Spatial Data Indexing Using Grid-Based Method and KD-Tree Method*

Kaichun Mo†
ACM Honered Class, SJTU
Shanghai, China
daerduoCarey@163.com

ABSTRACT

In this report, I will introduce you my work on spatial data indexing using Grid-Based method and KD-Tree method respectively.

To be more specific, I write a C++ program which can successfully retrieve all POIs in any given query range(i.e.*Range Query*) as well as all top-k nearest POIs regarding to any given point(i.e.*KNN Query*). I implement both using two distinct methods, which are widely known as Grid-Base Method and KD-Tree Method.

Categories and Subject Descriptors

[**Data Management**]: Spatial Data Indexing

Keywords

Spatial Data Indexing, POI, Grid-Base Method, KD-Tree Algorithm, Range Query, KNN Query

1. INTRODUCTION

Spatial data indexing plays an indispensable role in almost all modern web service providers. All web services involved map indexing or minimum routine recommendation are pursuing efficient methods to store and index untold many POIs (Point of Interests) in the real world.

Many basic methods work pretty well, such as Grid-Based Method, which is the most intuitive method and easiest method to implement as well. But, it's apparent drawbacks definitely impede the usage of this method. More tricky but efficient methods are proposed by computer scientists.

In this report, I will use KD-Tree to implement the both two kinds of queries and show you that this method does a better tradeoff between spatial complexity and temporal complexity.

In the last part of this report, I will compare the performance of the two different methods using the real data.

2. GRID-BASED METHOD

This section introduces the basic idea of Grid-Based method as well as the concrete implementation details involved in programming.

2.1 Brief Introduction

Grid-Based method is the most intuitive method to fulfill the job. The basic idea is just to divide the whole space into a bunch of grids. Each grid are considered as the basic impartible block containing a relatively small number of POIs. When encountering range query, we can just search all grids which intersect with the this range for all possible contained POIs. If a KNN query is asked, we can search the surrounding grids to the query point for the nearest k POIs.

This method is easy to implement and have both pros and cons. The more specific performance analysis will be mentioned in the last part of this section.

2.2 Algorithm Overview

The methods used to deal with range queries and KNN queries are quite easy. Following are the brief overviews for all methods respectively.

2.2.1 Initialization

- Linearly scan all the POIs and figure out the smallest rectangle range that contains all POIs.
- Divide the huge rectangle into grids and store all the grids in a list. All the POIs in one specific grid are linked sequentially.
- Each grid can be directly accessed by some trifle computing.

2.2.2 Rectangle Range Query

- For each rectangle queried range, denotes it by left-upper point and right-bottom point.

*This report is intended to be the final report of *Spatial-Temporal Data Analysis*, which is a class lectured by Professor *Zheng Yu* in the summer of 2014.

†I'm a undergraduate student from ACM Honered Class, SJTU, majoring in CS.

- Compute out all grids intersecting with this rectangle. This is easy since we can know the grid where the left-upper point lies and the gird where the right-bottom point lies. All the grids in the rectangle constituted up by this two corner grids should be involved.
- Linearly scan all the POIs in all possible grids searching for the result.

2.2.3 *Circle Range Query*

- Generate the external tangent square of this queried circle.
- Invoke the rectangle range query procedure.
- For each grid considered, first check whether or not this grid intersects with the queried circle. If not, throw away this grid.

2.2.4 *K Nearest Neighbours Query*

- Figure out the grid that the point p lies in, denoted as (x, y), and linearly scan all POIs in it. Use a max heap to maintain the k nearest POIs.
- For each round $k = 1, 2, \cdots$, check all additional grids in range

$$[x - k, x + k] \times [y - k, y + k] \quad (1)$$

deducting

$$[x - k + 1, x + k - 1] \times [y - k + 1, y + k - 1]. \quad (2)$$

- For each grid in the above range, if its distance to p is less than the current root of the priority queue, then we linearly scan all the POIs in it and update the heap.
- Repeat until no grid in round k are scanned to update the heap.

2.3 Performance Analysis

This method is so intuitive that it has natural drawbacks such as ineluctable large storage required and temporal inefficiency when dealing with high unbalanced distribution of POIs. The method requires the program to store at least a pointer for each grid, which means we need to store intolerant Θ(Grid Number) data at least. Meanwhile, if the distribution of POIs are highly unbalanced, one grid may contain too many POIs which leads to more time consuming when linearly scanning the POIs in this grid.

Notwithstanding the mortal drawbacks, the method is the easiest method to implement. Direct Accesses to a specific grid is another merits of this method.

3. KD-TREE METHOD

This is a more enlightened method to solve the possible unbalanced distribution of POIs.

3.1 Brief Introduction

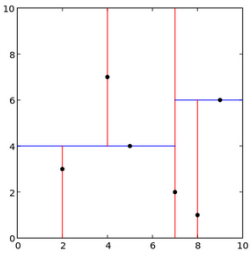
Using the basic Grid-Based method can perform as inefficient as possible when the distribution of POIs is unbalanced. In this case, we can use a wiser way to divide the whole range heuristically.

3.2 Algorithm Overview

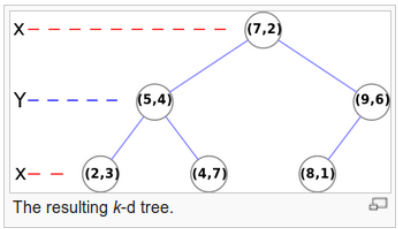
The methods are wiser than those of original Grid-Based method.

