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 №4

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 ограничением глубины.

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

Ход работы:

Поиск с ограничением глубины

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

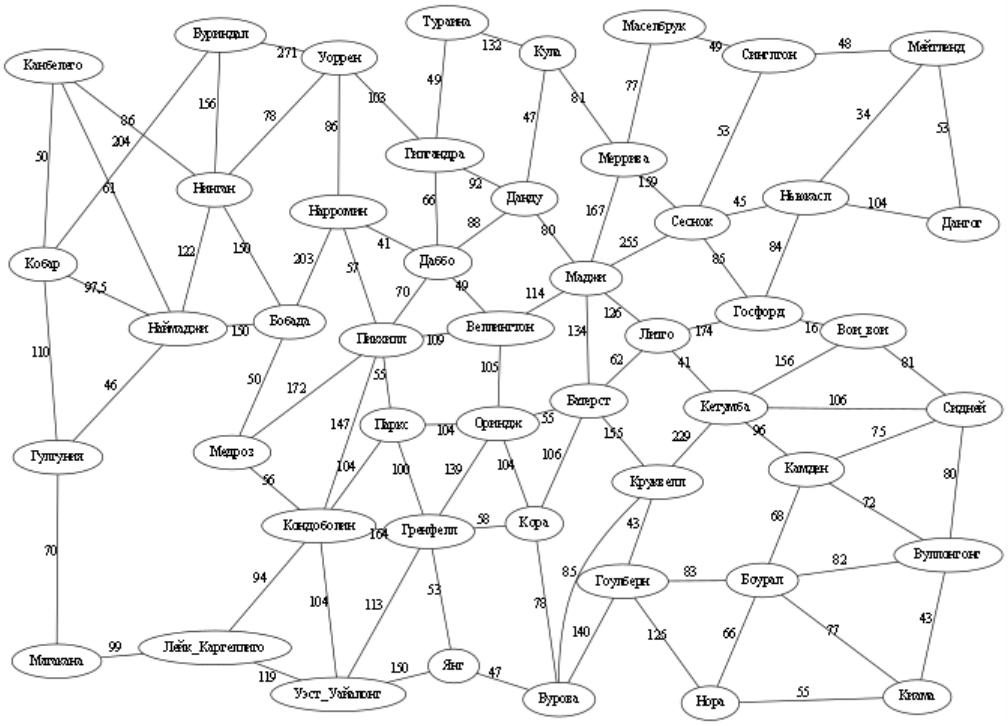

Рисунок 1 – Граф городов Австралии на языке DOT

Граф в программе был описан в виде словаря словарей с узлами и ребрами графа. Пусть глубина поиска limit = 7. Код программы нахождения кратчайшего пути в графе:

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

import math
from abc import ABC, abstractmethod
```

```python
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):
        """Для приоритетных очередей; здесь не нужно, но пусть будет."""
        return self.path_cost < other.path_cost

    # Глубина узла — длина пути от корня (считаем рекурсивно)
    def __len__(self):
        if self.parent is None:
            return 0
        else:
            return 1 + len(self.parent)
```

```python
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=10):
    """
    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


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},
        "Киама": {"Нора": 55, "Вуллонгонг": 43, "Боурал": 77},
        "Вуллонгонг": {"Киама": 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)

    # Ищем решение с ограничением глубины
    solution_node = depth_limited_search(problem, limit=7)

    if solution_node is None or solution_node is failure:
        print("Путь не найден!")
    elif solution_node is cutoff:
        print("Поиск прерван (cutoff): достигнут лимит глубины!")
    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).

```
Найден маршрут: Буриндал -> Уоррен -> Гилгандра -> Данду -> Маджи -> Литго -> Кетумба -> Сидней
Суммарная стоимость: 819.0
```

Рисунок 2 – Результаты работы алгоритма с ограничением глубины

Если увеличивать глубину поиска, то путь от данного города к конечному может быть больше, причем при глубине поиска равной 6 решение найти уже нельзя. Путь не является оптимальнейшим, но с точки зрения посещения количества городов он является оптимальным и вполне коротки по сравнению с решением, найденным с помощью поиска в глубину без ограничения. Путь, найденный с помощью алгоритма с ограничением глубины на графе (рис. 3).

