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 №3**

**дисциплины «Искусственный интеллект в профессиональной сфере»**

|  |  |  |  |
| --- | --- | --- | --- |
|  | | Выполнил:  Кожуховский Виктор Андреевич  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:  Воронкин Роман Александрович  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись) | |
|  | |  | |

Отчет защищен с оценкой \_\_\_\_\_\_\_\_\_\_\_ Дата защиты\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Ставрополь, 2024 г.

Тема: Исследование поиска в глубину

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

Порядок выполнения работы:

1. Изучил теоретический материал работы.

2. Создал общедоступный репозиторий на GitHub, в котором использована лицензия MIT и язык программирования Python.

3. Выполнил клонирование созданного репозитория.

4. Дополнил файл .gitignore необходимыми правилами для работы с IDE.

5. Организовал свой репозиторий в соответствие с моделью ветвления git-flow.

6. Создал проект в папке репозитория.

7. Проработал примеры лабораторной работы.

8. Решите задания лабораторной работы с помощью языка программирования Python и элементов программного кода лабораторной работы 1 (имя файла начинается с PR.AI.001.). Проверьте правильность решения каждой задачи на приведенных тестовых примерах.

Flood fill (также известный как seed fill) - это алгоритм, определяющий область, связанную с заданным узлом в многомерном массиве.

Он используется в инструменте заливки "ведро" в программе рисования для заполнения соединенных одинаково окрашенных областей другим цветом, а также в таких играх, как Go и Minesweeper, для определения того, какие фигуры очищены. Когда заливка применяется на изображении для заполнения цветом определенной ограниченной области, она также известна как заливка границ.

Алгоритм заливки принимает три параметра: начальный узел, целевой цвет и цвет замены.

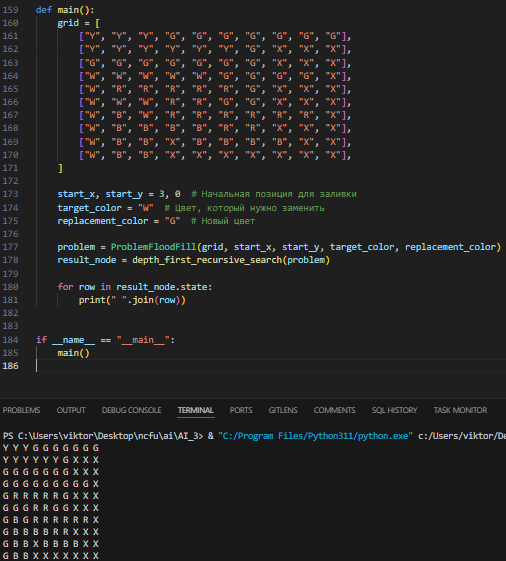


Рисунок 1. Решение задачи Flood fill

#!/usr/bin/env python3

# -\*- coding: utf-8 -\*-

import math

from collections import deque

class Problem:

def \_\_init\_\_(self, initial=None, goal=None, \*\*kwds):

self.\_\_dict\_\_.update(initial=initial, goal=goal, \*\*kwds)

def actions(self, state):

raise NotImplementedError

def result(self, state, action):

raise NotImplementedError

def is\_goal(self, state):

return state == self.goal

def action\_cost(self, s, a, s1):

return 1

def h(self, node):

return 0

def \_\_str\_\_(self):

return "{}({!r}, {!r})".format(type(self).\_\_name\_\_, self.initial, self.goal)

class Node:

def \_\_init\_\_(self, state, parent=None, action=None, path\_cost=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

@staticmethod

def is\_cycle(node):

parent = node.parent

while parent:

if parent.state == node.state:

return True

parent = parent.parent

return False

failure = None

cutoff = None

@staticmethod

def expand(problem, node):

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(s1, node, action, cost)

Node.failure = Node("failure", path\_cost=math.inf)

Node.cutoff = Node("cutoff", path\_cost=math.inf)

def depth\_first\_recursive\_search(problem, node=None):

if node is None:

node = Node(problem.initial)

if problem.is\_goal(node.state):

return node

elif Node.is\_cycle(node):

return Node.failure

else:

for child in Node.expand(problem, node):

result = depth\_first\_recursive\_search(problem, child)

if result is not Node.failure:

return result

return Node.failure

class ProblemFloodFill(Problem):

def \_\_init\_\_(self, grid, start\_x, start\_y, target\_color, replacement\_color):

initial = self.find\_initial\_state(grid)

goal = None

super().\_\_init\_\_(

initial=initial,

goal=goal,

grid=grid,

start\_x=start\_x,

start\_y=start\_y,

target\_color=target\_color,

replacement\_color=replacement\_color,

)

def find\_initial\_state(self, grid):

return tuple(tuple(row) for row in grid)

def actions(self, state):

return [

(dx, dy)

for dx, dy in (

(1, 0),

(-1, 0),

(0, 1),

(0, -1),

)

