

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)

ІІ-ІІ, Книш Д.О.
(шифр, прізвище, ім'я, по батькові)

Перевірів

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

Київ 2022

ЗМІСТ

1. МЕТА ЛАБОРАТОРНОЇ РОБОТИ.....	3
2. ЗАВДАННЯ	4
3. ВИКОНАННЯ.....	8
3.1. ПСЕВДОКОД АЛГОРИТМІВ	8
3.2. ПРОГРАМНА РЕАЛІЗАЦІЯ	8
3.2.1. Вихідний код	8
3.2.2. Приклади роботи	18
3.3. ДОСЛІДЖЕННЯ АЛГОРИТМІВ	19
ВИСНОВОК	22
КРИТЕРІЇ ОЦІНЮВАННЯ.....	23

1.

МЕТА ЛАБОРАТОРНОЇ РОБОТИ

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

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 відрізнятися 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 A*.

- **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 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	АП	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

3.

ВИКОНАННЯ

3.1. Псевдокод алгоритмів

```
IDS()
begin
    limit = 0
    while not LDFS(root, limit) do
        begin
            limit := limit + 1
        end
    end
end

LDFS(node, limit)
begin
    last_node = node
    if solved for node then return true
    if node's depth < limit then
        begin
            expand node
            for i := 0 to number of node's children do
                if LDFS(i-th children, limit) then return true else return false
            end
            return false
        end
    end
end

AStar()
begin
    opened := PriorityQueue()
    closed := HashSet()
    put root into opened
    while opened is not empty do
        begin
            top := first element of opened
            if solved for top then break
            add top to closed
            expand top
            for i := 0 to number of top's children do
                begin
                    if i-th children is in closed then continue
                    put i-th children into opened
                end
            end
        end
    end
end
```

3.2. Програмна реалізація

3.2.1. Вихідний код

IDS

tree.py

```
from copy import deepcopy
from node import Node
from board import Board, Optional
from helpers.logger import NQLogger
```

```
class NQueens:
    def __init__(self, queens: int, board: Optional[Board] = None) ->
None:
    self.memory_states: int = 1
```



```

        self.total_states: int = 1
        self.iter: int = 0
        self.size = queens
        self.last_node: Node
        self.root = Node(queens=queens, other=board if board else None)

    def IDS(self):
        NQLogger.info("** IDS Algorithm **")

        limit = 1
        while not self._LDFS(self.root, limit):
            limit += 1

        # print info
        self.info()

        return True

    def _LDFS(self, node: Node, limit: int):
        self.iter += 1
        self.last_node = deepcopy(node)

        if (node.is_solved()):
            NQLogger.info("** IDS Solved **")

            print("Solved board:")
            node.board.print()

            print("bounds:")
            print(f"    :- depth: {node.depth}")
            print(f"    :- limit: {limit}\n")

            return True

        if node.depth < limit:
            NQLogger.info(f"#{self.iter}: Expand with {len(node.children)}
child nodes")

            node.expand()
            self.total_states += len(node.children)

            for i in range(len(node.children)):
                if self._LDFS(node.children[i], limit):
                    self.memory_states += len(node.children)
                    return True

            return False

    def info(self, show_depth: bool = False):
        print("total:")
        print(f"    :- iterations: {self.iter}")
        print(f"    :- states: {self.total_states}")
        print(f"    :- memory states: {self.memory_states}")

        if show_depth:
            print(f"    :- depth: {self.last_node.depth}")

        print()

```

node.py

```

from typing import Any, Optional
from copy import deepcopy
from board import Board

```

```

class Node:
    def __init__(self, queens: Optional[int] = None, other=None) -> None:
        self.depth: int
        self.board: Board
        self.children: list[Any]

        if other and isinstance(other, Node):
            self.depth = other.depth + 1
            self.board = Board(other=other.board)
            self.children = [None] * (self.board.size * (self.board.size -
1))

        elif other and isinstance(other, Board):
            self.depth = 1
            self.board = deepcopy(other)
            self.children = [None] * (self.board.size * (self.board.size -
1))

        elif not other:
            self.depth = 1
            self.board = Board(queens=queens) # create empty board
            self.board.generate_board()
            self.children = [None] * (self.board.size * (self.board.size -
1))

    def is_solved(self):
        return self.board.conflict_number() == 0

    def expand(self):
        row = 0
        shift = 1

        for i in range(len(self.children)):
            if shift == self.board.size:
                row += 1
                shift = 1

            cp = deepcopy(self)

            self.children[i] = Node(other=cp)
            self.children[i].board.move_figure(row, shift)

            shift += 1

