COMP251: Dynamic programming (1)

Jérôme Waldispühl
School of Computer Science
McGill University

Based on (Cormen et al., 2002) & (Kleinberg & Tardos, 2005)
Algorithms paradigms

• Greedy:
  o Build up a solution incrementally.
  o Iteratively decompose and reduce the size of the problem.
  o Top-down approach.

• Dynamic programming:
  o Solve all possible sub-problems.
  o Assemble them to build up solutions to larger problems.
  o Bottom-up approach.
INTRODUCTION
Activity-selection Problem

• **Input:** Set $S$ of $n$ activities, $a_1, a_2, ..., a_n$.
  – $s_i =$ start time of activity $i$.
  – $f_i =$ finish time of activity $i$.

• **Output:** Subset $A$ of maximum number of compatible activities.
  – 2 activities are compatible, if their intervals do not overlap.

Example:

Activities in each line are compatible.
Activity-selection Problem

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>$f_i$</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>9</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

Activities sorted by finishing time.
Activity-selection Problem

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$f_i$</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Activities sorted by finishing time.

Activities sorted by finishing time.
Activity-selection Problem

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>$f_i$</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>9</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

Activities sorted by finishing time.
### Activity-selection Problem

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>$f_i$</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>9</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

Activities sorted by finishing time.

![Diagram showing activities sorted by finishing time.](image-url)
Optimal sub-structure

- Let $S_{ij}$ = subset of activities in $S$ that start after $a_i$ finishes and finish before $a_j$ starts.

$$S_{ij} = \{ a_k \in S : \forall i, j \ f_i \leq s_k < f_k \leq s_j \}$$

- $A_{ij}$ = optimal solution to $S_{ij}$

- $A_{ij} = A_{ik} \cup \{ a_k \} \cup A_{kj}$
Greedy choice

<table>
<thead>
<tr>
<th># subproblems in optimal solution</th>
<th>Before theorem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td># choices to consider</td>
<td>j-i-1</td>
</tr>
</tbody>
</table>

\[ A_{ij} = A_{ik} \cup \{ a_k \} \cup A_{kj} \]

We can solve the problem \( S_{ij} \) top-down:

- Consider all \( a_m \in S_{ij} \)
- Solve \( S_{im} \) and \( S_{mj} \)
- Pick the best \( m \) such that \( A_{im} = A_{im} \cup \{ a_k \} \cup A_{im} \)
Greedy choice

Theorem:
Let $S_{ij} \neq \emptyset$, and let $a_m$ be the activity in $S_{ij}$ with the earliest finish time: $f_m = \min\{ f_k : a_k \in S_{ij} \}$. Then:

1. $a_m$ is used in some maximum-size subset of mutually compatible activities of $S_{ij}$.
2. $S_{im} = \emptyset$, so that choosing $a_m$ leaves $S_{mj}$ as the only nonempty subproblem.
### Greedy choice

<table>
<thead>
<tr>
<th></th>
<th>Before theorem</th>
<th>After theorem</th>
</tr>
</thead>
<tbody>
<tr>
<td># subproblems in</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>optimal solution</td>
<td></td>
<td></td>
</tr>
<tr>
<td># choices to consider</td>
<td>$j - i - 1$</td>
<td>1</td>
</tr>
</tbody>
</table>

$A_{ij} = A_{ik} \cup \{ a_k \} \cup A_{kj}
\quad A_{ij} = \{ a_m \} \cup A_{mj}$

We can now solve the problem $S_{ij}$ top-down:

- Choose $a_m \in S_{ij}$ with the earliest finish time (greedy choice).
- Solve $S_{mj}$. 
Challenges

- Greedy choice is not always available.
- How to solve problem that have optimal substructures?
WEIGHTED INTERVAL SCHEDULING
Weighted interval scheduling

- **Input:** Set $S$ of $n$ activities, $a_1, a_2, \ldots, a_n$.
  - $s_i =$ start time of activity $i$.
  - $f_i =$ finish time of activity $i$.
  - $w_i =$ weight of activity $i$.

