Dynamic Programming

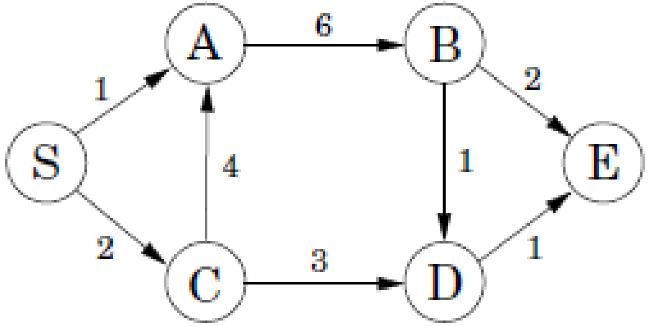
Gogos Christos

November 2019

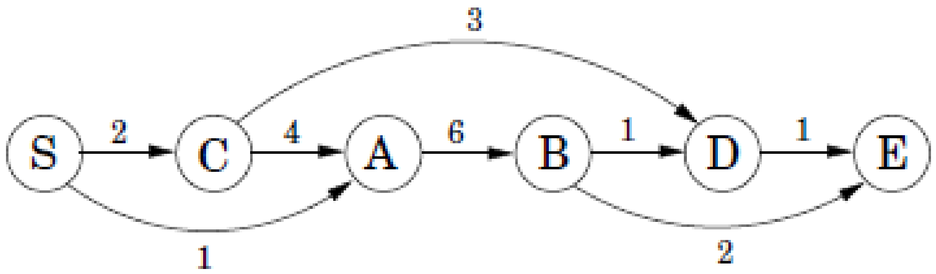
Department of Informatics and Telecommunications

University of Ioannina

Shortest path in a DAG



Linearization of a DAG



$\texttt{dist}(D) = \min\{\texttt{dist}(B) + 1, \texttt{dist}(C) + 3\}$

initialize all $\texttt{dist}(\cdot)$ values to ∞
$\texttt{dist}(s) = 0$
for each $v \in V \backslash \{s\}$, in linearized order:
    $\texttt{dist}(v) = \min_{(u,v) \in E}\{\texttt{dist}(u) + l(u, v)\}$

Longest increasing subsequence problem

5 2 8 6 3 6 9 7

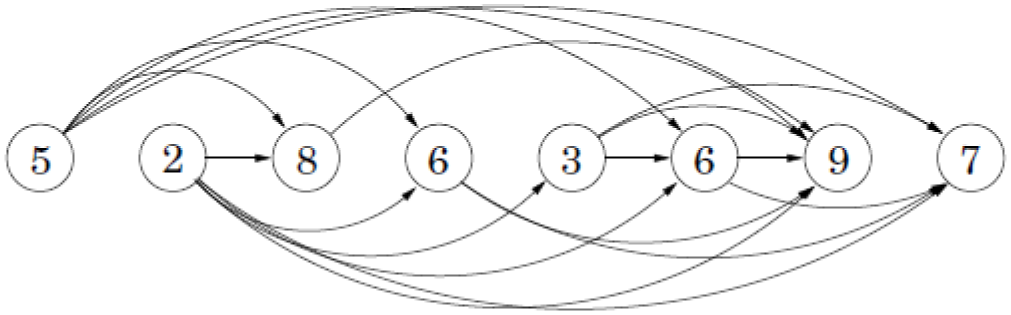
5 2 8 6 3 6 9 7

for $j = 1, 2, \ldots, n$:

  $L(j) = 1 + \max\{L(i) : (i, j) \in E\}$

return $\max_j L(j)$

Implicit DAG



Edit distance problem

- A natural measure of the distance between two strings is the extent to which they can be *aligned*, or matched up.
- Example: SNOWY vs SUNNY

S	—	N	O	W	Y
S	U	N	N	—	Y

Cost: 3

—	S	N	O	W	—	Y
S	U	N	—	—	N	Y

Cost: 5

- A dynamic programming solution
  - x[1..m] is the first substring
  - y[1..n] is the second substring
- Subproblem E(i,j): find the edit distance between a prefix of the first substring x[1..i] and a prefix of the second substring y[1..j]

E(7,5)

$\boxed{\text{E X P O N E N}}$ T I A L

$\boxed{\text{P O L Y N}}$ O M I A L

Express subproblem in terms of smaller subproblems

- Problem E(i,j)
  - Find the best alignment between x[1..i] and y[1..j]
- The rightmost column can only be one of three things:

$\begin{matrix} x[i] \\ - \end{matrix}$ or $\begin{matrix} - \\ y[j] \end{matrix}$ or $\begin{matrix} x[i] \\ y[j] \end{matrix}$

