**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-12 Васильєв Єгор Костянтинович*

**Перевірив**

(прізвище, ім'я, по батькові)

*Сопов О.О.*

Київ 2022

Зміст

[1 Мета лабораторної роботи 3](#_Toc117445813)

[2 ЗаВдання 4](#_Toc117445814)

[3 Виконання 8](#_Toc117445815)

[3.1 Псевдокод алгоритмів 8](#_Toc117445816)

[3.1.1 Пошук з обмеженням глибини 8](#_Toc117445817)

[3.1.2 Пошук А\* 8](#_Toc117445818)

[3.2 Програмна реалізація 10](#_Toc117445819)

[3.2.1 Вихідний код 10](#_Toc117445820)

[3.2.2 Приклади роботи 15](#_Toc117445821)

[3.3 Дослідження алгоритмів 17](#_Toc117445822)

[Висновок 19](#_Toc117445823)

[Критерії оцінювання 20](#_Toc117445824)

# 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

Варіант 7

## Псевдокод алгоритмів

### Пошук з обмеженням глибини

stack = []

number\_of\_iterations = 0

number\_of\_states = 0

solution = None

curr\_board = Board()

curr\_board.add\_board(self.board)

stack.append(curr\_board)

**поки** (stack):

curr\_board = stack.pop()

**якщо** len(curr\_board.moving\_history) > limit:

**продовжити**

**якщо** curr\_board.amount\_of\_collisions == 0:

solution = curr\_board

**вихід з циклу**

**для** queen\_position **у** curr\_board.queens\_positions\_lst:

new\_col = queen\_position[1]

**для** row\_index **у діапазоні**(1, curr\_board.AMOUNT\_OF\_QUEENS + 1):

number\_of\_iterations += 1

**якщо** row\_index != queen\_position[0]:

new\_row = row\_index

new\_position = (new\_row, new\_col)

new\_board = Board()

new\_board.add\_board(curr\_board.move\_queen

(queen\_position, new\_position))

stack.append(new\_board)

### Пошук А\*

open\_ = []

closed = []

solutions = []

curr\_board = Board()

curr\_board.add\_board(self.board)

open\_.append(curr\_board)

**поки** open\_:

curr\_board = open\_.pop(0)

**якщо** solutions and solutions[0].path\_cost <= open\_[0].path\_cost

+ open\_[0].amount\_of\_collisions + 1:

**вихід з циклу**

**для** board **у** closed:

**якщо** curr\_board.board == board.board:

**продовжити**

closed.append(curr\_board)

**для** queen\_position **у** curr\_board.queens\_positions\_lst:

new\_col = queen\_position[1]

**для** row\_index **в діапазоні**(1, curr\_board.AMOUNT\_OF\_QUEENS + 1):

**якщо** row\_index != queen\_position[0]:

flag = True

new\_row = row\_index

new\_position = (new\_row, new\_col)

new\_board = Board()

new\_board.add\_board(curr\_board.move\_queen(

queen\_position, new\_position))

new\_board.path\_cost = curr\_board.path\_cost + 1

new\_cost = new\_board.amount\_of\_collisions +

new\_board.path\_cost + 1

**якщо** new\_board.amount\_of\_collisions == 0:

**якщо** solutions **і** len(solutions[0].moving\_history)

> len(new\_board.moving\_history):

solutions.insert(0, new\_board)

**інакше якщо** not solutions:

solutions.append(new\_board)

**продовжити**

**для** board **у** open\_:

**якщо** new\_board.board==board.board **і** board.path\_cost

+ new\_board.amount\_of\_collisions < new\_cost:

flag = False

**якщо** flag:

open\_.append(new\_board)

open\_.sort(key=lambda x: x.amount\_of\_collisions

+ x.path\_cost)

solution = solutions[0]

## Програмна реалізація

### Вихідний код

import copy

import random

import time

class Board:

AMOUNT\_OF\_QUEENS = 8

def \_\_init\_\_(self):

self.board = [[] for \_ in range(self.AMOUNT\_OF\_QUEENS)]

self.queens\_positions\_lst = []

self.moving\_history = []

self.amount\_of\_collisions = float('inf')

self.path\_cost = 0

def add\_board(self, board):

self.board = board

self.queens\_positions\_lst = self.queens\_positions()

self.amount\_of\_collisions = self.number\_of\_collisions()

def generate\_board(self):

for col in range(self.AMOUNT\_OF\_QUEENS):

queen\_index = random.randint(0, self.AMOUNT\_OF\_QUEENS - 1)

for row in range(self.AMOUNT\_OF\_QUEENS):

if row == queen\_index:

self.board[row].append("Q")

else:

self.board[row].append("0")

self.queens\_positions\_lst = self.queens\_positions()

self.amount\_of\_collisions = self.number\_of\_collisions()

def queens\_positions(self):

