

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»
Вариант 1

Выполнил:
Бабенко Артём Тимофеевич
3 курс, группа ИВТ-б-о-22-1,
09.03.01 «Информатика и
вычислительная техника»,
направленность (профиль)
«Программное обеспечение средств
вычислительной техники и
автоматизированных систем», очная
форма обучения

(подпись)

Проверил:
Ассистент департамента цифровых,
робототехнических систем и
электроники Богданов С.С

(подпись)

Отчет защищен с оценкой _____ Дата защиты _____

Ставрополь, 2024 г.

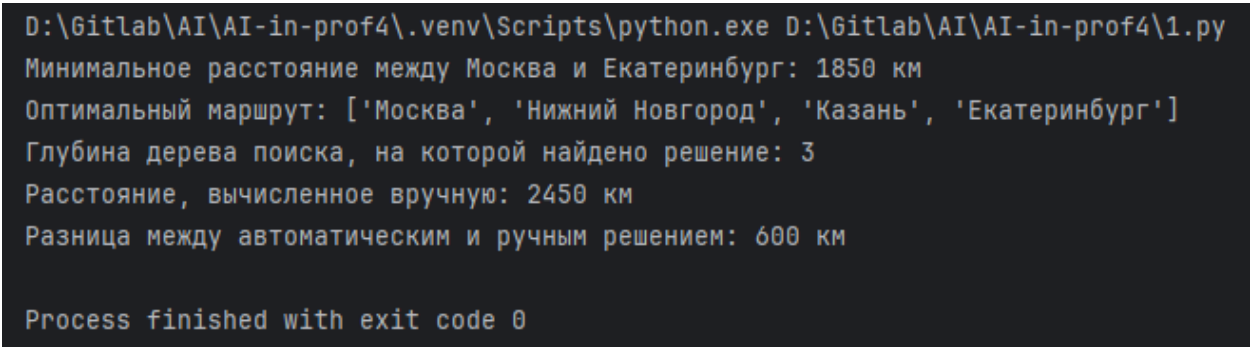
Тема: Исследование поиска с ограничением глубины

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

Порядок выполнения работы:

Задание 1. Для построенного графа лабораторной работы 1 (имя файла начинается с PR.AI.001.) напишите программу на языке программирования Python, которая с помощью алгоритма поиска с ограничением глубины находит минимальное расстояние между начальным и конечным пунктами. Определите глубину дерева поиска, на которой будет найдено решение. Сравните найденное решение с решением, полученным вручную.

Результат работы программы:



```
D:\6itlab\AI\AI-in-prof4\.venv\Scripts\python.exe D:\6itlab\AI\AI-in-prof4\1.py
Минимальное расстояние между Москва и Екатеринбург: 1850 км
Оптимальный маршрут: ['Москва', 'Нижний Новгород', 'Казань', 'Екатеринбург']
Глубина дерева поиска, на которой найдено решение: 3
Расстояние, вычисленное вручную: 2450 км
Разница между автоматическим и ручным решением: 600 км

Process finished with exit code 0
```

Рисунок 1 – Результат работы программы

Код программы:

```
import networkx as nx
import matplotlib.pyplot as plt
```

```
# Шаг 1: Создание списка населённых пунктов и расстояний между ними
places = [
    "Москва", "Санкт-Петербург", "Новосибирск", "Екатеринбург", "Казань",
    "Нижний Новгород", "Челябинск", "Омск", "Самара", "Ростов-на-Дону",
    "Уфа", "Красноярск", "Воронеж", "Пермь", "Волгоград", "Краснодар",
    "Саратов", "Тюмень", "Тольятти", "Ижевск", "Барнаул"
]
```

```
# Матрица расстояний
distances = {
    ("Москва", "Санкт-Петербург"): 650,
    ("Москва", "Казань"): 800,
    ("Москва", "Нижний Новгород"): 450,
    ("Санкт-Петербург", "Новосибирск"): 3200,
    ("Санкт-Петербург", "Казань"): 1000,
    ("Новосибирск", "Екатеринбург"): 1700,
```

```

("Екатеринбург", "Казань"): 1200,
("Екатеринбург", "Челябинск"): 300,
("Екатеринбург", "Омск"): 900,
("Казань", "Нижний Новгород"): 200,
("Казань", "Самара"): 600,
("Нижний Новгород", "Челябинск"): 1500,
("Челябинск", "Омск"): 800,
("Омск", "Красноярск"): 1200,
("Самара", "Ростов-на-Дону"): 1100,
("Самара", "Уфа"): 400,
("Ростов-на-Дону", "Краснодар"): 400,
("Уфа", "Красноярск"): 2000,
("Уфа", "Пермь"): 500,
("Уфа", "Екатеринбург"): 500,
("Казань", "Екатеринбург"): 1200,
("Красноярск", "Воронеж"): 3000,
("Пермь", "Волгоград"): 1000,
("Волгоград", "Краснодар"): 500,
("Краснодар", "Саратов"): 700,
("Саратов", "Тюмень"): 1200,
("Тюмень", "Тольятти"): 1300,
("Тольятти", "Ижевск"): 400,
("Ижевск", "Барнаул"): 2000,
}

