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-15 Шабанов М. Ш.* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив *Соколовський В.В.* _____
(прізвище, ім'я, по батькові)

Київ 2022

ЗМІСТ

1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ

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

2 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:

- **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.
- **BFS** – Пошук вшир.
- **A*** – Пошук A*.
- **H3** – Евклідова відстань.

Таблиця 2.1 – Варіанти алгоритмів

№	Задача	АНП	АІП	АЛП	Func
27	Лабіринт	BFS	A*		H3

3 ВИКОНАННЯ

3.1 Псевдокод алгоритмів

Алгоритм BFS:

1. ПОЧАТОК
2. frontier.put(start)
3. came_from[start] := null
4. cost_so_far[start] := 0
5. ПОКИ frontier не порожня:
    5.1. current := frontier.pop()
    5.2. ЯКЩО current == goal, ТО вийти з циклу та перейти до пункту 6.
    5.3. ДЛЯ next_node в (current.neighbour – visited):
        5.3.1. new_cost := cost_so_far[current] + cost(current, next)
        5.3.2. ЯКЩО cost_so_far не містить next_node АБО new_cost < cost_so_fat[next_node]:
            5.3.2.1. priority := new_cost
            5.3.2.2. frontier.push(next, priority)
            5.3.2.3. came_from[next_node] := current
    5.4. visited.append(current)
6. current := goal
7. path := [current]
8. ПОКИ current != start:
    8.1. current := came_from[current]
    8.2. path.append(current)
9. path.reverse()
10. КІНЕЦЬ

Алгоритм A*:

1. ПОЧАТОК
2. frontier.put(start)
3. came_from[start] := null
4. cost_so_far[start] := 0
5. ПОКИ frontier не порожня:
    5.1. current := frontier.pop()

5.2. ЯКЩО current == goal, ТО вийти з циклу та перейти до пункту 6.

5.3. ДЛЯ next_node в (current.neighbour – visited):

5.3.1. new_cost := cost_so_far[current] + cost(current, next)

5.3.2. ЯКЩО cost_so_far не містить next_node АБО new_cost < cost_so_fat[next_node]:

5.3.2.1. dx := abs(next_node.x – goal.x)

5.3.2.2. dy := abs(next_node.y – goal.y)

5.3.2.3. priority := new_cost + sqrt(dx * dx + dy * dy)

5.3.2.4. frontier.push(next, priority)

5.3.2.5. came_from[next_node] := current

5.4. visited.append(current)

6. current := goal
7. path := [current]
8. ПОКИ current != start:
   8.1. current := came_from[current]
   8.2. path.append(current)
9. path.reverse()
10. КІНЕЦЬ

3.2 Програмна реалізація

3.2.1 Вихідний код

Source code

Алгоритм BFS:

```
def BFS(self, start, goal):
    border = PriorityQueue()
    came_from = {}
    path_cost = dict()
    visited = []
    border.put((0, (0, start)))
    came_from[start] = None
    path_cost[start] = 0

    while not border.empty():
        current = border.get()[1]

        if current == (1, goal):
            return Graph.reconstruct_path(came_from, start, goal), visited

        for next_node in list(set(self.graph[current[1]]) - set(visited)):
            if next_node != (1, start):
                new_cost = path_cost[current[1]] + next_node[0]
```

```
                if next_node not in path_cost or (next_node in path_cost and
new_cost < path_cost[next_node][0]):
                    path_cost[next_node[1]] = new_cost
                    priority = new_cost
                    border.put((priority, next_node))
                    came_from[next_node[1]] = current
        visited.append(current)
    return -1, -1
```

```
def reconstruct_path(came_from, start, goal):
    current = (1, goal)
    path = [current]
    while current[1] != start:
        current = came_from[current[1]]
        path.append(current)
    path.reverse()
    return path
```