+1 cost
remains to align
x[1..i-1] and y[1..j]

+1 cost
remains to align
x[1..i] and y[1..j-1]

+1 cost if x[i]≠y[j]
remains to align x[1..i-1]
and y[1..j-1]

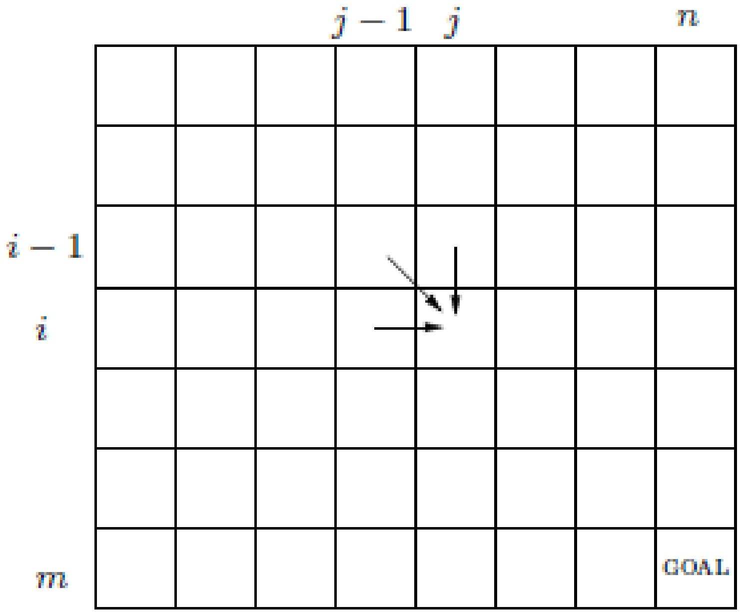
$$E(i,j) = \min\{1 + E(i-1,j),\ 1 + E(i,j-1),\ \textbf{diff}(i,j) + E(i-1,j-1)\}$$

EXPONENTIAL vs POLYNOMIAL
E(4,3) refers to EXPO vs POL

$\begin{matrix} \text{O} \\ - \end{matrix}$ or $\begin{matrix} - \\ \text{L} \end{matrix}$ or $\begin{matrix} \text{O} \\ \text{L} \end{matrix}$

$$E(4,3) = \min\{1 + E(3,3),\ 1 + E(4,2),\ 1 + E(3,2)\}$$

Table of subproblems



The table of subproblems

		P	O	L	Y	N	O	M	I	A	L
	0	1	2	3	4	5	6	7	8	9	10
E	1	1	2	3	4	5	6	7	8	9	10
X	2	2	2	3	4	5	6	7	8	9	10
P	3	2	3	3	4	5	6	7	8	9	10
O	4	3	2	3	4	5	5	6	7	8	9
N	5	4	3	3	4	4	5	6	7	8	9
E	6	5	4	4	4	5	5	6	7	8	9
N	7	6	5	5	5	4	5	6	7	8	9
T	8	7	6	6	6	5	5	6	7	8	9
I	9	8	7	7	7	6	6	6	6	7	8
A	10	9	8	8	8	7	7	7	7	6	7
L	11	10	9	8	9	8	8	8	8	7	6

