**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-21 Скрипець Ольга*

**Перевірив**

(прізвище, ім'я, по батькові)

*Головченко М.М.*

Київ 2023

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

**функція** bfs(maze):  
 directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  
 start = (maze.shape[0] - 1, maze.shape[1] - 1)  
 end = (1, 0)  
 visited = np.zeros\_like(maze, dtype=bool)  
 visited[start] = True  
 queue = Queue()  
 queue.put((start, []))  
  
 **коли не** queue.empty()  
 (node, path) = queue.get()  
 neighbors = [  
 (node[0] + dx, node[1] + dy) for dx, dy in directions  
 ]  
 valid\_neighbors = [  
 neighbor  
 **для** neighbor in neighbors  
 **якщо** (  
 0 <= neighbor[0] < maze.shape[0]  
 and 0 <= neighbor[1] < maze.shape[1]  
 and maze[neighbor] == 0  
 )  
 ]

**для** next\_node **в** valid\_neighbors:  
 if next\_node == end:  
 return path + [next\_node]  
 **якщо** (  
 next\_node[0] >= 0  
 and next\_node[1] >= 0  
 and next\_node[0] < maze.shape[0]  
 and next\_node[1] < maze.shape[1]  
 and maze[next\_node] == 0  
 and not visited[next\_node]  
 ):  
 visited[next\_node] = True  
 queue.put((next\_node, path + [next\_node]))  
 return None

def h(cell, goal):  
 x1, y1 = cell  
 x2, y2 = goal  
 return ((x1 - x2) \*\* 2 + (y1 - y2) \*\* 2) \*\* 0.5  
  
  
def reconstruct\_path(aPath, current):  
 total\_path = [current]  
 **коли** current in aPath:  
 current = aPath[current]  
 total\_path.insert(0, current)  
 return total\_path  
  
  
**функція** aStar(maze, start, goal, closed\_set=None):  
 directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  
 open = PriorityQueue()  
 open.put((0, start))  
 aPath = {}  
 g\_score = np.full(maze.shape, float("inf"))  
 g\_score[start] = 0  
 f\_score = np.full(maze.shape, float("inf"))  
 f\_score[start] = h(start, goal)  
  
 **коли не** open.empty():  
 current = open.get()[1]  
 **якщо** current == goal:  
 return reconstruct\_path(aPath, current)

**якщо** closed\_set is not None:  
 closed\_set.add(current)  
 neighbors = [(current[0] + dx, current[1] + dy) for dx, dy in directions]  
 valid\_neighbors = [  
 neighbor  
 **для** neighbor in neighbors  
 **якщо** (  
 0 <= neighbor[0] < maze.shape[0]  
 and 0 <= neighbor[1] < maze.shape[1]  
 and maze[neighbor] == 0  
 )  
 ]  
  
   
 **для** next\_cell in valid\_neighbors:  
 **якщо** next\_cell not in visited\_states

tentative\_g\_score = g\_score[current] + 1  
 **якщо** tentative\_g\_score < g\_score[next\_cell]:  
 aPath[next\_cell] = current  
 g\_score[next\_cell] = tentative\_g\_score  
 f\_score[next\_cell] = tentative\_g\_score + h(next\_cell, goal)  
 open.put((f\_score[next\_cell], next\_cell))  
  
 return Nonе

## Програмна реалізація

### Вихідний код

import matplotlib.pyplot as plt  
import numpy as np  
import random  
from queue import Queue  
from queue import PriorityQueue  
  
def create\_maze(dim):  
 maze = np.ones((dim, dim))  
 x, y = (0, 0)  
  
 maze[x, y] = 0  
 stack = [(x, y)]  
 while len(stack) > 0:  
 x, y = stack[-1]  
  
 directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  
 random.shuffle(directions)  
  
 path\_available = False  
 for dx, dy in directions:  
 nx, ny = x + 2 \* dx, y + 2 \* dy  
 if 0 <= nx < dim and 0 <= ny < dim and maze[nx, ny] == 1:  
 maze[nx, ny] = 0  
 maze[x + dx, y + dy] = 0  
 stack.append((nx, ny))  
 path\_available = True  
 break  
  
 if not path\_available:  
 stack.pop()  
  