3.2.1 *Initialization*

- We build a tree-like structure to store the data. The tree looks like the following example.



- Each time, we prefer to divide the POIs as balanced as possible.
- Each round i, if i is even, we sort all POIs in current range with respect to the x value of the points and use the middle point as the separator. If i is odd, choose the separator with respect to y value of the points.
- Now, we separate the original range into two parts. Next, we recursively invoke this procedure itself to each part, which is the procedure of round $i + 1$.
- Repeat until the current range only contain less than m points, where m is a given number indicating the minimum number of POIs a grid can held.
- Note that this procedure generates a tree-like structure for us. Each interior node contains a separator point, while the leaves are indivisible grids holding a relatively small fracture of POIs.



3.2.2 *Rectangle Range Query*

- Start from the aspect of the root.
- For each node we are considering,
  - If the node is a leaf, then linearly scan all POIs stored in this node.
  - If the queried range is separated by the separator stored in this node, we just split the queried range into two small ranges with respect to this separator and invoke this procedure itself to each part to collect more informations.
  - If the queried range lies to the left or to the right with respect to the separator POI, we just recursively invoke this procedure on the part the range lies.

- Repeat until all nodes are visited or cut off.
- For each processing to one POI, just check whether or not the point is in the queried range. If the answer is yes, just add it to the final result set.

3.2.3 Circle Range Query

- Just find out the exterior tangent square of the queried circle.
- Gather informations from the rectangle range query procedure and select the valid POIs lying in the range of the queried circle.

3.2.4 K Nearest Neighbours Query

- Start from the root of the obtained KD-Tree.
- Maintain a max heap to store all temporary k nearest POIs and guarantee that the size of this heap is by no means larger than k.
- For each node visiting, we can consider this node as a rectangle range.
  - If the node is a leaf, then just linearly scan all the POIs contained in this node and process each POI point.
  - If the node is a interior point, it must contain a separator. Use the separator to divide the current node into two small nodes. Compute the distances between each node(the corresponding rectangle) to the queried point.
    * If both distances are less than the worst distance in the heap, then we need to recursively invoke on each small range. We invoke the one having relatively small distance to the queried point first.
    * If only one distance is less than the worst distance in the heap, just invoke on that viable part.
    * If both not, do nothing and quit this call of the procedure.
- Note that there is a little trick to make this program more efficient, which is that each time we are deciding whether or not to invoke other small range, we should recheck the condition that whether or not this range's distance to queried point are less than the worst distance in the updated heap.
- Repeat until all the nodes are visited or cut off.

3.3 Performance Analysis

Since the KD-Tree can deal with the situation of unbalanced distribution of POIs, we regard this method as a self-adaptive method. This property guarantees that each query is efficiently enough no matter what the inputs or the query range is.

Meanwhile, the spatial complexity is tolerable, since all the grids contain proportionable points, which means no grid is empty and wasted. This leads to the fact that this method has a dramatically less spatial complexity compared to the original Grid-Based method.

But, this method also has some drawbacks. Since the data are constructed into a tree-like structure, we can not access one node directly, which means we need more time to access one node. Besides, the implementation is relatively harder.

4. PERFORMANCE COMPARISON

Since the given data is of a small scale, so I decide to regard all POI categories as one general category.

I adopt the following method to compute the running time.

- Each POI scanning accounts for ONE unit time.
- Each main procedure invocation accounts for ONE unit time.
- Each node or grid accounts for ONE unit time when it has been focused on.

I DO NOT take the following running time in my consideration.

- The time of copying a result when the copy is only needed for passing the returned value.
  - Since the copy of returned value is just counted when counting the number of procedure invocation, counting it again seems not very necessary.
- The time of processing general operations and comparisons.
  - It's hard to decide how to count the running time of this part. If the data scale increases to a large extent, this part of time consuming is just a constant factor multiplied to the total time consuming.
- The time wasted due to my lack of ability to optimize the algorithm.
  - Actually, I can figure out some wiser way to eliminate this part of time consuming. But, in fact, I do not want to re-program due to the time limitation.

Following is the result of time consuming based on my evaluation method. (Here we assume in KD-Tree method, each leaf cannot contain more than 3 points.)

Method	**Rec Query**	**Cir Query**	**10-NN**
Brute Scan	2020	2020	2020
5×5 G-B	619	237	244
10×10 G-B	128	202	227
50×50 G-B	255	110	156
KD-Tree	101	117	68

From the above result, we can know

- Using Grid-Based method, one should find a good trade-off between too many grids and too concentrated POIs in one grid.

- Using KD-Tree method, the result is relatively compelling and attractive.

5. CONCLUSION

In this report, I use two distinct methods to achieve spatial data indexing. The two methods are Grid-Based method and KD-Tree method. They both have advantages and disadvantages. Some parameters in both methods should be set according to the concrete application data. Besides, when dealing with the real data, one should choose from several extant methods with respect to the feature of the data. If the data is distributed relatively balanced, using Grid-Based method is not always bad.

6. ACKNOWLEDGEMENT

Very grateful for the helps from Professor *Zheng Yu* and affable TA *ZhengYang Liu*.

This LaTeXcontemplate is from *ACM SIG Proceedings Templates*. http://www.acm.org/sigs/publications/proceedings-templates#aL1.