Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Курсовой проект по дисциплине «Информационный поиск»

Студент: И. А. Мариничев
Преподаватель: А. А. Кухтичев
Группа: М8О-408Б-19

Дата:
Оценка:
Подпись:

Москва, 2023

Курсовой проект на тему «Поисковые подсказки»

Нужно реализовать вывод поисковых подсказок пользователю:

- Поиск по подсказкам должен быть полнотекстовым, то есть должен уметь искать по текстам подсказок все слова из пользовательского запроса в любой их последовательности. Например, если пользователь ввёл запрос «смотреть», а в базе мы располагаем подсказками, то пользователь должен увидеть все три, несмотря на то, что слово «смотреть» находится в разных местах этих подсказок.
- Запрос пользователя может быть неполным, пока набирает его на клавиатуре. Поэтому искать нужно не по словам, а по их префиксам. Для предыдущего примера мы должны увидеть все три подсказки не только по целому слову «смотреть», но и для любой его префиксной части. Например, «смотр».
- Разнообразие возможных запросов велико, и система должна уметь искать среди десятков миллионов подсказок.
- Скорость реакции крайне важна, поэтому мы хотим выдавать ответ за считанные миллисекунды.
- Для отказоустойчивости, а, следовательно, ради простоты реализации и отладки, нам крайне желательно иметь в основе сервиса некую идею, которая была бы крайне проста и элегантна.

1 Описание

Разберём нахождение поисковых подсказок на примере:

1. Предположим, что в нашем распоряжении есть указанный выше индекс, и пользователь уже ввёл часть запроса. Следующая введённая буква на клавиатуре, и вот мы получили неполный запрос «omnia v».

2. Разбиваем запрос на слова-префиксы: получаем «omnia» и «v».

3. По аналогии с поисковой системой, сначала по заданным словам-префиксам находим списки id подсказок в обратном индексе. Обратим внимание, что наш обратный индекс состоит из двух частей:

   - префиксное дерево (оно же «trie», оно же «бор»), содержащее слова, которые мы «выпотрошили» из текстов подсказок;
   - списки id подсказок — индексы текстов подсказок в прямом индексе, о котором речь пойдёт ниже. Списки отсортированы по возрастанию значения id и находятся в тех вершинах, где заканчиваются слова.

   Надо сказать, что trie хорош для нас по двум причинам:

   - по нему можно найти любой префикс слова за время O(log(n)), где n — длина префикса;
   - для заданного префикса легко определить все варианты его продолжений.

   Итак, для каждого префикса углубляемся вниз по дереву и оказываемся в промежуточных узлах. Теперь нужно получить правильные списки id подсказок.

4. Объединяем списки всех дочерних узлов. Для чего? По аналогии с обычным поисковиком, мы должны бы пересечь списки id для каждого префикса, однако есть два «НО»:

   - во-первых, не всякий узел содержит список id подсказок, а только те узлы, в которых закачивается целое слово;
   - а во-вторых, каждый узел дерева имеет некое продолжение, за исключением листовых узлов. Это значит, что продолжений у одного префикса может быть целое множество, и эти продолжения нужно учесть.

   Поэтому, прежде чем найти пресечение, нужно «просуммировать» списки id подсказок для каждого отдельно взятого префикса. Таким образом, для каждого префикса обходим дерево рекурсивно в глубину, начиная с того узла, где мы остановились в дереве по данному префиксу, и конструируем объединение