 maze[dim - 2, dim - 1] = 0 # Початок  
 maze[1, 0] = 0 # Кінець  
 return maze  
  
  
def h(cell, goal):  
 x1, y1 = cell  
 x2, y2 = goal  
 return ((x1 - x2) \*\* 2 + (y1 - y2) \*\* 2) \*\* 0.5  
  
  
def reconstruct\_path(aPath, current):  
 total\_path = [current]  
 while current in aPath:  
 current = aPath[current]  
 total\_path.insert(0, current)  
 return total\_path  
  
  
def aStar(maze, start, goal, closed\_set=None):  
 directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  
 open = PriorityQueue()  
 open.put((0, start))  
 aPath = {}  
 g\_score = np.full(maze.shape, float("inf"))  
 g\_score[start] = 0  
 f\_score = np.full(maze.shape, float("inf"))  
 f\_score[start] = h(start, goal)  
  
 states = 0  
 iterations = 0  
 dead\_ends = 0  
 visited\_states = set()  
 while not open.empty():  
 states = max(states, len(open.queue))  
 iterations += 1  
 current = open.get()[1]  
 visited\_states.add(current)  
  
 if current == goal:  
 iterations += 1  
 return reconstruct\_path(aPath, current), states, iterations, dead\_ends, len(visited\_states)  
  
 if closed\_set is not None:  
 iterations += 1  
 closed\_set.add(current)  
  
 neighbors = [(current[0] + dx, current[1] + dy) for dx, dy in directions]  
 valid\_neighbors = [  
 neighbor  
 for neighbor in neighbors  
 if (  
 0 <= neighbor[0] < maze.shape[0]  
 and 0 <= neighbor[1] < maze.shape[1]  
 and maze[neighbor] == 0  
 )  
 ]  
  
 if len(valid\_neighbors) == 1: # If there is only one valid neighbor, it's a dead end  
 if current[0] != maze2.shape[0] - 2 or current[1] != maze2.shape[1] - 1:  
 if current[0] != maze2.shape[0] - 2 or current[1] != maze2.shape[1] - 2:  
 dead\_ends += 1  
  
 for next\_cell in valid\_neighbors:  
 iterations += 1  
 if next\_cell not in visited\_states: # Check if the neighbor has been visited  
 tentative\_g\_score = g\_score[current] + 1  
 if tentative\_g\_score < g\_score[next\_cell]:  
 aPath[next\_cell] = current  
 g\_score[next\_cell] = tentative\_g\_score  
 f\_score[next\_cell] = tentative\_g\_score + h(next\_cell, goal)  
 open.put((f\_score[next\_cell], next\_cell))  
  
 return None, states, iterations, dead\_ends, len(visited\_states)  
  
  
  
def bfs(maze):  
 directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  
 start = (maze.shape[0] - 1, maze.shape[1] - 1)  
 end = (1, 0)  
 visited = np.zeros\_like(maze, dtype=bool)  
 visited[start] = True  
 queue = Queue()  
 queue.put((start, []))  
  
 iterations = 0  
 dead\_ends = 0  
 states = 0  
 visited\_states = set()  
  
 while not queue.empty():  
 states = max(states, len(queue.queue))  
 iterations += 1  
 (node, path) = queue.get()  
 visited\_states.add(node)  
  
 neighbors = [  
 (node[0] + dx, node[1] + dy) for dx, dy in directions  
 ]  
 valid\_neighbors = [  
 neighbor  
 for neighbor in neighbors  
 if (  
 0 <= neighbor[0] < maze.shape[0]  
 and 0 <= neighbor[1] < maze.shape[1]  
 and maze[neighbor] == 0  
 )  
 ]  
  
 if len(valid\_neighbors) == 1 and len(path) > 1:  
 dead\_ends += 1  
  
 for next\_node in valid\_neighbors:  
 iterations += 1  
 if next\_node == end:  
 return path + [next\_node], states, iterations, dead\_ends, len(visited\_states)  
 if (  
 next\_node[0] >= 0  
 and next\_node[1] >= 0  
 and next\_node[0] < maze.shape[0]  
 and next\_node[1] < maze.shape[1]  
 and maze[next\_node] == 0  
 and not visited[next\_node]  
 ):  
 visited[next\_node] = True  
 queue.put((next\_node, path + [next\_node]))  
  
