Algorithms

LCS

Dynamic Programming

The dynamic programming algorithm for computing the LCS between two input texts (X, Y, with lengths m, n, respectively) is found as follows (adapted from Amaury Habrard’s slides):

Optimal Substructure:

1. For an LCS Z = <z1,…,zk>, if xm = yn, then xm = yn = zk, and Zk-1 is the LCS of Xm-1 and Yn-1
2. If xm ≠ yn, then zk ≠ xm implies Z is LCS of Xm-1 and Yn
3. If xm ≠ yn, then zk ≠ yn implies Z is LCS of Xm and Yn-1

Recursive Solution:

1. c[i,j] = 0 if i = 0 or j = 0
2. c[i,j] = c[i-1,j-1] + 1 if i,j > 0 and xi = yj
3. c[i,j] = max(c[i-1,j],c[i,j-1]) if i,j > 0 and xi ≠ yj

LCS-Length Algorithm(input: X,Y):

1. *m = X.length*
2. *n = Y.length*
3. *b[1..m,1..n], c[0..m,0..n]*
4. *for i in 1..m*
5. *c[i,0] = 0*
6. *for j in 0..n*
7. *c[0,j] = 0*
8. *for i in1..m*
9. *for j in 1..n*
10. *if xi == yj*
11. *c[i,j] = c[i-1,j-1] + 1*
12. *b[i,j] = “d” #for diagonal*
13. *else if c[i-1,j] >= c[i,j-1]*
14. *c[i,j] = c[i-1,j]*
15. *b[i,j] = “u” #for up*
16. *else*
17. *c[i,j] = c[i,j-1]*
18. *b[i,j] = “l” #for left*
19. *return c and b*

Algorithm for building LCS from b matrix *Print-LCS*(input: b, X, X.length, Y.length):

1. *if i == 0 or j == 0*
2. *return*
3. *if b[i,j] == “d”*
4. *Print-LCS(b,X,i-1,j-1)*
5. *elseif b[i,j] == “u”*
6. *Print-LCS(b,X,i-1,j)*
7. *else*
8. *Print-LCS(b,X,i,j-1)*

LCS Linear Space Forwards: The following algorithm finds the length of the LCS of two input strings using dynamic programming with linear space complexity.

LCS\_LSF (input: X,Y):

1. *m = X.length*
2. *n = Y.length*
3. *c[1..2,1..n]*
4. *col = [] #for use in divide and conquer alg (see next section)*
5. *for i in 1..m*
6. *c[1,0] = 0*
7. *for j in 1..n*
8. *if xi == yj*
9. *c[1,j] = c[0,j-1] + 1*
10. *else if c[1,j-1] >c[0,j]*
11. *c[1,j] = c[1,j-1]*
12. *else*
13. *c[1,j] = c[0,j]*
14. *c[0,:] = c[1,:]*
15. *col.append(c[1,n])*
16. *return c[1,length\_Y] , col #return length of LCS, column*

LCS Linear Space Backwards: The following algorithm is a linear space implementation of the LCS dynamic programming algorithm, except it starts at the end of texts and works towards the beginning.

LCS\_LSB(input: X,Y):

1. *m = X.length*
2. *n = Y.length*
3. *c[1..2,1..n]*
4. *col = [] #for use in divide and conquer alg (see next section)*
5. *for i in m-1..0*
6. *for j in n-1..0*
7. *if X[i+1]==Y[j+1]*
8. *c[1,j] = c[0,j+1]+1*
9. *else if c[1,j+1]>c[0,j]*
10. *c[1,j] = c[1,j+1]*
11. *else*
12. *c[1,j] = c[0,j]*
13. *c[0,:] = c[1,:]*
14. *col.insert(0,c[1,0])*
15. *return c[1,0], col #return length of LCS, column*

LCS Divide and Conquer: The following algorithm uses the LCS linear space forward and backward algorithms to compute the LCS of two input spaces in linear space, with the functionality to return the actual LCS (not just the length).

LCS\_DC(input: X,Y):

1. *m = X.length*
2. *n = Y.length*
3. *if m\*n == 0*
4. *return [] #if one of the texts is empty, return an empty LCS*
5. *elseif m <= 2 or n <= 2*
6. *return LCS\_DyProg(X,Y) #if one of the texts is small, can do normal Dynamic Prog*
7. *else*
8. *breakpt = floor(n / 2)*
9. *length, c = LCS\_LSF(X, Y[0:breakpt])*
10. *length2, c2 = LCSLSB(X, Y[breakpt:n])*
11. *q = maxindex(c, c2) #q is the index that maximizes (c[q] + c2[q])*
12. *LCSL = LCS\_DC(X[0..q],Y[0..breakpt])*
13. *LCSR = LCS\_DC(X[q+1,m],Y[breakpt+1,n])*
14. *return LCSL + LCSR #return concatenation of sub LCS*