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-21 Сергієнко Сергій* _____
(шифр, прізвище, ім'я, по батькові)

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

Київ 2023

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:

- **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 впорядкування 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 A*.
- **RBFS** – Рекурсивний пошук за першим найкращим співпадінням.
- **F1** – кількість пар ферзів, які б’ють один одного з урахуванням видимості (ферзь A може стояти на одній лінії з ферзем B, проте між ними стоїть ферзь C; тому A не б’є B).
- **F2** – кількість пар ферзів, які б’ють один одного без урахування видимості.
- **H1** – кількість фішок, які не стоять на своїх місцях.
- **H2** – Манхеттенська 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 T від часу роботи алгоритму t. Можна розглядати лінійну залежність: $T = 1000 - k \cdot t$, 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
22	8-puzzle	LDFS	RBFS		H2

3 Виконання

1.1 Псевдокод алгоритмів

Function Depth-Limited-Search (problem, limit) returns рішення result або індикатор невдачі failure\cutoff

    return Recursive-DLS(Make-Node(Initial-State[problem]), Problem, limit)

Function Recursive-DLS(node, problem, limit) returns рішення result або індикатор невдачі failure\cutoff

    cutoff_occurred? ← неправдиве значення
    if Goal-Test[problem](State[node]) **then return** Solution(node)
    else if Depth[node] = limit **then return** індикатор невдачі cutoff
    else for each спадкоємець successor in Expand(node, problem) do
        result ← **Recursive-DLS**(successor, problem, limit)
        if result = cutoff **then** cutoff_occured? ← правдиве значення
        else if result != failure **then** return рішення result
    if cutoff_occurred?
        then return індикатор невдачі cutoff
    else return індикатор невдачі failure

Function Recursive-Best-First-Search(problem) returns рішення result або індикатор невдачі failure

    RBFS(problem, Make-Node(Initial-State[problem]), ∞)

Function RBFS(problem, node, f_limit) returns рішення result або індикатор невдачі failure і нова межа f-вартості f_limit

    if Goal-Test[problem](State[node]) **then return** вузол node
    successors ← Expand(node, problem)
    if множина вузлів спадкоємців successors пуста
        then return failure, ∞

for each s **in** successors **do**

f[s] ← max(g(s)+h(s),f[node])

repeat

best ← вузол з найменшим f-значенням у множині successors

if f[best] > f_limit **then return** failure, f[best]

alternative ← наступне після найменшого f-значення у множині successors

result, f[best] ← **RBFS**(problem, best, min(f_limit, alternative))

