**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-15, Дацьо Іван*

**Перевірив**

(прізвище, ім'я, по батькові)

*Головченко М.М.*

Київ 2022

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 10](#_Toc81070692)

[3.2.1 Вихідний код 10](#_Toc81070693)

[3.2.2 Приклади роботи 12](#_Toc81070694)

[3.3 Дослідження алгоритмів 12](#_Toc81070695)

[Висновок 25](#_Toc81070696)

[Критерії оцінювання 27](#_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.
* **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

## Псевдокод алгоритмів

### Псевдокод функції F2

**DEFINE FUNCTION** get\_conflict(self, i, j):

conf\_n = 0

# Horizontal conflict:

# 1. Before Q

**FOR** col **IN** range(j):

count = 0

**IF** matrix[i][col] **EQUALS** 1:

count += 1

**IF** count:

conf\_n += count

**break**

# Horizontal conflict:

# 2. After Q

**FOR** col **IN** range(j + 1, size):

Count = 0

**IF** matrix[i][col] **EQUALS** 1:

count += 1

**IF** count:

conf\_n += count

**break**

# Vertical conflict:

# 1.Before Q

**FOR** row **IN** range(i):

Count = 0

**IF** matrix[row][j] **EQUALS** 1:

count += 1

**IF** count:

conf\_n += count

**break**

# Vertical conflict:

# 2. After Q

**FOR** row **IN** range(i + 1, size):

count = 0

**IF** matrix[row][j] **EQUALS** 1:

count += 1

**IF** count:

conf\_n += count

break

**SET** row **TO** i - 1

**SET** col **TO** j - 1

# Diagonal conflict:

# 1.Before Q

**WHILE** correct\_index(row, col):

**IF** matrix[row][col] **EQUALS** 1:

conf\_n += 1

**break**

row -= 1

col -= 1

**SET** row **TO** i + 1

**SET** col **TO** j + 1

# Diagonal conflict:

# 2.After diagonal

**WHILE** correct\_index(row, col):

**IF** matrix[row][col] **EQUALS** 1:

conf\_n += 1

**break**

row += 1

col += 1

**SET** row **TO** i - 1

**SET** col **TO** j + 1

# Anti-Diagonal conflict:

# 1.Before Q

**WHILE** correct\_index(row, col):

**IF** matrix[row][col] **EQUALS** 1:

conf\_n += 1

**break**

row -= 1

col += 1

**SET** row **TO** i + 1

**SET** col **TO** j - 1

# Anti-Diagonal conflict:

# 2.After Q

**WHILE** correct\_index(row, col):

**IF** matrix[row][col] **EQUALS** 1:

conf\_n += 1

**break**

row += 1

col -= 1

**RETURN** conf\_n

### Псевдокод алгоритму LDFS

**DEFINE FUNCTION** solve(limit):

**IF** LDFS(root, limit):

OUTPUT("There are solution: ")

**ELSE**:

OUTPUT("No solution with this limit")

**RETURN** True

**DEFINE FUNCTION** LDFS(node: Node, limit):

iteration += 1

**SET** last\_node **TO** node

**IF** (node.is\_solved()):

OUTPUT("Solved board:")

node.board.OUTPUT()

**RETURN** True

**IF** node.depth < limit:

# move queen into new spot

node.expand()

total\_st += len(node.children)

**FOR** i **IN** range(len(node.children)):

**IF** LDFS(node.children[i], limit):

mem\_states += len(node.children)

**RETURN** True

**ELSE:**

dead\_ends += 1

RETURN False

### Псевдокод алгоритму A\*

**DEFINE FUNCTION** AStar():

# priority queue that uses heuristic function that defined IN node file

**SET** opened: PriorityQueue[Node] **TO** PriorityQueue()

**SET** closed: set[Board] **TO** set()

# Root into queue

opened.put(self.root)

**WHILE** not opened.empty():

**SET** top **TO** opened.get()

**IF** top.is\_solved():

OUTPUT("Solved board:")

top.board.OUTPUT()

**SET** total\_states **TO** opened.qsize() + len(closed)

**SET** memory\_states **TO** opened.qsize() + len(closed)

closed.add(top.board)

# move the queen to a new slot

top.expand()

**SET** successors: list[Node] **TO** top.children

**FOR** i **IN** range(len(successors)):

**IF** successors[i].board **IN** closed:

**continue**

opened.put(successors[i])

self.iter += 1

## Програмна реалізація

### Вихідний код

#### LDFS:

**node.py:**

**from** board **import** Board  
**import** pickle  
**from** typing **import** Any, Optional  
  
**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 = pickle.loads(pickle.dumps(other, -1))  
 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))  
  
 *# look at if it is solved* **def** is\_solved(self):  
 **return** self.board.conflict\_number() == 0  
  
