

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»**

Выполнил:

Пустяков Андрей Сергеевич

3 курс, группа ИВТ-б-о-22-1,

09.03.01 «Информатика и  
вычислительная техника (профиль)  
«Программное обеспечение средств  
вычислительной  
техники и автоматизированных  
систем», очная форма обучения

---

(подпись)

Руководитель практики:

Воронкин Р. А., доцент департамента  
цифровых и робототехнических  
систем и электроники и института  
перспективной инженерии

---

(подпись)

Ставрополь, 2024 г.

Тема: Исследование поиска с итеративным углублением.

Цель: приобрести навыки по работе с поиском с итеративным углублением с помощью языка программирования Python версии 3.x.

Ход работы:

### Поиск с итеративным углублением

Для построенного графа городов Австралии (рис. 1) лабораторной работы 1 была написана программа на языке программирования Python, которая с помощью алгоритма поиска с итеративным углублением находит минимальное расстояние между начальным и конечным пунктами (для лабораторной работы 1 начальным пунктом являлся город Буринда, а конечный город Сидней).

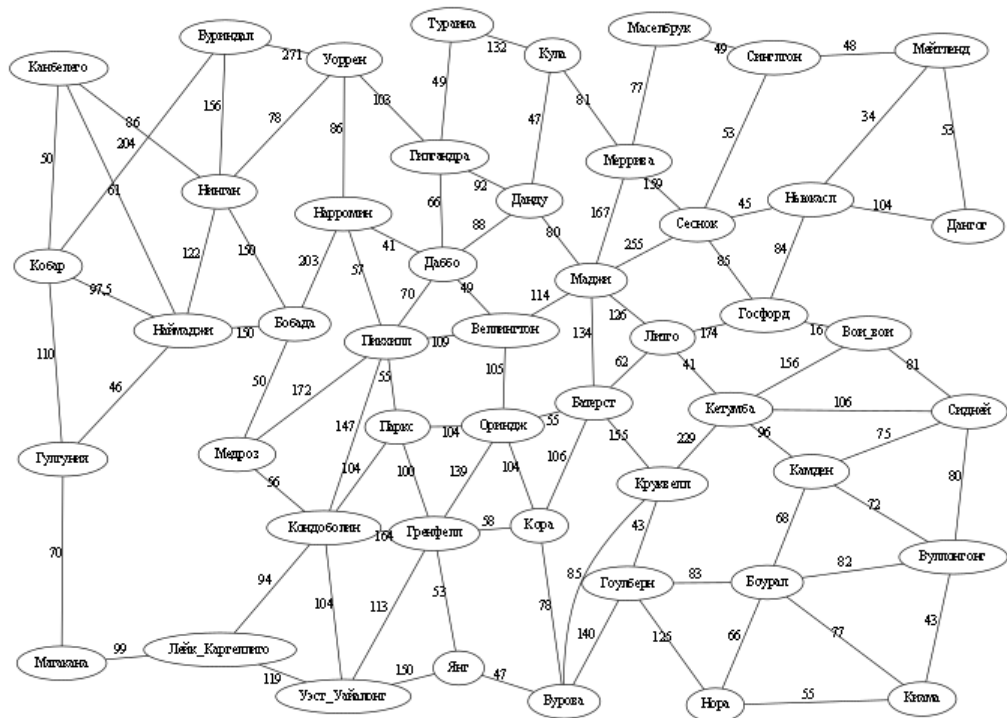


Рисунок 1 – Граф городов Австралии на языке DOT

Граф в программе был описан в виде словаря словарей с узлами и ребрами графа. Код программы нахождения пути в графе:

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

import math
import sys
from abc import ABC, abstractmethod

class Problem(ABC):
```

```

"""
Абстрактный класс для формальной постановки задачи.
Новый домен (конкретная задача) должен специализировать этот класс,
переопределяя методы actions и result, а при необходимости
action_cost, h и is_goal.
"""

def __init__(self, initial=None, goal=None, **kwargs):
    self.initial = initial
    self.goal = goal
    # Сохраняем все остальные переданные параметры (при желании).
    for k, v in kwargs.items():
        setattr(self, k, v)

@abstractmethod
def actions(self, state):
    """
    Вернуть доступные действия (операторы) из данного состояния.
    """
    pass

@abstractmethod
def result(self, state, action):
    """
    Вернуть результат применения действия к состоянию.
    """
    pass

def is_goal(self, state):
    """
    Проверка, является ли состояние целевым.
    """
    return state == self.goal

def action_cost(self, s, a, s1):
    """
    Возвращает стоимость применения действия a, переводящего s в s1 (по
    умолчанию 1).
    """
    return 1

def h(self, node):
    """
    Эвристическая функция; по умолчанию = 0.
    """
    return 0

def __str__(self):
    return f"{type(self).__name__}({self.initial!r}, {self.goal!r})"

class Node:
    """Узел в дереве поиска."""