if result ≠ failure **then return** result

1.2 Програмна реалізація

1.2.1 Вихідний код

```
import threading
from puzzle import *
from helpers import *
from memory_limiter import *
def main():
    file_name = "input.txt"
    memory_limit = 1024 * 1024 * 1024
    time_limit = 30 * 60

    while True:
        puzzle_choice = input("Random puzzle or from txt file?(1/0):")
        puzzle_choice = is_choice_num(puzzle_choice)
        if puzzle_choice is not None:
            break
    if puzzle_choice:
        initial_state = generate_puzzle()
    else:
        initial_state = read_matrix(file_name)
        if not is_solvable(initial_state):
            print(f"Puzzle read from file is not solvable")
            return
    print_matrix(initial_state)
    puzzle = Puzzle(initial_state)
```

```
        assign_job(create_job())
        limit_memory(memory_limit)
        try:
            while True:
                solution_choice = input("RBFS or LDFS?(1/0):")
                solution_choice = is_choice_num(solution_choice)
                if solution_choice is not None:
                    break
            if solution_choice:
                            search_thread  =  threading.Thread(target=run_search,
args=(puzzle, "RBFS"))
            else:
                while True:
                    limit = input("Enter LDFS depth limit:")
                    if validate_positive_integer(limit):
                        break
                            search_thread  =  threading.Thread(target=run_search,
args=(puzzle, "LDFS", int(limit)))

            search_thread.start()
            search_thread.join(time_limit)
            if search_thread.is_alive():
                print("Search exceeded time limit")
                os.abort()
        except (MemoryError, RuntimeError):
            print('Memory ran out')
        return 0


    main()

    class Node:
        def __init__(self, state, parent, action, depth):
            self.state = state
            self.parent = parent
            self.action = action
            self.depth = depth
            self.f = 0

    class Puzzle:
```

```
        def __init__(self, initial_state):
            self.initial_state = initial_state
            self.goal_state = [1, 2, 3, 4, 5, 6, 7, 8, 0]
            self.unique_states = set()
            self.actions = {"UP": -3, "DOWN": 3, "LEFT": -1, "RIGHT": 1}

        def goal_test(self, state):
            return state == self.goal_state

        def get_successors(self, state):
            successors = []
            zero_index = state.index(0)
            for action in self.actions:
                target_index = zero_index + self.actions[action]
                if (action in ["UP", "DOWN"] and 0 <= target_index < len(state))
or \
                        (action == "LEFT" and zero_index % 3 != 0) or \
                        (action == "RIGHT" and zero_index % 3 != 2):
                    new_state = state.copy()
                            new_state[zero_index], new_state[target_index] =
new_state[target_index], new_state[zero_index]
                    successors.append((new_state, action))
                    self.unique_states.add(tuple(new_state))

            return successors

    from abc import ABC, abstractmethod
    from node import *


    class SolvingMethod(ABC):
        def __init__(self, puzzle):
            self.puzzle = puzzle
            self.node_count = 0
            self.max_nodes_in_memory = 0

            super().__init__()

        @abstractmethod
        def search(self):
            pass
```

```
class DLS(SolvingMethod):
    def __init__(self, puzzle, limit):
        super().__init__(puzzle)
        self.limit = limit
        self.nodes_at_depth = [0] * (limit + 1)

    def search(self):
            return  self.recursive_dls(Node(self.puzzle.initial_state,  None,
"INIT", 0))

    def recursive_dls(self, node):
        self.node_count += 1
        cutoff_occurred = False
        if self.puzzle.goal_test(node.state):
                    self.max_nodes_in_memory  =  max(self.max_nodes_in_memory,
sum(self.nodes_at_depth))
            return node
        elif node.depth == self.limit:
                    self.max_nodes_in_memory  =  max(self.max_nodes_in_memory,
sum(self.nodes_at_depth))
            self.nodes_at_depth[node.depth] = 0
            return 'cutoff'
        else:
            children = self.puzzle.get_successors(node.state)
            self.nodes_at_depth[node.depth] = len(children)
            for child in children:
                  result = self.recursive_dls(Node(child[0], node, child[1],
node.depth + 1))
                if result == 'cutoff':
                    cutoff_occurred = True
                elif result is not None:
                    return result
            return 'cutoff' if cutoff_occurred else 'failure'


class RBFS(SolvingMethod):
    def __init__(self, puzzle):
        super().__init__(puzzle)
        self.current_nodes_in_memory = 0
```

```
        def search(self):
            start_node = Node(self.puzzle.initial_state, None, "INIT", 0)
            start_node.f = self.manhattan_distance(start_node.state)
            return self.rbfs(start_node, float('inf'))

        def rbfs(self, node, f_limit):
                        self.max_nodes_in_memory  =  max(self.max_nodes_in_memory,
self.current_nodes_in_memory)
            if self.puzzle.goal_test(node.state):
                return node, node.f
            successors = []
            children = self.puzzle.get_successors(node.state)
            if not children:
                return None, float('inf')
            for child in children:
                new_node = Node(child[0], node, child[1], node.depth + 1)
                                        new_node.f   =   max(new_node.depth   +
self.manhattan_distance(new_node.state), node.f)
                successors.append(new_node)
                self.node_count += 1
                self.current_nodes_in_memory += 1
            while True:
                successors.sort(key=lambda x: x.f)
                best = successors[0]
                if best.f > f_limit:
                    self.current_nodes_in_memory -= (len(successors) - 0)
                    return None, best.f
                     alternative = successors[1].f if len(successors) > 1 else
float('inf')
                result, best.f = self.rbfs(best, min(f_limit, alternative))
                if result is not None:
                    return result, best.f

        def manhattan_distance(self, state):
            distance = 0
            for i in range(1, 9):
                xs, ys = divmod(state.index(i), 3)
                xg, yg = divmod(self.puzzle.goal_state.index(i), 3)
                distance += abs(xs - xg) + abs(ys - yg)
            return distance
```

