CSOR 4231, Section 3: Analysis of Algorithms I - Problem Set #5

Kangwei Ling - `kl3076@columbia.edu`

March 6, 2019

Collaborators

Zefeng Liu (zl2715), Kunyan Han (kh2931), Luoyao Hao (lh2913).

1 DPV 6.1

- Initialization: maxSum = 0, start = 0, maxStart = -1, maxEnd = -1, currentSum = 0.
- For i in $1..n$:
  - currentSum = currentSum $+a_i$.
  - if currentSum $>$ maxSum:
    - maxStart = start
    - maxEnd = i
    - maxSum = currentSum
  - if currentSum $\leq$ 0:
    - start = $i + 1$
    - currentSum = 0
- if maxStart $\neq -1$, then return the sequence $a_{maxStart}, ...a_{maxEnd}$ with the sum of maxSum, otherwise return empty sequence with the sum of 0.

This algorithm runs in linear time since it only iterate through the sequence of numbers.

The algorithm is correct because the following invariant holds: Before each iteration of the for loop of i in $1..n$, currentSum holds the maximum possible sum of the contiguous sequence ends in $i - 1$ (or an empty sequence sum 0).

For $i = 0$, this holds obviously. For $i > 0$, after the iteration of i, currentSum is updated to hold the maximum possible sum of the continguous sequence ends in i, or empty sequence with sum = 0 if it adds up to a negative number or zero. And in each iteration, the maxSum is updated with respect to every possible maximum sum at each position.

2 DPV 6.5

(a) If we vertically number the squares in one column A,B,C,D. The legal patterns we can choose to place the pebbles are: N, A, B, C, D, AC, AD, BD, where for instance, BD means to place one pebble at the second row and another at the fourth row. N means that we don't place any pebbles.

(b) The table below gives the compatible patterns:

pattern	compatible patterns in adjacent column
N	N, A, B, C, D, AC, AD, BD
A	B, C, D, BD
B	A, C, D, AC, AD
C	A, B, D, AD
D	A, B, C, AC
AC	B, D, BD
AD	B, C
BD	A, C, AC

Let $V(P)$ be the set of compatible patterns of pattern P.

Let $S(P,i)$ the sum of value of the squares if we place column i with pattern P.

Let $maxPlace(P,i)$ be the maximal sum we can get for placing at column 1...i by place with pattern P at i th column, and $Prev(P,i)$ be pattern chosen for the previous column.

With the above definitions, we can derive the following transition:

$$maxPlace(P,i) = S(P,i) + \max_{p \in V(P)} maxPlace(p, i-1)$$

$$Prev(P,i) = \arg\max_{p \in V(P)} \quad maxPlace(p, i-1)$$

$$maxPlace(P,0) = S(P,0)$$

With the above transition fomula, we can iterate through the columns 1...n. We can obtain the optimal placement by: first we get the max sum and the pattern for the last column by finding the maximum among ($maxPlace(p,n)$, where $p \in \{N, A, B, C, D, AC, AD, BD\}$). And we can use $Prev$ to find the pattern for previous columns.

This algorithm runs in linear time since the number of available patterns is only a small constant, and we just iterate through columns 1...n, each step only takes O(1) time to compute $maxPlace, Prev$.

3 DPV 6.8

Let $LCS(i,j)$ be the longest common substring of x, y that ends in x_i, y_j. By definition, the following formulas hold:

$$LCS(i,j) = \begin{cases} 1 + LCS(i-1, j-1) & \text{if } x_i \neq y_j \\ 0 & \text{otherwise} \end{cases}$$
$$LCS(0,0) = 0$$

Therefore we can get the longest common substring by looping through x, y in the following manner while keeping a best record:

- For i in 1...n
  - For j in 1...m
    update $LCS(i,j)$ according to the formula above.
    If $LCS(i,j) >$ best, update best to $LCS(i,j)$.

This algorithm runs in $O(mn)$ time.

