CS180 Homework 2

Zhehao Wang 404380075 (Dis 1B)

Apr 10, 2016

1 Number of inversions remains unchanged for any permutation

Proof by induction on the number of pair swaps in the permutation: we denote the original lists as $A = a_1...a_n$ and $B = b_1...b_n$. Each permutation consists of a number of pair swaps for songs in both list A and B. We call a pair (a_i, a_j) *flipped* if it used to be (a_i, a_j) before the permutation, and becomes (a_j, a_i) after the permutation, and the songs a_i, a_j to be *involved* in the flip.

Base case: consider the case where only one pair in both A and B is swapped in the permutation. Denote the pair as (a_i, a_j) in the original list A, where $i < j$. We can find the same songs in B where $a_i = b_m$ and $a_j = b_n$. The sequence in B could be (b_m, b_n), or (b_n, b_m). Consider the (b_m, b_n) where $m < n$ case first:

The *flipped* pairs in A caused by this permutation include: (a_i, a_j), (a_p, a_j) and (a_i, a_p), where $i < p < j$. Similarly, *flipped* pair in B include: (b_m, b_n), (b_q, b_n) and (b_m, b_q), where $m < q < n$. Since $a_i = b_m$ and $a_j = b_n$, number of inversions is not changed by A and B both having the (a_i, a_j), (b_m, b_n) flips. Thus we consider each a_p and b_q *involved* in the flip, and the total number of inversions does not change if each *involved* a_p and b_q do not cause changes in the total number of inversions. Case analysis on the position of each b_l in B where each $b_l = a_p$.

- If $l < m$, then list B used to have (b_l, b_m) and (b_l, b_n), list A used to have (a_i, a_p) and (a_p, a_j), number of inversions used to be 1. After the permutation, B still has (b_l, b_m) and (b_l, b_n), and A has (a_p, a_i), (a_j, a_p). Number of inversions is still 1.
- If $l > n$, the case is similar with above. The number of inversions before and after the permutation are both 1.
- If $m < l < n$, then B used to have (b_m, b_l) and (b_l, b_n), A used to have (a_i, a_p) and (a_p, a_j), and number of inversions used to be 0. After the permutation, B has (b_l, b_m) and (b_n, b_l), A has (a_p, a_i) and (a_j, a_p). The number of inversions is still 0.

Similar case analysis can be done for each a_k in list A where each $a_k = b_q$. We have the number of inversions does not change when B's sequence is (b_m, b_n).

Similar case analysis can be done when B's sequence is (b_n, b_m), with the only difference being that in case 3 ($n < l < m$), the number of inversions before and after the permutation are both 2 instead of 0. Thus to summarize, we have the number of inversions does not change when only one pair is swapped in the permutation.

Induction case: assume that the conclusion holds for any permutation involving n pair swaps. For any permutation involving $n + 1$ pair swaps, by the induction hypothesis, we know that the conclusion holds for its sub-permutation with one pair excluded. By applying the analysis of the base case on the results of the sub-permutation, we know that the conclusion holds for any permutations involving $n + 1$ pair swaps as well.

2 Number of intersection and inversions

(a)

Proof: an inversion is defined by a pair (i, j) such that q_i is before q_j in list q, but p_i is after p_j in list p.

For each pair (i, j), consider the sequences of p_i, p_j and q_i, q_j, and the lines from p_i to q_i, and from p_j to q_j.

- If there's an intersection between the two lines, the sequences of p_i, p_j and q_i, q_j have to be different in lists q and p, as illustrated in figure 1. Thus for each intersection, there is at least one corresponding inversion. Total number of inversions $\geq$ Total number of intersections.
- If the sequences of p_i, p_j and q_i, q_j are different in list q and p, there has to be an intersection between the lines $(p_i, q_i)$and (p_j, q_j), as illustrated in figure 1. Thus for each inversion, there is at least one corresponding intersection. Total number of intersections $\geq$ Total number of inversions.

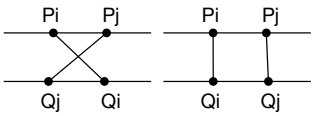

Figure 1: Two-node cases of inversion vs intersection

Summarizing the two cases, we have Total number of intersections = Total number of inversions, thus the theorem holds.

(b)

Given the conclusion in Problem 1, permutations do not cause the number of inversions to change, this algorithm uses one list as the standard, takes in the other list, and calculates its number of inversions when compared against the standard. The algorithm's given in alg. 1.

Time complexity: this algorithm adds a constant-time inversion count to a recursive merge sort, thus the complexity is the same as merge sort, which is $O(n \log n)$. In other words, the above described algorithm satisfies $T(n) = 2 \cdot T(\frac{n}{2}) + n$, by Master Theorem case 2, we have $T(n) = O(n \log n)$.

Correctness: this algorithm takes over after both arrays are permutated, such that one array is sorted and considered a standard array, and the other array is given as input for counting the number of inversions.

This algorithm uses divide and conquer: each time we split the input array into two, if we assume the two sub arrays are sorted and already have their number of inversions counted, the number of inversions of the merged array is the sum of the number of inversions of the two sub arrays, plus the number of inversions introduced by the merge, which is $\sum_{i=1}^{n} a_i$, where n is the number of elements in the right array, and a_i is the left array's number of elements that are to the right of the i^th element in the right array, if that element were to be inserted in the left array.

