BK Tree

The BK Tree data structure was proposed by Burkhard and Keller in 1973 as a solution to the problem of searching a set of keys to find a key which is closest to the query key. The simplest solution to solve the problem is compare the query key with every element in the set, but with this solution, the time complexity is $O(n)$ where n is the number of elements in the list to check and doesn't calculate the complexity of Levenshtein distance. On the contrary, this structure allows fewer comparisons to solve the problem.

BK Tree can be implemented by discrete metric e.g., Levenshtein distance, Damerau-Levenshtein distance. In this paper, we will focus on BK Tree using Levenshtein distance metric which we will explain more in the next topic. The metric is used for calculating the weight from one node to another. With this weight and triangular inequality concept, we can search much faster because we can filter out a branch that you don't need to compare with.

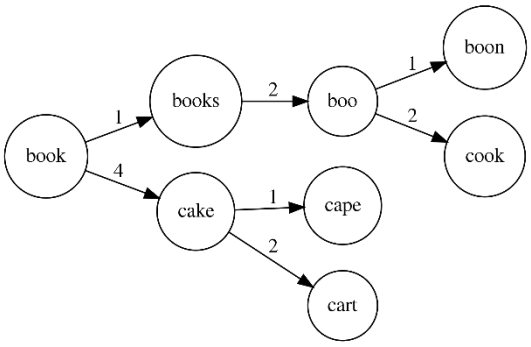

Figure 1: BK Tree [1]

Levenshtein distance

Let's know more about Levenshtein distance or edit distance. Edit distance is a string metric for measuring the difference between two sequences. The Levenshtein distance between two strings a, b (of length |a| and |b| respectively) is given by **lev (a, b)** where

$$\operatorname{lev}(a,b) = \begin{cases} |a| & \text{if } |b| = 0, \\ |b| & \text{if } |a| = 0, \\ \operatorname{lev}\left(\operatorname{tail}(a), \operatorname{tail}(b)\right) & \text{if } a[0] = b[0] \\ 1 + \min \begin{cases} \operatorname{lev}\left(\operatorname{tail}(a), b\right) \\ \operatorname{lev}\left(a, \operatorname{tail}(b)\right) \\ \operatorname{lev}\left(\operatorname{tail}(a), \operatorname{tail}(b)\right) \end{cases} & \text{otherwise,} \end{cases}$$

Figure 2: Levenshtein distance [2]

- tail(x) means the string of x except the first character of x; example, x = "cat" then tail(x) = "at".

- x[n] means the n^{th} character of the string x where we count the first character as 0.

There are 3 ways to edit a sequence to another sequence.

1. Insertion 2. Deletion 3. Substitution

[1] - https://upload.wikimedia.org/wikipedia/commons/thumb/d/de/Bk_tree.svg/220px-Bk_tree.svg.png
[2] - https://wikimedia.org/api/rest_v1/media/math/render/svg/6224efffbe9a4e01afbddeeb900bfd1b3350b335

Edit distance is the minimum number of single-character edits required to change one word into the other. For example, the edit distance between “kitten” and “sitting” is 3, since we can do as the following:

kitten → sitten (Substitution of “s” for “k”)

sitten → sittin (Substitution of “i” for “e”)

sittin → sitting (Insertion of “g” at the end)

```cpp
size_t lev(string a, string b) {
    size_t a_size = a.length();
    size_t b_size = b.length();
    if (a_size == 0) return b_size;
    if (b_size == 0) return a_size;
    if (a[0] == b[0]) return lev(a.substr(1), b.substr(1));
    return 1 + min(min(lev(a.substr(1), b), lev(a, b.substr(1))), lev(a.substr(1), b.substr(1)));
}
```

Figure 3: Levenshtein distance

Damerau-Levenshtein distance

Damerau-Levenshtein distance differs from the classical Levenshtein distance by including transpositions of two adjacent characters among its allowable operation.

$$d_{a,b}(i,j) = \min \begin{cases} 0 & \text{if } i = j = 0, \\ d_{a,b}(i-1,j) + 1 & \text{if } i > 0, \\ d_{a,b}(i,j-1) + 1 & \text{if } j > 0, \\ d_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} & \text{if } i,j > 0, \\ d_{a,b}(i-2,j-2) + 1 & \text{if } i,j > 1 \text{ and } a_i = b_{j-1} \text{ and } a_{i-1} = b_j, \end{cases}$$

Figure 4: Damerau-Levenshtein distance [3]

where i = |a| denotes the length of string a, and j = |b| denotes the length of string b

- $1_{(ai \neq bi)}$ is the indicator function equal to 0 when $a_i = b_i$ otherwise equal to 1.

Triangular inequality

This concept is the reason why we just consider the range $[d_u - T, d_u + T]$ while using spell-checking.

