Dynamic Programming (Ch 6)

- **Dynamic Programming Approach**
- **Weighted Interval Scheduling**
- **Knapsack Problem**
- **Shortest Path on Graphs with Negative Edge Length**
- **Sequence Alignment**

The lecture notes/slides are adapted from those associated with the text book by J. Kleinberg and E. Tardos.

Dynamic Programming Approach

- **Dynamic programming: Partition a problem into sub-problems (can be overlapping), find a solution for each sub-problem and keep the solution in a table, and find a solution of the original problem based on the solutions in the table.**

  R. Bellman pioneered the systematic study of dynamic programming in 1950s.

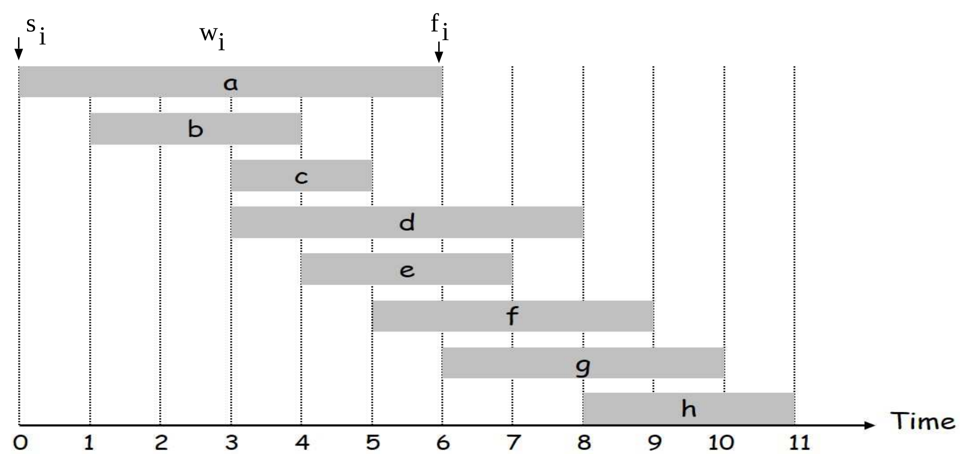
- **Divide and conquer: Partition a problem into independent sub-problems, find a solution for each sub-problem and combine the solution of sub-problems to a solution of the original problem.**

- **Greedy: Start from a partial solution and increment the partial solution step-by-step, in each step, use a local optimum to increment the partial solution.**

Weighted Interval Scheduling Problem

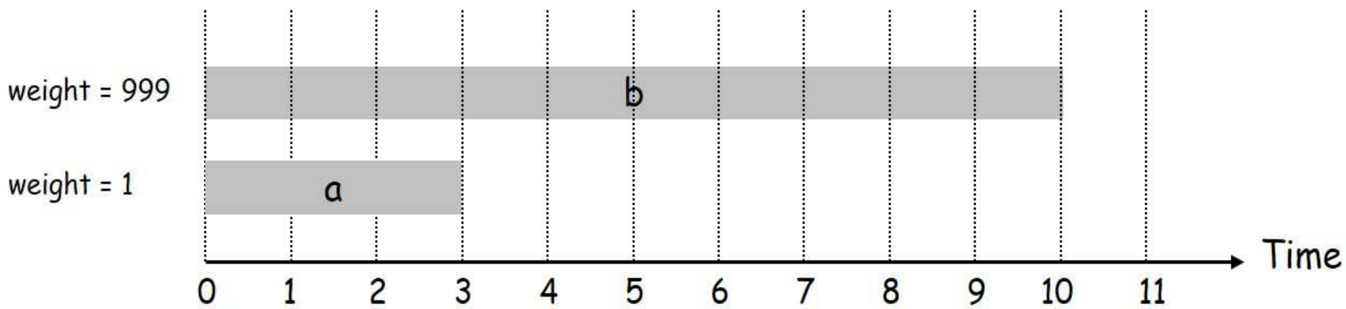
Weighted interval scheduling problem

- **Given a set $S = \{a_1, .., a_n\}$ of proposed jobs that wish to use a resource which can serve one job at a time. Each a_i has a start time s_i and a finish time f_i with $0 \leq s_i < f_i < \infty$, and a weight $w_i > 0$. If selected, a_i takes place in time interval $[s_i, f_i)$.**
- **Jobs a_i and a_j are compatible if $[s_i, f_i) \cap [s_j, f_j) = \emptyset$.**
- **The weight of a subset S' of jobs is $w(S') = \sum_{a_i \in S'} w_i$.**
- **Goal, Find a max-weight subset of mutually compatible jobs from S.**



Greedy algorithm: earliest-finish-time-first

- **In each step, select a job a_k with the earliest finish time.**
- **Intuition, select a_k which leaves resource available for as many other jobs as possible.**
- **The greedy algorithm is correct if $w_i = 1$ for all $a_i \in S$, but fails for weighted interval scheduling problem.**

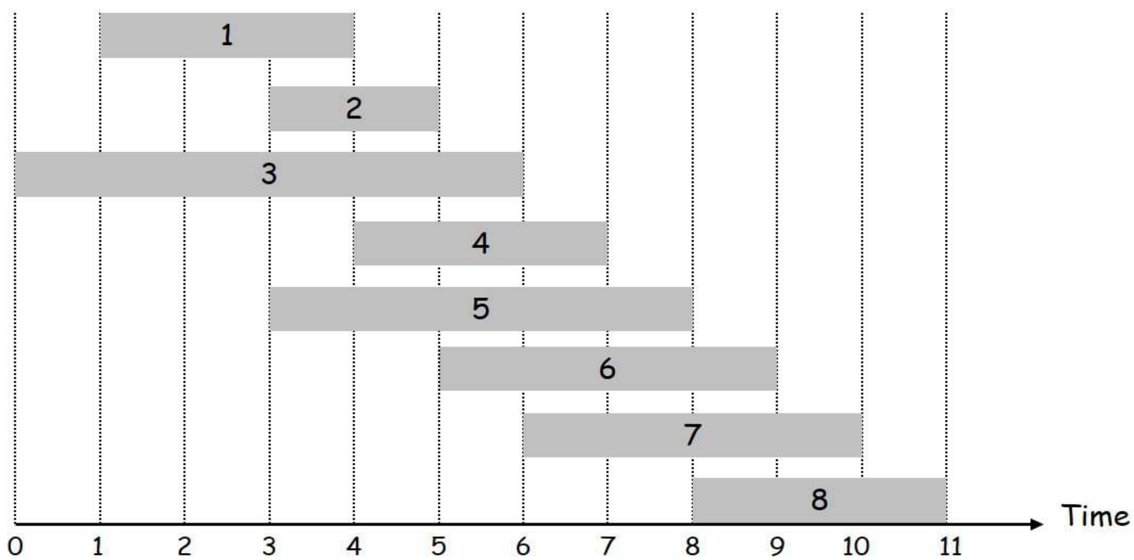


Dynamic programming algorithm

- **Jobs are listed in ascending order** $a_1, .., a_n$ **of finish time:** $f_1 \leq .. \leq f_n$**.**
- **Subproblems: Find subset** S_j **of jobs** $\{a_1, .., a_j\}, 1 \leq j \leq n$**, s.t. jobs in** S_j **are mutually compatible and** $w(S_j)$ **maximized.**

