CS180 Homework 2

Zhehao Wang 404380075 (Dis 1B)

Apr 10, 2016

1 Number of inversions remains unchanged for any permutation

Lemma: a permutation consists of a set of pair swaps in both list A and B. A pair swap is defined as the sequence used to be (a_i, a_j) in list A, and (b_m, b_n) (or (b_n, b_m)) in list B, where $a_i = b_m, a_j = b_n$, and becomes (a_j, a_i) in list A, and (b_n, b_m) (or (b_m, b_n)) after the swap.

Proof by induction on the number of pair swaps in the permutation: we denote the original lists as $A = a_1...a_n$ and $B = b_1...b_n$.

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 $i < j$, and (b_m, b_n) (or (b_n, b_m)) in list B. 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 the remaining flips involving a_p and b_q, and if each of them does not cause changes in the number of inversions between A and B, then the total number of inversions remains the same. For each a_p, we find a corresponding b_l where $b_l = a_p$, and do case analysis on the position of b_l.

- 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.

Similarly for each b_q, we find a corresponding a_k where $a_k = b_q$, and do case analysis on the position of a_k.

Summarizing both cases, we have the total number of inversions remains the same when A swaps (a_i, a_j) and B's original sequence is (b_m, b_n).

Similar case analysis can be done when B's original 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.

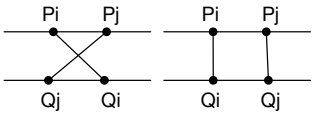

Figure 1: Two-node cases of inversion vs intersection

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

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
```

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.

(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.

Correctness: starting from the root in the tree, diameter is the maximum among the diameters of its

Algorithm 2 Celebrity iterative

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

sub-trees, and the sum of heights of its two highest sub-trees plus two. Assuming the diameters of its sub-trees are already found, we combine this algorithm with *tree-height* algorithm from last homework, and produce the largest among these values as the diameter of the tree.

Time complexity: this algorithm is $O(n \log n)$, where n is the number of nodes in the tree. This algorithm satisfies $T(n, i) = \sum_{i'|(root,i') \in E} T(n', i') + n$, where n is the number of nodes that are lower than node i, and n' is the number of nodes that are lower than node i'. The $+n$ comes from the cost of *findHeight*, which gets called each time we recursively call *findDiameter*. For simplicity, assuming it's a full tree with fanout k, we have $T(n) = k \cdot \sum_{T(\frac{n}{k})} + n$. And by Master Theorem case 2, this algorithm is $O(n \log n)$.

(b)

The iterative version of the algorithm is given in alg 4. Starting from a given root, this algorithm first does a BFS traversal. The last dequeued node is then used as a starting point for the same *findHeight* algorithm used in part (a) and the previous homework.

Correctness: starting from the root r, after the first BFS, a leaf node n who's farthest away from the r is returned; we want to prove that starting from n, we can find the diameter. Idea of the proof centers around the property: for any pair of nodes (a, b), there exists only one path (a, b).

Assume we have diameter $D = d(s, e)$, case analysis on whether the path (s, e) and path (r, n) share edges or not.

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

```
1: function findDiameter(root)
2:     heights ← []
3:     diameter ← 0
4:     diameters ← []
5:     for each {n|n ∈ V, (n, root) ∈ E} do
6:         heights.push(findHeight(n))
7:         diameters.push(findDiameter(n))
8:     if len(heights) = 0 then
9:         diameter ← 0
10:    else if len(heights) = 1 then
11:        diameter ← heights[0] + 1
12:    else
13:        diameter ← 2 + max(heights) + 2^nd heighest(heights)
14:    return max(diameter, diameters)
15: function findHeight(root)
16:    queue ← [root]
17:    height ← 0
18:    while True do
19:        nodeCount ← queue.size()
20:        if nodeCount = 0 then
21:            return height
22:        height ← height + 1
23:        while nodeCount > 0 do
24:            r ← queue.dequeue()
25:            r.visited ← true
26:            for each {n|n ∈ V, (n, r) ∈ E, n.visited = false} do
27:                queue.enqueue(n)
28:            nodeCount ← nodeCount − 1
```

- Path (s, e) and path (r, n) do not share edges. Path (s, n) should then contain node r. (Otherwise assume that (s, n) and (r, n) first converges at $l, l \neq r$, we have $d(r, n) > d(l, n) \geq d(l, e) \geq d(s, e) = D$, a contradiction with the definition of diameter). Similarly, path (e, n) should contain node r, and node r is on path (s, e). Since $d(r, n) \geq max(d(r, s), d(r, e))$, we have $max(d(n, s), d(n, e)) = d(r, n) + max(d(r, s), d(r, e)) \geq d(s, e) = D$. Then we can always find the diameter starting from n.
- Path (s, e) and path (r, n) share a path (a, b). Assume that a being closer to s, and b being e. We have $d(a, n) \geq d(a, e)$, $d(a, n) \geq d(a, s)$, $d(b, n) \geq d(b, e)$, $d(b, n) \geq d(b, s)$. Thus in case of a being closer to n than b is, we have $D = d(s, e) = d(s, a) + d(a, b) + d(b, e) \leq d(n, a) + d(a, b) + d(b, e) = d(n, e)$; otherwise we have $D = d(s, e) = d(s, a) + d(a, b) + d(b, e) \leq d(s, a) + d(a, b) + d(b, n) = d(n, s)$. Then we can always find the diameter starting from n.

Summarizing both cases, we prove that we can always find the diameter starting from the last node dequeued by BFS starting from any node.

Time complexity: this algorithm is $O(n)$, where n is the number of nodes in the tree, because this algorithm first does BFS which is $O(n)$, then resets the visited flag which is $O(n)$, and finally does

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

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

findHeight, which is also $O(n)$ per the analysis in the previous homework.