Chapter 6 Dynamic Programming

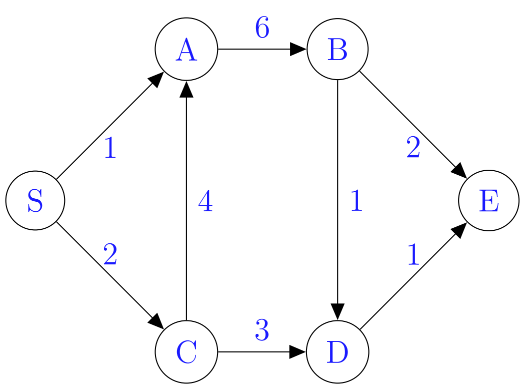
Joe Song

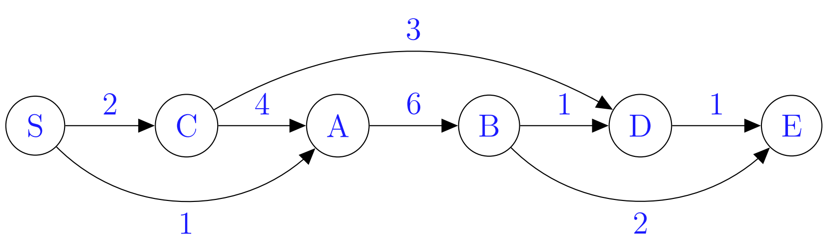
November 18, 2019

The notes are based on Chapter 6 of Dasgupta, Papadimitriou and Vazirani. Algorithms. 2008. McGraw-Hill. New York.

Contents

1 Shortest paths in dags (revisiting)





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

Features of dynamic programming:

Subproblems: What are the distances at each node $\{dist(v) : v \in V\}$?

Recurrence equation: dist(v) $= \min\limits_{(u,v)\in E} dist(u) + l(u,v)$

Optimality: shortest path to v requires evaluating shortest path to u.

Dynamic programming (DP): store the subproblem solutions and reuse them in larger problems.

Difference from divide-and-conquer:

- DP “stores” all solutions to sub-problems; but divide-and-conquer does not
- DP uses iterative solutions (for-loops); but divide-and-conquer uses recursive calls

The term ‘dynamic programming’

- was invented by Richard Bellman in 1953.
- ‘programming’ meant planning, not coding.

2 Longest increasing subsequences

In the following sequence of numbers, a longest increasing subsequence is 2, 3, 6, 9:

5 2 8 6 3 6 9 7

Let x be a sequence of n numbers.

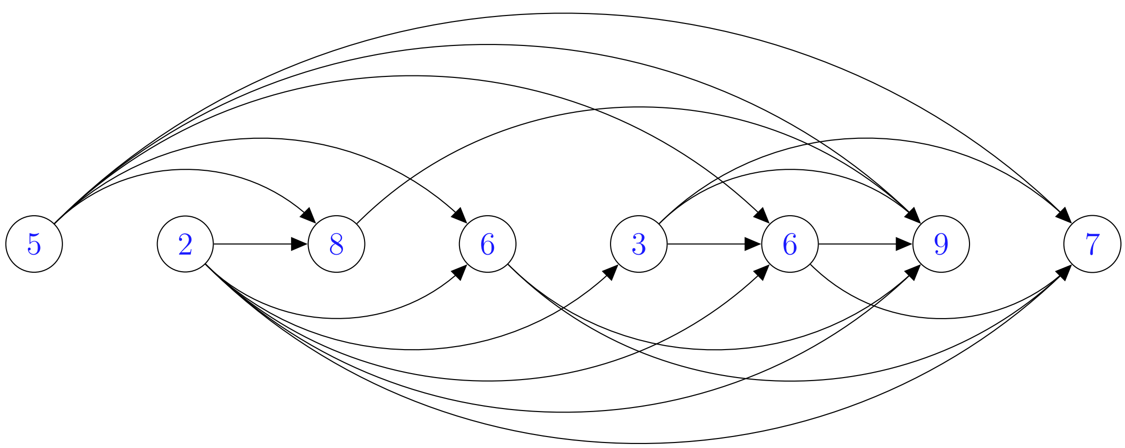
The number of subsequences is exponential:

$$2^n$$

Build a graph $G = (V, E)$ that contains n nodes which are the values in x.

