CS 180, Fall 2019
Problem Set 6
Due November 27, 2019

Tian Ye
UID: 704931660

December 4, 2019

Exercise 19 Page 329

Suppose the given interleaved string s has n total characters, and x' as well as y', being the repititions of x and y, also have exactly n characters. We will iterate through the given string s to find the characters that are matching those in x' and y'. If there is a match in x', we will move both x' and s forward by a single character. Likewise for y', we will perform the same action. If there is no match for either, then there is no interleaving and we will return false. We will use the following subproblem: opt[i, j] = true if s[1: i+j] is an interleaving of x'[1:i] and y'[j:1]. If such an interleaving exists, the final character belongs to either x' or y'. Hence we have the following recurrence relation:
opt[i, j = true iff opt[i-1, j] = true and s[i+j]=x'[i], OR iff opt[i, j-1] = true and s[i+j] = y'[j].

```
opt[0,0] = true
For (int k = 1; k<=n; k++)
        For all pairs (i.j) such that i+j = k
                If opt[i-1, j] = true and
                  s[i+j]=x'[i]
                        opt[i,j] = true
                Else if opt[i, j-1] = true and
                  s[i+j]=y'[j]
                        opt[i,j] = true
                Else
                        opt[i,j] = false
Return true iff there exists some pair i,j
 such that i+j = n so that opt[i,j]=true
```

Proof. The algorithm runs in $O(n^2)$ time complexity as there are n^2 values for opt[i,j] to build up. Each takes constant time complexity to fill based on the results of the previous subproblem. □

Exercise 22 Page 330

We will use the notation c_{uv} to denote the cost of the edge e between the two nodes u and v. We will also use the following recurrence relation:
$\text{opt}(i, v) = \min_{u,\,(u,v) \in \text{ E}} \{\text{opt}(i-1, u) + c_{uv}\}$
We will achieve this reccurence by setting the opt of the starting node to 0 and all other nodes to ∞.To get the number of paths, we will utilize a counter that increments every time we observe a path from v to w with the same weight as our current minimum. If a new optimal path is found, we will simply reset the counter to 1. The recurrence is known to provide the minimum path, since it is covered in the textbook.

$\text{opt}(i, v) = \min_{u,\,(u,v) \in \text{ E}} \{\text{opt}(i-1, u) + c_{uv}\}$

```
Initialize array opt[0..n-1][1..n] = inf
opt[0][v] = 0
For (int L = 1; L < n; L++)
        For (int i = 1; i <= n; i++)
                For (int j = 1; j < n; j++)
                        If opt[L-1][x_i] != inf and
                         x_j has an edge to x_i
                                If x_i == w
                                        If opt[L-1][x_i] + w_j = opt[L][x_i]
                                                counter++
                                        Else if opt[L-1][x_i] + w_j < opt[L][x_i]
                                                opt[L][x_i] =  opt[L-1][x_i] + w_j
                                                counter = 1
                                Else
                                        opt[L][x_i] = min(opt[L][x_i],
                                          opt[L-1][x_i]+w_j)
return counter
```

Proof. The algorithm runs in $O(n^3)$ time complexity. There are three nested loops and the operations within the innermost loop are of all constant time complexity. $\square$

Exercise 24 Page 331

We will consider the recurrence relation with three parameters: the number of precincts i, the total number of precinct subsets j, and the total number of votes s. We will build the recurrence where a_i will be the number of votes for party A in the ith precinct, and where s is bound by $\frac{mn}{4} < s <$ votes for A. This yields opt(i, j, s) = true iff(opt(i-1, j-1, s-a_i) or opt(i-1, j, s)); else false, with the base case opt(1, 1, s) = true if s=a_i, else false. This yields us with two choices at the ith precinct: in the former, if the ith precinct had over a_ith votes such that it brought us over the limit of s, then we will simply add the previous $i-1$ precincts and its subsets to get the recurrence relation of opt(i, j, s) = opt(i-1, j-1, s-a_i). In the latter case, the previous precinct was already a majority, we just maintain the result of the previous ith precinct, expressed as opt(i, j, s) = opt(i-1, j, s). Otherwise, it is false. At the very end, we will check for opt(n, n/2, s) to see if there is a true value in that row to see if gerrymandering is possible.

```
Get total votes for A called h
Create opt[1...n][1...][1..m]
opt[1][1][a_1] = True
For (int i = 1; i <= n; i++)
        For (int i = 1; j <=n; j++)
                For (int s = nm/4; s <= h; s++)
                        opt[i][j][s]=opt[i-1][j-1][s-a_i]
                         or opt[i-1][j][s] else False
If there is a true in A[n][n/2], return True
Else, return False
```

Proof. The outermost loop iterates n, the next loop iterates n times, and the innermost loop iterates m times, and the internal operations are all constant, yielding a total time complexity of O(n^2m). □

Exercise 7 Page 417

- Create a source node s and add an edge from it with capacity 1 to every client node c_i
- Create a sink node t and add an edge from each base station b_j to t with capacity L_j
- For each pair (c_i, b_j), if the client is in range r of the base station, add an edge from c_i to b_j with capacity1
- Run Ford-Fulkerson algorithm on the graph, using a DFS for finding the augmenting path in the residual graph.
- If the max flow is n, then every client can be connected to at least one base station.

