**مستندات پروژه: الگوریتم‌های جستجو با استفاده از بلوم فیلتر**

**مقدمه**

این پروژه شامل پیاده‌سازی چندین الگوریتم جستجو و ارزیابی زمان عملکرد آن‌ها بر روی مجموعه‌ای از یوزرنیم‌ها است. الگوریتم‌های پیاده‌سازی شده به شرح زیر هستند:

1. **جستجوی خطی**
2. **جستجوی باینری**
3. **جستجوی هش**
4. **جستجوی بلوم فیلتر**

یوزرنیم‌ها از یک فایل بارگذاری می‌شوند و به وسیله الگوریتم‌های مختلف جستجو می‌شوند. زمان هر عملیات جستجو محاسبه و در یک فایل ذخیره می‌شود.

**پیش‌نیازها**

* **فایل یوزرنیم‌ها**: یک فایل متنی شامل لیستی از یوزرنیم‌ها (usernames.txt).

**ویژگی‌ها**

**جستجوی خطی**

جستجوی خطی از ابتدا تا انتهای پایگاه داده یوزرنیم‌ها را جستجو می‌کند و بررسی می‌کند که آیا یوزرنیم وارد شده با یکی از یوزرنیم‌ها تطابق دارد یا خیر. زمان جستجو از شروع تا پایان محاسبه می‌شود.

* **پیچیدگی زمانی**: O(n) که در آن n تعداد یوزرنیم‌ها است.
* **استفاده**: مناسب برای پایگاه داده‌های کوچک یا داده‌های بدون ترتیب خاص.

**2. جستجوی باینری**

جستجوی باینری بر روی یک لیست مرتب از یوزرنیم‌ها اعمال می‌شود. این الگوریتم محدوده جستجو را به‌طور مداوم نصف می‌کند تا زمانی که یوزرنیم پیدا شود.

* **پیچیدگی زمانی**: O(log n)، که n تعداد یوزرنیم‌ها است.
* **استفاده**: مناسب برای پایگاه داده‌های بزرگ و مرتب.

**3. جستجوی هش**

در این روش از یک هش‌مپ برای ذخیره‌سازی یوزرنیم‌ها استفاده می‌شود. جستجو بررسی می‌کند که آیا یوزرنیم وارد شده در هش‌مپ وجود دارد یا خیر.

* **پیچیدگی زمانی**: O(1)، در صورتی که از یک تابع هش مناسب استفاده شود.
* **استفاده**: سریع برای عملیات جستجو در پایگاه داده‌های بزرگ.

**4. جستجوی بلوم فیلتر**

بلوم فیلتر یک ساختار داده احتمالاتی است که بررسی می‌کند آیا یک عنصر *ممکن است* در مجموعه باشد یا *قطعاً* نیست. این فیلتر از دو تابع هش برای نگاشت یوزرنیم به یک بیت‌ست استفاده می‌کند.

* **پیچیدگی زمانی**: O(k)، که k تعداد توابع هش است (در اینجا 2 تابع هش استفاده شده است).
* **استفاده**: کارآمد برای بررسی عضویت در مجموعه‌های بزرگ.

**شرح کد**

**1. توابع هش**

### 1. تابع getHashFunctionCount

این تابع تعداد توابع هش مورد نیاز برای فیلتر بلوم را محاسبه می‌کند. ورودی‌ها شامل تعداد عناصر (n) و نرخ خطای مثبت کاذب (p) است. از فرمولی برای محاسبه تعداد بهینه توابع هش استفاده می‌شود.

### 2. تابع getBloomFilterSize

این تابع اندازه فیلتر بلوم را بر اساس تعداد عناصر (n) و نرخ خطای مثبت کاذب (p) محاسبه می‌کند. هدف این است که فیلتر بلوم به‌اندازه کافی بزرگ باشد تا خطای مثبت کاذب را کاهش دهد.

### 3. تابع hash1

این تابع یک هش ساده با استفاده از ضرب و جمع بر اساس مقدار ثابت 31 برای یک کلید (key) تولید می‌کند. این مقدار هش سپس با اندازه فیلتر بلوم محدود می‌شود.

### 4. تابع hash2

این تابع مشابه hash1 است، ولی از ضریب ثابت 17 برای تولید هش استفاده می‌کند. استفاده از دو تابع هش با ضرایب مختلف، احتمال برخورد هش‌ها را کاهش می‌دهد.

تمام این توابع برای پیاده‌سازی فیلتر بلوم استفاده می‌شوند تا جستجوها سریع‌تر و فضای ذخیره‌سازی بهینه‌تر باشد.

