به نام خدا

آنالیز الگوریتم ها
نیم‌سال دوم ۰۱-۰۲

دانشکده علوم ریاضی

تمرین ۴ - امیرحسین محمودی - ۹۸۱۰۸۷۷۹

1

We can handle this bya dynamic programming approach.we fill the matrix like the below algorithm and value in the last cell is the length of the LCS.also we use a backtracking method to construct the LCS between S ans T.

Algorithm 1: LCS-Length

```
Input: S[1...m], T[1...n]
Output: C[m,n]
C = Array(0...m, 0...n);
for ( i in (0, m) ) {
    C[i, 0] = 0;
}
for ( j in (0, n) ) {
    C[0, j] = 0;
}
for ( i in (1, m) ) {
    for ( j in (1, n) ) {
        if S[i] = T[j] then
            C[i, j] = C[i-1, j-1] + 1;
        else
            C[i,j] = max(C[i, j-1], C[i-1, j]);
        end
    }
}
Return C[m, n];
```

Algorithm 2: Backtracking-LCS

```
Result: LCS of S and T
Function backtrack(C[0...m,0...n], S[1...m], T[1...n], i, j):
    if $i = 0$ or $j = 0$ then
        return “”
    if $X[i] = Y[j]$ then
        return backtrack($C, S, T, i-1, j-1$) + X[i]
    if $C[i,j-1] > C[i-1,j]$ then
        return backtrack($C, S, T, i, j-1$)
    return backtrack($C, S, T, i-1, j$)
```

Time complexity of this algorithm is $T(|s| \times |t|)$ which is for traversing and filling the matrix.

An explanation to the algorithm:

Let’s say we have two sequences: S (with size s) and T (with size t). We can use Opt(a, b) to represent the length of the Longest Common Subsequence (LCS) of S[1 ... a] and T[1 ... b].

If either s or t is equal to 0, then the only LCS between S and T is an empty sequence. This means that the length of the LCS is Opt(0, a) = Opt(a, 0) = 0 for any value of a.

Now, let’s assume that both s and t are positive. We can look at two different scenarios: either S[s] is equal to T[t], or S[s] is not equal to T[t].

In the case where S[s] is equal to T[t], any LCS of S and T can be created by adding (s, t) to any LCS of S[1 ... s-1] and T[1 ... t-1]. This means that Opt(s, t) is equal to Opt(s-1, t-1) + 1.

Proof:

Let’s say Y (with length y) is a Longest Common Subsequence (LCS) of S and T. We can prove that Opt(s, t) is less than or equal to Opt(s-1, t-1) + 1 as follows:

If the last element of Y is (s, t), then Y’ = Y[1 ... y-1] is a common subsequence of S[1 ... s-1] and T[1 ... t-1]. The length of Y’ is y - 1, which is equal to Opt(s, t) - 1. This shows that Opt(s, t) - 1 is less than or equal to Opt(s-1, t-1).

If the last element of Y is (u, v) where u is not equal to s and v is not equal to t, then we can create a new sequence Y* with length y + 1 by adding (s, t) to Y. This contradicts the fact that Y is an optimal LCS.

If the last element of Y is (u, t) where u is not equal to s (or if the last element of Y is (s, u) where u is not equal to t), then S[u] must be equal to S[s] (or T[u] must be equal to T[t]), since Y is a common subsequence. We can replace u with s (or with t) and use the first point above to conclude.

Therefore, we can say that Opt(s, t) is equal to Opt(s-1, t-1) + 1.

In the scenario where S[s] is not equal to T[t], the Longest Common Subsequence (LCS) of S and T is either the same as the LCS of S[1 ... s - 1] and T or the same as the LCS of S and T[1 ... t - 1]. This means that Opt(s, t) is equal to the maximum value between Opt(s-1, t) and Opt(s, t-1).

Proof:

Here’s a rephrased version of the selected text that you can use for your homework:

If Y is a common subsequence of S[1 ... s-1] and T, then it is also a common subsequence of S and T. This means that Opt(s-1, t) is less than or equal to Opt(s, t).

Let’s say X (with length x) is a Longest Common Subsequence (LCS) of S and T. We can prove that LCSs of S and T are also LCSs of S[1 ... s - 1] and T or S and T[1 ... t - 1]. If x is equal to 0, the proof is straightforward. So let’s assume that X is not empty. Let u and v be equal to X[x]. Since (u, v) is not equal to (s, t), we can assume that u is not equal to s (a similar argument can be made if v is not equal to t). This means that X is a common subsequence of S[1 … s - 1] and T. Therefore, x is less than or equal to Opt(s-1, t).

From the first point above, we can conclude that Opt(s, t) is equal to the maximum value between Opt(s-1, t) and Opt(s, t-1). This shows that X is an LCS of [1 … s-1] and T. This completes the proof.

