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 Алгоритмы поиска в словаре

Студент Зайцева А.А.

Группа ИУ7-52Б

Преподаватели Волкова Л.Л.

Москва — 2021 г.

Оглавление

Введение

Словарь (англ. dictionary, map) — абстрактный тип данных, позволяющий хранить набор значений, обращение к которым происходит по ключам. Ключи должны допускать сравнение друг с другом. Примеры словарей достаточно разнообразны. Например, обычный толковый словарь хранит определения слов (являющиеся значениями), сопоставленные с самими словами (являющимися ключами), а банковская база данных может хранить данные клиентов, сопоставленные с номерами счетов.

Одной из основных операций в словаре является поиск значения по ключу. Словари могут содержать большое количество элементов, поэтому вопрос скорости поиска в них является важным [1].

Целью данной работы является разработка эффективного алгоритма поиска по словаре.

В рамках выполнения работы необходимо решить следующие задачи:

1) изучить три алгоритма поиска в словаре: алгоритм полного перебора, алгоритм бинарного поиска и алгоритм поиска в сегментированном словаре;

2) разработать и реализовать изученные алгоритмы;

3) провести сравнительный анализ трудоёмкости реализаций алгоритмов на основе теоретических расчетов (в среднем; в лучшем и худшем случаях);

4) провести сравнительный анализ процессорного времени выполнения реализаций алгоритмов на основе экспериментальных данных;

5) провести сравнительный анализ количества сравнений с заданным ключом, необходимых для поиска каждым алгоритмом значения по ключу и для определения отсутствия заданного ключа в словаре.

1 Аналитическая часть

В данном разделе будет приведена теория, необходимая для разработки и реализации трех алгоритмов поиска в словаре: алгоритма полного перебора, алгоритма бинарного поиска и алгоритма поиска в сегментированном слолваре.

1.1 Алгоритм полного перебора

Алгоритм полного перебора сводится к последовательному прохождению по всем ключам словаря и их сравнению с заданным ключом.

Этот алгоритм считается методом «грубой силы» [2]. Зато он может производиться в неотсортированном словаре, а добавление новых элементов в такой словарь не вызывает затруднений – их можно добавлять на любую позицию. Дополнительных затрат по памяти также не требуется.

1.2 Алгоритм бинарного поиска

Двоичный поиск (бинарный поиск) [3] — алгоритм поиска объекта (в нашем случае - значения) по заданному признаку (ключу) в множестве объектов, упорядоченных по тому же самому признаку (по ключу), принцип работы которого заключается в том, что на каждом шаге множество объектов делится на две части и в работе остаётся та часть множества, где находится искомый объект.

Данный алгоритм работает за логарифмическое время, дополнительных затрат по памяти не требует. Однако алгоритм бинарного поиска подразумевает, что массив ключей отсортирован. Это вызывает трудности при добавлении новых элементов в словарь, особенно если словарь хранится в постоянной памяти (например, на диске).

Алгоритм вставки элемента в упорядоченный массив заключается в последовательном анализе элементов массива, начиная с последнего и, при необходимости, сдвиге анализируемого элемента вправо на одну позицию, для того, чтобы освободить место для вставляемого элемента. Сдвиги про-

водятся до тех пор, пока не будет найдено место для вставляемого элемента, соответствующее его значению [4].

1.3 Алгоритм поиска в сегментированном словаре

Словарь можно разбить на сегменты так, чтобы все элементы с некоторым общим признаком попадали в один сегмент. Признаком для строк может быть первая буква, для чисел – остаток от деления. Элементы внутри одного сегмента можно также разбивать на подсегменты и так далее.

Затем сегменты упорядочиваются по значению частотной характеристики так, чтобы к элементам с наибольшей частотной характеристикой был самый быстрый доступ (в случае поиска полным перебором они должны быть ближе к началу). Такой характеристикой может послужить, например, размер сегмента.

Вероятность обращения к определенному сегменту равна сумме вероятностей обращений к его ключам, то есть $P_i = \sum_j p_j$, где P_i - вероятность обращения к i-ому сегменту, p_j - вероятность обращения к j-ому элементу, который принадлежит i-ому сегменту. Если обращения ко всем ключам равновероятны, то можно заменить сумму на произведение: $P_i = N \cdot p$, где N - количество элементов в i-ом сегменте, а p - вероятность обращения к произвольному ключу.

Таким образом, для поиска ключа в словаре необходимо найти сегмент, которму принадлежит этот ключ (например, полным перебором), а затем осуществить поиск ключа в пределах найденного сегмента. Чтобы поиск в пределах сегмента осуществлялся эффективнее, можно упорядочить ключи в каждом сегменте и реализовать бинарный поиск.