  To find S_j**, two choices for** a_j**,** $a_j \in S_j$ **or** $a_j \notin S_j$**. If** $a_j \in S_j$ **then** $S_j \setminus \{a_j\} = S_i$ **with** $i < j$ **the largest index that job** i **is compatible with job** j **(**$f_i < s_j$**).**
- $p(j)$**: largest index** $i < j$ **s.t. job** i **is compatible with job** j **(**$f_i < s_j$**).**

  Example: $p(8) = 5, p(7) = 3, p(2) = 0$**.**



- **Structure of optimal solution**

  $\mathrm{opt}(j)$**: max-weight of any subset of mutually compatible jobs for subproblem consisting of jobs** $a_1, .., a_j$**.**

  Goal: find $\mathrm{opt}(n)$**.**

  - **Case 1,** $\mathrm{opt}(j)$ **does not select** a_j**.** $\mathrm{opt}(j) = \mathrm{opt}(j-1)$**.**
  - **Case 2,** $\mathrm{opt}(j)$ **selects** a_j**. Jobs in** $\{p(j)+1, .., j-1\}$ **are incompatible with** a_j**. So,** $\mathrm{opt}(j) = w_j + \mathrm{opt}(p(j))$**.**

- **Recursive definition of optimal value (Bellman equation)**

$$\mathrm{opt}(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{\mathrm{opt}(j-1), w_j + \mathrm{opt}(p(j))\} & \text{if } j > 0 \end{cases}$$

Compute optimal solution

- **Weighted-Interval-Scheduling**(n, S) **/* Compute optimal value**

  sort jobs of S **by finish time s.t.** $f_1 \leq f_2 \leq f_n$**;**

  compute $p(1), p(2), .., p(n)$**;** $M[0] = 0$**;**

  for $j = 1$ **to** n **do** $M[j] = \max\{M[j-1], w_j + M[p(j)]\}$**;**

  return M

- **Weighted-Interval-Scheduling-Jobs**(n, S) **/* compute optimal solution**

  sort jobs of S **by finish time s.t.** $f_1 \leq f_2 \leq f_n$**;**

  compute $p(1), p(2), .., p(n)$**;** $M[0] = 0$**;** $A_0 = \emptyset$**;**

  for $j = 1$ **to** n **do**

    if $M[j-1] \geq w_j + M[p(j)]$ **then** $\{M[j] = M[j-1]$**;** $A_j = A_{j-1}\}$

    else $\{M[j] = w_j + M[p(j)]$**;** $A_j = A_{p(j)} \cup \{a_j\}; \}$

  return M **and** A

- **Running time of the algorithms is** $O(n \log n)$**.**

Knapsack problem

- **Given a set $I = \{1, ..., n\}$ of items, each item i has a positive integer value v_i and a positive integer weight w_i, and a knapsack with a positive integer capacity $W \geq w_i$ for $i \in I$, find a subset $S \subseteq I$ s.t. $\sum_{i \in S} w_i \leq W$ and $\sum_{i \in S} v_i$ is maximized.**
- **Example, $S = \{3, 4\}$ has value 40 and total weight 11.**

  Greedy: repeatedly add items with maximum v_i/w_i ratio gives $S = \{5, 2, 1\}$ with value 35 and total weight 10, not optimal.

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Dynamic programming for Knapsack

- **Idea: find optimal solution S from $\{1,..,i\}, 1 \leq i \leq n$, with weight limit W.**

  If $i \notin S$ then S is an optimal solution from $\{1,..,i-1\}$ with weight limit W.

  If $i \in S$ then S is an optimal solution from $\{1,..,i-1\}$ with weight limit $W - w_i$.

- **Subproblems: Select items from $\{1,..,i\}, 1 \leq i \leq n$, with total value maximized and weight limit $w, 0 \leq w \leq W$.**

- **Structure of an optimal solution.**

  $\mathrm{opt}(i, w)$**: max-value of items from $\{1,..,i\}$ with weight limit w.**
  Goal: find $\mathrm{opt}(n, W)$.

  - **Case 1: $\mathrm{opt}(i, w)$ does not have i but optimal solution of $\{1,..,i-1\}$ with weight limit w.**
  - **Case 2: $\mathrm{opt}(i, w)$ has i and optimal solution of $\{1,..,i-1\}$ with weight limit $w - w_i$.**

- **Bellman equation**

$$\mathrm{opt}(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ \mathrm{opt}(i-1, w) & \text{if } w_i > w \\ \max\{\mathrm{opt}(i-1, w), v_i + \mathrm{opt}(i-1, w - w_i)\} & \text{otherwise} \end{cases}$$

Knapsack(n, W) /* Compute optimal solution

Input: $\{v_1, .., v_n\}$, $\{w_1, .., w_n\}$ **and** W**.**

Output: $S \subseteq \{1, .., n\}$ **s.t.** $\sum_{i \in S} w_i \leq W$ **and** $\sum_{i \in S} v_i$ **is maximized.**

Let M **be an** $(n+1) \times (W+1)$ **array indexed from** 0 **to** n **and from** 0 **to** W**;**

$M[0, w] = 0$ **for each** $w = 0, 1, .., W$**;**

for $i = 1$ **to** n **do**

  for $w = 0$ **to** W **do**

    if $w_i > w$ **then** $M[i, w] = M[i-1, w]$

    else $M[i, w] = \max\{M[i-1, w], M[i-1, w-w_i] + v_i\}$**;**

return $M[n, W]$**;**

Knapsack algorithm solves Knapsack problem in $O(nW)$ **time and** $O(nW)$ **space.**

This is not a $\mathrm{Poly}(n)$ **time algorithm when** W **is not upper bounded by** $\mathrm{Poly}(n)$

(W **can be as large as** 2^{cn} **for** $c > 0$**).**

W + 1

n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

OPT: { 4, 3 }
value = 22 + 18 = 40

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

More efficient dynamic programming for Knapsack problem [Lawler 1979]

- $S \subseteq \{1, ..., j\}$ **is a solution from the first** j **elements if** $\sum_{i \in S} w_i \leq W$.

  A solution S **dominates another solution** S' **if** $\sum_{i \in S} w_i \leq \sum_{i \in S'} w_i$ **and** $\sum_{i \in S} v_i \geq \sum_{i \in S'} v_i$.

  A set $\mathcal{S}$ **of solutions is non-dominating if any solution of** $\mathcal{S}$ **does not dominate any other solution of** $\mathcal{S}$.

- **Examples,**

```
item i    value v_i    weight w_i    W=11
   1          3             3
   2          6             4
   3         18             5
```

  For $j = 1$, $\mathcal{S} = \{S = \emptyset, S' = \{1\}\}$ **is non-dominating since**
  $\sum_{i \in S} v_i = 0 < \sum_{i \in S'} v_i = v_1 = 3$ **and**
  $\sum_{i \in S} w_i = 0 < \sum_{i \in S'} w_i = w_1 = 3$.
  For $j = 3$, $S = \{3\}$ **dominates** $S' = \{1, 2\}$ **since**
  $\sum_{i \in S} v_i = v_3 = 18 > \sum_{i \in S'} v_i = v_1 + v_2 = 9$ **and**
  $\sum_{i \in S} w_i = w_3 = 5 < \sum_{i \in S'} w_i = w_1 + w_2 = 7$.

- **Structure of an optimal solution**

  Let $A(j)$ be set of value and weight pairs (t, w) of non-dominating solutions S for items $\{1, 2, .., j\}$, $t = \sum_{i \in S} v_i$ and $w = \sum_{i \in S} w_i$.
  Goal: find $A(n)$ and a solution in $A(n)$ with maximum value t.

  – $A(1) = \{(0, 0), (v_1, w_1)\}$.

