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) *ІП-з21 Клименко М. М.* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив *Головченко М.М.* _____
(прізвище, ім'я, по батькові)

Київ 2024

ЗМІСТ

1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ

Мета роботи полягає у розгляді та дослідженні алгоритмів неінформативного, інформативного та локального пошуку. Важливим аспектом є проведення порівняльного аналізу ефективності цих алгоритмів. Це включає в себе реалізацію вказаних алгоритмів для розв'язання конкретних задач, таких як пошук шляху в лабіринті, використовуючи різні методи пошуку, а також оцінку їхньої ефективності на основі критеріїв, таких як кількість кроків до досягнення розв'язку, частота потрапляння в тупикові ситуації, кількість генерованих станів та обсяг використаної пам'яті під час виконання програм.

2 ЗАВДАННЯ

Записати алгоритм розв’язання задачі у вигляді псевдокоду, відповідно до варіанту 5 (таблиця 2.1).

Реалізувати програму, яка розв’язує поставлену задачу згідно варіанту 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:

- **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.
- **IDS** – Пошук вглиб з ітеративним заглибленням.
- **A*** – Пошук A*.
- **H2** – Манхетенська відстань.

Таблиця 2.1 – Варіанти алгоритмів

№	Задача	АНП	АІП	АЛП	Func
5	Лабіринт	IDS	A*		H2

3 ВИКОНАННЯ

3.1 Псевдокод алгоритмів

Псевдокод алгоритму IDS (Iterative Deepening Search)

```
ФУНКЦІЯ IDS(maze, start, goal, max_depth)
  ДЛЯ depth ВІД 0 ДО max_depth
    path, depth_reached, generated = DLS(maze, start, goal, depth)
    ЯКЩО path НЕ ПОРОЖНІЙ
      ПОВЕРНУТИ path, depth, depth_reached, generated
  КІНЕЦЬ ДЛЯ
  ПОВЕРНУТИ порожній список, 0, 0, generated

ФУНКЦІЯ DLS(maze, node, goal, depth)
  ЯКЩО depth = 0 І node = goal
    ПОВЕРНУТИ список [node], 1, 1
  ЯКЩО depth > 0
    ДЛЯ кожного successor ВІД get_successors(maze, node)
      path, depth_reached, gen = DLS(maze, successor, goal, depth - 1)
      ЯКЩО path НЕ ПОРОЖНІЙ
        ПОВЕРНУТИ [node] + path, depth_reached, gen
    КІНЕЦЬ ДЛЯ
  ПОВЕРНУТИ порожній список, 0, gen
КІНЕЦЬ ФУНКЦІЇ
```

Псевдокод алгоритму A* (A Star)

```
ФУНКЦІЯ ManhattanDistance(node, goal)
  ПОВЕРНУТИ abs(node.x - goal.x) + abs(node.y - goal.y)

ФУНКЦІЯ A*(maze, start, goal)
  ВІДКРИТИ_МНОЖИНА = {start}
  came_from = порожній словник
  g_score = словник з start: 0
  f_score = словник з start: ManhattanDistance(start, goal)
  max_in_memory = 0
  total_generated = 0

  ПОКИ ВІДКРИТИ_МНОЖИНА НЕ ПОРОЖНЯ
    current = вузол з ВІДКРИТИ_МНОЖИНА з мінімальним f_score[current]
    ЯКЩО current = goal
```

ПОВЕРНУТИ шлях від start до goal за допомогою came_from, довжина шляху, max_in_memory, total_generated
ВІДКРИТИ_МНОЖИНА видалити current
ДЛЯ кожного neighbor ВІД get_successors(maze, current)
total_generated += 1
tentative_g_score = g_score[current] + 1
ЯКЩО tentative_g_score < g_score.get(neighbor, нескінченність)
came_from[neighbor] = current
g_score[neighbor] = tentative_g_score
f_score[neighbor] = tentative_g_score + ManhattanDistance(neighbor, goal)
ЯКЩО neighbor не у ВІДКРИТИ_МНОЖИНА
ДОДАТИ neighbor до ВІДКРИТИ_МНОЖИНА
max_in_memory = максимум(max_in_memory, розмір ВІДКРИТИ_МНОЖИНА + розмір came_from)
КІНЕЦЬ ДЛЯ
КІНЕЦЬ ПОКИ
ПОВЕРНУТИ порожній список, 0, 0, total_generated
КІНЕЦЬ ФУНКЦІЇ

