Michael Inoue
405171527

Ling
Discussion 1C 2-3:50PM

CS 180

Homework 6

27 November 2019

1 Solution

Algorithm 1: Determine if string s is an interleaving of strings x and y

```
   Input  : Strings s, x, y
   Output: Determination yes or no
 1 if |x| + |y| > n then
 2 |  return no
 3 end
 4 2D array A is empty
 5 Extend x to be a repetition of x in an n-length string
 6 Extend y to be a repetition of y in an n-length string
 7 A[0][0] = yes
 8 for 1 ≤ k ≤ n do
 9 |  for all i, j such that i + j = k do
10 |  |  if A[i-1][j] = yes and s[i+j] = x[i] then
11 |  |  |  A[i][j] = yes
12 |  |  else if A[i][j-1] = yes and s[i+j] = y[i] then
13 |  |  |  A[i][j] = yes
14 |  |  else
15 |  |  |  A[i][j] = no
16 |  end
17 end
18 for all i, j such that i + j = n do
19 |  if A[i][j] = yes then
20 |  |  return yes
21 |  end
22 end
23 return false
```

We will first prove the correctness of our algorithm. We first check the length of our strings, x and y. If their sum is longer than the length of our string s, then it is impossible for them to form an interleaving of s, so we must return *no* in this case.

Next, we extend our strings, x and y so that they form a string of length n (by simply adding their repetitions over and over until we reach the nth character, at which point, we cut their strings off). The motivation for this is to account for repeating periods of x and y without needing a counter variable to keep track of it. Our results will be the same regardless, as we have effectively 'wrapped' around our periods into one big sequence.

Now, we may consider the following fact: for any subsequence of our string s, $(1, i + j)$ starting at the 1st character and ending at the $i + jth$ character, if A[i][j] is the answer as to whether such a subsequence contains a valid interleaving of our sequences x and y up to the $i + jth$ character, then A[i][j] = yes if A[i-1][j] is yes and A[i+j] = x[i] or A[i][j] = yes if A[i][j-1] is yes and A[i+j] = y[j].

Now, our algorithm accomplishes this utilizing dynamic programming, performing an organized, exhaustive search by searching through each pair (i, j) such that $i + j = k$ to build up a set of solutions to smaller subproblems that may be used in further iterations.

Indeed, if there is a valid interleaving for our sequence $1, n$ (the length of our string, s), we must have that by the nth iteration of our outer for loop, that some pair (i, j) such that $i + j = n$ satisfies that A[i][j] = *yes*. Now, our algorithm accounts for this, checking for such a pair at line (18).

Thus, we conclude that our algorithm yields correct results.

Time Complexity:

Our algorithm utilizes two for loops, an inner loop and an outer loop. The inner loop iterates at most n times, and our outer loop iterates for the number of pairs i, j such that $i + j = k$. Applying the pigeonhole principle, this means it iterates at most $\frac{k}{2}$ times. Now, k can be at most n, so this loop will iterate at most $\frac{n}{2}$ times in each iteration. Each inner loop iteration performs a set of constant operation, so this portion of our algorithm runs in $O((n)(\frac{n}{2})) = O(n^2)$

Next, our algorithm checks for the pairs i, j such that $i + j = n$ and A[i][j] = yes. As mentioned before, there are at most $\frac{n}{2}$ such pairs, so this portion of the algorithm runs in $O(n)$ time.

Thus, we have our total time complexity, $T(n)$:

$$T(n) = O(n^2) + O(n) \tag{1}$$
$$= O(n^2) \tag{2}$$

2 Solution

Algorithm 2: Computes the number of shortest paths from a vertex v to a vertex w

Input : Graph G with vertices V and edges E
Output: number of shortest paths from vertex v to vertex w

```
1  Initially array of layer of nodes L is empty
2  Initially shortest path to every node a, a_sp = −∞
3  Initially number of shortest paths to every node a is 0
4  Set v_sp = 0
5  Set v_n = 1
6  Add v to layer L[1]
7  foreach i = 1, 2, ... do
8      foreach node a ∈ L[i] do
9          foreach node b connected to a by an edge e do
10             Add b to layer L[i+1]
11             if a_sp + c_e = b_sp then
12                 Increment b_n by 1
13             end
14             if a_sp + c_e < b_sp then
15                 Set b_n = a_n
16                 b_sp = a_sp + c_e
17             end
18         end
19     end
20 end
21 return w_n
```

We will first prove the correctness of our algorithm. Our algorithm simply performs a modified version of the shortest-path algorithm with negative lengths demonstrated in class. The only addition is that every node b keeps track of the number of shortest paths to b, which is held in a variable b_n. Indeed, if we demonstrate that b_n always maintains the number of shortest paths to b, then we will be finished, as then, w_n will contain the number of shortest paths to w when the algorithm finishes.

