**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-35 Брага Артем*

**Перевірив**

(прізвище, ім'я, по батькові)

*Головченко М.М.*

Київ 2024

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.
* **F3** – кількість ферзів, які стоять не на свої місцях, в порівнянні з одним з еталонних розвʼязків.
* **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 | COLOR |  |  | HILL | MRV |
| 2 | COLOR |  |  | ANNEAL | MRV |
| 3 | COLOR |  |  | BEAM | MRV |
| 4 | COLOR |  |  | HILL | DGR |
| 5 | COLOR |  |  | ANNEAL | DGR |
| 6 | COLOR |  |  | BEAM | DGR |
| 7 | 8-ферзів | LDFS | A\* |  | F1 |
| 8 | 8-ферзів | LDFS | A\* |  | F2 |
| 9 | 8-ферзів | LDFS | RBFS |  | F3 |
| 10 | 8-ферзів | LDFS | RBFS |  | F1 |
| 11 | 8-ферзів | BFS | A\* |  | F2 |
| 12 | 8-ферзів | BFS | A\* |  | F3 |
| 13 | 8-ферзів | BFS | RBFS |  | F1 |
| 14 | 8-ферзів | BFS | RBFS |  | F2 |
| 15 | 8-ферзів | IDS | A\* |  | F3 |
| 16 | 8-ферзів | IDS | A\* |  | F1 |
| 17 | 8-ферзів | IDS | RBFS |  | F2 |
| 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 | Лабіринт | LDFS | A\* |  | H2 |
| 32 | Лабіринт | LDFS | RBFS |  | H3 |
| 33 | Лабіринт | BFS | A\* |  | H2 |
| 34 | Лабіринт | BFS | RBFS |  | H3 |
| 35 | Лабіринт | IDS | A\* |  | H2 |
| 36 | Лабіринт | IDS | RBFS |  | H3 |

# Виконання

## Псевдокод алгоритмів

# Загальні функції  
FUNCTION is\_solvable(state):  
 RETURN True IF state CAN BE SOLVED, OTHERWISE False  
  
FUNCTION get\_neighbors(state):  
 RETURN LIST OF POSSIBLE NEXT STATES BASED ON EMPTY TILE MOVES  
  
FUNCTION h1(state, goal\_state):  
 RETURN NUMBER OF MISPLACED TILES  
  
FUNCTION reconstruct\_path(came\_from, current):  
 PATH = []  
 WHILE current EXISTS IN came\_from:  
 ADD current TO PATH  
 current = came\_from[current]  
 RETURN REVERSED(PATH)  
  
# BFS Algorithm  
FUNCTION bfs\_solve(initial\_state, goal\_state):  
 IF NOT is\_solvable(initial\_state):  
 RETURN "Unsolvable"  
 QUEUE = [initial\_state]  
 VISITED = SET()  
 CAME\_FROM = {}  
 WHILE QUEUE NOT EMPTY:  
 current = DEQUEUE(QUEUE)  
 IF current == goal\_state:  
 RETURN reconstruct\_path(CAME\_FROM, current)  
 ADD current TO VISITED  
 FOR neighbor IN get\_neighbors(current):  
 IF neighbor NOT IN VISITED:  
 ENQUEUE(QUEUE, neighbor)  
 CAME\_FROM[neighbor] = current  
 RETURN "No solution found"  
  
# A\* Algorithm  
FUNCTION a\_star\_solve(initial\_state, goal\_state):  
 IF NOT is\_solvable(initial\_state):  
 RETURN "Unsolvable"  
 OPEN\_SET = PRIORITY\_QUEUE()  
 OPEN\_SET.ADD(initial\_state, h1(initial\_state, goal\_state))  
 G\_SCORE = {initial\_state: 0}  
 F\_SCORE = {initial\_state: h1(initial\_state, goal\_state)}  
 CAME\_FROM = {}  
 WHILE OPEN\_SET NOT EMPTY:  
 current = OPEN\_SET.REMOVE\_MIN()  
 IF current == goal\_state:  
 RETURN reconstruct\_path(CAME\_FROM, current)  
 FOR neighbor IN get\_neighbors(current):  
 tentative\_g\_score = G\_SCORE[current] + 1  
 IF neighbor NOT IN G\_SCORE OR tentative\_g\_score < G\_SCORE[neighbor]:  
 CAME\_FROM[neighbor] = current  
 G\_SCORE[neighbor] = tentative\_g\_score  
 F\_SCORE[neighbor] = G\_SCORE[neighbor] + h1(neighbor, goal\_state)  
 IF neighbor NOT IN OPEN\_SET:  
 OPEN\_SET.ADD(neighbor, F\_SCORE[neighbor])  
 RETURN "No solution found"  
  
