COMP2160/2860: Data Structures

The University of Sydney
School of Information Technologies

Week 11:
- Data structures related to geometric problems
- kd-trees and range trees

Examples of geometric problems

- Convex hulls
- Range searching
- Point location
- Polygon triangulation
- Clustering, nearest neighbor, outlier detection

Geometric problems

- Example: Given \( n \) points on a plane, compute their convex hull
- Points are given by their \((x,y)\) coordinates

Convex hulls

- A set \( S \) is called convex iff for any \( p,q \) in \( S \) the line \( pq \) is completely contained in \( S \)
- The convex hull of \( S \) is the smallest convex set that contains \( S \)
- The convex hull of \( S \) is denoted \( \text{CH}(S) \)
Computing the convex hull

- Given a set of points $P = \{p_1, p_2, \ldots, p_n\}$, the convex hull is usually given as a list of points in $P$ that are vertices of $P$ in clockwise order.

- Testing if points $p, q$ are on the convex hull: all of $P$ except $p$ and $q$ must be on the same side of $pq$.

Computing the convex hull

- Graham's scan (1972) (a modification by Andrew 1979)
  - Sort the points by x-coordinate
  - $p_1, p_2, \ldots, p_n$ is the x-sorted sequence
  - Compute the upper hull
  - Compute the lower hull

Checking for left turns

- If there is a left turn remove the middle point, and
- check again the "new" last three points.

Main idea of the algorithm:
- Build the (upper) hull incrementally, left to right
- When adding a new point to it check if the last 3 points make a right turn
- If not, remove the middle one
Convex hull algorithm

- Input: set of points P. Output: list of points on CH(P) clockwise

1. sort points by x coordinate in sequence p_1, p_2,...,p_n
2. add p_1 and p_2 to the upper hull list L_{upper}
3. for i := 3 to n
4. append p_i to L_{upper}
5. while last 3 points in L_{upper} do not make a right turn
6. delete the middle of the last 3 points from L_{upper}
7. Add the points p_n and p_{n-1} to the lower hull list L_{lower}
8. for i := n-2 down to 1
9. append p_i to L_{lower}
10. while last 3 points in L_{lower} do not make a right turn
11. delete the middle of the last 3 points from L_{lower}
12. To avoid duplication, remove first and last points of L_{upper}
13. return the two lists (L_{lower}, appended to L_{upper})

Convex hulls

- Graham’s scan works in time O(n \log n) (sorting)
- Convex hull computation requires n \log n steps (lower bound)
- However, the lower bound corresponds to cases where almost all the points are on the hull
- In practice very few points may appear on the hull

2d searching

Orthogonal range searching

- given a point set P, find all points that are in a given box
- this problem is called
  - rectangular range query
  - orthogonal range query
1-dimensional query
- \( P = \{p_1, p_2, \ldots, p_n\} \) points on a line
  - in other words this is a set of numbers (x-coordinates only)
- locate point query: find the number \( x \).
- use a binary search tree

2d point queries and kd-trees
- Instead of a binary search tree, we use a **kd-tree**
- The main idea for the construction of the kd-tree is as follows:
  - split the given point set in two sets of roughly the same size using a vertical line (splitting on the x-coordinates)
  - now split the two sets independently in two roughly the same size sets each, by horizontal lines (splitting on y-coordinates)
  - continue like this alternating between splitting on x-coordinates or y-coordinates

**kd-trees**
- alternate splitting on x and y coordinates
- pick the point with the median coordinate (to split evenly: “balanced” tree)
- kd-trees may also be defined to store points only at the leaves

**kd-trees**
- alternate splitting on x and y coordinates
- pick the point with the median coordinate (to split evenly: “balanced” tree)
- left subtree is the “low” sub tree, right subtree is the “high” subtree
split on x-coords. splitting on 4
4 is the median x-coordinate not
the middle x-coordinate of the region

split on x-coords median is (3,2)

split on y-coords
median is (3,2)

left or low subtree on y-splits:
points below the parent

• find point with coordinates (7,3)

x split
y split
2d point query

• every level of the kd-tree is either an x-split level or y-split level
• do a binary search for the query point taking into account the level of the kd-tree during the search

example range query

• find all points with coordinates 0<x<8 and 2<y<5 (query "range" is the green box)

range query

• region: current region of the subtree we are searching
• range: the range query
1. kd-search(node, region, range)
2. if node is empty return
3. if region is contained in range output the whole subtree and return
4. if the node in range add it to the result list
5. if we are at an x-level:
   6. low (left) region is the intersection of region with x<node.x
   7. high (right) region is the intersection of region with x>node.x
8. if we are at an y-level:
   9. low (left) region is the intersection of region with y<node.y
10. high (right) region is the intersection of region with y>node.y
11. if the region of the left(low) subtree intersects the range
12. kd-search(node.left, left region, range)
13. if the region of the right(high) subtree intersects the range
14. kd-search(node.right, right region, range)

example range query

• find all points with coordinates 0<x<8 and 2<y<5 (in the green box)
performance of kd-trees

- a kd-tree
  - requires $O(n)$ space
  - can be built in $O(n \log n)$
  - supports $O(\sqrt{n+k})$ range queries
    where $k$ is the number of reported points

Better query time?