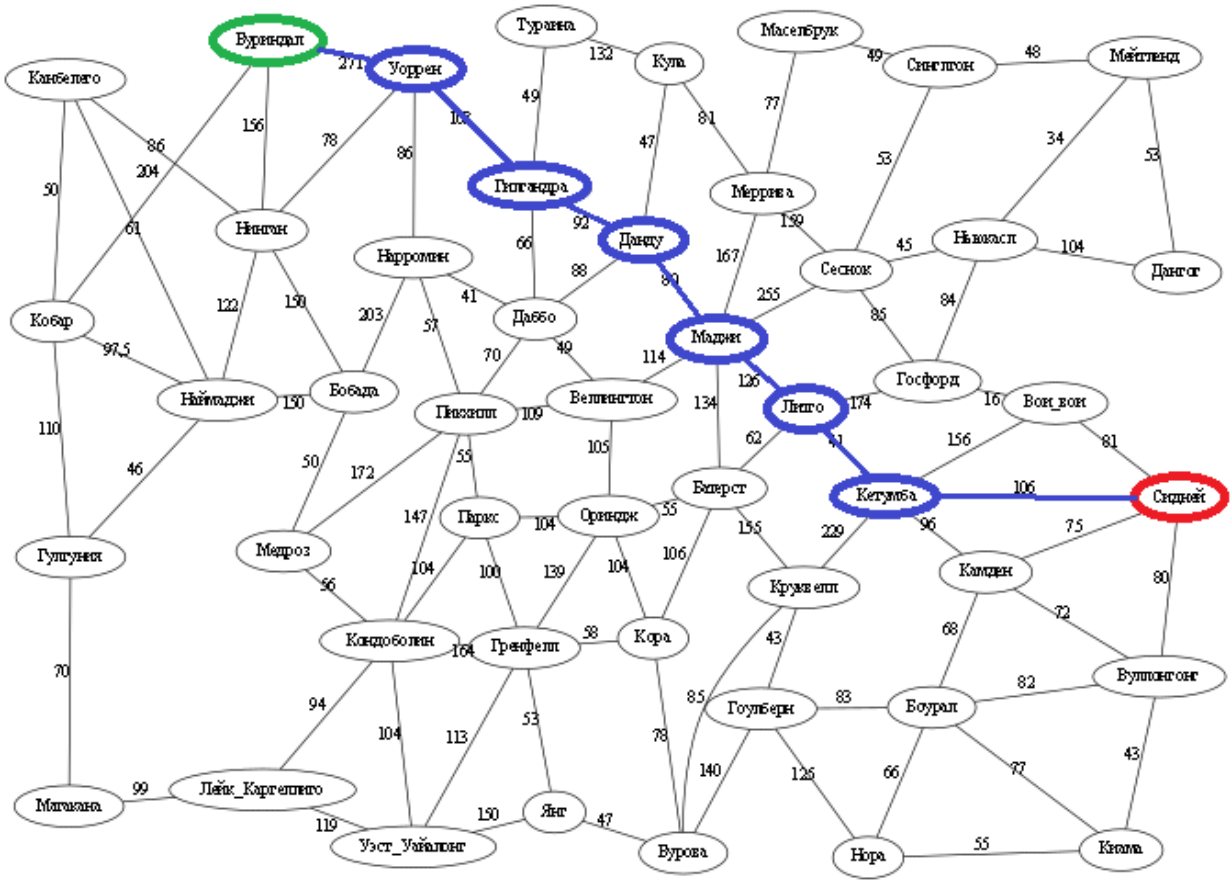

Рисунок 3 – Путь, найденный с помощью ограничения глубины

Система навигации робота-пылесоса

Робот способен передвигаться по различным комнатам в доме, но из-за ограниченности ресурсов (например, заряда батареи) и времени на уборку, важно эффективно выбирать путь. Необходимо реализовать алгоритм, который поможет роботу определить, существует ли путь к целевой комнате, не превышая заданное ограничение по глубине поиска.

Дано дерево, где каждый узел представляет собой комнату в доме. Узлы связаны в соответствии с возможностью перемещения робота из одной комнаты в другую. Необходимо определить, существует ли путь от начальной комнаты (корень дерева) к целевой комнате (узел с заданным значением), так, чтобы робот не превысил лимит по глубине перемещения.

