Lecture 8: dynamic programming

- Knapsack
- Dynamic programming approach to knapsack
- A practical example for knapsack
- Dijkstra's algorithm revisited
- Dynamic programming idea behind Dijkstra's algorithm
- How to construct dynamic programming algorithms
- Landing scheduling via dynamic programming
- Travelling salesman

Knapsack problem

How to pack as much value with a weight constraint W?

\[
\begin{align*}
\max & \quad \sum_{i=1}^{p} u_i x_i \\
\text{s.t.} & \quad \sum_{i=1}^{p} w_i x_i \leq W \\
& \quad x_i \in \{0, 1\} \quad \text{for all } i
\end{align*}
\]

Dynamic programming solution of knapsack

Let us index by i the items.
Let us index by j the weight restriction.

Question (to be answered by induction)
- If I can take objects 1, 2, 3, … i,
- How much value can I take away
- Given that I am restricted to take a maximum weight of j

This question is to be answered by induction, on i AND j

Dynamic programming induction relation

Introduce a quantity d(i,j), indexed by
- i, the number of items to be taken away
  \(i = 0, 1, 2, \ldots, p\)
- j, the weight restriction
  \(j = 1, 2, \ldots, W\)

Note that all quantities of this problem have to be integer

Introduce d(i,j), the maximum value of the selected items, if we are allowed to take items 1 to i, and we have a weight restriction of j

We will thus compute d(i,j) recursively, in an array of dimension p x W.

Dynamic programming induction relation

\[
d(i, j) = \max \{d(i - 1, j), u_i + d(i - 1, j - w_i)\}
\]

maximum value of the selected items, if we are allowed to take items 1 to i, and we have a weight restriction of j

Dynamic programming induction relation

\[
d(i, j) = \max \{d(i - 1, j), u_i + d(i - 1, j - w_i)\}
\]

maximum value of the selected items, if we are allowed to take items 1 to i, and we have a weight restriction of j
Dynamic programming induction relation

\[ d(i, j) = \max \{d(i - 1, j), u_i + d(i - 1, j - w_i)\} \]

- value of for i-1 items, with restriction j minus weight of object i
- maximum value of the selected items, if we are allowed to take items 1 to i, and we have a weight restriction of j

Dynamic programming induction relation

\[ d(i, j) = \max \{d(i - 1, j), u_i + d(i - 1, j - w_i)\} \]

- keep i-1 items (do nothing)
- maximum value of the selected items, if we are allowed to take items 1 to i, and we have a weight restriction of j

Dynamic programming induction relation

\[ d(i, j) = \max \{d(i - 1, j), u_i + d(i - 1, j - w_i)\} \]

- keep i-1 items (do nothing)
- value of item i

Dynamic programming induction relation

\[ d(i, j) = \max \{d(i - 1, j), u_i + d(i - 1, j - w_i)\} \]

- keep i-1 items (do nothing)
- maximum value of the selected items, if we are allowed to take items 1 to i, and we have a weight restriction of j

Dynamic programming induction relation

\[ d(i, j) = \max \{d(i - 1, j), u_i + d(i - 1, j - w_i)\} \]

- keep i-1 items (do nothing)
- value of item i

Dynamic programming induction relation

\[ d(i, j) = \max \{d(i - 1, j), u_i + d(i - 1, j - w_i)\} \]

- keep i-1 items (do nothing)
- maximum value of the selected items, if we are allowed to take items 1 to i, and we have a weight restriction of j

Dynamic programming induction relation

\[ d(i, j) = \max \{d(i - 1, j), u_i + d(i - 1, j - w_i)\} \]

- keep i-1 items (do nothing)
- value of item i

Best option: - keep the previous selection
- add the new object

What do we do next?

