Hash-based key-value pair storage for efficient range search

Pham Ngoc Lam s1222006 Supervised by Prof. Maxim Mozgovoy

Abstract

In games that are applied AI algorithms, we inevitably provide data sets for them for training so that data sets should be organizing well by using a data structure providing adequate performance. The runtime of data queries must be concerned about when the system could not be a strong machine (such as smart phones). In this article, we introduce to an enhanced method of designing a new strategy of data store in a system which can boost the runtime compared to built-in libraries. This article is aimed to resolve the problem how to store data of keys as distinct integer tuples with their properties.

1 Introduction

AI algorithms that rely on case-based reasoning contain a retrieval step which returns matching situations from a knowledgebase. That is one of reasons why applications in which scientists apply AI algorithms demand a large number of data set for training (supervised and unsupervised). For a large scalable application as a complicated system, we must provide it with labeled data sets which are stored in data structures. These data structures must support us with a number of must-have actions such as insertion, deletion and data search. In this report, we will go into detail of soccer game AI in which we supply data set of coordinators and distances between 2 players represented as a vector which contains integers and ranges of integers.

When system of soccer game answers queries of ranges between in-game soccer players controlled by AI, a clear mechanism must be provided because of complicated structure of ranges as tuples of integer. A key may contain integers or ranges of integers that represent parameters of in-game characters to support the system in finding best coordinators for characters to move to. For example, in Soccer game on smartphones, a mechanism is that in-game players run to good places based on distances from them to others, we represent players' coordinator as a tuple of integers and this tuple can also store other information which supports our system to find good strategies for AI players.

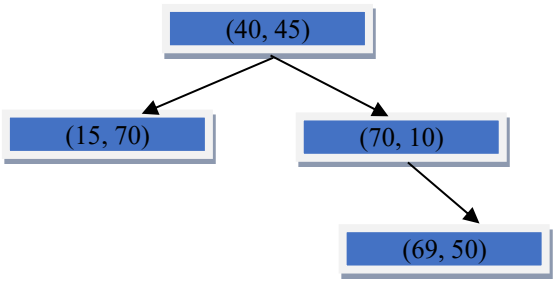
My solution for this problem is applying a method as a combination of KD-Tree and Hash Function to store keys as tuples of integers. Because of its complicated structure and no library provided a specific method to support it, we proposed to hash "ranges of integer" into single values as integers and then, we can apply K-D tree to store their key, values and other information.

2 Methods

2.1 K-D Tree (K-dimensional Tree)

In computer science, a k-d tree[1] is a space-partitioning data structure for organizing points in a k-dimensional space. K-d tree is a useful data structure for several applications.

In the case of no ranges of integer appeared in key (For example: <-100, 22, 3, 400>). We can store key with its value in K-D tree without concerning about collisions. We can consider the number of element in total as the number of dimension in order to insert this key into K-D tree.
In this article, we will introduce operations on a k-d tree.



2.1.1 K-D tree Construction

The traditional method of k-d tree construction has the following constraints:
- As one moves down on the tree, one cycles through the axes used to select the splitting planes.
- Points are inserted by selecting the median of the points being put into the subtree.

Pseudocode below is a way of k-d tree construction.

```
function KDTreeBuild(List of points, int depth){
    //Select axis based on depth
    int axis := depth % k;
    //Sort point list and choose median as pivot element
    select median by axis from list of points.
    //create node and construct subtree
```

```
        node.location := median;
        node.leftChildren := KDTreeBuild(points in list before
    median, depth + 1);
        node.rightChildren :=  KDTreeBuild(points in list
    after median, depth + 1);
      }
```

2.1.2 Adding new elements

Insert operation in a k-d tree is the same way as one adds an element to any other search tree(AVL tree, red black tree).

- Traverse the tree, starting at root node and moving to either the left or the right child depending on whether the point to be inserted is on the left or right side of the splitting plane.
- Add the new point as either the left or right child of the leaf node, again depending on which side of the node's splitting plane contains the new node.

Pseudocode below describes the way of how to insert new elements into a k-d tree.

```
Node insert(Node root, point, int depth){
    if root == null: return new Node(point).
    //calculate current dimension of comparison.
    int axis := depth % k;
    if point[axis] < root->point[cd]:
      root->left = insert(root->left, point, depth + 1);
    else
      root->right = insert(root->right, point, depth + 1);
    return root;
}
```

2.1.3 Find minimum element

To find minimum point, we traverse nodes starting from root. If dimension of current level is same as given dimension, then required minimum lies on left side if there is left child.

Pseudocode below describes the way of how to find minimum element.

```
int findMin(Node root, int d, int depth){
    if root == null: return INT_MAX;
    int axis := depth % k;
    if axis == d:
      if root->left == null:
        return root->point[d];
      return findMin(root->left, d, depth + 1);
    return min(root->point[d], findMin(root->left, d, depth +
1), findMin(root->right, d, depth + 1));
}
```

2.1.4 Removing elements

If we want to remove a point from an existing k-d tree, without breaking the invariant, the easiest way is to form the set of all nodes and leaves from the children of the target node, and then recreate the part of the tree.