  – **For $1 < j \leq n$,**
  $A(j) = (A(j-1) \cup \{(t + v_j, w + w_j) | (t, w) \in A(j-1)\}) \setminus \tilde{A}$,
  where $\tilde{A}$ is a minimal subset of
  $A(j-1) \cup \{(t + v_j, w + w_j) | (t, w) \in A(j-1)\}$ **s.t. removing $\tilde{A}$ makes $A(j)$ a set of pairs of non-dominating solutions.**

- **Bellman equation**

$$A(j) = \begin{cases} \{(0, 0), (v_1, w_1)\} & \text{if } j = 1 \\ (A(j-1) \cup \{(t + v_j, w + w_j) | (t, w) \in A(j-1)\}) \setminus \tilde{A} & \text{if } j > 1 \end{cases}$$

Algorithm

- **For** $j = 1$**, non-dominating** $\mathcal{S}$ **has two solutions** $S = \emptyset$ **and** $S = \{1\}$**. We keep** $(0, 0)$ **for** $S = \emptyset$ **and** (v_1, w_1) **for** $S = \{1\}$ **in table** $A(1)$**.**
- **For** $j > 1$**, non-dominating** $\mathcal{S}$ **of solutions from** $\{1, .., j - 1\}$ **have been found and for each solution** S**, a pair** (t, w)**,** $t = \sum_{i \in S} v_i$ **and** $w = \sum_{i \in S} w_i$**, is kept in table** $A(j - 1)$**.**
- **Find non-dominating** $\mathcal{S}$ **of solutions from** $\{1, .., j\}$ **by checking the pairs** (t, w) **in** $A(j - 1)$ **and values of** v_j **and** w_j**.**

Examples,

item i	value v_i	weight w_i	W=11
1	3	3	
2	6	4	
3	18	5	

$j = 1, S \subseteq \{1\}$: $A(1) = \{(0,0),(3,3)\}$

$j = 2, S \subseteq \{1,2\}$: $A(2) = \{(0,0),(3,3),(6,4),(9,7)\}$

$j = 3, S \subseteq \{1,2,3\}$:

All subsets $(0,0),(3,3),(6,4),(9,7),(18,5),(21,8),(24,9),(27,12)$

$A(3) = \{(0,0),(3,3),(6,4),(18,5),(21,8),(24,9)\}$

$S = (18,5)$ **dominates** $S' = (9,7)$ **because**

$\sum_{i \in S} v_i = 18 > \sum_{i \in S'} v_i = 9$ **and** $\sum_{i \in S} w_i = 5 < \sum_{i \in S'} w_i = 7$.

$S = (27,12)$ **is not a solution because** $\sum_{i \in S} w_i = 12 > W = 11$.

Knapsack1(n, W)

Input: $\{v_1, .., v_n\}$, $\{w_1, .., w_n\}$ **and** W.

Output: $S \subseteq \{1, .., n\}$ **s.t.** $\sum_{i \in S} w_i \leq W$ **and** $\sum_{i \in S} v_i$ **is maximized.**

$A(1) = \{(0, 0), (v_1, w_1)\}$;

for $j = 2$ **to** n **do**

$A(j) = A(j - 1)$;

for each $(t, w) \in A(j - 1)$ **do**

if $w + w_j \leq W$ **then** $A(j) = A(j) \cup \{(t + v_j, w + w_j)\}$;

Remove dominated pairs from $A(j)$;

Return a solution in $A(n)$ **with the maximum** t;

Example: Solution (t, w) dominates solution (t', w') if $t \geq t'$ and $w \leq w'$. $A(j)$ keeps all non-dominating solutions (t, w) for $\{1, 2, .., j\}$.

item	value	weight	W=11
1	3	3	
2	6	4	
3	18	5	
4	22	6	
5	28	7	

```
{1}     A(1)={(0,0),(3,3)}
{1,2}   A={(0,0),(3,3),
           (0+6,0+4),(3+6,3+4)}
        A(2)={(0,0),(3,3),(6,4),(9,7)}
{1,2,3} A={(0,0),(3,3),(6,4),(9,7),
           (0+18,0+5),(3+18,3+5),(6+18,4+5),(9+18,7+5)}
        A(3)={(0,0),(3,3),(6,4),(18,5),(21,8),(24,9)}
```

Theorem. [Lawler 1979] Knapsack1 algorithm solves the knapsack problem in $O(n \min\{T, W\})$ time and $O(\min\{T, W\})$ space, where $T = \sum_{i \in I} v_i$.

Proof. **Statement: for any solution $S \subseteq \{1, ..., j\}$, there is a solution $(t, w) \in A(j)$ s.t. (t, w) dominates S ($\sum_{i \in S} v_i \leq t$ and $\sum_{i \in S} w_i \geq w$).**

It is true for $j = 1$. Assume it is true for $j - 1 \geq 1$, we prove it for j.

For solution $S \subseteq \{1, ..., j\}$, if $j \notin S$ then $S \subseteq \{1, .., j - 1\}$ and by induction hypothesis, there is a solution $(t', w') \in A(j - 1)$ s.t. (s', t') dominates S. Since $A(j) := A(j - 1)$ and only dominated solutions are removed from $A(j)$, there is a solution $(t, w) \in A(j)$ s.t. (t, w) dominates S.

Statement: for any solution $S \subseteq \{1, ..., j\}$, there is a solution $(t, w) \in A(j)$ s.t. (t, w) dominates S ($\sum_{i \in S} v_i \leq t$ and $\sum_{i \in S} w_i \geq w$.)

Assume that $j \in S$. Then $S' = S \setminus \{j\}$ is a solution, $S' \subseteq \{1, .., j-1\}$, and by induction hypothesis, there is a solution $(t', w') \in A(j-1)$ s.t. (t', w') dominates S' ($\sum_{i \in S'} v_i \leq t'$ and $\sum_{i \in S'} w_i \geq w'$).

Then $\sum_{i \in S} v_i = (\sum_{i \in S'} v_i) + v_j \leq t' + v_j$ and $\sum_{i \in S} w_i = (\sum_{i \in S'} w_i) + w_j \geq w' + w_j$.

Let $t = t' + v_j$ and $w = w' + w_j$. Then (t, w) is a solution added to $A(j)$ by the algorithm and dominates S, the statement holds for j.

By the statement, the algorithm solves the knapsack problem.

By non-dominating property, solutions in each $A(j)$ can be listed as $(t_1, w_1), ..., (t_k, w_k)$ s.t. $t_1 < t_2 < ... < t_k$ and $w_1 < w_2 < ... < w_k$.

