,18,19,20

CS 240 - Data Structures and Data Management **Spring 2017**

Lecture 17,18,19,20: July 3 - 13, 2017

Lecturer: Taylor Smith *Notes By: Harsh Mistry*

17.1 Multi-Dimensional Data

- Each item has d aspects
- Aspect values (x_ii) are numbers
- Each item corresponds to a point in d-dimensional space
- We concentrate on d = 2, i.e., points in Euclidean plane

One-Dimension Range Search

- First Solution : Ordered Arrays
  Running Time : $O(\log n + k$, k is then number of reported items
- Balanced BST
  - Nodes visited during search
    $O(\log n)$ Boundary nodes
    $O(k)$ inside nodes
    No outside notes
  - Running Time : $O(\log n + k)$

```
1  BST-RangeSearch(T, k1, k2) {
2    If T = null then return
3    if key(T) < k1 then
4      BST-RangeSearch(T.right, k1, k2)
5    if key(T) > k2 then
6      BST-RangeSearch(T.left, k1, k2)
7    if k1 <= key(T) && k2 >= key(T) then
8      BST-RangeSearch(T.left, k1, k2)
9      report key(T)
10     BST-RangeSearch(T.right, k1, k2)
11 }
```

2-Dimensional Search

- Each item has 2 aspects
- Each item corresponds to a point in Euclidean plan

Quadtrees

- Find a square R that contains all the points of P (We can compute minimum and maximum x and y values among n points)
- Root of the quadtree corresponds to R
- Split: Partition R into four equal subsquares (quadrants), each correspond to a child of R
- Recursively repeat this process for any node that contains more than one point
- Points on split lines belong to left/bottom side
- Each leaf stores (at most) one point
- We can delete a leaf that does not contain any point

Operations

- Insert
  - Search for the point
  - Split the leaf if there a two points
- Delete
  - Search for the point
  - Remove the point

  If its parent has only one child left, delete that child and continue the process towards the root
- Range Search

```
1  RSearch(T,R)
2    if (T is a leaf) then
3      if (T.point is in R) then
4        report T.point
5    for each child C of T do
6      if C.region interest R is not empty then
7        RSearch(C,R)
```

  - Spread factor $P : \beta(P) = \frac{d_{max}}{d_{min}}$
  - d_{max} is the maximum distance between two points
  - Height : $h \in \theta(\log_2 \frac{d_{max}}{d_{min}})$
  - Complexity to build : $\theta(nh)$
  - Complexity of range search: $\theta(nh$ even if the answer is $\emptyset$

Quad trees are very easy to compute and handle and require no complicated arithmetic, butare very wasteful of space. One major drawback is that they can have very large heights for non-uniform distributions.

kd-trees

kd-trees split trees into two Split the points into two (roughly) equal subsets

- Building a kd-tree

  - Split P into two equal subsets using a vertical line
  - Split each of the two subsets into two equal pieces using horizontal lines
  - Continue splitting, alternating vertical and horizontal lines, until every point is in a separate region

- Complexity : $\theta(n \log n)$
- Height of tree : $\theta(\log n)$

Operations

```
1  RSearch(T,R)
2    if (T is a leaf) then
3      if (T.point is in R) then
4        report T.point
5    for each child C of T do
6      if C.region interest R is not empty then
7        RSearch(C,R)
```

- The complexity is $O(k+u)$ where k is the number of keys reported and U is the number of regions we go to but unsuccessfully
- U corresponds to the number of regions which intersect but are not fully in R
- Q(n) : Maximum number of regions in a kd-tree with n points that intersect a vertical line
- Q(n) satisfies the following recurrence relation

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

- Q(n) = $O(\sqrt{n})$
- Thus the complexity of range search in kd-trees is $O(k + \sqrt{n})$

kd-tree : Higher Dimensions

- Storage : $O(n)$
- Construction : $O(n \log n)$
- Range query time : $O(n^{1-1/d} + k)$, where d is constant

Range Trees

A range tree is a tree of trees (a multi-level data structure). To build one follow the following steps

- Build a balanced BST τ determined by the x-coordinates of the n points
- For every node $v \in \tau$ build a balanced binary search tree $T_{assoc}(V)$ determined by the y-coordinates of the nodes in the subtree of τ with root node v

Range Tree - Range Search

Running Time is $O(k + \log^2 n$ and Space Tree usage is $O(n \log n)$

- Perform a range search (on the x-coordinates) for the interval [x1; x2] in (BST-RangeSearch(τ; x1; x2))
- For every outside node, do nothing.
- For every "top" inside node v, perform a range search (on the y-coordinates) for the interval [y1; y2] in $T_{assoc}(v)$. During the range search of $T_{assoc}(v)$, do not check any x-coordinates (they are all within range).
- For every boundary test to see if the corresponding point is within the region R.

