**Міністерство освіти і науки України**

**Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського"**

**Факультет інформатики та обчислювальної техніки**

**Кафедра інформатики та програмної інженерії**

**Звіт**

з лабораторної роботи № 2 з дисципліни

«Проектування алгоритмів»

**«Неінформативний, інформативний та локальний пошук»**

**Виконав(ла)**

(шифр, прізвище, ім'я, по батькові)

*ІП-13 Вдовиченко С.Ю.*

**Перевірив**

(прізвище, ім'я, по батькові)

*Сопов О.О.*

Київ 2022

Зміст

[1 Мета лабораторної роботи 3](#_Toc81070688)

[2 ЗаВдання 4](#_Toc81070689)

[3 Виконання 8](#_Toc81070690)

[3.1 Псевдокод алгоритмів 8](#_Toc81070691)

[3.2 Програмна реалізація 8](#_Toc81070692)

[3.2.1 Вихідний код 8](#_Toc81070693)

[3.2.2 Приклади роботи 8](#_Toc81070694)

[3.3 Дослідження алгоритмів 8](#_Toc81070695)

[Висновок 11](#_Toc81070696)

[Критерії оцінювання 12](#_Toc81070697)

# Мета лабораторної роботи

Мета роботи – розглянути та дослідити алгоритми неінформативного, інформативного та локального пошуку. Провести порівняльний аналіз ефективності використання алгоритмів.

# ЗаВдання

Записати алгоритм розв’язання задачі у вигляді псевдокоду, відповідно до варіанту (таблиця 2.1).

Реалізувати програму, яка розв’язує поставлену задачу згідно варіанту (таблиця 2.1) за допомогою алгоритму неінформативного пошуку **АНП**, алгоритму інформативного пошуку **АІП,** що використовує задану евристичну функцію Func, або алгоритму локального пошуку **АЛП та бектрекінгу,** що використовує задану евристичну функцію Func.

Програму реалізувати на довільній мові програмування.

**Увага!** Алгоритм неінформативного пошуку **АНП,** реалізовується за принципом «AS IS», тобто так, як є, без додаткових модифікацій (таких як перевірка циклів, наприклад).

Провести серію експериментів для вивчення ефективності роботи алгоритмів. Кожний експеримент повинен відрізнятись початковим станом. Серія повинна містити не менше 20 експериментів для кожного алгоритму. Початковий стан зафіксувати у таблиці експериментів. За проведеними серіями необхідно визначити:

* середню кількість етапів (кроків), які знадобилось для досягнення розв’язку (ітерації);
* середню кількість випадків, коли алгоритм потрапляв в глухий кут (не міг знайти оптимальний розв’язок) – якщо таке можливе;
* середню кількість згенерованих станів під час пошуку;
* середню кількість станів, що зберігаються в пам’яті під час роботи програми.

Передбачити можливість обмеження виконання програми за часом (30 хвилин) та використання пам’яті (1 Гб).