```

board.py

```

from typing import Literal, Optional
from random import randint
from copy import deepcopy
from helpers.logger import NQLogger

class Board:
    def __init__(self, queens: Optional[int] = None, other=None) -> None:
        self.size: int
        self.matrix: list[list[Literal[0, 1]]]

        if other:
            # debug: initial board
            if isinstance(other, Board):
                self.size = other.size
                self.matrix = deepcopy(other.matrix)

```

```

        elif isinstance(other, list):
            self.size = len(other)
            self.matrix = other

    # create empty board
    else:
        self.size = queens or 4
        self.matrix = [[0 for i in range(self.size)] for j in
range(self.size)]

    self.conf = self.conflict_number()

def generate_board(self):
    # populate with random Q placement
    for i in range(self.size):
        j = randint(0, self.size - 1)
        self.matrix[i][j] = 1

def conflict_number(self):
    conflicts = 0
    for i in range(self.size):
        for j in range(self.size):
            if self.matrix[i][j] == 1:
                conflicts += self.get_conflict(i, j)

    return conflicts

def correct_index(self, i, j):
    return i >= 0 and i < self.size and j >= 0 and j < self.size

def get_conflict(self, i, j):
    conf_number = 0

    for col in range(j):
        count = 0
        if self.matrix[i][col] == 1:
            count += 1

        if count:
            conf_number += 1
            break

    for col in range(j + 1, self.size):
        count = 0
        if self.matrix[i][col] == 1:
            count += 1

        if count:
            conf_number += 1
            break

    for row in range(i):
        count = 0
        if self.matrix[row][j] == 1:
            count += 1

        if count:
            conf_number += 1
            break

    for row in range(i + 1, self.size):
        count = 0
        if self.matrix[row][j] == 1:
            count += 1

```

```

        if count:
            conf_number += 1
            break

    row = i - 1
    col = j - 1

    while self.correct_index(row, col):
        if self.matrix[row][col] == 1:
            conf_number += 1
            break

        row -= 1
        col -= 1

    row = i + 1
    col = j + 1

    while self.correct_index(row, col):
        if self.matrix[row][col] == 1:
            conf_number += 1
            break

        row += 1
        col += 1

    row = i - 1
    col = j + 1

    while self.correct_index(row, col):
        if self.matrix[row][col] == 1:
            conf_number += 1
            break

        row -= 1
        col += 1

    row = i + 1
    col = j - 1

    while self.correct_index(row, col):
        if self.matrix[row][col] == 1:
            conf_number += 1
            break

        row += 1
        col -= 1

    return conf_number

def move_figure(self, row: int, shift: int):
    # for logging
    new_row = 0
    new_col = 0
    last_j = 0

    for j in range(self.size):
        if self.matrix[row][j] == 1:
            self.matrix[row][j] = 0

            col = j + shift
            if col >= self.size:
                col -= self.size

```

```

        self.matrix[row][col] = 1

        # for logging
        [new_row, new_col, last_j] = [row, col, j]
        break

    self.conf = self.conflict_number()

    # log the move
    NQLogger.info(f"Move Q: ({row},{last_j}) -> ({new_row},{new_col})
:: {self.conf} conflicts after moving")

def print(self, pre: str | None = None, end: str | None = '\n') ->
None:
    if pre:
        print(pre)

    def print_black_cell(q: Literal['Q', ' ']) -> None:
        print(f"\033[100m \033[97m{q} \033[0m", end="")

    def print_white_cell(q: Literal['Q', ' ']) -> None:
        print(f'\033[47m \033[30m{q} \033[0m', end="")

    for i in range(len(self.matrix)):
        print("|", end="")

        for j in range(len(self.matrix[i])):
            q = 'Q' if self.matrix[i][j] == 1 else ' '

            if (i % 2 == 0):
                if (j % 2 == 0):
                    print_white_cell(q)
                else:
                    print_black_cell(q)
            else:
                if (j % 2 == 0):
                    print_black_cell(q)
                else:
                    print_white_cell(q)

        print("|")
    print(end, end='')

```

A*

tree.py

```

from queue import PriorityQueue
from typing import Optional
from node import Node
from board import Board
from helpers.logger import NQLogger

class NQueens:
    def __init__(self, queens: int, board: Optional[Board] = None) ->
None:
        self.memory_states: int = 1
        self.total_states: int = 1
        self.iter: int = 0
        self.size = queens

        # initial `other` may be given manually

