Def. \( OPT(v) \) = length of shortest \( v \rightarrow t \) path \( P \).
Def. \( OPT(i, v) \) = length of shortest \( v \rightarrow t \) path \( P \) using at most \( i \) edges.

- Case 1: \( P \) uses at most \( i-1 \) edges.
  - \( OPT(i, v) = OPT(i-1, v) \)

- Case 2: \( P \) uses exactly \( i \) edges.
  - \( OPT \) uses \((v, w)\) edge, and then uses \( OPT(i-1, w) \)

\[
OPT(i, v) = \begin{cases} 
0 & \text{if } i = 0 \\
\min \left\{ OPT(i-1, v), \min_{(v, w) \in E} \{ OPT(i-1, w) + c_{vw} \} \right\} & \text{otherwise}
\end{cases}
\]

Remark. By previous observation, if no negative cycles, then \( OPT(n-1, v) = \) length of shortest \( v \rightarrow t \) path.

Shortest Paths: Implementation

```java
Shortest-Path(G, t) {
    foreach node \( v \in V \)
        \( M[0, v] \leftarrow \infty \)
        \( M[0, t] \leftarrow 0 \)
    for \( i = 1 \) to \( n-1 \)
        foreach node \( v \in V \)
            \( M[i, v] \leftarrow M[i-1, v] \)
        foreach edge \((v, w)\) \in E
            \( M[i, v] \leftarrow \min \{ M[i, v], M[i-1, w] + c_{vw} \} \)
}
```

Analysis. \( \Theta(mn) \) time, \( \Theta(n^2) \) space.
Shortest Paths: Practical Improvements

Traceback. Maintain a "successor" for each table entry.

Practical improvements.
- Maintain only one array $M[v] = \text{shortest } v \rightarrow t \text{ path found so far.}$
- No need to check edge $(v, w)$ unless $M[w]$ changed.

Theorem. $M[v]$ is length of some $v \rightarrow t$ path throughout. After $i$ rounds, $M[v]$ is at most the length of shortest $v \rightarrow t$ path using $\leq i$ edges.

Bellman-Ford: Efficient Implementation

```
Push-Based-Shortest-Path(G, s, t) {
    foreach node $v \in V$ {
        $M[v] \leftarrow \infty$
        successor[v] \leftarrow \phi
    }
    $M[t] = 0$
    for $i = 1$ to $n-1$ {
        foreach node $w \in V$ {
            if ($M[w]$ has been updated in previous iteration) {
                foreach node $v$ such that $(v, w) \in E$ {
                    if ($M[v] > M[w] + c_{vw}$) {
                        $M[v] \leftarrow M[w] + c_{vw}$
                        successor[v] \leftarrow w
                    }
                }
                If no $M[w]$ value changed in iteration $i$, stop.
            }
        }
    }
}
```
Shortest Paths: Practical Improvements

Traceback. Maintain a "successor" for each table entry.

Practical improvements.
- Maintain only one array $M[v] = $ shortest $v \rightarrow t$ path found so far.
- No need to check edge $(v, w)$ unless $M[w]$ changed.

Theorem. $M[v]$ is length of some $v \rightarrow t$ path throughout. After $i$ rounds, $M[v]$ is at most the length of shortest $v \rightarrow t$ path using $\leq i$ edges.

Overall impact.
- Space: $O(m + n)$.
- Running time: $O(mn)$ worst case, but substantially faster in practice.

6.10 Negative Cycles in a Graph
Detecting Negative Cycles

**Lemma.** If $\text{OPT}(n,v) = \text{OPT}(n-1,v)$ for all $v$, then no negative cycles.

**Pf.** Bellman-Ford algorithm.

**Lemma.** If $\text{OPT}(n,v) < \text{OPT}(n-1,v)$ for some node $v$, then (any) shortest path from $v$ to $t$ contains a cycle $W$. Moreover $W$ has negative cost.

**Pf.** (by contradiction)
- Since $\text{OPT}(n,v) < \text{OPT}(n-1,v)$, we know $P$ has exactly $n$ edges.
- By pigeonhole principle, $P$ must contain a directed cycle $W$.
- Deleting $W$ yields a $v$-$t$ path with $< n$ edges $\Rightarrow$ $W$ has negative cost.

![Diagram](image1)

Detecting Negative Cycles

**Theorem.** Can detect negative cost cycle in $O(mn)$ time.
- Add new node $t$ and connect all nodes to $t$ with 0-cost edge.
- Check if $\text{OPT}(n,v) = \text{OPT}(n-1,v)$ for all nodes $v$.
  - if yes, then no negative cycles
  - if no, then extract cycle from shortest path from $v$ to $t$
Detecting Negative Cycles: Application

Currency conversion. Given n currencies and exchange rates between pairs of currencies, is there an arbitrage opportunity?

Remark. Fastest algorithm very valuable!