**Використані позначення:**

* **8-ферзів** – Задача про вісім ферзів полягає в такому розміщенні восьми ферзів на шахівниці, що жодна з них не ставить під удар один одного. Тобто, вони не повинні стояти в одній вертикалі, горизонталі чи діагоналі.
* **8-puzzle** – гра, що складається з 8 однакових квадратних пластинок з нанесеними числами від 1 до 8. Пластинки поміщаються в квадратну коробку, довжина сторони якої в три рази більша довжини сторони пластинок, відповідно в коробці залишається незаповненим одне квадратне поле. Мета гри – переміщаючи пластинки по коробці досягти впорядковування їх по номерах, бажано зробивши якомога менше переміщень.
* **Лабіринт** – задача пошуку шляху у довільному лабіринті від початкової точки до кінцевої з можливими випадками відсутності шляху. Структура лабіринту зчитується з файлу, або генерується програмою.
* **LDFS** – Пошук вглиб з обмеженням глибини.
* **BFS** – Пошук вшир.
* **IDS** – Пошук вглиб з ітеративним заглибленням.
* **A\*** – Пошук А\*.
* **RBFS** – Рекурсивний пошук за першим найкращим співпадінням.
* **F1** – кількість пар ферзів, які б’ють один одного з урахуванням видимості (ферзь А може стояти на одній лінії з ферзем В, проте між ними стоїть ферзь С; тому А не б’є В).
* **F2** – кількість пар ферзів, які б’ють один одного без урахування видимості.
* **H1** – кількість фішок, які не стоять на своїх місцях.
* **H2** – Манхетенська відстань.
* **H3** – Евклідова відстань.
* **COLOR** – Задача розфарбування карти самостійно обраної країни, не менше 20 регіонів (областей). Необхідно розфарбувати карту не більше ніж у 4 різні кольори. Мається на увазі приписування кожному регіону власного кольору так, щоб кольори сусідніх регіонів відрізнялись. Використовувати евристичну функцію, яка повертає кількість пар суміжних вузлів, що мають однаковий колір (тобто кількість конфліктів). Реалізувати алгоритм пошуку із поверненнями (backtracking) для розв’язання поставленої задачі. Для підвищення швидкодії роботи алгоритму використати евристичну функцію, а початковим станом вважати випадкову вершину.
* **HILL –** Пошук зі сходженням на вершину з використанням із використанням руху вбік (на 100 кроків) та випадковим перезапуском (кількість необхідних разів запуску визначити самостійно).
* **ANNEAL** – Локальний пошук із симуляцією відпалу. Робоча характеристика – залежність температури Т від часу роботи алгоритму t. Можна розглядати лінійну залежність: T = 1000 - k∙t, де k – змінний коефіцієнт.
* **BEAM** – Локальний променевий пошук. Робоча характеристика – кількість променів k. Експерименти проводи із кількістю променів від 2 до 21.
* **MRV** – евристика мінімальної кількості значень;
* **DGR** – ступенева евристика.

