CURSO DE ESTRUCCTURAS DE DATOS AVANZADOS

PRESENTADO POR: BELTRAN EDWIN MAMANI MAMANI

ÁRBOLES PARA FUNCIONES DE DISTANCIAS DISCRETAS

Para estudiar árboles para funciones de distancias discretas, es esencial comprender varios conceptos clave. Estos conceptos proporcionan la base teórica y práctica necesaria para trabajar con este tipo de estructuras de datos

EJEMPLO PRÁCTICO DE ÁRBOLES PARA FUNCIONES DE DISTANCIAS DISCRETAS

Problema: aplicación para corregir errores tipográficos en nombres de usuarios. Cuando un usuario introduce un nombre que no existe en la base de datos, queremos sugerir nombres similares.

FUNCIÓN DE DISTANCIA DE LEVENSHTEIN

La función de distancia de Levenshtein mide la diferencia entre dos cadenas, calculando el número mínimo de operaciones necesarias para transformar una cadena en otra. Las operaciones pueden ser inserciones, eliminaciones o sustituciones de un solo carácter.

```
1         función levenshteinDistance(a, b) {
2           matriz constante = [];
3           para ( sea i =  0 ; i <= b.length; i ++ ) {
4             matriz[i] = [i];
5           }
6           para ( sea j =  0 ; j <= a.length; j ++ ) {
7             matriz[ 0 ][j] = j;
8           }
9           para ( sea i =  1 ; i <= b.length; i ++ ) {
10            para ( sea j =  1 ; j <= a.length; j ++ ) {
11              si (b.charAt(i -  1 ) === a.charAt(j -  1 )) {
12                matriz[i][j] = matriz[i -  1 ][j -  1 ];
13              } demás {
14                matriz[i][j] =  Matemáticas .min(
15                  matrix[i -  1 ][j -  1 ] +  1 ,    // Sustitución
dieciséis                 Math .min(matrix[i][j -  1 ] +  1 , // Inserción
17                          matrix[i -  1 ][j] +  1 )   // Eliminación
18                );
19              }
20            }
21          }
22          matriz de retorno [b.length][a.length];
23        }
```

La matriz matrix se utiliza para almacenar los costos acumulados de las operaciones. Cada celda matrix[i][j] contiene la distancia de Levenshtein entre las subcadenas a[0..j-1] y b[0..i-1].
Las primeras filas y columnas se llenan con los índices, representando las distancias al convertir la cadena vacía en la subcadena correspondiente mediante inserciones o eliminaciones.
Llenado de la Matriz: Se comparan caracteres de las dos cadenas. Si los caracteres coinciden, no hay costo (se toma el valor diagonal). Si no, se calcula el mínimo entre inserción, eliminación o sustitución y se suma 1.

CONSTRUCCIÓN DEL ÁRBOL BK

Un Árbol BK (Burkhard-Keller Tree) es una estructura de datos eficiente para buscar elementos similares según una métrica de distancia, en este caso, la distancia de Levenshtein.

Clase Node:

```
1 clase Nodo {
2    constructor(punto) {
3       este .punto = punto;
4        este .hijos = {};
5    }
6 }
```

Representa un punto en el árbol. Cada nodo tiene un point (la cadena) y un diccionario children que mapea distancias a otros nodos hijos.

Clase BKTree:

```
1          clase BKTree {
2             constructor(funcionDistancia) {
3                este .root =  null ;
4                 este .distanceFunc = distanceFunc;
5             }
6
7             añadir(punto) {
8                si ( este . raíz ===  null ) {
9                   este . raíz =  nuevo Nodo(punto);
10               } else {
11                  este ._add( este .root, punto);
12               }
13            }
14
15            _add(nodo, punto) {
dieciséis        const dist =  this .distanceFunc(nodo.punto, punto);
17               si (nodo.niños [dist] ===  indefinido ) {
```

```
18                  nodo.children[dist] =  nuevo Nodo(punto);
19              } else {
20                  this ._add(node.children[dist], punto);
21              }
22          }
23      }
```

Inicializa el árbol con una función de distancia y la raíz como null.

Agrega un punto al árbol. Si el árbol está vacío, el nuevo punto se convierte en la raíz. De lo contrario, se llama a add.

Calcula la distancia entre el nuevo punto y el punto en el nodo actual, y coloca el nuevo punto en el lugar adecuado en el árbol.

BÚSQUEDA DE NOMBRES SIMILARES

La búsqueda en un Árbol BK se realiza explorando los nodos en función de la distancia al punto de consulta.

Implementación de la búsqueda:

```
1       clase BKTree {
2           // Métodos anteriores ...
3
4           buscar(punto, distanciamáxima) {
5               const resultados = [];
6               este ._search( este .root, punto, maxDistance, resultados);
7               devolver resultados;
8           }
9
10          _search(nodo, punto, distanciamáxima, resultados) {
11              si ( ! nodo) retorna ;
12
13              const dist =  this .distanceFunc(nodo.punto, punto);
14              if (dist <= maxDistance) {
15                  resultados.push({ punto : nodo.punto, distancia : dist });
dieciséis       }
17              para ( sea d =  Math .max( 0 , dist - maxDistance); d <= dist +
18      maxDistance; d ++ ) {
19                  if (node.children[d] !==  undefined ) {
20                      this ._search(node.children[d], punto, maxDistance, resultados);
21                  }
22              }
23          }
```

```
}
```

Inicia la búsqueda en el árbol. Llama a search para explorar recursivamente los nodos. Calcula la distancia entre el punto de consulta y el nodo actual. Si la distancia está dentro del máximo permitido (maxDistance), añade el nodo a los resultados. Luego, explora los hijos dentro del rango de distancias permitidas.