**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МОЭВМ**

отчет

**по лабораторной работе № 2**

**по дисциплине «Построение и Анализ Алгоритмов»**

Тема: «**Коммивояжёр**»

|  |  |  |
| --- | --- | --- |
| Студент гр. 3343 |  | Коршков А.А. |
| Преподаватель |  | Жангиров Т. Р. |

Санкт-Петербург

2025

# Цель работы

Изучить различные алгоритмы для решения задачи коммивояжёра. Написать программу решения коммивояжёра через динамическое программирование.

# Задания

**№1 Динамическое программирование**

Напишите программу, решающую задачу коммивояжера. Нужно найти кратчайший маршрут, который проходит через все заданные города ровно один раз и возвращается в исходный город. Не все города могут быть напрямую связаны друг с другом.

**Входные данные:**

* n - количество городов (5 ≤ n ≤ 15).
* Матрица расстояний между городами размером n×n, где graph[i][j] обозначает расстояние от города i до города j. Если graph[i][j]=0  (и i ≠ j), это означает, что прямого пути между городами нет.

**Выходные данные:**

* Минимальная стоимость маршрута, проходящего через все города и возвращающегося в начальный город.
* Оптимальный путь в виде последовательности посещаемых городов, начинающейся и заканчивающейся в начальном городе.
* Если такого пути не существует, вывести "no path".

**Sample Input 1:**

5

0 1 13 23 7

12 0 15 18 28

21 29 0 33 28

23 19 34 0 38

5 40 7 39 0

**Sample Output 1:**

78

0 4 2 3 1 0

**Sample Input 2:**

3

0 1 0

1 0 1

0 1 0

**Sample Output 2:**

no path

**№2 2-приближённый алгоритм**

Разработайте программу, которая решает задачу коммивояжера при помощи 2-приближенного алгоритма. В данной постановке задачи нужно вернуться в исходную вершину после прохождения всех остальных вершин. При обходе остовного дерева (MST) необходимо идти по минимальному допустимому ребру из текущего. Каждая вершина в графе обозначается неотрицательным числом, начиная с 0, каждое ребро имеет неотрицательный вес. В графе нет рёбер из вершины в саму себя, в матрице весов на месте таких отсутствующих рёбер стоит значение -1.

Пример входных данных

2

-1 18.97 22.36 19.42 3.61

18.97 -1 35.61 38.01 17.0

22.36 35.61 -1 16.28 21.19

19.42 38.01 16.28 -1 21.02

3.61 17.0 21.19 21.02 -1

В первой строке указывается начальная вершина.

Далее идёт матрица весов.

В качестве выходных данных необходимо представить длину пути, полученного при помощи алгоритма. Следующей строкой необходимо представить путь, в котором перечислены вершины, по которым необходимо пройти от начальной вершины. Для приведённых в примере входных данных ответом будет

91.92

2 3 0 4 1 2

**№3 Коммивояжёр: ветви и границы**

В волшебной стране Алгоритмии великий маг, Гамильтон, задумал невероятное путешествие, чтобы связать все города страны заклятием процветания. Для этого ему необходимо посетить каждый город ровно один раз, создавая тропу благополучия, и вернуться обратно в столицу, используя минимум своих чародейских сил. Вашей задачей является помощь в прокладывании маршрута с помощью древнего и могущественного алгоритма ветвей и границ.

Карта дорог Алгоритмии перед Гамильтоном представляет собой полный граф, где каждый город соединён магическими порталами с каждым другим. Стоимость использования портала из города в город занимает определённое количество маны, и Гамильтон стремится минимизировать общее потребление магической энергии для закрепления проклятия.

**Входные данные:**

Первая строка содержит одно целое число N (N — количество городов). Города нумеруются последовательными числами от 0 до N−1.

Следующие N строк содержат по N чисел каждая, разделённых пробелами, формируя таким образом матрицу стоимостей M. Каждый элемент Mi,j​ этой матрицы представляет собой затраты маны на перемещение из города i в город j.

Выходные данные:

Первая строка: Список из N целых чисел, разделённых пробелами, обозначающих оптимальный порядок городов в магическом маршруте Гамильтона. В начале идёт город 0, с которого начинается маршрут, затем последующие города до тех пор, пока все они не будут посещены.

Вторая строка: Число, указывающее на суммарное количество израсходованной маны для завершения пути.