# Backtracking with H1  
FUNCTION backtrack\_solve(initial\_state, goal\_state):  
 IF initial\_state == goal\_state:  
 RETURN [initial\_state]  
 FOR neighbor IN SORT(get\_neighbors(initial\_state), BY h1(neighbor, goal\_state)):  
 PATH = backtrack\_solve(neighbor, goal\_state)  
 IF PATH IS NOT None:  
 RETURN [initial\_state] + PATH  
 RETURN None  
  
# Main  
FUNCTION run\_experiments():  
 INITIAL\_STATES = GENERATE\_RANDOM\_STATES(20)  
 FOR algorithm IN [bfs\_solve, a\_star\_solve, backtrack\_solve]:  
 FOR state IN INITIAL\_STATES:  
 START TIMER  
 solution = algorithm(state, goal\_state)  
 END TIMER  
 COLLECT STATISTICS (steps, generated states, memory, etc.)  
 RETURN EXPERIMENT\_RESULTS

## Програмна реалізація

### Вихідний код

import time  
from collections import deque  
import heapq  
import random  
import tracemalloc  
  
# Heuristic function (H1)  
def h1(state, goal\_state):  
 return sum(1 for i in range(len(state)) if state[i] != goal\_state[i] and state[i] != 0)  
  
# Генерування сусідів  
def get\_neighbors(state):  
 neighbors = []  
 zero\_index = state.index(0)  
 row, col = divmod(zero\_index, 3)  
  
 directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  
 for dr, dc in directions:  
 new\_row, new\_col = row + dr, col + dc  
 if 0 <= new\_row < 3 and 0 <= new\_col < 3:  
 new\_index = new\_row \* 3 + new\_col  
 new\_state = list(state)  
 new\_state[zero\_index], new\_state[new\_index] = new\_state[new\_index], new\_state[zero\_index]  
 neighbors.append(tuple(new\_state))  
  
 return neighbors  
  
# BFS алгоритм  
def bfs\_solve(initial\_state, goal\_state):  
 queue = deque([(initial\_state, [])])  
 visited = set()  
 generated = 0  
  
 while queue:  
 state, path = queue.popleft()  
 if state == goal\_state:  
 return path, generated, len(visited)  
  
 visited.add(state)  
 for neighbor in get\_neighbors(state):  
 if neighbor not in visited:  
 queue.append((neighbor, path + [neighbor]))  
 generated += 1  
  
 return None, generated, len(visited)  
  
# A\* алгоритм  
def a\_star\_solve(initial\_state, goal\_state):  
 open\_set = []  
 heapq.heappush(open\_set, (h1(initial\_state, goal\_state), 0, initial\_state, []))  
 closed\_set = set()  
 generated = 0  
  
 while open\_set:  
 \_, cost, state, path = heapq.heappop(open\_set)  
 if state == goal\_state:  
 return path, generated, len(closed\_set)  
  
 closed\_set.add(state)  
 for neighbor in get\_neighbors(state):  
 if neighbor not in closed\_set:  
 new\_cost = cost + 1  
 heapq.heappush(open\_set, (new\_cost + h1(neighbor, goal\_state), new\_cost, neighbor, path + [neighbor]))  
 generated += 1  
  
 return None, generated, len(closed\_set)  
  
# Backtracking алгоритм  
def backtrack\_solve(initial\_state, goal\_state, path=None, visited=None, max\_depth=50):  
 if path is None:  
 path = []  
 if visited is None:  
 visited = set()  
  