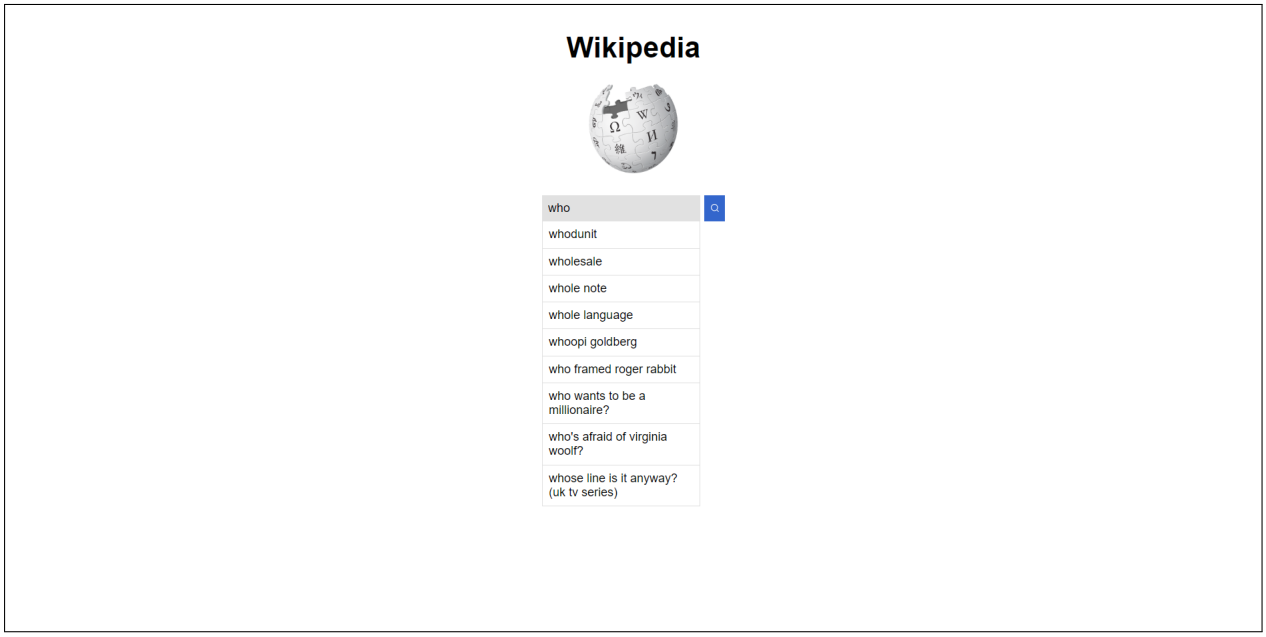
всех списков его дочерних узлов алгоритмом слияния. На рисунке узлы, в которых мы остановились по префиксу, помечены красным кружком, а операция объединения помечена значком «U».

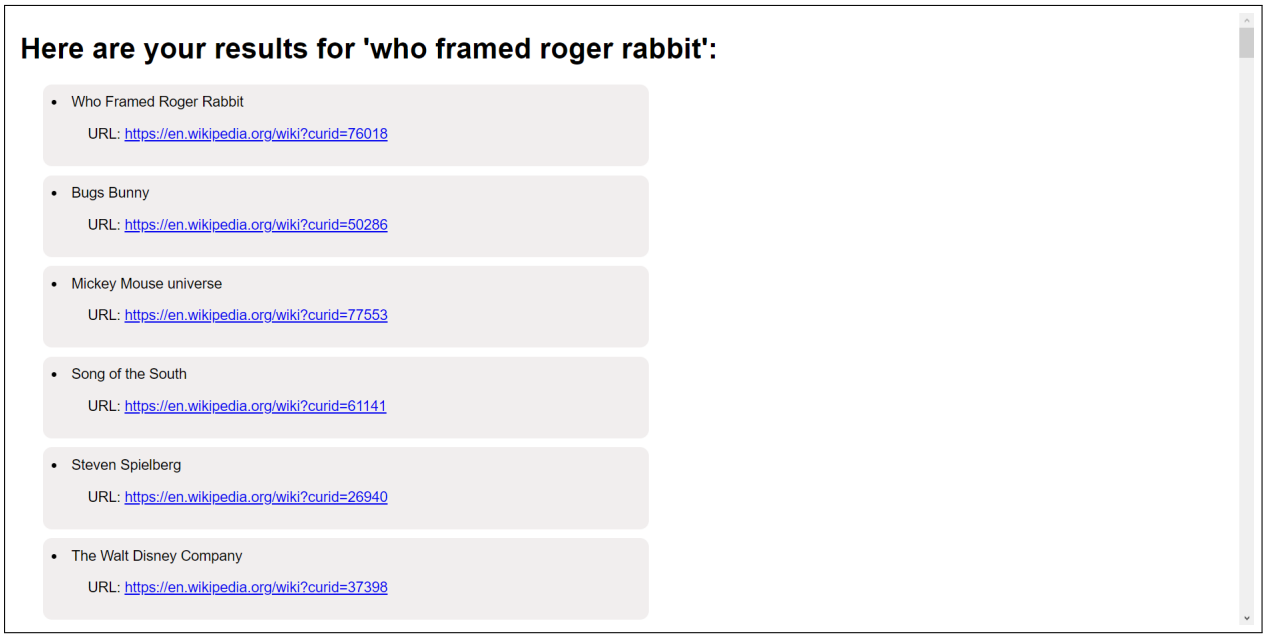
5. Теперь пересекаем синтетические списки-объединения каждого из префиксов. Надо сказать, что различных алгоритмов пересечения сортированных списков просто море, и каждый подходит больше для различных типов последовательностей. Один из самых эффективных — алгоритм Рикардо Баеза-Ятеса и Алехандро Салингера. В боевых подсказках мы используем свой алгоритм, который наиболее подходит для решения конкретной задачи, однако алгоритм Баеза-Ятеса-Салингера был для нас в своё время вдохновляющим.

