Assignment 4 – COMP 252

Alexandre St-Aubin and Jonathan Campana

February 26, 2024

1. Browsing the small elements in a red-black tree.

2. GREEDY ALGORITHM. On a flat table, we have placed n disks of radii $r_1, \ldots, r_n$, numbered from left to right. We push them together without creating overlap, as in the figure below. Give an $O(n)$ time algorithm to compute the size of the smallest axis-aligned rectangle that can hold the disks.

   Solution:
   See the algorithm on the next page. We show that its complexity is $O(n)$. To begin, the outermost loop at line 8 iterates a total of n times. The maximum number of elements pushed onto the stack is also n (since at most one is pushed at each iteration), hence an equal number is removed, maintaining the overall $O(n)$ complexity. Finally, as we traverse through each circle once more at the end of the algorithm, the complexity remains $O(n)$.

Algorithm 1: Greedy circle packing

Input: An ordered list $\Omega := \{1, 2, 3, \ldots, n\}$ of n circles.
Output: The minimum width of a rectangle that can hold the disks.

```
// radius of the circle a
Function Radius(circle a):
    return radius of a;

// distance between the centres of a and b if they are pushed together.
Function centre_dist(circle a, circle b):
    return sqrt((Radius(a) + Radius(b))^2 - (Radius(a) - Radius(b))^2);

// largest subarray of Ω ending with k, decreasing in radii.
MAKENULL(possible_adjacent_stack);
// distance from the left side of the rectangle to circle at index
left_to_circle[] ← new array;
left_to_circle[0] ← 0;
for all i ∈ Ω do
    if i = 1 then
        left_to_circle[i − 1] ← Radius(i);
        PUSH(i, possible_adjacent_stack)
    else
        // initialize max_d, the maximum distance between left side of
           rectangle and circle i to Radius (i).  to account for the case
           where the circle would touch the side of the rectangle.
        max_d ← Radius(i);
        // POP each circle on the stack that's smaller than i.
        while Radius(PEEK(possible_adjacent_stack)) ≤ Radius(i) do
            max_d ← max{max_d, left_to_circle(POP(possible_adjacent_stack)) +
             centre_dist(i, possible_adjacent_stack(j))};
        // peek the first circle that's larger, but keep it on the stack
        max_d ← max{max_d, left_to_circle(PEEK(possible_adjacent_stack)) +
         centre_dist(i, possible_adjacent_stack(j))};
        // push i to stack, keeping the decreasing order
        PUSH(i, possible_adjacent_stack);
        // max_d is the distance from the left side of the rectangle to the
           center of circle i when it is pushed as much as possible without
           overlapping
        left_to_circle[i − 1] ← max_d;
// find the width
Width ← 0;
for i ← n − 1 to 0 do
    if left_to_circle[i] + Radius(i) > Width then
        Width ← left_to_circle[i] + Radius(i);
return Width;
```

3. AUGMENTED DATA STRUCTURES. Show how to maintain a dynamic set of numbers that supports the operation `min-gap`, which gives the magnitude of the difference of the two closest numbers in a set of numbers, A. For example, if $A = \{1,5,9,15,18,22\}$, then `min-gap` (A) returns $18-15=3$, since 15 and 18 are the two closest numbers in A. Make the operations `insert, delete, search`, and `min-gap` as efficient as possible, and analyze the running times. Be concise!

4. THE NODE OF SMALLEST TENSION IN A TREE. Given is an unrooted free tree of size n. The nodes are labeled from 1 to n, and for each node, we have a linked list of its neighbors. When a node u is deleted, it breaks the tree up into a forest of disjoint trees, say $T_1, ..., T_k$. Let the sizes of these trees be denoted by $|T_1|, ..., |T_k|$. We define the tension of u as $\max_{1 \le i \le k} T_i$. The objective is to find a node u of smallest tension. Intuitively, it should be near the "center" of the tree. Write an algorithm that takes $O(n)$ worst-case time.

   Solution:
   See the algorithm on the next page. The complexity is easily seen to be $O(n)$, given that line 21 executes at most n times, and both loops at lines 28 and 34 iterate n times. Additionally, the post_order_traversal function is $O(n)$, as it traverses every node in the tree.

   The algorithm uses the following data structure,

```
struct NODE
{
linkedList neighbour;
int tension;
int size;
bool visited;
}
```

Algorithm 2: Smallest Tension

Input: A list $\Omega := \{1,2,3,...,n\}$ of nodes, and for each node, a linked list of neighbours.
Output: The node of minimum tension.

```
Function post_order_traversal(root):
    root.visited ← True;
    if |root.neighbours| = 1 then
        root.size ← 1;
        root.tension ← n − 1;
        return;
    forall NODE child in root.neighbours do
        if child.visited = True then
            continue // means the node is a parent, so don't visit.
        post_order_traversal(child);
    // visit node
    forall NODE child in root.neighbours do
        root.size ← root.size + child.size; // one of the nodes is going to be the
            parent, but since it hasn't been visited yet, its field size will
            be 0, so this is an accurate calculation of the size.
    root.tension ← max{max_{i∈root.neighbours}{i.size}, n − root.size};
    return;
// -- Driver code --
// add fields to each node.
forall i ∈ Ω do
    temp ← i.neighbours;
    i ← new struct NODE;
    i.visited ← False;
    i.neighbours ← temp;
    i.tension ← 0;
    i.size ← 0;
// find a node that's not a leaf and initialize it as root.
forall i ∈ Ω do
    if |i.neighbours| > 1 then
        TopRoot ← i;
        exit;
// call the traversal at the top root
post_order_traversal(TopRoot);
temp ← ∞;
for all i ∈ Ω do
    if i.tension < temp then
        temp ← i.tension;
return temp;
```