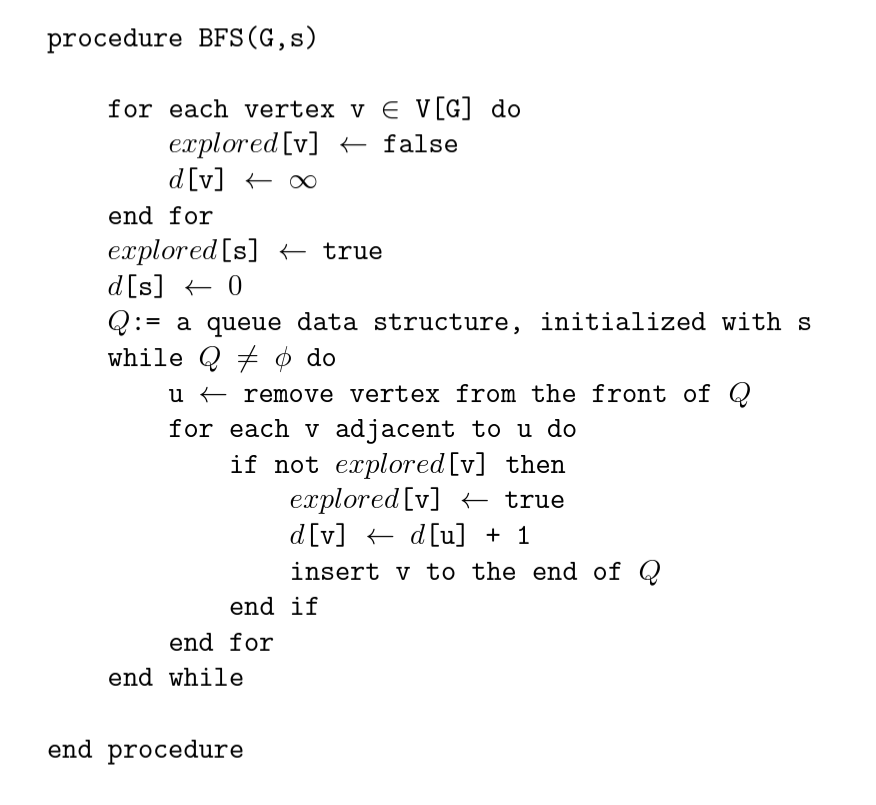
Таблиця 2.1 – Варіанти алгоритмів

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **№** | **Задача** | **АНП** | **АІП** | **АЛП** | **Func** |
| 1 | Лабіринт | LDFS | A\* |  | H2 |
| 2 | Лабіринт | LDFS | RBFS |  | H3 |
| 3 | Лабіринт | BFS | A\* |  | H2 |
| 4 | Лабіринт | BFS | RBFS |  | H3 |
| 5 | Лабіринт | IDS | A\* |  | H2 |
| 6 | Лабіринт | IDS | RBFS |  | H3 |
| 7 | 8-ферзів | LDFS | A\* |  | F1 |
| 8 | 8-ферзів | LDFS | A\* |  | F2 |
| 9 | 8-ферзів | LDFS | RBFS |  | F1 |
| 10 | 8-ферзів | LDFS | RBFS |  | F2 |
| 11 | 8-ферзів | BFS | A\* |  | F1 |
| 12 | 8-ферзів | BFS | A\* |  | F2 |
| 13 | 8-ферзів | BFS | RBFS |  | F1 |
| 14 | 8-ферзів | BFS | RBFS |  | F2 |
| 15 | 8-ферзів | IDS | A\* |  | F1 |
| 16 | 8-ферзів | IDS | A\* |  | F2 |
| 17 | 8-ферзів | IDS | RBFS |  | F1 |
| 18 | Лабіринт | LDFS | A\* |  | H3 |
| 19 | 8-puzzle | LDFS | A\* |  | H1 |
| 20 | 8-puzzle | LDFS | A\* |  | H2 |
| 21 | 8-puzzle | LDFS | RBFS |  | H1 |
| 22 | 8-puzzle | LDFS | RBFS |  | H2 |
| 23 | 8-puzzle | BFS | A\* |  | H1 |
| 24 | 8-puzzle | BFS | A\* |  | H2 |
| 25 | 8-puzzle | BFS | RBFS |  | H1 |
| 26 | 8-puzzle | BFS | RBFS |  | H2 |
| 27 | Лабіринт | BFS | A\* |  | H3 |
| 28 | 8-puzzle | IDS | A\* |  | H2 |
| 29 | 8-puzzle | IDS | RBFS |  | H1 |
| 30 | 8-puzzle | IDS | RBFS |  | H2 |
| 31 | COLOR |  |  | HILL | MRV |
| 32 | COLOR |  |  | ANNEAL | MRV |
| 33 | COLOR |  |  | BEAM | MRV |
| 34 | COLOR |  |  | HILL | DGR |
| 35 | COLOR |  |  | ANNEAL | DGR |
| 36 | COLOR |  |  | BEAM | DGR |

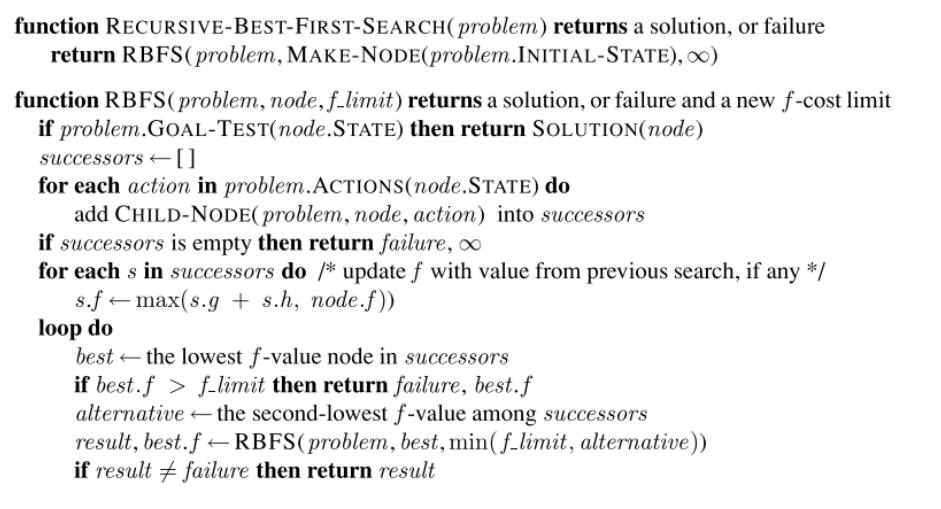
# Виконання

## Псевдокод алгоритмів

**BFS:**

****

**RBFS:**



## Програмна реалізація

### Вихідний код

from collections import deque  
import random  
from math import sqrt  
  
def print\_labyrinth(maze):  
 for row in maze:  
 for element in row:  
 print(element, end=' ')  
 print()  
  
def bfs\_labyrinth(maze, start, end, iterations, dead\_ends, states):  
 queue = deque()  
 queue.append(start)  
 visited = [[False for i in range(len(maze[0]))] for i in range(len(maze))]  
 visited[start[0]][start[1]] = True  
 parent = [[None for i in range(len(maze[0]))] for i in range(len(maze))]  
 row = [-1, 0, 0, 1]  
 col = [0, -1, 1, 0]  
 unique\_nodes = set()  
 unique\_nodes.add(start)  
  