 # Додаємо перевірку на максимальну глибину  
 if len(path) > max\_depth:  
 return None, 0, 0 # Повертаємо, якщо глибина перевищена  
  
 # Якщо стан вже відвідано, пропускаємо його  
 if tuple(initial\_state) in visited:  
 return None, 0, 0  
  
 visited.add(tuple(initial\_state))  
  
 if initial\_state == goal\_state:  
 return path + [initial\_state], len(visited), len(visited)  
  
 for neighbor in sorted(get\_neighbors(initial\_state), key=lambda s: h1(s, goal\_state)):  
 result, generated, memory = backtrack\_solve(neighbor, goal\_state, path + [neighbor], visited, max\_depth)  
 if result is not None:  
 return result, generated, memory  
  
 return None, len(visited), len(visited)  
  
# запуск праці алгоритмів  
def run\_experiments():  
 initial\_states = [tuple(random.sample(range(9), 9)) for \_ in range(20)]  
 goal\_state = tuple(range(9))  
 algorithms = [  
 ("BFS", bfs\_solve),  
 ("A\*", a\_star\_solve),  
 ("Backtracking", lambda s, g: backtrack\_solve(s, g))  
 ]  
  
 results = {}  
  
 for name, algorithm in algorithms:  
 steps = []  
 generated\_states = []  
 memory\_usage = []  
  
 for state in initial\_states:  
 start\_time = time.time()  
 tracemalloc.start()  
 solution, generated, memory = algorithm(state, goal\_state)  
 \_, peak\_memory = tracemalloc.get\_traced\_memory()  
 tracemalloc.stop()  
  
 if solution is not None:  
 steps.append(len(solution))  
 else:  
 steps.append(None)  
  
 generated\_states.append(generated)  
 memory\_usage.append(peak\_memory // 1024)  
  
 if time.time() - start\_time > 1800:   
 print(f"{name} algorithm exceeded time limit for state {state}")  
 break  
  
 results[name] = {  
 "Average Steps": sum(filter(None, steps)) / len(steps),  
 "Average Generated States": sum(generated\_states) / len(generated\_states),  
 "Average Memory Usage (KB)": sum(memory\_usage) / len(memory\_usage)  
 }  
  
 for name, result in results.items():  
 print(f"Results for {name}:")  
 for key, value in result.items():  
 print(f" {key}: {value}")  
 print()  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 run\_experiments()

