8.1 Dynamic Programming

8.1.1 Knapsack Problem

**Story:** A thief with a knapsack breaks into a store and finds $n$ items, each of weight $w_i$ and value $v_i$. His goal is to take items worth as much as possible, but the knapsack can only hold a total weight of $W$. The problem now is to find those items that maximize the value in the knapsack, while keeping the total weight $\leq W$.

**Formal problem description:**

**Input:** Set of $n$ items, each of value $v_i$ and weight $w_i$ ($i \in \{1, 2, \ldots, n\}$) and one knapsack of capacity $W$ (weights, values and the capacity are integers).

**Output:** Subset $S$ of the items such that \( \sum_{i \in S} w_i \leq W \) and \( \sum_{i \in S} v_i \) is maximized.

In the following example we will only consider a simplified version of the knapsack problem by defining: \( \forall i : v_i = w_i \)

**Example 1:**

- **Input:** Ordered set \{6, 5, 5\}, knapsack capacity $W = 10$
- **Output:** Greedy: \{6\}, optimal: \{5, 5\}

Clearly the greedy approach is not optimal.

8.1.1.1 Solving the Problem

To start with, we will just try to find the optimal value of the knapsack (not the subset of items that yields this value)

**Definition 8.1** $Knap(i, j)$: Optimal solution (value) using only items 1 to $i$ (of the $n$ available items) and knapsack capacity $j$

**Example 1 (continued):**

\[ Knap(2, 5) = 5, \quad Knap(2, 8) = 6, \quad Knap(1, 5) = 0 \]

8.1.1.2 Divide and Conquer

One way to solve this problem is to use a divide and conquer scheme:

\[
Knap(i, j) = \begin{cases} 
\max (Knap(i-1, j), Knap(i-1, j - w_i) + w_i) & : w_i \leq j \\
Knap(i-1, j) & : w_i > j 
\end{cases}
\]

\[
i \geq 1 \quad : \quad Knap(i, j) = \begin{cases} 
\max (Knap(i-1, j), Knap(i-1, j - w_i) + w_i) & : w_i \leq j \\
Knap(i-1, j) & : w_i > j 
\end{cases}
\]

\[
i = 0 \quad : \quad Knap(i, j) = 0
\]
When this algorithm is run for $Knap(n,W)$, it spans a tree:

![Diagram of tree](image)

**Figure 8.1:** Tree spanned by divide and conquer algorithm

The number of nodes in this tree is $O(2^n)$. This means that the algorithm will have an exponential running time. Since the height of the tree is $n$, we have $2^n$ leaf nodes. But there can be at most $W + 1$ unique subproblems at each level in the tree ($j$ in $Knap(i,j)$ can only range from 0 to $W$ and $j$ is an integer). So there is a level in the tree, at which the number of generated subproblems (at that level) exceeds the possible number of knapsack problems ($W + 1$):

![Diagram of tree structure](image)

**Figure 8.2:** wasteful tree structure
We can improve our wasteful strategy by using a table representation which insures that we only consider each subproblem exactly once:

\[ j: \]

\[
\begin{array}{cccc}
0 & 1 & 2 & \text{W} \\
0 & 0 & 0 & 0 \\
1 & \text{Knap}(i-1, j-w_i) & \text{Knap}(i-1, j) & \text{Knap}(i-1, j-w_i) + w_i, \text{Knap}(i-1, j) \\
2 & \text{include item i (add } w_i \text{ to knapsack)} & \text{do not include item i} & \\
i: & i-1 & i & \text{Knap}(n, W) \\
n & & & \\
\end{array}
\]

Figure 8.3: Table representation of the knapsack problem

The table is generated row by row from the top to the bottom. For each entry \( \text{Knap}(i, j) \), the maximum of \( \text{Knap}(i-1, j) \) and \( \text{Knap}(i-1, j-w_i) + w_i \) is computed (see divide and conquer scheme). We also store a pointer to the entries in the table from which the current entry was computed, (the current entry results from adding an item or from not adding an item).

The running time of this procedure can be easily calculated:

- each entry: \( \Theta(1) \)
- total time: \( \Theta(n \cdot W) \) (table has \((n+1)(W+1)\) entries)

It seems as if the running time of the dynamic programming algorithm is polynomial, but the knapsack problem is known to be NP-complete. How can this be? NP-completeness is defined in terms of bit-complexity and therefore, our input length is not \( W \), but rather \( k \approx \log W \) (length of bit representation of \( W \)). Hence, the bit complexity of our algorithm is \( O(n \cdot 2^k) \) (which is exponential). Therefore the algorithm is called a pseudo-polynomial algorithm.

The space complexity of the algorithm depends on whether we want to obtain the subset of items that yields an optimal solution or whether we are only interested in the optimal value (sum of the weights/values of the items in the knapsack):
• compute only optimal knapsack value: $\Theta(W)$

• also compute subset of items that yields optimal knapsack value: $\Theta(n \cdot W)$

This results from the fact that if we are only looking for the optimal value, we only have to keep 2 rows of the table in memory at each time. If we want the subset of items that produces the optimal value, we have to keep the whole table in memory. When the table is built, we start with the solution on the bottom right side and go backwards along the pointers that we stored for each entry to obtain the items that we have added.