Let $T = \sum_{i \in I} v_i$. Since each of t_i and w_i is a positive integer, $t_i \leq T$ and $w_i \leq W$, there are at most $\min\{T+1, W+1\}$ solutions in $A(j)$. This gives $O(n \min\{T, W\})$ running time and $O(\min\{T, W\})$ space of the algorithm. $\square$

- **The running time may not be $\mathrm{Poly}(n)$, the value of $\min\{T, W\}$ can be 2^{cn}, $c > 0$. If $\min\{T, W\}$ is $\mathrm{Poly}(n)$, the algorithm has a polynomial running time. In this case, the input instance in unary form has $\mathrm{Poly}(n)$ size.**

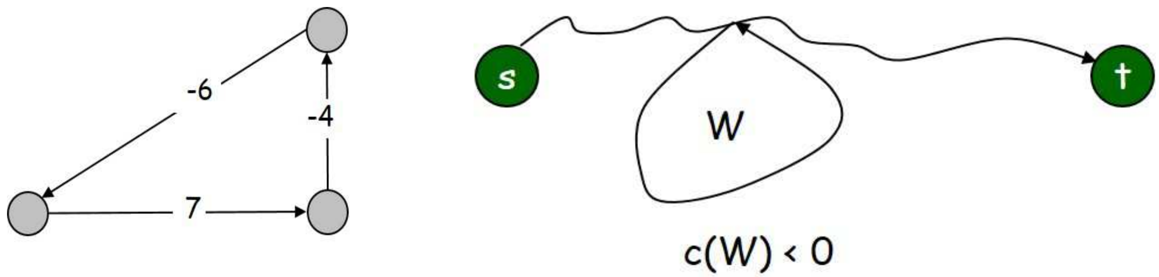
- **An algorithm for a problem is a *pseudo-polynomial time algorithm* if its running time is polynomial in the size of the input when the numeric part of the input is encoded in unary.**

- **Examples of unary codes**

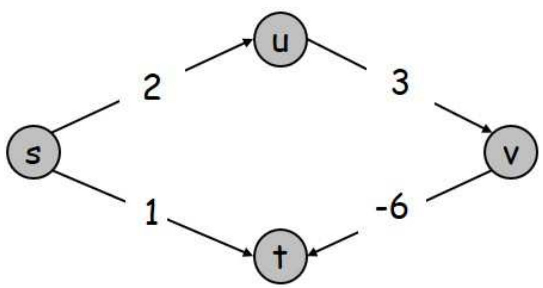
  111 **for 3,** W 1**'s for integer of value** W**.**

Shortest Path in Graphs with Negative Edge Length

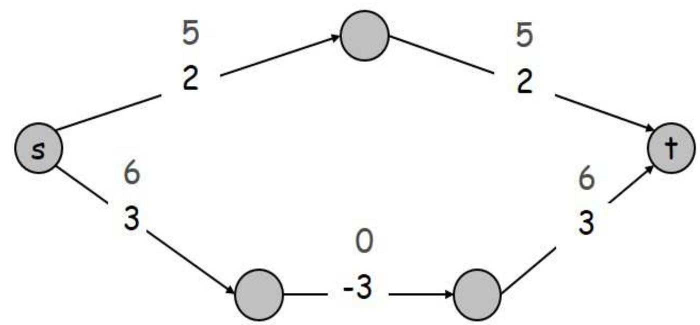
- **Given a weighted digraph G with each arc assigned an arbitrary real edge length, find a shortest path from node s to a node t (or to all nodes other than s).**
- **Hurdles**
  - **Negative cycle, a cycle has a negative length. If G has a negative cycle, then a shortest path from s to t may not exist.**
  - **Dijkstra (greedy) algorithm may not find a shortest path from s to t even one exists.**
  - **Change arc lengths to non-negative by adding a uniform length to all arcs may not work.**



Negative cycles



Counter example for Dijkstra's algorithm



Counter example for adding uniform edge length

Dynamic programming for shortest path problem

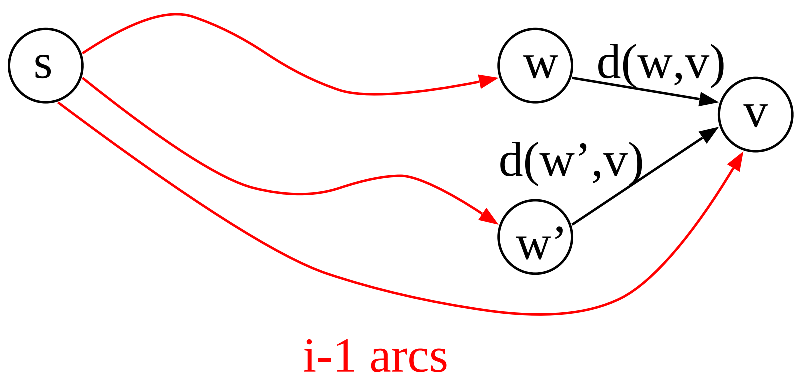
- **If G has no negative cycle, then there is a shortest path of at most $n-1$ arcs from s to every other node t.**
- **Dynamic programming, find shortest path with increasing number of arcs.**
- **Structure of an optimal solution.**

  $\mathrm{opt}(i,v)$ **: the length of shortest path $s \to v$ with at most i arcs.**
  Goal: find $\mathrm{opt}(n-1,v)$ for every $v \in V$.

  - **Case 1: shortest path $s \to v$ has at most $i-1$ arcs,**
    $\mathrm{opt}(i,v) = \mathrm{opt}(i-1,v)$.
  - **Case 2: shortest path $s \to v$ has i arcs,**
    $\mathrm{opt}(i,v) = \min_{(w,v)\in E}\{\mathrm{opt}(i-1,w) + d(w,v)\}$.

- **Bellman equation**

$$\mathrm{opt}(i,v) = \begin{cases} 0 & \textbf{if } i=0 \textbf{ and } v=s \\ \infty & \textbf{if } i=0 \textbf{ and } v\neq s \\ \min\{\mathrm{opt}(i-1,v), & \\ \qquad \min_{(w,v)\in E}\{\mathrm{opt}(i-1,w)+d(w,v)\}\} & \textbf{if } i>0 \end{cases}$$



Brute force implementation of Bellman-Ford algorithm [Bellman-Ford 1956 1958]

ShortestPath1(G, s) /* Compute shortest distances

Input: Weighted digraph G with no negative cycle and source s.

Output: Shortest distance from s to every other node.

Let M be an $n \times n$ array indexed from 0 to $n-1$, $n = |V(G)|$;

$M[0, s] = 0$ **and for every $v \in V(G) \setminus \{s\}$, $M[0, v] = \infty$;**

for $i = 1$ to $n-1$ do

for $v \in V(G)$ do

$M[i, v] = \min\{M[i-1, v], \min_{w \in V(G)}\{M[i-1, w] + d(w, v)\}\}$;

Return $M[n-1, v]$ for $v \neq s$;

Time for $M[i, v] = \min\{\cdots\}$ is $O(n)$, time of the inner for loop is $O(n^2)$, total time is $O(n^3)$.

Theorem. [Bellman-Ford 1956 1958] ShortestPath1 algorithm computes the minimum distance from s to all other nodes in a weighted digraph with no negative cycle in $O(n^3)$ time and $\Theta(n^2)$ space.

Proof. Since G does not have negative cycle, there is a shortest path $s \rightarrow v$ of at most $n-1$ arcs for each v reachable from s. We prove by induction on i that if there is a shortest path $s \rightarrow v$ of at most i arcs, the algorithm computes the length of the path $s \rightarrow v$.

For $i = 0$, the statement is true. Assume the statement is true for $i - 1 \geq 0$ and we prove it for i. Assume there is a shortest path $s \rightarrow v$ of i arcs and let (w', v) be the last arc in $s \rightarrow v$. Then there is a shortest path $s \rightarrow w'$ of $i-1$ arcs. Since G does not have negative cycle and $s \rightarrow v$ is a shortest path, for any path $w' \rightarrow v$, $d(w' \rightarrow v) \geq d(w', v)$. By the algorithm the statement holds.