1.2.2 Приклади роботи

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

```
Random puzzle or from txt file?(1/0):0
[1, 2, 6]
[7, 5, 4]
[0, 8, 3]
RBFS or LDFS?(1/0):0
Enter LDFS depth limit:22
Solving started
      [1, 2, 6]
INIT  [7, 5, 4]
      [0, 8, 3]

      [1, 2, 6]
UP    [0, 5, 4]
      [7, 8, 3]

      [0, 2, 6]
UP    [1, 5, 4]
      [7, 8, 3]

      [1, 2, 6]
DOWN  [0, 5, 4]
      [7, 8, 3]

      [0, 2, 6]
UP    [1, 5, 4]
      [7, 8, 3]
```

```
      [1, 2, 6]
DOWN  [0, 5, 4]
      [7, 8, 3]

      [0, 2, 6]
UP    [1, 5, 4]
      [7, 8, 3]

      [2, 0, 6]
RIGHT [1, 5, 4]
      [7, 8, 3]

      [2, 6, 0]
RIGHT [1, 5, 4]
      [7, 8, 3]

      [2, 6, 4]
DOWN  [1, 5, 0]
      [7, 8, 3]

      [2, 6, 4]
DOWN  [1, 5, 3]
      [7, 8, 0]

      [2, 6, 4]
LEFT  [1, 5, 3]
      [7, 0, 8]
```

```
       [2, 6, 4]
UP     [1, 0, 3]
       [7, 5, 8]

       [2, 0, 4]
UP     [1, 6, 3]
       [7, 5, 8]

       [2, 4, 0]
RIGHT  [1, 6, 3]
       [7, 5, 8]

       [2, 4, 3]
DOWN   [1, 6, 0]
       [7, 5, 8]

       [2, 4, 3]
LEFT   [1, 0, 6]
       [7, 5, 8]

       [2, 0, 3]
UP     [1, 4, 6]
       [7, 5, 8]

       [0, 2, 3]
LEFT   [1, 4, 6]
       [7, 5, 8]
```