 while queue:  
 curr = queue.popleft()  
 iterations += 1  
 unvisited\_neighbors = 0  
 if curr == end:  
 return construct\_path(parent, end), iterations, dead\_ends, states, unique\_nodes  
 x, y = curr[0], curr[1]  
 for i in range(4):  
 states += 1  
 if is\_valid(maze, x + row[i], y + col[i]) and not visited[x + row[i]][y + col[i]]:  
 unvisited\_neighbors += 1  
 visited[x + row[i]][y + col[i]] = True  
 parent[x + row[i]][y + col[i]] = curr  
 queue.append((x + row[i], y + col[i]))  
 unique\_nodes.add((x + row[i], y + col[i]))  
 if unvisited\_neighbors == 0:  
 dead\_ends += 1  
 return None  
  
def is\_valid(maze, x, y):  
 if x < 0 or y < 0 or x >= len(maze) or y >= len(maze[0]):  
 return False  
 if maze[x][y] == '#':  
 return False  
 return True  
  
def construct\_path(parent, curr):  
 if not parent[curr[0]][curr[1]]:  
 return [curr]  
 return construct\_path(parent, parent[curr[0]][curr[1]]) + [curr]  
  
def generate\_labyrinth(n, m, density):  
 maze = [['.' for i in range(m)] for i in range(n)]  
 for i in range(n):  
 for j in range(m):  
 if random.random() < density:  
 maze[i][j] = '#'  
 start = (random.randint(0, n-1), random.randint(0, m-1))  
 end = (random.randint(0, n-1), random.randint(0, m-1))  
 while maze[start[0]][start[1]] == '#':  
 start = (random.randint(0, n-1), random.randint(0, m-1))  
 while maze[end[0]][end[1]] == '#':  
 end = (random.randint(0, n-1), random.randint(0, m-1))  
 maze[start[0]][start[1]] = 'S'  
 maze[end[0]][end[1]] = 'E'  
 return maze, start, end  
  
def euclidean\_distance(p1, p2):  
 return sqrt((p1[0] - p2[0]) \*\* 2 + (p1[1] - p2[1]) \*\* 2)  
  
def rbfs(maze, start, end, iterations, dead\_ends, states):  
 visited = [[False for i in range(len(maze[0]))] for i in range(len(maze))]  
 parent = [[None for i in range(len(maze[0]))] for i in range(len(maze))]  
 f = euclidean\_distance(start, end)  
 unique\_states = set()  
 unique\_states.add(start)  
 path, iterations, dead\_ends, states, unique\_states = rbfs\_recursive(maze, start, end, f, visited, parent, iterations, dead\_ends, states, unique\_states)  
 return path, iterations, dead\_ends, states, unique\_states  
  
def rbfs\_recursive(maze, curr, end, f, visited, parent, iterations, dead\_ends, states, unique\_states):  
 visited[curr[0]][curr[1]] = True  
 # increment the counter variable  
 iterations += 1  
 states += 1  
 unique\_states.add(curr)  
 if curr == end:  
 return construct\_path(parent, end), iterations, dead\_ends, states, unique\_states  
 x, y = curr[0], curr[1]  
 min\_f = float('inf')  
 next\_node = None  
 dead\_end = True  
 for dx, dy in [(0, 1), (1, 0), (0, -1), (-1, 0)]:  
 if is\_valid(maze, x + dx, y + dy) and not visited[x + dx][y + dy]:  
 dead\_end = False  
 g = euclidean\_distance(curr, (x + dx, y + dy))  
 h = euclidean\_distance((x + dx, y + dy), end)  
 temp\_f = max(f, g + h)  
 if temp\_f < min\_f:  
 min\_f = temp\_f  
 next\_node = (x + dx, y + dy)  
 parent[x + dx][y + dy] = curr  
 if dead\_end:  
 dead\_ends += 1  
 if not next\_node:  
 return None, iterations, dead\_ends, states, unique\_states  
 return rbfs\_recursive(maze, next\_node, end, min\_f, visited, parent, iterations, dead\_ends, states, unique\_states)  
  