In G, there is a directed edge $(x[i], x[j])$ if and only if $i < j$ and $x[i] \leq x[j]$.

Graph for the above sequence of eight numbers:



Longest-increasing-subsequence-1(E)
for $j = 1, 2, \ldots, n$
    $L(j) = 1 + \max\{L(i) : (i, j) \in E\}$
return $\max_j L(j)$

The following is a graph free solution:

Longest-increasing-subsequence-2(x)
n = length(x)
for $j = 1, \ldots, n$
    L[j] = 1
    for $i = 1, \ldots, j$

if $x_i < x_j$ and $L[j] < L[i]+1$
  $L[j] = L[i]+1$
return $\max_j L[j]$

- There is an ordering on the subproblems
- A relation shows how to solve a subproblem given the answers to "smaller" subproblems, that is, subproblems that appear earlier in the ordering.

Running time: $O(|E|)$, or $O(n^2)$.

Features of dynamic programming:

1. Optimality of subproblems: A longest-increase-subsequence requires smaller longest-increase-subsequence
2. Overlapping subproblems: A longest-increase-subsequence can be reused in longer such sequences.

3 Edit distance

What is edit distance? The smallest number of editing operations needed to convert one string to another.

Allowed editing operations: insertion, deletion, substitution.

No swapping of characters is allowed.

Alignment: The way to align two possible strings

```
S - N O W Y
S U N N - Y
```

Editing cost = 3

```
- S N O W - Y
S U N - - N Y
```

Editing cost = 5

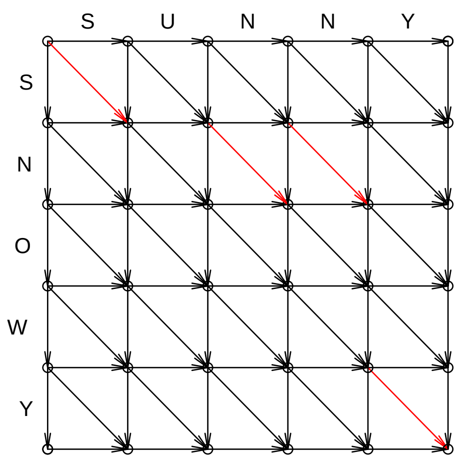
Cost: number of columns in which the letters differ

Edit distance: is the cost of the best possible alignment

A dynamic programming solution

Transforming the edit distance problem to the shortest path problem in a directed acyclic graph (dag):

- All black edges have a weight of one, indicating a insertion/deletion/substitution.
- All red edges have a weight of zero, indicating sharing a same symbol from the two strings.
- The goal is to find a shortest path from top left to bottom right in this dag.



	S	U	N	N	Y
0	1	2	3	4	5
S 1	0	1	2	3	4
N 2	1	1	1	2	3
O 3	2	2	2	2	3
W 4	3	3	3	3	3
Y 5	4	4	4	4	3

Subproblem: Align prefixes $x[1...i]$ and $y[1...j]$. Let the lowest cost be $E(i, j)$.

Three possibilities for aligning the last pair of elements

1. $x[i] : y[j]$
2. $- : y[j]$
3. $x[i] : -$

Example: Align EXPONENTIAL and POLYNOMIAL

		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

		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*

Algorithms:

```
diff(u, v)
    if(u == v) return 0
    else return 1
```

Edit-Distance($x[1..m]$, $y[1..n]$)

  for $i = 0, 1, 2, ..., m$:

    $E(i,0) = i$

  for $j = 0, 1, 2, ..., n$:

    $E(0, j) = j$

  for $i = 1, 2, ..., m$:

    for $j = 1, 2, ..., n$:

$$E(i, j) = \min \begin{cases} E(i-1, j) + 1 \\ E(i, j-1) + 1 \\ E(i-1, j-1) + \text{diff}(x[i], y[j]) \end{cases}$$

  return $E(m,n)$

Running time: $O(mn)$.