3 Celebrity iterative

Given an incident matrix representation of the graph, the algorithm's given in alg. 2. The algorithm takes in an $n \times n$ incident matrix (whose *len* is defined as n; and $matrix[i][j] = 1$ means person i knows person j), and returns the index of the celebrity if there's one, otherwise returns -1. Matrix with less than two persons is considered to not have celebrities.

Algorithm 1 Number of inversions for one list against a permutated standard

```
1: function numberOfInversions(array)
2:     n ← len(array)
3:     if n < 2 then
4:         return 0
5:     l1 ← numberOfInversions(array[0]..array[n/2])
6:     l2 ← numberOfInversions(array[n/2 + 1]..array[n])
7:     return l1 + l2 + countInversions(array, array[0]..array[n/2], array[n/2 + 1]..array[n])
8: function countInversions(original,l1,l2)
9:     inversions ← 0
10:    n ← len(l1)
11:    pos ← 0
12:    while l1.hasNext() ∨ l2.hasNext() do
13:        if l1.hasNext() = false then
14:            original[pos] ← l2.next()
15:        else if l2.hasNext() = false then
16:            original[pos] ← l1.next()
17:        else if l1.next() < l2.next() then
18:            original[pos] ← l1.next()
19:        else
20:            original[pos] ← l2.next()
21:            i ← n − l2.next()'s position in l1
22:            inversions ← inversions + i
23:        pos ← pos + 1
24:    return inversions
```

(Minor detail: the following algorithm assumes incident matrix index starts from 0)

Time complexity: this algorithm is $O(n)$, where n is the number of persons. In each execution of while, we get rid of at least one person from an arbitrary pair. So the loop's executed at most n times, and each time it executes a constant number of operations. The check for whether the last remaining person is a celebrity or not is also $O(n)$, so the overall algorithm is $O(n)$.

Correctness: as discussed in class, there is at most one celebrity in a group of people. So each time the algorithm takes a pair of persons, see if they know each other, and remove the one that's definitely not a celebrity (or remove both if neither is a celebrity), and move on until there's only one person left. We then check if that person is a celebrity and returns the result.

4 Diameter of tree

(a)

Define the **height** of a rooted directed tree as the number of edges on the longest path from the root to a leaf. Algorithm is given in Alg 3.

This recursive algorithm takes in the root of a tree and produces the height of the tree, by each time removing the root and finding the maximum height among all resulting sub trees. The diameter of the tree would be the sum of the heights of two highest subtrees. Initial call to the algorithm should look like $findHeightOrDiameter(root, false, nil)$. This algorithm is $O(n)$, where n is the number of nodes in the tree, because each node in the tree will be visited exactly once.

Algorithm 2 Celebrity iterative

```
1: function hasCelebrity(matrix)
2:     l ← len(matrix)
3:     if  thenl < 2 return −1
4:     i ← 0
5:     j ← 1
6:     while  domax(i, j) < l
7:         if matrix[i][j] = 1 ∧ matrix[j][i] = 0 then
8:             i ← max(i, j) + 1
9:         else if matrix[j][i] = 1 ∧ matrix[i][j] = 0 then
10:            j ← max(i, j) + 1
11:        else
12:            i ← max(i, j) + 1
13:            j ← max(i, j) + 1
14:    if min(i, j) >= l then
15:        return −1
16:    if i ≥ l then
17:        return isCelebrity(matrix, j)
18:    else
19:        return isCelebrity(matrix, i)
20: function isCelebrity(matrix, i)
21:    for person p in matrix, p ≠ i do
22:        if matrix[p][i] = 0 ∨ matrix[i][p] = 1 then
23:            return −1
24:    return i
```

(b)

The iterative version of the algorithm is given in Alg 4.

This algorithm is $O(n)$, where n is the number of nodes in the tree, because each node in the tree will be visited exactly once.

Algorithm 3 Diameter of a rooted directed tree's underlying undirected tree, recursive

```
1: function findHeightOrDiameter(root, findHeight, prevRoot)
2:     if degree(root) = 1 then
3:         return 0
4:     heights ← []
5:     for each {n|n ∈ V, (n, root) ∈ E, n ≠ prevRoot} do
6:         heights.push(1 + findHeightOrDiameter(n, true, root))
7:     if findHeight then
8:         return max(heights)
9:     else
10:        return max(heights) + 2nd highest(heights)
```

Algorithm 4 Diameter of a rooted directed tree's underlying undirected tree, iterative

```
1: function findHeightOrDiameter(root)
2:     queue ← [root]
3:     height0 ← 0
4:     height1 ← 0
5:     while True do
6:         nodeCount ← queue.size()
7:         if nodeCount = 0 then
8:             return height0 + height1
9:         height ← height + 1
10:        while nodeCount > 0 do
11:            r ← queue.dequeue()
12:            r.visited ← true
13:            if degree(r) = 1 then
14:                if height > height0 then
15:                    height1 ← height0
16:                    height0 ← height
17:                else if height > height1 then
18:                    height1 ← height
19:            else
20:                for each {n|n ∈ V, (n, r) ∈ E, n.visited = false} do
21:                    queue.enqueue(n)
22:            nodeCount ← nodeCount − 1
```