 return None, states, iterations, dead\_ends - 1, len(visited\_states)  
  
  
def visualize\_bfs(maze, path=None, algorithm\_name="Algorithm"):  
 fig, ax = plt.subplots(figsize=(10, 10))  
  
 fig.patch.set\_edgecolor('white')  
 fig.patch.set\_linewidth(0)  
  
 ax.imshow(maze, cmap=plt.cm.binary, interpolation='nearest')  
  
 if path is not None:  
 x\_coords = [x[1] for x in path]  
 y\_coords = [y[0] for y in path]  
 ax.plot(x\_coords, y\_coords, color='red', linewidth=3)  
  
 ax.set\_xticks([])  
 ax.set\_yticks([])  
  
 start\_x, start\_y = maze.shape[1] - 1, maze.shape[0] - 2  
 end\_x, end\_y = 0, 1  
  
 ax.plot(start\_x, start\_y, 'o', markersize=10, color='blue') # Кругла крапка для початку  
 ax.plot(end\_x, end\_y, 'x', markersize=20, color='red') # Хрестик для кінця  
  
 plt.title(algorithm\_name)  
 plt.show()  
  
  
def visualize\_astar(maze, path=None, closed\_set=None, algorithm\_name="Algorithm"):  
 fig, ax = plt.subplots(figsize=(10, 10))  
 fig.patch.set\_edgecolor('white')  
 fig.patch.set\_linewidth(0)  
 ax.imshow(maze, cmap=plt.cm.binary, interpolation='nearest')  
  
 if path:  
 x\_coords = [x[1] for x in path]  
 y\_coords = [y[0] for y in path]  
 ax.plot(x\_coords, y\_coords, color='blue', linewidth=3)  
  
 if closed\_set is not None:  
 # Позначення проглянутих клітинок жовтим кубиком  
 for cell in closed\_set:  
 x, y = cell  
 ax.add\_patch(plt.Rectangle((y - 0.4, x - 0.4), 0.8, 0.8, color='yellow'))  
  
 ax.set\_xticks([])  
 ax.set\_yticks([])  
 ax.plot(start[1], start[0], 'o', markersize=10, color='blue')  
 ax.plot(end[1], end[0], 'x', markersize=20, color='red')  
  
 plt.title(algorithm\_name)  
 plt.show()  
  
while True:  
 try:  
 dim = int(input("Enter the dimension of the maze: "))  
 if 3 <= dim <= 100:  
 break   
 else:  
 print("Please enter a value within the range of 3 to 100.")  
 except ValueError:  
 print("Please enter a valid number.")  
  
maze1 = create\_maze(dim)  
maze2 = np.copy(maze1)  
  
start = (maze2.shape[0] - 2, maze2.shape[1] - 1)  
end = (1, 0)  
  
closed\_set = set()  
  
path\_bfs, states\_bfs, iterations\_bfs, dead\_ends\_bfs, total\_states\_bfs = bfs(maze1)  
path\_astar, states\_astar, iterations\_astar, dead\_ends\_astar, total\_states\_astar = aStar(maze2, start, end, closed\_set)  
  
print("BFS states:", states\_bfs)  
print("BFS dead ends:", dead\_ends\_bfs)  
print("BFS iterations:", iterations\_bfs)  
print("BFS total states:", total\_states\_bfs)  
  
  
print("A\* states:", states\_astar)  
print("A\* dead ends:", dead\_ends\_astar)  
print("A\* iterations:", iterations\_astar)  
print("A\* total states:", total\_states\_astar)  
  
visualize\_bfs(maze1, path\_bfs, "BFS")  
visualize\_astar(maze2, path\_astar, closed\_set, "A\*")