**Sample Input 1:**

3

-1 1 3

3 -1 1

1 2 -1

**Sample Output 1:**

0 1 2

3.0

**Sample Input 2:**

4

-1 3 4 1

1 -1 3 4

9 2 -1 4

8 9 2 -1

**Sample Output 2:**

0 3 2 1

6.0

**Задание варианта:**

**№2.** МВиГ: Алгоритм Литтла с модификацией: после приведения матрицы, к нижней оценке веса решения добавляется нижняя оценка суммарного веса остатка пути на основе МОД. Приближённый алгоритм: АБС.

Начинать АБС со стартовой вершины.

**Примечания для варианта:**

## Основные теоретические положения

**Описание алгоритмов:**

Алгоритм создает префиксное дерево из букв искомых подстрок. Вершины, в которых искомая подстрока заканчивается называет терминальной и выделяется специальным цветом в графическом представлении. Суффиксная ссылка вершины u – это вершина v, такая что строка v является максимальным суффиксом строки u. Для корня и вершин, исходящих из корня, суффиксной ссылкой является корень. Для остальных вершин осуществляется переход по суффиксной ссылке родителя и, если оттуда есть ребро с заданным символом, суффиксная ссылка назначается в вершину, куда это ребро ведет. Cуффиксные ссылки находятся не автоматически для каждой вершины, а вычисляются во время работы программы при обращении к специальному методу.

Текст, в котором нужно найти подстроки побуквенно передается в автомат. Начиная из корня, автомат переходит по ребру, соответствующему переданному символу. Если нужного ребра нет, переходит по ссылке. Если встреченная вершина является терминальной, значит была встречена подстрока. Если найдено совпадение нужно пройти по терминальным ссылкам, если они не None, чтобы вывести все шаблоны заканчивающиеся на этом месте. Номер подстроки хранится в поле pattern\_numbers вершины. В ответ сохранятся индекс, на котором началась эта подстрока в тексте и сам номер подстроки.

Алгоритм Ахо-Корасик иногда называют «расширенной версией КМП». Схожесть этих алгоритмов заключается в нахождении всех вхождений заданных шаблонов в тексте за линейное время относительно его длины. Однако если в КМП используется префикс-функция для вычисления максимального совпадения префикса и суффикса шаблона (что позволяет избежать полного перебора при несовпадении символов), то в алгоритме Ахо-Корасик применяется бор (дерево строк) и автомат с дополнительными ссылками, что делает его применимым для множественного поиска сразу нескольких шаблонов.

**Оценка сложности по памяти и операциям**

O(M · α + N + Z) – сложность алгоритма Ахо-Корасик по времени, где

M – суммарный размер всех шаблонов, α – размер алфавита, N – длина текста, Z – количество найденных вхождений. Эта сложность получается из сложности создания автомата с O(M · α) и сложности поиска в тексте O(N + Z).

O(M · α) – сложность алгоритма Ахо-Корасик по памяти, где M – суммарная длина всех шаблонов, α - размер алфавита.

Для алгоритма Ахо-Корасик с джокером сложность по времени O(M · α + N + Z· K), где K – количество сегментов в паттерне (без джокеров), M – суммарная длина всех сегментов паттерна (без джокеров). Сложность по памяти не изменяется.

## Выполнение работы

**Описание работы**

Для решения заданий были написаны два класса Vertex и Trie, которые представляют вершину автомата и сам бор.

В классе Vertex описаны следующие методы:

\_\_init\_\_(self, id\_: int, alpha: int, parent: "Vertex" or None = None, pchar: str or None = None) -> None – констурктор класса вершины Vertex. В качестве аргументов принимает номер вершины, размер алфавита, родительскую вершину (если есть), символ родительской вершины (если есть).

is\_terminal(self) -> bool – возвращает True, если вершина является терминальной

@is\_terminal.setter

is\_terminal(self, value: bool) -> None – позволяет установить флаг для терминальной вершины

id(self) -> int – возвращает присвоенный идентификатор вершины

sufflink(self) -> "Vertex" or None – возвращает суффиксную ссылку на вершину, если суффиксная ссылка была вычислена для данной вершины.

@sufflink.setter

sufflink(self, value) -> None - позволяет установить значение для суффиксной ссылки

parent(self) -> "Vertex" or None – возвращает родительскую вершину, если это не корень бора.

pchar(self) -> str or None – возвращает символ родительской вершины, если это не корень бора.