4 Of mice and men

Deoxyribonucleic acid (DNA) molecules:

- Each is a large molecule of a double-helix shape
- Each composed of A, C, G, T
- Exists in the nucleus of a cell
- DNA molecules in a human nucleus are 3 billion letter long
- DNA carries a complex program for the human machine
- DNA differs at $\approx 1\%$ between two persons

Computational problems:

1. By comparative studies, one can find similar genes between mouse and human. Functions of genes studied on the mouse can be transferred to human.
2. Sequencing DNA requires assembly of short pieces. Huge problems arise by sequence data of hundreds of giga bytes with the Next-Generation Sequencing technology.
3. Sequence similarity among different species can offer a clue for evolutionary history of species.
4. Measurements on number of RNA and proteins (all derived from genomic DNA sequences by transcription and translation) can help build a virtual machine (mathematical model) that operates like a cell

5 Knapsack

The Knapsack (bagging) problem

Input:

- Knapsack can hold at most W pounds. W is an integer
- n items
- The i-th item weighs w_i pounds. w_i is an integer
- The i-th item has a value of v_i (can be any real number)

Output: Most valuable combination of items the thief can fit into the knapsack.

Total weight limit is W.

5.1 Knapsack with repetition

Repetition: each item has unlimited quantities

Dynamic programming:

Subproblem: $K(w)$ the maximum total value with a total weight no more than w pounds.

Recurrence:

$$K(w) = \max_{i:w_i \le w} (\{0\} \cup \{K(w - w_i) + v_i\}) \quad 0 < w \le W$$

Initialization: $K(w) = 0$ for $0 \leq w < \min w_i$.

Example 1: Select among the four items (repetition allowed) such that the total value is maximized and the total weight W is no more than 10.

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

w	0	1	2	3	4	5	6	7	8	9	10
$K(w)$	0	0	9	14	18	23	30	32	39	44	48

Algorithm:

Knapsack-with-repeat(v, w, W)
$K(0) = 0$
for $w = 1$ to W
    $K(w) = \max(\{0\} \cup \{K(w - w_i) + v_i : w_i \leq w\})$
return $K(W)$

Runtime complexity: $O(nW)$.

Remark on runtime: This seemingly is a polynomial algorithm as it is linear in n and W. However, if we consider the number of bits used for W, which is $\log W$, then the algorithm is exponential.

5.2 Knapsack without repetition

Without repetition: there is one of each item

Dynamic programming:

Subproblem: $K(w, j)$ the maximum total value with a total weight no more than w pounds, that one can do for item 1 to j.

Recurrence:

$$K(w, j) = \begin{cases} \max\{K(w - w_j, j-1) + v_j, K(w, j-1)\} & w \geq w_j, 0 < j \leq n \\ K(w, j-1) & w < w_j, 0 < j \leq n \end{cases}$$

Initialization:

