Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Отчет по лабораторной работе №7 по курсу "Анализ алгоритмов"

Тема Эффективный поиск по словарю

Студент Саркисов А.С.

Группа ИУ7-53Б

Оценка (баллы) _____

Преподаватели Волкова Л.Л., Строганов Ю.В.

Москва — 2021 г.

Оглавление

Введение **3**

1 Аналитическая часть **4**

1.1 Алгоритм полного перебора 4

1.2 Частотный анализ 4

1.3 Алгоритм эффективного поиска по словарю 5

2 Конструкторская часть **6**

2.1 Структура ПО 6

3 Технологическая часть **7**

3.1 Требования к ПО 7

3.2 Средства реализации 7

3.3 Листинг кода 8

4 Исследовательская часть **10**

4.1 Технические характеристики 10

4.2 Время выполнения алгоритмов 10

4.3 Производительность алгоритмов 11

Заключение **12**

Литература **13**

Введение

Цель работы: изучить способ эффективного поиска по словарю. Необходимо сравнить времени работы алгоритма полного перебора с алгоритмом эффективного поиска по словарю.

В ходе лабораторной работы предстоит:

- реализовать алгоритм полного перебора;
- реализовать алгоритм эффективного поиска по словарю;
- изучить применение частотного анализа в задаче эффективного поиска по словарю;
- сравнить алгоритм эффективного поиска с полным перебором по времени работы.

1 Аналитическая часть

Поиск по словарю является задачей, которая стоит во многих сферах программирования. Поиск по словарю является задачей, которую требуется решать быстро, поэтому необходимы методы для оптимизации данной задачи.

1.1 Алгоритм полного перебора

Алгоритмом полного перебора [1] называют метод решения задачи, при котором по очереди рассматриваются все возможные варианты. В нашем случае мы последовательно будем перебирать элементы словаря до тех пор, пока не найдём нужный. Сложность такого алгоритма зависит от количества всех возможных решений, а время решения может потребовать экспоненциального времени работы.

Пусть алгоритм нашёл элемент на первом сравнении, тогда, в лучшем случае, будет затрачено k_0+k_1 операций, на втором - k_0+2k_1, на последней - k_0+Nk_1. Тогда средняя трудоёмкость может быть рассчитана по формуле 1.1, где Ω – множество всех возможных случаев.

$$\begin{aligned}\sum_{i\in\Omega} p_i \cdot f_i =&(k_0+k_1)\frac{1}{N+1}+(k_0+2k_1)\frac{1}{N+1}+\\ &+(k_0+3k_1)\frac{1}{N+1}+(k_0+Nk_1)\frac{1}{N+1}+(k_0+Nk_1)\frac{1}{N+1}=\\ &=k_0\frac{N+1}{N+1}+k_1+\frac{1+2+...+N+N}{N+1}=\\ &=k_0+k_1(\frac{N}{N+1}+\frac{N}{2})=k_0+k_1(1+\frac{N}{2}-\frac{1}{N+1})\end{aligned} \tag{1.1}$$

1.2 Частотный анализ

Прежде чем перейти к рассмотрению алгоритма эффективного поиска по словарю стоит рассмотреть процедуру частотного анализа [2], которая

лежит в основе данного алгоритма. Чтобы провести частотный анализ нужно взять первый элемент каждого значения в словаре по ключу и подсчитать частотную характеристику, т.е. сколько раз этот элемент встречается в качестве первого элемента.

Таким образом мы повторяем алгоритм для i-го элемента каждого значения, вычисляя для каждого i-го набора частотную характеристику.

1.3 Алгоритм эффективного поиска по словарю

Алгоритм на вход получает словарь и на его основе составляется частотный анализ. По полученным значениям словарь разбивается на сегменты так, что все элементы с одинаковым первым элементом оказываются в одном сегменте.

Сегменты упорядочиваются по значению частотной характеристики так, чтобы к элементы с наибольшей частотной характеристикой был самый быстрый доступ.

Далее каждый сегмент упорядочивается по значению. Это необходимо для реализации бинарного поиска, который обеспечит эффективный поиск в сегменте при сложности $O(\log n)$

Таким образом, сначала выбирается нужный сегмент, а затем в нем проводится бинарный поиск нужного элемента. Средняя трудоёмкость при длине алфавита M может быть рассчитана по формуле 1.2.

$$\sum_{i \in [1,M]} (f_{\text{выбор i-го сегмента}} + f_{\text{поиск в i-ом сегменте}}) \cdot p_i \tag{1.2}$$

Вывод

Были рассмотрены алгоритмы полного перебора и эффективного поиска по словарю и их трудоёмкость. Так же был рассмотрен частотный анализ, являющийся частью одного из рассмотренных алгоритмов.

2 Конструкторская часть

2.1 Структура ПО

В данном разделе будет рассмотрена структура ПО 2.1.

