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 **3**

2 Завдання **4**

3 Виконання **7**

3.1 Псевдокод алгоритмів 7

3.2 Програмна реалізація 8

3.2.1 Вихідний код *8*

3.2.2 Приклади роботи *13*

3.3 Дослідження алгоритмів 20

Висновок **27**

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}
              self.past_actions = {"UP": "DOWN", "DOWN": "UP", "LEFT": "RIGHT",
"RIGHT": "LEFT"}

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

        def get_successors(self, state, past_action):
            successors = []
            zero_index = state.index(0)
            for action in self.actions:
                                if past_action != "INIT" and action ==
self.past_actions[past_action]:
                    continue
                target_index = zero_index + self.actions[action]
                # перевіряємо чи після свапу "0" не буде за межами пазлу
                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.iteration_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):
            cutoff_occurred = False
             if self.puzzle.goal_test(node.state):  # досягнуто цільовий стан -
завершуємо рекурсію
                    self.max_nodes_in_memory  =  max(self.max_nodes_in_memory,
sum(self.nodes_at_depth))
                return node
               elif node.depth == self.limit:  # досягнуто обмеження глибини -
повертаємось назад
                    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:
                self.iteration_count += 1
                children = self.puzzle.get_successors(node.state, node.action)
                self.nodes_at_depth[node.depth] = len(children)
                for child in children:  # перебираємо нащадків вузла
                      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):  # досягнуто цільовий стан -
завершуємо рекурсію
                return node, node.f
            successors = []
            children = self.puzzle.get_successors(node.state, node.action)
            self.iteration_count += 1
            if not children:
                return None, float('inf')
            for child in children:
                # перебираємо нащадків, визначаємо для них f-значення
                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.current_nodes_in_memory += 1
            while True:
                  # обираємо нащадка з найменшим f-значенням та як альтернативу
наступного після нього
                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(0, 9):  # перебираємо цифри
                # divmod() повертає частку та остачу
                   xs, ys = divmod(state.index(i), 3)  # частка - ряд цифри у
пазлі, остача - колонка цифри у пазлі
                 xg, yg = divmod(self.puzzle.goal_state.index(i), 3)  # те саме,
тільки у цільовому стані
                 distance += abs(xs - xg) + abs(ys - yg)  # сама манхеттенська
відстань
            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]

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

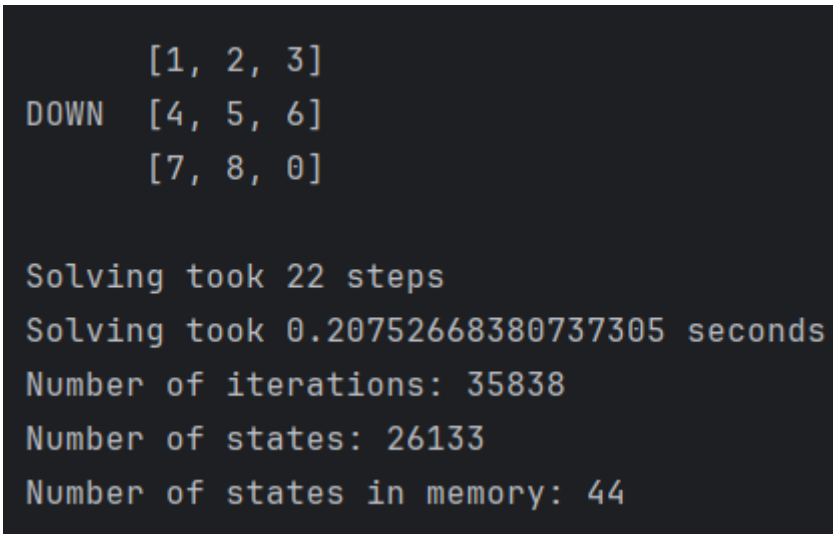

Рисунок 3.1 – Алгоритм LDFS

```
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]
```

