Yaqzan Ali

250 644 709

1)

PRINT-LCS (c, X, i, j)

If (i = 0 or j = 0)

Return

If (c[i,j] = c[i-1, j-1])

PRINT-LCS(c, X, i-1, j-1)

Print xi

Else if ( c[i-1, j] ≥ c[I, j-1] )

PRINT-LCS(c, X, i-1, j)

Else

PRINT-LCS(c, X, i, j-1)

2)

Let *i1 , . . . , in* be the items with values *v1 , . . . , vn* and *w1 , . . . , wn* be their weights. Let *W* be the maximum knapsack weight:

1. *w1 ≤ w2 ≤ . . . ≤ wn*
2. *v1 ≥ v2 ≥ . . . ≥ nn*

**The Algorithm:**

w = 0

S = null

For (i = 1 ; i ≤ n ; i++){

If (w + wi ≤ W){

w += wi

S = S U {i}

}

}

**Proof:**

For the optimal packing of S with i1 ∈ S, the packing of Sn = (S \ ik), is optimal for the items

*I2 , . . . , in*  and Wn = W – w1. Indeed, if Sn is not optimal, we can improve the original packing S by improving Sn.

3)

Sort the list of points so that *x1 ≤ . . . ≤ xn*

While (list is not empty)

Pick the leftmost point in the list, *x1*

Take the Interval [*x1, x1 +1]*

Remove all points *x1, x1 +1, . . . , x1 + k,* such that *x1+k -x1 ≤ 1* from the list

Proof:

Assume, by contradiction, that when looking at the leftmost point *x1,* we selected an interval

[*x1* – a, *x1* + b], such that a ≥ 0, b ≥ 0, and b – a = 1. Since *x1* is the leftmost point in the set of points, we know that we will not reach any points in in the interval [*x1* – a, *x1* ). Therefore, we’d be able to maximize the total number of possible points covered iff we select the interval

[*x1* – a, *x1* + b], assuming that a = 0, and b =1.

4)

If we were to look at the possible source files S using n bits and using compressed files E (using n bits), we’d see that it is 2n+1 -1. Since all compression algorithm must assign each element s ∈ S to a distinct element e ∈ E, the algorithm cannot possibly compress the source file.

5)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| P | B | A | B | B | A | B | B | A | B | B | A | B | A | B | B | A | B | B |
| next | 0 | 0 | 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 | 1 | 1 | 2 | 3 | 4 |

6)

Input: Strings T[0…n] of P[0…n]

Output: Starting index of substring of T matching P

F 🡨 compute failure function of Pattern P

i 🡨 0

j 🡨 0

While i < length[T] do

If j 🡨 m – 1

Return i – m+1

i 🡨 i + 1

j 🡨 j + 1

else if (j > 0)

j 🡨 f(j - 1)

else

i 🡨 i + 1

end while

7) Algorithm

Put all edges in a heap, put each vertex in a set by itself

**While** (not found a MST yet) do begin

Delete max edge, {u,v} from heap;

**If** (u and v are not in the same set)

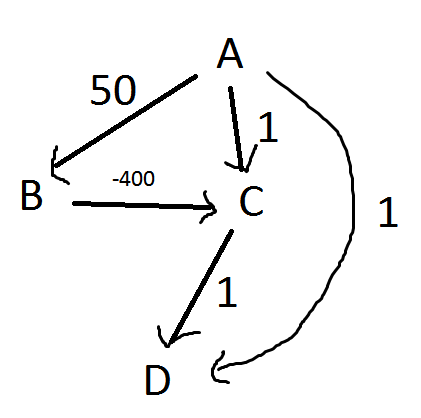
Mark {u,v} as tree edge;

Union sets containing u and v

**End if**

**End while**

8)



First, we set A to 0, and all others to infinity. Expanding node A, we set C to 1, B to 50, and D to 1. We expand C and D which give no effect. Then we expand B. This changes C to -350. However, D is still set to 1, despite the shortest path to it being -349. Thus the algorithm fails to accurately compute.

9)

All-Pair-shortest-path is still correct in this case.

Let d**k**(i,j) = shortest path from i to j involving **O** k edges.

d**1**(i,j) = is original weight matrix.

Compute d**k+1**’s from d**k**’s by seeing if adding edge helps:

d**k+1**(i,j) = Min { d**k**(i,m) + d**1**(m,j) }

Assuming that there are no negative cycles, d**V-1**(i,j) is the solution.

10)

We use Dijkstra’s algorithm the shortest paths to all the vertices, and then use the distances to calculate the shortest cycle.

Apply Dijkstra’s algorithm

Min weight = infinity

**For each** u ∈ V[G]

For each w ∈ Adj[u]

If (w = v && min-weight < d[u] + weight(u,w)

Min-weight = d[u] + weight(u,w)

**Return min-weight**