Алгоритм A*:

```
def AStarAlgorithm(self, start, goal, heuristic):
    border = PriorityQueue()
    came_from = {}
    path_cost = dict()
    visited = []

    border.put((0, (0, start)))
    came_from[start] = None
    path_cost[start] = 0

    while not border.empty():
        current = border.get()[1]

        if current == (1, goal):
            return Graph.reconstruct_path(came_from, start, goal), visited

        for next_node in list(set(self.graph[current[1]]) - set(visited)):
            new_cost = path_cost[current[1]] + next_node[0]
            if next_node not in path_cost or (next_node in path_cost and
new_cost < path_cost[next_node][1]):
                path_cost[next_node[1]] = new_cost
                priority = new_cost + GraphForAStar.euclidHeuristic(goal,
next_node[1])
                border.put((priority, next_node))
                came_from[next_node[1]] = current
        visited.append(current)
    return -1, -1

@staticmethod
def euclidHeuristic(dot1, dot2):
    return math.sqrt((dot1[0] - dot2[0]) ** 2 + (dot1[1] - dot2[1]) ** 2)

def reconstruct_path(came_from, start, goal):
    current = (1, goal)
    path = [current]
    while current[1] != start:
        current = came_from[current[1]]
        path.append(current)
    path.reverse()
    return path
```