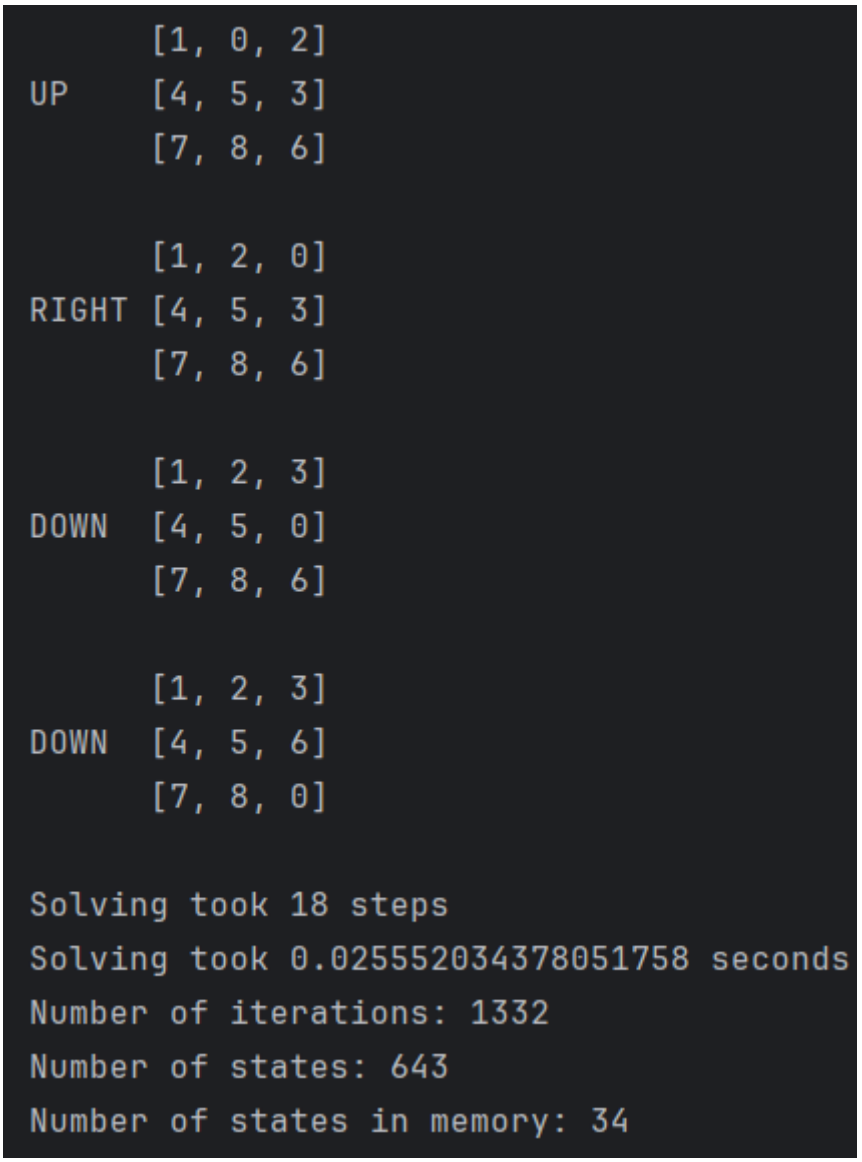

Рисунок 3.2 – Алгоритм RBFS

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 = 31

Початкові стани	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пам'яті
[1, 2, 6] [7, 5, 4] [0, 8, 3]	526428 30 кроків	–	109562	61
[2, 6, 5] [0, 8, 1] [3, 4, 7]	833412 31 крок	–	133201	62
[7, 1, 5] [4, 3, 8] [2, 6, 0]	275861 30 кроків	–	83846	61
[1, 8, 6] [3, 2, 0] [4, 7, 5]	799769 31 крок	–	130549	62
[0, 7, 1] [2, 5, 8] [6, 4, 3]	987878 30 кроків	–	134180	61
[3, 4, 5] [8, 7, 1] [0, 6, 2]	1353911 30 кроків	–	150420	61
[8, 3, 5] [7, 1, 6] [2, 4, 0]	38908 30 кроків	–	26287	61
[1, 7, 5] [3, 2, 4] [8, 0, 6]	31940 31 крок	–	23674	62

