Chapter 6
Dynamic Programming

CS 573: Algorithms, Fall 2013
September 12, 2013

6.1 Maximum Weighted Independent Set in Trees

6.1.0.1 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\}

6.1.0.2 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: ??
6.1.0.3 Towards a Recursive Solution

For an arbitrary graph $G$:

(A) Number vertices as $v_1, v_2, \ldots, v_n$

(B) 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 $v_n$).

(C) 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 $T$?

6.1.0.4 Towards a Recursive Solution

Natural candidate for $v_n$ is root $r$ of $T$? Let $O$ be an optimum solution to the whole problem.

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

Case $r \in O$: None of the children of $r$ can be in $O$. $O - \{r\}$ contains an optimum solution for each subtree of $T$ hanging at a grandchild of $r$.

Subproblems? Subtrees of $T$ hanging at nodes in $T$.

6.1.0.5 A Recursive Solution

$T(u)$: subtree of $T$ hanging at node $u$

$OPT(u)$: max weighted independent set value in $T(u)$

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

6.1.0.6 Iterative Algorithm

(A) Compute $OPT(u)$ bottom up. To evaluate $OPT(u)$ need to have computed values of all children and grandchildren of $u$

(B) What is an ordering of nodes of a tree $T$ to achieve above? Post-order traversal of a tree.

6.1.0.7 Iterative Algorithm

\[
\text{MIS-Tree}(T):
\begin{align*}
\text{Let } v_1, v_2, \ldots, v_n \text{ be a post-order traversal of nodes of } T \\
\text{for } i = 1 \text{ to } n \text{ do} \\
M[v_i] = \max \left( \sum_{v_j \text{ child of } v_i} M[v_j], \right. \\
\left. w(v_i) + \sum_{v_j \text{ grandchild of } v_i} M[v_j] \right) \\
\text{return } M[v_n] \text{ (* Note: } v_n \text{ is the root of } T \text{ *)}
\end{align*}
\]

Space: $O(n)$ to store the value at each node of $T$

Running time:

(A) Naive bound: $O(n^2)$ since each $M[v_i]$ evaluation may take $O(n)$ time and there are $n$ evaluations.

(B) Better bound: $O(n)$. A value $M[v_j]$ is accessed only by its parent and grand parent.
6.1.0.8 Example

6.1.0.9 Dominating set

Definition 6.1.1. \( G = (V, E) \). The set \( X \subseteq V \) is a dominating set, if any vertex \( v \in V \) is either in \( X \) or is adjacent to a vertex in \( X \).

Problem 6.1.2. Given weights on vertices, compute the minimum weight dominating set in \( G \).

Dominating Set is NP-Hard!

6.2 DAGs and Dynamic Programming

6.2.0.10 Recursion and DAGs

Observation 6.2.1. Let \( A \) be a recursive algorithm for problem \( \Pi \). For each instance \( I \) of \( \Pi \) there is an associated DAG \( G(I) \).

(A) Create directed graph \( G(I) \) as follows...
(B) For each sub-problem in the execution of \( A \) on \( I \) create a node.
(C) If sub-problem \( v \) depends on or recursively calls sub-problem \( u \) add directed edge \((u, v)\) to graph.
(D) \( G(I) \) is a DAG. Why? If \( G(I) \) has a cycle then \( A \) will not terminate on \( I \).

6.2.1 Iterative Algorithm for...

6.2.1.1 Dynamic Programming and DAGs

Observation 6.2.2. An iterative algorithm \( B \) obtained from a recursive algorithm \( A \) for a problem \( \Pi \) does the following:

For each instance \( I \) of \( \Pi \), it computes a topological sort of \( G(I) \) and evaluates sub-problems according to the topological ordering.
(A) Sometimes the DAG $G(I)$ can be obtained directly without thinking about the recursive algorithm $A$.

(B) In some cases (not all) the computation of an optimal solution reduces to a shortest/longest path in DAG $G(I)$.

(C) Topological sort based shortest/longest path computation is dynamic programming!