def main():  
 iterations = 0  
 states = 0  
 dead\_ends = 0  
 unique\_states = 0  
 m = int(input('Enter the size of maze: '))  
 maze, start, end = generate\_labyrinth(m, m, 0.2)  
  
 option = -1  
 while option != 1 and option != 2:  
 option = int(input('Print 1 to use BFS\nPrint 2 to use RBFS\n'))  
  
  
 if option == 1:  
 bfs\_path, iterations, dead\_ends, states, unique\_states = bfs\_labyrinth(maze, start, end, iterations, dead\_ends,  
 states)  
 print(f'bfs {bfs\_path}\n')  
 print\_labyrinth(maze)  
  
 if option == 2:  
 rbfs\_path, iterations, dead\_ends, states, unique\_states = rbfs(maze, start, end, iterations, dead\_ends, states)  
 while rbfs\_path == None:  
 maze, start, end = generate\_labyrinth(m, m, 0.2)  
 rbfs\_path, iterations, dead\_ends, states, unique\_states = rbfs(maze, start, end, iterations, dead\_ends, states)  
 print(f'rbfs {rbfs\_path}')  
 print\_labyrinth(maze)  
  
 print(f'start: {start}, end: {end}\niterations: {iterations}\ndead ends: {dead\_ends}\namount of states: {states}\nstates in memory: {len(unique\_states)}')  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

### Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми для різних алгоритмів пошуку.