Pseudocode below is one way of deleting elements existing in a k-d tree.

```
Node deleteNode(Node root, point p, int depth){
    if root == null: return null;
    int axis := depth % k;
    if point of root and point p are the same:
      if root->right != null:
        //find minium of root's dimension in right subtree
        Node min := findMin(root->right, axis);
        copyPoint(root->point, min->point);
        root->right := deleteNode(root->right, min->point,
depth + 1);
      else if root->left != null:
    Node min := findMin(root->left, axis);
    copyPoint(root->point, min->point);
    root->left := deleteNode(root->left, min->point, depth + 1);
  else return null;
  return root;
}
```

2.1.5 Searching elements

To search elements existing in a k-d tree, we only need to traverse the tree, starting at root node and then moving to the left or right child depends on comparison between target point and point of node we are traversing at.

```
bool searchElement(Node root, point p, int depth){
    if root == null: return false;
    if root->point and p are the same: return true;
    int axis = depth % k;
    if p[axis] < root->point[axis]:
      return searchElement(root->left, p, depth + 1);
    else
      return searchElement(root->right, p, depth + 1);
}
```

2.1.6 Complexity for each operation

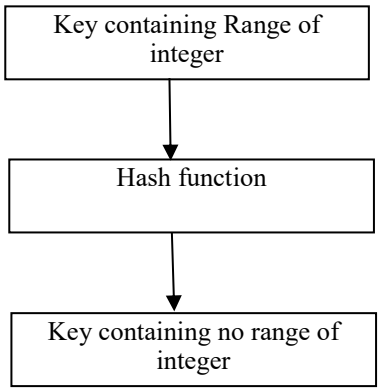
Algorithm	Average	Worst case
Space	O(n)	O(n)
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)

2.2 Hash Function

For keys containing ranges of integer elements, we cannot directly use k-d tree, so that we try to find a method in which we can apply k-d tree on this case. This method is to choose hash function and hash ranges of integer elements. Therefore, we can get only one value for each range and as a result, we can convert a key containing ranges of integer into one containing no ranges of integer.

First, let recall the definition of hash function[2]:

A hash function[3] is any function that can be used to map data of arbitrary size to data of fixed size.



At step of choosing hash function for range of integer, say [a, b] (a, b are integers), because we can know boundary of a and b, and in this range, there are (a – b + 1) integers in total. We come up with a idea of using sums of series which we can find closed-form expressions.

There are some sums of series with their closed-form expressions we can apply on this problem.

1. $$\sum_{i=0}^{n} i^2 = \frac{n*(n+1)(2\,n+1)}{6}$$
2. $$\sum_{i=0}^{n} i = \frac{n*(n+1)}{2}$$
3. $$\sum_{i=0}^{n} i^3 = \frac{(n^2+n)^2}{4}$$
4. $$\sum_{i=0}^{n} i^4 = \frac{n*(n+1)^*(2n+1)(3n^2+3n-1)}{30}$$

In general case: we use sums of series:

$$f(n,a) = \sum_{i=0}^{n} i^a \ (a = 1,2,3,4\ldots.)$$

For each a, we can find a closed-form expression but if a is large, this is a very tough task.

We can control the boundary of a and b (Let assume that b > a ≥ M) so that we can avoid the situation that a is a negative integer because b – M > a – M ≥ 0.

Now [a, b] becomes [a – M, b – M] and then, we apply expressions above. We have:

Hash value of [a, b] $= f(b-M,a) - f(\mathrm{a}-M,a)$

However, because we use hash functions so that collisions can happen. If n is a large number, the number of collision is not a big integer and we can generate all of them which are the key to solve collision avoidance. Now, let us show a method below of calculating the number of collision.

If two different ranges have a same hash value, we will have an equation below:

$f(y1-M,a) - f(\mathrm{x1}\ -\ \mathrm{M},\ a) = f(y2-M,a) - f(x2-M,a)$

$$\langle = \rangle \sum_{i=x1-M}^{y1-M} i^a = \sum_{i=x2-M}^{y2-M} i^a$$

After solving this equation on computer and counting all of ranges that causes collisions, we have a statistical table below:

n \ a	1	2	3	4
1000	120123	1949	199	0
10000	11193351	33791	1079	0

For each value of integer n, we can calculate its universe – the total number of pair a and b (a < b):

$$\Omega = n + (n-1) + (n-2) + \cdots + 1 = \frac{n*(n+1)}{2}$$

And then, based on the above table, we calculate probability of collision which can happen.

n \ a	1	2	3	4
1000	24%	0.39%	0.0397%	0%
10000	22.38%	0.0675%	0.002%	0%

However, when n is large, an big prime P will be a good choice if we want to minimize hash values.

In other word, all hash values (call H) will sastify: 0 ≤ H < P.

The pros of using a prime is we only need to handle not too large numbers of hash values and the cons is more collisions can be happend. Let apply birthday paradox on calculating the probability of two distinct ranges which have a same hash value, we have:

$$P(collision\ exists) = \prod_{i=1}^{\Omega} \left(1 - \frac{i}{P}\right)$$

Let presume that $\sqrt{P} \approx 3^{*}10^{4}$, we can calculate P for specific integers of Ω.

Ω	P(collision exists)
10000	0.9512
100000	0.0067
1000000	6.0273e-218
10000000	2.42092e-322

From the table above, we can conclude that in case Ω is not a large number, we should use more than one big prime to avoid collisions because

$$\lim_{t\to\infty} a^t = 0, (0 < a < 1)$$

t: the number of primes.
a: supremum of P(collision exists).

3 Results

The mechanism above provides a method for us to implement a specific data structure including hashing data set, insertion, deletion and data search.

This method has all advantages of K-D tree and hash-function technique to solve the problem of lacking built-in libraries which support us to store range-of-integer key in database. The table below shows us time complexity for each action.

Algorithm	Average	Worst case
Space	O(n)	O(n)
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)

Although it can solve our problem partially, drawbacks of this method is collision which we can minimize its probability of happening by choosing a good hash function. On the other hand, its implementation in programming languages is complicated.

4 Acknowledgement

Reference

1. Steven S Skiena "The algorithm design Manual – second edition", k-d tree – page 389, 390, 391
2. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. "*Introduction to Algorithms",* hash functions. McGraw-Hill, 2001, page 262 - 268.
3. Steven S Skiena "The algorithm design Manual – second edition", hashing and string – page 89 - 92