]

def result(self, state, action):

grid = [list(row) for row in state]

x, y = self.start\_x, self.start\_y

target\_color = grid[x][y]

if target\_color == self.replacement\_color:

return state

self.flood\_fill(grid, x, y, target\_color)

return tuple(tuple(row) for row in grid)

def flood\_fill(self, grid, x, y, target\_color):

queue = deque([(x, y)])

visited = set()

while queue:

x, y = queue.popleft()

if (

(x, y) in visited

or x < 0

or x >= len(grid)

or y < 0

or y >= len(grid[0])

or grid[x][y] != target\_color

):

continue

grid[x][y] = self.replacement\_color

visited.add((x, y))

for dx, dy in self.actions(grid):

nx, ny = x + dx, y + dy

queue.append((nx, ny))

def is\_goal(self, state):

for row in state:

for cell in row:

if cell == self.target\_color:

return False

return True

def main():

grid = [

["Y", "Y", "Y", "G", "G", "G", "G", "G", "G", "G"],

["Y", "Y", "Y", "Y", "Y", "Y", "G", "X", "X", "X"],

["G", "G", "G", "G", "G", "G", "G", "X", "X", "X"],

["W", "W", "W", "W", "W", "G", "G", "G", "G", "X"],

["W", "R", "R", "R", "R", "R", "G", "X", "X", "X"],

["W", "W", "W", "R", "R", "G", "G", "X", "X", "X"],

["W", "B", "W", "R", "R", "R", "R", "R", "R", "X"],

["W", "B", "B", "B", "B", "R", "R", "X", "X", "X"],

["W", "B", "B", "X", "B", "B", "B", "B", "X", "X"],

["W", "B", "B", "X", "X", "X", "X", "X", "X", "X"],

]

start\_x, start\_y = 3, 0 # Начальная позиция для заливки

target\_color = "W" # Цвет, который нужно заменить

replacement\_color = "G" # Новый цвет

problem = ProblemFloodFill(grid, start\_x, start\_y, target\_color, replacement\_color)

result\_node = depth\_first\_recursive\_search(problem)

for row in result\_node.state:

print(" ".join(row))

if \_\_name\_\_ == "\_\_main\_\_":

main()

Дана матрица символов размером M×N. Необходимо найти длину самого длинного пути в матрице, начиная с заданного символа. Каждый следующий символ в пути должен алфавитно следовать за предыдущим без пропусков. Разработать функцию поиска самого длинного пути в матрице символов, начиная с заданного символа. Символы в пути должны следовать в алфавитном порядке и быть последовательными. Поиск возможен во всех восьми направлениях.

Для задачи "Поиск самого длинного пути в матрице" подготовить собственную матрицу, для которой с помощью разработанной в предыдущем пункте программы, подсчитать самый длинный путь.

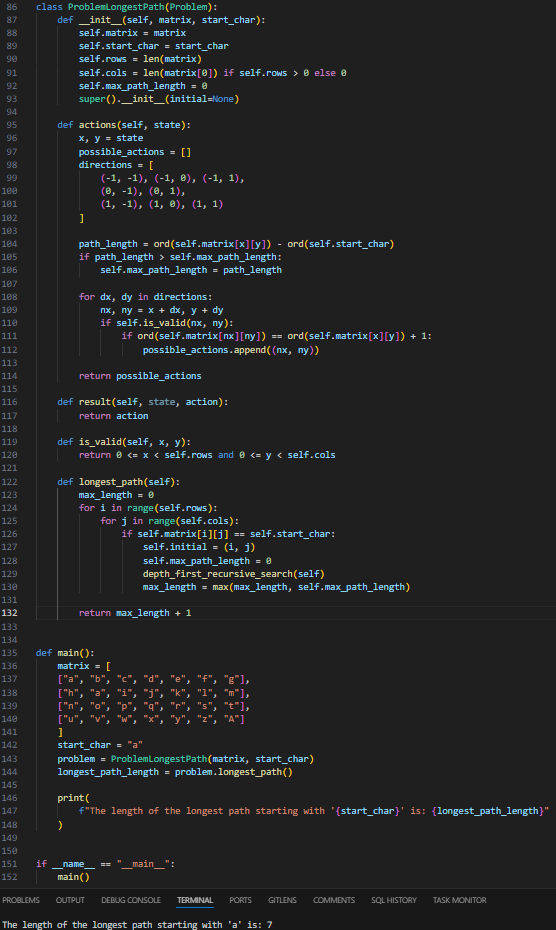


Рисунок 2. Решение задачи поиск самого длинного пути в матрице

class ProblemLongestPath(Problem):