- d (x, y) = 0 <-> x = y (If the distance between x and y is 0, then x = y)
- d (x, y) = d (y, x) (The distance from x to y is the same as the distance from y to x)
- d (x, z) + d (y, z) >= d (x, y)
- d (a, b) denotes the distance between point a and b

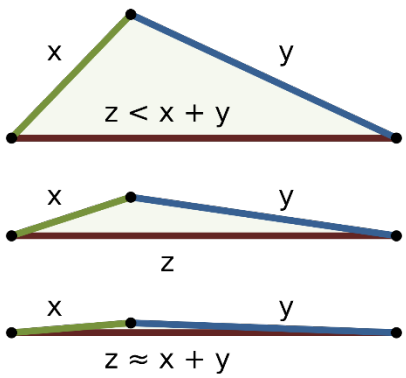

Figure 5: Triangular inequality [4]

[3] - https://wikimedia.org/api/rest_v1/media/math/render/svg/d50fab8cc0233e2b1b5b420f72cb23fdf1d56c59
[4] - https://upload.wikimedia.org/wikipedia/commons/thumb/b/b2/TriangleInequality.svg/1200px-TriangleInequality.svg.png

Usage of BK Tree

Let's say we have a dictionary = {"what", "water", "wait", "whale", "hat", "white", "wheat", "heat"} which we can derive into BK Tree as Figure 3.

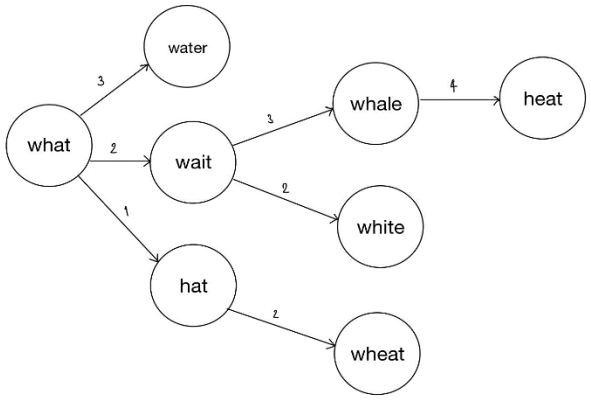

Figure 6

1. Spell-checking

Algorithm

Let's say we set the tolerance limit as T which is a nonnegative integer.

First step: We will declare the expected correct word list L and the empty stack S.

Second step: If the tree is empty, then return the empty list L. But, if it doesn't, then push a root node into S.

Third step: We will loop doing this step until S is empty. Pop the node u out from S, then find its edit distance (d_u) between u's word and the target word. If d_u <= T, then we will insert u's word into our expected correct word list. Next, inspecting the edge between u and u's child that contain the weight in range [d_u – T, d_u + T], we will push the child node into S only if its weight in the range.

Remark if d – T <= 0, we can use 1 instead because edit distance is a nonnegative integer, and it can be 0 only if it is the same word.

Last step: return the expected correct word list L

Time complexity (Without calculating Levenshtein distance)

The time complexity is $O(n^{\propto})$ where $\propto$ is a real number satisfying $0 < \propto < 1$ which depends on the tolerance limit (search radius) and the structure of the tree.

Example (using Figure 6)

Let's say we have a misspelled word "sweat" and the tolerance limit (T) is 2.

As the BK Tree is not empty. Then, we will start checking from the root node as node u. Let d_u = lev ("sweat", "what") = 2. We found that d_u <= T so we add "what" to the expected correct word list. Now we will iterate over its child having edit distance in the range [d_u – T, d_u + T] i.e. [1, 4]

Let's start from the lowest possible edit distance child i.e., node "hat" with edit distance 1. Now let find its edit distance from our target word. Let d_u = lev ("hat", "sweat") = 3. Since this case $d_u = 3 > T = 1$, the word "hat" will not be added to our expected list. Next, let's process its child nodes having edit distance in the range [d_u - T, d_u + T] i.e. [1,5] as the first step.

We will continue iterating through these steps for all the words in the range [d_u - T, d_u + T] starting from the root node to the bottom-most leaf node. For this example, at the end, we will have the expected correct word list with only 3 words i.e. {"what", "heat", "wheat"}

```cpp
vector<string> spell_checking(const string &target, const size_t &T) {
    vector<string> output;
    stack<node *> nodes;
    if (mRoot == nullptr) return output;
    nodes.push(mRoot);
    while (!nodes.empty()) {
        node *u = nodes.top();
        nodes.pop();
        size_t du = lev(u->word, target);
        if (du <= T) output.push_back(u->word);
        int start = du - T, end = du + T;
        if (start <= 0) start = 1;
        for (auto &it : u->children) {
            // variable it is an iterator of pair{size_t, node*}
            // first = edit distance, second = child node
            if (start <= it->first && it->first <= end) nodes.push(it->second);
        }
    }
    return output;
}
```