$$K(0,j) = 0 \quad (j = 0,\dots,n)$$
$$K(w,0) = 0 \quad (w = 0,\dots,W)$$

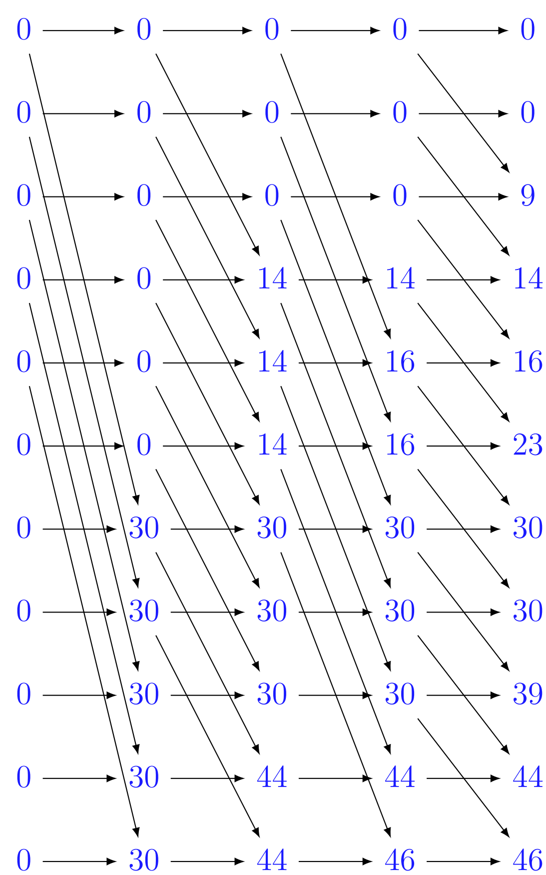
Example 2: Select among the four items without repetition such that the total value is maximized and the total weight W is no more than 10.

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

Dynamic programming matrix $K(w,j)$:

	j				
w	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	9
3	0	0	14	14	14
4	0	0	14	16	16
5	0	0	14	16	23
6	0	30	30	30	30
7	0	30	30	30	30
8	0	30	30	30	39
9	0	30	44	44	44
10	0	30	44	46	46

Directed acyclic graph corresponding to the dynamic programming solution:



Algorithm:

Knapsack-without-repeat(v, w, W)
Initialize all $K(0, j) = 0$ and $K(w, 0) = 0$
for $j = 1$ to n:
  for $w = 1$ to W:
    if $w_j > w$: $K(w, j) = K(w, j-1)$
    else $K(w, j) = \max\{K(w - w_j, j-1) + v_j, K(w, j-1)\}$
return $K(W, n)$

Runtime complexity: $O(nW)$.

Remark on runtime: This seemingly is a polynomial algorithm as it is linear in n and W. However, if we consider the number of bits used for W, which is $\log W$, then the algorithm is exponential in the number of bits used for W.

6 More on shortest paths in graphs

6.1 Shortest reliable paths

Problem statement:

Input: A weighted graph G, a source node s, an integer k for the maximum number of edges allowed on a path.

Output: The distance to s using at most k edges.

Subproblem: $dist(v, i)$: the length of the shortest path from s to v via at most i edges.

Recurrence:

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

Algorithm:

---

procedure Shortest-Reliable-Paths(G, s)
for $v = 1, 2, \ldots, n$:
  $dist(v, 0) = \infty$
$dist(s, 0) = 0$
for $i = 1, 2, \ldots, k$:
  for each $v \in V$:

$dist(v,i) = dist(v,i-1)$
for each $(u,v) \in E$:
if $dist(v,i) > dist(u,i-1) + l(u,v)$
$dist(v,i) = dist(u,i-1) + l(u,v)$
return $dist$

Running time: $O(k(|V|+|E|))$.

6.2 All-pair shortest paths—Floyd-Warshall algorithm

Problem statement:

Input: A graph G with possibly negative weights, but no negative cycles.

Output: Distance between all pairs of nodes in the graph.

Subproblem:

dist(i, j, k): distance between i and j whose intermediate nodes can only be $1, \ldots, k$.

Recurrence:

dist(i, j, k)= min {dist$(i, k, k-1)$ + dist$(k, j, k-1)$, dist$(i, j, k-1)$}

Algorithm:

```
procedure Floyd-Warshall(G)
for i = 1,2,...,n:
    for j = 1,2,...,n:
        dist(i,j,0)=∞
for all (i,j) ∈ E:
    dist(i,j,0)=l(i,j)
for k = 1,2,...,n:
    for i = 1,2,...,n:
        for j = 1,2,...,n:
            dist(i,j,k)= min {dist(i,k,k−1) + dist(k,j,k−1), dist(i,j,k−1)}
return dist
```

Running time: $O(|V|^3)$.