6.2.2 A quick reminder...

6.2.2.1 A Recursive Algorithm for weighted interval scheduling

Let $O_i$ be value of an optimal schedule for the first $i$ jobs.

```
Schedule(n):
    if n = 0 then return 0
    if n = 1 then return $w(v_1)$
    $O_{p(n)}$ ← Schedule($p(n)$)
    $O_{n-1}$ ← Schedule($n$ - 1)
    if ($O_{p(n)} + w(v_n) < O_{n-1}$) then
        $O_n = O_{n-1}$
    else
        $O_n = O_{p(n)} + w(v_n)$
    return $O_n$
```

6.2.3 Weighted Interval Scheduling via...

6.2.3.1 Longest Path in a DAG

Given intervals, create a DAG as follows:

(A) Create one node for each interval, plus a dummy sink node 0 for interval 0, plus a dummy source node $s$.

(B) For each interval $i$ add edge $(i, p(i))$ of the length/weight of $v_i$.

(C) Add an edge from $s$ to $n$ of length 0.

(D) For each interval $i$ add edge $(i, i - 1)$ of length 0.

6.2.3.2 Example

$p(5) = 2$, $p(4) = 1$, $p(3) = 1$, $p(2) = 0$, $p(1) = 0$
6.2.3.3 Relating Optimum Solution

Given interval problem instance \( I \) let \( G(I) \) denote the DAG constructed as described.

Claim 6.2.3. Optimum solution to weighted interval scheduling instance \( I \) is given by longest path from \( s \) to \( 0 \) in \( G(I) \).

Assuming claim is true,
(A) If \( I \) has \( n \) intervals, DAG \( G(I) \) has \( n + 2 \) nodes and \( O(n) \) edges. Creating \( G(I) \) takes \( O(n \log n) \) time: to find \( p(i) \) for each \( i \). How?
(B) Longest path can be computed in \( O(n) \) time — recall \( O(m + n) \) algorithm for shortest/longest paths in DAGs.

6.2.3.4 DAG for Longest Increasing Sequence

Given sequence \( a_1, a_2, \ldots, a_n \) create DAG as follows:
(A) add sentinel \( a_0 \) to sequence where \( a_0 \) is less than smallest element in sequence
(B) for each \( i \) there is a node \( v_i \)
(C) if \( i < j \) and \( a_i < a_j \) add an edge \( (v_i, v_j) \)
(D) find longest path from \( v_0 \)

6.3 Edit Distance and Sequence Alignment

6.3.0.5 Spell Checking Problem

Given a string “exponen” that is not in the dictionary, how should a spell checker suggest a nearby string?

What does nearness mean?

Question: Given two strings \( x_1 x_2 \ldots x_n \) and \( y_1 y_2 \ldots y_m \) what is a distance between them?
**Edit Distance**: minimum number of “edits” to transform $x$ into $y$.

### 6.3.0.6 Edit Distance

**Definition 6.3.1.** *Edit distance* between two words $X$ and $Y$ is the number of letter insertions, letter deletions and letter substitutions required to obtain $Y$ from $X$.

**Example 6.3.2.** The edit distance between *FOOD* and *MONEY* is at most 4:

$$
\text{FOOD} \rightarrow \text{MOOD} \rightarrow \text{MONOD} \rightarrow \text{MONED} \rightarrow \text{MONEY}
$$

### 6.3.0.7 Edit Distance: Alternate View

Alignment Place words one on top of the other, with gaps in the first word indicating insertions, and gaps in the second word indicating deletions.

```
  F O O D
  M O N E Y
```

Formally, an **alignment** is a set $M$ of pairs $(i, j)$ such that each index appears at most once, and there is no “crossing”: $i < i'$ and $i$ is matched to $j$ implies $i'$ is matched to $j' > j$. In the above example, this is $M = \{(1, 1), (2, 2), (3, 3), (4, 5)\}$. Cost of an alignment is the number of mismatched columns plus number of unmatched indices in both strings.

