Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт перспективной инженерии
Департамент цифровых робототехнических систем и электроники

ОТЧЕТ
ПО ЛАБОРАТОРНОЙ РАБОТЕ №5
дисциплины «Искусственный интеллект в профессиональной сфере»

Выполнила:
Мурашко Анастасия Юрьевна
3 курс, группа ИВТ-б-о-22-1,
09.03.02 «Информатика и вычислительная техника», очная форма обучения

_____
(подпись)

Проверил:
Богданов С.С., ассистент департамента цифровых робототехнических систем и электроника института перспективно инженерии

_____
(подпись)

Отчет защищен с оценкой_____Дата защиты_____

Ставрополь, 2024 г.

Тема: Исследование поиска с итеративным углублением.

Цель работы: приобретение навыков по работе с поиском с итеративным углублением с помощью языка программирования Python.

Ход работы:

Изучила теоретический материал работы, создала общедоступный репозиторий на GitHub, в котором использована лицензий MIT и язык программирования Python, также добавил файл .gitignore с необходимыми правилами.

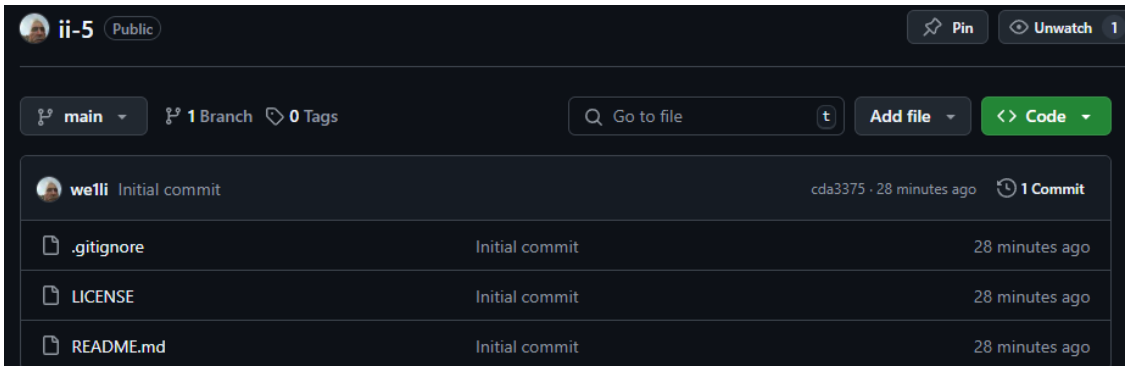

Рисунок 1. Новый репозиторий

Задача 1. "Поиск элемента в дереве с использованием итеративного углубления"

Необходимо представить систему управления доступом, где каждый пользователь представлен узлом в дереве. Каждый узел содержит уникальный идентификатор пользователя. Поставлена задача — разработать метод поиска, который позволит проверить существование пользователя с заданным идентификатором в системе, используя структуру дерева и алгоритм итеративного углубления.

Код программы:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

class BinaryTreeNode:
    """
    Класс для представления узла бинарного дерева.
    """
    def __init__(self, value, left=None, right=None):
```

```
        """
        Инициализация узла дерева.
        :param value: Значение узла.
        :param left: Левый дочерний узел.
        :param right: Правый дочерний узел.
        """
        self.value = value
        self.left = left
        self.right = right

    def add_children(self, left, right):
        """
        Добавляет левого и правого дочерних узлов.
        :param left: Левый дочерний узел.
        :param right: Правый дочерний узел.
        """
        self.left = left
        self.right = right

    def __repr__(self):
        """
        Возвращает строковое представление узла.
        """
        return f"<{self.value}>"


def iterative_deepening_search(root, goal):
    """
    Реализация алгоритма итеративного углубления для поиска элемента в
дереве.
    :param root: Корень дерева.
    :param goal: Целевое значение, которое нужно найти.
    :return: True, если целевое значение найдено, иначе False.
    """
    depth = 0
    while True:  # Постепенно увеличиваем глубину поиска
        print(f"Проверка на глубине: {depth}")
        found = depth_limited_search(root, goal, depth)
        if found:  # Если цель найдена, возвращаем результат
            return True
        depth += 1  # Увеличиваем глубину поиска


def depth_limited_search(node, goal, limit):
    """
    Поиск элемента с ограничением по глубине.
    :param node: Текущий узел.
    :param goal: Целевое значение.
    :param limit: Ограничение по глубине поиска.
    :return: True, если целевое значение найдено, иначе False.
    """
```