### 8.1.2 When To Use Dynamic Programming?

Properties of a problem that indicate that a dynamic programming algorithm may be the right choice are:

• optimal substructure (the solution for a specific problem size is composed of optimal solutions of smaller size)

• overlap of subproblems (e.g. redundancy in above tree structure)

### 8.1.3 Shortest Path Algorithms

**Base case:**

*Input:* Directed graph $G = (E, V)$, weight function $w : E \rightarrow \mathbb{R}$.

*Output:* $\delta(u, v) =$ minimum over path weights of all paths from vertex $u$ to vertex $v$ ($u, v \in V$)

(a path weight is the sum of the weights of the edges that lie on the path).

**Single Source Shortest Paths:**

*Input:* Source vertex $s \in V$

*Output:* $\forall v \in V$, compute $\delta(s, v)$.

**Single Pair Shortest Path:**

*Input:* Source vertex $u \in V$, destination vertex $v \in V$

*Output:* $\delta(u, v)$

*Algorithm:* Use single source algorithm with $s = u$ and ignore everything but $\delta(u, v)$.

**Single Destination Shortest Paths:**

*Input:* Destination vertex $d \in V$

*Output:* $\forall u \in V$, compute $\delta(u, d)$

*Algorithm:* Reverse all of the edges of the graph, then use the single source algorithm with $s = d$.

**All-pairs Shortest-paths:**

*Input:* Just base case input

*Output:* $\forall u, v \in V$, compute $\delta(u, v)$

*Algorithm:* Use single source shortest path for every vertex $u \in V$.

A better algorithm uses dynamic programming; The Floyd-Warshall algorithm.
8.1.3.1 Floyd-Warshall Algorithm

Definition 8.2 $d_{ij}^{(k)}$: Length of shortest path from $i$ to $j$ where the only intermediate vertices used are $v_1, v_2 \ldots v_k$.

Example 2:

![Diagram](https://via.placeholder.com/150)

Figure 8.4:

The distance between $v_1$ and $v_4$ passing through $v_1$ is 5, the distance them passing through $v_2$ is 4, and the distance between them passing through $v_3$ is 3. Thus:

$$
\begin{align*}
    d_{1,4}^{(1)} &= 5, \\
    d_{1,4}^{(2)} &= 4, \\
    d_{1,4}^{(3)} &= 3,
\end{align*}
$$

We compute $d_{ij}^k$ recursively using the following formula:

$$d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) \text{ all } k \geq 1$$

then the shortest paths will be in $d_{ij}^{[v]}$. A single step is shown in the following figure:

![Diagram](https://via.placeholder.com/150)

Figure 8.5:

This procedure can be represented by the following diagram with the top level representing $k = 0$ and the final level representing the result.
Algorithm:

1. $d_{ij}^{(0)} = w_{ij}$, where $w_{ij} = \infty$ if $(i,j) \notin E$
2. for $k = 1$ to $|V|$
3. for $i = 1$ to $|V|$
4. for $j = 1$ to $|V|$
5. $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
6. return $D(|V|)$

Running time: $O(|V|^3)$

8.1.3.2 Single Source Shortest Path

Special case: All edges have weight = 1.
Use breadth first search from source $s$:

$\delta(s, v) =$ layer where $v$ is discovered.

The visited vertices are expanded by adding new nodes to the outermost layer.
8.1.3.3 Dijkstra’s Algorithm

In this case, the edge weights vary

Maintain \( d[v] \): Shortest path to the vertex \( v \), using only “visited vertices”.

**Algorithm:**

1. \( d[s] = 0 \) (starting from source)
2. \( \forall v \neq s, d[v] = w(s, v) \quad \leftarrow O(|V|) \)
3. \( R = \{s\}, \quad Q = V - \{s\} \)
4. **while** \( |Q| \geq 1 \) (still has unvisited vertices)
5. \( u = v \) such that \( d[V] \) is minimized \( \leftarrow O(|V|) \)
6. \( R = R \cup \{u\}, \quad Q = Q - \{u\} \)
7. **for each** \( v \in \text{adj}[u] \)
8. \( d[v] = \text{min}(d[v], d[u] + w(u, v)) \quad \leftarrow O(|E|) \)
9. **return** \( d \)

**Running time:**

If \( d[v] \) is kept in a binary heap, \textit{update} and \textit{extract minimum} take time \( O(\log |V|) \). The total time is then \( O(|E| \log |V|) \).

If \( d[v] \) is kept in a Fibonacci heap, \textit{update} can be performed in \( O(1) \), and \textit{extract minimum} in \( O(\log |V|) \). The total running time is then \( O(|E| + |V| \cdot \log |V|) \).