### Приклади роботи

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

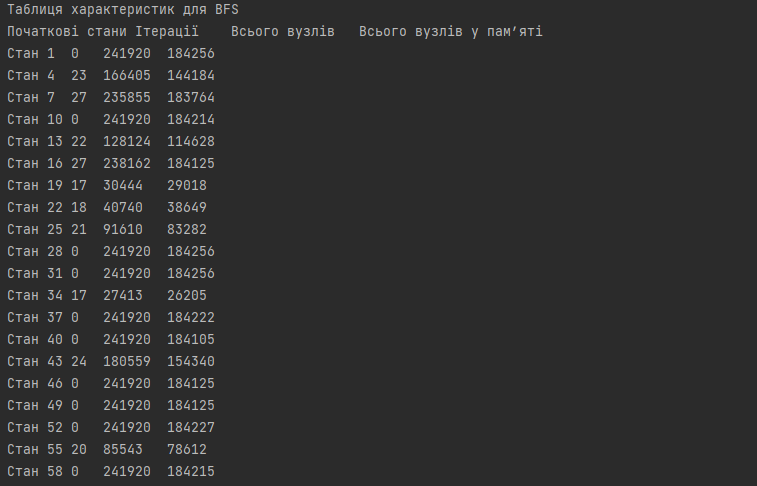


Рисунок 3.1 – Алгоритм BFS

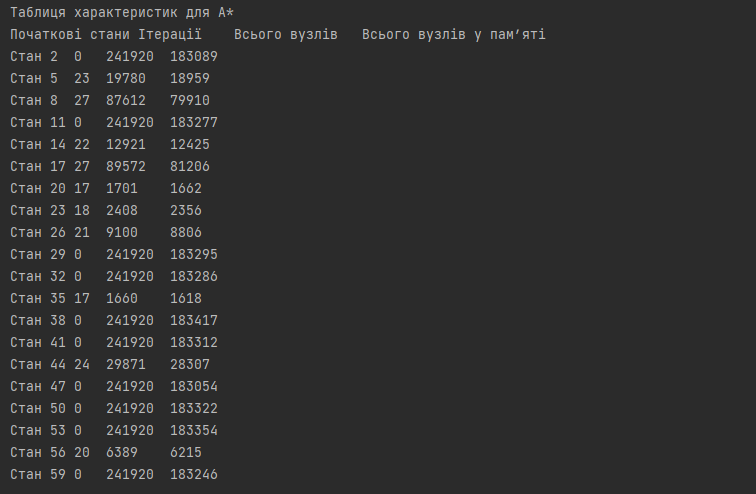


Рисунок 3.2 – Алгоритм A\*

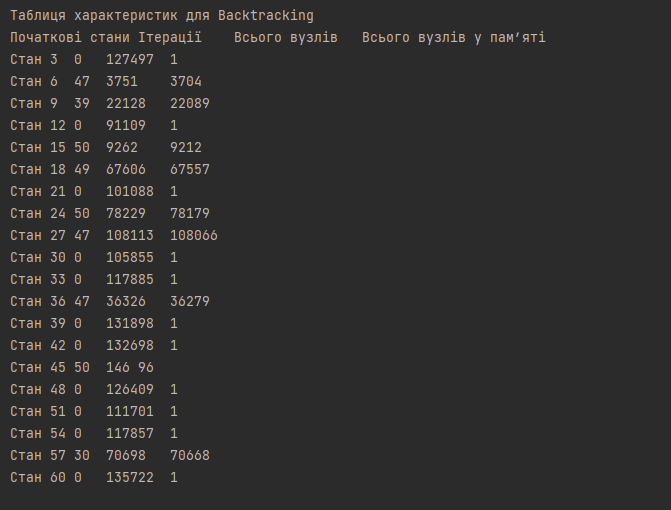


Рисунок 3.3 – Алгоритм Backtracking

## Дослідження алгоритмів

В таблиці 3.1 наведені характеристики оцінювання алгоритму BFS

, задачі 8 puzzle для 20 початкових станів.

Таблиця 3.1 – Характеристики оцінювання алгоритму BFS

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Початкові стани | Ітерації | К-сть гл. кутів | Всього вузлів | Всього вузлів у пам’яті |
| Стан 1 | 0 |  | 241920 | 184256 |
| Стан 4 | 23 |  | 166405 | 144184 |
| Стан 7 | 27 |  | 235855 | 183764 |
| Стан 10 | 0 |  | 241920 | 184214 |
| Стан 13 | 22 |  | 128124 | 114628 |
| Стан 16 | 27 |  | 238162 | 184125 |
| Стан 19 | 17 |  | 30444 | 29018 |
| Стан 22 | 18 |  | 40740 | 38649 |
| Стан 25 | 21 |  | 91610 | 83282 |
| Стан 28 | 0 |  | 241920 | 184256 |
| Стан 31 | 0 |  | 241920 | 184256 |
| Стан 34 | 17 |  | 27413 | 26205 |
| Стан 37 | 0 |  | 241920 | 184222 |
| Стан 40 | 0 |  | 241920 | 184105 |
| Стан 43 | 24 |  | 180559 | 154340 |
| Стан 46 | 0 |  | 241920 | 184125 |
| Стан 49 | 0 |  | 241920 | 184125 |
| Стан 52 | 0 |  | 241920 | 184227 |
| Стан 55 | 20 |  | 85543 | 78612 |
| Стан 58 | 0 |  | 241920 | 184215 |

В таблиці 3.2 наведені характеристики оцінювання алгоритму A\*, задачі 8 Puzzle для 20 початкових станів.