Now, we will prove by induction. For our base case, we have $k = 1$ node. We only have our starting node in this case, v. Our algorithm will hence return $v_n = 1$, which is correct, as the number of shortest paths from v to itself is 1.

Now, for our inductive step, assume that for every node b at a layer $k \leq i - 1$ that b_n is the number of shortest paths from v to b. We want to show that for a node d at layer i, d_n is the number of shortest paths from v to d. Now, we know that a node d must be connected to a node at some layer $l \leq i - 1$. Moreover, there must be some minimum node m such that the path from m to d is the shortest path. There may be multiple such nodes, which we will account for.

Now, our algorithm finds such a node m in line (7), by iterating through every node in a previous layer and finding the one which creates the shortest path to d. Now, if the path from m, p is the shortest path, then there was either a path discovered to d already, or there isn't. If there is, then we may compare that path, call it p', to our path p. If $|p| = |p'|$, then we have discovered another shortest path to d. Hence, we should increment the number of shortest paths, d_n by 1, which our algorithm accounts for in lines (10-11). In this case, our algorithm must be optimal, as m is a shortest path, and we compute this for all such m, which, by our inductive hypothesis, yields a shortest path.

If there has been no path discovered to d, then we simply set the number of paths d_n equal to the number of shortest paths to our path m. As m_n contains the number of shortest paths to m, our number of shortest paths d_n is optimal.

Hence, we conclude that our algorithm yields the number of shortest paths to w and is optimal.

Time Complexity:

Now, our algorithm utilizes 3 for loops. The outermost loop iterates through a counter variable, i which corresponds to the node layer we are iterating through. There may be n such layers, so this iterates at most n times. In the second loop, we consider each node in our particular layer, L[i]. There may be n such nodes in each layer, so this iterates at most n times. Finally, our innermost loop iterates through each node connected to a particular node in our layer. There may be n such nodes such to our current node, so this iterates at most n times. Now, the body of our innermost loop runs in constant time, performing a constant number of comparisons and updates to our nodes. Hence, our total time complexity is $O((n)(n)(n)) = O(n^3)$.

3 Solution

Algorithm 3: Determines whether precincts are susceptible to gerrymandering

Input : Precincts $P_1, P_2, ..., P_n$ with m voters and votes for A, $P_1[A], P_2[A], ..., P_n[A]$ and total votes for A and B, $P_1[AB], ..., P_n[AB]$

Output: Determination if the precincts are susceptible to gerrymandering

```
Initially a 4D array A is empty
for i = 1, 2, ..., n do
    for j = 1; j ≤ i; j++ do
        for a = 1, 2, ..., m do
            for b = 1, 2, ..., m do
                if a + b > m then
                    A[i][j][a][b] = true
                end
                if i − j = n/2 then
                    if nm/4 ≤ Pj[A] + ... + Pi[A] ≤ a − nm/4 − 1 then
                        A[i][j][a][b] = true
                    else
                        A[i][j][a][b] = false
                    end
                else
                    A[i][j][a][b] = false
                end
                c = P_i[AB]
                if A[i-1][j-1][a-c][b] = true then
                    A[i][j][a][b] = true
                else
                    A[i][j][a][b] = false
                end
                if A[i-1][j-1][a][b-c] = true then
                    A[i][j][a][b] = true
                else
                    A[i][j][a][b] = false
                end
            end
        end
    end
end
for a = 1, 2, ..., m do
    for b = 1, 2, ..., m do
        if A[n][n/2][a][b] = true then
            return true
        end
    end
end
return false
```

We will first prove the correctness of our algorithm. We will use the following notation: for an array element A[i][j][a][b], we say that the current precinct is precinct i, that j is the number of precincts allocated from the first i precincts, that a is the number of votes for A and that b is the number of votes for b.

Now, we utilize the following recurrence relation. We note that if j of the first i precincts has a majority, it must follow that $j-1$ of the first $i-1$ precincts also must have a majority without the votes contributed from precinct i. More formally, if a precinct i has c votes for party A, then A[i][j][a][b] is true if A[i-1][j-1][a-c][b] is true. We take the analogous case for if a precinct i has c votes for party B: A[i][j][a][b] is true if A[i-1][j-1][a][b-c] is true.