```

```

        self.root = Node(queens=queens, other=board)

    def AStar(self):
        NQLogger.info("*** A* Algorithm ***")

        # priority queue that uses pre-defined heuristic function
        implemented in node's comparator
        opened: PriorityQueue[Node] = PriorityQueue()
        closed: set[Board] = set()

        opened.put(self.root)
        NQLogger.info("Put root into queue")

        while not opened.empty():
            top = opened.get()

            if top.is_solved():
                NQLogger.info("*** A* Solved ***")

                print("Solved board:")
                top.board.print()

                self.total_states = opened.qsize() + len(closed)
                self.memory_states = opened.qsize() + len(closed)

                self.info()
                break

            closed.add(top.board)

            NQLogger.info(f"#{self.iter}: Expand with {len(top.children)}
successors")
            top.expand()

            successors: list[Node] = top.children

            for i in range(len(successors)):
                if successors[i].board in closed:
                    continue

            opened.put(successors[i])

            self.iter += 1

    def info(self):
        print("total:")
        print(f"    :- iterations: {self.iter}")
        print(f"    :- states: {self.total_states}")
        print(f"    :- memory states: {self.memory_states}\n")

```

node.py

```

from typing import Any, Optional
from copy import deepcopy
from board import Board

```

```

class Node:
    def __init__(self, queens: Optional[int] = None, other=None) -> None:
        self.depth: int
        self.board: Board
        self.children: list[Any]

        if other and isinstance(other, Node):
            self.depth = other.depth + 1

```

```

        self.board = Board(other=other.board)
        self.children = [None] * (self.board.size * (self.board.size -
1))

        elif other and isinstance(other, Board):
            self.depth = 1
            self.board = deepcopy(other)
            self.children = [None] * (self.board.size * (self.board.size -
1))

        elif not other:
            self.depth = 1
            self.board = Board(queens=queens) # create empty board
            self.board.generate_board()
            self.children = [None] * (self.board.size * (self.board.size -
1))

    @property
    def cost(self):
        _g = self.depth
        _h = self.board.conflict_number()
        return _g + _h

    # Comparator for priority queue
    def __lt__(self, node):
        return self.cost < node.cost

    def is_solved(self):
        return self.board.conflict_number() == 0

    def expand(self):
        row = 0
        shift = 1

        for i in range(len(self.children)):
            if shift == self.board.size:
                row += 1
                shift = 1

            cp = deepcopy(self)

            self.children[i] = Node(other=cp)
            self.children[i].board.move_figure(row, shift)

            shift += 1

```

board.py

```

from typing import Literal, Optional
from random import randint
from copy import deepcopy
from helpers.logger import NQLogger

class Board:
    def __init__(self, queens: Optional[int] = None, other=None) -> None:
        self.size: int
        self.matrix: list[list[Literal[0, 1]]]

        if other:
            # debug: initial board
            if isinstance(other, Board):
                self.size = other.size
                self.matrix = deepcopy(other.matrix)

```

```

        elif isinstance(other, list):
            self.size = len(other)
            self.matrix = other

    # create empty board
    else:
        self.size = queens or 4
        self.matrix = [[0 for i in range(self.size)] for j in
range(self.size)]

    self.conf = self.conflict_number()

def generate_board(self):
    # populate with random Q placement
    for i in range(self.size):
        j = randint(0, self.size - 1)
        self.matrix[i][j] = 1

def conflict_number(self):
    raw_pairs = set()
    for i in range(self.size):
        for j in range(self.size):
            if self.matrix[i][j] == 1:
                self.get_conflict(i, j, raw_pairs)

    # eliminate repeated pairs
    pairs = set()
    for pair in raw_pairs:
        rev = tuple(reversed(pair))
        if rev not in pairs:
            pairs.add(pair)

    # return the number of pairs as a conflicts number
    return len(pairs)

def correct_index(self, i, j):
    return i >= 0 and i < self.size and j >= 0 and j < self.size

def get_conflict(self, i, j, seen_pairs: set):

    for col in range(abs(j - 1), 0):
        if self.matrix[i][col] == 1:
            seen_pairs.add((i, j), (i, col))
            seen_pairs.add((i, col), (i, j))
            break

    for col in range(j + 1, self.size):
        if self.matrix[i][col] == 1:
            seen_pairs.add((i, j), (i, col))
            seen_pairs.add((i, col), (i, j))
            break

    for row in range(abs(i - 1), 0):
        if self.matrix[row][j] == 1:
            seen_pairs.add((i, j), (row, j))
            seen_pairs.add((row, j), (i, j))
            break

    for row in range(i + 1, self.size):
        if self.matrix[row][j] == 1:
            seen_pairs.add((i, j), (row, j))
            seen_pairs.add((row, j), (i, j))
            break

