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, $a_1, a_2, ..., a_n$.
  - $s_i$ = 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 sorted by finishing time.

Recursive Algorithm

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.

### Optimal sub-structure
- Let $S_i$ = subset of activities in $S$ that start after $a_i$ finishes and finish before $a_i$ starts.
  
  \[ S_i = \{ a_j \in S : \forall i, j \quad f_i \leq s_j < f_j \leq s_i \} \]

- $A_j$ = optimal solution to $S_j$
- $A_i = A_k \cup \{ a_k \} \cup A_j$

### Greedy choice

**Theorem:**

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

1. $a_m$ is used in some maximum-size subset of mutually compatible activities of $S_j$.
2. $S_{m'} = \emptyset$, so that choosing $a_m$ leaves $S_{m'}$ 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:**

- **Input:** $S$ = $\{ a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10} \}$
  - $s_1 = 0$, $f_1 = 2$
  - $s_2 = 1$, $f_2 = 3$
  - $s_3 = 2$, $f_3 = 4$
  - $s_4 = 3$, $f_4 = 4$
  - $s_5 = 3$, $f_5 = 5$
  - $s_6 = 1$, $f_6 = 3$
  - $s_7 = 1$, $f_7 = 3$
  - $s_8 = 4$, $f_8 = 8$
  - $s_9 = 4$, $f_9 = 6$
  - $s_{10} = 5$, $f_{10} = 9$

- **Output:** maximum weight subset of mutually compatible activities.
  - $w_1 = 9$, $w_2 = 3$
  - $w_1 = 9$, $w_2 = 3$

**Application of the greedy algorithm**

- **Input:** $S$ = $\{ a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10} \}$
  - $s_1 = 0$, $f_1 = 2$
  - $s_2 = 1$, $f_2 = 3$
  - $s_3 = 2$, $f_3 = 4$
  - $s_4 = 3$, $f_4 = 4$
  - $s_5 = 3$, $f_5 = 5$
  - $s_6 = 1$, $f_6 = 3$
  - $s_7 = 1$, $f_7 = 3$
  - $s_8 = 4$, $f_8 = 8$
  - $s_9 = 4$, $f_9 = 6$
  - $s_{10} = 5$, $f_{10} = 9$

- **Output:** maximum weight subset of mutually compatible activities.
  - $w_1 = 9$
  - $w_2 = 3$
Discussion

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

- 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: $p(6)=4, p(5)=2, p(4)=2, p(2)=0.$

![Data structure diagram](image)

Binary choice

Notation. $OPT()= $ value of optimal solution to the problem consisting of
job requests $1, 2, \ldots, j$.

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

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

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

Recursive call

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

$Compute-Opt(j)$
- if $j = 0$
  - return 0.
- else
  - return $\max(v[j] + Compute-Opt(p[j]), Compute-Opt(j-1)))$.

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 $f[1] \leq f[2] \leq \ldots \leq f[n]$.
Compute $p[1], p[2], \ldots, p[n]$.

for $j = 1$ to $n$
- $M[j] \leftarrow$ empty.
- $M[0] \leftarrow 0.$

$M-Compute-Opt(j)$
- if $M[j]$ is empty
- 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 $p(j)$: $O(n \log n)$ 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 $\Phi = n$ nonempty entries of $M[.]$.
  - initially $\Phi = 0$, throughout $\Phi \leq n$.
  - (ii) increases $\Phi$ by 1 at most $\Delta$ in recursive calls.
- Overall running time of $M$-COMPUTE-OPT is $O(n \log n)$.

Remark: Our $M$ if jobs are presorted by start and finish times.

Finding a solution

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

$\text{Find-Solution}(j)$
if $j = 0$
  return $\emptyset$;
else if $(v[j] + M[p[j]] > M[j-1])$
  return $\{j\} \cup \text{Find-Solution}(p[j])$
else
  return 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>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$V \cdot M[p[j]]$</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</td>
<td>2</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$V \cdot M[p[j]]$</td>
<td>-</td>
<td>-</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.

Bottom-up

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

$\text{BOTTOM-UP}(n, s_1, ... s_n, f_1, ... f_n, v_1, ... v_m)$

Sort jobs by finish time so that $f_1 \leq f_2 \leq ... \leq f_n$.
Compute $p(1), p(2), ..., p(n)$.

$M[0] \leftarrow 0$
for $j = 1$ TO $n$
  $M[j] \leftarrow \max \{ v[j] + M[p[j]], M[j-1] \}$

Main idea of Dynamic Programming: Solve the subproblems in an order that makes sure when you need an answer, it’s already been computed.
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>3</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>8</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.
Modeling as graphs

Input:
- Directed graph \( G = (V, E) \)
- Weight function \( w : E \rightarrow R \)

Weight of path \( p = (v_0, v_1, ..., v_k) \)
\[ = \sum_{i=1}^{k} w(v_{i-1}, v_i) \]
= sum of edge weights on path \( p \).

Shortest-path weight \( u \) to \( v \):
\[ \delta(u,v) \]
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

\text{DIJKSTRA}(V, E, w, s)
\text{INIT-SINGLE-SOURCE}(V, s)
\begin{align*}
S & \leftarrow \emptyset \\
Q & \leftarrow V \\
\text{while } Q \neq \emptyset & \text{ do} \\
\quad u & \leftarrow \text{EXTRACT-MIN}(Q) \\
\quad S & \leftarrow S \cup \{u\} \\
\text{for each vertex } v \in \text{Adj}[u] & \text{ do} \\
\quad \text{RELAX}(u, v, w) \\
\end{align*}

Example

\[
\begin{array}{c}
\text{Q} \\
\text{S}
\end{array}
\]
Bellman-Ford Algorithm

- Allows negative-weight edges.
- Computes $d[v]$ and $π[v]$ 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 $O(VE)$. 
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.

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