Running time and space, trivial. □

More efficient implementation of Bellman-Ford algorithm

ShortestPath2(G, s**) /* Compute shortest distances and paths**

Input: Weighted digraph G **with no negative cycle and source** s**.**

Output: Shortest distance and path from s **to every other node.**

    Let M **and** P **be** $1 \times n$ **arrays indexed from** 0 **to** $n-1$**,** $n = |V(G)|$**;**

    $M[s] = 0$ **and for every** $v \in V(G) \setminus \{s\}$**,** $M[v] = \infty$**;**

    for every $v \in V(G)$**,** $P[v] = null$**;**

    for $i = 1$ **to** $n-1$ **do**

        for $v \in V(G)$ **do**

            for each arc $(w, v) \in E(G)$ **do**

                if $M[v] > M[w] + d(w, v)$ **then**

                    $M[v] = M[w] + d(w, v)$**;** $P[v] = w$**;**

    Return $M[v]$ **and** $P[v]$ **for** $v \neq s$**;**

Time for computing $M[v]$ **is** $O(\text{indeg}(v))$**, time if the inner for loop is** $\sum_{v \in G} O(\text{indeg}(v)) = O(m)$**, total time is** $O(nm)$**.**

ShortestPath2 algorithm computes the shortest distance and path from s to all other nodes in a weighted digraph with no negative cycle in $O(mn)$ time and $\Theta(m)$ space.

Proof. The proof of soundness of the algorithm is similar to that for ShortestPath1 algorithm.

In each iteration of the for loop from $i = 1$ to $n - 1$, each node v is checked $\mathrm{indeg}(v)$ times. Thus the running time of the algorithm is

$$O(n \sum_{v \in V(G)} \mathrm{indeg}(v)) = O(mn).$$

Space, trivial. $\square$

Negative cycle

- **Let $H(V, E)$ be the weighted digraph with $V(H) = V(G)$ and $E(H) = \{(w, v)|v \in V(H)$ and $w = P[v]$ computed in ShortestPath2$\}$. If H has a cycle C, then C is a negative cycle.**

Proof. For any $w = P[v]$, $M[v] > M[w] + d(w, v)$. Let $v_1, .., v_k$ be the nodes in cycle C and (v_k, v_1) be the last arc added to C. Then for $i = 2, .., k$

$$\begin{aligned} M[v_i] &> M[v_{i-1}] + d(v_{i-1}, v_i) \text{ and} \\ M[v_1] &> M[v_k] + d(v_k, v_1). \end{aligned}$$

Sum the inequalities up

$$d(v_k, v_1) + \sum_{i=2}^{k} d(v_{i-1}, v_i) < 0.$$

□

- **For G with no negative cycle, H is a tree, called shortest path tree.**

Find negative cycles

- **Questions**

  How to decide if the input digraph has a negative cycle?

  How to find one?

- **Observation: If a negative cycle C can be reached from a vertex v in G, then C can be found.**

- **Given G, let $A(G)$ be the augmented graph obtained by adding a new node x and a new arc (x, v) for every $v \in E(G)$. G has a negative cycle iff $A(G)$ has a negative cycle.**

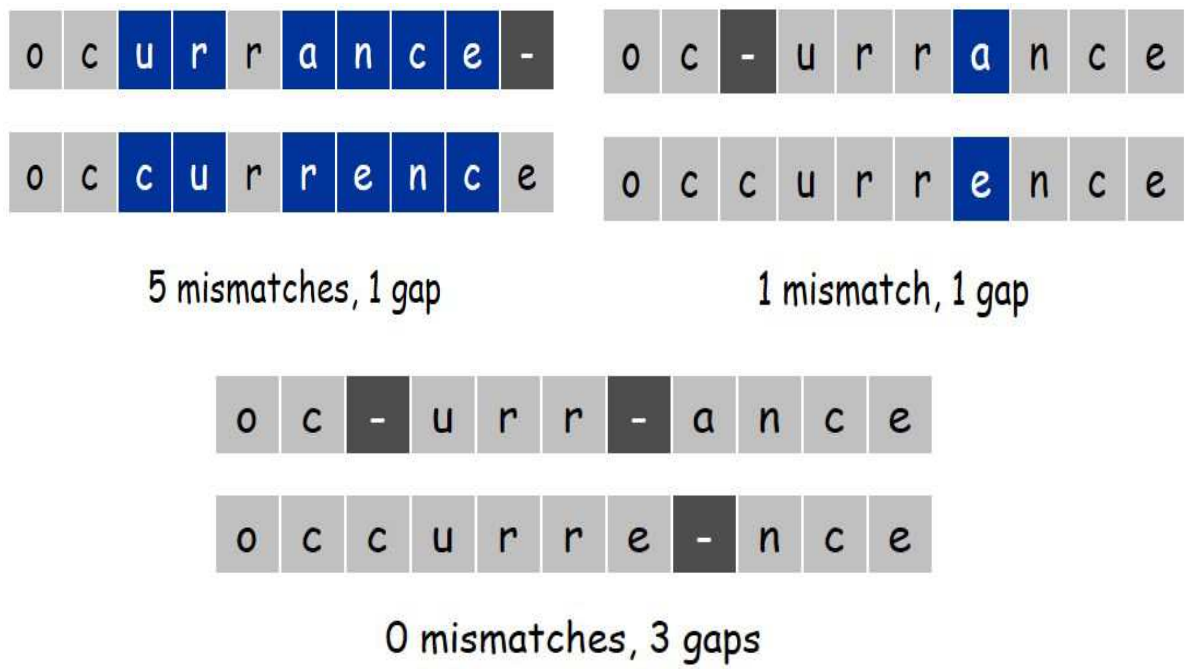
  - **If G does not have a negative cycle, then for every $v \in V(G)$, $\text{opt}(i, v) = \text{opt}(n-1, v)$ for $i \geq n$.**
  - **If $\text{opt}(n, v) \neq \text{opt}(n-1, v)$ for some $v \in V(G)$, then G has a negative cycle and the path from x to v found by the algorithm has a negative cycle.**

Sequence Alignment

- **Given two strings of characters, how similar they are?**

  Example, ocurrance and occurrence.

- **Problem, minimize the number of gaps and mismatches.**



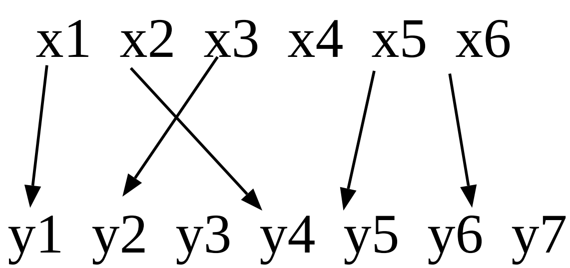
Alignments

Let $X = x_1 x_2 .. x_m$ **and** $Y = y_1 y_2 .. y_n$**.**

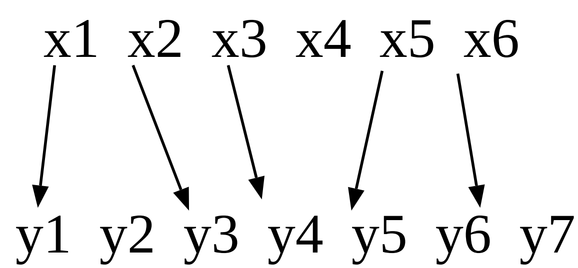
A matching between X **and** Y **is a set of pairs** (x_i, y_j) **s.t. each element of** X **and** Y **appears in at most one pair.**

Two pairs (x_i, y_j) **and** $(x_{i'}, y_{j'})$ **are crossing if** $i < i'$ **and** $j > j'$**.**

A matching is an alignment if there is no crossing pairs



A matching



An alignment

- **Let M be an alignment between X and Y.**

  Each position of X or Y that is not matched in M is called a gap.