### Приклади роботи

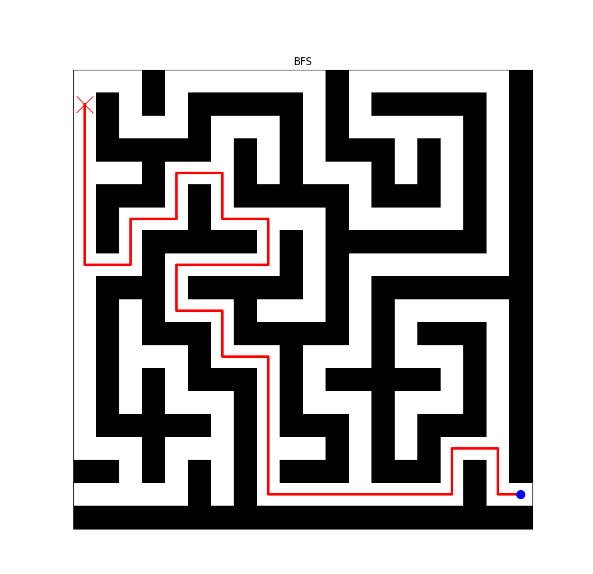
На рисунках 3.1 і 3.2 показані приклади роботи програми для різних алгоритмів пошуку.

Рисунок 3.1 – Алгоритм BFS для лабіринту 20\*20

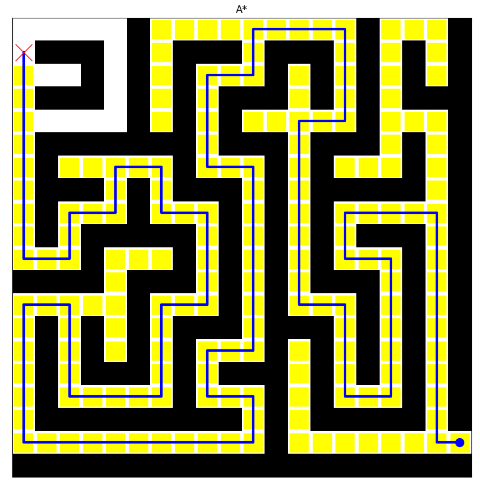


Рисунок 3.2 – Алгоритм Astar 20\*20

## Дослідження алгоритмів

В таблиці 3.1 наведені характеристики оцінювання алгоритму BFS задачі Лабіринт 20\*20 для 20 початкових станів.

Таблиця 3.1 – Характеристики оцінювання алгоритму BFS для Лабіринту 20\*20

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Початкові стани | Ітерації | К-сть гл. кутів | Всього станів | Всього станів у пом’яті |
| Стан 1 | 567 | 9 | 3 | 189 |
| Стан 2 | 380 | 2 | 5 | 126 |
| Стан 3 | 494 | 8 | 5 | 165 |
| Стан 4 | 237 | 2 | 3 | 79 |
| Стан 5 | 544 | 8 | 6 | 181 |
| Стан 6 | 595 | 10 | 4 | 199 |
| Стан 7 | 413 | 2 | 5 | 137 |
| Стан 8 | 537 | 7 | 6 | 179 |
| Стан 9 | 595 | 14 | 6 | 199 |
| Стан 10 | 565 | 8 | 5 | 188 |
| Стан 11 | 595 | 11 | 5 | 199 |
| Стан 12 | 441 | 6 | 4 | 147 |
| Стан 13 | 458 | 3 | 6 | 152 |
| Стан 14 | 595 | 10 | 5 | 199 |
| Стан 15 | 483 | 6 | 4 | 161 |
| Стан 16 | 584 | 9 | 7 | 195 |
| Стан 17 | 595 | 10 | 5 | 199 |
| Стан 18 | 572 | 7 | 4 | 191 |
| Стан 19 | 595 | 12 | 5 | 199 |
| Стан 20 | 584 | 11 | 5 | 195 |
| **Середнє** | **544** | **8** | **5** | **174** |

В таблиці 3.2 наведені характеристики оцінювання алгоритму Astar задачі Лабіринт 20\*20 для 20 початкових станів.