Код программы, определяющей существует ли путь от начальной комнаты к целевой комнате при заданной ограниченной глубине:

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

"""
```

```
Дано дерево, где каждый узел представляет собой комнату в доме.
Узлы связаны в соответствии с возможностью перемещения робота
из одной комнаты в другую. Необходимо определить, существует ли
путь от начальной комнаты (корень дерева) к целевой комнате (узел
с заданным значением), так, чтобы робот не превысил лимит по
глубине перемещения.
"""

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


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

    def __init__(self, initial=None, goal=None):
        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, s1):
        """
        Стоимость действий, по умолчанию == 1.
        """
        return 1

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


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}>"

    # Глубина узла — длина пути от корня
    def depth(self):
        if self.parent is None:
            return 0
        return 1 + self.parent.depth()


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


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


def depth_limited_search(problem, limit):
    """
    Функция поиска пути к целевой комнате (узлу), но не глубже, чем limit.
    Если находим цель, возвращаем узел.
    Если все пути исчерпаны и ничего не найдено, возвращаем failure.
    Если достигли глубины limit, то возврат cutoff.
    """

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

    while frontier:
        node = frontier.pop()

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

        # Проверка ограничения по глубине
        if node.depth() >= limit:
            result = cutoff
        else:
            # Расширение узла
            for child in expand(problem, node):
                frontier.append(child)

    return result


class RoomNavigationProblem(Problem):
    """
    Имеется дерево (BinaryTreeNode) с начальным узлом root.
    Нужно найти узел, значение которого = goal.
    Движения: можно пойти в left или right.
```

```
    """

    def actions(self, state):
        """
        Возвращаем список "дочерних" узлов (left, right), если они есть.
        """

        children = []
        if state.left:
            children.append(state.left)
        if state.right:
            children.append(state.right)
        return children

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


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

    # Создание бинарного дерева
    root = BinaryTreeNode(
        1,
        left=BinaryTreeNode(2, None, BinaryTreeNode(4)),
        right=BinaryTreeNode(3, BinaryTreeNode(5), None),
    )
    goal = 4
    limit = 2

    problem = RoomNavigationProblem(root, goal)
    solution_node = depth_limited_search(problem, limit)

    if solution_node is failure:
        print(f"Найден на глубине: False")
    elif solution_node is cutoff:
        print(f"Найден на глубине: False (достигли limit={limit}, нужна
большая глубина)")
    else:
        # решение найдено
        print("Найден на глубине: True")


if __name__ == "__main__":
    main()
```

Результаты работы программы для заданного бинарного дерева и глубины 2, если глубина будет меньше 2, то решение не будет найдено, к комнате нельзя будет подобраться (рис. 4).

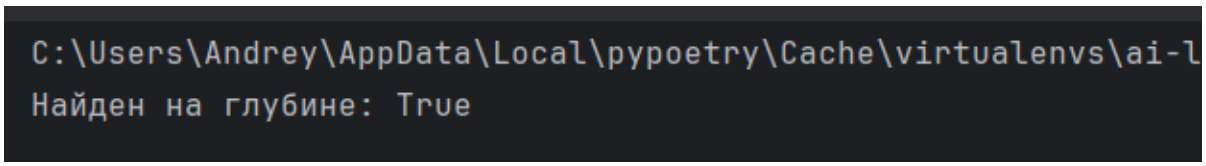

Рисунок 4 – Результаты работы программы поиска целевой комнаты с ограничением глубины

Система управления складом

В системе управления складом товары упорядочены в структуре, похожей на двоичное дерево. Каждый узел дерева представляет место хранения, которое может вести к другим местам хранения (левому и правому подразделу). Необходимо найти наименее затратный путь к товару, ограничив поиск заданной глубиной, чтобы гарантировать, что поиск займет приемлемое время.

Код программы нахождения целевого место в системе с заданной глубиной:

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

"""
В системе управления складом товары упорядочены в структуре,
похожей на двоичное дерево. Каждый узел дерева представляет
место хранения, которое может вести к другим местам хранения
(левому и правому подразделу). Необходимо найти наименее
затратный путь к товару, ограничив поиск заданной глубиной,
чтобы гарантировать, что поиск займет приемлемое время.
"""

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


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

    def __init__(self, initial=None, goal=None):
        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):
        """
        Проверяем, достигли ли мы узла с нужным value = self.goal.
        state: объект BinaryTreeNode.
        """
        return (state.value == self.goal)

    def action_cost(self, s, a, s1):
        """По умолчанию = 1 (не используется в глубинном поиске)."""
        return 1


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 depth(self):
        """Глубина узла — расстояние от корня."""
        if self.parent is None:
            return 0
        return self.parent.depth() + 1


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


def expand(problem, node):
    """
    Функция раскрытия узлов.
    """
    for action in problem.actions(node.state):
        s_next = problem.result(node.state, action)
        yield Node(
            state=s_next,
            parent=node,
            action=action,
            path_cost=node.path_cost + 1,
        )
```

```
def depth_limited_search(problem, limit=2):
    """
    Проверяет существование пути к цели, не превышая глубину 'limit'.
    """

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

    while frontier:
        node = frontier.pop()

        # Проверяем, не цель ли это
        if problem.is_goal(node.state):
            return node

        # Если достигли предельной глубины, то cutoff
        if node.depth() >= limit:
            result = cutoff
        else:
            # Расширение узла
            for child in expand(problem, node):
                frontier.append(child)

    return result


class WarehouseProblem(Problem):
    """
    Описание задачи системы управления складом в двоичном дереве товаров.
    """

    def actions(self, state):
        """
        Список "действий" — это переходы к left или right.
        """
        children = []
        if state.left is not None:
            children.append(state.left)
        if state.right is not None:
            children.append(state.right)
        return children

    def result(self, state, action):
        """
        Переход в узел 'action'.
        """
        return action


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

    root = BinaryTreeNode(
        1,
        left=BinaryTreeNode(2, None, BinaryTreeNode(4)),
        right=BinaryTreeNode(3, BinaryTreeNode(5), None),
    )

    goal = 4
    limit = 2

    # Создаём задачу
```

```
    problem = WarehouseProblem(initial=root, goal=goal)

    # Ищем узел со значением 4, глубина не более limit=2
    solution_node = depth_limited_search(problem, limit=limit)

    if solution_node is failure:
        print("Цель не найдена!")
    elif solution_node is cutoff:
        print(f"Глубина поиска достигла лимита={limit}, решение не найдено на
этой глубине.")
    else:
        # Решение найдено
        print("Цель найдена:", solution_node.state)


if __name__ == "__main__":
    main()
```