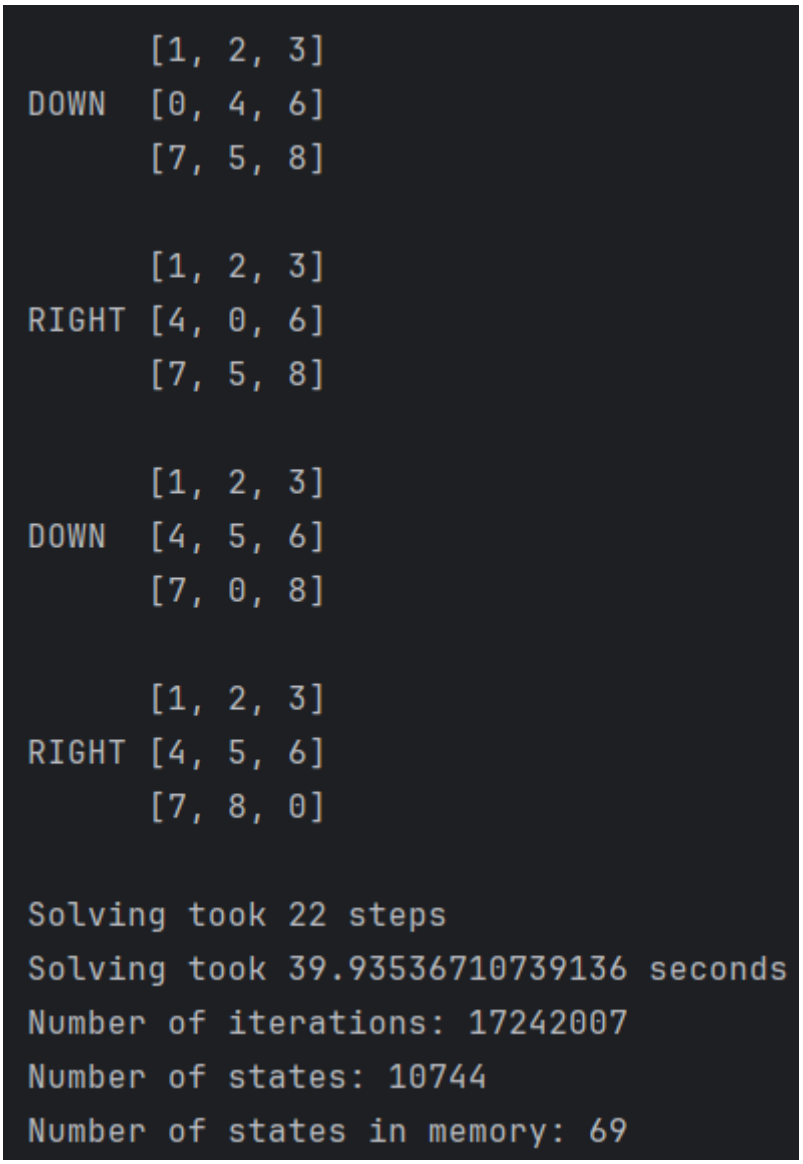

Рисунок 3.1 – Алгоритм LDFS

```
E:\KPI\3_\algorithms\lab_2\venv\Scripts
Random puzzle or from txt file?(1/0):0
[1, 2, 6]
[7, 5, 4]
[0, 8, 3]
RBFS or LDFS?(1/0):1
Solving started
      [1, 2, 6]
INIT  [7, 5, 4]
      [0, 8, 3]

      [1, 2, 6]
RIGHT [7, 5, 4]
      [8, 0, 3]

      [1, 2, 6]
RIGHT [7, 5, 4]
      [8, 3, 0]

      [1, 2, 6]
UP    [7, 5, 0]
      [8, 3, 4]

      [1, 2, 0]
UP    [7, 5, 6]
      [8, 3, 4]
```

```
       [1, 0, 2]
LEFT   [7, 5, 6]
       [8, 3, 4]

       [1, 5, 2]
DOWN   [7, 0, 6]
       [8, 3, 4]

       [1, 5, 2]
DOWN   [7, 3, 6]
       [8, 0, 4]

       [1, 5, 2]
RIGHT  [7, 3, 6]
       [8, 4, 0]

       [1, 5, 2]
UP     [7, 3, 0]
       [8, 4, 6]

       [1, 5, 2]
LEFT   [7, 0, 3]
       [8, 4, 6]

       [1, 5, 2]
DOWN   [7, 4, 3]
       [8, 0, 6]
```

```
      [1, 5, 2]
LEFT  [7, 4, 3]
      [0, 8, 6]

      [1, 5, 2]
UP    [0, 4, 3]
      [7, 8, 6]

      [1, 5, 2]
RIGHT [4, 0, 3]
      [7, 8, 6]

      [1, 0, 2]
UP    [4, 5, 3]
      [7, 8, 6]

      [1, 2, 0]
RIGHT [4, 5, 3]
      [7, 8, 6]

      [1, 2, 3]
DOWN  [4, 5, 0]
      [7, 8, 6]

      [1, 2, 3]
DOWN  [4, 5, 6]
      [7, 8, 0]
```

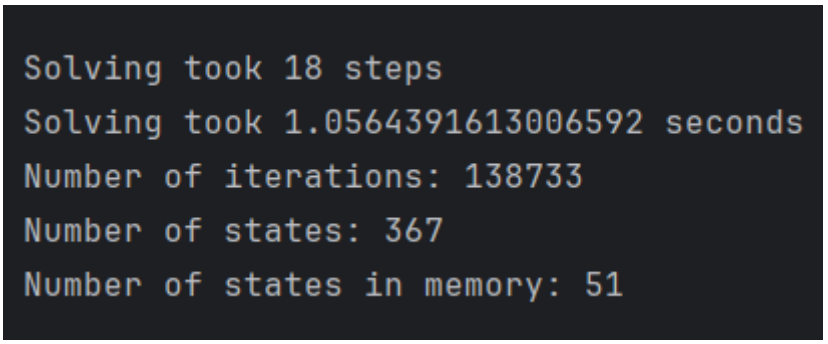

Рисунок 3.2 – Алгоритм RBFS

1.3 Дослідження алгоритмів

В таблиці 3.1 наведені характеристики оцінювання алгоритму LDFS задачі 8-puzzle для 20 початкових станів.