3.2 Програмна реалізація

3.2.1 Вихідний код

```
import time
import sys
# Функція для читання лабіринту з файлу та визначення стартової, кінцевої точок і самого лабіринту
def read_maze(file_path):
    with open(file_path, 'r') as file:
        lines = file.readlines()
        start = tuple(map(int, lines[0].split()))
        goal = tuple(map(int, lines[1].split()))
        maze = [[int(num) for num in line.split()] for line in lines[2:]]
    return maze, start, goal

# Функція для отримання наступників вузла в лабіринті
def get_successors(maze, node):
    successors = []
    directions = [(1, 0), (0, 1), (-1, 0), (0, -1)]  # Вниз, Вправо, Вгору, Вліво
```

```
        for dx, dy in directions:
            x, y = node[0] + dx, node[1] + dy
            if 0 <= x < len(maze) and 0 <= y < len(maze[0]) and maze[x][y] == 0:
                successors.append((x, y))
        return successors

    # Функція для перевірки часового обмеження
    def check_time_limit(start_time, time_limit_sec):
        if time.time() - start_time > time_limit_sec:
            print("Часове обмеження перевищено")
            sys.exit()

    # Функція для пошуку в глибину з обмеженням
    def dls(maze, node, goal, depth, max_depth=0, generated=0, start_time=None, time_limit_sec=None):
        check_time_limit(start_time, time_limit_sec)
        generated += 1
        if depth == 0 and node == goal:
            return [node], 1, generated
        if depth > 0:
            max_depth_reached = max_depth
            for successor in get_successors(maze, node):
                path, depth_reached, gen = dls(maze, successor, goal, depth - 1, max_depth_reached + 1, generated, start_time, time_limit_sec)
                max_depth_reached = max(max_depth_reached, depth_reached)
                generated = gen
                if path:
                    return [node] + path, max_depth_reached, generated
        return [], max_depth, generated

    # Функція для ітеративного поглиблення
    def ids(maze, start, goal, max_depth=50, start_time=None, time_limit_sec=None):
        total_generated = 0
        for depth in range(max_depth):
            path, depth_reached, generated = dls(maze, start, goal, depth, start_time=start_time, time_limit_sec=time_limit_sec)
            total_generated += generated
            if path:
                return path, depth, depth_reached, total_generated
```

```
    return [], 0, 0, total_generated

# Функція для обчислення манхеттенської відстані
def manhattan_distance(node, goal):
    return abs(node[0] - goal[0]) + abs(node[1] - goal[1])
# Функція для алгоритму A*
def a_star(maze, start, goal, start_time=None, time_limit_sec=None):
    open_set = {start}
    came_from = {}
    g_score = {start: 0}
    f_score = {start: manhattan_distance(start, goal)}
    max_in_memory = 0
    total_generated = 0

    while open_set:
        check_time_limit(start_time, time_limit_sec)
        current = min(open_set, key=lambda x: f_score[x])
        if current == goal:
            path = [current]
            while current in came_from:
                current = came_from[current]
                path.append(current)
            path.reverse()
            return path, len(path), max_in_memory, total_generated

        open_set.remove(current)
        for neighbor in get_successors(maze, current):
            total_generated += 1
            tentative_g_score = g_score[current] + 1
            if tentative_g_score < g_score.get(neighbor, float('inf')):
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g_score
                f_score[neighbor] = tentative_g_score + manhattan_distance(neighbor, goal)
                if neighbor not in open_set:
                    open_set.add(neighbor)
        max_in_memory = max(max_in_memory, len(open_set) + len(came_from))

    return [], 0, 0, 0
```

```
# Функція для виведення лабіринту з шляхом
def print_maze(maze, start, goal, path=None):
    for i, row in enumerate(maze):
        for j, val in enumerate(row):
            if (i, j) == start:
                print("S", end=" ")
            elif (i, j) == goal:
                print("G", end=" ")
            elif path and (i, j) in path:
                print("·", end=" ")
            else:
                print("#" if val == 1 else " ", end=" ")
        print()
# Виконання алгоритмів
maze_file = 'maze.txt'
maze, start, goal = read_maze(maze_file)

# Часові обмеження
start_time = time.time()
time_limit_sec = 30 * 60  # 30 хвилин

# Виконання алгоритмів IDS і A*
path_ids, steps_ids, memory_ids, generated_ids = ids(maze, start, goal, start_time=start_time, time_limit_sec=time_limit_sec)
path_a_star, steps_a_star, memory_a_star, generated_a_star = a_star(maze, start, goal, start_time=start_time, time_limit_sec=time_limit_sec)

# Виведення результатів
if len(maze) <= 20 and len(maze[0]) <= 20:
    print("Лабіринт:")
    print_maze(maze, start, goal)

print("Початкова точка:", start)
print("Кінцева точка:", goal)
print("IDS: шлях -", path_ids, "\nІтерації -", steps_ids, "\nВсього станів у пам'яті -", memory_ids, "\nКількість згенерованих станів -", generated_ids)
print("A*: шлях -", path_a_star, "\nІтерації -", steps_a_star, "\nВсього станів у пам'яті -", memory_a_star, "\nКількість згенерованих станів -", generated_a_star)
```