We fill an array of size p x W

\[
\begin{align*}
  w_1 &= 1 \\
  w_2 &= 7 \\
  w_3 &= 4 \\
  w_4 &= 2
\end{align*}
\]

Weight = value

p = 4

W = 11 (arbitrary, but less than 14 obviously)

\[ d(i, j) = \max \{d(i - 1, j), u_i + d(i - 1, j - w_i)\} \]
What do we do next?

We fill an array of size $p \times W$

$$d(i, j) = \max \{W(i-1,j), u_i + d(i-1, j - w_i)\}$$

<table>
<thead>
<tr>
<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>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>(1) (1) (1) (1) (1) (1) (1) (1) (1) (1) (1)</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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>(1) (1) (1) (1) (1) (1) (1) (1) (1) (1) (1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Weight restriction $j$ ($j=1,2, \ldots, W$)

Items picked ($i=1, 2, 3, 4, \ldots$)

---

We fill an array of size $p \times W$

$$d(i, j) = \max \{d(i-1, j), u_i + d(i-1, j - w_i)\}$$

<table>
<thead>
<tr>
<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>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>(1) (1) (1) (1) (1) (1) (1) (1) (1) (1) (1)</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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>(1) (1) (1) (1) (1) (1) (1) (1) (1) (1) (1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Weight restriction $j$ ($j=1,2, \ldots, W$)

Items picked ($i=1, 2, 3, 4, \ldots$)

---

We fill an array of size $p \times W$

$$w_1 = 1, \ w_2 = 7, \ w_3 = 4, \ w_4 = 2$$

$$d(i, j) = \max \{d(i-1, j), u_i + d(i-1, j - w_i)\}$$

<table>
<thead>
<tr>
<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>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>(1) (1) (1) (1) (1) (1) (1) (1) (1) (1) (1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>(1) (1) (1) (1) (1) (1) (1) (1) (1) (1) (1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>(1) (1) (1) (1) (1) (1) (1) (1) (1) (1) (1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Weight restriction $j$ ($j=1,2, \ldots, W$)

Items picked ($i=1, 2, 3, 4, \ldots$)

---

We fill an array of size $p \times W$

$$w_1 = 1, \ w_2 = 7, \ w_3 = 4, \ w_4 = 2$$

$$d(i, j) = \max \{d(i-1, j), u_i + d(i-1, j - w_i)\}$$

<table>
<thead>
<tr>
<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>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>(1) (1) (1) (1) (1) (1) (1) (1) (1) (1) (1)</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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>(1) (1) (1) (1) (1) (1) (1) (1) (1) (1) (1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Weight restriction $j$ ($j=1,2, \ldots, W$)

Items picked ($i=1, 2, 3, 4, \ldots$)
We fill an array of size $p \times W$

$$w_1 = 1, \ w_2 = 7, \ w_3 = 4, \ w_4 = 2$$

No uniqueness: could pick also item 2

Weight restriction $j$ ($j=1,2, \ldots, W$)

<table>
<thead>
<tr>
<th></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>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
</tr>
<tr>
<td>3</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
</tr>
<tr>
<td>4</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
</tr>
</tbody>
</table>

Items picked ($j=1,2, \ldots, W$)

<table>
<thead>
<tr>
<th></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>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>(4)</td>
<td>(4)</td>
<td>(4)</td>
<td>(4)</td>
<td>(4)</td>
<td>(4)</td>
<td>(4)</td>
<td>(4)</td>
<td>(4)</td>
<td>(4)</td>
<td>(4)</td>
</tr>
</tbody>
</table>

What do we do next?

We fill an array of size $p \times W$

$$d(i, j) = \max \{d(i - 1, j), u_i + d(i - 1, j - w_i)\}$$

Weight restriction $j$ ($j=1,2, \ldots, W$)

<table>
<thead>
<tr>
<th></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>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</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>2</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>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Final result

Weight restriction $j$ ($j=1,2, \ldots, W$)

<table>
<thead>
<tr>
<th></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>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
</tr>
<tr>
<td>3</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
</tr>
<tr>
<td>4</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
</tr>
</tbody>
</table>

Items picked ($j=1,2, \ldots, W$)

<table>
<thead>
<tr>
<th></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>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Dynamic programming: main idea

Main idea: when solving an integer program, in order to avoid enumeration (too expensive computationally), cut the problem in two and use induction:

At current step: assume you know the best solution at previous steps

Compute the best solution for the current step, and pair it with the solution at the previous steps

A more precise definition of dynamic programming will be given later in class.