```

Шаг 2: Создание графа

```
G = nx.Graph()
```

Добавление узлов (населённых пунктов)

```
G.add_nodes_from(places)
```

Добавление рёбер с весами (расстояниями)

```
for (place1, place2), distance in distances.items():
```

```
    G.add_edge(place1, place2, weight=distance)
```

Шаг 3: Алгоритм поиска с ограничением глубины (DLS)

```
def depth_limited_search(graph, start, end, max_depth):
```

```
    # Проверка, существуют ли стартовый и конечный узлы в графе
```

```
    if not graph.has_node(start) or not graph.has_node(end):
```

```
        return None, float('inf'), -1
```

```
    # Инициализация переменных
```

```
    min_distance = float('inf')
```

```
    best_path = None
```

```
    solution_depth = -1
```

```
def dls(current_node, current_path, current_distance, current_depth):
```

```
    nonlocal min_distance, best_path, solution_depth
```

```
    # Если достигли целевой узел
```

```
    if current_node == end:
```

```
        if current_distance < min_distance:
```

```

        min_distance = current_distance
        best_path = current_path[:]
        solution_depth = current_depth
    return

# Если достигнута максимальная глубина
if current_depth >= max_depth:
    return

# Исследуем соседей
for neighbor in graph.neighbors(current_node):
    if neighbor not in current_path: # Избегаем циклов
        edge_weight = graph[current_node][neighbor]['weight']
        current_path.append(neighbor)
        dls(neighbor, current_path, current_distance + edge_weight, current_depth + 1)
        current_path.pop() # Возвращаемся назад (backtracking)

# Запуск DLS
dls(start, [start], 0, 0)

return best_path, min_distance, solution_depth

# Шаг 4: Выбор начального и конечного пунктов
start_city = "Москва"
end_city = "Екатеринбург"

# Ограничение глубины поиска
max_depth = 5 # Максимальная глубина дерева поиска

# Поиск минимального расстояния с помощью DLS
best_path, min_distance, solution_depth = depth_limited_search(G, start_city, end_city,
max_depth)

# Вывод результатов
if best_path is None:
    print(f"Путь между {start_city} и {end_city} не найден на глубине {max_depth}.")
else:
    print(f"Минимальное расстояние между {start_city} и {end_city}: {min_distance} км")
    print(f"Оптимальный маршрут: {best_path}")
    print(f"Глубина дерева поиска, на которой найдено решение: {solution_depth}")

manual_distance = 2450
print(f"Расстояние, вычисленное вручную: {manual_distance} км")
print(f"Разница между автоматическим и ручным решением: {abs(min_distance -
manual_distance)} км")

# Визуализация графа и маршрута
pos = nx.spring_layout(G, seed=42)
nx.draw(G, pos, with_labels=True, node_size=500, node_color="lightblue", font_size=10,
font_weight="bold")

# Выделение маршрута (если он найден)