"""return sorted by column index positions of the queens """

position = []

for row, value in enumerate(self.board):

for column, item in enumerate(value):

if self.board[int(row)][int(column)] == "Q":

position.append((row + 1, column + 1))

return sorted(position, key=lambda k: k[1])

def number\_of\_collisions(self):

"""counts the number of pairs of queens that "beat" each other"""

collisions = 0

for count\_1, attack\_queen in enumerate(self.queens\_positions\_lst):

conflict\_row = []

conflict\_diagonal = []

for count\_2 in range(count\_1 + 1, Board.AMOUNT\_OF\_QUEENS):

attacked\_queen = self.queens\_positions\_lst[count\_2]

(atk\_row, atk\_column) = attack\_queen

(attacked\_row, attacked\_column) = attacked\_queen

if atk\_row == attacked\_row and attacked\_row not in conflict\_row:

collisions += 1

conflict\_row.append(attacked\_row)

elif abs(int(atk\_row) - int(attacked\_row)) == abs(

int(atk\_column) - int(attacked\_column)):

if atk\_row + atk\_column == attacked\_column +

attacked\_row and "up" + str(

atk\_row + atk\_column) not in conflict\_diagonal:

conflict\_diagonal.append("up" +

str(attacked\_column + attacked\_row))

collisions += 1

elif attacked\_column - attacked\_row == atk\_column –

atk\_row and "down" + str(attacked\_column -

attacked\_row) not in conflict\_diagonal:

conflict\_diagonal.append("down" +

str(attacked\_column - attacked\_row))

collisions += 1

return collisions

def print\_board(self):

for row in range(self.AMOUNT\_OF\_QUEENS):

for col in range(self.AMOUNT\_OF\_QUEENS):

if (row + 1, col + 1) in self.queens\_positions\_lst:

print("1", end=" ")

else:

print("-", end=" ")

print()

print()

def move\_queen(self, start\_pos, finish\_pos):

new\_matrix = copy.deepcopy(self.board)

start\_row, start\_col = start\_pos

finish\_row, finish\_col = finish\_pos

new\_matrix[int(start\_row - 1)][int(start\_col - 1)] = 0

new\_matrix[int(finish\_row - 1)][int(finish\_col - 1)] = "Q"

return new\_matrix

def A\_star(self):

"""searching for solutions among the nodes added to the stack

until the heuristic function of the best solution becomes less

than the length of the element from the stack with the smallest

value of this function"""

print("\nInitial arrangement")

self.print\_board()

number\_of\_iterations = 0

number\_of\_states = 0

open\_ = []

closed = []

solutions = []

curr\_board = Board()

curr\_board.add\_board(self.board)

curr\_board.path\_cost = 0

open\_.append(curr\_board)

while open\_:

curr\_board = open\_.pop(0)

if solutions and solutions[0].path\_cost <= open\_[0].path\_cost

+ open\_[0].amount\_of\_collisions + 1:

break

for board in closed:

number\_of\_iterations += 1

if curr\_board.board == board.board:

continue

closed.append(curr\_board)

for queen\_position in curr\_board.queens\_positions\_lst:

new\_col = queen\_position[1]

for row\_index in range(1, curr\_board.AMOUNT\_OF\_QUEENS + 1):

number\_of\_iterations += 1

if row\_index != queen\_position[0]:

flag = True

new\_row = row\_index

new\_position = (new\_row, new\_col)

new\_board = Board()

number\_of\_states += 1

new\_board.add\_board(curr\_board.move\_queen(queen\_position, new\_position))

new\_board.moving\_history =

curr\_board.moving\_history.copy()

new\_board.moving\_history.append(str(queen\_position) +

" => " + str(new\_position))

new\_board.path\_cost = curr\_board.path\_cost + 1

new\_cost = new\_board.amount\_of\_collisions +

new\_board.path\_cost + 1

if new\_board.amount\_of\_collisions == 0:

if solutions and len(solutions[0].moving\_history)

> len(new\_board.moving\_history):

solutions.insert(0, new\_board)

print(new\_board.moving\_history)

elif not solutions:

solutions.append(new\_board)

continue

for board in open\_:

number\_of\_iterations += 1

if new\_board.board == board.board and board.path\_cost

+ new\_board.amount\_of\_collisions < new\_cost:

flag = False

if flag:

open\_.append(new\_board)

open\_.sort(key=lambda x: x.amount\_of\_collisions +

x.path\_cost)

print("\nFinal arrangement")

solutions[0].print\_board()

print("Total permutations done:", len(solutions[0].moving\_history))

print(solutions[0].moving\_history)

print("Amount of iterations:", number\_of\_iterations)

print("Total generated states:", number\_of\_states)

print("Total states in memory:", len(open\_) + len(closed))

def ldfs(self, limit=10):