```



```

row = i - 1
col = j - 1

while self.correct_index(row, col):
    if self.matrix[row][col] == 1:
        seen_pairs.add((i, j), (row, col))
        seen_pairs.add((row, col), (i, j))
        break

    row -= 1
    col -= 1

row = i + 1
col = j + 1

while self.correct_index(row, col):
    if self.matrix[row][col] == 1:
        seen_pairs.add((i, j), (row, col))
        seen_pairs.add((row, col), (i, j))
        break

    row += 1
    col += 1

row = i - 1
col = j + 1

while self.correct_index(row, col):
    if self.matrix[row][col] == 1:
        seen_pairs.add((i, j), (row, col))
        seen_pairs.add((row, col), (i, j))
        break

    row -= 1
    col += 1

row = i + 1
col = j - 1

while self.correct_index(row, col):
    if self.matrix[row][col] == 1:
        seen_pairs.add((i, j), (row, col))
        seen_pairs.add((row, col), (i, j))
        break

    row += 1
    col -= 1

def move_figure(self, row: int, shift: int):
    # for logging
    new_row = 0
    new_col = 0
    last_j = 0

    for j in range(self.size):
        if self.matrix[row][j] == 1:
            self.matrix[row][j] = 0

            col = j + shift
            if col >= self.size:
                col -= self.size

            self.matrix[row][col] = 1

```

```

        # for logging
        [new_row, new_col, last_j] = [row, col, j]
        break

    self.conf = self.conflict_number()

    # log the move
    NQLogger.info(f"Move Q: ({row},{last_j}) -> ({new_row},{new_col})
:: {self.conf} conflicts after moving")

def print(self, pre: str | None = None, end: str | None = '\n') ->
None:
    if pre:
        print(pre)

    def print_black_cell(q: Literal['Q', ' ']) -> None:
        print(f"\033[100m \033[97m{q} \033[0m", end="")

    def print_white_cell(q: Literal['Q', ' ']) -> None:
        print(f'\033[47m \033[30m{q} \033[0m', end="")

    for i in range(len(self.matrix)):
        print("|", end="")

        for j in range(len(self.matrix[i])):
            q = 'Q' if self.matrix[i][j] == 1 else ' '

            if (i % 2 == 0):
                if (j % 2 == 0):
                    print_white_cell(q)
                else:
                    print_black_cell(q)

            else:
                if (j % 2 == 0):
                    print_black_cell(q)
                else:
                    print_white_cell(q)

        print("|")
    print(end, end='')

```

3.2.2. Приклади роботи

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

```

knysh@MacBook-Pro:~/FICT/3term/algos/lab2/python/ids
+ ids git:(main) python nqueens.py -q 6
Generated 6x6 board:
  Q  .  .  .  .  .
  .  Q  .  .  .  .
  .  .  Q  .  .  .
  .  .  .  Q  .  .
  .  .  .  .  Q  .
  .  .  .  .  .  Q
  :- conflicts: 8

Solved board:
  Q  .  .  .  .  .
  .  Q  .  .  .  .
  .  .  Q  .  .  .
  .  .  .  Q  .  .
  .  .  .  .  Q  .
  .  .  .  .  .  Q

bounds:
  :- depth: 5
  :- limit: 5

total:
  :- iterations: 35067
  :- states: 35131
  :- memory states: 6241

--- elapsed in 396.24 seconds ---
** logged to nq.log **

```

Рис. 3.1 — Результат виконання алгоритма IDS для дошки 6x6

```

knysh@MacBook-Pro:~/FICT/3term/algos/lab2/python/astar
+ astar git:(main) python nqueens.py -q 8
Generated 8x8 board:
  .  .  .  .  .  .  .  Q
  Q  .  .  .  .  .  .  .
  .  Q  .  .  .  .  .  .
  .  .  Q  .  .  .  .  .
  .  .  .  Q  .  .  .  .
  .  .  .  .  Q  .  .  .
  .  .  .  .  .  Q  .  .
  .  Q  .  .  .  .  .  .
  :- conflicts: 6

Solved board:
  .  .  .  .  .  .  .  Q
  Q  .  .  .  .  .  .  .
  .  Q  .  .  .  .  .  .
  .  .  Q  .  .  .  .  .
  .  .  .  Q  .  .  .  .
  .  .  .  .  Q  .  .  .
  .  .  .  .  .  Q  .  .
  .  Q  .  .  .  .  .  .

total:
  :- iterations: 27
  :- states: 1512
  :- memory states: 1485

--- elapsed in 13.91 seconds ---
** logged to nq.log **