Now, our recurrence relationship is correct, but we must check our base cases and verify that they are correct as well. We have the following base cases:

1.) If $a + b > m$, then A[i][j][a][b] is false

2.) If $j \leq \frac{n}{2} = m$ and $\frac{nm}{4} \leq P_j[A] + ... + Pi[A] \leq a - \frac{nm}{4} - 1$ then A[i][j][a][b] is true, otherwise false

Now, to prove (1.), we simply note that the total number of votes for A and B cannot exceed m, the total number of votes. Hence, we denote that such a voting setup is impossible and thus false.

Now, to prove (2.), we note that the total number of precincts from precinct j to precinct i must be $\frac{n}{2}$ exactly, according to the setup of our problem (we need to have exact $\frac{n}{2}$ precincts in each district. Hence, we must have $i - j = \frac{n}{2}$. Moreover, we observe that these precincts must be be between $\frac{nm}{4}$ and $a - \frac{nm}{4} - 1$. The lower bound is because we will have nm voters total, and if we want a majority in one district, we need $\frac{nm}{4}$ voters minimum. Now, the upper bound is because we cannot exceed the total number of votes for party A with our votes taken away from our one district, as we need a majority in both districts. Hence, we must have that $P_j[A] + ... + Pi[A] \leq a - \frac{nm}{4} - 1$.

So our base case and recurrence relations hold, so for any of the j precincts between i and j, if there the district is susceptible to gerrymandering, then A[i][j][a][b] is true for some a, b. Now, in our original problem, we have n precincts and we will use $\frac{n}{2}$ of them to create the first district. This will leave us with the fact that if the district is susceptible to gerrymandering, then A[n][$\frac{n}{2}$][a][b] is true for some a, b. But our algorithm checks for this in lines (32-37), returning true if such a, b exist, and false otherwise.

Hence, we have that our algorithm yields correct results.

Time Complexity:

In the first part of our algorithm, we have four for loops. The first iterates at most n times, the second iterates at most i times, and as i can be at most n, iterates at most n times, and the third and fourth iterate at most m times each. The body of the innermost loop runs in constant time, performing a constant number of updates and comparisons. Thus, we have that this portion of our algorithm runs in $O((n)(n)(m)(m))$ or $O(n^2m^2)$.

In the next part of our algorithm, we utilize two for loops which iterate at most m times each. The body of our innermost loop runs in constant time, so this portion of the algorithm runs in $O((m)(m)) = O(m^2)$. Thus, adding up the time complexities of both components of our algorithm, we have our total time complexity, $T(n)$:

$$T(n) = O(n^2m^2) + O(m^2) \tag{3}$$

$$= O(n^2m^2) \tag{4}$$

4 Solution

Algorithm 4: Determines if every client can be paired with a base station

Input : Clients $c_1, ..., c_n$ with ranges r, base stations $b_1, ..., b_k$ with load capacities L
Output: Whether each client may be paired with a base station

```
1  Create a super source s and super sink t
2  Each c_i (i = 1, ..., n) is a node with an edge of capacity ∞ coming from s
3  Each b_j (j = 1, ..., k) is a node with an edge of capacity L coming to t
4  foreach pair (c_i, b_j) for i=1,...,n, j=1,...,k do
5  |  if dist(c_i, b_j) < c_i 's range r then
6  |  |  Add an edge from c_i to b_j of capacity 1
7  |  end
8  end
9  Perform Ford-Fulkerson Algorithm on constructed graph to find max flow f from s to t
10 if f = n then
11 |  return true
12 else
13 |  return false
14 end
```

We will now prove the correctness of our algorithm. We will convert our given problem into a variant of a network flow problem. Let us treat each client c_i and base stations b_j as nodes; moreover, let us connect each c_i to our super source node s with an edge capacity of ∞ and each b_j to our super sink node t with an edge capacity of L.

Now, we see that if more than L clients are connected to a particular base station b_j, then b_j faces a bottleneck, as it may only distribute a flow of at most L to s; this accounts for the the fact that at most L clients can be connected to b_j in our original problem, so our new problem is analogous in this aspect.

Moreover, each client c_i can only send a unit of flow over to a base station b_j if $dist(c_i, b_j) < c_i' sranger$, which accounts for a fact that a client may only be connected to a base station if it is within a range r. Hence, this component of the algorithm is analogous.