Сегментированный словарь позволяет уменьшить количество сравнений, необходимых для поиска в словаре. Однако он требует предобработки произвольного словаря для превращения его в сегментированный, а также дополнительной памяти под такую организацию. Объем дополнительной памяти определяется количеством выделяемых сегментов и подсегментов.

Также из-за необходимости поддержания структуры словаря усложня-

ется операция добавления в словарь новых элементов. Необходимо определить, к какому сегменту относится новый ключ, если сегмент разбит на подсегменты – соответсвующий подсегмент, а затем найти позицию внутри сегмента, в которую этот ключ необходимо вставить для сохранения упорядоченности. Также после добавления нового ключа сегмент становится больше, поэтому необходимо проверить упорядоченность сегментов по частотной характеристике и пересортировать их в случае необходимости.

1.4 Вывод из аналитической части

Были рассмотрены идеи и материалы, необходимые для разработки и реализации трех алгоритмов поиска в словаре: алгоритма полного перебора, алгоритма бинарного поиска и алгоритма поиска в сегментированном словаре.

2 Конструкторская часть

В данном разделе будут приведены структура используемого словаря и схемы рассмотренных в предыдущем разделе алгоритмов поиска в словаре.

2.1 Структура импользуемого словаря

Словарь состоит из пар вида <en - rus>, где en - слово на английском языке, rus - его перевод на русский.

2.2 Схема алгоритма полного перебора

На рисунке 2.1 приведена схема алгоритма полного перебора поиска в словаре.

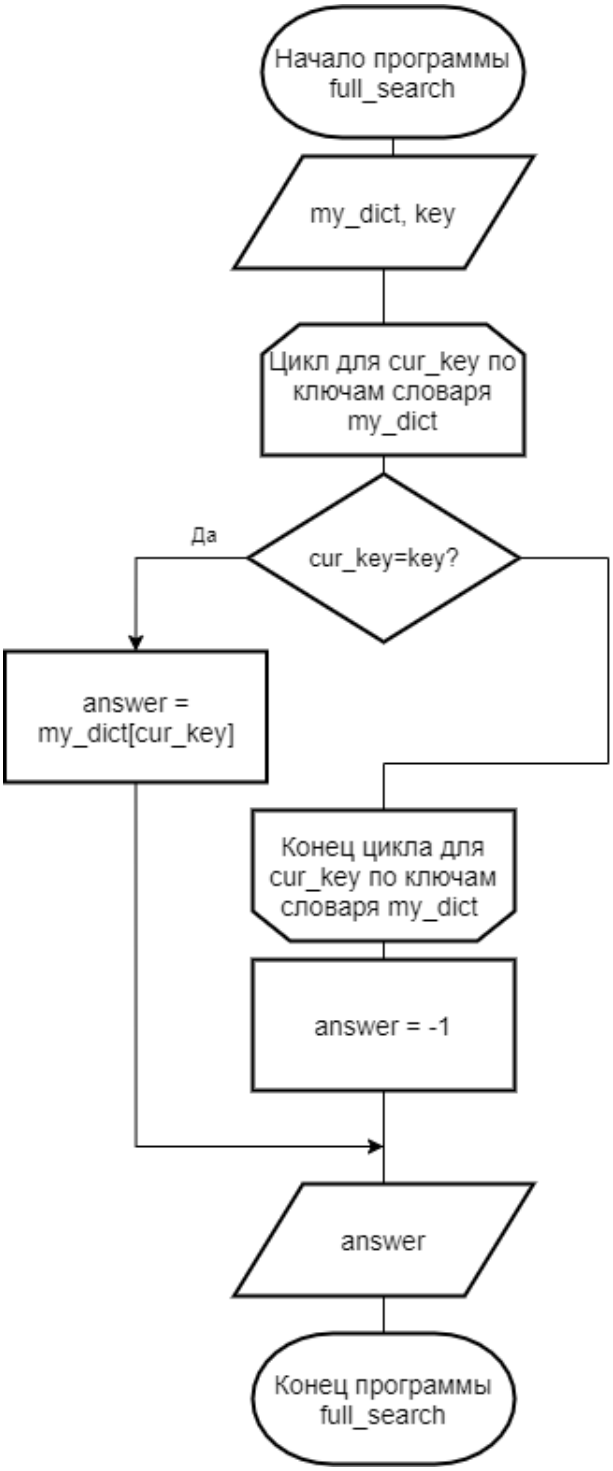

Рисунок 2.1 – Схема алгоритма полного перебора

2.3 Схема алгоритма бинарного поиска

На рисунке 2.2 приведена схема алгоритма бинарного поиска в словаре. Предполагается, что передаваемый словарь my_dict лексикографически упорядочен по возрастанию ключей.