    def __init__(self, state, parent=None, action=None, path_cost=0.0):
        self.state = state # Текущее состояние
        self.parent = parent # Родительский узел
        self.action = action # Действие, которое привело к этому узлу
        self.path_cost = path_cost # Стоимость пути

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

    def __lt__(self, other):

```

```

        return self.path_cost < other.path_cost

    def __len__(self):
        if self.parent is None:
            return 0
        else:
            return 1 + len(self.parent)

failure = Node("failure", path_cost=math.inf)
cutoff = Node("cutoff", path_cost=math.inf)

def expand(problem, node):
    """
    Генерируем (расширяем) дочерние узлы, применяя все действия к node.state.
    """
    s = node.state
    for action in problem.actions(s):
        s1 = problem.result(s, action)
        cost = node.path_cost + problem.action_cost(s, action, s1)
        yield Node(state=s1, parent=node, action=action, path_cost=cost)

def path_actions(node):
    """
    Восстанавливаем последовательность действий от корня до данного узла.
    """
    if node.parent is None:
        return []
    return path_actions(node.parent) + [node.action]

def path_states(node):
    """
    Восстанавливаем последовательность состояний (городов)
    от корня до данного узла.
    """
    if node.parent is None:
        return [node.state]
    return path_states(node.parent) + [node.state]

def is_cycle(node):
    """
    Проверяем, не образуется ли цикл (повтор состояния) при движении.
    Если состояние node.state уже встречалось в цепочке родительских узлов,
    значит это цикл.
    """
    s = node.state
    p = node.parent
    while p is not None:
        if p.state == s:
            return True
        p = p.parent
    return False

class LIFOQueue(list):
    """
    Реализация стека в виде списка
    """
    pass

```

```

def depth_limited_search(problem, limit):
    """
    Реализация алгоритма поиска с ограничением глубины.
    Пока стек не пуст извлекаем вершину. Если это цель, возвращаем.
    Иначе, если глубина >= limit, запоминаем result = cutoff.
    Если нет цикла, расширяем и складываем потомков в стек.
    Если цель не найдена, то возврат failure.
    """

    frontier = LIFOQueue([Node(problem.initial)])
    result = failure

    while frontier:
        node = frontier.pop()

        # Проверка на цель
        if problem.is_goal(node.state):
            return node

        # Проверка на лимит глубины
        if len(node) >= limit:
            result = cutoff
        elif not is_cycle(node):
            # Расширяем
            for child in expand(problem, node):
                frontier.append(child)

    return result

def iterative_deepening_search(problem):
    """
    Выполняем поиск с ограничением глубины, с возрастающим лимитом.
    Начинаем с limit=1 и до бесконечности.
    """
    for limit in range(1, sys.maxsize):
        result = depth_limited_search(problem, limit)
        if result != cutoff:
            return result

class MapProblem(Problem):
    """
    Дочерний класс для постановки задачи:
    Найти путь из одного города в другой по графу.
    """

    def __init__(self, initial, goal, graph):
        super().__init__(initial=initial, goal=goal)
        self.graph = graph

    def actions(self, state):
        """Все соседние города из state."""
        return list(self.graph[state].keys())

    def result(self, state, action):
        """Перейти в город 'action'."""
        return action

    def action_cost(self, s, a, s1):
        """Стоимость (расстояние) из s в s1."""
        return self.graph[s][s1]

```

```

def main():
    """
    Главная функция программы.
    """