Результаты работы программы, нахождение места на заданной глубине (если указать меньшую глубину, цель не будет найдена) (рис. 5).

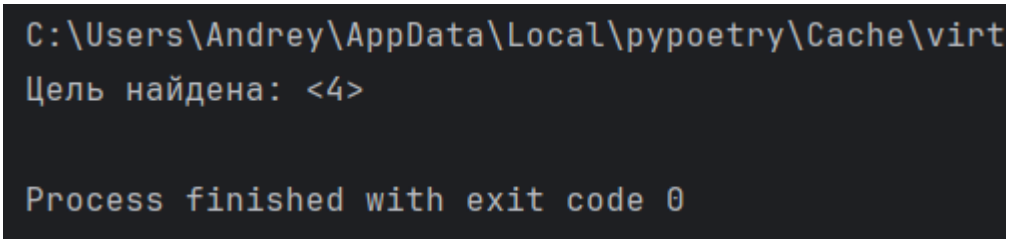

Рисунок 5 – Результаты работы программы поиска места в структуре склада

Система автоматического управления инвестициями

Представьте, что вы разрабатываете систему для автоматического управления инвестициями, где дерево решений используется для представления последовательности инвестиционных решений и их потенциальных исходов. Цель состоит в том, чтобы найти наилучший исход (максимальную прибыль) на определённой глубине принятия решений, учитывая ограниченные ресурсы и время на анализ.

Код программы поиска наилучшего исхода (максимальной прибыли) на определенной глубине:

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

"""
Представьте, что вы разрабатываете систему для автоматического управления
инвестициями, где дерево решений используется для представления
последовательности инвестиционных решений и их потенциальных исходов.
Цель состоит в том, чтобы найти наилучший исход (максимальную прибыль) на
определённой глубине принятия решений, учитывая ограниченные ресурсы и
время на анализ.
```

```
"""

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


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

    def __init__(self, initial=None):
        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


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 depth(self):
        """
        Глубина узла — расстояние от корня (считаем по цепочке parent).
        """
        if self.parent is None:
            return 0
        return 1 + self.parent.depth()


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


def expand(problem, node):
```

```
    """
    Генерация дочерних узлов.
    """

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


def depth_limited_search_max(problem, limit):
    """
    Функция ищет максимум среди значений узлов, не опускаясь глубже limit.
    Возвращает максимальную прибыль число (max_value).
    """

    if problem.initial is None:
        return None

    frontier = [Node(problem.initial)]
    max_value = -math.inf

    while frontier:
        node = frontier.pop()

        # Обновляем максимум
        if node.state.value > max_value:
            max_value = node.state.value

        # Ограничение глубины раскрытия
        if node.depth() < limit:
            # Генерирация дочерних узлов
            for child in expand(problem, node):
                frontier.append(child)