Finally, if our max flow is n, then that implies that n clients have sent a unit of flow through a base station. As Ford-Fulkerson was shown to be optimal in lecture, that means that our algorithm indeed finds the max flow, and further, if such a flow is equal to n, then we have achieved the maximum number of clients that can be connected, or equivalently, each client is connected to a base station (we accomplish this in lines 9-11). This component is entirely analogous, and hence we observe that our algorithm correctly solves the original problem given.

Thus, our algorithm yields correct results and is optimal.

Time Complexity:

First our algorithm constructs a graph. It first adds an edge to each client c_i and s and each base station b_j and t. There are n clients and k base stations, so this takes $O(n + k)$ time.

Next, our algorithm checks each pair (c_i, b_j) for $i = 1, ..., n, j = 1, ..., k$, adding an edge between c_i and b_j if the distance between them is less than c_i's range r. There are nk such pairs, so this takes $O(nk)$ time.

Next, our algorithm performs Ford-Fulkerson on our constructed graph. In lecture, we proved this to take $O(fe)$ time, where f is the max flow, and e is the number of edges in our graph. Our graph has nk edges (as mentioned in the previous paragraph), so this portion of our algorithm runs in $O(fnk)$.

Thus, adding up our time complexities, we have our total time complexity, $T(n)$:

$$T(n) = O(n+k) + O(nk) + O(fnk) \tag{5}$$

$$= O(fnk) \tag{6}$$

5 Solution

Algorithm 5: Determines if every person can be sent to a hospital such that every hospital is balanced

Input : Injured people $i_1, ..., i_n$ with current location c, base stations $h_1, ..., h_k$ with capacity $\lceil \frac{n}{k} \rceil$
Output: Whether each person can be sent to a hospital such that every hospital is balanced

```
1  Create a super source s and super sink t
2  Each i_a (a = 1, ..., n) is a node with an edge of capacity ∞ coming from s
3  Each h_b (b = 1, ..., k) is a node with an edge of capacity ⌈n/k⌉ coming to t
4  foreach pair (i_a, h_b) for a=1,...,n, b=1,...,k do
5  |  if i_a is within a 30 minute drive from h_b then
6  |  |  Add an edge from i_a to h_b of capacity 1
7  |  end
8  end
9  Perform Ford-Fulkerson Algorithm on constructed graph to find max flow f from s to t
10 if f = n then
11 |  return true
12 else
13 |  return false
14 end
```

We will now prove the correctness of our algorithm. We will convert our given problem into a variant of a network flow problem. Let us treat each injurent patient i_a and hospital h_b as nodes; moreover, let us connect each i_a to our super source node s with an edge capacity of ∞ and each h_b to our super sink node t with an edge capacity of $\lceil \frac{n}{k} \rceil$.

Now, we see that if more than $\lceil \frac{n}{k} \rceil$ injured patients are admitted to a particular hospital h_b, then h_b faces a bottleneck, as it may only distribute a flow of at most $\lceil \frac{n}{k} \rceil$ to s; this accounts for the the fact that at most $\lceil \frac{n}{k} \rceil$ clients can be connected to h_b in our original problem, so our new problem is analogous in this aspect.

Moreover, each patient i_a can only send a unit of flow over to a hospital h_b if i_a is within a 30 minute drive from h_b, which accounts for a fact that a patient will only attend a hospital if it is within a 30 minute driving distance. Hence, this component of the algorithm is analogous.

Finally, if our max flow is n, then that implies that n clients have sent a unit of flow through a base station. As Ford-Fulkerson was shown to be optimal in lecture, that means that our algorithm indeed finds the max flow, and further, if such a flow is equal to n, then we have achieved the maximum number of injured patients that can be admitted, or equivalently, each patient is admitted to a hospital (we accomplish this in lines 9-11). This component is entirely analogous, and hence we observe that our algorithm correctly solves the original problem given.

Thus, our algorithm yields correct results and is optimal.

Time Complexity:

First our algorithm constructs a graph. It first adds an edge to each injured patient i_a and s and each hospital h_b and t. There are n patients and k hospitals, so this takes $O(n + k)$ time.

Next, our algorithm checks each pair (i_a, h_b) for $a = 1, ..., n, b = 1, ..., k$, adding an edge between i_a and h_b if the i_a is within a 30 minute drive from h_b. There are nk such pairs, so this takes $O(nk)$ time.

Next, our algorithm performs Ford-Fulkerson on our constructed graph. In lecture, we proved this to take $O(fe)$ time, where f is the max flow, and e is the number of edges in our graph. Our graph has nk

edges (as mentioned in the previous paragraph), so this portion of our algorithm runs in $O(fnk)$.