    # Граф городов Австралии
    graph = {
        "Буриндал": {"Уоррен": 271, "Нинган": 156, "Кобар": 204},
        "Уоррен": {"Буриндал": 271, "Нинган": 78, "Гилгандра": 103,
        "Нарромин": 86},
        "Нинган": {"Буриндал": 156, "Канбелего": 86, "Уоррен": 78, "Бобада":
        150, "Наймаджи": 122},
        "Кобар": {"Буриндал": 204, "Канбелего": 50, "Наймаджи": 97.5,
        "Гулгуния": 110},
        "Канбелего": {"Кобар": 50, "Наймаджи": 61, "Нинган": 86},
        "Наймаджи": {"Канбелего": 61, "Нинган": 122, "Бобада": 150, "Кобар":
        97.5, "Гулгуния": 46},
        "Гулгуния": {"Кобар": 110, "Наймаджи": 46, "Матакана": 70},
        "Матакана": {"Гулгуния": 70, "Лейк Каргеллиго": 99},
        "Бобада": {"Наймаджи": 150, "Нинган": 150, "Нарромин": 203, "Медроз":
        50},
        "Нарромин": {"Уоррен": 86, "Бобада": 203, "Пикхилл": 57, "Даббо":
        41},
        "Пикхилл": {"Нарромин": 57, "Даббо": 70, "Веллингтон": 109, "Медроз":
        172, "Кондоболин": 147, "Паркс": 55},
        "Даббо": {"Нарромин": 41, "Гилгандра": 66, "Данду": 88, "Веллингтон":
        49, "Пикхилл": 70},
        "Гилгандра": {"Уоррен": 103, "Даббо": 66, "Тураина": 49, "Данду":
        92},
        "Данду": {"Даббо": 88, "Маджи": 80, "Гилгандра": 92, "Кула": 47},
        "Медроз": {"Бобада": 50, "Кондоболин": 56, "Пикхилл": 172},
        "Кондоболин": {
            "Медроз": 56,
            "Паркс": 104,
            "Гренфелл": 164,
            "Уэст Уайалонг": 104,
            "Лейк Каргеллиго": 94,
            "Пикхилл": 147,
        },
        "Паркс": {"Пикхилл": 55, "Кондоболин": 104, "Гренфелл": 100,
        "Ориндж": 104},
        "Гренфелл": {"Паркс": 100, "Ориндж": 139, "Кора": 58, "Янг": 53,
        "Кондоболин": 164, "Уэст Уайалонг": 113},
        "Ориндж": {"Кора": 104, "Паркс": 104, "Веллингтон": 105, "Батерст":
        55, "Гренфелл": 139},
        "Веллингтон": {"Маджи": 114, "Пикхилл": 109, "Даббо": 49, "Ориндж":
        105},
        "Маджи": {"Веллингтон": 114, "Данду": 80, "Меррива": 167, "Сеснок":
        255, "Батерст": 134, "Литго": 126},
        "Батерст": {"Литго": 62, "Кора": 106, "Круквелл": 155, "Ориндж": 55,
        "Маджи": 134},
        "Литго": {"Батерст": 62, "Кетумба": 41, "Маджи": 126, "Госфорд":
        174},
        "Кора": {"Гренфелл": 58, "Ориндж": 104, "Батерст": 106, "Бурова":
        78},
        "Бурова": {"Янг": 47, "Кора": 78, "Круквелл": 85, "Гоулберн": 140},
        "Круквелл": {"Батерст": 155, "Кетумба": 229, "Гоулберн": 43,
        "Бурова": 85},
        "Гоулберн": {"Бурова": 140, "Круквелл": 43, "Нора": 125, "Боурал":
        83},
        "Нора": {"Гоулберн": 125, "Боурал": 66, "Киама": 55},
        "Боурал": {"Нора": 66, "Киама": 77, "Вуллонгонг": 82, "Камден": 68,
        "Гоулберн": 83},
    }

```

```

    "Киамма": {"Нора": 55, "Вуллонгонг": 43, "Боурал": 77},
    "Вуллонгонг": {"Киамма": 43, "Боурал": 82, "Камден": 72, "Сидней":
80},
    "Камден": {"Боурал": 68, "Вуллонгонг": 72, "Кетумба": 96, "Сидней":
75},
    "Сидней": {"Вуллонгонг": 80, "Камден": 75, "Кетумба": 106, "Вои_вои":
81},
    "Кетумба": {"Литго": 41, "Сидней": 106, "Камден": 96, "Вои_вои": 156,
"Круквелл": 229},
    "Вои_вои": {"Сидней": 81, "Кетумба": 156, "Госфорд": 16},
    "Госфорд": {"Вои_вои": 16, "Ньюкасл": 84, "Сеснок": 85, "Литго":
174},
    "Ньюкасл": {"Госфорд": 84, "Сеснок": 45, "Мейтленд": 34, "Дангог":
104},
    "Сеснок": {"Госфорд": 85, "Ньюкасл": 45, "Меррива": 159, "Синглтон":
53, "Маджи": 255},
    "Мейтленд": {"Ньюкасл": 34, "Дангог": 53, "Синглтон": 48},
    "Дангог": {"Ньюкасл": 104, "Мейтленд": 53},
    "Синглтон": {"Сеснок": 53, "Мейтленд": 48, "Маселбрук": 49},
    "Маселбрук": {"Синглтон": 49, "Меррива": 77},
    "Меррива": {"Сеснок": 159, "Маселбрук": 77, "Кула": 81, "Маджи":
167},
    "Кула": {"Меррива": 81, "Данду": 47, "Тураина": 132},
    "Тураина": {"Кула": 132, "Гилгандра": 49},
    "Лейк_Каргеллиго": {"Матакана": 99, "Кондоболин": 94,
"Уэст_Уайалонг": 119},
    "Уэст_Уайалонг": {"Лейк_Каргеллиго": 119, "Кондоболин": 104,
"Гренфелл": 113, "Янг": 150},
    "Янг": {"Уэст_Уайалонг": 150, "Гренфелл": 53, "Бурова": 47},
}

# Создаём задачу
problem = MapProblem(initial="Буриндал", goal="Сидней", graph=graph)

solution_node = iterative_deepening_search(problem)

if solution_node is None or solution_node is failure:
    print("Путь не найден!")
elif solution_node is cutoff:
    print("Поиск прерван (cutoff).")
else:
    # Восстанавливаем маршрут
    route = path_states(solution_node)
    print("Найден маршрут:", " -> ".join(route))
    print("Суммарная стоимость:", solution_node.path_cost)

if __name__ == "__main__":
    main()

```

Результаты выполнения данного кода, маршрут из начального города в конечный (из города Буриндал в город Сидней) (рис. 2).