6. Теперь по найденным id ищем тексты подсказок. Прямой индекс в нашем случае ничем не отличается от прямого индекса любой поисковой системы, то есть представляет собой простой массив (вектор) строк. Кроме текстов подсказок здесь может быть любая дополнительная информация. В частности, здесь мы храним веса популярности.

Итак, к концу шестого этапа мы уже имеем все подсказки, в которых есть все префиксы из пользовательского запроса. Очевидно, что на деле количество подсказок, которые мы получаем к этому этапу, может быть очень много — тысячи или даже сотни тысяч, а «подсказать» пользователю нам нужно только лучшие. Такую задачу решает другой алгоритм — алгоритм ранжирования.

Внешний вид веб-сервиса с поисковыми подсказками:





2 Исходный код

Ниже приведен код реализующий нахождение и вывод пользователю поисковых подсказок:

```
// autocomplete.js
// we start with the TrieNode
const TrieNode = function (key) {
    // the "key" value will be the character in sequence
    this.key = key;

    // we keep a reference to parent
    this.parent = null;

    // we have hash of children
    this.children = {};

    // check to see if the node is at the end
    this.end = false;

    this.getWord = function () {
        let output = [];
        let node = this;

        while (node !== null) {
            output.unshift(node.key);
            node = node.parent;
        }

        return output.join('');
    };
}

const Trie = function () {
    this.root = new TrieNode(null);
    // this.topK = 9;

    // inserts a word into the trie.
    this.insert = function (word) {
        let node = this.root; // we start at the root

        // for every character in the word
        for (let i = 0; i < word.length; i++) {
            // check to see if character node exists in children.
            if (!node.children[word[i]]) {
                // if it doesn't exist, we then create it.
                node.children[word[i]] = new TrieNode(word[i]);

                // we also assign the parent to the child node.
```

```
            node.children[word[i]].parent = node;
        }

        // proceed to the next depth in the trie.
        node = node.children[word[i]];

        // finally, we check to see if it's the last word.
        if (i == word.length - 1) {
            // if it is, we set the end flag to true.
            node.end = true;
        }
    }
};

// check if it contains a whole word.
this.contains = function (word) {
    let node = this.root;

    // for every character in the word
    for (let i = 0; i < word.length; i++) {
        // check to see if character node exists in children.
        if (node.children[word[i]]) {
            // if it exists, proceed to the next depth of the trie.
            node = node.children[word[i]];
        } else {
            // doesn't exist, return false since it's not a valid word.
            return false;
        }
    }

    // we finished going through all the words, but is it a whole word?
    return node.end;
};

// returns every word with given prefix
this.find = function (prefix) {
    let node = this.root;
    let output = [];

    // for every character in the prefix
    for (let i = 0; i < prefix.length; i++) {
        // make sure prefix actually has words
        if (node.children[prefix[i]]) {
            node = node.children[prefix[i]];
        } else {
            // there's none. just return it.
            return output;
        }
    }
```

```
94
95          // recursively find all words in the node
96          findAllWords(node, output);
97
98          return output;
99      };
100
101     // recursive function to find all words in the given node.
102     const findAllWords = (node, arr) => {
103         // base case, if node is at a word, push to output
104         if (node.end) {
105             arr.unshift(node.getWord());
106         }
107
108         // iterate through each children, call recursive findAllWords
109         for (let child in node.children) {
110             findAllWords(node.children[child], arr);
111             // if (arr.length > this.topK) {
112             // break;
113             // }
114         }
115     }
116
117     // removes a word from the trie.
118     this.remove = function (word) {
119         let root = this.root;
120
121         if (!word) return;
122
123         // recursively finds and removes a word
124         const removeWord = (node, word) => {
125
126             // check if current node contains the word
127             if (node.end && node.getWord() === word) {
128
129                 // check and see if node has children
130                 let hasChildren = Object.keys(node.children).length > 0;
131
132                 // if has children we only want to un-flag the end node that marks the
                        end of a word.
133                 // this way we do not remove words that contain/include supplied word
134                 if (hasChildren) {
135                     node.end = false;
136                 } else {
137                     // remove word by getting parent and setting children to empty
                            dictionary
138                     node.parent.children = {};
139                 }
140
```

```
            return true;
        }

        // recursively remove word from all children
        for (let key in node.children) {
            removeWord(node.children[key], word)
        }

        return false
    };

    // call remove word on root node
    removeWord(root, word);
};
}


const inputEl = document.querySelector("#autocomplete-input");

inputEl.addEventListener("input", onInputChange);
getTitleData();

const trie = new Trie();

function readTextFile(file, callback) {
    var rawFile = new XMLHttpRequest();
    rawFile.overrideMimeType("application/json");
    rawFile.open("GET", file, true);
    rawFile.onreadystatechange = function () {
        if (rawFile.readyState === 4 && rawFile.status == "200") {
            callback(rawFile.responseText);
        }
    }
    rawFile.send(null);
}

async function getTitleData() {
    readTextFile("/static/id.json", function (text) {
        const data = JSON.parse(text);
        data.forEach((doc) => {
            if (doc.title.length > 2) {
                trie.insert(doc.title.toLowerCase());
            }
        });
    });
}

function onInputChange() {
    removeAutocompleteDropdown();
```

```
190
191     const value = inputEl.value.toLowerCase();
192     if (value.length === 0) return;
193
194     // const filteredNames = trie.find(value);
195     const filteredNames = trie.find(value).sort(function (a, b) {
196         return a.length - b.length || a.localeCompare(b);
197     }).slice(0, 10);
198
199     createAutocompleteDropdown(filteredNames);
200 }
201
202 function createAutocompleteDropdown(list) {
203     const listEl = document.createElement("ul");
204     listEl.className = "autocomplete-list";
205     listEl.id = "autocomplete-list";
206
207     list.forEach((title) => {
208         const listItem = document.createElement("li");
209
210         const coutryButton = document.createElement("button");
211         coutryButton.innerHTML = title;
212         coutryButton.addEventListener("click", onTitleButtonClick);
213         listItem.appendChild(coutryButton);
214
215         listEl.appendChild(listItem);
216     });
217
218     document.querySelector("#autocomplete-wrapper").appendChild(listEl);
219 }
220
221 function removeAutocompleteDropdown() {
222     const listEl = document.querySelector("#autocomplete-list");
223     if (listEl) listEl.remove();
224 }
225
226 function onTitleButtonClick(event) {
227     event.preventDefault();
228
229     const buttonEl = event.target;
230     inputEl.value = buttonEl.innerHTML;
231
232     removeAutocompleteDropdown();
233 }
```