*// Function to calculate the number of hash functions*

int getHashFunctionCount(int n, float p) {

    return round((bloomFilter.size() / n) \* log(2));

}

*// Function to calculate the size of the Bloom filter*

int getBloomFilterSize(int n, float p) {

    return round(- (n \* log(p)) / (log(2) \* log(2)));

}

int hash1(const string& key, int filterSize) {

    int hash = 0;

    for (char c : key) {

        hash = (hash \* 31 + c) % filterSize;

    }

    return hash;

}

int hash2(const string& key, int filterSize) {

    int hash = 0;

    for (char c : key) {

        hash = (hash \* 17 + c) % filterSize;

    }

    return hash;

}

**2. جستجوها**

**جستجوی خطی**

در جستجوی خطی، برای هر یوزرنیم در پایگاه داده، بررسی می‌شود که آیا یوزرنیم وارد شده با آن تطابق دارد یا خیر.

void linearSearch(const vector<string>& db, const string& username) {

    auto start = chrono::steady\_clock::now(); *// Start time*

    for (const auto& user : db) {

        if (user == username) {

            auto end = chrono::steady\_clock::now(); *// End time*

            cout << "Username found (Linear Search).\n";

            chrono::duration<double> duration = end - start;

            cout << "Time taken: " << fixed << setprecision(10) << duration.count() << " seconds.\n";

            TimeTaken = duration.count(); *// Time in seconds*

            return;

        }

    }

    auto end = chrono::steady\_clock::now(); *// End time after loop*

    cout << "Username not found (Linear Search).\n";

    chrono::duration<double> duration = end - start;

    cout << "Time taken: " << fixed << setprecision(10) << duration.count() << " seconds.\n";

    TimeTaken = duration.count(); *// Time in seconds*

}

**جستجوی باینری**

در جستجوی باینری، ابتدا پایگاه داده به‌صورت صعودی مرتب می‌شود و سپس برای پیدا کردن یوزرنیم با استفاده از الگوریتم باینری جستجو عمل می‌شود.

sort(sortedDatabase.begin(), sortedDatabase.end());

void binarySearch(const vector<string>& sortedDb, const string& username) {

    auto start = chrono::steady\_clock::now(); *// Start time*

    int left = 0;

    int right = sortedDb.size() - 1;

    bool found = false;

    while (left <= right) {

        int mid = left + (right - left) / 2;

        if (sortedDb[mid] == username) {

            found = true;

            break;

        }

        if (sortedDb[mid] < username) {

            left = mid + 1;

        } else {

            right = mid - 1;

        }

    }

    auto end = chrono::steady\_clock::now(); *// End time*

    if (found) {

        cout << "Username found (Binary Search).\n";

    } else {

        cout << "Username not found (Binary Search).\n";

    }

    chrono::duration<double> duration = end - start;

    cout << "Time taken: " << fixed << setprecision(10) << duration.count() << " seconds.\n";

    TimeTaken = duration.count(); *// Time in seconds*

}

**جستجوی هش**

در جستجوی هش، از یک هش‌مپ برای ذخیره یوزرنیم‌ها استفاده می‌شود و جستجو بررسی می‌کند که آیا یوزرنیم وارد شده در هش‌مپ وجود دارد یا خیر.

*// Fill hash map*

    for (const auto& user : database) {

        hashMap[user] = true;

    }

void hashSearch(const unordered\_map<string, bool>& hashMap, const string& username) {

    auto start = chrono::steady\_clock::now(); *// Start time*

    bool ch = hashMap.find(username) != hashMap.end();

    auto end = chrono::steady\_clock::now(); *// End time*

    if (ch) {

        cout << "Username found (Hash Search).\n";

    } else {

        cout << "Username not found (Hash Search).\n";

    }

    chrono::duration<double> duration = end - start;

    cout << "Time taken: " << fixed << setprecision(10) << duration.count() << " seconds.\n";

    TimeTaken = duration.count(); *// Time in seconds*

}

**جستجوی بلوم فیلتر**

در این بخش از کد، عملکردهای مختلفی برای محاسبه اندازه بلوم فیلتر و تعداد توابع هش انجام می‌شود:

1. **محاسبه اندازه بلوم فیلتر و تعداد توابع هش**:
   * getBloomFilterSize(database.size(), 0.1) اندازه بلوم فیلتر را بر اساس تعداد عناصر موجود در database و احتمال خطای مثبت کاذب (0.1 که به معنی 10 درصد احتمال خطای مثبت کاذب است) محاسبه می‌کند.
   * resize(filterSize) اندازه و ظرفیت bloomFilter را طبق اندازه محاسبه‌شده تنظیم می‌کند.
2. **محاسبه تعداد توابع هش**:
   * getHashFunctionCount(database.size(), 0.1) تعداد توابع هش مورد نیاز برای دستیابی به احتمال خطای مثبت کاذب مورد نظر را محاسبه می‌کند.
3. **پر کردن بلوم فیلتر**:
   * بلوم فیلتر با مقادیر true پر می‌شود تا حضور هر عنصر در دیتابیس را نشان دهد.
   * برای هر کاربر، چندین تابع هش (hash1 و hash2) مقدار هش را تولید می‌کنند و ایندکس‌های مربوطه در بلوم فیلتر به true تغییر می‌کنند تا حضور آن عنصر را در فیلتر نشان دهند.

این بخش از کد برای اضافه کردن داده‌ها به بلوم فیلتر و انجام جستجوهای احتمالی استفاده می‌شود.

*// Calculate the number of hash functions and the size of the Bloom filter*

    int filterSize = getBloomFilterSize(database.size(), 0.1);

    bloomFilter.resize(filterSize); *// Now using resize for vector<bool>*

*// Number of hash functions*

    int hashCount = getHashFunctionCount(database.size(), 0.1);

*// Fill the Bloom filter*

    for (const auto& user : database) {

        for (int i = 0; i < hashCount; ++i) {

            int h = hash1(user, filterSize) + i \* hash2(user, filterSize); *// Combining hash functions*

            bloomFilter[h % filterSize] = true;

        }

    }

void bloomFilterSearch(const string& username) {

    auto start = chrono::steady\_clock::now(); *// Start time*

    int h1 = hash1(username);

    int h2 = hash2(username);

    bool ch = bloomFilter[h1] && bloomFilter[h2];

    auto end = chrono::steady\_clock::now(); *// End time*

    if (ch) {

        cout << "Username might be in the database (Bloom Filter Search).\n";

    } else {

        cout << "Username definitely not in the database (Bloom Filter Search).\n";

    }

    chrono::duration<double> duration = end - start;

    cout << "Time taken: " << fixed << setprecision(10) << duration.count() << " seconds.\n";

    TimeTaken = duration.count(); *// Time in seconds*

}

**3. بارگذاری یوزرنیم‌ها از فایل**

یوزرنیم‌ها از فایل usernames.txt بارگذاری می‌شوند و برای استفاده در الگوریتم‌های جستجو در پایگاه داده ذخیره می‌شوند.

vector<string> database;

    loadUsernamesFromFile(database); *// Load usernames from file*

void loadUsernamesFromFile(vector<string>& database) {

    ifstream file("usernames.txt");

    string username;

    while (getline(file, username)) {

        database.push\_back(username);

    }

    file.close();

}

**4. محاسبه زمان**

زمان هر جستجو از لحظه‌ای که جستجو آغاز می‌شود تا پایان آن محاسبه می‌شود. زمان‌ها سپس در یک فایل ذخیره می‌شوند.

resultFile << "Searching for: " << username << endl

           << "Time taken: " << fixed << setprecision(10) << TimeTaken << " seconds" << endl

           << "Search method: " << choice << endl; *// Save search details to file*

        resultFile << "----------------------------------------\n"; *// Separator*

**فایل‌ها**

1. **usernames.txt**: فایل ورودی حاوی یوزرنیم‌ها که برای جستجو بارگذاری می‌شود.
2. **search\_results.txt**: فایل خروجی که نتایج جستجو و زمان‌های مربوطه در آن ذخیره می‌شوند.

**نتایج آزمایش جستجو در دیتابیس با استفاده از روش‌های مختلف**

در این آزمایش، چندین روش جستجو برای بررسی عملکرد در جستجوی نام‌های کاربری در یک دیتابیس بزرگ استفاده شده است. نتایج این آزمایش‌ها شامل مقایسه زمان‌های اجرای جستجو با استفاده از چهار روش مختلف است: جستجوی خطی (Linear Search)، جستجوی دودویی (Binary Search)، جستجوی هشی (Hash Search) و جستجوی با فیلتر بلوم (Bloom Filter Search).