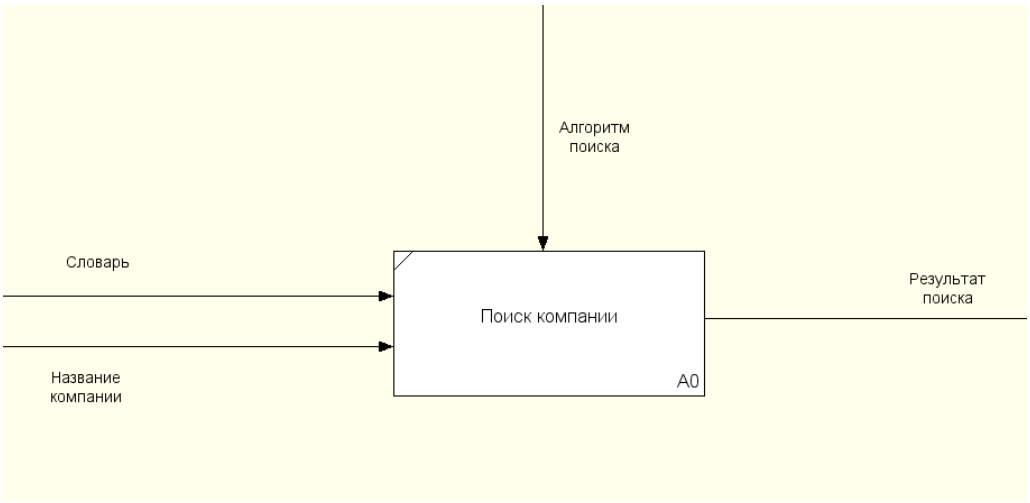

Рисунок 2.1 – Структура ПО

Структура словаря:

Словарь представлен типом данных `map[string]string`. Каждому ключу типа `string` соответствует значение такого же типа.

Словарь представлен следующими парами:

name : value – имя компании

email : value – электронный адрес компании

city : value – город компании

Требование ко вводу:

На вход подается словарь от 1000 вхождений.

Вывод

В данном разделе были рассмотрены структуры ПО и словаря, требования ко вводу.

3 Технологическая часть

В данном разделе приведены требования к программному обеспечению, средства реализации и листинги кода.

3.1 Требования к ПО

К программе предъявляется ряд требований:

- корректная сортировка.

3.2 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран многопоточный язык GO [3]. Данный выбор обусловлен моим желанием расширить свои знания в области применения данного язкыа. Так же данный язык предоставляет средства тестирования разработанного ПО. Так же была использована библиотека `gofakeit` [4] для генерации случайных записей.

3.3 Листинг кода

В листингах 3.1 – 3.3 приведены реализации алгоритмов.

Листинг 3.1 – Алгоритм полного перебора

```
    res = d[0]
    for _, v := range d {
        if v["name"] == name {
            res = v
            return
        }
    }
    return
}
```

Листинг 3.2 – Реализация частотного анализа

```
    alf := "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    st = make([]FA, len(alf))
    for i, v := range alf {
        a := FA{
            letter: string(v),
            count: 0,
            array: make([]Dictionary, 0),
        }
        st[i] = a
    }

    for _, v := range d {
        l := v["name"][:1]
        for i := range st {
            if st[i].letter == l {
                st[i].count++
            }
        }
    }

    sort.Slice(st, func(i, j int) bool {
        return st[i].count > st[j].count
    })

    for i := range st {
        for j := range d {
            if d[j]["name"][:1] == st[i].letter {
                st[i].array = append(st[i].array, d[j])
            }
        }
```

```
        sort.Slice(st[i].array, func(l, m int) bool {
            return st[i].array[l]["name"] < st[i].array[m]["name"]
        })
    }

    return
}
```

Листинг 3.3 – Реализация алгоритма эффективного поиска

```
    letter := n[:1]
    r = fa[0].array[0]
    for _, v := range fa {
        if v.letter == letter {
            r = Binary(v.array, n)
        }
    }

    return r
}
```

В таблице 3.1 приведены функциональные тесты программы.

Используемый в тестах словарь:

$$\text{Dict} = \begin{matrix}[\{\text{name} : \text{apple}, & \text{email} : \text{apple@apple.com}, & \text{city} : \text{coupertino }\} \\ \{\text{name} : \text{microsoft}, & \text{email} : \text{m@msn.com}, & \text{city} : \text{washington }\}]\end{matrix}$$

Таблица 3.1 – Функциональные тесты

Описание теста	**Входные данные**	**Ожидаемый результат**
успешный поиск	*Dict apple*	“*name*“ : “*apple*“ “*email*“ : “*apple@apple.com*“ “*city*“ : “*coupertino*“
неуспешный поиск	*Dict xiaomi*	Not found

Вывод

В данном разделе была рассмотрена структура ПО и листинги кода программы.

4 Исследовательская часть

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- Операционная система: Kali [5] Linux [6] 5.9.0-kali2-amd64.
- Память: 8 GB.
- Процессор: Intel® Core™ i5-8250U [7] CPU @ 1.60GHz

Тестирование проводилось на ноутбуке при включённом режиме производительности. Во время тестирования ноутбук был нагружен только системными процессами.

4.2 Время выполнения алгоритмов

Алгоритмы тестировались при помощи написания «бенчмарков» [8], предоставляемых встроенными в Go средствами. Для такого рода тестирования не нужно самостоятельно указывать количество повторов. В библиотеке для тестирования существует константа N, которая динамически варьируется в зависимости от того, был ли получен стабильный результат или нет.