```
    if node is None:
        return False
    if node.value == goal:  # Если найдено целевое значение
        print(f"Найдено: {node.value}")
        return True
    if limit <= 0:  # Если достигли ограничения глубины
        return False

    # Рекурсивно ищем в левом и правом поддеревьях
    return (depth_limited_search(node.left, goal, limit - 1) or
            depth_limited_search(node.right, goal, limit - 1))


if __name__ == '__main__':
    # Создание дерева и установка значений
    root = BinaryTreeNode(1)
    left_child = BinaryTreeNode(2)
    right_child = BinaryTreeNode(3)
    root.add_children(left_child, right_child)
    right_child.add_children(BinaryTreeNode(4), BinaryTreeNode(5))

    # Целевое значение
    goal = 4

    # Вызов функции итеративного углубления
    if iterative_deepening_search(root, goal):
        print("Целевое значение найдено.")
    else:
        print("Целевое значение не найдено.")
```

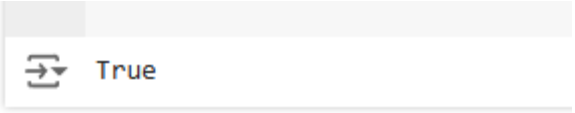

Рисунок 2. Результат работы программы

Задача 2. "Поиск в файловой системе"

Рассмотрим задачу поиска информации в иерархических структурах данных, например, в файловой системе, где каждый каталог может содержать подкаталоги и файлы. Алгоритм итеративного углубления идеально подходит для таких задач, поскольку он позволяет исследовать структуру данных постепенно, углубляясь на один уровень за раз и возвращаясь, если целевой узел не найден. Для этого необходимо:

- Построить дерево, где каждый узел представляет каталог в файловой системе, а цель поиска — определенный файл.

- Найти путь от корневого каталога до каталога (или файла), содержащего искомый файл, используя алгоритм итеративного углубления.

Код программы:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

class BinaryTreeNode:
    """
    Класс для представления узла бинарного дерева.
    """
    def __init__(self, value, left=None, right=None):
        """
        Инициализация узла дерева.
        :param value: Значение узла.
        :param left: Левый дочерний узел.
        :param right: Правый дочерний узел.
        """
        self.value = value
        self.left = left
        self.right = right

    def add_children(self, left, right):
        """
        Добавляет левого и правого дочерних узлов.
        :param left: Левый дочерний узел.
        :param right: Правый дочерний узел.
        """
        self.left = left
        self.right = right

    def __repr__(self):
        """
        Возвращает строковое представление узла.
        """
        return f"<{self.value}>"


def iterative_deepening_search(root, goal):
    """
    Реализация алгоритма итеративного углубления для поиска элемента в
дереве.
    :param root: Корень дерева.
    :param goal: Целевое значение, которое нужно найти.
    :return: True, если целевое значение найдено, иначе False.
    """
    depth = 0
    while True:  # Постепенно увеличиваем глубину поиска
        print(f"Проверка на глубине: {depth}")
```

```
        found = depth_limited_search(root, goal, depth)
        if found:  # Если цель найдена, возвращаем результат
            return True
        depth += 1  # Увеличиваем глубину поиска


def depth_limited_search(node, goal, limit):
    """
    Поиск элемента с ограничением по глубине.
    :param node: Текущий узел.
    :param goal: Целевое значение.
    :param limit: Ограничение по глубине поиска.
    :return: True, если целевое значение найдено, иначе False.
    """
    if node is None:
        return False
    if node.value == goal:  # Если найдено целевое значение
        print(f"Найдено: {node.value}")
        return True
    if limit <= 0:  # Если достигли ограничения глубины
        return False

    # Рекурсивно ищем в левом и правом поддеревьях
    return (depth_limited_search(node.left, goal, limit - 1) or
            depth_limited_search(node.right, goal, limit - 1))


if __name__ == '__main__':
    # Создание дерева и установка значений
    root = BinaryTreeNode(1)
    left_child = BinaryTreeNode(2)
    right_child = BinaryTreeNode(3)
    root.add_children(left_child, right_child)
    right_child.add_children(BinaryTreeNode(4), BinaryTreeNode(5))

    # Целевое значение
    goal = 4

    # Вызов функции итеративного углубления
    if iterative_deepening_search(root, goal):
        print("Целевое значение найдено.")
    else:
        print("Целевое значение не найдено.")
```

```
Проверка на глубине: 0
Проверка на глубине: 1
Проверка на глубине: 2
Найдено: 4
Целевое значение найдено.
```

Рисунок 3. Результат работы программы

Индивидуальное задание.