```

```

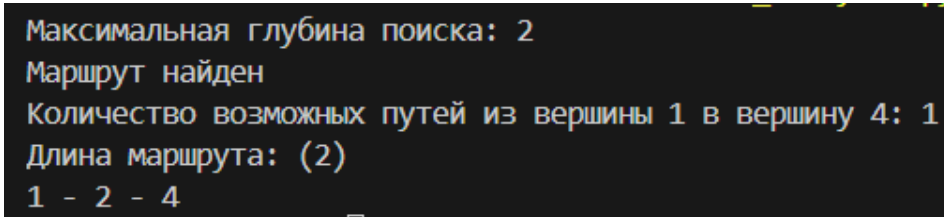
if best_path:
    route_edges = [(best_path[i], best_path[i + 1]) for i in range(len(best_path) - 1)]
    nx.draw_networkx_edges(G, pos, edgelist=route_edges, edge_color='red', width=2)

plt.title("Граф дорог между населёнными пунктами")
plt.show()

```

Задание 2. Система навигации робота-пылесоса.

Результат работы программы:



```

Максимальная глубина поиска: 2
Маршрут найден
Количество возможных путей из вершины 1 в вершину 4: 1
Длина маршрута: (2)
1 - 2 - 4

```

Рисунок 2 – Результат работы программы

Код программы:

```

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

from collections import defaultdict, deque, Counter
from itertools import combinations
from collections import deque

class Node:
    "Узел в дереве поиска"
    def __init__(self, state, parent=None, action=None, path_cost=0.0):
        self.__dict__.update(state=state, parent=parent, action=action,
                               path_cost=path_cost)
    def __repr__(self): return '<{}>'.format(self.state)
    def __len__(self): return 0 if self.parent is None else (1 + len(self.parent))
    def __lt__(self, other): return self.path_cost < other.path_cost
    def cost_calc(self, matrix):

        return matrix[self.action[len(self.action)-2]][self.action[len(self.action)-1]]

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}>'

def expand(node, max_depth = 10**27, finish = 5):
    "Раскрываем узел, создав дочерние узлы."
    s = node.state
    last_node = node.action
    #print(last_node)

```

```
#print(tree[int(last_node)])
if node.path_cost <= max_depth:
    lst_nodes = [n for n in [last_node.left,last_node.right] if n != None]
    result = []
    for item in lst_nodes:
        temp = s.copy()
        if item not in s:
            temp.append(item)
            dist = node.path_cost + 1
            result.append(Node(temp,node, item, dist))

    return result
else:
    return []
if __name__ == '__main__':
    root = BinaryTreeNode(
        1,
        BinaryTreeNode(2, None, BinaryTreeNode(4)),
        BinaryTreeNode(3, BinaryTreeNode(5), None),
    )
    start = root.value
    max_depth = 2
    finish = 4
    print(f"Максимальная глубина поиска: {max_depth}")
    first = Node([root], None, root, 0)
    paths = []
    q = deque()
    q.appendleft(first)
    counter = 0
    while q:
        counter+=1
        temp = expand(q.popleft(),max_depth)
        for i in temp:
            if i.action.value != finish:
                q.appendleft(i)
                #print(counter, i.state)
            else:
                paths.append(i)
    counter = 0
    if len(paths) == 0:
        print("Маршрут не найден")
    else:
        print("Маршрут найден")
        mn = [1000000000000000000000000, []]
        result = []
        for i in paths:
            if i.path_cost < mn[0]:
                mn[0] = i.path_cost
                mn[1] = i.state
                #print(i.state,i.path_cost,counter)
                counter+=1
```

```

print(f"Количество возможных путей из вершины {start} в вершину {finish}:
{len(paths)}")
print(f"Длина маршрута: ({mn[0]})")
for id in mn[1]:
    if id.value!= finish:
        print(id.value,end=" - ")
    else:
        print(id.value)

```

Задание 3. Система управления складом

Результат работы программы:

```

Максимальная глубина поиска: 2
Цель найдена: 4
Количество возможных путей из вершины 1 в вершину 4: 1
Длина маршрута: 2
Маршрут:
1 - 2 - 4

```

Рисунок 3 – Результат работы программы

Код программы:

```

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

from collections import defaultdict, deque, Counter
from itertools import combinations
from collections import deque

class Node:
    "Узел в дереве поиска"
    def __init__(self, state, parent=None, action=None, path_cost=0.0):
        self.__dict__.update(state=state, parent=parent, action=action,
                               path_cost=path_cost)
    def __repr__(self): return '<{}>'.format(self.state)
    def __len__(self): return 0 if self.parent is None else (1 + len(self.parent))
    def __lt__(self, other): return self.path_cost < other.path_cost
    def cost_calc(self, matrix):

        return matrix[self.action[len(self.action)-2]][self.action[len(self.action)-1]]

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}>"

def expand(node, max_depth = 10**27, finish = 5):