Thus, adding up our time complexities, we have our total time complexity, $T(n)$:

$$T(n) = O(n+k) + O(nk) + O(fnk) \tag{7}$$

$$= O(fnk) \tag{8}$$

6 Solution

Algorithm 6: Find the longest alternating subsequence

Input : Sequence of numbers (array) S
Output: Longest alternating subsequence

```
Initially an nxn 2D array A is empty
A[0][0] = 0
A[0][1] = 0
Global variable globalmax = −∞
Global variable maxindex = 0
Global variable x = 0
for i = 1, ..., n do
    A[i][0] = 1
    A[i][1] = 1
    for j = 0, ..., i − 1 do
        if S[i] > S[j] then
            A[i][0] = max(A[j][1] + 1, A[i][0])
        end
        if S[i] < S[j] then
            A[i][1] = max(A[j][0] + 1, A[i][1])
        end
    end
    if globalmax < A[i][0] then
        maxindex = i
        globalmax = A[i][0]
        x = 0
    end
    if globalmax < A[i][1] then
        maxindex = i
        globalmax = A[i][1]
        x = 1
    end
end
for k = maxindex − A[maxindex][x]; k ≤ maxindex; k++ do
    Output S[k]
end
```

We will now prove the correctness of our algorithm. Our algorithm utilizes dynamic programming to store the longest subsequence of a number at a given index. It relies upon the following facts:

1.) For $i = 1, ..., n$, A[i][0] is the longest alternating sequence ending at the index i with its last element being greater than its previous element, A[i][1] is the longest alternating sequence ending at the index i with its last element being being smaller than its previous element.

2.) Each number is an alternating sequence of length 1

3.) If we are at position i and some position $j < i$, we may extend our sequence from j to i according to the following recurrence relation:

A[i][0] = max(A[i][0], A[j][1] + 1) if S[j] < S[i] (extending our subsequence ending at j by at most 1 or leaving it alone)

A[i][1] = max(A[i][1], A[j][1] + 1) if S[j] > S[i] (extending our subsequence ending at j by at most 1 or leaving it alone)

Now, we observe that using recurrence relation (3.), we may choose a $j < i$ such that the length of our sequence up to i, A[i][0] and A[i][1], is maximized. Indeed, there must be some subsequence prior to these indices, such that A[i][0] < A[j][0] + 1 and A[i][1] < A[j][1]. But, our algorithm accounts for this by updating A[i][0] and A[i][1] once it encounters a j that satisfies increases our sequence. Hence, as our recurrence relation is valid, and our base case (fact (1.)) is valid, we have that for any index i, A[i][0] and A[i][1] contain the length of the longest subsequences ending at the index i with its last element being greater than its previous element and its last element being smaller than its previous element, respectively.

Now, this means that for some i, either A[i][0] or A[i][1] is the longest subsequence. But, our algorithm accounts for this, checking if the current subsequence is longer than the longest subsequence, *globalmax*, and if so, keeps track of its index, i in a variable *maxindex* and whether the sequence belonged to A[i][0] or A[i][1] in our variable x. Now, as our algorithm updates these values if it finds a greater subsequence ending at index i, we may assume that *maxindex*, *globalmax* and x will have the appropriate values by the end of our for loop.

Now, we simply output the subsequence. We observe that the length of our sequence is stored in A[*maxindex*][x]. Hence, our output must contain every element between *maxindex* - A[*maxindex*][x] and *maxindex*. But, our algorithm accomplishes this in lines 29-20 in our last for loop.

Hence, we conclude that our algorithm yields correct results.

Time Complexity:

We first utilize two for loops, an outer loop and an inner loop. The outer loop iterates at most n times, and performs constant time operations (updating our array A) between the outer loop and the inner loop. Our inner loop iterates at most i times, and as i can be at most n, our inner loop iterates at most n times. The body of our loop runs in constant time, performing a constant number of comparisons and updates to our global variables and our array A. Hence, this portion of our algorithm runs in $O((n)(n)) = O(n^2)$.

Next, our algorithm outputs the longest subsequence of our array utilizing a for loop iterating from *maxindex* - A[*maxindex*][x] to *maxindex*. *maxindex* can be at most n, and A[*maxindex*][x] is at least 0, so this loop iterates at most n times. The body of the loop runs in constant time, so in total, this portion of our algorithm runs in $O(n)$.

Adding up our time complexities, we have our total time complexity, $T(n)$:

$$T(n) = O(n^2) + O(n) \tag{9}$$

$$= O(n^2) \tag{10}$$