- **Output:** find maximum weight subset of mutually compatible activities.
  - 2 activities are compatible, if their intervals do not overlap.

Example:
Application of the greedy algorithm

W=9

W=3
Discussion

- **Optimal substructure:** ✓
  - $A_{ij} = \text{optimal solution to } S_{ij}$
  - $A_{ij} = A_{ik} \cup \{ a_k \} \cup A_{kj}$

- **Greedy Choice:** ❌
  - Select the activity with earliest finish time.
Data structure

Notation: All activities are sorted by finishing time $f_1 \leq f_2 \leq \ldots \leq f_n$

Definition: $p(j)$ = largest index $i < j$ such that activity/job $i$ is compatible with activity/job $j$.

Examples: $p(6)=4$, $p(5)=2$, $p(4)=2$, $p(2)=0$. 
Binary Choice

**Notation:** \( \text{OPT}(j) = \text{value of the optimal solution to the problem} \)
\[= \max \text{ total weight of compatible activities } 1 \ldots j \]

**Case 1:** \( \text{OPT} \) selects activity \( j \)
- Add weight \( w_j \)
- Cannot use incompatible activities
- Must include optimal solution on remaining compatible activities \( \{1, 2, \ldots, p(j)\} \).

**Case 2:** \( \text{OPT} \) does not select activity \( j \)
Must include optimal solution on others activities \( \{1, 2, \ldots, j-1\} \).

\[
\text{OPT}(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max\{w_j + \text{OPT}(p(j)), \text{OPT}(j-1)\} & \text{otherwise}
\end{cases}
\]

Optimal substructure property
Recursive call

Input: \( n, s[1..n], f[1..n], v[1..n] \)

Sort jobs by finish time so that \( f[1] \leq f[2] \leq ... \leq f[n] \).

Compute \( p[1], p[2], ..., p[n] \).

\( \text{Compute-Opt}(j) \)

if \( j = 0 \)
   return \( 0 \).
else
   return \( \max(v[j] + \text{Compute-Opt}(p[j]), \text{Compute-Opt}(j-1)) \).
Brute Force Approach

Observation: $\text{OPT}(j)$ is calculated multiple times...
Memoization

Memoization: Cache results of each subproblem; lookup as needed.

Input: n, s[1..n], f[1..n], v[1..n]
Sort jobs by finish time so that f[1]≤f[2]≤...≤f[n].
Compute p[1], p[2], ..., p[n].

for j = 1 to n
    M[j] ← empty.
M[0] ← 0.

M-Compute-Opt(j)
if M[j] is empty
    M[j] ← max(v[j]+M-Compute-Opt(p[j]), M-Compute-Opt(j-1)).
return M[j].
Running time

Claim. Memoized version of algorithm takes \( O(n \log n) \) time.
- Sort by finish time: \( O(n \log n) \).
- Computing \( p(\cdot) \): \( O(n \log n) \) via sorting by start time.

- \textsc{M-Compute-Opt}(j): each invocation takes \( O(1) \) time and either
  - (i) returns an existing value \( M[j] \)
  - (ii) fills in one new entry \( M[j] \) and makes two recursive calls

- Progress measure \( \Phi = \# \) nonempty entries of \( M[] \).
  - initially \( \Phi = 0 \), throughout \( \Phi \leq n \).
  - (ii) increases \( \Phi \) by 1 \( \Rightarrow \) at most \( 2n \) recursive calls.

- Overall running time of \textsc{M-Compute-Opt}(n) is \( O(n) \).