  Range Tree : Higher Dimensions

  - Storage : $O(n(\log n)^{d-1})$
  - Construction time : $O(n(\log n)^{d-1})$
  - Range query : $O((\log n)^d + k)$

17.2 Pattern Matching

Definition 17.1
- ***Substring*** *is a string that is within a string*
- ***Prefix*** *is a string of characters that precedes a specified*
- ***Suffix*** *is a string of characters that follows a specified value*

Pattern matching algorithms consists of guesses and checks:

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

Brute Force Algorithm

- Worst case performance $\theta((n - m + 1)m)$
- $m \leq n/2 \implies \theta(mn)$

Strign matching with finite automata

Utilize a finite automata and use characters as input to determine which state to proceed to.

- Matching time on a text string of length n is $\theta(n)$
- This does not include the preprocessing time required to compute the transition function.

KMP Algorithm

- Knuth-Morris-Pratt algorithm (1997)
- Compares teh pattern to the text in left-to-right
- shifts the pattern more intelligently
- A failure table is calculated and used to determine how far we should shift the pattern

```
1  KMP(T,P) {
2    F = failureArray(P)
3    i = 0;
4    j = 0;
5    while i <n  do
6      if T[i] = p[j] then
7        if j = m -1 then
8          return i -j //Match
9        else
10         i = i + 1
11         j = j +1
12     else
13       if j > 0 then
14         j = F[j-1]
15       else
16         i = i + 1
17   return -1 // No match.
18 }
```

- failureArray running time : $\theta(m)$, there are no more than 2m iterations
- KMP running time $\theta(n)$, there a re no more than 2n iterations.

Boyer-Moore Algorithm

Based on 3 key ideas

- Reverse-order searching : Compare P with a subsequence of T moving backwards
- Bad character jumps : When mismatch occurs at T[i] = c
  - if P contains c, we can shift P to align the last occurrence of c in P with T[i]
  - Otherwise, we can shift P to align P[0] with T[i+1]
- Good suffix jumps : If we have already matched a suffix of P, then get a mismatch, we can shift P forward to align with the previous occurrence of that suffix. If none exists, look for the longest prefix of P that is a suffix of what we read.
- Can skip large parts of T.

Last-Occurrence Function

- Pre-process the pattern P and the alphabet $\sum$
- Build the last-occurrence function L mapping $\sum$ to integers

- L(c) is defined as the largest index i such that P[i] = c or -1
- The last occurrence function can be computed in time $O(m+ | \sum |)$

```
1  Boyer-moore(T,P)
2    L = last occurrence array computed
3    S = good suffix array computed from P
4    i = m -1
5    j - m -1
6    while i < n and j >= 0 do
7      if T[i] = P[j] then
8        i = i - 1
9        j = j - 1
10     else
11       i = i + m - 1 - min(L[T[i]], S[j])
12       j = m -1
13   if j == -1 return i + 1
14   else return FAIL
```

- Worst-case running time $\in O(n+ | \sum |)$
- This complexity is difficult to prove
- Worst-case running time $O(nm)$ if we want to report all occurrences.

Rabin-Karp Fingerprint Algorithm

- Compute hash function for each text position
- No explicit hash table: just compare with pattern hash
- If a match of the hash value of the pattern and a text position found, then compares the pattern with the substring by naive approach
- Expected running time is $\theta(m + n)$
- Worst case is $\theta(mn)$

Suffix Tries and Suffix Trees

If we want to search for multiple patterns in a fixed text, we can produce a suffix trie consisting of all suffixes of the string and then search for the pattern normally

Running Time Overview

	Brute Force	DFA	KMP	BM	RK	Suffix trees
Preproc.	-	$O(m \mid \sum \mid)$	O(m)	$O(m+ \mid \sum \mid)$	O(m)	$O(n^2)$
Search	$O(nm)$	O(n)	O(n)	O(n)	O(n + m)	O(m)
Extra Space	–	$O(m \mid \sum \mid)$	O(m)	$O(m+ \mid \sum \mid)$	O(1)	O(n)