### 6.3.0.8 Edit Distance Problem

**Problem** Given two words, find the edit distance between them, i.e., an alignment of smallest cost.

### 6.3.0.9 Applications

(A) Spell-checkers and Dictionaries  
(B) Unix *diff*  
(C) DNA sequence alignment . . . but, we need a new metric

### 6.3.0.10 Similarity Metric

**Definition 6.3.3.** For two strings $X$ and $Y$, the cost of alignment $M$ is

(A) **[Gap penalty]** For each gap in the alignment, we incur a cost $\delta$.

(B) **[Mismatch cost]** For each pair $p$ and $q$ that have been matched in $M$, we incur cost $\alpha_{pq}$; typically $\alpha_{pp} = 0$.

*Edit distance is special case when $\delta = \alpha_{pq} = 1$.*

### 6.3.0.11 An Example

**Example 6.3.4.**

```
  o | c | u | r | r | a | n | c | e  | Cost = $\delta + \alpha_{ae}$
  o | c | c | u | r | r | e | n | c | e
```

Alternative:

```
  o | c | c | u | r | r | a | n | c | e  | Cost = 3$\delta$
  o | c | c | u | r | r | e | n | c | e
```

Or a really stupid solution (delete string, insert other string):

```
  o | c | u | r | r | a | n | c | e  | o | c | u | r | r | e | n | c | e  | Cost = 19$\delta$.
  o | c | c | u | r | r | e | n | c | e
```
6.3.0.12 Sequence Alignment

Input Given two words $X$ and $Y$, and gap penalty $\delta$ and mismatch costs $\alpha_{pq}$

Goal Find alignment of minimum cost

6.3.1 Edit distance

6.3.1.1 Basic observation

Let $X = \alpha x$ and $Y = \beta y$

$\alpha, \beta$: strings.

$x$ and $y$ single characters.

Think about optimal edit distance between $X$ and $Y$ as alignment, and consider last column of alignment of the two strings:

$$
\begin{array}{c|c}
\alpha & x \\
\hline
\beta & y \\
\end{array}
$$

or

$$
\begin{array}{c|c}
\alpha x & \\
\hline
\beta y & \\
\end{array}
$$

or

$$
\begin{array}{c|c}
\alpha x & \\
\hline
\beta & y \\
\end{array}
$$

Observation 6.3.5. Prefixes must have optimal alignment!

6.3.1.2 Problem Structure

Observation 6.3.6. Let $X = x_1 x_2 \cdots x_m$ and $Y = y_1 y_2 \cdots y_n$. If $(m, n)$ are not matched then either the $m$th position of $X$ remains unmatched or the $n$th position of $Y$ remains unmatched.

(A) Case $x_m$ and $y_n$ are matched.

(A) Pay mismatch cost $\alpha_{x_m y_n}$ plus cost of aligning strings $x_1 \cdots x_{m-1}$ and $y_1 \cdots y_{n-1}$

(B) Case $x_m$ is unmatched.

(A) Pay gap penalty plus cost of aligning $x_1 \cdots x_{m-1}$ and $y_1 \cdots y_n$

(C) Case $y_n$ is unmatched.

(A) Pay gap penalty plus cost of aligning $x_1 \cdots x_m$ and $y_1 \cdots y_{n-1}$

6.3.1.3 Subproblems and Recurrence

Optimal Costs Let $Opt(i, j)$ be optimal cost of aligning $x_1 \cdots x_i$ and $y_1 \cdots y_j$. Then

$$
Opt(i, j) = \min \left\{ \alpha_{x_i y_j} + Opt(i - 1, j - 1), \delta + Opt(i - 1, j), \delta + Opt(i, j - 1) \right\}
$$

Base Cases: $Opt(i, 0) = \delta \cdot i$ and $Opt(0, j) = \delta \cdot j$

6.3.1.4 Dynamic Programming Solution

```plaintext
for all $i$ do $M[i, 0] = i\delta$
for all $j$ do $M[0, j] = j\delta$
for $i = 1$ to $m$ do
  for $j = 1$ to $n$ do
    $M[i, j] = \min \left\{ \alpha_{x_i y_j} + M[i - 1, j - 1], \delta + M[i - 1, j], \delta + M[i, j - 1] \right\}$
```
Figure 6.1: Iterative algorithm in previous slide computes values in row order. Optimal value is a shortest path from $(0, 0)$ to $(m, n)$ in DAG.