Dijkstra's algorithm revisited

begin
  S:=Ø
  d[i]:=∞ for each node i
  d[s]:=0 and pred[s]=Ø
  while |S|<n do
    begin
      let i in S for which d[i]=min(d[j], j in S)
      S = S U {i}
      S* = S* \ {i}
      for each (i,j) in the graph do
        if d[j]>d[i]+cij
          then
            d[j]:=d[i]+cij
            pred[j]:=i
    end
  end
end

Core of Dijkstra's algorithm

$$d(j) := \min_{\text{connected nodes}} \{d(j), d(i) + c_{ij}\}$$

Value at next iteration

best between the previous iteration and whatever is best at this iteration

previous iteration (other path)

for all connected node i, compute the path length to node i plus the length from i to j

This is the main idea under dynamic programming algorithms
How to construct dynamic programming algorithms

1) View the choice of a feasible solution as a sequence of decisions occurring in stages, and so that the total cost is the sum of the costs of individual decisions.

2) Define the state as a summary of all relevant decisions

3) Determine which state transitions are possible. Let the cost of each state transition be the cost of the corresponding decision.

4) Write a recursion on the optimal cost from the origin state to a destination state

Shaded square indicates that time is meant for the algorithm (more about this in the lab).

[Introduction to linear optimization, Bertsimas, Tsitsiklis, 1997]

Landing scheduling through dynamic programming

Example of data used for this type of problem:

| t_{ij} | j-th possible landing time of aircraft i |
| n_i | number of possible landing times for aircraft i |
| δ(t, j) | Maximal minimum spacing between any two aircraft, for the subset of aircraft 1, 2, … , i, if the aircraft number i is assigned the arrival time t_{ij} |

Initialization of the recursion:
- Aircraft 1 should obviously arrive as early as possible
- If aircraft 2 is assigned the j-th arrival time, the spacing between aircraft 1 and j is obviously

δ(2, j) = t_{2,j} - t_{1,1}

This looks like a reasonable solution.
Landing scheduling through dynamic programming

\[ \delta(i,j) = \max_{j'=1}^{n_i-1} \left\{ \min\{t_{i,j} - t_{i-1,j'}, \delta(i-1,j')\} \right\} \]

Recursion is done with variable i, i.e. with the number of aircraft. For a number i of aircraft, this represents the largest smallest spacing if aircraft i arrives at time j.

Initialization of the recursion:
\[ \delta(2,j) = t_{2,j} - t_{1,1} \]

Number of possible landing times for aircraft i
\[ n_i \]

Landing scheduling through dynamic programming

\[ \delta(i,j) = \max_{j'=1}^{n_i-1} \left\{ \min\{t_{i,j} - t_{i-1,j'}, \delta(i-1,j')\} \right\} \]

Maximal minimum separation between any two aircraft within the first i-1 aircraft, if aircraft i is assigned arrival time j.

Initialization of the recursion:
\[ \delta(2,j) = t_{2,j} - t_{1,1} \]

j-th possible landing time of aircraft i
\[ t_{i,j} \]

Landing scheduling through dynamic programming

Time separation resulting from assigning aircraft i to its j-th arrival time and aircraft i-1 to its j'-th arrival time.