3.2.2 Приклади роботи

На рисунку 3.1 показаний приклад роботи програми для двох алгоритмів пошуку.

```
========== RESTART: /Users/mari/Desktop/Проектування алгоритмів/ЛАБОРАТОРНА 2/лабораторна 2.py ==========
Лабіринт:
S #           #
  #   # # #   #
              #
  # # #   # # #
        #
# #   # # # #
  #       #
  # # #       G
Початкова точка: (0, 0)
Кінцева точка: (7, 7)
IDS: шлях – [(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (4, 1), (4, 2), (5, 2), (6, 2), (6, 3), (6, 4), (7, 4), (7, 5), (7, 6), (7, 7)]
Ітерації – 14
Всього станів у пам'яті – 13
Кількість згенерованих станів – 36864
A*: шлях – [(0, 0), (1, 0), (2, 0), (2, 1), (2, 2), (2, 3), (2, 4), (3, 4), (4, 4), (4, 5), (4, 6), (4, 7), (5, 7), (6, 7), (7, 7)]
Ітерації – 15
Всього станів у пам'яті – 29
Кількість згенерованих станів – 43
```

Рис. 3.1 Приклад роботи програми для двох алгоритмів

3.3 Дослідження алгоритмів

В таблиці 3.1 наведені характеристики оцінювання алгоритму **“Пошук вглиб з ітеративним заглибленням”** задачі “Лабіринт” для 20 початкових станів.

Таблиця 3.1 – Характеристики оцінювання **алгоритму неінформативного пошуку**

Початкові стани	Ітерації	К-сть гл. кутів	Згенеровані стани	Всього станів у пам’яті
Стан 1 (0,0)	14	-	36864	13
Стан 2 (0,2)	12	-	16250	11
Стан 3 (0,4)	14	-	48560	15
Стан 4 (0, 6)	12	-	12037	13
Стан 5 (4, 0)	10	-	3000	9

Стан 6 (2, 4)	8	-	1287	7
Стан 7 (2, 2)	10	-	7352	9
Стан 8 (2, 6)	10	-	5227	11
Стан 9 (1, 2)	11	-	10660	10
Стан 10 (2, 1)	12	-	13227	10
Стан 11 (3, 0)	11	-	8875	10
Стан 12 (3, 4)	7	-	392	6
Стан 13 (1, 6)	11	-	7491	12
Стан 14 (4, 2)	8	-	529	7
Стан 15 (4, 4)	6	-	144	5
Стан 16 (4, 7)	3	-	15	2
Стан 17 (0, 5)	13	-	22173	14
Стан 18 (5, 1)	8	-	529	7
Стан 19 (2, 3)	9	-	2694	8
Стан 20 (4, 5)	5	-	64	4