Analysis
(A) Running time is $O(mn)$.
(B) Space used is $O(mn)$.

6.3.1.5 Matrix and DAG of Computation
6.3.1.6 Sequence Alignment in Practice

(A) Typically the DNA sequences that are aligned are about $10^5$ letters long!
(B) So about $10^{10}$ operations and $10^{10}$ bytes needed
(C) The killer is the 10GB storage
(D) Can we reduce space requirements?

6.3.1.7 Optimizing Space

(A) Recall

$$M(i, j) = \min \begin{cases} 
\alpha_{x,y} + M(i - 1, j - 1), \\
\delta + M(i - 1, j), \\
\delta + M(i, j - 1)
\end{cases}$$

(B) Entries in $j$th column only depend on $(j - 1)$st column and earlier entries in $j$th column
(C) Only store the current column and the previous column reusing space; $N(i, 0)$ stores $M(i, j - 1)$ and $N(i, 1)$ stores $M(i, j)$

6.3.1.8 Computing in column order to save space
6.3.1.9 Space Efficient Algorithm

```plaintext
for all $i$ do $N[i, 0] = i\delta$
for $j = 1$ to $n$ do
    $N[0, 1] = j\delta$ (* corresponds to $M(0, j)$ *)
    for $i = 1$ to $m$ do
        $N[i, 1] = \min \begin{cases} 
\alpha_{x,y} + N[i - 1, 0], \\
\delta + N[i - 1, 1], \\
\delta + N[i, 0]
\end{cases}$
    for $i = 1$ to $m$ do
        Copy $N[i, 0] = N[i, 1]$
```
Figure 6.2: $M(i,j)$ only depends on previous column values. Keep only two columns and compute in column order.

Analysis Running time is $O(mn)$ and space used is $O(2m) = O(m)$

6.3.1.10 Analyzing Space Efficiency

(A) From the $m \times n$ matrix $M$ we can construct the actual alignment (exercise)

(B) Matrix $N$ computes cost of optimal alignment but no way to construct the actual alignment

(C) Space efficient computation of alignment? More complicated algorithm — see textbook.

6.3.1.11 Takeaway Points

(A) Dynamic programming is based on finding a recursive way to solve the problem. Need a recursion that generates a small number of subproblems.

(B) 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.

(C) 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.

6.4 All Pairs Shortest Paths

6.4.0.12 Shortest Path Problems

Shortest Path Problems

Input A (undirected or directed) graph $G = (V, E)$ with edge lengths (or costs). For edge $e = (u, v)$, $\ell(e) = \ell(u,v)$ is its length.

(A) Given nodes $s, t$ find shortest path from $s$ to $t$.

(B) Given node $s$ find shortest path from $s$ to all other nodes.

(C) Find shortest paths for all pairs of nodes.

6.4.0.13 Single-Source Shortest Paths

Single-Source Shortest Path Problems

Input A (undirected or directed) graph $G = (V, E)$ with edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.
(A) Given nodes $s, t$ find shortest path from $s$ to $t$.
(B) Given node $s$ find shortest path from $s$ to all other nodes.

**Dijkstra’s algorithm** for non-negative edge lengths. Running time: $O((m + n) \log n)$ with heaps and $O(m + n \log n)$ with advanced priority queues.