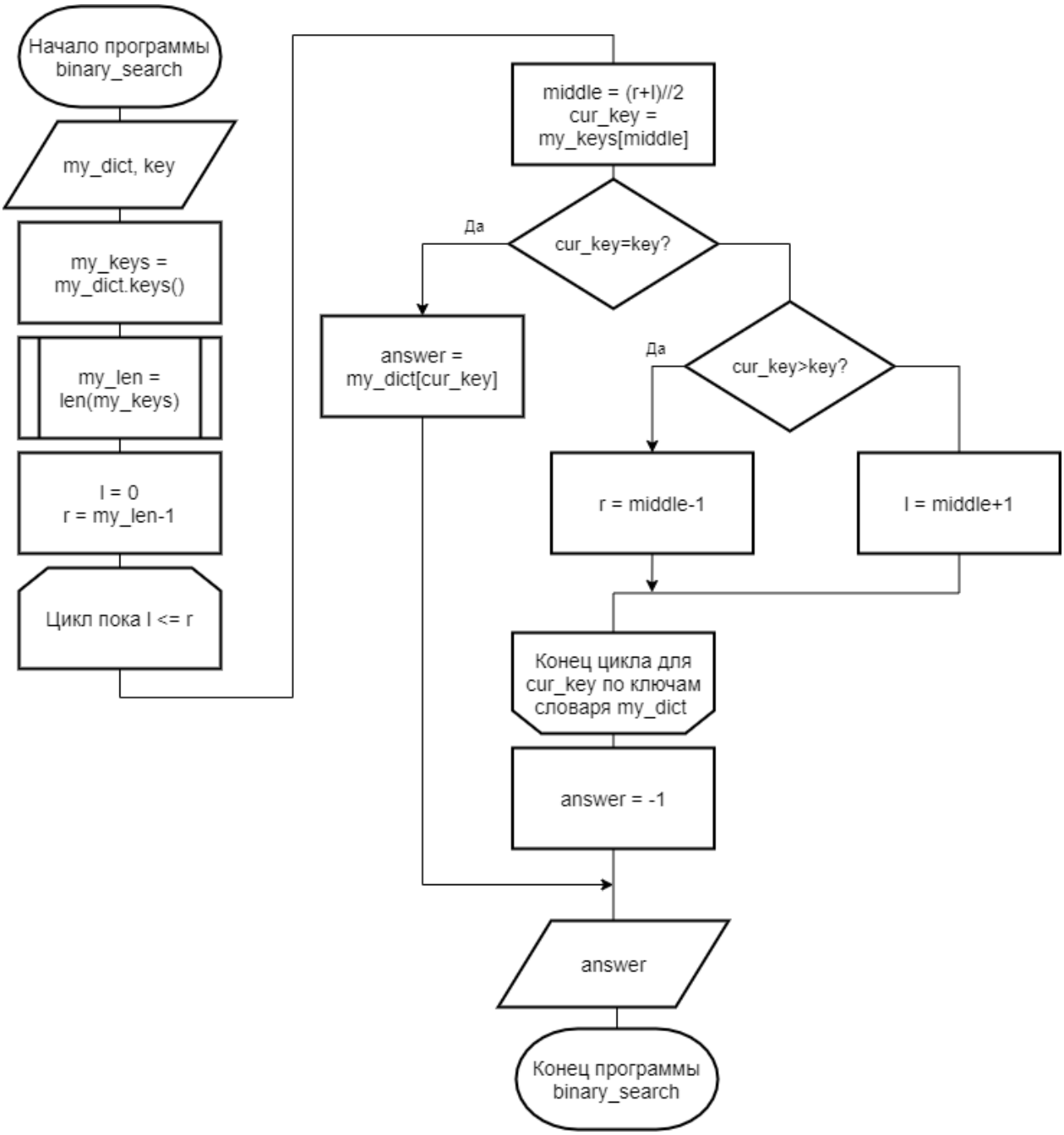

Рисунок 2.2 – Схема алгоритма бинарного поиска

2.4 Схема алгоритма поиска в сегментированном словаре

На рисунке 2.3 приведена схема создания словаря, сегментированного по первым буквам ключей. Такой подход к сегментации выбран, так как он является наиболее нативным для используемого словаря. Предполагается, что обращения ко всем ключам равновероятны, поэтому для ускорения поиска нужного сегмента полным перебором сегментированный словарь упорядочен по убыванию количества пар (ключ: значени) в нем. Пары внутри одого сегмента упорядочены лексикографически по возрастанию ключей для возможности реализации бинарного поиска в пределах сегмента.