To prove the correctness of our algorithm, assume there is some optimal flow that our solution could not fine, This would insinuate our algorithm failed to find a max flow that was n; however, there exists a method to do so. This contradicts the Ford-Fulkerson algorithm, which always finds the maximum flow.

```
Create graph G as above
flow = 0
While there is a path from s to t
        DFS to find an augmenting path
        Let bottleneck = minimum capacity edge
        flow += bottleneck
        For each edge uv on the path
                Decrease uv capacity by bottleneck
                Increase vu capacity by bottleneck
If flow == n
        Return True
Return False
```

Proof. The algorithm uses Ford-Fulkerson, with a runtime complexity of O(f(n+m)), where f is the max flow, n is the number of nodes, and m is the number of edges. □

Exercise 9 Page 419

Assume for every patient p there is a node p' and for every hospital h there exists a node h'. An edge will exist between p' and h' if p can reach h within half an hour. The source node s will connect all patients p' with an edge value of 1, and the sink node t will connect to all h' nodes with a value of n/k.

- Create a source node s and add an edge from it with capacity 1 to every patient node p'_i
- Create a sink node t and add an edge from each hospital h'_j to t with capacity n/k
- We will send a 1 flow unit from s to t along the path s, p', h', t whenever the edge (p', h') has at least 1 capacity.
- Run Ford-Fulkerson algorithm on the graph.

To prove the correctness of our algorithm, assume there is some optimal flow that our solution could not fine, This would insinuate our algorithm failed to find a max flow that was n; however, there exists a method to do so. This contradicts the Ford-Fulkerson algorithm, which always finds the maximum flow.

```
Create graph G as above
flow = 0
While there is a path from s to t
        DFS to find an augmenting path
        Let bottleneck = minimum capacity edge
        flow += bottleneck
        For each edge uv on the path
                Decrease uv capacity by bottleneck
                Increase vu capacity by bottleneck
If flow == n
        Return True
Return False
```

Proof. The algorithm uses Ford-Fulkerson, with a runtime complexity of O(f(n+m)), where f is the max flow, n is the number of nodes, and m is the number of edges. □

Exercise 6

We will consider this as two different problems: one will be to find the maximum sequence of of length i, with the last element being an ”up”, defined as optmax(i). The other will be the conversem with the last element being ”down”, defined as optmin(i). We will combine these two recurrence relations to find the longest alternating sequence by taking the max of each index for these recurrence relations. We will build the reccurence relations as follows, given that 0 ¡ j ¡ i:
optmax(i) = max(1 + optmin(j)) where a[j] ¡ a[i]
optmin(i) = min(1 + optmax(j)) where a[j] ¿ a[i]
Our base case will have optmin(i) and optmax(i) = 1. This leaves us with two choices at each iteration: in the case of optmax(i), we consider all the longest alternating sequences before index i that ended in ”down” with an element that has a value that is less than a[i]. If we consider our current index as j, we are left with the condition optmax(i) = max(1 + optmin(j)) where a[j] ¡ a[i]. In the case of optmin(i), we consider all the longest alternating sequences before index i that ended in ”up” with an element that has a value that is greater than a[i]. Again considering our current index as j, we are left with the condition optmin(i) = min(1 + optmax(j)) where a[j] ¿ a[i]. We will combine these two recurrences to take the maximum at each index to find the true longest alternating sequence at index i. To find the actual sequence, we will use two arrays to store the subsequences, which we will return.

```
Create optmin[1...n]
Create optmax[1..n]
Create minsub[1...n]
Create maxsub[1...n]
Set each element of optmin and optmax as 1
Set each element of maxsub and minsub = a[i] for all n
For (int i=1; i<=n; i++)
        For (int j=1; j<i; j++)
                If a[i]>a[j] and optmax[i] < 1 + optmin[j]
                        optmax[i] = 1 + optmin[j]
                        maxsub[i] = minsub[j].add(a[i])
                Else if a[i]<a[j] and optmin[i] < 1 + optmax[j]
                        optmin[i] = max(optmin[i], 1+optmax[j])
                        minsub[i] = maxsub[j].add(a[i])
```

```
For (int i=1; i<=n; i++)
        If optmin[i] > optmax[i]
                optmax[i] = optmin[i]
                maxsub[i] = minsub[i]
Return max subsequence
```

Proof. The outer loop iterates at most n times, as does the inner loop, while the internal operations are constant. This is the case of greatest loop nesting, hence the overall time complexity is $O(n^2)$. $\square$