```

C:\Users\Andrey\AppData\Local\pypoetry\Cache\virtualenvs\ai-lab-5-5gsaALYv-py3.12\Scripts\python.exe
Найден маршрут: Буриндал -> Уоррен -> Гилгандра -> Данду -> Маджи -> Литго -> Кетумба -> Сидней
Суммарная стоимость: 819.0

```

Рисунок 2 – Результаты работы алгоритма с ограничением глубины





Код программы поиска пользователя в системе с помощью алгоритма итеративного углубления:

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

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

import math
import sys
from abc import ABC, abstractmethod

class BinaryTreeNode:
    def __init__(self, value, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

    def add_children(self, left, right):
        self.left = left
        self.right = right

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

class Problem(ABC):
    """
    Абстрактный класс для формальной постановки задачи.
    Новый домен (конкретная задача) должен специализировать этот класс,
    переопределяя методы actions и result, а при необходимости
    action_cost, h и is_goal.
    """

    def __init__(self, initial=None, goal=None):
        """
        initial: корневой узел (BinaryTreeNode)
        goal: искомое значение
        """
        self.initial = initial
        self.goal = goal

    @abstractmethod
    def actions(self, state):
        """
        Вернуть доступные действия (операторы) из данного состояния.
        """
        pass

    @abstractmethod
    def result(self, state, action):
        """
        Вернуть результат применения действия к состоянию.
        """
        pass
```

```

def is_goal(self, state):
    """
    Проверка, является ли состояние целевым.
    """
    return state.value == self.goal

def action_cost(self, s, a, sl):
    """
    По умолчанию 1.
    """
    return 1

def h(self, node):
    """
    Эвристическая функция, по умолчанию 0.
    """
    return 0

class Node:
    """
    Узел в дереве поиска.
    """

    def __init__(self, state, parent=None, action=None, path_cost=0.0):
        self.state = state          # Текущее состояние
        self.parent = parent        # Родительский узел
        self.action = action        # Действие, которое привело сюда
        self.path_cost = path_cost

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

    def __lt__(self, other):
        return self.path_cost < other.path_cost

    def __len__(self):
        if self.parent is None:
            return 0
        else:
            return 1 + len(self.parent)

failure = Node("failure", path_cost=math.inf)
cutoff = Node("cutoff", path_cost=math.inf)

def expand(problem, node):
    """
    Генерируем дочерние узлы, применяя actions(state).
    """
    s = node.state
    for action in problem.actions(s):
        s_next = problem.result(s, action)
        yield Node(state=s_next, parent=node, action=action,
                    path_cost=node.path_cost + problem.action_cost(s, action,
s_next))

class LIFOQueue(list):
    """
    Реализация стека в виде списка.
    """
    pass

```

```

def depth_limited_search(problem, limit=10):
    """
    Поиск с ограничением глубины/
    Пока стек не пуст, извлекаем вершину.
    Если это цель, возвращаем её.
    Если глубина >= limit, запоминаем result = cutoff.
    Если все исчерпано, return failure.
    """
    frontier = LIFOQueue([Node(problem.initial)])
    result = failure

    while frontier:
        node = frontier.pop()

        if problem.is_goal(node.state):
            return node

        if len(node) >= limit:
            result = cutoff

        for child in expand(problem, node):
            frontier.append(child)

    return result

def iterative_deepening_search(problem):
    """
    Функция осуществления итеративного углубления.
    """
    for limit in range(1, sys.maxsize):
        result = depth_limited_search(problem, limit=limit)
        if result != cutoff:
            return result

class UserSearchProblem(Problem):
    """
    Описание конкретной задачи.
    Дано бинарное дерево (BinaryTreeNode) с пользователями.
    Нужно проверить, существует ли узел с value == goal.
    """

    def actions(self, state):
        """
        Действия: перейти в left или в right, если они существуют
        """
        moves = []
        if state.left:
            moves.append(state.left)
        if state.right:
            moves.append(state.right)
        return moves

    def result(self, state, action):
        return action

def main():
    """
    Главная функция программы.
    """