**Bellman-Ford algorithm** for arbitrary edge lengths. Running time: $O(nm)$.

### 6.4.0.14 All-Pairs Shortest Paths

All-Pairs Shortest Path Problem

**Input** A (undirected or directed) graph $G = (V, E)$ with edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

(A) Find shortest paths for all pairs of nodes.
   - Apply single-source algorithms $n$ times, once for each vertex.
   (A) Non-negative lengths. $O(nm \log n)$ with heaps and $O(nm + n^2 \log n)$ using advanced priority queues.
   (B) Arbitrary edge lengths: $O(n^2 m)$.
   - $\Theta(n^4)$ if $m = \Omega(n^2)$.
   - Can we do better?

### 6.4.0.15 Shortest Paths and Recursion

(A) Compute the shortest path distance from $s$ to $t$ recursively?
(B) What are the smaller sub-problems?

**Lemma 6.4.1.** Let $G$ be a directed graph with arbitrary edge lengths. If $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ is a shortest path from $s$ to $v_k$ then for $1 \leq i < k$:
(A) $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_i$ is a shortest path from $s$ to $v_i$

Sub-problem idea: paths of fewer hops/edges

### 6.4.0.16 Hop-based Recur’: Single-Source Shortest Paths

Single-source problem: fix source $s$.

$OPT(v, k)$: shortest path dist. from $s$ to $v$ using at most $k$ edges.

Note: $\text{dist}(s, v) = OPT(v, n - 1)$. Recursion for $OPT(v, k)$:

$$OPT(v, k) = \min \left\{ \min_{u \in V} (OPT(u, k - 1) + c(u, v)), OPT(v, k - 1) \right\}$$

Base case: $OPT(v, 1) = c(s, v)$ if $(s, v) \in E$ otherwise $\infty$

Leads to Bellman-Ford algorithm — see text book.

$OPT(v, k)$ values are also of independent interest: shortest paths with at most $k$ hops
6.4.0.17 All-Pairs: Recursion on index of intermediate nodes

(A) Number vertices arbitrarily as \( v_1, v_2, \ldots, v_n \)

(B) \( \text{dist}(i, j, k) \): shortest path distance between \( v_i \) and \( v_j \) among all paths in which the largest index of an intermediate node is at most \( k \)

\[
\begin{align*}
\text{dist}(i, j, 0) &= 100 \\
\text{dist}(i, j, 1) &= 9 \\
\text{dist}(i, j, 2) &= 8 \\
\text{dist}(i, j, 3) &= 5
\end{align*}
\]

6.4.0.18 All-Pairs: Recursion on index of intermediate nodes

\[
dist(i, j, k-1) \quad \text{dist}(k, j, k-1) \\
\text{dist}(i, j, k-1)
\]

\[
dist(i, j, k) = \min \left\{ \text{dist}(i, j, k-1), \text{dist}(i, k, k-1) + \text{dist}(k, j, k-1) \right\}
\]

Base case: \( \text{dist}(i, j, 0) = c(i, j) \) if \((i, j) \in E\), otherwise \( \infty \)

Correctness: If \( i \rightarrow j \) shortest path goes through \( k \) then \( k \) occurs only once on the path — otherwise there is a negative length cycle.
6.4.1 Floyd-Warshall Algorithm

6.4.1.1 for All-Pairs Shortest Paths

Check if $G$ has a negative cycle // Bellman-Ford: $O(mn)$ time if there is a negative cycle then return ‘Negative cycle’;

for $i = 1$ to $n$
  for $j = 1$ to $n$
    $dist(i, j, 0) = c(i, j)$ (* $c(i, j) = \infty$ if $(i, j) \notin E$, 0 if $i = j$ *)

for $k = 1$ to $n$
  for $i = 1$ to $n$
    for $j = 1$ to $n$
      $dist(i, j, k) = \min\left\{dist(i, j, k - 1), \begin{array}{l}dist(i, k, k - 1) + dist(k, j, k - 1)\end{array}\right\}$

Correctness: Recursion works under the assumption that all shortest paths are defined (no negative length cycle).

Running Time: $\Theta(n^3)$, Space: $\Theta(n^3)$.