3 Выводы

Выполнив курсовой проект по дисциплине «Информационный поиск», я реализовал алгоритм нахождения поисковых подсказок, смог вывести их пользоваетлю в веб-сервисе. Конечно, представленный здесь алгоритм описан в самом общем виде, и многие вопросы остались за рамками. Над чем ещё было бы полезно подумать разработчику подсказок:

- алгоритм ранжирования подсказок с учётом позиций слов;
- инвертирование языковой раскладки клавиатуры: «ghbdtn» -> «привет»;
- исправление опечаток в пользовательском запросе;
- конструирование концовки запроса пользователя для случая, когда нам нечего подсказать из базы заготовленных подсказок: «что пела в середине 80х алла пу» -> «что пела в середине 80х алла пугачева»;
- учёт географии пользователя: «кинотеатр» -> для пользователя из Саратова не стоит подсказывать московские и питерские кинотеатры, которые ищут часто из-за большой аудитории пользователей;
- масштабирование алгоритма на 2, 3 и более серверов, когда мы захотим добавить 100-200-500 миллионов подсказок и упрёмся в ресурсы памяти и процессора;
- и прочее, и прочее, что только можно ориентировать на нашего любимого пользователя и на наши потребности.

Список литературы

[1] Маннинг, Рагхаван, Шютце *Введение в информационный поиск* — Издательский дом «Вильямс», 2011. Перевод с английского: доктор физ.-мат. наук Д. А. Клюшина — 528 с. (ISBN 978-5-8459-1623-4 (рус.))

[2] *Поисковые подсказки изнутри*
URL: `https://habr.com/ru/company/vk/blog/267469/` (дата обращения: 05.01.2023).

[3] *Autocomplete input dropdown | HTML, CSS, JS*
URL: `https://youtu.be/OXd_wv7Qi4g` (дата обращения: 06.01.2023).