    # Если max_value остался -inf, значит дерево было пустое
    return None if max_value == -math.inf else max_value


class InvestmentProblem(Problem):
    """
    Реализация класса задачи управления складом
    Необходимо найти максимальную прибыль (value), не спускаясь глубже limit.
    """

    def actions(self, state):
        """
        Возвращает список дочерних узлов: left и right, если они есть.
        """
        children = []
        if state.left is not None:
            children.append(state.left)
        if state.right is not None:
            children.append(state.right)
        return children

    def result(self, state, action):
        """
        Переход: из state идём в action (его потомок).
        """
        return action


def main():
    """
```

```
    Главная функция программы.
    """

    root = BinaryTreeNode(
        3,
        left=BinaryTreeNode(1, BinaryTreeNode(0), None),
        right=BinaryTreeNode(5, BinaryTreeNode(4), BinaryTreeNode(6)),
    )
    limit = 2

    # Создаём задачу
    problem = InvestmentProblem(initial=root)

    max_val = depth_limited_search_max(problem, limit=limit)

    if max_val is None:
        print("Дерево пустое, нет максимального значения (максимальной
прибыли).")
    else:
        print("Максимальное значение на указанной глубине:", max_val)


if __name__ == "__main__":
    main()
```

Результаты работы программы, поиск максимального значения в бинарном дереве с ограничением глубины (рис. 6).

```
C:\Users\Andrey\AppData\Local\pypoetry\Cache\virtu
Максимальное значение на указанной глубине: 6