The final table of values found by dynamic programming

Algorithm and the base cases

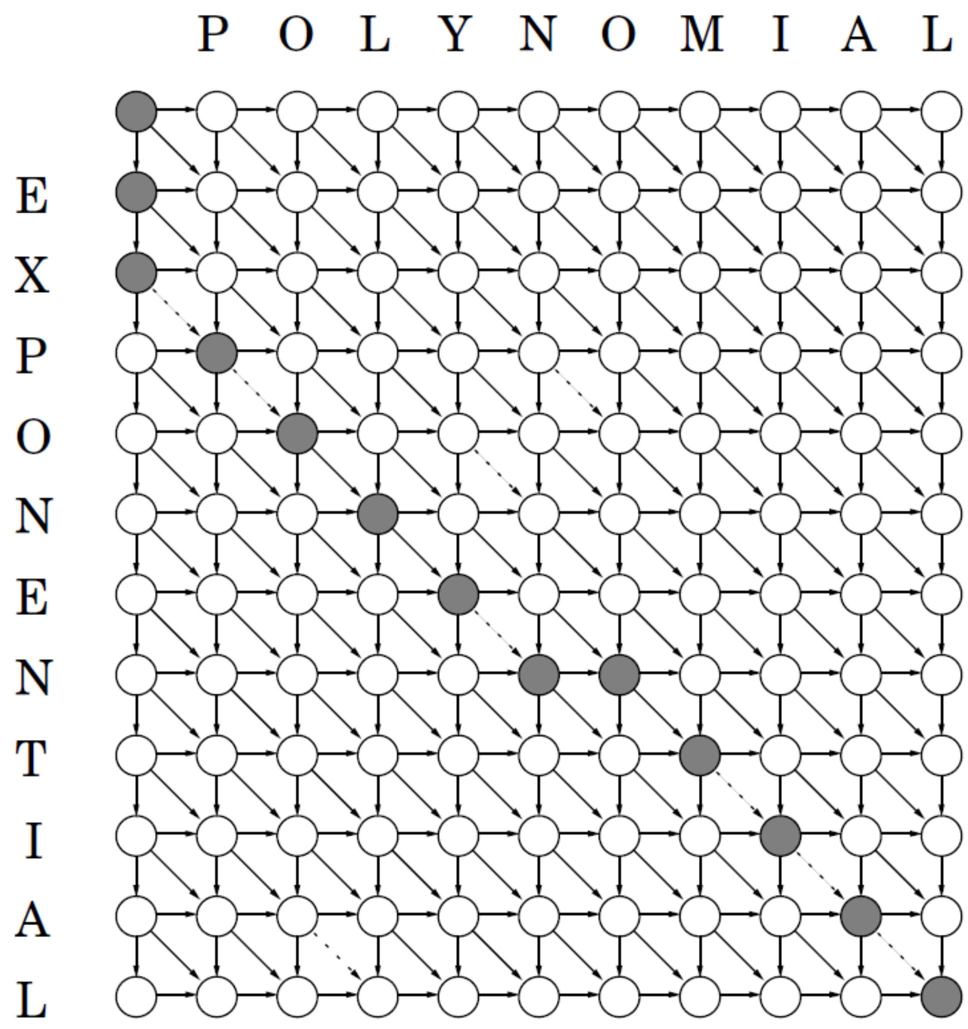
```
for i = 0,1,2,...,m:
    E(i,0) = i
for j = 1,2,...,n:
    E(0,j) = j
for i = 1,2,...,m:
    for j = 1,2,...,n:
        E(i,j) = min{E(i-1,j)+1, E(i,j-1)+1, E(i-1,j-1)+diff(i,j)}
return E(m,n)
```

E	X	P	O	N	E	N	–	T	I	A	L
–	–	P	O	L	Y	N	O	M	I	A	L

Edit distance = 6

- Base cases:
  - E(i,0) is the edit distance between the 0-length prefix of y (the empty string) and the first letters of i ➔ E(i,0)=i
  - Similarly E(0,j)=j
- The procedure fills in the table row by row, and left to right within each row
- Each entry takes constant time to fill in, so the overall running time is just the size of the table, O(mn)

The underlying DAG



E	X	P	O	N	E	N	–	T	I	A	L
–	–	P	O	L	Y	N	O	M	I	A	L

- Edges:
  - (i-1,j)→(i,j)
  - (i,j-1)→(i,j)
  - (i-1,j-1)→(i,j)
- Set all edge lengths to 1, except for:

  {(i-1,j-1)→(i,j): x[i]=y[j]}

  shown dotted in the figure
- Each move:
  - down → deletion
  - right → insertion
  - diagonal → match or substitution

Knapsack problem

- During a robbery, a burglar finds much more loot than he had expected and has to decide what to take
- His bag (or “knapsack”) will hold a total weight of at most W
- There are n items to pick from, of weight $w_1,\dots,w_n$ and dollar value $v_1,\dots, v_n$
- What's the most valuable combination of items he can fit into his bag?

$$W = 10$$

Item	Weight	Value
1	6	\$30
2	3	\$14
3	4	\$16
4	2	\$9

→ 1 and 3 (total: \$46)

Subproblem definition + DP algorithm

- K(w,j): maximum value achievable using a knapsack of capacity w and items 1,…,j
- The answer we seek is K(W,n)
- We can express K(w,j) in terms of problems K(.,j-1)