"""search in depth; new nodes obtained by moving one

queen are added to the stack, 56 new states are generated

for each new arrangement of pieces, the search is carried out

until an arrangement without collisions is found"""

print("\nInitial arrangement")

self.print\_board()

stack = []

number\_of\_iterations = 0

number\_of\_states = 0

solution = None

curr\_board = Board()

curr\_board.add\_board(self.board)

stack.append(curr\_board)

while len(stack):

if len(curr\_board.moving\_history) > limit:

number\_of\_iterations += 1

continue

if curr\_board.amount\_of\_collisions == 0:

solution = curr\_board

break

for queen\_position in curr\_board.queens\_positions\_lst:

new\_col = queen\_position[1]

for row\_index in range(1, curr\_board.AMOUNT\_OF\_QUEENS + 1):

number\_of\_iterations += 1

if row\_index != queen\_position[0]:

new\_row = row\_index

new\_position = (new\_row, new\_col)

new\_board = Board()

number\_of\_states += 1

new\_board.add\_board(curr\_board.move\_queen

(queen\_position, new\_position))

new\_board.moving\_history =

curr\_board.moving\_history.copy()

new\_board.moving\_history.append(str(queen\_position) +

" => " + str(new\_position))

stack.append(new\_board)

print("\nFinal arrangement")

solution.print\_board()

print("Total permutations done:", len(solution.moving\_history))

print(solution.moving\_history)

print("Amount of iterations:", number\_of\_iterations)

print("Total generated states:", number\_of\_states)

print("Total states in memory:", len(stack))

start = time.time()

board\_ = Board()

board\_.generate\_board()

# board\_.ldfs()

board\_.A\_star()

end = time.time()

print("Time taken: ", str(end - start), "seconds / ", str((end - start) / 60), "minutes")

### Приклади роботи

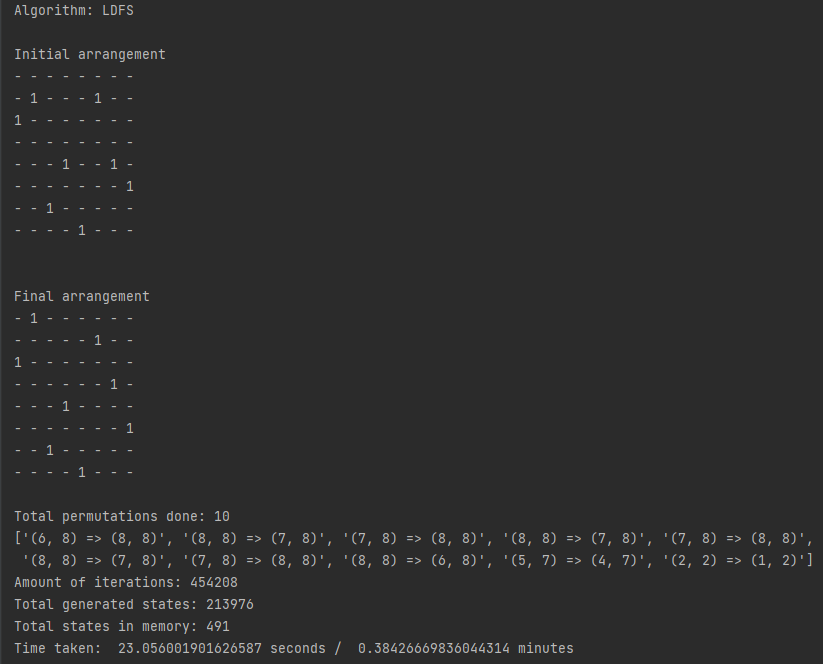
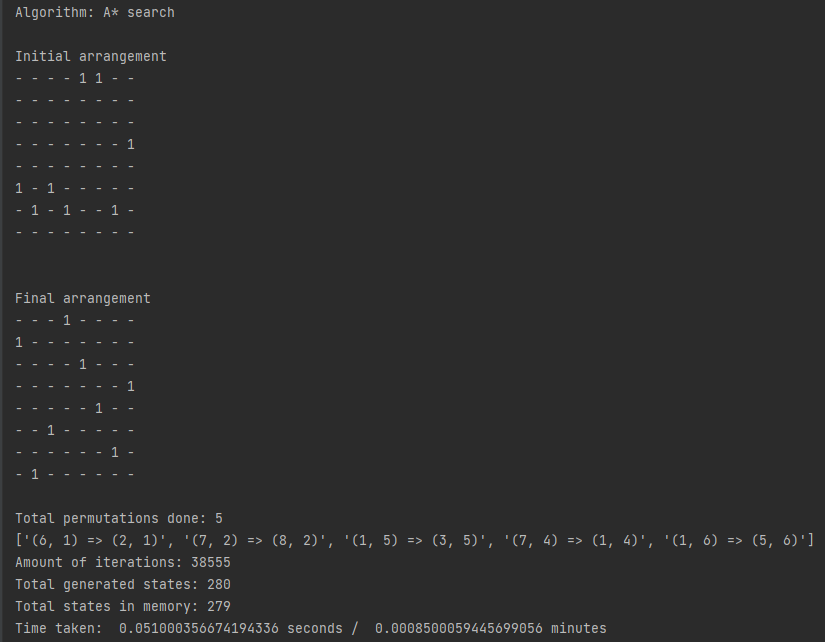
На рисунках 3.1 і 3.2 показані приклади роботи програми для різних алгоритмів пошуку.