Условие: Поиск файлов с дублирующимся содержимым. Найдите два разных файла в файловом дереве, которые имеют одинаковое содержимое (по побайтовому сравнению), используя итеративное углубление. Дерево более ста файлов, и их глубина может достигать 10 уровней.

Чтобы проверить эту программу, выполним следующие шаги:

Создадим каталог и файлы. Папка 1 будет содержать два файла - file1.txt, duplicate1.txt (дубликат файла из второй папки). Папка 2 - file2.txt, duplicate2.txt.

Напишем код для создания данных файлов:

```
import os

# Создаем тестовую файловую структуру
os.makedirs("/content/test/dir1", exist_ok=True)
os.makedirs("/content/test/dir2", exist_ok=True)
os.makedirs("/content/test/dir3/subdir1", exist_ok=True)

# Создаем тестовые файлы
with open("/content/test/dir1/file1.txt", "w") as f:
    f.write("Hello, this is file 1.")
with open("/content/test/dir1/duplicate1.txt", "w") as f:
    f.write("This is a duplicate file.")
with open("/content/test/dir2/file2.txt", "w") as f:
    f.write("Hello, this is file 2.")
with open("/content/test/dir2/duplicate2.txt", "w") as f:
    f.write("This is a duplicate file.")
with open("/content/test/dir3/subdir1/duplicate3.txt", "w") as f:
    f.write("This is a duplicate file.")
```

Итеративное углубление:

Функция iterative_deepening_file_search запускает поиск, постепенно увеличивая глубину, пока не будет найдено совпадение или не будет достигнуто максимальное ограничение глубины.

Используется `hashlib.sha256` для создания хэша, что гарантирует уникальность на уровне содержимого. hashlib.sha256 — это метод из стандартной библиотеки Python для создания криптографически стойкого хэша. На основе содержимого файла создается уникальный 256-битный хэш.

Если содержимое двух файлов идентично, их хэши будут совпадать. Вместо сравнения файлов по содержимому (что может быть долго для больших файлов), сравниваются их хэши. Это значительно быстрее.

Функция depth_limited_search выполняет обход узлов с учетом ограничения глубины. Она проверяет хэш файлов и добавляет дубликаты в список.

Рекурсивная обработка каталогов: если узел не является файлом (то есть это каталог), то для каждого дочернего узла (child) запускается рекурсивная функция depth_limited_search. Глубина поиска уменьшается на 1 (depth - 1), чтобы учитывать ограничение глубины(10). Найденные дубликаты из дочерних узлов добавляются в список duplicates с помощью extend

Код программы:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import os
import hashlib


class TreeNode:
    """
    Класс для представления узла дерева файловой системы.
    """
    def __init__(self, path):
        self.path = path
        self.children = []

    def add_child(self, child):
        """
        Добавление дочернего узла.
        """
        self.children.append(child)

    def __repr__(self):
        return f"<{self.path}>"


def get_file_hash(file_path):
    """
    Вычисляет хэш содержимого файла для сравнения.
    :param file_path: Путь к файлу.
    :return: Хэш файла в виде строки.
    """
```

```
    hasher = hashlib.sha256()
    try:
        with open(file_path, 'rb') as f:
            while chunk := f.read(8192):  # Чтение файла по 8 КВ
                hasher.update(chunk)
        return hasher.hexdigest()
    except (FileNotFoundError, IsADirectoryError, PermissionError):
        return None


def iterative_deepening_file_search(root, depth_limit):
    """
    Поиск дублирующихся файлов в файловом дереве с использованием
итеративного углубления.
    :param root: Корневой узел дерева.
    :param depth_limit: Максимальная глубина поиска.
    :return: Список пар путей дублирующихся файлов.
    """
    for depth in range(depth_limit + 1):
        print(f"Проверка на глубине: {depth}")
        duplicates = depth_limited_search(root, {}, depth)
        if duplicates:
            return duplicates
    return []


def depth_limited_search(node, file_hashes, depth):
    """
    Ограниченный по глубине поиск дублирующихся файлов.
    :param node: Текущий узел дерева.
    :param file_hashes: Словарь для хранения хэшей файлов и их путей.
    :param depth: Оставшаяся глубина поиска.
    :return: Список пар путей дублирующихся файлов.
    """
    if depth < 0:
        return []