Рисунок 3.1 – Алгоритм BFS

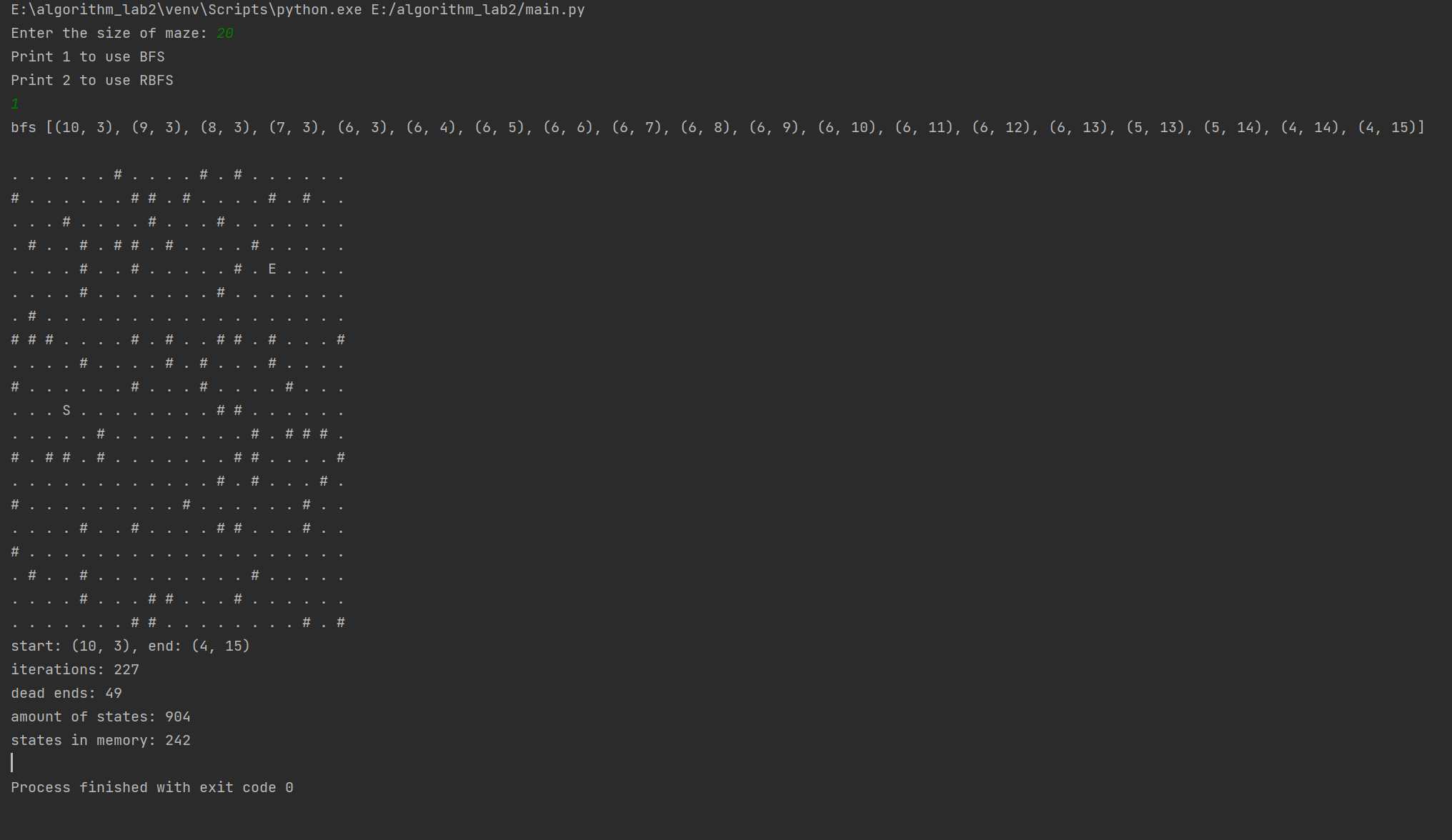
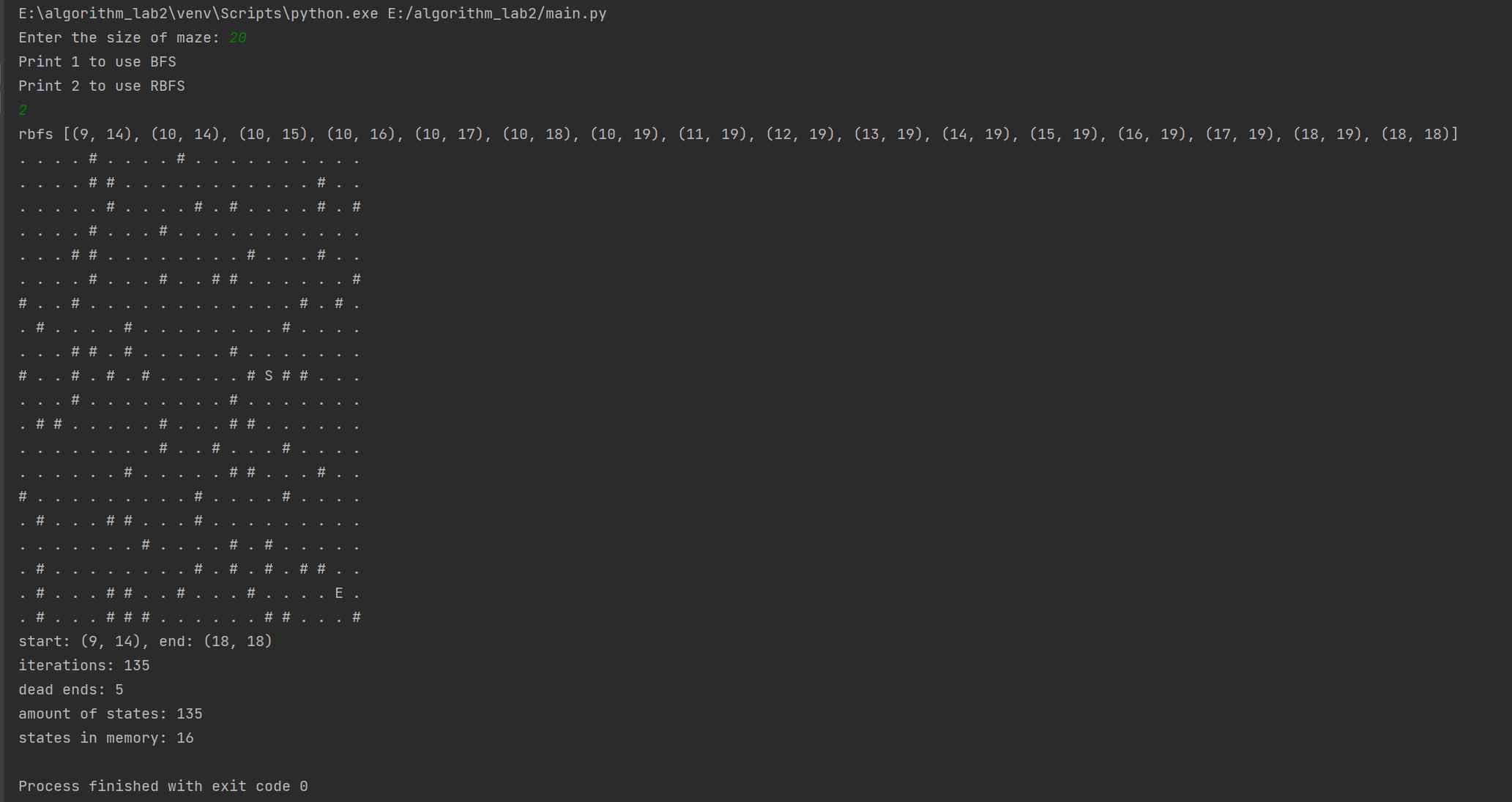


Рисунок 3.2 – Алгоритм RBFS



## Дослідження алгоритмів

В таблиці 3.1 наведені характеристики оцінювання алгоритму BFS, задачі лабіринту для 20 початкових станів.