```

```

root = BinaryTreeNode(1)
left_child = BinaryTreeNode(2)
right_child = BinaryTreeNode(3)

# Подвешиваем их к корню
root.add_children(left_child, right_child)

# Расширяем поддерево слева
left_left = BinaryTreeNode(6, left=BinaryTreeNode(8))
left_right = BinaryTreeNode(7)
left_child.add_children(left_left, left_right)

# Расширяем поддерево справа
right_left = BinaryTreeNode(9, right=BinaryTreeNode(4))
right_right = BinaryTreeNode(5)
right_child.add_children(right_left, right_right)

# Целевое значение (идентификатор пользователя)
goal = 4

problem = UserSearchProblem(initial=root, goal=goal)

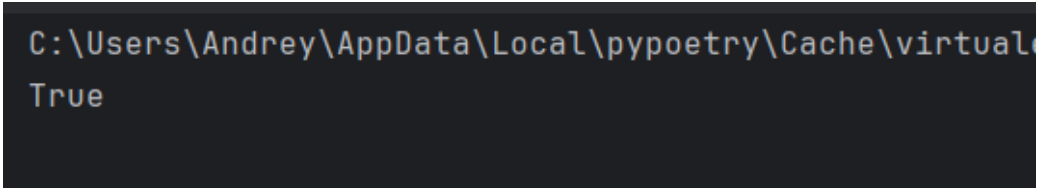
solution_node = iterative_deepening_search(problem)

if solution_node is None or solution_node is failure:
    print(False)
else:
    # Если это не cutoff, либо реальный узел с решением
    if solution_node is cutoff:
        print(False)
    else:
        print(True)

if __name__ == "__main__":
    main()

```

Результаты работы программы, результат нахождения узла в созданном дереве (рис. 4).



```

C:\Users\Andrey\AppData\Local\pypoetry\Cache\virtual...
True

```

Рисунок 4 – Пользователь с указанным идентификатором был найден в дереве

### Поиск в файловой системе

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 поиска файла в дереве каталогов с использованием алгоритма с итеративным углублением:

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

"""
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.
"""

import math
import sys
from abc import ABC, abstractmethod

class TreeNode:
    def __init__(self, value):
        self.value = value
        self.children = []

    def add_child(self, child):
        self.children.append(child)

    def add_children(self, *args):
        for child in args:
            self.add_child(child)

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

class Problem(ABC):
    """
    Абстрактный класс для формальной постановки задачи.
    Новый домен (конкретная задача) должен специализировать этот класс,
    переопределяя методы actions и result, а при необходимости
    action_cost, h и is_goal.
    """

    def __init__(self, initial=None, goal=None):
        """
        initial: корневой узел (TreeNode);
        goal: имя искомого файла.
        """
        self.initial = initial
        self.goal = goal

    @abstractmethod
    def actions(self, state):
        """
        Вернуть доступные действия (операторы) из данного состояния.
        """
        pass

    @abstractmethod
```

```

def result(self, state, action):
    """
    Вернуть результат применения действия к состоянию.
    """
    pass

def is_goal(self, state):
    """
    Проверка, является ли состояние целевым.
    """
    return state.value == self.goal

def action_cost(self, s, a, sl):
    """По умолчанию = 1."""
    return 1

def h(self, node):
    """Эвристическая функция; по умолчанию 0."""
    return 0

class Node:
    """
    Узел в дереве поиска.
    """

    def __init__(self, state, parent=None, action=None, path_cost=0.0):
        self.state = state          # Текущее состояние
        self.parent = parent        # Родительский узел
        self.action = action        # Действие
        self.path_cost = path_cost

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

    def __lt__(self, other):
        """Для приоритетных очередей; здесь не используется."""
        return self.path_cost < other.path_cost

    # Глубина узла – длина пути от корня (по цепочке parent)
    def __len__(self):
        if self.parent is None:
            return 0
        return 1 + len(self.parent)

# Специальные «сигнальные» узлы
failure = Node("failure", path_cost=math.inf)
cutoff = Node("cutoff", path_cost=math.inf)

def expand(problem, node):
    """
    Генерируем (расширяем) дочерние узлы, применяя actions(state).
    """

    s = node.state
    for action in problem.actions(s):
        s_next = problem.result(s, action)
        yield Node(state=s_next, parent=node, action=action,
                    path_cost=node.path_cost + problem.action_cost(s, action,
s_next))

```

```

def path_states(node):
    """
    Восстановление последовательности состояний
    (каталогов/файлов) от корня до данного узла.
    """
    if node.parent is None:
        return [node.state.value]
    return path_states(node.parent) + [node.state.value]

class LIFOQueue(list):
    """
    Реализация стека в виде списка.
    """
    pass

def depth_limited_search(problem, limit):
    """
    Поиск с ограничением глубины.
    Пока стек не пуст, извлекаем вершину.
    Если это цель, возвращаем её.
    Если глубина >= limit, result = cutoff.
    Иначе расширяем вершину, добавляя потомков в стек.
    Если всё исчерпано и цель не найдена, возвращаем failure.
    """
    frontier = LIFOQueue([Node(problem.initial)])
    result = failure

    while frontier:
        node = frontier.pop()