def \_\_init\_\_(self, matrix, start\_char):

self.matrix = matrix

self.start\_char = start\_char

self.rows = len(matrix)

self.cols = len(matrix[0]) if self.rows > 0 else 0

self.max\_path\_length = 0

super().\_\_init\_\_(initial=None)

def actions(self, state):

x, y = state

possible\_actions = []

directions = [

(-1, -1), (-1, 0), (-1, 1),

(0, -1), (0, 1),

(1, -1), (1, 0), (1, 1)

]

path\_length = ord(self.matrix[x][y]) - ord(self.start\_char)

if path\_length > self.max\_path\_length:

self.max\_path\_length = path\_length

for dx, dy in directions:

nx, ny = x + dx, y + dy

if self.is\_valid(nx, ny):

if ord(self.matrix[nx][ny]) == ord(self.matrix[x][y]) + 1:

possible\_actions.append((nx, ny))

return possible\_actions

def result(self, state, action):

return action

def is\_valid(self, x, y):

return 0 <= x < self.rows and 0 <= y < self.cols

def longest\_path(self):

max\_length = 0

for i in range(self.rows):

for j in range(self.cols):

if self.matrix[i][j] == self.start\_char:

self.initial = (i, j)

self.max\_path\_length = 0

depth\_first\_recursive\_search(self)

max\_length = max(max\_length, self.max\_path\_length)

return max\_length + 1

def main():

matrix = [

["a", "b", "c", "d", "e", "f", "g"],

["h", "a", "i", "j", "k", "l", "m"],

["n", "o", "p", "q", "r", "s", "t"],

["u", "v", "w", "x", "y", "z", "A"]

]

start\_char = "a"

problem = ProblemLongestPath(matrix, start\_char)

longest\_path\_length = problem.longest\_path()

print(

f"The length of the longest path starting with '{start\_char}' is: {longest\_path\_length}"

)

if \_\_name\_\_ == "\_\_main\_\_":

main()

Вам дана матрица символов размером M × N. Ваша задача — найти и вывести список всех возможных слов, которые могут быть сформированы из последовательности соседних символов в этой матрице. При этом слово может формироваться во всех восьми возможных направлениях (север, юг, восток, запад, северо-восток, северо-запад, юго-восток, юго-запад), и каждая клетка может быть использована в слове только один раз.

Для задачи "Генерирование списка возможных слов из матрицы символов" подготовить собственную матрицу для генерирования списка возможных слов с помощью разработанной программы.

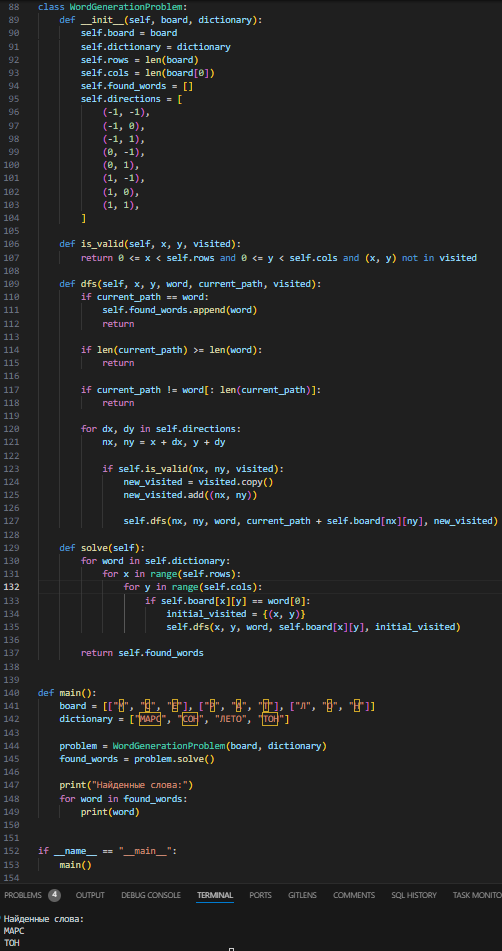


Рисунок 3. Решение задачи генерирование списка возможных слов из матрицы символов

10. Для построенного графа лабораторной работы 1 (имя файла начинается с PR.AI.001.) напишите программу на языке программирования Python, которая с помощью алгоритма поиска в глубину находит минимальное расстояние между начальным и конечным пунктами.