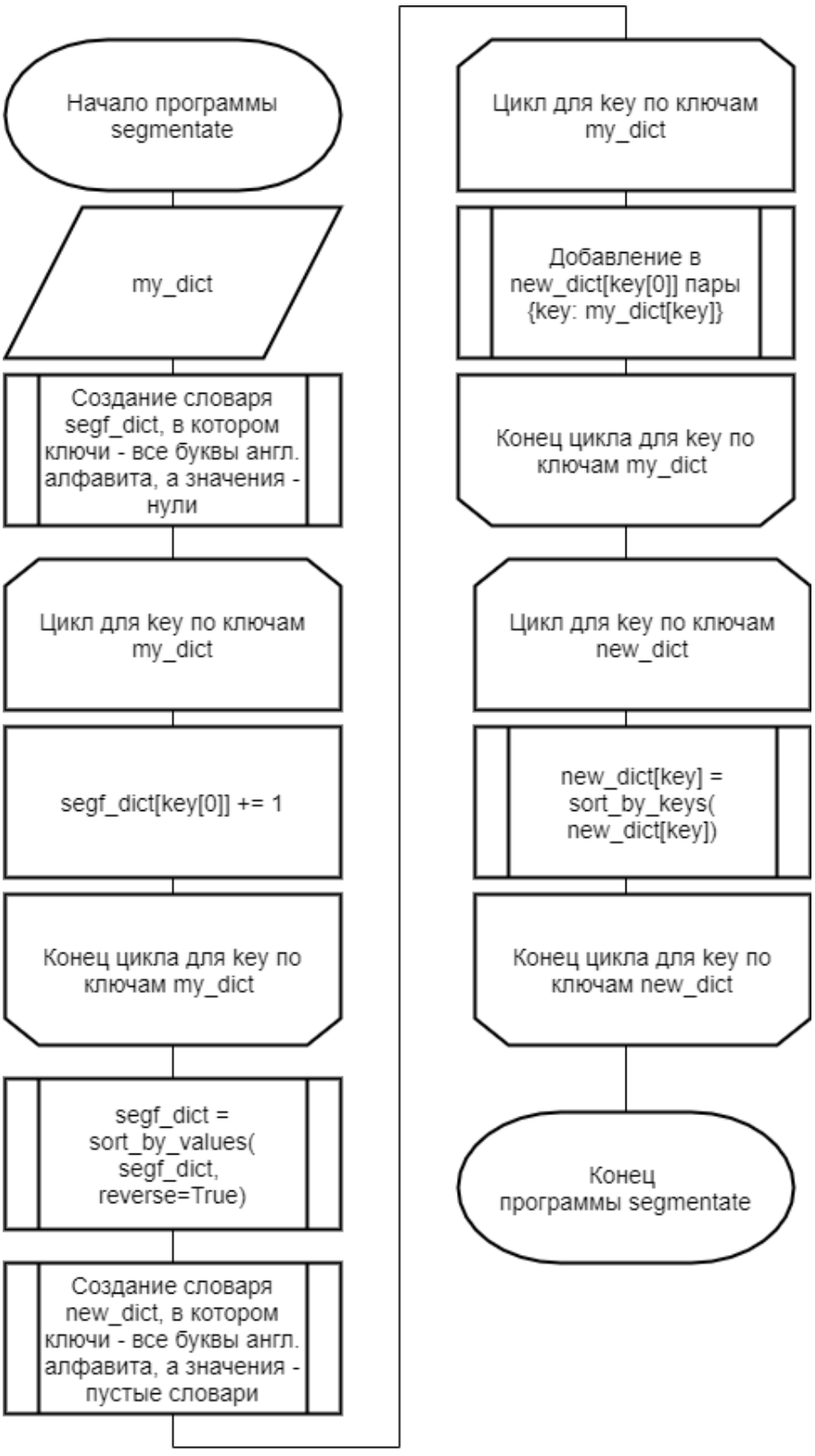

Рисунок 2.3 – Схема создания сегментированного словаря

На рисунке 2.4 приведена схема поиска в созданном сегментированном словаре. Сначала полным перебором осуществляется поиск нужного сегмента, а затем – бинарный поиск ключа в выбранном сегменте.