- Середня кількість етапів (кроків), які знадобилось для досягнення розв’язку (ітерації) – 9,7
- Середня кількість згенерованих станів під час пошуку – 9868,5
- Середня кількість станів, що зберігаються в пам’яті під час роботи програми – 9,2

В таблиці 3.2 наведені характеристики оцінювання алгоритму **“Пошук А*”** задачі “Лабіринт” для 20 початкових станів.

Таблиця 3.2 – Характеристики оцінювання **алгоритму інформативного пошуку**

Початкові стани	Ітерації	К-сть гл. кутів	Згенеровані станів	Всього станів у пам'яті
Стан 1 (0,0)	15	-	43	29
Стан 2 (0,2)	13	-	41	24
Стан 3 (0,4)	15	-	41	24
Стан 4 (0, 6)	13	-	30	21
Стан 5 (4, 0)	11	-	21	15
Стан 6 (2, 4)	9	-	22	17
Стан 7 (2, 2)	11	-	27	21
Стан 8 (2, 6)	11	-	24	18
Стан 9 (1, 2)	12	-	29	22
Стан 10 (2, 1)	12	-	29	22
Стан 11 (3, 0)	12	-	23	16
Стан 12 (3, 4)	8	-	15	12
Стан 13 (1, 6)	12	-	28	20
Стан 14 (4, 2)	9	-	17	13
Стан 15 (4, 4)	7	-	13	11
Стан 16 (4, 7)	4	-	7	8
Стан 17 (0, 5)	14	-	32	22
Стан 18 (5, 1)	9	-	17	15
Стан 19 (2, 3)	10	-	24	18
Стан 20 (4, 5)	6	-	11	10

- Середня кількість етапів (кроків), які знадобилось для досягнення розв'язку (ітерації) – 10,65

- Середня кількість згенерованих станів під час пошуку – 24,7
- Середня кількість станів, що зберігаються в пам’яті під час роботи програми – 17,9

ВИСНОВОК

При виконанні даної лабораторної роботи було розглянуто Під час виконання даної лабораторної роботи було проведено детальне дослідження та аналіз трьох алгоритмів пошуку: неінформативного пошуку (IDS), інформативного пошуку (A*), та локального пошуку з бектрекінгом. Ключовими аспектами дослідження були реалізація цих алгоритмів для розв'язання задачі знаходження шляху в лабіринті, а також оцінка їх ефективності з урахуванням різних параметрів.

Основні висновки з дослідження:

Алгоритм IDS (Iterative Deepening Search): цей алгоритм показав високу ефективність у задачах з обмеженим простором станів. Однак, збільшення кількості генерованих станів і високий середній рівень ітерацій свідчать про потенційні труднощі при масштабуванні на більш складні задачі.

Алгоритм A (A Star)*: цей алгоритм продемонстрував високу ефективність у більшості випробувань, особливо з точки зору кількості згенерованих станів і збереження станів у пам'яті. Його евристичний підхід (Манхетенська відстань) значно покращує продуктивність, зменшуючи загальний простір пошуку.

Обидва алгоритми показали свої переваги та недоліки в різних умовах. IDS був ефективніший у сценаріях з обмеженим простором пошуку, тоді як A* був більш універсальним і ефективним у різних умовах завдяки своїй евристичній природі.

КРИТЕРІЇ ОЦІНЮВАННЯ

За умови здачі лабораторної роботи до 5.11.2023 включно максимальний бал дорівнює – 5. Після 5.11.2023 максимальний бал дорівнює – 4,5.

Критерії оцінювання у відсотках від максимального балу:

- псевдокод алгоритму – 10%;
- програмна реалізація алгоритму – 40%;
- робота з гіт – 20%;
- дослідження алгоритмів – 25%;
- висновок – 5%.