        # Если это цель — возврат
        if problem.is_goal(node.state):
            return node

        # Если глубина достигла лимита, помечаем как cutoff
        if len(node) >= limit:
            result = cutoff
        else:
            # Расширяем
            for child in expand(problem, node):
                frontier.append(child)

    return result

def iterative_deepening_search(problem):
    """
    Реализация итеративного углубления.
    Многократно вызывает depth_limited_search,
    увеличивая limit от 1 до "бесконечности".
    Если результат != cutoff, то возвращает его.
    """
    for limit in range(1, sys.maxsize):
        result = depth_limited_search(problem, limit=limit)
        if result != cutoff:
            return result

class FileSearchProblem(Problem):
    """
    Описание задачи.
    Имеется дерево (TreeNode, файловая система),

```

```

        нужно найти узел, чье value == goal.
        """
    def actions(self, state):
        """
        Действия: перейти к дочернему каталогу/файлу.
        """
        return state.children

    def result(self, state, action):
        return action

def main():
    """
    Главная функция программы.
    """

    # Создание структуры файлов
    root = TreeNode("root")
    subdir_1 = TreeNode("subdir_1")
    subdir_2 = TreeNode("subdir_2")
    file_a = TreeNode("file_a")
    subdir_3 = TreeNode("subdir_3")
    file_b = TreeNode("file_b")
    file_c = TreeNode("file_c")
    file_d = TreeNode("file_d")

    root.add_children(subdir_1, subdir_2)
    subdir_1.add_children(file_a, subdir_3)
    subdir_3.add_children(file_b, file_d)
    subdir_2.add_child(file_c)

    goal = "file_d"

    problem = FileSearchProblem(initial=root, goal=goal)

    solution_node = iterative_deepening_search(problem)

    if solution_node is None or solution_node is failure:
        print("False")
    else:
        if solution_node is cutoff:
            print("False")
        else:
            # Восстанавливаем путь до целевого файла
            route = path_states(solution_node)
            print(" -> ".join(route))

if __name__ == "__main__":
    main()

```

Результаты работы программы, найденный путь до файла в файловой системе (рис. 5).



```
C:\Users\Andrey\AppData\Local\pypoetry\O
root -> subdir_1 -> subdir_3 -> file_d

Process finished with exit code 0
```

Рисунок 5 – Путь до указанного файла в системе

Выполнение индивидуального задания:

Вариант 25(5)

Задание 5.

Поиск файла с заданным уровнем доступа. Необходимо найти все файлы, у которых права доступа установлены как «`rwxr-xr--`», начиная с глубины 3 уровней. Используйте итеративное углубление и остановите поиск при нахождении первых 10 таких файлов.

Данная запись означает, что у файла есть разрешения и на запись, и на чтение, и на исполнение для владельца файла, для группы пользователей есть право только на чтение и исполнение, а для всех остальных только на чтение.

Необходимо создать программу, которая будет искать файлы поиском в глубину с итеративным углублением и проверять файлы на соответствующие права доступа в файловой системе Windows.

Так как в ОС Windows права доступа устроены иначе, чем в Linux, то будем искать файлы с правами на чтение и запись данного пользователя.

Код программы поиска файлов с требуемыми правами:

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

"""
Поиск файла с заданным уровнем доступа. Необходимо найти все файлы,
у которых права доступа установлены как «rwxr-xr--», начиная с глубины
3 уровней. Используйте итеративное углубление и остановите поиск при
нахождении первых 10 таких файлов.
"""

import os
import sys
import math
from abc import ABC, abstractmethod

class FSNode:
```

```

        Содержит путь (str) к файлу/директории.
        """
    def __init__(self, path):
        self.path = path

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

class Problem(ABC):
    """
    Абстрактный класс для формальной постановки задачи.
    """

    def __init__(self, initial=None):
        """
        initial: начальная директория (FSNode).
        """
        self.initial = initial

    @abstractmethod
    def actions(self, state):
        """
        Вернуть доступные действия (операторы) из данного состояния.
        """
        pass

    @abstractmethod
    def result(self, state, action):
        """
        Вернуть результат применения действия к состоянию.
        """
        pass

    def is_goal(self, state):
        """
        Проверяем, является ли файл подходящим.
        """
        return False

    def action_cost(self, s, a, sl):
        """
        По умолчанию = 1.
        """
        return 1

    def h(self, node):
        """
        Эвристика, по умолчанию = 0.
        """
        return 0

class Node:
    """
    Узел в дереве поиска.
    """

    def __init__(self, state, parent=None, action=None, path_cost=0.0):
        self.state = state
        self.parent = parent
        self.action = action
        self.path_cost = path_cost