Таблиця 3.2 – Характеристики оцінювання алгоритму A\*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Початкові стани | Ітерації | К-сть гл. кутів | Всього вузлів | Всього вузлів у пам’яті |
| Стан 2 | 0 |  | 241920 | 183089 |
| Стан 5 | 23 |  | 19780 | 18959 |
| Стан 8 | 27 |  | 87612 | 79910 |
| Стан 11 | 0 |  | 241920 | 183277 |
| Стан 14 | 22 |  | 12921 | 12425 |
| Стан 17 | 27 |  | 89572 | 81206 |
| Стан 20 | 17 |  | 1701 | 1662 |
| Стан 23 | 18 |  | 2408 | 2356 |
| Стан 26 | 21 |  | 9100 | 8806 |
| Стан 29 | 0 |  | 241920 | 183295 |
| Стан 32 | 0 |  | 241920 | 183286 |
| Стан 35 | 17 |  | 1660 | 1618 |
| Стан 38 | 0 |  | 241920 | 183417 |
| Стан 41 | 0 |  | 241920 | 183312 |
| Стан 44 | 24 |  | 29871 | 28307 |
| Стан 47 | 0 |  | 241920 | 183054 |
| Стан 50 | 0 |  | 241920 | 183322 |
| Стан 53 | 0 |  | 241920 | 183354 |
| Стан 56 | 20 |  | 6389 | 6215 |
| Стан 59 | 0 |  | 241920 | 183246 |

В таблиці 3.3 наведені характеристики оцінювання алгоритму Backtracking , задачі 8 Puzzle для 20 початкових станів.

Таблиця 3.3 – Характеристики оцінювання алгоритму Backtracking

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Початкові стани | Ітерації | К-сть гл. кутів | Всього вузлів | Всього вузлів у пам’яті |
| Стан 3 | 0 |  | 127497 | 1 |
| Стан 6 | 47 |  | 3751 | 3704 |
| Стан 9 | 39 |  | 22128 | 22089 |
| Стан 12 | 0 |  | 91109 | 1 |
| Стан 15 | 50 |  | 9262 | 9212 |
| Стан 18 | 49 |  | 67606 | 67557 |
| Стан 21 | 0 |  | 101088 | 1 |
| Стан 24 | 50 |  | 78229 | 78179 |
| Стан 27 | 47 |  | 108113 | 108066 |
| Стан 30 | 0 |  | 105855 | 1 |
| Стан 33 | 0 |  | 117885 | 1 |
| Стан 36 | 47 |  | 36326 | 36279 |
| Стан 39 | 0 |  | 131898 | 1 |
| Стан 42 | 0 |  | 132698 | 1 |
| Стан 45 | 50 |  | 146 | 96 |
| Стан 48 | 0 |  | 126409 | 1 |
| Стан 51 | 0 |  | 111701 | 1 |
| Стан 54 | 0 |  | 117857 | 1 |
| Стан 57 | 30 |  | 70698 | 70668 |
| Стан 60 | 0 |  | 135722 | 1 |

Висновок

У рамках цієї лабораторної роботи було реалізовано три алгоритми для розв'язування класичної задачі про 8-пазл: BFS (Breadth-First Search), A\* (A-star) та Backtracking. Метою роботи було дослідити ефективність цих алгоритмів на різних початкових станах та порівняти їх характеристики, зокрема кількість ітерацій, кількість згенерованих вузлів та використану пам'ять.

**Результати експериментів:**

**Алгоритм BFS** показав стабільні результати для більшості початкових станів, але в деяких випадках страждав через великі вимоги до пам'яті та часу виконання. У таких випадках BFS може не встигнути знайти розв'язок через перевищення ліміту пам'яті або часу.

**Алгоритм A\*** продемонстрував кращі результати порівняно з BFS в плані кількості згенерованих вузлів, завдяки використанню евристики. Проте, як і BFS, він може потребувати великих обсягів пам'яті, що впливає на його ефективність при складних початкових станах.

**Алгоритм Backtracking** виявився найбільш залежним від обмеження глибини рекурсії (max\_depth). У багатьох випадках, якщо глибина була обмежена занадто низько, алгоритм не міг знайти розв'язок або використовував дуже мало пам'яті, оскільки часто повертався до попередніх станів. Збільшення глибини рекурсії дозволило покращити результати, але це також збільшило час виконання.

Критерії оцінювання

Максимальний бал за лабораторну роботу №2 дорівнює – 5.

Критерії оцінювання у відсотках від максимального балу:

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