  Each pair $(x_i, y_j) \in M$ s.t. $x_i \neq y_j$ is called a mismatch.

  Cost of M

  - **For each gap, there is a gap penalty $\delta > 0$.**
  - **For each pair (p, q), there is a mismatch cost $\alpha_{pq} > 0$, usually $\alpha_{pp} = 0$.**
  - **The cost of M is the sum of the gap penalties and mismatch costs in M.**

- **Sequence alignment problem, given X and Y, find an alignment between X and Y of minimum cost.**

- **Example,**

  ```
  X=  m  e  a  n
  Y=  n  a  m  e
  ```

  $\delta = 2$, $\alpha_{m,n} = \alpha_{a,e} = 1$ **(mismatch between two vowels or two consonants)**

  $\alpha_{m,a} = \alpha_{m,e} = \alpha_{n,a} = \alpha_{n,e} = 3$ **(mismatch between a vowel and a consonant)**

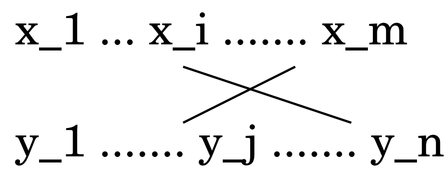
  $\{(m, n), (a, a), (n, m)\}$ **is an alignment between X and Y of minimum cost 6**

  ($\alpha_{m,n} = 1$, one gap $\delta = 2$ in X, $\alpha_{n,m} = 1$, one gap $\delta = 2$ in Y).

Alignment properties

- **For any alignment M of $X = x_1..x_m$ and $Y = y_1..y_n$, if $(x_m, y_n) \notin M$ then either x_m or y_n is not matched in M.**

  Proof. Assume $(x_m, y_j), (x_i, y_n) \in M$ with $i < m$ and $j < n$. Then this is a crossing pair, contradicts with M an alignment. $\square$



- **For an optimal alignment M of X and Y, at least one of the following is true:**
  - $(x_m, y_n) \in M$,
  - x_m **is not matched in** M**, or**
  - y_n **is not matched in** M**.**

Dynamic programming approach

- **Structure of an optimal solution.**

  $\mathrm{opt}(i,j)$**: minimum cost of an alignment** M **between** $x_1..x_i$ **and** $y_1..y_j$**.**

  Goal: find $\mathrm{opt}(m,n)$**.**

  - **Case 1:** $\mathrm{opt}(i,j)$ **matches** $x_i - y_j$**,** $\mathrm{opt}(i,j) = \mathrm{opt}(i-1,j-1) + \alpha_{x_i y_j}$**.**
  - **Case 2a:** $\mathrm{opt}(i,j)$ **leaves** x_i **unmatched,** $\mathrm{opt}(i,j) = \mathrm{opt}(i-1,j) + \delta$**.**
  - **Case 2b:** $\mathrm{opt}(i,j)$ **leaves** y_j **unmatched,** $\mathrm{opt}(i,j) = \mathrm{opt}(i,j-1) + \delta$**.**

- **Bellman equation**

$$\mathrm{opt}(i,j) = \begin{cases} j\delta & \textbf{if } i = 0 \\ i\delta & \textbf{if } j = 0 \\ \min\{\mathrm{opt}(i-1,j-1) + \alpha_{x_i y_j}, & \\ \qquad \mathrm{opt}(i-1,j) + \delta, \mathrm{opt}(i,j-1) + \delta\} & \textbf{otherwise} \end{cases}$$

Alignment(X, Y) /* Compute optimal solution

Input: Strings $X = x_1..x_m$ and $Y = y_1..y_n$.

Output: A minimum cost alignment M between X and Y.

Array $M[0..m, 0..n]$;

For each i, $M[i, 0] = i\delta$;

For each j, $M[0, j] = j\delta$;

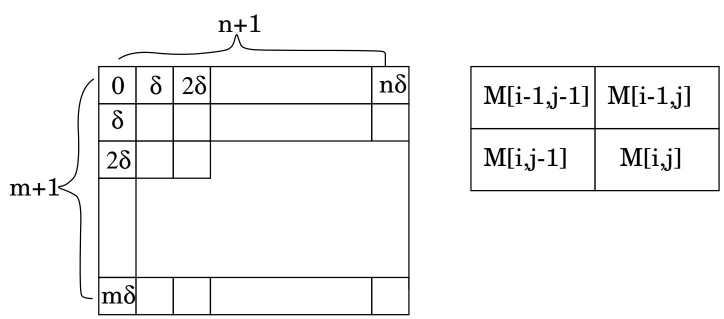
for $i = 1$ **to** m **do**

for $j = 1$ **to** n **do**

$$M[i, j] = \min\{M[i-1, j-1] + \alpha_{x_i y_j}, M[i-1, j] + \delta, M[i, j-1] + \delta\}$$

Return M;

A minimum alignment is computed in $O(mn)$ time and $O(mn)$ space.



```
X=  m  e  a  n
Y=  n  a  m  e
```

$\delta = 2$, $\alpha_{m,n} = \alpha_{a,e} = 1$ **(mismatch between two vowels or two consonants)**

$\alpha_{m,a} = \alpha_{m,e} = \alpha_{n,a} = \alpha_{n,e} = 3$ **(mismatch between a vowel and a consonant)**

		n	a	m	e
	0	2	4	6	8
m	2	1	3	4	6
e	4	3	2	4	4
a	6	5	3	5	5
n	8	6	5	4	6

$$M[1,1] = \min\{M[0,0]+\alpha_{mn}, M[0,1]+\delta, M[1,0]+\delta\} = \min\{0+1, 2+2, 2+2\} = 1$$

$$M[1,2] = \min\{M[0,1]+\alpha_{ma}, M[0,2]+\delta, M[1,1]+\delta\} = \min\{2+3, 4+2, 1+2\} = 3$$

$$M[1,3] = \min\{M[0,2]+\alpha_{mm}, M[0,3]+\delta, M[1,2]+\delta\} = \min\{4+0, 6+2, 3+2\} = 4$$

$$M[1,4] = \min\{M[0,3]+\alpha_{me}, M[0,4]+\delta, M[1,3]+\delta\} = \min\{6+3, 8+2, 4+2\} = 6$$

Save-Space-Alignment algorithm computes the cost of an optimal alignment between X and Y in $O(mn)$ time and $O(m+n)$ space.

Save-Space-Alignment(X, Y)

Input: Strings $X = x_1..x_m$ and $Y = y_1..y_n$.

Output: Cost of a minimum alignment between X and Y.

