ECE374 SP23 HW5

Contributors

Zhirong Chen (zhirong4)

Ziyuan Chen (ziyuanc3)

Problem 5

A certain string processing language allows the programmer to break a string into two pieces. It costs n units of time to break a string of n characters into two pieces, since this involves copying the old string.

A programmer wants to break a string into many pieces, and the order in which the breaks are made can affect the total amount of time used. Give a dynamic programming algorithm that takes a list of character positions after which to break and determines the cheapest break cost in $O(n^3)$ time.

Solution

Preprocessing. Assume the string s has length n and the array of breaking points B has length m. After breaking, the string is divided into

$$s[0:B[0]], \quad s[B[0]+1:B[1]], \quad \ldots, \quad s[B[m-1]+1:n-1]$$

For convenient indexing, we add -1 to the beginning and $n-1$ to the end of the breaking array. Now B becomes

$$\{-1, \quad B[0], \quad \ldots, \quad B[m-1], \quad n-1\}$$

and each subarray has a unified representation of $s[B[i]+1:B[i+1]], 0 \leq i \leq m$.

Recurrence function. Let $cost(i,j)$ represents the breaking cost from point i to point j. This function is only defined when $i < j$. Obviously $cost(i, i+1) = 0$ since the substring need not to be cut.

$$cost(i,j) = \begin{cases} 0 & j = i+1 \\ \min\limits_{i<k<j} \left(cost(i,k) + cost(k,j) + B(j) - B(i)\right) & \text{otherwise} \end{cases}$$

The intuition is that we iterate k between (i,j) and find the breaking point with minimal cost. The algorithm outputs $cost(0, m+1)$ as the final answer.

Time complexity. The time complexity is $O(n^3)$ since there are n^2 states (we are filling out a $m \times m$ DP matrix) and each takes $O(n)$ time to compute (there's an iteration).

Pseudocode.

```
BreakString(s, B)
    n ← len(s)
    m ← len(B)
    B.pushFront(−1)    // make B 0-indexed
    B.pushBack(n − 1)
    cost ← −1 × MatrixOfOnes(m + 2, m + 2)
    for i ← 0 to m + 1
        cost(i, i + 1) ← 0
    for i ← 0 to m
        for j ← i + 2 to m + 2
            MinCost ← ∞
            for k ← i + 1 to j − 1
                ThisCost ← cost(i, k) + cost(k, j) + B(j) − B(i)
                if (ThisCost < MinCost)
                    MinCost ← ThisCost
            cost(i, j) ← MinCost
    return cost(0, m + 1)
```