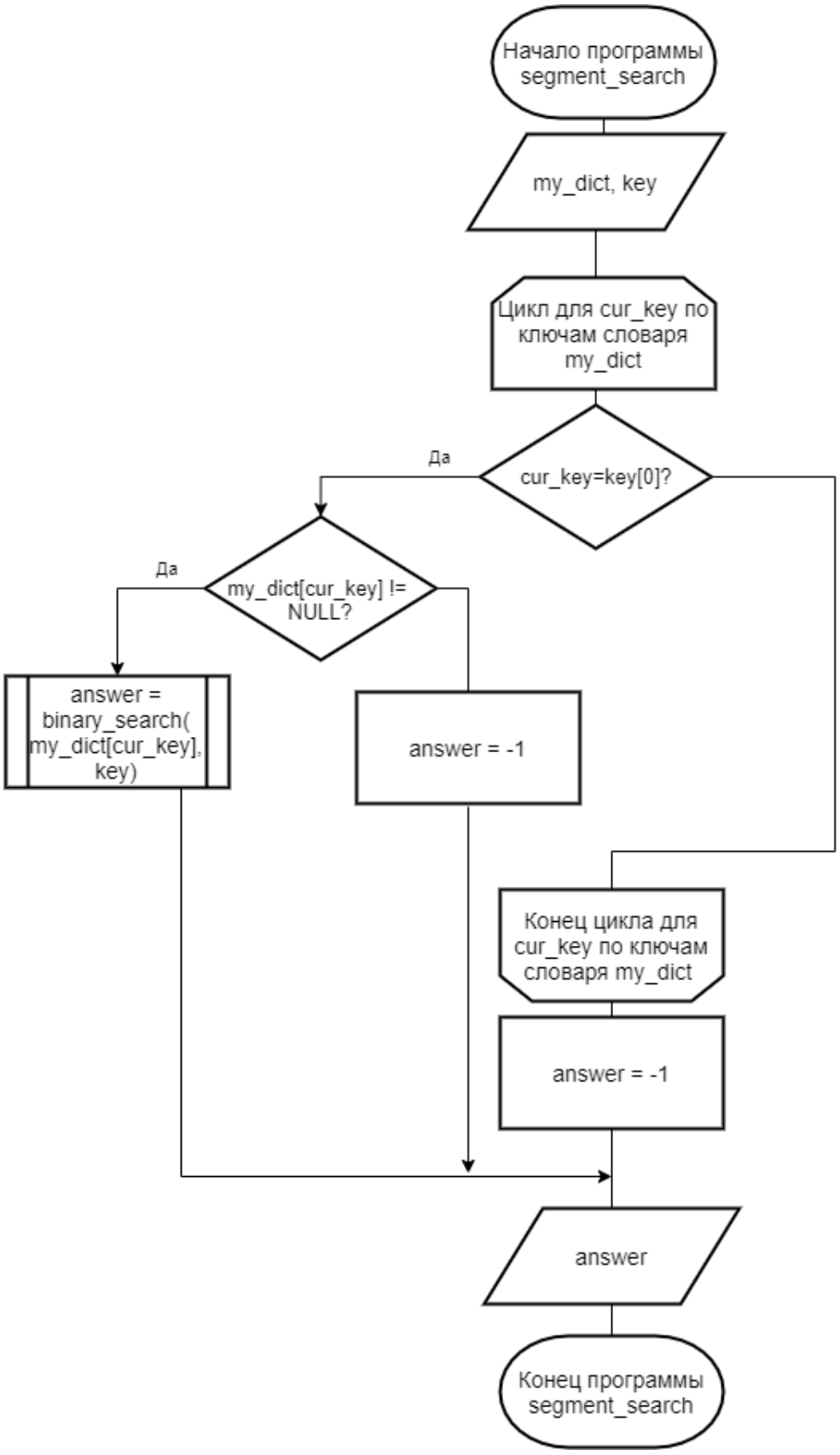

Рисунок 2.4 – Схема поиска в сегментированном словаре

2.5 Вывод из конструкторской части

Были приведены схемы разрабатываемых алгоритмов поиска в словаре: алгоритма полного перебора, алгоритма бинарного поиска и алгоритма поиска в сегментированном слолваре.

3 Технологическая часть

В данном разделе производится выбор средств реализации, приводятся требования к ПО, листинги реализованных алгоритмов поиска в словаре, а также результаты их тестированиия.

3.1 Требования к ПО

На вход программе подается ключ, на выходе ожидается значение, найденное в словаре по заданному ключу каждым реализованным алгоритмом, или значение -1, если такого ключа в словаре нет.

3.2 Выбор средств реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран язык Python [5]. Он позволяет быстро реализовывать различные алгоритмы без выделения большого времени на проектирование сруктуры программы и выбор типов данных.

Кроме того, в Python есть библиотека time, которая предоставляет функцию process_time для замера процессорного времени [6].