```

```

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

def __len__(self):
    if self.parent is None:
        return 0
    return 1 + len(self.parent)

failure = Node("failure", path_cost=math.inf)
cutoff = Node("cutoff", path_cost=math.inf)

def expand(problem, node):
    s = node.state
    for action in problem.actions(s):
        s_next = problem.result(s, action)
        cost = node.path_cost + problem.action_cost(s, action, s_next)
        yield Node(state=s_next, parent=node, action=action, path_cost=cost)

class LIFOQueue(list):
    pass

def depth_limited_search(problem, limit, found_files=None):
    """
    Поиск с ограничением глубины.
    Ищет файлы, соответствующие условию (is_goal),
    но только начиная с глубины >= 3, которые собираются в found_files.
    При достижении 10 найденных файлов — возврат.
    """

    if found_files is None:
        found_files = []

    frontier = LIFOQueue([Node(problem.initial)])
    result = failure

    while frontier:
        node = frontier.pop()

        # Если глубина >= 3 и это цель — добавляем в список
        depth = len(node)
        if depth >= 3 and problem.is_goal(node.state):
            found_files.append(node.state.path) # строка пути
            # если достигли 10 — прерываем
            if len(found_files) >= 10:
                return node

        if depth >= limit:
            result = cutoff
        else:
            for child in expand(problem, node):
                frontier.append(child)

    return result

def iterative_deepening_search(problem, start_depth, found_files):
    """
    Реализация итеративного углубления.
    Начинаем с limit = start_depth, увеличиваем,
    пока не соберём 10 файлов или не закончатся пути.
    """

```

```

"""

for limit in range(start_depth, sys.maxsize):
    result = depth_limited_search(problem, limit=limit,
found_files=found_files)
    # Если вернулся "failure", значит вообще нет путей
    # Если вернулся не cutoff, значит закончились пути
    if len(found_files) >= 10:
        return

    if result != cutoff:
        # Значит все пути рассмотрены и либо ничего не найдено,
        # либо нашлось меньше 10, и расширять дальше смысла нет
        return

class WindowsFileSearchProblem(Problem):
    """
    Описание задачи поиска файлов.
    Ищем файлы в файловой системе Windows,
    у которых есть права и на чтение и на запись,
    начиная с заданной директории (initial.path).
    """

    def actions(self, state):
        """
        Действия: перейти (спуститься) в каждый файл / подпапку
        внутри текущей папки (если это папка).
        Возвращает список путей (FSNode) для подпапок и файлов.
        """
        path = state.path
        if os.path.isdir(path):
            try:
                items = os.listdir(path)
            except (PermissionError, FileNotFoundError):
                return [] # нет доступа/нет файла
            result = []
            for item in items:
                full = os.path.join(path, item)
                result.append(FSNode(full))
            return result
        else:
            return [] # если это файл, не расширяем

    def result(self, state, action):
        return action

    def is_goal(self, state):
        """
        Цель: проверить права, на чтение и запись.
        """
        path = state.path
        if os.path.isfile(path):
            try:
                mode = os.stat(path).st_mode
                # Проверяем, равны ли биты mode == 33206, то есть проверка на
чтение и запись
                return mode == 33206
            except PermissionError:
                return False
        return False

def main():

```

```

"""
Главная функция программы.
"""

# Начальная папка
root_path = r"C:\Users\Andrey\Desktop"

# По условию задачи необходимо начинать с глубины 3
start_depth = 3

problem = WindowsFileSearchProblem(initial=FSNode(root_path))

# Список для найденных файлов
found_files = []

iterative_deepening_search(problem, start_depth, found_files)

# Выводим результаты
if not found_files:
    print("Ничего не найдено.")
else:
    print("Найденные файлы (не более 10):")
    for f in found_files:
        print(f)

if __name__ == "__main__":
    main()

```

Результаты работы программы, найденные файлы и путь к ним в директории «C:\Users\Andrey\Desktop» (на рабочем столе), причем файлы, лежащие в директории глубже 3х и первые 10 штук (рис. 6).

```

Найденные файлы (не более 10):
C:\Users\Andrey\Desktop\Схемотехника\Лекции\Лекция Полевые транзисторы.docx
C:\Users\Andrey\Desktop\Схемотехника\Лекции\ЛЕКЦИЯ 1 Физ. основы .docx
C:\Users\Andrey\Desktop\Схемотехника\Лекции\Lektsia.docx
C:\Users\Andrey\Desktop\Схемотехника\Лабораторная работа 4\Отчет по ЛР 4 Пустяков.docx
C:\Users\Andrey\Desktop\Схемотехника\Лабораторная работа 4\Лаб_раб_4_иссл_обр_связей.pdf
C:\Users\Andrey\Desktop\Схемотехника\Лабораторная работа 4\Design1.ms14 (Security copy)
C:\Users\Andrey\Desktop\Схемотехника\Лабораторная работа 4\Design1.ms14
C:\Users\Andrey\Desktop\Схемотехника\Лабораторная работа 3\расчеты.xmcd
C:\Users\Andrey\Desktop\Схемотехника\Лабораторная работа 3\Отчет по ЛР 3 Пустяков.docx
C:\Users\Andrey\Desktop\Схемотехника\Лабораторная работа 3\Отчет по ЛР 3 Пустяков v2.docx

Process finished with exit code 0