Рисунок 4. Код программы решения задания и его выполнение

#!/usr/bin/env python3

# -\*- coding: utf-8 -\*-

import math

import json

from collections import deque

class Problem:

def \_\_init\_\_(self, initial=None, goal=None, \*\*kwds):

self.\_\_dict\_\_.update(initial=initial, goal=goal, \*\*kwds)

def actions(self, state):

raise NotImplementedError

def result(self, state, action):

raise NotImplementedError

def is\_goal(self, state):

return state == self.goal

def action\_cost(self, s, a, s1):

return 1

def h(self, node):

return 0

def \_\_str\_\_(self):

return "{}({!r}, {!r})".format(type(self).\_\_name\_\_, self.initial, self.goal)

class Node:

def \_\_init\_\_(self, state, parent=None, action=None, path\_cost=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

@staticmethod

def is\_cycle(node):

parent = node.parent

while parent:

if parent.state == node.state:

return True

parent = parent.parent

return False

failure = None

cutoff = None

@staticmethod

def expand(problem, node):

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(s1, node, action, cost)

Node.failure = Node("failure", path\_cost=math.inf)

Node.cutoff = Node("cutoff", path\_cost=math.inf)

class CityProblem(Problem):

def \_\_init\_\_(self, cities, distances, initial, goal):

super().\_\_init\_\_(initial=initial, goal=goal)

self.cities = cities

self.distances = distances

def actions(self, state):

return [target for target in self.cities if (state, target) in self.distances]

def result(self, state, action):

return action

def action\_cost(self, s, a, s1):

return self.distances.get((s, s1), 1)

def depth\_first\_recursive\_search(problem, node=None):

if node is None:

node = Node(problem.initial)

if problem.is\_goal(node.state):

return node

elif Node.is\_cycle(node):

return Node.failure

else:

for child in Node.expand(problem, node):

result = depth\_first\_recursive\_search(problem, child)

if result is not Node.failure:

return result

return Node.failure

def reconstruct\_path(came\_from, current):

path = []

while current is not None:

path.append(current)

current = came\_from[current]

path.reverse()

return path

if \_\_name\_\_ == "\_\_main\_\_":

with open("elem.json", "r", encoding="utf-8") as file:

data = json.load(file)

selected\_ids = {"8", "9", "2", "15", "6", "1", "3", "7", "13", "18"}

cities = {}

distances = {}

for item in data:

if "label" in item["data"]:

if item["data"]["id"] in selected\_ids:

cities[item["data"]["id"]] = item["data"]["label"]

elif "source" in item["data"]:

source = item["data"]["source"]

target = item["data"]["target"]

if source in selected\_ids and target in selected\_ids:

weight = item["data"]["weight"]

distances[(source, target)] = weight

distances[(target, source)] = weight

start\_city = "8"

goal\_city = "15"

problem = CityProblem(cities, distances, start\_city, goal\_city)

solution = depth\_first\_recursive\_search(problem)

if solution is None:

print("Решение не найдено.")

else:

path = []

current = solution

while current is not None:

path.append(current.state)

current = current.parent

path.reverse()

print("Кратчайший путь:")

print(" -> ".join(cities[city] for city in path))

11. Зафиксировал сделанные изменения в репозитории.

12. Выполнил слияние ветки для разработки с веткой master/main.

13. Отправил сделанные изменения на сервер GitHub.

Ссылка: https://github.com/Viktorkozh/AI\_3

Контрольные вопросы:

1. В чем ключевое отличие поиска в глубину от поиска в ширину?

Поиск в глубину расширяет самый глубокий из нерасширенных узлов, в то время как поиск в ширину возвращается к узлам на том же уровне.

2. Какие четыре критерия качества поиска обсуждаются в тексте для оценки алгоритмов?

Временная сложность, пространственная сложность, оптимальность и полнота.

3. Что происходит при расширении узла в поиске в глубину?

При расширении узла в поиске в глубину узел становится серым, а его дочерние узлы добавляются в список на рассмотрение.

4. Почему поиск в глубину использует очередь типа "последним пришел — первым ушел" (LIFO)?

Он расширяет самый глубокий из нерасширенных узлов.

5. Как поиск в глубину справляется с удалением узлов из памяти, и почему это преимущество перед поиском в ширину?

Поиск в глубину справляется с удалением узлов из памяти, освобождая узлы, когда достигает конца ветви и не находит цель. Это преимущество перед поиском в ширину, который не может освобождать узлы, так как ему нужно хранить информацию обо всех узлах на текущем уровне.