2

2.1

every tree can have at most one perfect matching. Consider we have two perfect matchings for a tree. since the matchings are perfect, each vertex has degree 0 or 2 in the symmetric difference, so every component is an isolated vertex or a cycle. After removing all isolated vertices in the symmetric difference, all vertices have degree two. Take any such vertex and follow its two edges. What you get is a growing path that eventually closed to a cycle since the graph is finite. Since trees have no cycles, this implies that any two perfect matching are equal, by consisting their symmetric difference .

in a simpler way suppose there are two perfect matchings M and M_0 in a tree $T = (V, E)$. Then, the graph on V with edge set $M \cup M_0$ has no cycles since T is a tree. Therefore, every component of this new graph is either a single edge (common to both M and M_0) or a cycle. Since M and M_0 both cover all the vertices, it follows that $M = M_0$.

also we know that if the number of vertices is odd then no perfect mathching can exists.

an instruction by induction on the number of vertices. It is obvious for n= 0 and n= 1. that it holds for n=k.now for n=k+1 We know that there is a leaf that exists in all perfect matchings Because the edge that connects this leaf to his parent is the only way, remove these two vertices and all the edges connected to them from the graph. The remaining graph will be a forest that contains trees with the number of vertices less than k According to the assumption of induction, each of them has a maximum of only 1 perfect match. As a result, the original graph according to induction has a maximum 1 perfect match.

Algorithm 3: Perfect Matching in a Tree

```
Result: Perfect matching in a tree
Function FindPerfectMatching(T):
    if n is odd then
        return 0
    end
    while T is not empty do
        if there are only isolated nodes then
            return 0;
        end
        Choose any leaf node x;
        Add edge (x, parent(x)) to matching set;
        Remove x and parent(x) with its adjacent edges from T;
    end
    return 1;
```

2.2

To calculate the number of perfect matches in a bit-partite graph, we can use dynamic programming. First, we create a matrix of size $2^{|V|} \times |V|$ to represent all subgraphs. Each row of the matrix represents a subgraph of the original graph, where 1 indicates that the vertex is in the subgraph. We can then calculate the number of perfect matches for each subgraph using dynamic programming.

For each row in the matrix, we keep track of the number of perfect matches for that subgraph. If the total number of vertices in the subgraph is odd or if the number of vertices in the first and second parts are not equal, then the number of perfect matches will be zero. Otherwise, we calculate the number of perfect matches by considering each vertex in the first part and its connected vertices in the second part.

For each pair of connected vertices, we set their values to zero and obtain a subgraph whose value has already been calculated. The value of a subgraph is equal to the sum of all subgraphs obtained by removing one vertex and its neighbors.

The time complexity of this algorithm is $O(|V||E|2^{|V|})$ because we need to traverse all vertices ($O(|V|)$) of all subgraphs ($O(2^{|V|})$) and check each vertex's edges ($O(|E|)$).

Algorithm 4: Calculate number of perfect matches using dynamic programming in a bit-partite graoh

```
Input: Graph $G = (V, E)$
Output: Number of perfect matches in $G$
$C \leftarrow$ Array$(0 \dots 2^{|V|}, 0 \dots |V|)$;
for ( $i \leftarrow 0$ to $2^{|V|}$ ) {
    $C[i, 0] \leftarrow 0$;
    if $|V_i|$ is odd or $|V_{i,1}| \neq |V_{i,2}|$ then
        $C[i] \leftarrow 0$;
    end
    else
        for ( $v_1 \in V_{i,1}$ ) {
            for ( $v_2 \in V_{i,2}$ ) {
                if $(v_1, v_2) \in E$ then
                    $C[i] \leftarrow C[i] + C[i - 2^{v_1} - 2^{v_2}]$;
                end
            }
        }
    end
}
return $C[2^{|V|}]$;
```

3

to find the largest set of cells with the same color in an m*n table and return the coordinates of the corners of the sub-table containing that set is to use a Depth-First Search (DFS) algorithm. Here's how it could work:

1. Initialize variables `maxSize`, `topLeftRow`, `topLeftCol`, ,`bottomRightRow`, and

`bottomRightCol` to keep track of the size and coordinates of the largest set of cells with the same color.

2. Create an m*n matrix of booleans to keep track of which cells have been visited.

3. Iterate through each cell in the table. If the cell has not been visited, perform a DFS on that cell.

4. In the DFS function, mark the current cell as visited and increment a local variable `size` to keep track of the size of the current set of cells with the same color. Also update local variables `minRow`, `maxRow`, `minCol`, and `maxCol` to keep track of the coordinates of the corners of the sub-table containing the current set of cells with the same color.

5. Check all adjacent cells (top, bottom, left, right) of the current cell. If an adjacent cell has the same color and has not been visited, perform a DFS on that cell.