    duplicates = [] #Создание списка для хранения дубликатов
    if os.path.isfile(node.path):  # Проверяем, является ли узел файлом
        file_hash = get_file_hash(node.path) #вычисляет хэш
        if file_hash:
            if file_hash in file_hashes: #есть ли уже этот хэш в словаре
                duplicates.append((file_hashes[file_hash], node.path))
            else:
                file_hashes[file_hash] = node.path
    else:  # Если узел - каталог, рекурсивно обходим дочерние узлы
        for child in node.children:
            duplicates.extend(depth_limited_search(child, file_hashes,
depth - 1))

    return duplicates
```

```
def build_file_tree(root_path):
    """
    Построение дерева файловой системы.
    :param root_path: Корневой путь.
    :return: Корневой узел дерева файловой системы.
    """
    root = TreeNode(root_path)
    try:
        for entry in os.scandir(root_path):
            if entry.is_dir(follow_symlinks=False):
                root.add_child(build_file_tree(entry.path))
            elif entry.is_file(follow_symlinks=False):
                root.add_child(TreeNode(entry.path))
    except PermissionError:
        pass  # Игнорируем каталоги без прав доступа
    return root


if __name__ == '__main__':
    # Задаем корневой каталог и глубину поиска
    root_directory = "/content/test"
    max_depth = 10

    # Строим дерево файловой системы
    root_node = build_file_tree(root_directory)

    # Запускаем поиск дубликатов
    duplicates = iterative_deepening_file_search(root_node, max_depth)

    # Вывод результатов
    if duplicates:
        print("Найдены дублирующиеся файлы:")
        for file1, file2 in duplicates:
            print(f"{file1} <-> {file2}")
    else:
        print("Дублирующихся файлов не найдено.")
```