4 DPV 6.12

With the given x, y coordinates, we can sort the n points and label them by their polar angle with respect to $x - axis$. Let the sorted points be $p_1, p_2, ..., p_n$, then $A(i,j)$ denotes the minimum cost triangulation of the polygon spanned by $p_i, ..., p_j$. It's easy to see that for any i:

$$A(i,i) = 0, A(i,i+1) = 0, A(i,i+2) = 0$$

Let $d(i,j)$ be the distance of p_i, p_j, $d(i,j) = \sqrt{(p_{i_x} - p_{j_x})^2 + (p_{i_y} - p_{j_y})^2}$
Let $diag(i,j)$ be the length of the diagonal formed by p_i, p_j, by definition:

$$diag(i,j) = \begin{cases} d(i,j) & \text{if } |i-j| > 1 \text{ and } (i,j) \notin \{(1,n),(n,1)\} \\ 0 & \text{otherwise, } p_i, p_j \text{ are adjacent points} \end{cases}$$

For any triangulation of the polygon spanned by $p_i, ..., p_j$, there must be some $k, i < k < j$, such that the triangulation can be view as the combination of two separate trangulation by polygons $p_i, ..., p_k$ and $p_k, ..., p_j$ (It's easy to see by drawing a graph). From this, we can derive that:

$$A(i,j) = \min_{i<k<j}\{A(i,k) + A(k,j) + diag(i,k) + diag(k,j)\}$$

Using this formula, we can get the minimum triangulation cost, which is exactly $A(1,n)$, we can use $bp(i,j)$ to store the k for $A(i,j)$.

```
• For $l$ in 4...$n$
    – For $i$ in 1...$(n - l + 1)$
      let $j = i + l - 1$
      $A(i,j) = \min_{i<k<j}\{A(i,k) + A(k,j) + diag(i,k) + diag(k,j)\}$
      $bp(i,j) = \arg\min_{i<k<j}\{A(i,k) + A(k,j) + diag(i,k) + diag(k,j)\}$
```

With bp, we can restore the exact k we choose at each step to get the actual triangulation. For example, $bp(1,n)$ gives a k, we can have the diagonals $(1,k), (n,k)$ (may be just adjacent points), and then we use $bp(1,k), bp(k,n)$ to recursively find all diagonals.

Sorting and labeling the points take $O(n \log n)$, calculating $diag(i,j)$ takes $O(n^2)$. Go through all $A(i,j)$ takes $O(n^3)$ (a triple level loop).

Overall, this algorithm runs in $O(n^3)$ time.

5 6.17

Let $D(u)$ denotes that it is possible to make change for u using coins of denominations $x_1, ..., x_n$. Initially, only $D(0) = true$.

If $D(u) = true$, then $D(u + x_i) = true$ for $x_i \in \{x_1, ..., x_n\}$. With this holds, we can write a dynamic programming algorithm to solve for v:

```
• For $u$ in 0..$v$
    – if $D(u)$ is true, then
        ∗ For $x_i \in \{x_1, ..., x_n\}$
          set $D(u + x_i)$ to true. If $u + x_i = v$, we can directly return true.
    – otherwise do nothing.
• return $D(v)$
```

The outer loop runs $O(v)$ times, and inner loop runs $O(n)$ times. Overall this is a $O(nv)$ algorithm.

6 DPV 6.21

Since a unidirected tree is given, we can pick any vertex as the root and construct a tree structure from it. Suppose in the tree structure, we can use $children(v)$ to get the list of children of vertex v.

With each vertex v, let $VC(v)$ be the smallest vertex cover of the subtree rooted at v, let $VCC(v)$ be the smallest vertex cover of all child subtrees of vertex v (actually $VCC(v) = \sum_{n \in children(v)} VC(n)$).

VC, VCC of a vertex can be obtained by using the results of its children. Intuitively, we can write a recursive algorithm to do this.

Procedure $VCVCC(u)$:

- if u has no child, return $(0, 0)$. $(VC(u) = VCC(u) = 0)$
- otherwise
  - let its children be $c_1, c_2, ..., c_k$, we run $VCVCC$ on every child and get $(vc_1, vcc_1), ..., (vc_k, vcc_k)$.
  - calculate $VCC(u) = \sum_{i=1}^{k} vc_i$.
  - for vertex u, there are two ways to cover the tree rooted at u:
    (1) cover u, then we only need to cover all child subtrees. In this way,
    $$VC(u)_1 = 1 + VCC(u)$$
    (2) we don't cover u, but cover all child vertices to satisfy the edges from u to its children. In this way, we need to choose each child and cover all grandchild subtrees.
    $$VC(u)_2 = \sum_{i=1}^{k} (1 + vcc_i)$$
    We choose the minimum of these two: $VC(u) = min(VC(u)_1, VC(u)_2)$.
  - return $(VC(u), VCC(u))$

pick any root $ROOT$, $VCVCC(ROOT)$ will return the the size of the smallest vertex cover of T.

This algorithm only goes through each vertex and edge (parent-child edge) once, it runs in linear time.