Таблиця 3.1 – Характеристики оцінювання алгоритму BFS

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Початкові стани | Ітерації | К-сть глухих кутів | Всього станів | Всього станів у пам’яті |
| Стан 1 | 189 | 29 | 752 | 205 |
| Стан 2 | 235 | 43 | 936 | 247 |
| Стан 3 | 132 | 26 | 524 | 148 |
| Стан 4 | 333 | 80 | 1328 | 335 |
| Стан 5 | 126 | 27 | 500 | 135 |
| Стан 6 | 93 | 16 | 368 | 114 |
| Стан 7 | 118 | 23 | 468 | 129 |
| Стан 8 | 101 | 20 | 400 | 114 |
| Стан 9 | 35 | 3 | 136 | 48 |
| Стан 10 | 142 | 26 | 564 | 155 |
| Стан 11 | 259 | 58 | 1032 | 266 |
| Стан 12 | 33 | 4 | 128 | 42 |
| Стан 13 | 31 | 7 | 120 | 35 |
| Стан 14 | 307 | 77 | 1224 | 315 |
| Стан 15 | 80 | 9 | 316 | 96 |
| Стан 16 | 16 | 3 | 60 | 25 |
| Стан 17 | 260 | 68 | 1036 | 278 |
| Стан 18 | 189 | 41 | 752 | 206 |
| Стан 19 | 38 | 7 | 148 | 49 |
| Стан 20 | 259 | 53 | 1032 | 275 |

В таблиці 3.2 наведені характеристики оцінювання алгоритму RBFS, задачі Лабіринту для 20 початкових станів.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Початкові стани | Ітерації | К-сть глухих кутів | Всього станів | Всього станів у пам’яті |
| Стан 1 | 16 | 0 | 16 | 16 |
| Стан 2 | 43 | 2 | 43 | 16 |
| Стан 3 | 38 | 1 | 38 | 13 |
| Стан 4 | 44 | 1 | 44 | 1 |
| Стан 5 | 400 | 12 | 400 | 19 |
| Стан 6 | 119 | 4 | 119 | 28 |
| Стан 7 | 45 | 3 | 45 | 4 |
| Стан 8 | 425 | 18 | 425 | 42 |
| Стан 9 | 6 | 0 | 6 | 6 |
| Стан 10 | 320 | 15 | 320 | 52 |
| Стан 11 | 373 | 14 | 373 | 6 |
| Стан 12 | 25 | 0 | 25 | 25 |
| Стан 13 | 388 | 13 | 388 | 19 |
| Стан 14 | 235 | 10 | 235 | 5 |
| Стан 15 | 88 | 2 | 88 | 11 |
| Стан 16 | 640 | 34 | 640 | 6 |
| Стан 17 | 13 | 1 | 13 | 11 |
| Стан 18 | 6 | 0 | 6 | 6 |
| Стан 19 | 149 | 5 | 149 | 9 |
| Стан 20 | 302 | 12 | 302 | 9 |

Висновок

При виконанні даної лабораторної роботи було розглянуто алгоритми BFS та RBFS, було здійснено програмну реалізацію цих алгоритмів. Було здійснено 20 експериментів для кожного із алгоритмів і зафіксовано кількість ітерацій, кількість пройдених станів та максимальну кількість станів у пам’яті, чим виступала максимальна кількість елементів у черзі.

Критерії оцінювання

За умови здачі лабораторної роботи до 23.10.2022 включно максимальний бал дорівнює – 5. Після 23.10.2022 максимальний бал дорівнює – 1.

Критерії оцінювання у відсотках від максимального балу:

* псевдокод алгоритму – 10%;
* програмна реалізація алгоритму – 60%;
* дослідження алгоритмів – 25%;
* висновок – 5%.