В качестве среды разработки выбран PyCharm. Он является кроссплатформенным, а также предоставляет удобный и функциональнаый отладчик и средства для рефакторинга кода, что позволяет быстро находить и исправлять ошибки [7].

3.3 Листинги кода

В листинге 3.1 представлена реализация алгоритма полного перебора поиска в словаре.

Листинг 3.1 – Алгоритм полного перебора

```
def full_search(my_dict, key):
  for cur_key in my_dict.keys():
    if cur_key == key:
      return my_dict[cur_key]
  return -1
```

В листинге 3.2 представлена реализация алгоритма бинарного поиска в словаре.

Листинг 3.2 – Алгоритм бинарного поиска в словаре

```
def binary_search(my_dict, key):
  my_keys = list(my_dict.keys())
  my_len = len(my_keys)
  l = 0
  r = my_len - 1

  while l <= r:
    middle = (r + l) // 2
    cur_key = my_keys[middle]

    if cur_key == key:
      return my_dict[cur_key]
    elif cur_key > key:
      r = middle - 1
    else:
      l = middle + 1

  return -1
```

В листинге 3.3 представлена реализация функции для создания сегментированного словаря.

Листинг 3.3 – Функция для создания сегментированного словаря

```
1
2  def segmentate(my_dict):
3    seg_frequency_dict = {letter: 0 for letter in "
       abcdefghijklmnopqrstuvwxyz"}
4    for key in my_dict.keys():
5      seg_frequency_dict[key[0]] += 1
6
7    seg_frequency_dict = sort_by_values(seg_frequency_dict, reverse=
       True)
8
9    new_dict = {letter: dict() for letter in seg_frequency_dict.keys
       ()}
10   for key in my_dict.keys():
11     new_dict[key[0]].update({key: my_dict[key]})
12   for key in new_dict:
13     new_dict[key] = sort_by_keys(new_dict[key])
14
15   return new_dict
```

В листинге 3.4 представлена реализация алгоритма поиска в сегментированном словаре.

Листинг 3.4 – Алгоритм поиска в сегментированном словаре

```
1  def segment_search(my_dict, key):
2    for k in my_dict:
3      if key[0] == k:
4        if my_dict[k]:
5          return binary_search(my_dict[k], key)
6        else:
7          return -1
8    return -1
```

3.4 Тестирование

В таблице 3.1 приведены функциональные тесты для алгоритмов поиска в словаре. Все тесты пройдены успешно этим каждым алгоритмом.

Ключ	Словарь	Ожидание	Результат
"few"	{}	-1	-1
"few"	{"few": "неск.", "little": "немн."}	"неск."	"неск."
"much"	{"few": "неск.", "little": "немн."}	-1	-1

Таблица 3.1 – Функциональные тесты.

Вывод

Был производен выбор средств реализации, приведены требования к ПО, реализованы и протестированы алгоритмы поиска в словаре.

4 Исследовательская часть

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- операционная система: Windows 10;
- оперативная память: 16 Гб;
- процессор: Intel® Core™ i5-8259U;
- количество ядер: 4;
- количество логических процессоров: 8.

Во время тестирования ноутбук был включен в сеть питания и нагружен только встроенными приложениями окружения и системой тестирования.

4.2 Пример работы программы

На рисунке 4.1 приведен пример работы программы.

```
Loaded 2 items
Словарь:
{'few': 'несколько', 'little': 'немного'}
Input key: few
Python search: несколько
Full search: несколько
Binary search: несколько
Segment search: несколько

Process finished with exit code 0
```

Рисунок 4.1 – Пример работы программы

4.3 Сравнение трудоемкостей реализаций

полный перебор Трудоёмкость алгоритма зависит от того, присутствует ли искомый ключ в словаре. Если такого ключа нет, то она определяется количеством ключей в словаре, если есть – его удаленностью от начала массива ключей.

Пусть алгоритм нашёл элемент на первом сравнении (лучший случай), тогда будет затрачено $k_0 + k_1$ операций, на втором - $k_0 + 2 \cdot k_1$, на последнем (худший случай) - $k_0 + N \cdot k_1$. Если ключа нет в массиве ключей, то мы сможем понять это, только перебрав все ключи, таким образом трудоёмкость такого случая равно трудоёмкости случая с ключом на последней позиции. Средняя трудоёмкость может быть рассчитана как математическое ожидание по формуле (4.1), где Ω – множество всех возможных случаев.