6.4.2 Floyd-Warshall Algorithm

6.4.2.1 for All-Pairs Shortest Paths

Do we need a separate algorithm to check if there is negative cycle?

for $i = 1$ to $n$
  for $j = 1$ to $n$
    $dist(i, j, 0) = c(i, j)$ (* $c(i, j) = \infty$ if $(i, j) \notin E$, 0 if $i = j$ *)

not edge, 0 if $i = j$ *)

for $k = 1$ to $n$
  for $i = 1$ to $n$
    for $j = 1$ to $n$
      $dist(i, j, k) = \min(dist(i, j, k - 1), dist(i, k, k - 1) + dist(k, j, k - 1))$

for $i = 1$ to $n$
  if $(dist(i, i, n) < 0)$ then
    Output that there is a negative length cycle in $G$

Correctness: exercise

6.4.2.2 Floyd-Warshall Algorithm: Finding the Paths

Question: Can we find the paths in addition to the distances?

(A) Create a $n \times n$ array Next that stores the next vertex on shortest path for each pair of vertices
(B) With array Next, for any pair of given vertices $i, j$ can compute a shortest path in $O(n)$ time.
6.4.3 Floyd-Warshall Algorithm

6.4.3.1 Finding the Paths

for $i = 1$ to $n$ do
  for $j = 1$ to $n$ do
    $dist(i, j, 0) = c(i, j)$ (* $c(i, j) = \infty$ if $(i, j)$ not edge, 0 if $i = j$ *)
    $Next(i, j) = -1$
  
for $k = 1$ to $n$ do
  for $i = 1$ to $n$ do
    for $j = 1$ to $n$ do
      if ($dist(i, j, k - 1) > dist(i, k, k - 1) + dist(k, j, k - 1)$) then
        $dist(i, j, k) = dist(i, k, k - 1) + dist(k, j, k - 1)$
        $Next(i, j) = k$

for $i = 1$ to $n$ do
  if ($dist(i, i, n) < 0$) then
    Output that there is a negative length cycle in $G$

Exercise: Given $Next$ array and any two vertices $i, j$ describe an $O(n)$ algorithm to find a $i-j$ shortest path.

6.4.3.2 Summary of results on shortest paths

<table>
<thead>
<tr>
<th>Single vertex</th>
<th>Dijkstra</th>
<th>$O(n \log n + m)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>No negative edges</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Edges cost might be negative</td>
<td>Bellman Ford</td>
<td>$O(nm)$</td>
</tr>
<tr>
<td>But no negative cycles</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>All Pairs Shortest Paths</th>
<th>$n \times$ Dijkstra</th>
<th>$O(n^2 \log n + nm)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>No negative edges</td>
<td>$n \times$ Bellman Ford</td>
<td>$O(n^2 m) = O(n^4)$</td>
</tr>
<tr>
<td>No negative cycles</td>
<td>Floyd-Warshall</td>
<td>$O(n^3)$</td>
</tr>
</tbody>
</table>

6.5 Knapsack

6.5.0.3 Knapsack Problem

Input Given a Knapsack of capacity $W$ lbs. and $n$ objects with $i$th object having weight $w_i$ and value $v_i$; assume $W, w_i, v_i$ are all positive integers

Goal Fill the Knapsack without exceeding weight limit while maximizing value.

Basic problem that arises in many applications as a sub-problem.

6.5.0.4 Knapsack Example

<table>
<thead>
<tr>
<th>Item</th>
<th>$I_1$</th>
<th>$I_2$</th>
<th>$I_3$</th>
<th>$I_4$</th>
<th>$I_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>1</td>
<td>6</td>
<td>18</td>
<td>22</td>
<td>28</td>
</tr>
<tr>
<td>Weight</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

Example 6.5.1. If $W = 11$, the best is $\{I_3, I_4\}$ giving value 40.

Special Case When $v_i = w_i$, the Knapsack problem is called the Subset Sum Problem.
6.5.0.5  Greedy Approach

(A) Pick objects with greatest value

Let \( W = 2, w_1 = w_2 = 1, w_3 = 2, v_1 = v_2 = 2 \) and \( v_3 = 3 \); greedy strategy will pick \( \{3\} \), but the optimal is \( \{1, 2\} \).