```

```

"Раскрываем узел, создав дочерние узлы."
s = node.state
last_node = node.action
#print(last_node)
#print(tree[int(last_node)])
if node.path_cost <= max_depth:
    lst_nodes = [n for n in [last_node.left, last_node.right] if n != None]
    result = []
    for item in lst_nodes:
        temp = s.copy()
        if item not in s:
            temp.append(item)
            dist = node.path_cost + 1
            result.append(Node(temp, node, item, dist))

    return result
else:
    return []
if __name__ == '__main__':
    root = BinaryTreeNode(
        1,
        BinaryTreeNode(2, None, BinaryTreeNode(4)),
        BinaryTreeNode(3, BinaryTreeNode(5), None),
    )
    start = root.value
    max_depth = 2
    finish = 4
    print(f"Максимальная глубина поиска: {max_depth}")
    first = Node([root], None, root, 0)
    paths = []
    q = deque()
    q.appendleft(first)
    counter = 0
    while q:
        counter += 1
        temp = expand(q.popleft(), max_depth)
        for i in temp:
            if i.action.value != finish:
                q.appendleft(i)
                #print(counter, i.state)
            else:
                paths.append(i)
    counter = 0
    if len(paths) == 0:
        print("Цель не найдена")
    else:
        print(f"Цель найдена: {finish}")
        mn = [10000000000000000000, []]
        result = []
        for i in paths:
            if i.path_cost < mn[0]:
                mn[0] = i.path_cost

```



```

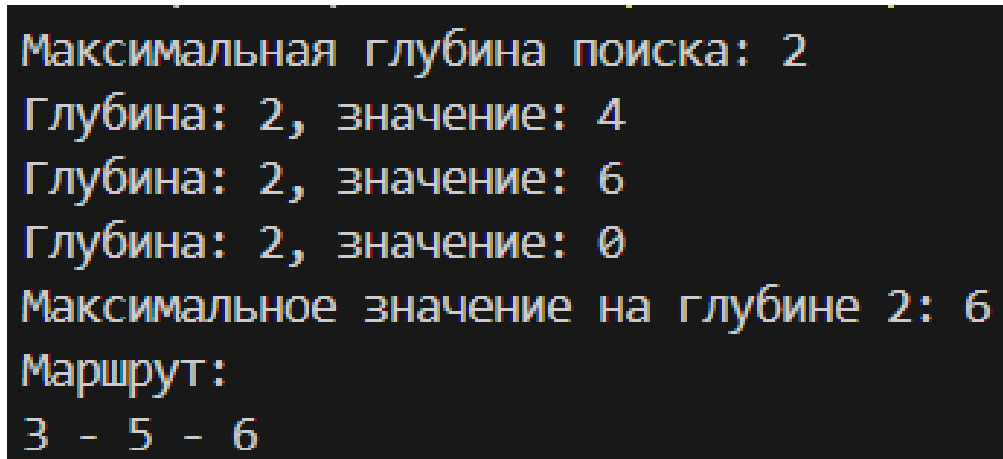
mn[1] = i.state
#print(i.state,i.path_cost,counter)
counter+=1

print(f'Количество возможных путей из вершины {start} в вершину {finish}:
{len(paths)}')
print(f'Длина маршрута: {mn[0]}')
print(f'Маршрут:')
for id in mn[1]:
    if id.value!= finish:
        print(id.value,end=" - ")
    else:
        print(id.value)

```

Задание 4. Система автоматического управления инвестициями.