Таблиця 3.1 – Характеристики оцінювання алгоритму LDFS при обмеженні глибини = 22

Початкові стани	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пам'яті
[1, 2, 6] [7, 5, 4] [0, 8, 3]	17242007 вузлів 22 кроки	–	10744	69
[2, 6, 5] [0, 8, 1] [3, 4, 7]	–	Перевищено допустимий час	–	–
[7, 1, 5] [4, 3, 8] [2, 6, 0]	–	Перевищено допустимий час	–	–
[1, 8, 6] [3, 2, 0] [4, 7, 5]	127898581 вузлів 22 кроки	–	24114	73
[0, 7, 1] [2, 5, 8] [6, 4, 3]	–	Перевищено допустимий час	–	–
[3, 4, 5] [8, 7, 1] [0, 6, 2]	71389541 вузлів 22 кроки	–	19797	71
[8, 3, 5] [7, 1, 6] [2, 4, 0]	12473630 вузлів 22 кроки	–	9783	69
[1, 7, 5] [3, 2, 4] [8, 0, 6]	–	Перевищено допустимий час	–	–

[2, 1, 4] [8, 0, 3] [6, 7, 5]	120802936 вузлів 22 кроки		21386	77
[7, 0, 5] [4, 6, 2] [3, 1, 8]	–	Перевищено допустимий час	–	–
[2, 5, 6] [1, 3, 7] [4, 0, 8]	373125183 вузлів 21 крок	–	32004	77
[8, 3, 5] [4, 0, 7] [2, 1, 6]	–	Перевищено допустимий час	–	–
[5, 8, 3] [4, 2, 6] [1, 7, 0]	–	Перевищено допустимий час	–	–
[0, 4, 6] [2, 3, 7] [5, 1, 8]	107358686 вузлів 22 кроки	–	23721	71
[2, 3, 0] [4, 6, 5] [8, 1, 7]	53593373 вузлів 22 кроки	–	17341	71
[8, 0, 7] [4, 6, 2] [3, 1, 5]	–	Перевищено допустимий час	–	–
[3, 5, 0] [2, 6, 7] [1, 4, 8]	107318009 вузлів 22 кроки	–	23562	71

[3, 7, 5] [0, 2, 1] [4, 8, 6]	–	Перевищено допустимий час	–	–
[2, 8, 7] [3, 4, 0] [5, 6, 1]	–	Перевищено допустимий час	–	–
[1, 2, 8] [4, 5, 6] [0, 3, 7]	68580748 22 кроки	–	18438	71

Не враховуючи задачі, час розв’язання яких перевищує допустимий. Середня кількість вузлів: 105978269. Середня кількість кроків: 22. Середня кількість станів: 20089. Середня кількість станів у пам’яті: 72.

В таблиці 3.2 наведені характеристики оцінювання алгоритму RBFS задачі 8-puzzle для 20 початкових станів.