```

Рисунок 6 – Результаты поиска файлов с правами доступа

Ссылка на репозиторий данной лабораторной работы:

[https://github.com/AndreyPust/Artificial\\_Intelligence\\_laboratory\\_work\\_5.git](https://github.com/AndreyPust/Artificial_Intelligence_laboratory_work_5.git)

Ответы на контрольные вопросы:

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

Параметр  $n$  в контексте поиска с ограниченной глубиной определяет текущую глубину узла. Он влияет на поиск, задавая ограничение на количество уровней, которые может пройти алгоритм, предотвращая исследование узлов, выходящих за пределы этой глубины.

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

Оптимальное значение глубины  $d$  зависит от неизвестной заранее глубины решения. Если  $d$  слишком мало, решение может быть пропущено; если слишком велико, алгоритм может потреблять лишние ресурсы или изучать ненужные пути.

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

Итеративное углубление использует меньше памяти, так как оно работает как поиск в глубину, при этом сохраняя полноту и оптимальность поиска в ширину, если стоимости переходов равны.

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

Алгоритм выполняет поиск в глубину с увеличением лимита глубины на каждой итерации. Это помогает избежать экспоненциального роста памяти, характерного для поиска в ширину, так как для поиска в глубину требуется только стек глубины.

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

Начинать поиск заново необходимо, потому что алгоритм в предыдущей итерации не сохраняет узлы, которые превышают текущий лимит глубины. Это позволяет экономить память.

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

Временная сложность:  $O(bd)$ , где  $b$  – коэффициент разветвления,  $d$  – глубина решения.

Пространственная сложность:  $O(d)$ , так как требуется память только для стека глубины.

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

Он использует память, как поиск в глубину, но сохраняет полноту и оптимальность, как поиск в ширину, постепенно исследуя узлы на увеличивающихся уровнях глубины.

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

Большая часть времени уходит на исследование узлов на самом глубоком уровне, так как число узлов растет экспоненциально. Повторное исследование более верхних уровней имеет относительно низкую стоимость.

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

Алгоритм генерирует порядка  $bd$  узлов. Однако за счет повторения на малых глубинах суммарное количество сгенерированных узлов составляет  $O(b^2d)$ .

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

Если генерация узлов на верхних уровнях дерева требует значительных вычислений, то повторное их исследование может быть дорогостоящим. Также он неэффективен, если стоимость переходов не одинакова.

11. Какую задачу решает функция `iterative_deepening_search`?

Она находит решение задачи методом итеративного углубления, возвращая путь к решению или указание на то, что решение отсутствует.

12. Каков основной принцип работы поиска с итеративным углублением?

Постепенное увеличение предела глубины и выполнение поиска в глубину на каждом уровне до тех пор, пока не будет найдено решение.

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

Это объект, описывающий задачу поиска, включающий начальное состояние, функцию определения целей и правила переходов.

14. Какова роль переменной `limit` в алгоритме?

Она задает текущую максимальную глубину поиска.

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

Диапазон задает последовательное увеличение предела глубины с минимального значения до максимально возможного в системе.

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

Это позволяет найти решение на минимальной глубине, избегая ненужных затрат ресурсов на более глубокие уровни.

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

Функция `depth_limited_search` выполняет поиск с ограничением глубины, проверяя, можно ли найти решение в пределах текущей глубины.

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

Она проверяет узлы до заданной глубины. Возвращает либо путь к решению, либо указание, что поиск был «обрезан» (`cutoff`), либо «неудачу».

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

`Cutoff` обозначает, что поиск достиг текущего предела глубины и не смог продолжить исследование.

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



Это позволяет алгоритму определить, нужно ли увеличить глубину на следующей итерации.

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

Алгоритм завершится, вернув найденное решение.

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

Это позволяет алгоритму исследовать все возможные уровни глубины, если решение расположено на большом расстоянии.

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

Он сохраняет полноту и оптимальность, чего не может гарантировать обычный поиск в глубину.

24. Какие потенциальные недостатки может иметь этот подход?

Повторное исследование узлов может быть дорогостоящим в терминах времени, особенно если дерево большое или генерация узлов сложна.

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

- использование эвристик для пропуска заведомо нерелевантных узлов;
- применение более эффективного способа хранения и повторного использования верхних уровней дерева.

Вывод: в ходе выполнения лабораторной работы были приобретены навыки по работе с поиском с итеративным углублением с помощью языка программирования Python версии 3.x.