6. Какие узлы остаются в памяти после того, как достигнута максимальная глубина дерева?

В памяти после достижения максимальной глубины дерева остаются только узлы, которые находятся на пути от корня до текущего узла, а также узлы, которые еще предстоит исследовать.

7. В каких случаях поиск в глубину может "застрять" и не найти решение?

Поиск в глубину может "застрять" в бесконечной ветви, не рассматривая другие потенциальные решения, если он углубляется в бесконечное дерево.

8. Как временная сложность поиска в глубину зависит от максимальной глубины дерева?

Временная сложность поиска в глубину зависит от максимальной глубины дерева, так как общее количество узлов, сгенерированных поиском в глубину, составляет b^m, где b — коэффициент ветвления, а m — максимальная глубина дерева.

9. Почему поиск в глубину не гарантирует нахождение оптимального решения?

Поиск в глубину не гарантирует нахождение оптимального решения, так как он может найти решение на большей глубине, чем наименьшая по стоимости.

10. В каких ситуациях предпочтительно использовать поиск в глубину, несмотря на его недостатки?

Поиск в глубину предпочтительно использовать в ситуациях, когда пространственная эффективность является приоритетной, и когда недостатки, связанные с неполнотой и потенциально высокой временной сложностью, являются приемлемыми рисками.

11. Что делает функция depth\_first\_recursive\_search, и какие параметры она принимает?

Функция depth\_first\_recursive\_search решает задачу поиска в глубину и принимает два параметра: problem, представляющий задачу, и node, который является текущим узлом в процессе поиска.

12. Какую задачу решает проверка if node is None ?

Проверка if node is None создает начальный узел с использованием начального состояния задачи, если текущий узел не указан.

13. В каком случае функция возвращает узел как решение задачи?

Функция возвращает узел как решение задачи, если состояние текущего узла соответствует целевому состоянию задачи.

14. Почему важна проверка на циклы в алгоритме рекурсивного поиска в глубину?

Проверка на циклы важна в алгоритме рекурсивного поиска в глубину, чтобы предотвратить зацикливание, когда алгоритм постоянно возвращается к уже посещенным узлам.

15. Что возвращает функция при обнаружении цикла?

При обнаружении цикла функция возвращает специальное значение failure, указывающее на неудачу в поиске пути.

16. Как функция обрабатывает дочерние узлы текущего узла?

Функция обрабатывает дочерние узлы текущего узла, генерируя их путем расширения текущего узла и рекурсивно вызывая себя для каждого дочернего узла.

17. Какой механизм используется для обхода дерева поиска в этой реализации?

Для обхода дерева поиска в этой реализации используется рекурсивный механизм.

18. Что произойдет, если не будет найдено решение в ходе рекурсии?

Если не будет найдено решение в ходе рекурсии, функция возвращает failure, указывая на то, что решение не найдено.

19. Почему функция рекурсивно вызывает саму себя внутри цикла?

Функция рекурсивно вызывает саму себя внутри цикла для поиска решения среди дочерних узлов.

20. Как функция expand(problem, node) взаимодействует с текущим узлом?

Функция expand(problem, node) генерирует всех дочерних узлов текущего узла, расширяя его.

21. Какова роль функции is\_cycle(node) в этом алгоритме?

Роль функции is\_cycle(node) в этом алгоритме заключается в проверке, создает ли текущий узел цикл, чтобы предотвратить зацикливание.

22. Почему проверка if result в рекурсивном вызове важна для корректной работы алгоритма?

Проверка if result в рекурсивном вызове важна для корректной работы алгоритма, так как она определяет, найдено ли решение среди дочерних узлов.

23. В каких ситуациях алгоритм может вернуть failure ?

Алгоритм может вернуть failure в ситуациях, когда ни один из дочерних узлов не привел к решению или если обнаружен цикл.

24. Как рекурсивная реализация отличается от итеративного поиска в глубину?

Рекурсивная реализация отличается от итеративного поиска в глубину тем, что в рекурсивной реализации используется стек вызовов для хранения информации о текущем пути, тогда как в итеративной реализации используется явный стек.

25. Какие потенциальные проблемы могут возникнуть при использовании этого алгоритма для поиска в бесконечных деревьях?

Потенциальные проблемы, которые могут возникнуть при использовании этого алгоритма для поиска в бесконечных деревьях, включают возможность застревания в бесконечных ветвях и высокую временную сложность.

Вывод: приобрел навыки по работе с поиском в глубину с помощью языка программирования Python версии 3.x