$$\begin{aligned}\sum_{i \in \Omega} p_i \cdot f_i =& (k_0 + k_1) \cdot \frac{1}{N+1} + (k_0 + 2 \cdot k_1) \cdot \frac{1}{N+1} + \\ &+ (k_0 + 3 \cdot k_1) \cdot \frac{1}{N+1} + (k_0 + N k_1) \frac{1}{N+1} + (k_0 + N \cdot k_1) \cdot \frac{1}{N+1} = \\ &= k_0 \frac{N+1}{N+1} + k_1 + \frac{1 + 2 + \cdots + N + N}{N+1} = \\ &= k_0 + k_1 \cdot \left(\frac{N}{N+1} + \frac{N}{2} \right) = k_0 + k_1 \cdot \left(1 + \frac{N}{2} - \frac{1}{N+1} \right)\end{aligned} \tag{4.1}$$

сегмент

Средняя трудоёмкость при множестве всех возможных случаев Ω может быть рассчитана по формуле (4.2).

$$\sum_{i \in \Omega} (f_{\text{выбор сегмента i-ого элемента}} + f_{\text{бинарный поиск i-ого элемента}}) \cdot p_i \qquad (4.2)$$

Возможны два случая определения отсутсвия ключа в словаре. Если в словаре нет ключей, начинающихся на ту же букву, то поиск значения по ключу закончится на этапе поиска нужного сегмента. Иначе понадобится осуществить еще и поиск внутри сегмента.

4.4 Сравнение времени выполнения реализаций алгоритмов

Сравнивалось процессорное время работы алгоритма полного перебора и муравьиного алгоритма с параметрами, выбранными в результате параметрзации ($\alpha = 0.1$, ро=0.75 и tmax=400). Эти реализации сравнивались по времени работы при количестве городов от 3 до 11 с шагом 1.

Так как некоторые задачи выполняются достаточно быстро, а замеры времени имеют некоторую погрешность, они для каждой реализации и каждого количества заявок выполнялись 10 раз, а затем вычислялось среднее время работы.

На рисунке 4.2 приведены результаты сравнения времени выполнения реализаций алгоритмов.

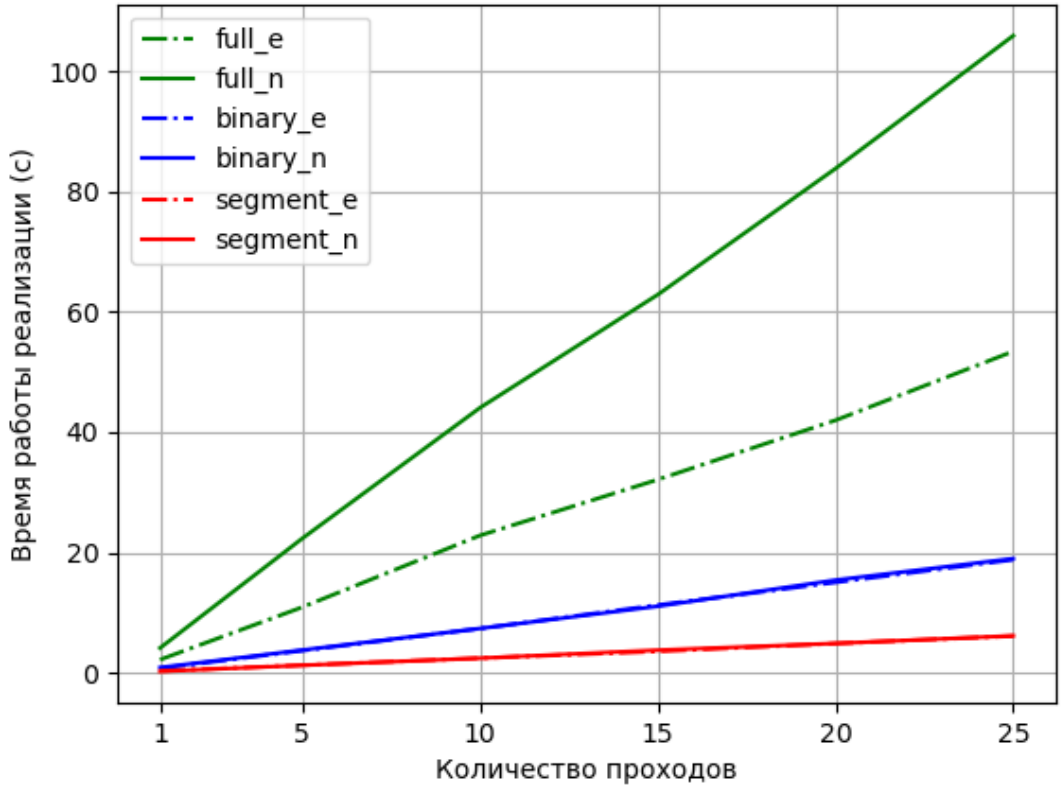

Рисунок 4.2 – Сравнение времени работы реализаций в зависимости от количества городов

Как видно из графиков, теоретическая оценка трудоемкости подтвердилась: время работы алгоритма полного перебора растет как O(n!), а муравьиного алгоритма - как O(n^3), где n - количество городов. В связи с этим при небольшом количестве городов (до 8 включительно) алгоритм полного перебора находит решение за меньшее количество времени по сравнению с муравьиным алгоритмом, однако далее с ростом числа городов начинает все стремительней уступать второму.