Process finished with exit code 0
```

Рисунок 6 – Результаты работы программы управления инвестициями

Ссылка на репозиторий данной лабораторной работы:

https://github.com/AndreyPust/Artificial_Intelligence_laboratory_work_4.git

Ответы на контрольные вопросы:

1. Что такое поиск с ограничением глубины, и как он решает проблему бесконечных ветвей?

Поиск с ограничением глубины (Depth-Limited Search, DLS) – это модификация поиска в глубину, которая ограничивает глубину рекурсии до заданного предела (limit). Узлы, которые находятся на уровне глубже заданного предела, не исследуются.

В случае бесконечных графов поиск в глубину может уйти в бесконечную рекурсию, так как он продолжает углубляться. Поиск с ограничением глубины останавливается, как только достигнута заданная глубина, предотвращая зацикливание.

2. Какова основная цель ограничения глубины в данном методе поиска?

Основная цель ограничения глубины – ограничить объем работы поиска, избегая бесконечных рекурсий или исследования ненужных частей дерева. Это особенно важно в графах с бесконечными или очень глубокими ветвями.

3. В чем разница между поиском в глубину и поиском с ограничением глубины?

Поиск в глубину углубляется до самого нижнего уровня дерева или графа, что может привести к зацикливанию в бесконечных графах.

Поиск с ограничением глубины ограничивает глубину поиска заданным значением limit, предотвращая исследование узлов, которые находятся глубже этого уровня.

4. Какую роль играет проверка глубины узла в псевдокоде поиска с ограничением глубины?

Проверка глубины узла определяет, следует ли продолжать исследование текущей ветви. Если глубина узла достигает значения limit, дальнейшее исследование прекращается, чтобы не нарушить ограничение.

5. Почему в случае достижения лимита глубины функция возвращает «обрезание»?

Когда достигается лимит глубины, алгоритм возвращает результат «обрезание» (cutoff), чтобы сигнализировать, что узел на этой ветви находится на пределе глубины и не может быть исследован дальше. Это помогает алгоритму сообщить, что в текущей ветви может находиться решение, но его невозможно проверить на данном уровне.

6. В каких случаях поиск с ограничением глубины может не найти решение, даже если оно существует?

Поиск с ограничением глубины не найдет решение, если:

– решение находится на глубине, превышающей заданный лимит;

– путь к решению слишком длинный, и алгоритм прекращает углубление до его нахождения.

7. Как поиск в ширину и в глубину отличаются при реализации с использованием очереди?

Поиск в ширину (BFS): использует очередь FIFO (first-in, first-out), добавляя узлы в конец очереди и извлекая из начала. Это обеспечивает уровень за уровнем исследование.

Поиск в глубину (DFS): использует стек LIFO (last-in, first-out), добавляя узлы в начало структуры данных, что позволяет углубляться по одной ветви.

8. Почему поиск с ограничением глубины не является оптимальным?

Поиск с ограничением глубины не гарантирует нахождение кратчайшего пути до цели, так как он прекращает исследование ветвей, которые превышают установленный лимит. Если решение находится на большой глубине, оно не будет найдено.

9. Как итеративное углубление улучшает стандартный поиск с ограничением глубины?

Итеративное углубление (Iterative Deepening Depth-First Search, IDDFS):

– выполняет поиск с ограничением глубины для увеличивающегося лимита;

– на каждом шаге лимит увеличивается на единицу;

– это позволяет находить решения на минимальной глубине, подобно поиску в ширину, при этом используя меньшую память (как в поиске в глубину).

10. В каких случаях итеративное углубление становится эффективнее простого поиска в ширину?

Итеративное углубление становится эффективнее, когда:

– пространство состояний очень большое или бесконечное;

– глубина целевого узла мала по сравнению с размером пространства состояний;

– ограничение памяти является критическим фактором, так как итеративное углубление использует память, эквивалентную поиску в глубину (не хранит все узлы).

11. Какова основная цель использования алгоритма поиска с ограничением глубины?

Алгоритм поиска с ограничением глубины предотвращает зацикливание в бесконечных пространствах состояний, ограничивая глубину поиска заданным параметром limit. Это помогает эффективно исследовать пространство состояний до фиксированной глубины.

12. Какие параметры принимает функция depth_limited_search, и каково их назначение?

Функция обычно принимает следующие параметры:

– problem: описание задачи, содержащей начальное состояние, операторы, функции проверки цели и т.д;

– limit: максимальная глубина поиска, которая предотвращает зацикливание и неконтролируемое углубление.

13. Какое значение по умолчанию имеет параметр limit в функции depth_limited_search?

Значение по умолчанию зависит от реализации. Часто значение не задается явно, и пользователь обязан указать его, или же используется большой фиксированный предел.

14. Что представляет собой переменная frontier, и как она используется в алгоритме?

frontier представляет собой структуру данных, хранящую узлы, которые нужно исследовать. В поиске с ограничением глубины это LIFO-очередь (стек), которая обеспечивает поведение поиска в глубину.

15. Какую структуру данных представляет LIFOQueue, и почему она используется в этом алгоритме?

LIFOQueue – это стек, реализованный на базе принципа "последним вошел – первым вышел". Он используется, чтобы исследовать узлы в порядке

обратного хода (углубляться в дочерние узлы перед возвратом к родительским).

16. Каково значение переменной result при инициализации, и что оно означает?

Обычно result инициализируется как None, failure или другое значение, указывающее, что целевой узел пока не найден.

17. Какое условие завершает цикл while в алгоритме поиска?

Цикл завершается, когда:

– frontier становится пустым (все возможные узлы исследованы);

– целевой узел найден.

18. Какой узел извлекается с помощью frontier.pop() и почему?

С помощью frontier.pop() извлекается последний добавленный узел (верхний элемент стека). Это обеспечивает углубление поиска, следуя стратегии "глубина сначала".

19. Что происходит, если найден узел, удовлетворяющий условию цели (условие problem.is_goal(node.state))?

Алгоритм немедленно завершает работу и возвращает найденный узел как решение.

20. Какую проверку выполняет условие elif len(node) >= limit, и что означает его выполнение?

Условие проверяет, достиг ли текущий узел максимальной глубины, определенной параметром limit. Если достиг, узел больше не расширяется, чтобы предотвратить зацикливание или избыточное углубление.

21. Что произойдет, если текущий узел достигнет ограничения по глубине поиска?

Алгоритм прекращает расширение этого узла. Обычно возвращается результат cutoff, чтобы показать, что поиск был прерван из-за ограничения глубины.

22. Какую роль выполняет проверка на циклы elif not is_cycle(node) в алгоритме?

Она предотвращает повторное исследование уже пройденных узлов в текущем пути, исключая зацикливание.

23. Что происходит с дочерними узлами, полученными с помощью функции expand(problem, node)?

Дочерние узлы добавляются в frontier для дальнейшего исследования, если они соответствуют условиям (например, не достигли предела глубины и не образуют цикл).

24. Какое значение возвращается функцией, если целевой узел не был найден?

Если целевой узел не найден, возвращается failure, что означает отсутствие решения в рамках заданного ограничения глубины.

25. В чем разница между результатами failure и cutoff в контексте данного алгоритма?

– failure: целевой узел не найден, и поиск завершен;

– cutoff: поиск был прерван из-за достижения ограничения глубины, возможно, целевой узел находится на большей глубине.

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