CS 240 — Lecture 16

Bartosz Antczak *Instructor: Troy Vasiga* *March 7, 2017*

16.1 More on 2-D Range Search

16.1.1 Kd-trees

Search complexity

We define $Q(n)$ to be the maximum number of regions in a kd-tree with n points that intersect a vertical (horizontal) line. $Q(n)$ satisfies

$$Q(n) = 2Q(n/4) + O(1)$$

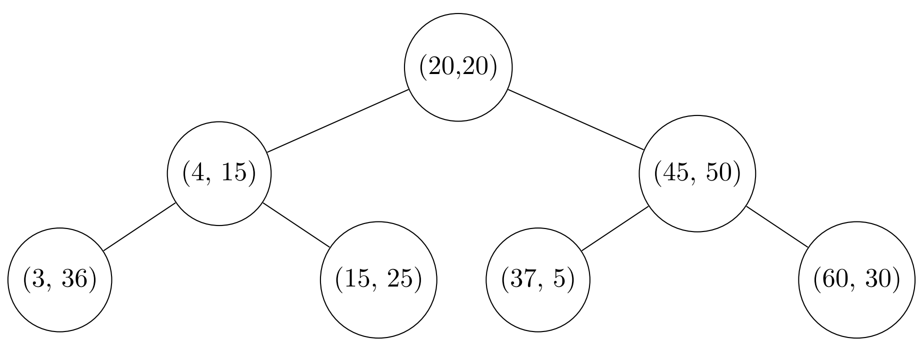
It solves that $Q(n) = O(\sqrt{n})$.

16.1.2 Range Trees

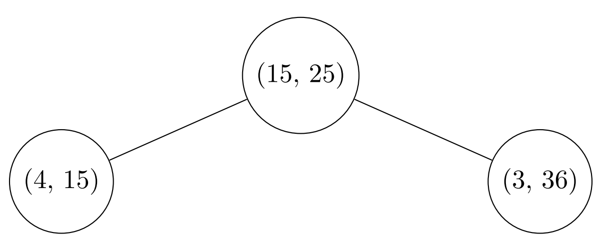
We have n points $P = \{(x_0, y_0), (x_1, y_1), \cdots, (x_{n-1}, y_{n-1})\}$. A range tree is a *tree of trees*. This tree is determined by the $x-$coordinates. We build it using the following steps:

- Build a balanced binary search tree τ determined by the $x-$coordinates
- For every node $v \in \tau$, build a balanced binary search tree $\tau_{assoc}(v)$ (associated structure of τ) determined by the $y-$coordinates of the nodes in the subtree of τ with root node v

Example 16.1.1. *A sample range tree. This tree is ordered by the x-coordinates (i.e., all x values less than the root's x-value are to the left, and the greater values to the right)*



Now if we consider the node (4, 15), we will draw its associated tree ($\tau_{(4,15)}$):



In short, every node in a range tree has two sets of children that compose of two different trees (τ and τ_{assoc}). One tree orders all of its children plus the root based on the order on the x and y coordinates respectively.

Range Tree Operations

- `Search`: trivially as in a binary search tree ($O(\log n)$
- `Insert`: insert a point in τ by $x-$coordinate. From the inserted leaf, walk back up to the root and insert the point in all associated trees $\tau_{assoc}(v)$ of nodes v on path to the root (there are $\log n$ trees and each insertion takes $O(\log n)$. So our running time is $O(\log^2 n)$)
- `Delete:` analogous to insertion

Note: rebalancing a range tree will be problematic! We must use another method, which we will not cover in this course.
The main operation which will concern us will be range search.

16.1.3 Range Search on a Range Tree

It is a two stage process. To perform a range search query $R = [x_1, x_2] \times [y_1, y_2]$:

- Perform a range search (on the $x-$coordinates) for the interval $[x_1, x_2]$ in τ
- For every outside node, do nothing ($O(1)$)
- For every "top" inside node v, perform a range search (on the $y-$coordinates) for the interval $[y_1, y_2]$ in $\tau_{assoc}(v)$. During the range search of $\tau_{assoc}(v)$, do not check any $x-$coordinates (they are all within range) ($\log n \times \log n = O(\log^2 n)$)
- For every boundary node, test to see if the corresponding point is within the region R ($O(\log n)$)

The running time of search if $O(k + \log^2 n)$. We need $O(n \log n)$ space. Why do we need that much space? Intuitively, you'd think we would need $O(n^2)$ space, but think about it: there are n nodes, and each node is in at most $\log n$ trees, ergo $O(n \log n)$. $\log^2 n < \sqrt{n}$.

16.1.4 Higher Dimensions

For every node, there are a total of d trees (where d is the dimension).

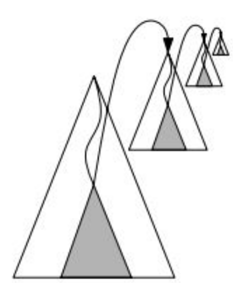


Figure 16.1: A 4-dimensional tree

16.2 Tries and String Matching

I sense this will merge topics with CS 241 (and the first few weeks of CS 246 too).

16.2.1 Pattern Matching

Involves searching for a string in a large body of text. Say our text is in $T[0, \cdots, n-1]$ (the *haystack*), and our pattern is $P[0, \cdots, m-1]$ (the *needle*). These strings are over the alphabet Σ. We want to return the first i such that

$$P[j] = T[i+j] \quad (0 \leq j \leq m-1)$$

This is the first occurrence of P in T. If P does not occur in T, then we return `FAIL`.
Some applications involving pattern matching include:

- Information Retrieval (text editors, search engines)
- Bioinformatics (your DNA is nothing more than C, G, T, A)
- Data Mining

Example 16.2.1. *Pattern matching on a particular string*

- $T =$ "Where is he?"
- $P_1 =$ "he"
- $P_2 =$ "who"

The search for P_1 returns 1 (index 1 in T), and the search for P_2 returns **FAIL**.

Some Definitions

- **Prefix**: a substring $T[0...i]$ of T
- **Suffix**: a substring $T[i...n-1]$ of T

16.2.2 General Idea of Pattern Matching Algorithms

Pattern matching algorithms consist of guesses and checks:

- A **guess** is a position i such that P might start at $T[i]$
- A **check** of a guess is a single position j with $0 \leq j < m$ where we compare $T[i+j]$ to $P[j]$. We must perform m checks of a single correct guess, but we may make (many) fewer checks of an incorrect guess

So how do we implement these algorithms?

16.2.3 Approach 1: brute force

We scan the entire string and check every index. We start at the 0th index, and check every single one until we match: Obviously, we can better than brute force. We will focus on four, more sophisticated algorithms. We'll dive into them starting in the next lecture.

$T =$ abbbababbab, $P =$ abba

a	b	b	b	a	b	a	b	b	a	b
a	b	b	a							
	a									
		a								
			a							
				a	b	b				
					a					
						a	b	b	a	