Рисунок 3.1 – Алгоритм пошуку в глибину з обмеженням глибини

Рисунок 3.2 – Алгоритм пошуку А\*

## Дослідження алгоритмів

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

Таблиця 3.1 – Характеристики оцінювання алгоритму LDFS

|  |  |  |  |
| --- | --- | --- | --- |
| Початкові стани | Ітерації | Всього станів | Всього станів у пам’яті |
| Стан 1 | 1199616 | 564760 | 499 |
| Стан 2 | 92548848 | 43552768 | 784 |
| Стан 3 | 46878032 | 46878032 | 471 |
| Стан 4 | 47657720 | 22427384 | 471 |
| Стан 5 | 28809944 | 13557824 | 432 |
| Стан 6 | 109695296 | 571541235 | 451 |
| Стан 7 | 249945176 | 117621448 | 401 |
| Стан 8 | 284807264 | 134027152 | 434 |
| Стан 9 | 516754317 | 261521781 | 441 |
| Стан 10 | 276445139 | 141325848 | 449 |
| Стан 11 | 361512954 | 138124762 | 461 |
| Стан 12 | 476414212 | 241365278 | 412 |
| Стан 13 | 312512154 | 162354780 | 408 |
| Стан 14 | 115231548 | 57412354 | 410 |
| Стан 15 | 250367142 | 126784354 | 415 |
| Стан 16 | 318452157 | 159871235 | 462 |
| Стан 17 | 412259763 | 208631742 | 473 |
| Стан 18 | 612354871 | 302147563 | 473 |
| Стан 19 | 22809944 | 10557824 | 435 |
| Стан 20 | 2199616 | 1064760 | 502 |
| СЕРЕДНЄ | 226942785 | 138066644 | 464 |

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

Таблиця 3.2 – Характеристики оцінювання алгоритму А\*

|  |  |  |  |
| --- | --- | --- | --- |
| Початкові стани | Ітерації | Всього станів | Всього станів у пам’яті |
| Стан 1 | 531239 | 1064 | 943 |
| Стан 2 | 24680 | 224 | 222 |
| Стан 3 | 38555 | 280 | 279 |
| Стан 4 | 36366382 | 9968 | 7834 |
| Стан 5 | 127804517 | 17920 | 14877 |
| Стан 6 | 24682 | 224 | 223 |
| Стан 7 | 2160579 | 2128 | 2068 |
| Стан 8 | 510634 | 1064 | 970 |
| Стан 9 | 122219 | 504 | 487 |
| Стан 10 | 3989935 | 3360 | 2389 |
| Стан 11 | 3075508 | 2800 | 2326 |
| Стан 12 | 74689 | 392 | 385 |
| Стан 13 | 38450 | 280 | 278 |
| Стан 14 | 2504213 | 2296 | 2160 |
| Стан 15 | 3043163 | 2688 | 2306 |
| Стан 16 | 602316 | 1176 | 997 |
| Стан 17 | 924710 | 1456 | 1161 |
| Стан 18 | 801676 | 1288 | 1266 |
| Стан 19 | 199952 | 672 | 609 |
| Стан 20 | 153809 | 560 | 554 |
| СЕРЕДНЄ | 9149595 | 2517 | 2116 |

Висновок

При виконанні даної лабораторної роботи було розглянуто алгоритми пошуку в глибину з обмеженням глибини та А\* для задачі розстановки 8 ферзів, було розроблено псевдокоди та здійснено програмну реалізацію цих алгоритмів; було здійснено 20 експериментів для кожного із алгоритмів і визначено середню кількість ітерацій, кількість пройдених станів та максимальну кількість станів у пам’яті.

Після дослідження даних алгоритмів, стає очевидним, що пошук з обмеженням глибини, як представник неінформованого пошуку, є неповним і неоптимальним алгоритмом, коли пошук А\* є повним і оптимальним допоки евристична функція є прийнятною; було встановлено, що пошук А\* працює набагато швидше та генерує в рази менше станів, однак зберігає майже їх всі у пам’яті, у порівнянні з LDFS, який у свою чергу досліджує усі можливі перестановки, допоки не знайде рішення, що може зайняти дуже довгий проміжок часу.

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%.