```

Рис. 3.2 — Результат виконання алгоритма A* для дошки 8x8

3.3. Дослідження алгоритмів

В таблиці 3.1 наведені характеристики оцінювання алгоритму IDS, задачі 8-ферзів для 20 початкових станів.

Таблиця 3.1

Початкові стани	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пом'яті
Стан 1 : 1x1, 0к	1	0	1	1

Стан 2 : 4x4, 6к	54	0	61	25
Стан 3 : 4x4, 4к	391	0	409	37
Стан 4 : 4x4, 6к	917	0	925	37
Стан 5 : 5x5, 8к	122	0	141	41
Стан 6 : 5x5, 6к	190	0	201	41
Стан 7 : 5x5, 6к	83	0	101	41
Стан 8 : 6x6, 8к	123613	0	123661	121
Стан 9 : 6x6, 8к	807	0	811	61
Стан 10 : 6x6, 8к	3940	0	3991	91
Стан 11 : 7x7, 6к	11876	0	11929	127
Стан 12 : 7x7, 8к	4341	0	4411	127
Стан 13 : 7x7, 6к	681	0	715	85
Стан 14 : 8x8, 8к	10471	0	10585	169
Стан 15 : 8x8, 8к	88543	0	88593	169
Стан 16 : 8x8, 8к	18012	0	18089	169
Стан 17 : 9x9, 8к	11650	0	11809	217
Стан 18 : 9x9, 8к	59072	1	59185	1
Стан 19 : 10x10, 10к	38865	1	38971	1
Стан 20 : 11x11, 14к	21872	1	22001	1

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

Таблиця 3.2

Початкові стани	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пом'яті
Стан 1 : 1x1, 0к	0	0	0	0
Стан 2 : 4x4, 4к	2	0	24	24

Стан 3 : 4x4, 3к	3	0	36	36
Стан 4 : 4x4, 4к	6	0	72	72
Стан 5 : 5x5, 5к	3	0	60	60
Стан 6 : 5x5, 4к	6	0	120	120
Стан 7 : 5x5, 4к	2	0	40	40
Стан 8 : 6x6, 4к	92	0	2760	2760
Стан 9 : 6x6, 5к	22	0	660	660
Стан 10 : 6x6, 5к	10	0	300	300
Стан 11 : 7x7, 7к	4	0	168	168
Стан 12 : 7x7, 5к	54	0	2268	2268
Стан 13 : 7x7, 4к	35	0	1470	1470
Стан 14 : 8x8, 6к	4	0	244	244
Стан 15 : 8x8, 8к	18	0	1008	1008
Стан 16 : 8x8, 6к	244	0	13664	13664
Стан 17 : 9x9, 8к	102	0	7344	7344
Стан 18 : 9x9, 9к	97	0	6984	6984
Стан 19 : 10x10, 7к	179	0	16110	16110
Стан 20 : 11x11, 10к	246	0	27060	27060

В таблиці 3.3 наведені середні значення характеристик оцінювання алгоритмів IDS та A*, задачі 8-ферзів.

Таблиця 3.3

Алгоритм	сер. Ітерації	сер. к-сть гл. кутів	сер. станів	сер. станів у пом'яті
IDS	19775.05	0.15	19829.5	78.1
A*	56.45	0	4019.6	4019.6

ВИСНОВОК

При виконанні даної лабораторної роботи було розглянуто алгоритми неінформативного (IDS) та інформативного (A^*) пошуку для розв'язання задачі 8-ферзів. Після проведення серії експериментів для кожного алгоритму, було виявлено, що в середньому алгоритм A^* з використанням наведеної в умові евристичною функцією працює краще: кількість ітерацій для досягнення кінцевого стану значно менша, ніж кількість ітерацій алгоритму IDS, що означає більшу швидкість розв'язання задачі; з зазначеними в умові обмеженнями пам'яті та часу алгоритм IDS може зайти в глухий кут. Але алгоритм A^* зберігає всі стани в пам'яті, коли IDS — тільки поточної ітерації.

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%.