Array $B[0..m, 0..1]$;

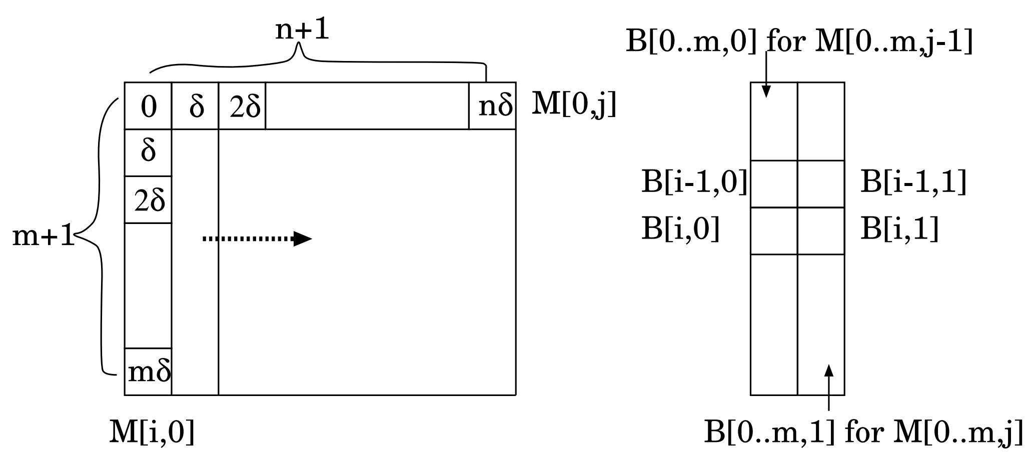
For each i, $B[i, 0] = i\delta$;

for $j = 1$ **to** n **do** $B[0, 1] = j\delta$;

for $i = 1$ **to** m **do**

$B[i, 1] = \min\{B[i-1, 0] + \alpha_{x_i y_j}, B[i-1, 1] + \delta, B[i, 0] + \delta\}$

$B[0..m, 0] = B[0..m, 1]$;



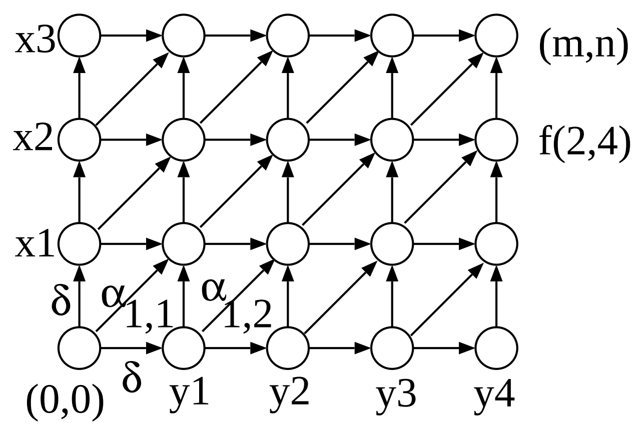
More work is needed to find an optimal alignment in $O(m+n)$ space, graph based approach [Hirschberg 1975]

- **For $X = x_1, .., x_m$ and $Y = y_1, .., y_n$, let G_{XY} be a weighted digraph with $V = \{v_{i,j} | 0 \leq i \leq m, 0 \leq j \leq n\}$ and $E = \{(v_{i,j}, v_{i+1,j}), (v_{i,j}, v_{i,j+1}), (v_{i,j}, v_{i+1,j+1})\}$,**

  each arc of $\{(v_{i,j}, v_{i+1,j}), (v_{i,j}, v_{i,j+1})\}$ is assigned a cost δ and

  each arc of $\{(v_{i,j}, v_{i+1,j+1})\}$ is assigned a cost $\alpha_{x_{i+1}y_{j+1}}$.

- **Let $f(i,j)$ be the distance of a shortest path from $v_{0,0}$ to $v_{i,j}$. Then $f(i,j) = \mathrm{opt}(i,j)$ for every pair of i, j (proof comes shortly).**

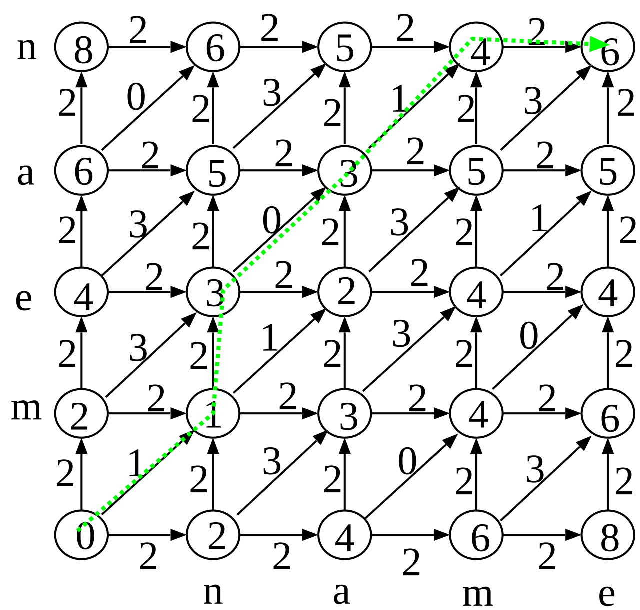


Examples

```
X=  m  e  a  n
Y=  n  a  m  e
```

$\delta = 2$, $\alpha_{m,n} = \alpha_{a,e} = 1$ **(mismatch between two vowels or two consonants)**

$\alpha_{m,a} = \alpha_{m,e} = \alpha_{n,a} = \alpha_{n,e} = 3$ **(mismatch between a vowel and a consonant)**



Soundness of graph based approach

Let $f(i, j)$ be the distance of a shortest path from $v_{0,0}$ to $v_{i,j}$ in G_{XY}. Then $f(i, j) = \mathrm{opt}(i, j)$ for every pair of i, j.

Proof. Prove the statement by induction on $i + j$. For $i + j = 0$,
$f(0, 0) = \mathrm{opt}(0, 0) = 0$,

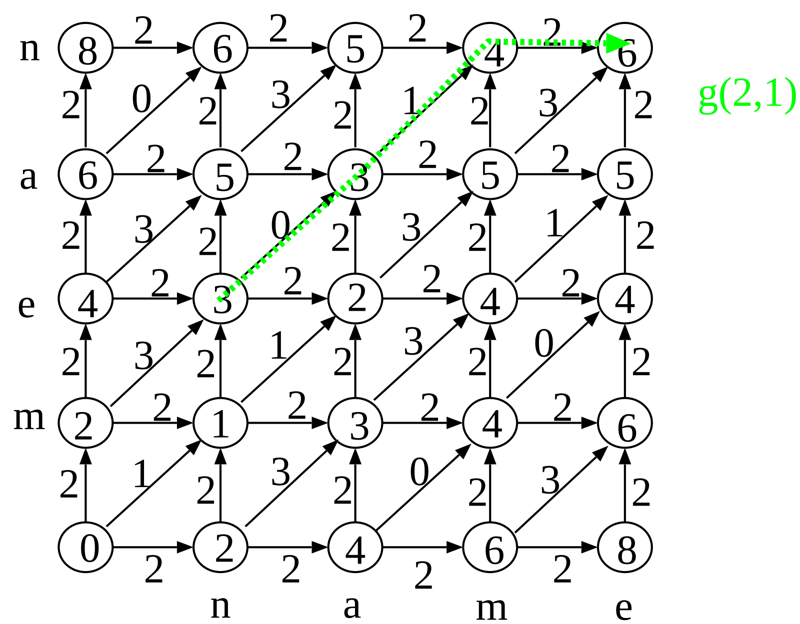
Induction hypothesis, statement is true for all pairs $i' + j' < i + j$.