[2, 1, 4] [8, 0, 3] [6, 7, 5]	496908 30 кроків		110642	63
[7, 0, 5] [4, 6, 2] [3, 1, 8]	480213 27 кроків	–	114560	62
[2, 5, 6] [1, 3, 7] [4, 0, 8]	16683 31 крок	–	14603	62
[8, 3, 5] [4, 0, 7] [2, 1, 6]	1954135 30 кроків	–	158553	63
[5, 8, 3] [4, 2, 6] [1, 7, 0]	275193 30 кроків	–	83761	61
[0, 4, 6] [2, 3, 7] [5, 1, 8]	166431 28 кроків	–	64644	61
[2, 3, 0] [4, 6, 5] [8, 1, 7]	210186 30 кроків	–	72806	61
[8, 0, 7] [4, 6, 2] [3, 1, 5]	23055 31 крок	–	18947	62
[3, 5, 0] [2, 6, 7] [1, 4, 8]	168695 30 кроків	–	65223	61
[3, 7, 5]	193399	–	76658	62

[0, 2, 1] [4, 8, 6]	31 крок			
[2, 8, 7] [3, 4, 0] [5, 6, 1]	1813434 29 кроків	–	160301	62
[1, 2, 8] [4, 5, 6] [0, 3, 7]	131479 30 кроків	–	58205	61

Середня кількість ітерацій: 538896. Середня кількість кроків: 30. Середня кількість глухих кутів: 0. Середня кількість станів: 89531. Середня кількість станів у пам’яті: 61.

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]	1332 18 кроків	–	643	34
[2, 6, 5] [0, 8, 1] [3, 4, 7]	12956 25 кроків	–	4572	49
[7, 1, 5] [4, 3, 8] [2, 6, 0]	19569 24 кроків	–	6554	45
[1, 8, 6] [3, 2, 0] [4, 7, 5]	310 17 кроків	–	252	35
[0, 7, 1] [2, 5, 8] [6, 4, 3]	2936 24 кроки	–	1610	47
[3, 4, 5] [8, 7, 1] [0, 6, 2]	119 18 кроків	–	145	36
[8, 3, 5] [7, 1, 6] [2, 4, 0]	327 18 кроків	–	295	36
[1, 7, 5] [3, 2, 4] [8, 0, 6]	1704 21 крок	–	1080	43
[2, 1, 4]	706	–	534	36

[8, 0, 3] [6, 7, 5]	18 кроків			
[7, 0, 5] [4, 6, 2] [3, 1, 8]	3744 23 кроки	–	1835	46
[2, 5, 6] [1, 3, 7] [4, 0, 8]	769 17 кроків	–	450	31
[8, 3, 5] [4, 0, 7] [2, 1, 6]	573 20 кроків	–	484	39
[5, 8, 3] [4, 2, 6] [1, 7, 0]	6949 23 кроки	–	2839	42
[0, 4, 6] [2, 3, 7] [5, 1, 8]	580 18 кроків	–	462	36
[2, 3, 0] [4, 6, 5] [8, 1, 7]	1612 18 кроків	–	764	34
[8, 0, 7] [4, 6, 2] [3, 1, 5]	4677 25 кроків	–	2556	47
[3, 5, 0] [2, 6, 7] [1, 4, 8]	366 18 кроків	–	371	32
[3, 7, 5] [0, 2, 1]	1451 19 кроків	–	827	38

[4, 8, 6]				
[2, 8, 7] [3, 4, 0] [5, 6, 1]	8454 27 кроків	–	5021	54
[1, 2, 8] [4, 5, 6] [0, 3, 7]	614 18 кроків	–	397	36

Середня кількість вузлів: 3487. Середня кількість кроків: 20 Середня кількість глухих кутів: 0. Середня кількість станів: 1585. Середня кількість станів у пам’яті: 41.

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 величезну, близьку до максимальної, кількість станів. 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, оскільки RBFS оцінює найбільш перспективні вузли для подальшого дослідження, він проходив набагато менше ітерацій та генерував менше станів, ніж LDFS. По-друге, середня кількість кроків для розв’язку задачі за допомогою LDFS була близькою до обмеження глибини, що вказує на неоптимальність LDFS. У RBFS середня кількість кроків до розв’язку в тих самих задачах була відчутно меншою. По-третє, хоч для 8-puzzle це не критично, та RBFS все ж зберігав менше станів у пам’яті. Однак, як і в LDFS використання пам’яті було лінійним і дорівнювало O(bd), де b – коефіцієнт розгалуження, d – глибина найбільш поверхневого цільового вузла.