Figure 7: Spell-checking

```cpp
string string_matching(const string &target, const size_t & max_distance){
    string output = "";
    size_t best_distance = max_distance;
    if(mRoot == nullptr) return output;
    stack<node*> nodes;
    nodes.push(mRoot);
    while(!nodes.empty()){
        node *u = nodes.top();
        nodes.pop();
        size_t du = lev(u->word,target);
        if(du < best_distance){
            // set the new best word and its edit distance
            best_distance = du;
            output = u->word;
        }
        if(du == 0) break;
        for(auto &it: u->children){
            // variable it is an iterator of pair{size_t, node*}
            // first = edit distance, second = child node
            if(abs(it->first - du) < best_distance) nodes.push(it->second);
        }
    }
    return output;
}
```

Figure 8: String-matching

2. String matching

Algorithm

Let's say we set the maximum distance allowed between the best match and the target word, by default it sets to +∞ .

First step: If the BK Tree is empty, then the best match is not found.

Second step: We will declare the best word as not found and best edit distance as maximum distance. We will push the root node into the empty stack (S).

Third step: We will loop doing this step until the stack S is empty. We will pop the node u from stack S and find the edit distance (d_u) between u's word and the target word. If its edit distance is less than the best distance, then we will set the best word and edit distance with the new record one. For each edge from the node u to any node v, if absolute of lev (u's word, v's word) - d_u < the best distance, then push node v into the stack S.

Last step: return the best word result.

Example (using Figure 6)

Let's say we want to search "whiten" and set the max distance as 2.

As the BK Tree is not empty. Then, we will start checking edit distance from the root node as node u.

Let d_u be the edit distance of (“whiten”, “what”) = 3. If $d_u < 2$ is false, then we do nothing. Checking each u’s child that if |its edit distance – 3 | < 2, then we need to check that node. We found that we need check {“water”, “wait”}. Let pick node “water” first and find its edit distance between it and “white” which equals to 3 so do nothing and node “water” has no child. Next, we will find the edit distance between node “wait” and “white” which equals to 2 so do nothing again. But node “wait” has a child that we need to consider. We found that we need to check {“whale”, “white”}. We will continue doing the same method for all nodes that we need to check. Finally, we will get the result as “white” which has the best edit distance as 1.

Construction of the BK Tree

Member variable of each node and BK Tree

Node

- word – collect word of that node
- parent – node pointer to the parent node
- children – a map which key refers to edit distance between node and a child, value refers to node pointer to the child node

BK Tree

- mRoot – node pointer to the root of the tree
- mSize – the size of the tree

```
1  class BK_Tree {
2  protected:
3      class node {
4          friend class BK_Tree;
5
6      protected:
7          std::string word;
8          node *parent;
9          std::map<size_t, node *> children;
10
11     public:
12         node() : word(""), parent(nullptr) {}
13         node(const std::string &word, node *parent) : word(word), parent(parent) {}
14     };
15
16     node *mRoot;
17     size_t mSize;
```

Figure 9: Member of the BK Tree class

1. Insertion operation

Algorithm

First step: If the tree is empty, then create a node with a selected word and mark it as a root node.

Second step: We will firstly select root node as node u and find the edit distance between node u and new node v with a new selected word. Next, we will consider node u to have node v as a child or not by condition shown below.

Condition each node can have exactly one child with the same edit distance.

Third step: If it passes a condition, then we will make it of as a child of node u. On the other hand, we will use the child with the same edit distance as node u. Then, looping the second step until the condition is fine.

```
node *insert(const string &word) {
    if (mRoot == nullptr) {
        // The BK Tree is empty add it to be a root node
        node *r = new node(word);
        mRoot = r;
        mSize++;
        return r;
    } else {
        size_t d; // Levenshtein distance
        // Start with root node
        node *u = mRoot;
        while (u != nullptr) {
            // lev(a,b) refers to edit distance between a and b
            d = lev(word, u->word);
            if (d == 0)
                return u; // it is the same word with u's word
            // u->childs is a map with size_t key and node* value
            // key of u->childs is a edit distance
            // value of u->childs is child node with key edit distance
            auto it = u->childs.find(d);
            if (it == u->childs.end()) {
                // Doesn't exist the child of the root with the same edit distance
                node *v = new node(word);
                u->children[d] = v;
                mSize++;
                return v;
            } else {
                // There exists the child with d edit distance
                // Set that child to u
                u = it->second;
            }
        }
    }
}
```

Figure 10: Insert

Time complexity (Without calculating Levenshtein distance)

The time complexity is $O(h)$ where h is a height of the tree.

