Lecture 19: Multidimensional Data

10/9/2020

Range-Finding and Nearest

Search Trees

- We've seen three different implementations of a Map
  - BST
  - 2-3 / B-Tree
  - Red Black Tree
- "search tree" data structures support very fast insert, remove, and delete operations for arbitrary amounts of data
  - Requires that data can be compared to each other with some total order
  - We used the "Comparable" interface as our comparison engine

Expanding the Power of our Set

- There are other operations we might want to include:
  - `select(int i)`: Returns the ith smallest element in the set
  - `rank(T x)`: Returns the "rank" of x
  - `subSet(T from, T to)`: returns all items between `from` and `to`
  - `nearest(T x)`: Returns the value closest to x

Implementing Fancier Set Operations with a BST

- It turns out that a BST can efficiently support the `select`, `rank`, `subSet`, and `nearest`
- How would you find `nearest(N)`?
  - Just search for `N` and record closest item seen
  - Exploits the BST structure

Sets and Maps on 2D Data

- So far we've only discussed "one dimensional data". That is, all data could be compared under some total order
- But not all data can be compared along a single dimension
  - We'll see that search trees require some design tweaks to function efficiently on multi-dimensional data

Multi-Dimensional Data

Motivation: 2D Range Finding and Nearest Neighbors

- Suppose we want to perform operations on a set of Body objects in 2D space
  - 2D range searching: How many objects are in a highlighted rectangle
  - Nearest: What is the closest object?

- Ideally, we'd like to store our data in a format that allows more efficient approaches than just iterating over all objects
- It's difficult to build a BST of 2 dimensional data
    - Difficult to compare objects, lose some information about ordering

QuadTrees

The QuadTree

- A QuadTree is the simplest solution conceptually
    - Every Node **four** children
        - Top left (northwest)
        - Top right (northeast)
        - Bottom left (southwest)
        - Bottom right (southeast)
- Just like a BST, insertion order determines the topology of a QuadTree

QuadTrees

- Quadtrees are a form of "spatial partitioning"
    - Quadtrees: Each node "owns" 4 subspaces
        - Space is more finely divided in regions where there are more points
        - Results in better runtime in many circumstances

QuadTree Range Search

- Quadtrees allow us to prune when performing a rectangle search
    - Simple idea: Prune subspaces that don't intersect the query rectangle

Higher Dimensional Data

3D Data

- Suppose we want to store objects in 3D space
    - Quadtrees have only four directions, but in 3D, there are 8
- One approach: Use an Oct-tree or Octree
    - Very widely used in practice

Even Higher Dimensional Space

- You may want to organize data on a larger number of dimensions
- In these cases, one somewhat common solution is a k-d tree
    - Fascinating data structure that handles arbitrary numbers of dimensions
        - k-d means "k dimensional"
    - For the sake of simplicity, we'll use 2D data, but the idea generalizes naturally

K-d Trees

- k-d tree example for 2-d
    - Basic idea, root node partitions entire space into left and right (by x)

  - All depth 1 nodes partition subspace into up and down (by y)
  - All depth 2 nodes partition subspace into left and right (by x)
  - And continue alternating down the depth of the tree
  - To break ties, we'll say items that are equal in one dimension go off to the right (or up) child of each node
- Each point owns 2 subspaces
  - Similar to a quadtree

K-d Trees and Nearest Neighbor

- You can simplify code by only measuring the length of vertical/horizontal lines instead of diagonal hypotenuses.
  - Optimization: Do not explore subspaces that can't possibly have a better answer than your current best

Nearest Pseudocode

- nearest(Node n, Point goal, Node best):
  - If n is null, return best
  - If n.distance(goal) < best.distance(goal), best = n
  - If goal < n (according to n's comparator):
    - goodSide = n."left"Child
    - badSide = n."right"Child
    - else:
      - goodSide = n."right"Child
      - badSide = n."left"Child
  - best = nearest(goodSide, goal, best)
  - If bad side could still have something useful
    - best = nearest(badSide, goal, best)
  - return best

Uniform Partitioning

Uniform Partitioning

- Not based on a tree at all
- Simplest idea: Partition space into uniform rectangular buckets (also called "bins")
- Algorithm is still Theta(N), but it's faster than iterating over all the points

Uniform vs. Hierarchical Partitioning

- All of our approaches today boil down to spatial partitioning
  - Uniform partitioning (perfect grid of rectangles)
  - Quadtrees and KdTrees: Hierarchical partitioning
    - Each node "owns" 4 and 2 subspaces, respectively
    - Space is more finely divided into subspaces when there are more points

Uniform Partitioning vs. Quadtrees and Kd-Trees

- Uniform partitioning is easier to implement than a QuadTree or Kd-Tree
    - May be good enough for many applications

Summary and Applications

Multi-Dimensional Data Summary

- Multidimensional data has interesting operations:
    - **Range Finding**
    - **Nearest**
- The most common approach is **spatial partitioning**:
    - **Uniform Partitioning**: Analogous to hashing
    - **QuadTree**: Generalized 2D BST where each node "owns" 4 subspaces
    - **K-d Tree**: Generalized k-d BST where each node "owns" 2 subspaces
        - Dimension of ownership cycles with each level of depth in tree
- Spatial partitioning allows for **pruning** of the search space