برای بررسی صحت و دقت این روش‌ها، 10000 نام کاربری به طور تصادفی از دیتابیس استخراج شده و برای هر کدام از این روش‌ها زمان اجرای جستجو اندازه‌گیری شده است. همچنین، تمام نتایج به صورت دقیق و با دقت بالا در فایل‌های جداگانه ذخیره شدند تا امکان مقایسه و ارزیابی بهتر فراهم شود.

نتایج این آزمایش‌ها در این مستندات آورده شده است، و هدف از این آزمایش‌ها، ارزیابی کارایی و سرعت مختلف این روش‌ها در جستجو در دیتابیس‌های بزرگ است.

**نتایج بدست امده**

**تعداد 10000 یوزرنیم رندوم و بلوم فیلتر درست اشتباه 10 درصد**

Linear Search: 0.00594976

Binary Search: 6.25191e-06

Hash Search: 9.53122e-06

Bloom Filter Search: 4.77492e-06

**تعداد 10000 یوزرنیم رندوم و بلوم فیلتر درست اشتباه 1 درصد**

Linear Search: 0.00433522

Binary Search: 5.00674e-06

Hash Search: 5.84497e-06

Bloom Filter Search: 6.09603e-06

**تعداد 1000 یوزرنیم رندوم و بلوم فیلتر درست اشتباه 10 درصد**

Linear Search: 0.00350869

Binary Search: 4.8833e-06

Hash Search: 4.2329e-06

Bloom Filter Search: 3.957e-06

**تعداد 1000 یوزرنیم رندوم و بلوم فیلتر درست اشتباه 1 درصد**

Linear Search: 0.00386266

Binary Search: 5.3633e-06

Hash Search: 3.7993e-06

Bloom Filter Search: 6.0847e-06

**تابع جنریت رندوم یوزرنیم و جست و جو با 4 روش**

void generateRandomUsernamesFromFile(const vector<string>& database, int count, const vector<string>& sortedDatabase, const unordered\_map<string, bool>& hashMap) {

    random\_device rd;

    mt19937 gen(rd());

    uniform\_int\_distribution<> dis(0, database.size() - 1);

    vector<string> randomUsernames;

    for (int i = 0; i < count; ++i) {

        int index = dis(gen);

        randomUsernames.push\_back(database[index]);

    }

    double one,two,three,four;

    int c = 1;

    ofstream timeFile("execution\_times.csv", ios::app);

    for (const auto& username : randomUsernames) {

        timeFile << c++ << endl;

        linearSearch(database, username);

*//timeFile << username << "," << database.size() << "," << fixed << setprecision(10) << TimeTaken << "," << "Linear Search" << endl;*

        one += TimeTaken;

        binarySearch(sortedDatabase, username);

*//timeFile << username << "," << database.size() << "," << fixed << setprecision(10) << TimeTaken << "," << "Binary Search" << endl;*

        two += TimeTaken;

        hashSearch(hashMap, username);

*//timeFile << username << "," << database.size() << "," << fixed << setprecision(10) << TimeTaken << "," << "Hash Search" << endl;*

        three += TimeTaken;

        bloomFilterSearch(username, bloomFilter.size(), getHashFunctionCount(database.size(), 0.1));

*//timeFile << username << "," << database.size() << "," << fixed << setprecision(10) << TimeTaken << "," << "Bloom Filter Search" << endl;*

        four += TimeTaken;

    }

    timeFile << "Linear Search: " << one/count << endl

             << "Binary Search: " << two/count << endl

             << "Hash Search: " << three/count << endl

             << "Bloom Filter Search: " << four/count << endl;

    timeFile.close();

    cout << "Search results for 10000 random usernames have been saved to 'execution\_times.csv'.\n";

}

**نتیجه‌گیری**

با استفاده از این چهار الگوریتم، مقایسه‌ای بین الگوریتم‌های جستجو از نظر **زمان عملکرد** و **کارایی** در شرایط مختلف انجام می‌شود. **جستجوی خطی** مناسب برای داده‌های کوچک یا زمانی است که داده‌ها مرتب نشده‌اند، اما در مجموعه‌های بزرگ ناکارآمد خواهد بود. **جستجوی دودویی** فقط برای مجموعه‌های مرتب‌شده مناسب است و در آن‌ها کارایی بهتری دارد. در حالی که **جستجوی هش** برای مجموعه‌های بزرگ و زمانی که داده‌ها به‌طور مکرر جستجو می‌شوند بسیار مؤثر است، **جستجوی فیلتر بلوم** با استفاده از حافظه کم و زمان جستجوی سریع، به‌ویژه در سیستم‌هایی با حجم داده بالا و محدودیت حافظه، گزینه‌ای عالی است.