(B) Pick objects with smallest weight

Let \( W = 2, w_1 = 1, w_2 = 2, v_1 = 1 \) and \( v_2 = 3 \); greedy strategy will pick \( \{1\} \), but the optimal is \( \{2\} \).

(C) Pick objects with largest \( \frac{v_i}{w_i} \) ratio

Let \( W = 4, w_1 = w_2 = 2, w_3 = 3, v_1 = v_2 = 3 \) and \( v_3 = 5 \); greedy strategy will pick \( \{3\} \), but the optimal is \( \{1, 2\} \).

Can show that a slight modification always gives half the optimum profit: pick the better of the output of this algorithm and the largest value item. Also, the algorithms gives better approximations when all item weights are small when compared to \( W \).

6.5.0.6  Towards a Recursive Solution

First guess: \( \text{Opt}(i) \) is the optimum solution value for items \( 1, \ldots, i \).

Observation 6.5.2. Consider an optimal solution \( O \) for \( 1, \ldots, i \).

Case item \( i \not\in O \) then \( O \) is an optimal solution to items \( 1 \) to \( i - 1 \).

Case item \( i \in O \) then \( O - \{i\} \) is an optimum solution for items \( 1 \) to \( n - 1 \) in knapsack of capacity \( W - w_i \).

Subproblems depend also on remaining capacity. Cannot write subproblem only in terms of \( \text{Opt}(1), \ldots, \text{Opt}(i - 1) \).

\( \text{Opt}(i, w) \): optimum profit for items \( 1 \) to \( i \) in knapsack of size \( w \)

Goal: compute \( \text{Opt}(n, W) \)

6.5.0.7  Dynamic Programming Solution

Definition 6.5.3. Let \( \text{Opt}(i, w) \) be the optimal way of picking items from \( 1 \) to \( i \), with total weight not exceeding \( w \).

\[
\text{Opt}(i, w) = \begin{cases} 
0 & \text{if } i = 0 \\
\text{Opt}(i - 1, w) & \text{if } w_i > w \\
\max \left\{ \text{Opt}(i - 1, w), \text{Opt}(i - 1, w - w_i) + v_i \right\} & \text{otherwise}
\end{cases}
\]

6.5.0.8  An Iterative Algorithm

```python
for w = 0 to W do
    M[0, w] = 0
for i = 1 to n do
    for w = 1 to W do
        if (w_i > w) then
            M[i, w] = M[i - 1, w]
        else
            M[i, w] = \text{max}(M[i - 1, w], M[i - 1, w - w_i] + v_i)
```

Running Time
6.5.0.9 Knapsack Algorithm and Polynomial time

(A) Input size for Knapsack: $O(n) + \log W + \sum_{i=1}^{n} (\log v_i + \log w_i)$.
(B) Running time of dynamic programming algorithm: $O(nW)$.
(C) Not a polynomial time algorithm.
(D) Example: $W = 2^n$ and $w_i, v_i \in [1..2^n]$. Input size is $O(n^2)$, running time is $O(n2^n)$ arithmetic/comparisons.
(E) Algorithm is called a \textit{pseudo-polynomial} time algorithm because running time is polynomial if \textit{numbers} in input are of size polynomial in the \textit{combinatorial size} of problem.
(F) Knapsack is \textbf{NP-Hard} if numbers are not polynomial in $n$.

6.6 Traveling Salesman Problem

6.6.0.10 Traveling Salesman Problem

\textbf{Input} A graph $G = (V, E)$ with non-negative edge costs/lengths. $c(e)$ for edge $e$

\textbf{Goal} Find a tour of minimum cost that visits each node.
6.6.0.11 Drawings using TSP
6.6.0.12 Example: optimal tour for cities of a country (which one?)