4.5 Вывод из исследовательской части

Таким образом, в ситуациях, когда количество городов велико (более 8, например), а задача коммивояжера не требует абсолютно точного ответа, для ее решения можно использовать муравьиный алгоритм вместо алгоритма полного перебора, что позволит сэкономить процессорное время. При этом заранее проведенная параметризация может помочь настро-

ить первый алгоритм так, что и он будет в большинстве случаев выдавать точный ответ.

При небольшом же размере графа (прмерно до размерности 8x8) муравьиный алгоритм с большой вероятностью даст точный ответ, однако в этом случае он не дает выигрыша по времени перед алгортмом полного перебора.

Заключение

В результате выполнения лабораторной работы была достигнута поставленная цель: был реализован муравьиный алгоритм для решения задачи коммивояжера и приобретены навыки параметризации алгоритмов.

В рамках выполнения работы были выаолнены следующие задачи:

1) реализован алгоритм полного перебора для решения задачи коммивояжера;

2) изучен и реализован муравьиный алгоритм для решения задачи коммивояжера;

3) проведена параметризаця муравьиного алгоритма на трех классах данных и подобраны оптимальные параметры;

4) проведен сравнительный анализ времени выполнения и трудоемкостей реализаций.

Литература

[1] Множество и словарь [Электронный ресурс]. Режим доступа: https://acm.khpnets.info/w/index.php?title=\T2A\CYRM\T2A\cyrn\T2A\cyro\T2A\cyrzh\T2A\cyre\T2A\cyrs\T2A\cyrt\T2A\cyrv\T2A\cyro_\T2A\cyri_\T2A\cyrs\T2A\cyrl\T2A\cyro\T2A\cyrv\T2A\cyra\T2A\cyrr\T2A\cyrsftsn._\T2A\CYRR\T2A\cyre\T2A\cyra\T2A\cyrl\T2A\cyri\T2A\cyrz\T2A\cyra\T2A\cyrc\T2A\cyri\T2A\cyrya_\T2A\cyrn\T2A\cyra_\T2A\cyrd\T2A\cyre\T2A\cyrr\T2A\cyre\T2A\cyrv\T2A\cyrsftsn\T2A\cyrya\T2A\cyrh_\T2A\cyrp\T2A\cyro\T2A\cyri\T2A\cyrs\T2A\cyrk\T2A\cyra (дата обращения: 28.11.2021).

[2] АЛГОРИТМИЧЕСКИЕ СТРАТЕГИИ [Электронный ресурс]. Режим доступа: https://portal.tpu.ru/f_ic/files/school/materials/ppt/2.pdf (дата обращения: 28.11.2021).

[3] Целочисленный двоичный поиск [Электронный ресурс]. Режим доступа: https://neerc.ifmo.ru/wiki/index.php?title=\T2A\CYRC\T2A\cyre\T2A\cyrl\T2A\cyro\T2A\cyrch\T2A\cyri\T2A\cyrs\T2A\cyrl\T2A\cyre\T2A\cyrn\T2A\cyrn\T2A\cyrery\T2A\cyrishrt_\T2A\cyrd\T2A\cyrv\T2A\cyro\T2A\cyri\T2A\cyrch\T2A\cyrn\T2A\cyrery\T2A\cyrishrt_\T2A\cyrp\T2A\cyro\T2A\cyri\T2A\cyrs\T2A\cyrk (дата обращения: 28.11.2021).

[4] Вставка элемента в отсортированный массив [Электронный ресурс]. Режим доступа: https://infopedia.su/17x10dd5.html (дата обращения: 28.11.2021).

[5] Лутц Марк. Изучаем Python, том 1, 5-е изд. Пер. с англ. — СПб.: ООО “Диалектика”, 2019. Т. 832.

[6] time — Time access and conversions [Электронный ресурс]. Режим доступа: https://docs.python.org/3/library/time.html (дата обращения: 05.09.2021).

[7] Python и Pycharm [Электронный ресурс]. Режим доступа: https://py-charm.blogspot.com/2017/09/pycharm.html (дата обращения: 05.09.2021).