Результат работы программы:



```

Максимальная глубина поиска: 2
Глубина: 2, значение: 4
Глубина: 2, значение: 6
Глубина: 2, значение: 0
Максимальное значение на глубине 2: 6
Маршрут:
3 - 5 - 6

```

Рисунок 4 – Результат работы программы

Код программы:

```

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

from collections import defaultdict, deque, Counter
from itertools import combinations
from collections import deque

class Node:
    "Узел в дереве поиска"
    def __init__(self, state, parent=None, action=None, path_cost=0.0):
        self.__dict__.update(state=state, parent=parent, action=action,
                               path_cost=path_cost)
    def __repr__(self): return '<{}>'.format(self.state)
    def __len__(self): return 0 if self.parent is None else (1 + len(self.parent))
    def __lt__(self, other): return self.path_cost < other.path_cost

```

```

def cost_calc(self, matrix):

    return matrix[self.action[len(self.action)-2]][self.action[len(self.action)-1]]

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}>"

def expand(node, max_depth = 10**27, finish = 5):
    "Раскрываем узел, создав дочерние узлы."
    s = node.state
    last_node = node.action
    #print(last_node)
    #print(tree[int(last_node)])
    if node.path_cost <= max_depth:
        lst_nodes = [n for n in [last_node.left, last_node.right] if n != None]
        result = []
        for item in lst_nodes:
            temp = s.copy()
            if item not in s:
                temp.append(item)
                dist = node.path_cost + 1
                result.append(Node(temp, node, item, dist))

        return result
    else:
        return []

if __name__ == '__main__':
    root = BinaryTreeNode(
        3,
        BinaryTreeNode(1, BinaryTreeNode(0), None),
        BinaryTreeNode(5, BinaryTreeNode(4), BinaryTreeNode(6)),
    )
    start = root.value
    max_depth = 2
    print(f"Максимальная глубина поиска: {max_depth}")
    first = Node([root], None, root, 0)
    paths = []
    q = deque()
    q.appendleft(first)
    counter = 0
    while q:
        counter+=1
        temp = expand(q.popleft(), max_depth)
        for i in temp:
            if i.path_cost != max_depth:
                q.appendleft(i)
                #print(counter, i.state)

```

Отвѣты на контрольные вопросы:

При таком поиске максимальная длина ветви ограничена, что делает бесконечный цикл невозможным.

Целью является решение проблемы бесконечных ветвей.

Поиск с ограничением глубины ищет до определенной глубины, а потом останавливается.

Это позволяет определить, достигнут ли предел глубины.

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

Чтобы ограничить глубину поиска.

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

Если оно расположено на большей глубине, чем задана в поиске.

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

При поиске в ширину используется очередь, а при поиске в глубину используется стек.

8. Почему поиск с ограничением глубины не является оптимальным?

Данный поиск может не найти решение, которое расположено на большей глубине, чем задана в алгоритме поиска.

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

Итеративное углубление позволяет найти нужный элемент и оптимизировать расход памяти.

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

Итеративное углубление эффективнее поиска в ширину в случаях, когда коэффициент ветвления относительно высокий.

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

Решение проблемы бесконечной глубины.

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

Данная функция принимает параметры `problem` и `limit`. Первый параметр представляет из себя задачу, которую нужно решить, а второй отвечает за глубину поиска.

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

Данный параметр по умолчанию имеет значение 10.

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

Переменная `frontier` представляет собой стек (LIFO очередь), содержащий начальный узел.

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

Данная структура данных представляет из себя стек, такой тип данных используется в поиске в глубину.

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

Переменная `result` будет хранить результат поиска. Изначально она установлена в значение `failure` , что означает неудачу поиска.

17. Какое условие завершает цикл `while` в алгоритме поиска?

Цикл выполняется до тех пор, пока `frontier` не станет пустым, то есть пока есть узлы для рассмотрения.

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

Так как данный алгоритм основан на поиске в глубину, извлекается последний добавленный узел из `frontier` для дальнейшей обработки.

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

Поиск завершается успешно.

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

Проверяется, достиг ли текущий узел ограничения по глубине. Если да, то дальнейший поиск в этом направлении прекращается.

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

Если текущий узел достиг ограничения по глубине, переменной `result` присваивается значение `cutoff` , что означает достижение лимита глубины поиска.

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

Проверяется, не ведет ли текущий узел к циклу. Если нет, то можно продолжать поиск.

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

Каждый дочерний узел добавляется в `frontier` для дальнейшей обработки.

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

Возвращается значение `failure`.

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

`failure` означает неудачу поиска, а `cutoff` возвращается в случае достижения максимальной глубины и отсутствия результата.

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