6.6.0.13 An Exponential Time Algorithm

How many different tours are there? \( n! \)

Stirling’s formula: \( n! \approx \sqrt{n}(n/e)^n \) which is \( \Theta(2^{n \log n}) \) for some constant \( c > 1 \)

Can we do better? Can we get a \( 2^{O(n)} \) time algorithm?

6.6.0.14 Towards a Recursive Solution

(A) Order vertices as \( v_1, v_2, \ldots, v_n \)

(B) \( OPT(S) \): optimum TSP tour for the vertices \( S \subseteq V \) in the graph restricted to \( S \). Want \( OPT(V) \).

Can we compute \( OPT(S) \) recursively?

(A) Say \( v \in S \). What are the two neighbors of \( v \) in optimum tour in \( S \)?

(B) If \( u, w \) are neighbors of \( v \) in an optimum tour of \( S \) then removing \( v \) gives an optimum path from \( u \) to \( w \) visiting all nodes in \( S - \{v\} \).

Path from \( u \) to \( w \) is not a recursive subproblem! Need to find a more general problem to allow recursion.

6.6.0.15 A More General Problem: TSP Path

**Input** A graph \( G = (V, E) \) with non-negative edge costs/lengths \( c(e) \) for edge \( e \) and two nodes \( s, t \)

**Goal** Find a path from \( s \) to \( t \) of minimum cost that visits each node exactly once.
Can solve TSP using above. Do you see how?
Recursion for optimum TSP Path problem:
(A) $OPT(u, v, S)$: optimum TSP Path from $u$ to $v$ in the graph restricted to $S$ (here $u, v \in S$).

6.6.1 A More General Problem: TSP Path

6.6.1.1 Continued...

What is the next node in the optimum path from $u$ to $v$? Suppose it is $w$. Then what is $OPT(u, v, S)$?

$$OPT(u, v, S) = c(u, w) + OPT(w, v, S - \{u\})$$

We do not know $w$! So try all possibilities for $w$.

6.6.1.2 A Recursive Solution

$OPT(u, v, S) = \min_{w \in S, w \neq u, v} \left( c(u, w) + OPT(w, v, S - \{u\}) \right)$

What are the subproblems for the original problem $OPT(s, t, V)$?

$OPT(u, v, S)$ for $u, v \in S$, $S \subseteq V$.

How many subproblems?

(A) number of distinct subsets $S$ of $V$ is at most $2^n$
(B) number of pairs of nodes in a set $S$ is at most $n^2$
(C) hence number of subproblems is $O(n^2 2^n)$

Exercise: Show that one can compute TSP using above dynamic program in $O(n^3 2^n)$ time and $O(n^2 2^n)$ space.

Disadvantage of dynamic programming solution: memory!

6.6.1.3 Dynamic Programming: Postscript

Dynamic Programming = Smart Recursion + Memoization

(A) How to come up with the recursion?
(B) How to recognize that dynamic programming may apply?

6.6.1.4 Some Tips

(A) Problems where there is a natural linear ordering: sequences, paths, intervals, DAGs etc. Recursion based on ordering (left to right or right to left or topological sort) usually works.
(B) Problems involving trees: recursion based on subtrees.
(C) More generally:
   (A) Problem admits a natural recursive divide and conquer
   (B) If optimal solution for whole problem can be simply composed from optimal solution for each separate pieces then plain divide and conquer works directly
   (C) If optimal solution depends on all pieces then can apply dynamic programming if interface/interaction between pieces is limited. Augment recursion to not simply find an optimum solution but also an optimum solution for each possible way to interact with the other pieces.

6.6.1.5 Examples

(A) Longest Increasing Subsequence: break sequence in the middle say. What is the interaction between the two pieces in a solution?
(B) Sequence Alignment: break both sequences in two pieces each. What is the interaction between the two sets of pieces?
(C) Independent Set in a Tree: break tree at root into subtrees. What is the interaction between the subtrees?
(D) Independent Set in an graph: break graph into two graphs. What is the interaction? Very high!
(E) Knapsack: Split items into two sets of half each. What is the interaction?
Bibliography