В листинге 4.1 пример реализации бенчмарка.

Листинг 4.1 – Реализация бенчмарка

```
func BenchmarkBrute_A(b *testing.B) {
    w := pick(d, "A")
    for i := 0; i < b.N; i++ {
        Brute(d, w)
    }
}
```

На рисунках 4.1 приведён график сравнения производительности конвейеров.

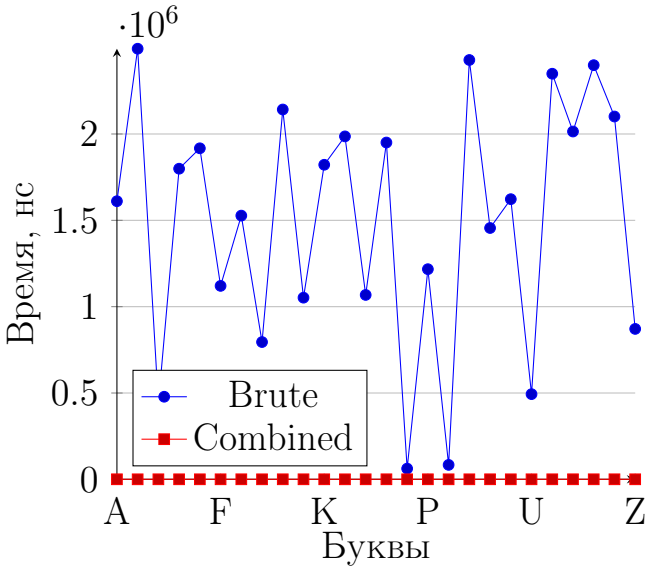

Рисунок 4.1 – Сравнение алгоритмов.

4.3 Производительность алгоритмов

Производительность и объем выделенной памяти при работе алгоритмов указаны на рисунке 4.2.

```
BenchmarkBrute_U-8              10180        493510 ns/op          0 B/op          0 allocs/op
BenchmarkBrute_V-8              1954        2349031 ns/op          0 B/op          0 allocs/op
BenchmarkBrute_W-8              8130        2014316 ns/op          0 B/op          0 allocs/op
BenchmarkBrute_X-8              12240       2398530 ns/op          0 B/op          0 allocs/op
BenchmarkBrute_Y-8              1290        2101275 ns/op          0 B/op          0 allocs/op
BenchmarkBrute_Z-8              3139         871221 ns/op          0 B/op          0 allocs/op
BenchmarkCombined_A-8           3066726         437 ns/op          0 B/op          0 allocs/op
BenchmarkCombined_B-8           5018848         306 ns/op          0 B/op          0 allocs/op
BenchmarkCombined_C-8           3178519         394 ns/op          0 B/op          0 allocs/op
BenchmarkCombined_D-8           3147026         383 ns/op          0 B/op          0 allocs/op
BenchmarkCombined_E-8           3646348         386 ns/op          0 B/op          0 allocs/op
BenchmarkCombined_F-8           2756048         376 ns/op          0 B/op          0 allocs/op
```

Рисунок 4.2 – Замеры производительности алгоритмов, выполненные при помощи команды `go test -bench . -benchmem`

Вывод

Экперимент показывает, что эффективность алгоритма полного перебора зависит от расположения искомого элемента.

Заключение

В результате выполнения данной работы были рассмотрены способы решения задачи поиска по словарю и реализованы алгоритмы эффективного поиска с использованием частотного анализа и бинарного поиска и поиска полным перебором.

Опыт показал, что рекомендуется использовать метод эффективного поиска. Было изучено применение частотного анализа в задаче эффективного поиска по словарю.

Сравнение времени алгоритмов показало, что эффективность алгоритма полного перебора зависит от места расположения искомого элемента.

Литература

[1] Silvio P. Brute-force algorithms. – Digital Humanities Advanced Research Centre (DHARC), Department of Classical Philology andItalian Studies, University of Bologna, Bologna, Italy, 2016.

[2] Boashash B. Time Frequency Analysis. – Elsevier Ltd., 2003.

[3] The Go Programming Language [Электронный ресурс]. Режим доступа: `https://golang.org/` (дата обращения: 09.09.2020).

[4] Fake library. [Электронный ресурс]. Режим доступа: `https://github.com/brianvoe/gofakeit` (дата обращения: 03.12.2020).

[5] Our Most Advanced Penetration Testing Distribution, Ever. [Электронный ресурс]. Режим доступа: `https://kali.org/` (дата обращения: 12.09.2020).

[6] Linux – Википедия [Электронный ресурс]. Режим доступа: `https://ru.wikipedia.org/wiki/Linux` (дата обращения: 12.09.2020).

[7] Intel Processors [Электронный ресурс]. Режим доступа: `https://www.intel.com/content/www/us/en/products/processors/core/i5-processors.html` (дата обращения: 12.09.2020).

[8] testing – The Go Programming Language [Электронный ресурс]. Режим доступа: `https://golang.org/pkg/testing/` (дата обращения: 12.09.2020).