\_\_str\_\_(self) -> str – возвращает информацию в строковом виде для вершины

В классе Trie описаны следующие методы:

\_\_init\_\_(self, alpha: int = 5) -> None – конструктор для класса автомата Ахо-Корасик. На вход принимает размер алфавита (по умолчанию 5, для заданного алфавита {A: 0, C: 1, G: 2, T: 3, N: 4})

size(self) -> int – возвращает кол-во вершин в дереве

last(self) -> Vertex – ввозвращает последнюю вершину в дереве

alpha(self) -> int – возвращает размер алфавита

vertices(self) -> list[Vertex] – возвращает список вершин

root(self) -> возвращает корень дерева

add(self, s: str, pattern\_num: int) -> None – добавляет образец в дерево

search(self, s: str) -> list[tuple[int, int]] - проверяет, есть ли строка в дереве и возвращает

get\_link(self, v: Vertex) -> Vertex – находит и возвращает суффиксную ссылку для вершины.

go(self, v: Vertex, char: str) -> Vertex - Возвращает вершину, в которую ведет переход по символу char из вершины v.

precompute\_sufflinks(self) -> None – предварительно вычисляет все суффиксные ссылки (нужно для визуализации автомата)

visualize(self, file\_name: str = "aho\_corasick") -> None - Создает графическое представление автомата Ахо-Корасик и сохраняет его в png файл. Также создаётся легенда для графа. На вход принимает имя файла, в который нужно сохранить визуализацию.

print\_bor\_structure(self) -> None - печатает структуру бора, а именно для каждой вершины пишет наименование, родителя, если вершина терминальная – указывает, каким шаблонам соответствует и возможные пути вниз.

print\_automaton\_structure(self) -> None - печатает структуру автомата, для каждой вершины отмечает суффиксную ссылку и возможные пути вниз.

Также для класса Trie есть вспомогательные внутренние методы:

\_num(c: str) -> int – возвращает номер буквы в заданном алфавите из 5 букв: {A: 0, C: 1, G: 2, T: 3, N: 4}

\_char(idx: int) -> int – возвращает символ буквы, соответствующий номеру в алфавите.

В файла main.py есть несколько функций для решения заданий:

main() -> None – главная функция, которая запускает функции для решения заданий

aho\_corasick\_search() -> None – функция, которая ищет позиции вхождения всех заданных образцов в тексте.

search\_with\_wildcard() -> None – функция, которая ищет индексы вхождения образца с джокером.

visualize\_and\_print(trie: Trie, filename: str) -> None - выводит информацию о вершинах и графическое представление автомата.

Для понимания графического представления есть наглядная легенда графа.

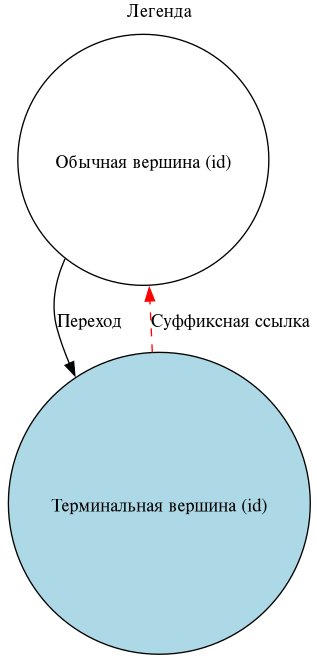


Рисунок 1 – Легенда графа

# Тестирование

Таблица 1 – Тестирование алгоритмов

|  |  |  |  |
| --- | --- | --- | --- |
| № | Входные данные | Выходные данные | Комментарии |
| 1 | NTAG  3  TAGT  TAG  T | 2 2  2 3 | Алгоритм Ахо-Корасик. Терминальные вершины отмечены корректно, суффиксные ссылки также корректны. Идентификаторы вершин отмечаются корректно. |
| 2 | ACGT  3  ACGT  CG  GT | 1 1  2 2  3 3 | Алгоритм Ахо-Корасик. Два последних образца содержатся в одном длинном образце. |
| 3 | ACGTNGGTCCG  5  ACG  CGT  TNG  CC  A | 1 1  1 5  2 2  4 3  9 4 | Алгоритм Ахо-Корасик. Алгоритм хорошо справляется с построением автомата для большого количества образцов. |
| 4 | ACTANCA  A$$A$  $ | 1 | Ахо-Корасик с джокером. Символ в маске только A. |
| 5 | ACGTNNTGCA  TNNXG  X | 4 | Ахо-Корасик с джокером. Корректно распознаёт маску в виде другой буквы. Дерево содержит в себе части TNN и G из маски. |
| 6 | ACGTNNTGCA  T&  & | 4  7 | Ахо-Корасик с джокером. Находит несколько вхождений в строке. |

# Выводы

Изучен принцип работы алгоритма Ахо-Корасик. Написаны программы, корректно решающие задачу поиска набора подстрок в строке, в также программа поиска подстроки с джокером. Также была написана визуализация для автомата Ахо-Корасик в графическом представлении.

ПРИЛОЖЕНИЕ А  
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py