```
Проверка на глубине: 0
Проверка на глубине: 1
Проверка на глубине: 2
Найдены дублирующиеся файлы:
/content/test/dir1/duplicate1.txt <-> /content/test/dir2/duplicate2.txt
```

Рисунок 5. Результат работы программы

Контрольные вопросы:

1. **Что означает параметр n в контексте поиска с ограниченной глубиной, и как он влияет на поиск?**

Параметр **n** — это максимальная глубина, до которой будет проводиться поиск. Чем больше **n**, тем глубже может быть исследовано дерево. Если **n** слишком мал, алгоритм может не найти решение, если оно находится на более глубоком уровне.

2. **Почему невозможно заранее установить оптимальное значение для глубины d в большинстве случаев поиска?**

Определение оптимальной глубины **d** заранее невозможно, так как в большинстве задач структура пространства поиска или глубина решения не известна, и решение может находиться на разных уровнях глубины.

3. **Какие преимущества дает использование алгоритма итеративного углубления по сравнению с поиском в ширину?**

Алгоритм итеративного углубления использует меньше памяти, чем поиск в ширину, так как на каждом уровне хранится только информация о текущем пути. Также он может быстрее найти решение, если оно находится на меньшей глубине.

4. **Опишите, как работает итеративное углубление и как оно помогает избежать проблем с памятью.**

Итеративное углубление выполняет поиск в глубину с постепенным увеличением максимальной глубины. Каждый поиск выполняется с ограничением по глубине, что позволяет избежать хранения всех путей, как в поиске в ширину, и тем самым экономит память.

5. **Почему алгоритм итеративного углубления нельзя просто продолжить с текущей глубины, а приходится начинать поиск заново с корневого узла?**

Алгоритм начинается заново с корня, потому что на каждом шаге глубина поиска ограничена, и необходимо пересматривать все возможные пути с новой глубиной, иначе не будет гарантировано, что все уровни исследуются корректно.

6. **Какие временные и пространственные сложности имеет поиск с итеративным углублением?**

Временная сложность — **O(b^d)**, где **b** — коэффициент разветвления, а **d** — глубина решения. Пространственная сложность — **O(b * d)**, так как память требуется для хранения узлов текущего уровня и пути.

7. **Как алгоритм итеративного углубления сочетает в себе преимущества поиска в глубину и поиска в ширину?**

Алгоритм сочетает эффективность поиска в глубину (низкие требования к памяти) и поиск в ширину (постепенное исследование всех уровней). Это позволяет находить решения быстрее при меньших затратах памяти.

8. **Почему поиск с итеративным углублением остается эффективным, несмотря на повторное генерирование дерева на каждом шаге увеличения глубины?**

Повторное генерирование дерева на каждом шаге не приводит к значительным потерям эффективности, так как на каждом уровне проверяются только новые узлы. Алгоритм быстро находит решение на более мелких уровнях, если оно существует.

9. **Как коэффициент разветвления b и глубина d влияют на общее количество узлов, генерируемых алгоритмом итеративного углубления?**

Количество генерируемых узлов растет экспоненциально с увеличением **b** и **d**. Для каждой глубины генерируется **b^n** узлов, и каждый уровень добавляется к общему числу узлов.

10. **В каких ситуациях использование поиска с итеративным углублением может быть не оптимальным, несмотря на его преимущества?**

Когда решение находится на очень глубоком уровне, и повторное генерирование узлов на каждом шаге значительно увеличивает затраты времени. В таких случаях может быть эффективнее использовать другие алгоритмы, например, поиск с ограничением по глубине.

11. **Какую задачу решает функция iterative_deepening_search?**

Функция iterative_deepening_search решает задачу поиска целевого значения в дереве или графе с использованием алгоритма итеративного углубления.

12. **Каков основной принцип работы поиска с итеративным углублением?**

Основной принцип заключается в том, чтобы выполнять поиск в глубину с увеличением ограничения по глубине на каждом шаге, начиная с глубины 0 и увеличивая её до тех пор, пока не будет найдено решение.

13. **Что представляет собой аргумент problem, передаваемый в функцию iterative_deepening_search?**

Аргумент problem представляет собой задачу, включающую начальное состояние, цели и действия, которые необходимо выполнить для поиска решения.

14. **Какова роль переменной limit в алгоритме?**

Переменная limit задает максимальную глубину, до которой будет проводиться поиск на текущем шаге итеративного углубления.

15. **Что означает использование диапазона range(1, sys.maxsize) в цикле?**

Диапазон range(1, sys.maxsize) используется для того, чтобы постепенно увеличивать глубину поиска до максимально возможного значения, ограниченного системными параметрами.

16. **Почему предел глубины поиска увеличивается постепенно, а не устанавливается сразу на максимальное значение?**

Постепенное увеличение предела глубины позволяет сначала искать на более мелких уровнях и не тратить ресурсы на большие глубины, если решение может быть найдено раньше.

17. **Какая функция вызывается внутри цикла и какую задачу она решает?**

Внутри цикла вызывается функция depth_limited_search, которая выполняет поиск в глубину с ограничением на текущую глубину.

18. **Что делает функция depth_limited_search, и какие результаты она может возвращать?**

Функция depth_limited_search выполняет поиск элемента с ограничением глубины. Она может возвращать True, если элемент найден, False, если элемент не найден, и "cutoff", если достигнута максимальная глубина.

19. **Какое значение представляет собой cutoff, и что оно обозначает в данном алгоритме?**

Значение "cutoff" обозначает, что алгоритм достиг предела глубины и не может продолжить поиск на этом уровне.

20. **Почему результат сравнивается с cutoff перед тем, как вернуть результат?**

Сравнение с "cutoff" нужно для того, чтобы понять, не было ли достигнуто ограничение глубины, и это предотвращает лишние рекурсивные вызовы.

21. **Что произойдет, если функция depth_limited_search найдет решение на первой итерации?**

Если решение найдено на первой итерации, функция сразу вернет True, и поиск завершится.

22. **Почему функция может продолжать выполнение до тех пор, пока не достигнет sys.maxsize?**

Функция может продолжать выполнение до достижения максимально возможной глубины (ограниченной системой), если решение не найдено до этого момента.

23. **Каковы преимущества использования поиска с итеративным углублением по сравнению с обычным поиском в глубину?**

Поиск с итеративным углублением использует меньше памяти, так как на каждом уровне хранится только информация о текущем поиске. Он также гарантирует, что решение будет найдено на минимальной глубине.

24. **Какие потенциальные недостатки может иметь этот подход?**

Основной недостаток — это повторное генерирование тех же узлов на каждом шаге, что может приводить к неэффективности, если решение находится на глубоком уровне.

25. **Как можно оптимизировать данный алгоритм для ситуаций, когда решение находится на больших глубинах?**

Можно использовать методы, такие как ограничение по глубине или комбинированные алгоритмы (например, A* или другие эвристические подходы), чтобы ускорить поиск решения на больших глубинах.

Вывод: в ходе лабораторной работы приобретены навыки по работе с поиском с итеративным углублением при написании программ помощью языка программирования Python.