- can we beat the square root of $n$ query time?
- the data structure must use more space to achieve a better query time
  (lower bound by Chazelle)
- range trees: use $O(n \log n)$ space and achieve $O(\log^2 n + k)$ query time
- with some additional techniques, query time can be improved to $O(\log n + k)$

kd-trees and range trees

- kd-trees: search alternates between $x$ and $y$ coordinates
- range trees: search on $x$-coordinates then $y$-coordinates

range trees

- the main tree is a balanced binary search tree on the $x$ coordinates
- for every internal or leaf node $v$, there is an associated balanced binary search tree $T_v$ on
  the $y$-coordinate for all nodes that appear in the subtree rooted at $v$. 

Diagram:
- Binary search tree
- $T_v$
- Same point set, different tree
range search in range trees

- start with a range search on the x coordinates using the main tree as you would with a range query in a usual binary search tree
- for every returned node, do a range search on the y-coordinates using the associated binary search trees of all the returned nodes

Performance of range trees

- previous search algorithm requires $O(\log^2 n + k)$ steps
- $\log n$ bound requires more work

The median

- the median is the “half-way” point of a set
- given a sequence of $n$ numbers, the median can be found as follows:
  - sort the numbers
  - the median is the middle element (element at position “n/2”)
- can we find the median in linear time?

median and selection

- The selection problem: given a set $A$ of $n$ (distinct) numbers and $1 \leq i \leq n$, find the element $x$ that is larger than exactly $i-1$ other elements of $A$
- the median is a special case of the selection problem
median and selection in linear time

- selection of $i$-th largest element:
  - divide the $n$ elements in $n/5$ groups of 5 elements each (and at most one group with the remaining elements)
  - find the median of the groups
  - recursively find the median of the $n/5$ medians
  - partition input array around the median of the medians (re-arrange groups around the median-of-the-medians)
  - let $k$ be one more than the number of elements on the low side of the partition and $x$ is the $k$th smallest element, or the $(i-k)$th smallest element on the high partition
  - if $i=k$ return $x$, otherwise, if $i<k$ then recursively find the $i$-th smallest element on the low part of the partition or the $(i-k)$-th element on the high part of the partition

selection in linear time

- start with $n/5$ groups of 5 elements (plus one more with the rest that may be smaller)

selection in linear time

- sort each (vertical) group size 5, sorting is constant time assume that the groups appear sorted above

selection in linear time

- the $n/5$ medians of the groups
  - find the median of the medians recursively
selection in linear time

- the n/5 medians of the groups
- find x the median of the medians recursively

selection in linear time

- partition the initial array of elements quicksort-style around x, the median of the medians

selection in linear time

array with all elements:

- partition the initial array of elements quicksort-style around x, the median of the medians

selection in linear time

- using the partitioning, count the elements to the left of x
- let k-1 be the number of elements smaller than x so x is at position k in the re-arranged array
• if \( k = i \) we are done, return \( x \)
• otherwise, if \( i < k \) recursively find the \( i \)-th largest element in the low partition, or the high partition otherwise

• bad case: partitioning around \( x \) is not so good so with every recursive call we do not get rid of enough elements
• this cannot be the case

• re-arrange the groups of 5 around the median of the medians
• the arrows are drawn from larger elements to smaller elements

• all elements in the shaded area are larger than \( x \)
selection in linear time

- all elements in the shaded area are smaller than x

- yellow area smaller than x
  blue area larger than x

the partitioning is good

- blue area:
  - “almost half” of the 5-groups are to the right of x
  - every 5-group contributes at least 3 elements
  - ...minus 2, just in case the last group has less than 5 elements

the partitioning is good

- the high partition has at least all the blue elements and low partition has at least all the yellow elements

- blue area:
  - “almost half” of the 5-groups are to the right of x
  - every 5-group contributes at least 3 elements
  - ...minus 2, just in case the last group has less than 5 elements
the median algorithm is linear

- each of the yellow and blue parts have at least \( \frac{3n}{10} - 6 \) elements, that’s:

\[ 3 \left\lfloor \frac{n}{2} \right\rfloor - 2 \geq \frac{3n}{10} - 6 \]

- so every recursive call discards at least this many elements, leaving only \( \frac{7n}{10} + 6 \) to work with

- number of steps required:
  \[ T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10} + 6\right) + O(n) \]

- solution \( T(n) = O(n) \)

medians in linear time

- break up in groups of 5
- sort each group
- find the median of the medians
- partition around the median of the medians
- if not done, continue recursively on one side of the partition

\[ O(n) \]

\[ T\left(\frac{n}{5}\right) \]

\[ O(n) \]

\[ T\left(\frac{7n}{10} + 6\right) \]

1d range query

- use a balanced search tree
- do a modified binary search

```java
1. rangeQuery(list, tree, low, high) {
2.   if (tree is empty) return list
3.   if (high < tree.value) return rangeQuery(tree.getLeft());
4.   if (low > tree.value) return rangeQuery(tree.getRight());
5.   if (low <= tree.value <= high) {
6.     add tree.value to the list
7.     list = rangeQuery(list,tree.getLeft(),low,high);
8.     return rangeQuery(list,tree.getRight(),low,high);
9.   }
10.}
```