6. After all cells in the current set have been visited, compare the value of `size` with `maxSize`. If size is greater than `maxSize`, update `maxSize` to be equal to `size` and update `topLeftRow`,`topLeftCol`, `bottomRightRow`, and `bottomRightCol` to be equal to `minRow`, `minCol`, `maxRow`, and `maxCol`, respectively. Continue iterating through the table until all cells have been visited. The value of `maxSize` will be the size of the largest set of cells with the same color, and `(topLeftRow, topLeftCol)` and `(bottomRightRow, bottomRightCol)` will be the coordinates of the corners of the sub-table containing that set.

```
Algorithm 5: Largest Set of Cells with the Same Color
  Input: An $m \times n$ table of characters representing the colors of the cells
  Output: The size of the largest set of cells with the same color and the coordinates of
          the corners of the sub-table containing that set
  maxSize ← 0;
  topLeftRow ← 0;
  topLeftCol ← 0;
  bottomRightRow ← 0;
  bottomRightCol ← 0;
  visited ← $m \times n$ matrix of booleans initialized to false;
  for ( $i \leftarrow 0$ to $m - 1$ ) {
      for ( $j \leftarrow 0$ to $n - 1$ ) {
          if not visited[i][j] then
              size, minRow, minCol, maxRow, maxCol ← DFS(table, visited, i, j);
              if size > maxSize then
                  maxSize ← size;
                  topLeftRow ← minRow;
                  topLeftCol ← minCol;
                  bottomRightRow ← maxRow;
                  bottomRightCol ← maxCol;
              end
          end
      }
  }
  return maxSize, topLeftRow, topLeftCol, bottomRightRow, bottomRightCol;
```

Algorithm 6: Largest Set of Cells with the Same Color

```
Input: An $m \times n$ table of characters representing the colors of the cells, an $m \times n$
       matrix of booleans representing visited cells, and the row and column indices of
       the current cell
Output: The size of the set of cells with the same color that includes the current cell
        and the coordinates of the corners of the sub-table containing that set
visited[row][col] ← true;
size ← 1;
minRow ← row;
maxRow ← row;
minCol ← col;
maxCol ← col;
// Check adjacent cells
if row > 0 and table[row-1][col] = table[row][col] and not visited[row-1][col] then
    resultSize,resultMinRow,resultMinCol,resultMaxRow,resultMaxCol←
      DFS(table,visited,row-1,col);
    size←size+resultSize;
    minRow← min(minRow,resultMinRow);
    maxRow←max(maxRow,resultMaxRow);
    minCol←min(minCol,resultMinCol);
    maxCol←max(maxCol,resultMaxCol);
end
if row < $m-1$ and table[row+1][col] = table[row][col] and not visited[row+1][col] then
    resultSize,resultMinRow,resultMinCol,resultMaxRow,resultMaxCol←
      DFS(table,visited,row+1,col);
    size← size←size+resultSize;
    minRow← min(minRow,resultMinRow);
    maxRow←max(maxRow,resultMaxRow);
    minCol←min(minCol,resultMinCol);
    maxCol←max(maxCol,resultMaxCol);
end
if col > 0 and table[row][col-1] = table[row][col] and not visited[row][col-1] then
    resultSize,resultMinRow,resultMinCol,resultMaxRow,resultMaxCol←
      DFS(table,visited,row,col-1);
    size← size←size+resultSize;
    minRow← min(minRow,resultMinRow);
    maxRow←max(maxRow,resultMaxRow);
    minCol←min(minCol,resultMinCol);
    maxCol←max(maxCol,resultMaxCol);
end
if col < $n-1$ and table[row][col+1] = table[row][col] and not visited[row][col+1] then
    resultSize,resultMinRow,resultMinCol,resultMaxRow,resultMaxCol←
      DFS(table,visited,row,col+1);
    size← size←size+resultSize;
    minRow← min(minRow,resultMinRow);
    maxRow←max(maxRow,resultMaxRow);
    minCol←min(minCol,resultMinCol);
    maxCol←max(maxCol,resultMaxCol);
    return size, minRow, minCol, maxCol;
end
```

The time complexity of the DFS function is O(m*n) because it visits each cell in the table exactly once. The function is called once for each cell in the table, and each call takes constant time because it only checks the adjacent cells (top, bottom, left, right) of the current cell. Since each cell has at most 4 adjacent cells, this operation takes constant time.

Therefore, the overall time complexity of the DFS function is O(m*n), since it visits each cell in the table exactly once and performs a constant-time operation for each cell.

4

5

First we model the last question: we can represent it as a bipartite graph with two sets of vertices: one for devices and one for tasks. We also include a source and a destination vertex. From the source vertex, we connect edges to all device vertices with weights equal to the negative cost of each device. For each task that requires a device, we connect an edge with infinite weight between the task and device vertices to ensure it is not included in the cut. From each task vertex, we connect an edge to the destination vertex with a weight equal to the profit of the task.

This creates a directed graph where we can calculate the maximum flux to find the maximum profit.

Now we reduce the current problem to the last one.

In fact, this problem can be reduced to the previous one by assuming that each task requires two devices. This reduction can be achieved by defining a bipartite graph with vertices in the first part and edges in the second part. An edge with infinite weight is added between a vertex and an edge in the bipartite graph if and only if there is an edge between them in the original graph. The maximum weight will be equal to the total weight of the edges minus the total weight of the vertices.

6