Таблиця 3.3 – Характеристики оцінювання алгоритму Astar для Лабіринту 20\*20

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Початкові стани | Ітерації | К-сть гл. кутів | Всього станів | Всього станів у пом’яті |
| Стан 1 | 767 | 10 | 4 | 192 |
| Стан 2 | 793 | 14 | 6 | 199 |
| Стан 3 | 700 | 7 | 4 | 175 |
| Стан 4 | 518 | 3 | 3 | 130 |
| Стан 5 | 663 | 8 | 6 | 166 |
| Стан 6 | 793 | 6 | 3 | 199 |
| Стан 7 | 657 | 8 | 5 | 164 |
| Стан 8 | 793 | 10 | 5 | 199 |
| Стан 9 | 793 | 10 | 6 | 199 |
| Стан 10 | 571 | 4 | 4 | 143 |
| Стан 11 | 385 | 3 | 5 | 96 |
| Стан 12 | 231 | 1 | 3 | 58 |
| Стан 13 | 611 | 4 | 5 | 153 |
| Стан 14 | 444 | 5 | 4 | 111 |
| Стан 15 | 778 | 7 | 5 | 195 |
| Стан 16 | 522 | 4 | 3 | 131 |
| Стан 17 | 793 | 13 | 4 | 199 |
| Стан 18 | 642 | 4 | 6 | 160 |
| Стан 19 | 793 | 12 | 5 | 199 |
| Стан 20 | 770 | 12 | 3 | 193 |
| **Середнє** | **651** | **7** | **4** | **163** |

Висновок

У результаті виконання лабораторної роботи, були розглянуті та реалізовані алгоритми пошуку шляху в лабіринті за допомогою Breadth-First Search (BFS) та A\* із використанням евристичної функції "Евклідова відстань". Обидва алгоритми були випробувані на лабіринті розміром 20x20.

Breadth-First Search (BFS) - це алгоритм пошуку шляху в графі, який розглядає всі можливі шляхи на одному рівні обходу перед переходом на наступний рівень. Принципова властивість BFS полягає в тому, що він розглядає всі сусіди поточного вузла перед розгляданням сусідів їх сусідів. Цей алгоритм застосовує чергу для визначення порядку обробки вузлів.

Алгоритм A \* покроково переглядає всі шляхи, що ведуть від початкової вершини в кінцеву, поки не знайде мінімальний. При виборі вершини він враховує, крім іншого, весь пройдений до неї шлях. На початку роботи розглядаються вузли, суміжні з початковим; вибирається той з них, який має мінімальне значення f (n), після чого цей вузол розкривається. На кожному етапі алгоритм оперує з множиною шляхів з початкової точки до всіх ще не розкритих (листових) вершин графа - множиною часткових рішень, - яке розміщується в черзі з пріоритетом. Пріоритет шляху визначається за значенням f (n) = g (n) + h (n). Алгоритм продовжує свою роботу до тих пір, поки значення f (n) цільової вершини не виявиться меншим, ніж будь-яке значення в черзі, або поки все дерево не буде переглянуто. З множини рішень вибирається рішення з найменшою вартістю. Отож було застосовано структуру даних черга з пріоритетом.

**Характеристики оцінювання алгоритму BFS для Лабіринту 20\*20 показали наступні результати:**

Ітерації: Середнє значення ітерацій для знаходження шляху складає 544.

Кількість глухих кутів: У середньому алгоритм знаходить 8 глухих кутів.

Всього станів: Кількість всіх розглянутих станів становить 5.

Всього станів у пам'яті: Середня кількість станів, які потрапили в пам'ять, складає 174.

**Характеристики оцінювання алгоритму A\* для Лабіринту 20\*20 показали такі результати:**

Ітерації: Середнє значення ітерацій для знаходження шляху складає 651.

Кількість глухих кутів: У середньому алгоритм знаходить 7 глухих кутів.

Всього станів: Кількість всіх розглянутих станів становить 4.

Всього станів у пам'яті: Середня кількість станів, які потрапили в пам'ять, складає 163.

**Аналіз отриманих результатів дозволяє зробити наступні висновки:**

Ітерації: Алгоритм A\* вимагає більше ітерацій для знаходження шляху, що може вказувати на більшу обчислювальну складність порівняно із BFS.

Кількість глухих кутів: Обидва алгоритми виявляють глухі кути у лабіринті, проте A\* виявляє їх менше, що свідчить про кращу адаптованість до структури лабіринту.

Всього станів: Алгоритм A\* розглядає менше станів у загальному, що може свідчити про його більшу ефективність в обробці станів.

Всього станів у пам'яті: A\* використовує менше пам'яті для зберігання станів під час виконання.

Отже, при розв'язанні задачі пошуку шляху у лабіринті, алгоритм A\* із використанням евристичної функції "Евклідова відстань" є більш ефективним порівняно із BFS у визначених умовах.

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%.