COMP251: Dynamic programming (1)

Jérôme Waldispühl
School of Computer Science
McGill University

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

• **Greedy**: Build up a solution incrementally, myopically optimizing some local criterion.

• **Dynamic programming**: Break up a problem into a series of overlapping subproblems, and build up solutions to larger and larger subproblems.

• **Divide-and-conquer**: Break up a problem into independent subproblems, solve each subproblem, and combine solution to subproblems to form solution to original problem.
Activity-selection Problem

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

• **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.
Recursive Algorithm

Recursive-Activity-Selector \((s, f, i, j)\)
1. \(m \leftarrow i+1\)
2. while \(m < j\) and \(s_m < f_i\)
3.   do \(m \leftarrow m+1\)
4. if \(m < j\)
5.   then return \(\{a_m\} \cup \) Recursive-Activity-Selector\((s, f, m, j)\)
6. else return \(\emptyset\)

Initial Call: Recursive-Activity-Selector \((s, f, 0, n+1)\)
Complexity: \(\Theta(n)\)

Remark: Straightforward to convert the algorithm to an iterative one.
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.
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.
Optimal sub-structure

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

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

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

• $A_{ij} = A_{ik} \cup \{ a_k \} \cup A_{kj}$
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.
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.** $OPT(j) =$ value of optimal solution to the problem consisting of job requests $1, 2, ..., j$.

**Case 1.** $OPT$ selects job $j$.
- Collect profit $v_j$.
- Can't use incompatible jobs $\{ p(j) + 1, p(j) + 2, ..., j - 1 \}$.
- Must include optimal solution to problem consisting of remaining compatible jobs $1, 2, ..., p(j)$.

**Case 2.** $OPT$ does not select job $j$.
- Must include optimal solution to problem consisting of remaining compatible jobs $1, 2, ..., j - 1$.

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

\textbf{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.** Recursive algorithm fails spectacularly because of redundant subproblems ⇒ exponential algorithms.

**Ex.** Number of recursive calls for family of "layered" instances grows like Fibonacci sequence.
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.

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

- Overall running time of **M-COMPUTE-OPT(n)** is $O(n)$.  

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

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

BOTTOM-UP (n, s1, ..., sn , f1, ..., fn , v1, ..., vn)

Sort jobs by finish time so that f1 ≤ f2 ≤ ... ≤ fn.
Compute p(1), p(2), ..., p(n).

M[0]←0
for j = 1 TO n
    M[j] ← max { vj + M[p(j)], M[j−1] }

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

Q. DP algorithm computes optimal value. How to find solution itself?
A. Make a second pass.

\[
\text{Find-Solution}(j) \\
\text{if } j = 0 \\
\quad \text{return } \emptyset. \\
\text{else if } (v[j] + M[p[j]] > M[j-1]) \\
\quad \text{return } \{ j \} \cup \text{Find-Solution}(p[j]) \\
\text{else} \\
\quad \text{return } \text{Find-Solution}(j-1). \\
\]

Analysis. # of recursive calls \( \leq n \Rightarrow O(n) \).
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</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.

![Diagram showing activities a1 to a5 with their respective durations and precedence relationships.](image-url)
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</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</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</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$V_j + M[p(j)]$</td>
<td>2</td>
<td>3</td>
<td>3</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</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>( V_j + M[p(j)] )</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>10</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</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>V_j+M[p(j)]</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>M[j-1]</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>10</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</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>$V_j + M[p(j)]$</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>$M[j-1]$</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>10</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>

| V_j+M[p(j)] | 2 | 3 | 4 | 10 | 10 |
| M[j-1]      | 0 | 2 | 3 | 4  | 10 |

(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</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>$V_j + M[p(j)]$</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>$M[j-1]$</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>10</td>
</tr>
</tbody>
</table>

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

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

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

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

= sum of edge weights on path $p$.

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

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

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 ($d[v]$).
• Have two sets of vertices:
  • $S =$ vertices whose final shortest-path weights are determined,
  • $Q =$ priority queue $= V − S$.
• Similar Prim’s algorithm, but computing $d[v]$, and using shortest-path weights as keys.
• Greedy choice: At each step we choose the light edge.
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

\begin{center}
\begin{tikzpicture}[node distance=2cm,auto,main node/.style={circle,draw}]

\node[main node] (s) {$s$};
\node[main node] (0) [right of=s] {$0$};
\node[main node] (t) [right of=0] {$t$};
\node[main node] (x) [right of=t] {$x$};
\node[main node] (y) [below of=0] {$y$};
\node[main node] (z) [below of=x] {$z$};
\node[main node] (infinity) [above of=t] {$\infty$};
\node[main node] (infinity2) [above of=x] {$\infty$};

\path[->]
(s) edge node [left] {5} (0)
(s) edge node [right] {10} (infinity)
(0) edge node [above] {2} (t)
(0) edge node [below] {2} (y)
(t) edge node [above] {2} (infinity)
(t) edge node [right] {1} (x)
(t) edge node [left] {1} (infinity2)
(y) edge node [above] {4} (infinity2)
(y) edge node [below] {3} (z)
(z) edge node [right] {7} (infinity2)
(z) edge node [left] {6} (infinity)

\end{tikzpicture}
\end{center}

Q \begin{array}{c|c|c|c|c|c}
\hline
s & t & y & x & z \\
\hline
\end{array}
Example

Q | y | t | x | z
---|---|---|---|---

Diagram with nodes labeled 0, 10, 5, t, x, y, z, and edges with weights 2, 1, 4, 3, 2, 7.
Example

\[ Q \begin{tabular}{|l|l|l|l|}
\hline
 t & x & z & \\
\hline
\end{tabular} \]
Example

```plaintext
Q
  x  z
  |   |
  |   |
  |   |
```

[Diagram of a graph with nodes and edges labeled with numbers, showing a path from s to x through y and z.]
Example

\[
\begin{array}{c}
\text{Q} \\
\end{array}
\]
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)
  od
od;
for each (u, v) in E[G] do
  if d[v] > d[u] + w(u, v) then
    return false
  fi
od;
return true

Time Complexity is O(VE).
Example
Example

Graph representation with nodes labeled 0, 6, 7, ∞, u, v, x, and y, and edges labeled with weights 6, 5, -2, -3, -4, 2, 7, and 9.
Example
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.

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

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>\infty</td>
<td>\infty</td>
<td>\infty</td>
<td>\infty</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>6</td>
<td>\infty</td>
<td>7</td>
<td>\infty</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>6</td>
<td>4</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>-2</td>
</tr>
</tbody>
</table>
General properties

**Lemma 24.1:** Let $p = \langle v_1, v_2, ..., v_k \rangle$ be a SP from $v_1$ to $v_k$. Then, $p_{ij} = \langle v_i, v_{i+1}, ..., v_j \rangle$ is a SP from $v_i$ to $v_j$, where $1 \leq i \leq j \leq k$.

So, we have the **optimal-substructure property**.

Bellman-Ford’s algorithm uses **dynamic programming**.

Dijkstra’s algorithm uses the **greedy approach**.

Let $\delta(u, v) =$ weight of SP from $u$ to $v$. 