Таблиця 3.3 – Характеристики оцінювання алгоритму RBFS

Початкові стани	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пам'яті
[1, 2, 6] [7, 5, 4] [0, 8, 3]	138733 вузлів 18 кроків	–	367	51
[2, 6, 5] [0, 8, 1] [3, 4, 7]	554749 вузлів 25 кроків	–	2181	73
[7, 1, 5] [4, 3, 8] [2, 6, 0]	784009 вузлів 24 кроків	–	1903	68
[1, 8, 6] [3, 2, 0] [4, 7, 5]	4913 вузлів 17 кроків	–	235	51
[0, 7, 1] [2, 5, 8] [6, 4, 3]	222288 вузлів 24 кроки	–	2252	70
[3, 4, 5] [8, 7, 1] [0, 6, 2]	106 вузлів 18 кроків	–	69	53
[8, 3, 5] [7, 1, 6] [2, 4, 0]	762 вузлів 18 кроків	–	136	53
[1, 7, 5] [3, 2, 4] [8, 0, 6]	13750 вузлів 21 крок	–	479	63
[2, 1, 4]	1161 вузлів	–	145	53

[8, 0, 3] [6, 7, 5]	18 кроків			
[7, 0, 5] [4, 6, 2] [3, 1, 8]	279431 вузлів 23 кроки	–	1658	66
[2, 5, 6] [1, 3, 7] [4, 0, 8]	8050 вузлів 17 кроків	–	159	47
[8, 3, 5] [4, 0, 7] [2, 1, 6]	776 вузлів 20 кроків	–	155	58
[5, 8, 3] [4, 2, 6] [1, 7, 0]	1442339 вузлів 23 кроки	–	1684	61
[0, 4, 6] [2, 3, 7] [5, 1, 8]	217 вузлів 18 кроків	–	134	53
[2, 3, 0] [4, 6, 5] [8, 1, 7]	10162 вузлів 18 кроків	–	260	51
[8, 0, 7] [4, 6, 2] [3, 1, 5]	67757 вузлів 25 кроків	–	1255	71
[3, 5, 0] [2, 6, 7] [1, 4, 8]	757 вузлів 18 кроків	–	182	49
[3, 7, 5] [0, 2, 1]	1970 вузлів 19 кроків	–	255	56

[4, 8, 6]				
[2, 8, 7] [3, 4, 0] [5, 6, 1]	142578 вузлів 27 кроків	–	2771	74
[1, 2, 8] [4, 5, 6] [0, 3, 7]	5920 вузлів 18 кроків	–	160	53

Середня кількість вузлів: 184020. Середня кількість кроків: 19,5. Середня кількість глухих кутів: 0. Середня кількість станів: 823. Середня кількість станів у пам’яті: 59.

Висновок

При виконанні даної лабораторної роботи було розглянуто розв’язання задачі 8-puzzle алгоритмом неінформативного пошуку LDFS та алгоритмом інформативного пошуку RBFS з застосуванням евристичної функції Манхеттенська відстань.

Внаслідок дослідження алгоритм пошуку вглиб з обмеженням глибини є неефективним для розв’язання задачі 8-puzzle. Оскільки цей алгоритм неінформативний, то він просто перебирає варіанти, поки один з них не буде цільовим. Це зумовлює високу часову складність, що при аналізі випливало у перевищення допустимого часу. Більш того, у цьому алгоритмі виникає додаткове джерело неповноти, адже з обмеженням глибини меншим, ніж глибина найбільш поверхневого цільового вузла, алгоритм перебере усі вузли і не знайде рішення. Якщо ж ліміт обрати більшим за глибину найбільш поверхневого цільового вузла виникне додаткове джерело неоптимальності – алгоритм досліджуватиме більше вузлів, ніж потрібно. Серед плюсів можна відокремити лише низький рівень просторової складності, дослідження підтвердили, що використання пам’яті зростає лінійно.

Алгоритм RBFS з застосуванням евристичної функції Манхеттенська відстань довів наскільки інформативний пошук краще неінформативного. По-перше, на відміну від RBFS, LDFS не зміг розв’язати кожен приклад задачі 8-puzzle через перевищення допустимого часу, хоча й обмеження глибини було в 22 вузли. По-друге, ті задачі, що LDFS розв’язував за 22 кроки RBFS розв’язував за 18-19 кроків. По-третє, оскільки RBFS оцінює найбільш перспективні вузли для подальшого дослідження, він генерував набагато менше вузлів та станів, ніж LDFS. Використання пам’яті, як і в LDFS було лінійним і дорівнювало O(bd), де b – коефіцієнт розгалуження, d – глибина найбільш поверхневого цільового вузла.