Remark. \( O(n) \) if jobs are presorted by start and finish times.
DYNAMIC PROGRAMMING
Bottom-up

Observation: When we compute $M[j]$, we only need values $M[k]$ for $k<j$.

```
BOTTOM-UP (n; s1, ..., sn; f1, ..., fn; v1, ..., vn)
Sort jobs by finish time so that $f_1 \leq f_2 \leq \ldots \leq f_n$.
Compute $p(1), p(2), \ldots, p(n)$.
$M[0] \leftarrow 0$
for $j = 1$ to $n$
    $M[j] \leftarrow \max \{ v_j + M[p(j)], M[j-1] \}$
```

Main Idea of Dynamic Programming: Solve the sub-problems in an order that makes sure when you need an answer, it's already been computed.
Finding a solution

Dyn. Prog. algorithm computes optimal value.
Q: How to find solution itself?
A: Bactrack!

Find-Solution(j)
if j = 0
    return ∅.
else if (v[j] + M[p[j]] > M[j−1])
    return { j } ∪ Find-Solution(p[j])
else
    return Find-Solution(j−1).

Analysis. # of recursive calls ≤ n ⇒ O(n).
Example: Computing solution

<table>
<thead>
<tr>
<th>activity</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>predecessor</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Best weight M</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$V_j + M[p(j)]$</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$M[j-1]$</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

(1) Activities sorted by finishing time. (2) Weight equal to the length of activity.
Example: Computing solution

<table>
<thead>
<tr>
<th>activity</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>predecessor</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Best weight M</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$V_j + M[p(j)]$</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$M[j-1]$</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

(1) Activities sorted by finishing time. (2) Weight equal to the length of activity.
Example: Computing solution

<table>
<thead>
<tr>
<th>activity</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>predecessor</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Best weight M</td>
<td>2</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>V_j+M[p(j)]</td>
<td>2</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>M[j-1]</td>
<td>0</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

(1) Activities sorted by finishing time. (2) Weight equal to the length of activity.
**Example: Computing solution**

<table>
<thead>
<tr>
<th>activity</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>predecessor</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Best weight M</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$V_j + M[p(j)]$</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$M[j-1]$</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

(1) Activities sorted by finishing time. (2) Weight equal to the length of activity.
Example: Computing solution

<table>
<thead>
<tr>
<th>activity</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>predecessor</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Best weight M</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>9</td>
<td>-</td>
</tr>
<tr>
<td>$V_j + M[p(j)]$</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>9</td>
<td>-</td>
</tr>
<tr>
<td>$M[j-1]$</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>-</td>
</tr>
</tbody>
</table>

(1) Activities sorted by finishing time. (2) Weight equal to the length of activity.
Example: Computing solution

<table>
<thead>
<tr>
<th>activity</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>predecessor</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Best weight M</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>$V_j+M[p(j)]$</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>$M[j-1]$</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>9</td>
</tr>
</tbody>
</table>

(1) Activities sorted by finishing time. (2) Weight equal to the length of activity.
Example: Reconstruction

<table>
<thead>
<tr>
<th>activity</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>predecessor</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Best weight M</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>$V_j + M[p(j)]$</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>$M[j-1]$</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>9</td>
</tr>
</tbody>
</table>

(1) Activities sorted by finishing time. (2) Weight equal to the length of activity.
Example: Reconstruction

<table>
<thead>
<tr>
<th>activity</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>predecessor</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Best weight M</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>$V_j + M[p(j)]$</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>$M[j-1]$</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>9</td>
</tr>
</tbody>
</table>

1. Activities sorted by finishing time.
2. Weight equal to the length of activity.
**Example: Reconstruction**

<table>
<thead>
<tr>
<th>activity</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>predecessor</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

| Best weight M | 2 | 3 | 4 | 9 | 9 |
| V_{j} + M[p(j)] | 2 | 3 | 4 | 9 | 8 |
| M[j-1] | 0 | 2 | 3 | 4 | 9 |

(1) Activities sorted by finishing time. (2) Weight equal to the length of activity.
SINGLE SOURCE SHORTEST PATHS
Modeling as graphs

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

Weight of path $p = \langle v_0, v_1, ..., v_k \rangle$

$$= \sum_{k=1}^{n} w(v_{k-1}, v_k)$$

= sum of edge weights on path $p$.