Example (Making a BK Tree as Figure 1)

Let's we have a dictionary = {"book", "books", "boo", "cake", "boon", "cook", "cape", "cart"}

Firstly, we choose one word from the dictionary. Let's say we choose "book" as a root node.

Next, choose another one and find its edit distance from “book”. For example, we choose “books” then we find its edit distance from “book” = 1. Draw the edge from “book” node to “books” node with weight 1 as Figure 10.1. Let choose another word “cake” and its edit distance from “book” is 4. Draw the edge from “book” node to “cake” node with weight 4 as Figure 10.2.

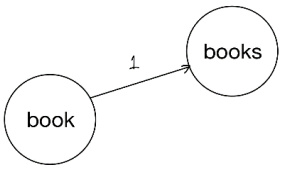

Figure 10.1

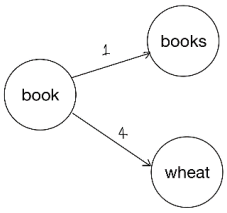

Figure 10.2

Let choose another wor d “boo” and its edit distance from “books” = 1. But there already exists the child with weight 1 from the root node, we will add “boo” node as a child of the same weight child of the root node. Before we add it, we need to calculate the edit distance again to be the weight of the edge as Figure 10.3. We can proceed this step to construct all word in dictionary and we will get the BK Tree as Figure 1.

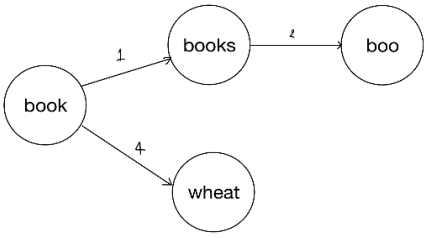

Figure 10.3

2.Deletion operation

There are 2 scenarios to remove a node from the tree.

Time complexity that we are going to show is not included the time complexity of Levenshtein distance.

First scenario: Removing a node and reconstructing a tree

In this scenario, we need to consider 2 cases e.g., remove non-root node and remove root node.

Case 1: Remove a non-root node

Firstly, we need to find a parent of that node and its child. Next, we re-add each node as a new node. This case is way more expensive because each insertion takes O(h) where h is a height of the tree. For all child of that node, it will be O(h*k) where k is the number of child nodes.

Case 2: Remove a root node

We just rebuild the tree from scratch without the previous root node. The time complexity is O(h*n) where h is a height of the tree and n is the number of nodes except the previous root node.

Second scenario: Marking the node as deleted and keep it in a tree

In this scenario, it’s better not to delete the node by marking it as deleted. This can be done in $\theta(1)$.

References

- https://en.wikipedia.org/wiki/Levenshtein_distance
- https://medium.com/@ethannam/understanding-the-levenshtein-distance-equation-for-beginners-c4285a5604f0
- https://www.geeksforgeeks.org/bk-tree-introduction-implementation/
- https://engineering.thinknet.co.th/%E0%B8%95%E0%B9%89%E0%B8%99%E0%B9%84%E0%B8%A1%E0%B9%89%E0%B8%97%E0%B8%B5%E0%B9%88%E0%B8%8A%E0%B8%B7%E0%B9%88%E0%B8%AD-burkhard-keller-tree-%E0%B8%AB%E0%B8%A3%E0%B8%B7%E0%B8%AD-bk-tree-%E0%B8%A1%E0%B8%B1%E0%B8%99%E0%B9%80%E0%B8%AD%E0%B8%B2%E0%B9%84%E0%B8%A7%E0%B9%89%E0%B9%83%E0%B8%8A%E0%B9%89%E0%B8%97%E0%B8%B3%E0%B8%AD%E0%B8%B0%E0%B9%84%E0%B8%A3%E0%B8%99%E0%B9%88%E0%B8%B0-a022188d41b2
- https://nullwords.wordpress.com/2013/03/13/the-bk-tree-a-data-structure-for-spell-checking/
- https://medium.com/future-vision/bk-trees-unexplored-data-structure-ec234f39052d
- https://en.wikipedia.org/wiki/Damerau%E2%80%93Levenshtein_distance
- https://en.wikipedia.org/wiki/Triangle_inequality
- https://stackoverflow.com/questions/42230648/deleting-a-node-in-a-bk-tree
- https://books.google.co.th/books?id=KTkWXsiPXR4C&pg=PA68&lpg=PA68&dq=query+time+complexity+in+BKT&source=bl&ots=YDdqKFy_7Y&sig=ACfU3U00Aesk6S-Q0lx2mSkDW8s5OZ-ZtA&hl=en&sa=X&ved=2ahUKEwjC3_P8xN30AhX-8HMBHXvdAHUQ6AF6BAgZEAM#v=onepage&q=query%20time%20complexity%20in%20BKT&f=false