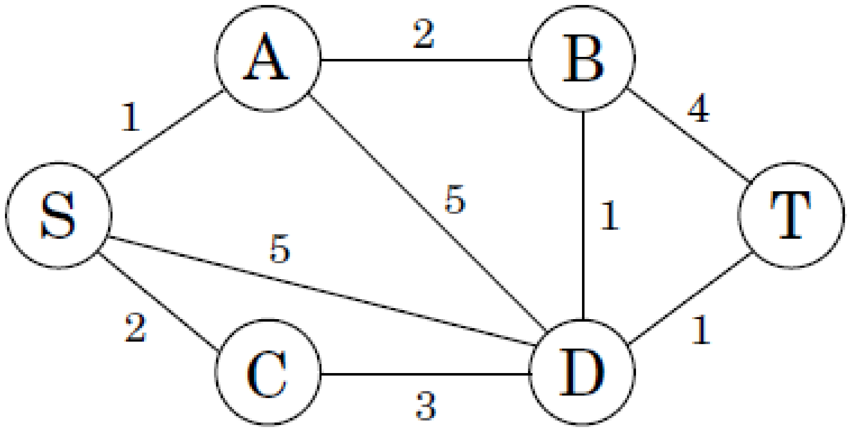
```
Initialize all K(0,j) = 0 and all K(w,0) = 0
for j = 1 to n:
   for w = 1 to W:
      if w_j > w:   K(w,j) = K(w,j-1)
      else:   K(w,j) = max{K(w,j-1), K(w-w_j,j-1) + v_j}
return K(W,n)
```

$$K(w,j) = \max\{K(w-w_j, j-1) + v_j, K(w, j-1)\}$$

either item j is needed to achieve the optimal value, or it isn't needed

Shortest reliable paths

- Find the shortest path from ***s*** to ***t*** that uses at most ***k*** edges
- In dynamic programming, the trick is to choose subproblems so that all vital information is remembered and carried forward



- Define for each vertex ***v*** and each integer ***i≤k***, ***dist(v,i)*** to be the length of the shortest path from ***s*** to ***v*** that uses ***i*** edges
- The starting values ***dist(v,0)*** are **∞** for all vertices except ***s***, for which it is 0

$$\mathtt{dist}(v,i) = \min_{(u,v)\in E}\{\mathtt{dist}(u,i-1)+\ell(u,v)\}$$

All-pairs shortest paths

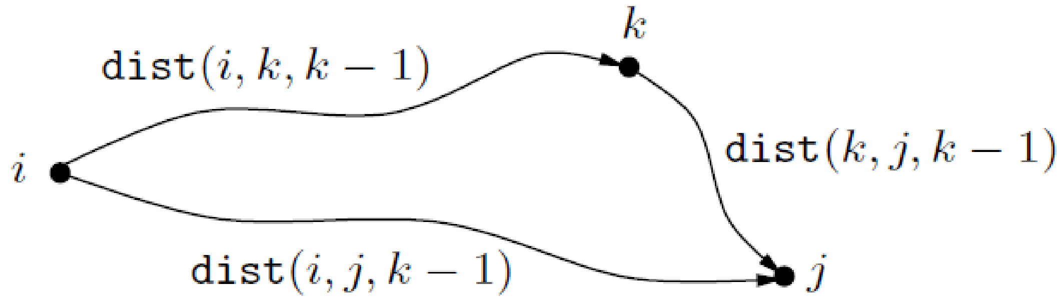
- We want to find the shortest path between all pairs of vertices
- **Approach 1**: execute |V| times the shortest path algorithm, once for each starting node, $O(|V|^2E)$
- **Approach 2**: Dynamic Programming, Floyd-Warshall algorithm, $O(|V|^3)$

- When no intermediate nodes are allowed, the shortest path from u to v is simply the direct edge (u,v), if it exists
- We expand the set of permissible intermediate nodes (one node at a time), updating the shortest path lengths at each step
- Eventually this set grows to all of V, at which point all vertices are allowed to be on all paths, and we have found the true shortest paths between vertices of the graph

All-pairs shortest paths subproblems

- Number the vertices in V as {1,2,…,n}
- dist(i,j,k): length of the shortest path from node i to node j in which only nodes {1,2,…,k} can be used as intermediates
- dist(i,j,0)=length of the edge between i and j if it exists, ∞ otherwise

- Expand the intermediate set to include an extra node:
  - reexamine all pairs i,j and check whether using k as an intermediate point gives us a shorter path from i to j



$$\texttt{dist}(i,k,k-1)+\texttt{dist}(k,j,k-1) \; < \; \texttt{dist}(i,j,k-1)$$

Floyd-Warshall algorithm

```
for i = 1 to n:
    for j = 1 to n:
        dist(i, j, 0) = ∞
for all (i, j) ∈ E:
    dist(i, j, 0) = ℓ(i, j)
for k = 1 to n:
    for i = 1 to n:
        for j = 1 to n:
            dist(i, j, k) = min{dist(i, k, k − 1) + dist(k, j, k − 1),  dist(i, j, k − 1)}
```

Sources

- Algorithms. S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani, 2006