Shortest-path weight $u$ to $v$:

$$\delta(u, v) = \begin{cases} 
\min \{ w(p) : u \rightarrow^p v \} & \text{If there exists a path } u \rightarrow^p v. \\
\infty & \text{Otherwise.}
\end{cases}$$

Shortest path $u$ to $v$ is any path $p$ such that $w(p) = \delta(u, v)$.

Generalization of breadth-first search to weighted graphs.
Dijkstra’s algorithm

• No negative-weight edges.
• Weighted version of BFS:
  • Instead of a FIFO queue, uses a priority queue.
  • Keys are shortest-path weights ($d[v]$).
• Greedy choice: At each step we choose the light edge.

How to deal with negative weight edges?
• Allow re-insertion in queue? $\implies$ Exponential running time...
• Add constant to each edge?

Not working...
Dijkstra’s algorithm

DIJKSTRA(V, E, w, s)
INIT-SINGLE-SOURCE(V, s)
S ← ∅
Q ← V
while Q ≠ ∅ do
  u ← EXTRACT-MIN(Q)
  S ← S ∪ {u}
  for each vertex v ∈ Adj[u] do
    RELAX(u, v, w)
Example
Example
Example
Example
Example

```
Q
```

Graph with nodes and edges:
- Node labels: s, 0, 6, 8, 5, y, x, z
- Edges with weights:
  - s to 0: 2
  - 0 to 5: 5
  - 5 to 6: 10
  - 6 to 8: 2
  - 8 to x: 7
  - 0 to t: 2
  - t to 8: 2
  - 6 to 10: 3
  - 5 to y: 1
  - y to 6: 6

Node 0 is the source (s), and node 10 is the sink (t).
Example
Bellman-Ford Algorithm

- Allows negative-weight edges.
- Computes $d[v]$ and $\pi[v]$ for all $v \in V$.
- Returns TRUE if no negative-weight cycles reachable from $s$, FALSE otherwise.

If Bellman-Ford has not converged after $V(G) - 1$ iterations, then there cannot be a shortest path tree, so there must be a negative weight cycle.
Bellman-Ford Algorithm

- Can have negative-weight edges.
- Will “detect” **reachable** negative-weight cycles.

```
Initialize(G, s);
for i := 1 to \(|V[G]| - 1\) do
    for each (u, v) in E[G] do
        Relax(u, v, w)
    for each (u, v) in E[G] do
        if d[v] > d[u] + w(u, v) then
            return false
    return true
```

Time Complexity is $O(VE)$. 
Example

Diagram of a weighted graph with nodes labeled z, 0, ∞, u, v, x, y, and edges labeled with weights 6, 8, 7, 2, 9, 5, 7, 3, 4. The graph includes cycles and paths with positive and negative weights.
Example
Example

Graph with nodes labeled 0, 2, 7, and 11, and edges with weights:
- (0, 6) with weight 6
- (6, 11) with weight 5
- (6, 11) with negative weight -2
- (11, 2) with weight 7
- (11, 2) with negative weight -3
- (2, 0) with weight 7
- (0, 6) with negative weight -4
- (6, 7) with weight 2
- (7, 11) with weight 9
- (11, 7) with weight 5
- (7, 2) with weight 2
- (2, 6) with weight 6
Example
Example
Another Look at Bellman-Ford

Note: This is essentially dynamic programming.
Let $d(i, j) =$ cost of the shortest path from $s$ to $i$ that is at most $j$ hops.

$$d(i, j) = \begin{cases} 
0 & \text{if } i = s \land j = 0 \\
\infty & \text{if } i \neq s \land j = 0 \\
\min\{d(k, j-1) + w(k, i) : i \in \text{Adj}(k)\} \cup \{d(i, j-1)\} & \text{if } j > 0
\end{cases}$$