Initialization of the recursion:
\[ \delta(2,j) = t_{2,j} - t_{1,1} \]

Number of possible landing times for aircraft i
\[ n_i \]

Landing scheduling through dynamic programming

\[ \delta(i,j) = \max_{j'=1}^{n_i-1} \left\{ \min\{t_{i,j} - t_{i-1,j'}, \delta(i-1,j')\} \right\} \]

For a given j' (aircraft i-1's arrival time), the overall resulting minimum separation is the worst between
- aircraft i and aircraft i-1
- any two other aircraft within 1, 2, ..., i-1

Initialization of the recursion:
\[ \delta(2,j) = t_{2,j} - t_{1,1} \]

For aircraft i, we have to compute the maximum spacing so far when trying all possible assignments for the previous i-1 aircraft

Initialization of the recursion:
\[ \delta(2,j) = t_{2,j} - t_{1,1} \]

Number of possible arrival times for aircraft i-1
\[ n_{i-1} \]

Landing scheduling through dynamic programming

\[ \delta(i,j) = \max_{j'=1}^{n_i-1} \left\{ \min\{t_{i,j} - t_{i-1,j'}, \delta(i-1,j')\} \right\} \]

For aircraft i, we have to compute the maximum spacing so far when trying all possible assignments for the previous i-1 aircraft

Initialization of the recursion:
\[ \delta(2,j) = t_{2,j} - t_{1,1} \]
Landing scheduling through dynamic programming

\[ \delta(i, j) = \max_{j' = 1}^{n_i - 1} \min \{ t_{i, j} - t_{i - 1, j'}, \delta(i - 1, j') \} \]

The goal of the dynamic algorithm is to compute the largest spacing, by trying the best among all possible assignments of aircraft \( i - 1 \) (since this is the induction step between \( i - 1 \) and \( i \)).

- \( t_{i, j} \): j-th possible landing time of aircraft \( i \)
- \( n_i \): number of possible landing times for aircraft \( i \)
- \( \delta(i, j) \): Maximal minimum spacing between any two aircraft, for the subset of aircraft 1, 2, ... , i, if the aircraft number i is assigned the arrival time \( t_{i,j} \)

Initialization of the recursion:

\[ \delta(2, j) = t_{2, j} - t_{1, j} \]

How to construct dynamic programming algorithms

1) View the choice of a feasible solution as a sequence of decisions occurring in stages, and so that the total cost is the sum of the costs of individual decisions.

2) Define the state as a summary of all relevant decisions

3) Determine which state transitions are possible. Let the cost of each state transition be the cost of the corresponding decision.

4) Write a recursion on the optimal cost from the initial state to a destination state

Shaded square indicates that time is meant for the algorithm (more about this in the lab).

[Introduction to linear optimization, Bertsimas, Tsitsiklis, 1997]
Traveling salesman: dynamic programming solution

\[ C(S, k) = \min_{m \in S \setminus \{k\}} (C(S\setminus \{k\}, m) + c_{mk}) \]

- **S**: subset of cities including city 1 (departure city)
- **k**: a city in S

Induction is done on S

---

Cost to go from city 1 to city k: zero

\[ C(S, 1) = \min_{m \in S \setminus \{1\}} (C(S\setminus \{1\}, m) + c_{mk}) \]

Where m is a node in S, but not k

Induction is done on S

---

**Best over all possible m (not equal to k)**

\[ C(S, k) = \min_{m \in S \setminus \{k\}} (C(S\setminus \{k\}, m) + c_{mk}) \]

- **C(S, k)**: shortest path from city 1 to city k that visits all nodes in S
- **S**: subset of cities including city 1 (departure city)
- **k**: a city in S

Induction is done on S
How to construct dynamic programming algorithms

1) View the choice of a feasible solution as a sequence of decisions occurring in stages, and so that the total cost is the sum of the costs of individual decisions.

2) Define the state as a summary of all relevant decisions

3) Determine which state transitions are possible. Let the cost of each state transition be the cost of the corresponding decision.

4) Write a recursion on the optimal cost from the state to a state

Shaded square indicates that time is meant for the algorithm (more about this in the lab).

[Introduction to linear optimization, Bertsimas, Tsitsiklis, 1997]