Induction step, the last arc in the shortest path to $v_{i,j}$ is one of $(v_{i-1,j}, v_{i,j})$, $(v_{i,j-1}, v_{i,j})$, $(v_{i-1,j-1}, v_{i,j})$. Therefore,

$$\begin{aligned} f(i, j) &= \min\{f(i-1, j) + \delta, f(i, j-1) + \delta, f(i-1, j-1) + \alpha_{x_i y_j}\} \\ &= \min\{\mathrm{opt}(i-1, j) + \delta, \mathrm{opt}(i, j-1) + \delta, \mathrm{opt}(i-1, j-1) + \alpha_{x_i y_j}\} \end{aligned}$$

$\square$

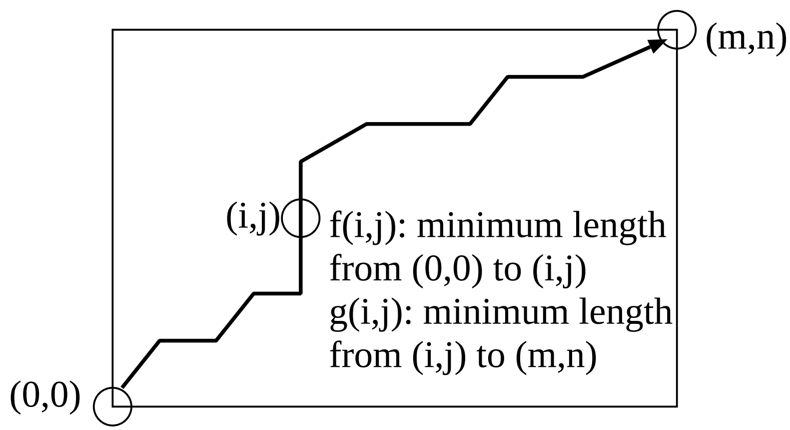
- **Backward search**
  - $g(i, j)$, **length of a shortest path from** $v_{i,j}$ **to** $v_{m,n}$ **in** G_{XY}.
  - $g(i, j) = \min\{g(i+1, j)+\delta, g(i, j+1)+\delta, g(i+1, j+1)+\alpha_{x_{i+1}y_{j+1}}\}$.



Compute optimal alignment in $O(m+n)$ space

- **If a shortest path P from $v_{0,0}$ to $v_{m,n}$ contains node $v_{i,j}$, then the length of P is $f(i,j)+g(i,j)$.**

  Proof. P consists of the path from $v_{0,0}$ to $v_{i,j}$, which has length $f(i,j)$, and the path from $v_{i,j}$ to $v_{m,n}$, which has length $g(i,j)$. $\square$

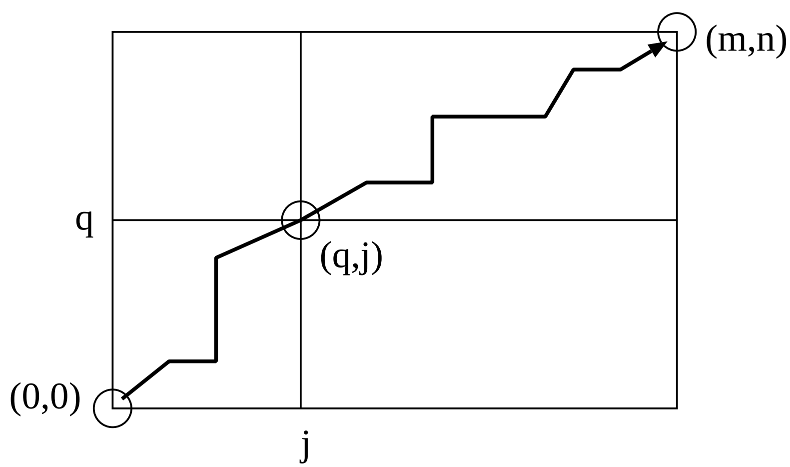


- **For any j with $0 \le j \le n$, let q be an index that minimizes $f(q,j) + g(q,j)$. Then there is a shortest path P from $v_{0,0}$ to $v_{m,n}$ that contains $v_{q,j}$.**

  Proof. For any j, any path P from $v_{0,0}$ to $v_{m,n}$ must have a node $v_{p,j}$, and the length of P is

  $$f(p,j) + g(p,j) \ge \min_{0 \le q \le m} \{f(q,j) + g(q,j)\}.$$

  Therefore, $f(q,j) + q(q,j)$ is the length of a shortest path from $v_{0,0}$ to $v_{m,n}$ and there is such a path containing $v_{q,j}$. $\square$

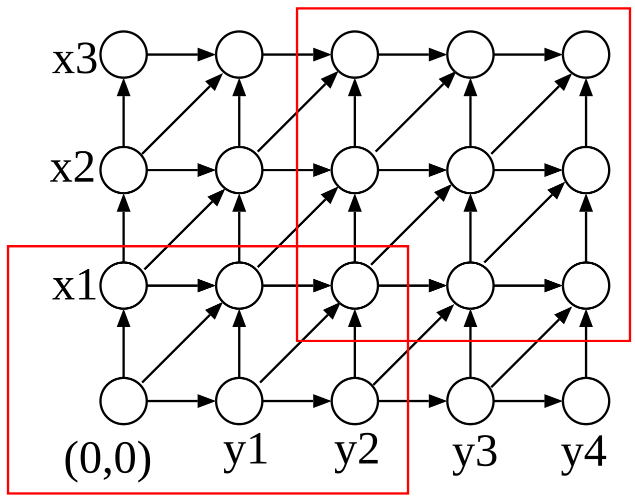


- **Find q which minimizes $f(q, j) + g(q, j)$ for each j.**
- **By divide-and-conquer approach**

  Divide, choose $j = n/2$ to partition the problem into subproblems;

  Conquer, find the shortest paths in subproblems;

  Combine, combine the shortest paths for subproblems to get a solution.



Divide-and-Conquer-Alignment($X[1..m], Y[1..n]$**)**

Input: Strings $X = x_1..x_m$ **and** $Y = y_1..y_n$**.**

Output: A minimum cost alignment M **between** X **and** Y**.**

$m = |X|$**;** $n = |Y|$**;**

if $m \leq 2$ **and** $n \leq 2$ **then Alignment(**X, Y**);**

$\text{opt} = \infty$**;** $q = 1$**;**

Space-Saving-Alignment($X[1..m], Y[1..n/2]$**);**

Space-Saving-Alignment-backward($X[1..m], Y[n/2..n]$**);**

Let q **be the index that minimize** $f(q, n/2) + g(q, n/2)$**;**

$P = P \cup \{v_{q,n/2}\}$**; /*** $P = \emptyset$ **when the algorithm is first called */**

Divide-and-Conquer-Alignment($X[1..q], Y[1..n/2]$**);**

Divide-and-Conquer-Alignment-backward($X[q..m], Y[n/2..n]$**);**

Theorem. [Hirschberg 1975] Divide-and-Conquer-Alignment algorithm runs in $O(mn)$ time and uses $O(m+n)$ space.

Proof. Let $T(m,n)$ be the running time. It takes $O(mn)$ time to compute q that minimizes $f(q,n/2)+g(q,n/2)$. Therefore,

$$T(m,n) \le cmn + T(q,n/2) + T(m-q,n/2), T(m,2) \le cm, T(2,n) \le cn.$$

An rough estimation, assume $m=n$ and $q=n/2$. Then $T(n,n) \le 2T(n/2,n/2)+cn^2$. By Master Theorem, $T(n,n)=O(n^2)$. So, we expect $T(m,n)=O(mn)$.

For $k \ge c$, $T(m,2) \le 2km$ and $T(2,n) \le 2kn$. Assume $T(m',n') \le km'n'$ for all m',n' s.t. $m'n' < mn$. Then

$$\begin{aligned} T(m,n) &\le cmn + T(q,n/2) + T(m-q,n/2) \le cmn + kqn/2 + k(m-q)n/2 \\ &= cmn + kmn/2 = (c+k/2)mn. \end{aligned}$$

Choose $k=2c$, $T(m,n) \le 2cmn$. $\square$