3.2.2 Приклади роботи

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

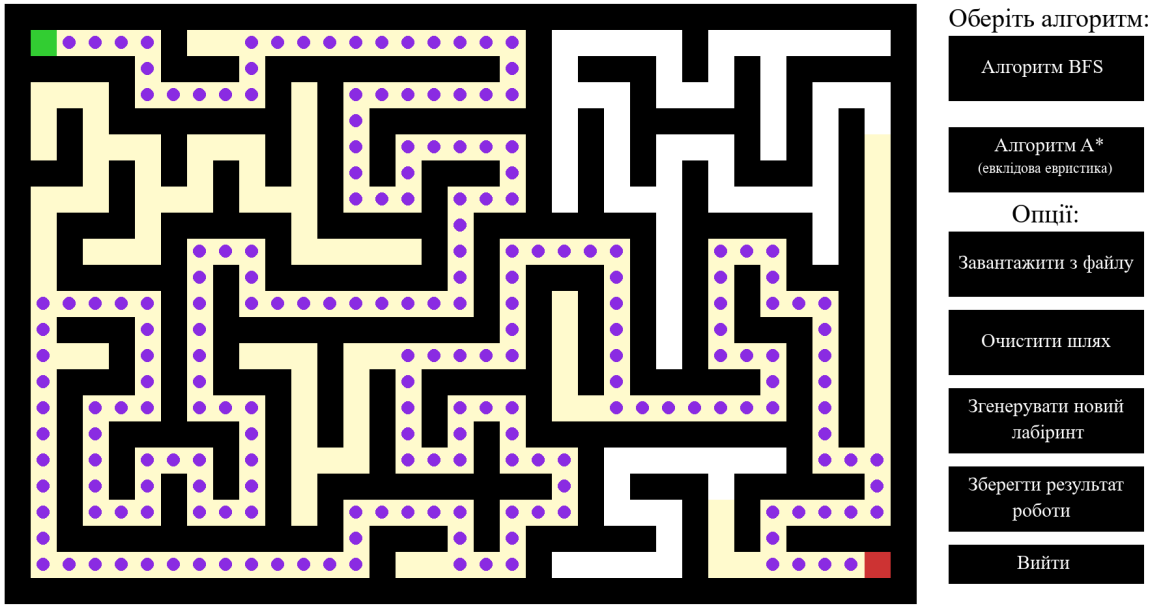

Рисунок 3.1 – Алгоритм BFS

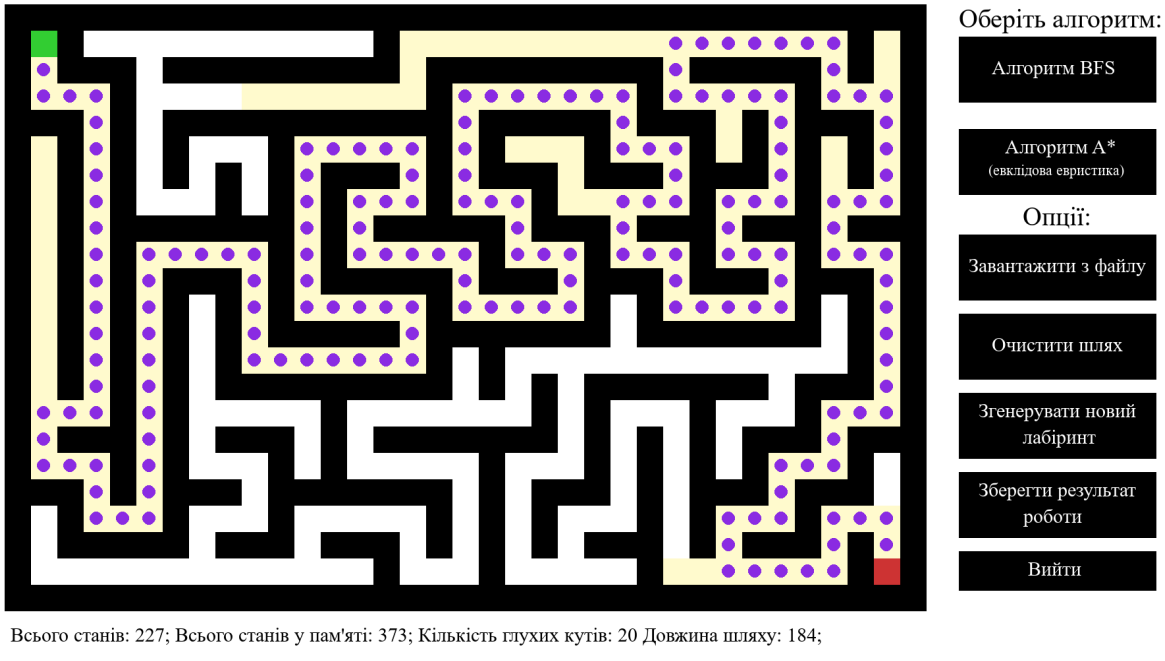

Рисунок 3.2 – Алгоритм A*

3.3 Дослідження алгоритмів

В таблиці 3.1 наведені характеристики оцінювання алгоритму BFS, задачі пошуку у лабіринті для 20 початкових станів.

Таблиця 3.1 – Характеристики оцінювання алгоритму BFS

Початкові стани	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пам’яті
Стан 1	95	9	92	139
Стан 2	122	12	118	139
Стан 3	66	7	64	139
Стан 4	125	12	122	139
Стан 5	127	10	124	139
Стан 6	178	19	176	373
Стан 7	361	23	360	373
Стан 8	291	19	288	373
Стан 9	367	20	366	373
Стан 10	208	18	206	373
Стан 11	705	74	702	1495
Стан 12	823	87	821	1495
Стан 13	1495	82	1495	1495
Стан 14	769	75	765	1495
Стан 15	1392	71	1389	1495
Стан 16	2881	312	2878	6439
Стан 17	1775	322	1771	6439
Стан 18	1601	343	1604	6439
Стан 19	6330	325	6325	6439
Стан 20	3400	325	3397	6439

В таблиці 3.2 наведені характеристики оцінювання алгоритму A*, задачі пошуку у лабіринті для 20 початкових станів.

Таблиця 3.3 – Характеристики оцінювання алгоритму A*

Початкові стани	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пам’яті
Стан 1	127	8	125	139
Стан 2	131	10	130	139
Стан 3	109	8	106	139
Стан 4	130	7	128	139
Стан 5	138	9	137	139
Стан 6	251	22	249	373
Стан 7	344	26	341	373
Стан 8	373	27	373	373
Стан 9	191	22	189	373
Стан 10	294	24	291	373
Стан 11	705	75	702	1495
Стан 12	698	75	697	1495
Стан 13	1483	89	1481	1495
Стан 14	1229	86	1227	1495
Стан 15	1427	88	1424	1495
Стан 16	6324	316	6320	6439
Стан 17	3429	320	3426	6439
Стан 18	4346	332	4345	6439
Стан 19	3989	325	3984	6439
Стан 20	5068	340	5065	6439

ВИСНОВОК

При виконанні даної лабораторної роботи було розглянуто та досліджено алгоритми неінформативного та інформативного пошуку, на прикладі алгоритмів пошуку вшир та A*. Проведено порівняльний аналіз ефективності використання алгоритмів. Згідно результатам тестування, можна зробити висновок, що у лабіринті алгоритм A* має невелику перевагу над алгоритмом пошуку вшир. Тим не менш, різниця у швидкості знаходження шляху стає більш очевидною, якщо пошук відбувається не в лабіринті, а на відкритій площині, і це наслідок використання евристичних функцій у роботі алгоритму інформованого пошуку.

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%.