 *# make move* **def** expand(self):  
 row = 0  
 shift = 1  
  
 **for** i **in** range(len(self.children)):  
 **if** shift == self.board.size:  
 row += 1  
 shift = 1  
  
 cp = pickle.loads(pickle.dumps(self, -1))  
  
 self.children[i] = Node(other=cp)  
 self.children[i].board.move\_figure(row, shift)  
  
 shift += 1

***Queens.py***

**from** node **import** Node  
**from** board **import** Board, Optional  
**from** Logger **import** NQLogger  
**import** pickle  
  
  
**class** Nqueens:  
 **def** \_\_init\_\_(self, queens: int, board: Optional[Board] = **None**) -> **None**:  
  
 *# for report info* self.mem\_states: int = 1  
 self.total\_st: int = 1  
 self.iteration: int = 0  
 self.dead\_ends = 0  
  
 self.size = queens  
 self.last\_node: Node  
  
 self.root = Node(queens=queens, other=board)  
  
 **def** solve(self, limit):  
 NQLogger.info(**"\*\* LDFS Algorithm \*\*"**)  
  
 **if** self.LDFS(self.root, limit):  
 print(**"There are solution: "**)  
 **else**:  
 print(**"No solution with this limit"**)  
  
 self.info()  
 **return True  
  
 def** LDFS(self, node: Node, limit):  
  
 self.iteration += 1  
  
 *# pickle for copy* self.last\_node = pickle.loads(pickle.dumps(node, -1))  
  
 **if** (node.is\_solved()):  
 NQLogger.info(**"\*\* IDS Solved \*\*"**)  
  
 print(**"Solved board:"**)  
 node.board.print()  
  
 print(**"limits:"**)  
 print(**f" ---: depth: {**node.depth**}"**)  
 print(**f" ---: limit: {**limit**}\n"**)  
  
 **return True  
  
 if** node.depth < limit:  
  
 NQLogger.info(**f"#{**self.iteration**}: Expand with {**len(node.children)**} child nodes"**)  
  
 node.expand()  
 self.total\_st += len(node.children)  
  
 **for** i **in** range(len(node.children)):  
 **if** self.LDFS(node.children[i], limit):  
 self.mem\_states += len(node.children)  
 **return True  
 else**:  
 self.dead\_ends += 1  
  
 **return False  
  
 def** info(self):  
 print(**"In total: "**)  
 print(**f" ---: iterations: {**self.iteration**}"**)  
 print(**f" ---: states: {**self.total\_st**}"**)  
 print(**f" ---: memory states: {**self.mem\_states**}"**)  
 print(**f" ---: dead ends {**self.dead\_ends**}"**)  
  
 print()

***main.py***

**from** argparse **import** ArgumentParser, ArgumentTypeError  
**from** Queens **import** Nqueens  
  
*# for analysis***from** timer **import** Timer  
**from** Logger **import** log\_file  
  
  
**def** val\_int(val):  
 tmp = int(val)  
 **if** tmp < 0:  
 **raise** ArgumentTypeError(**"must provide non-negative value"**)  
 **return** tmp  
  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 argparser = ArgumentParser()  
 argparser.add\_argument(**'-q'**, type=val\_int, default=8)  
 argparser.add\_argument(**'-l'**, type=str, default=**'info\_about\_alg.log'**)  
  
 *# get number of queens from cl args* queens: int = argparser.parse\_args().q  
 log\_path: str = argparser.parse\_args().l  
  
 *# Set up the logger* log\_file(log\_path)  
  
 *# Create root node's board* NQ\_LDFS = Nqueens(queens)  
  
 *# Print the root node's board* NQ\_LDFS.root.board.print(pre=**f"Generated {**queens**}x{**queens**} board:"**, end=**''**)  
 print(**f" ---: conflicts: {**NQ\_LDFS.root.board.conflict\_number()**}\n"**)  
  
  
 **def** \_\_solve():  
 **with** Timer():  
 NQ\_LDFS.solve(queens)  
  
  
 \_\_solve()  
  
 print(**f"\n\*\* logged to {**log\_path**} \*\*"**)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#### A\*:

***node.py***

**from** typing **import** Any, Optional  
**from** board **import** Board  
**import** pickle  
  
**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 = pickle.loads(pickle.dumps(other, -1))  
 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))  
  
 *# cost for A star* @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  
  
 *# look at if it is solved* **def** is\_solved(self):  
 **return** self.board.conflict\_number() == 0  
  
 *# make move* **def** expand(self):  
 row = 0  
 shift = 1  
  
 **for** i **in** range(len(self.children)):  
 **if** shift == self.board.size:  
 row += 1  
 shift = 1  
  
 cp = pickle.loads(pickle.dumps(self, -1))  
  
 self.children[i] = Node(other=cp)  
 self.children[i].board.move\_figure(row, shift)  
  
 shift += 1

***Queens.py***

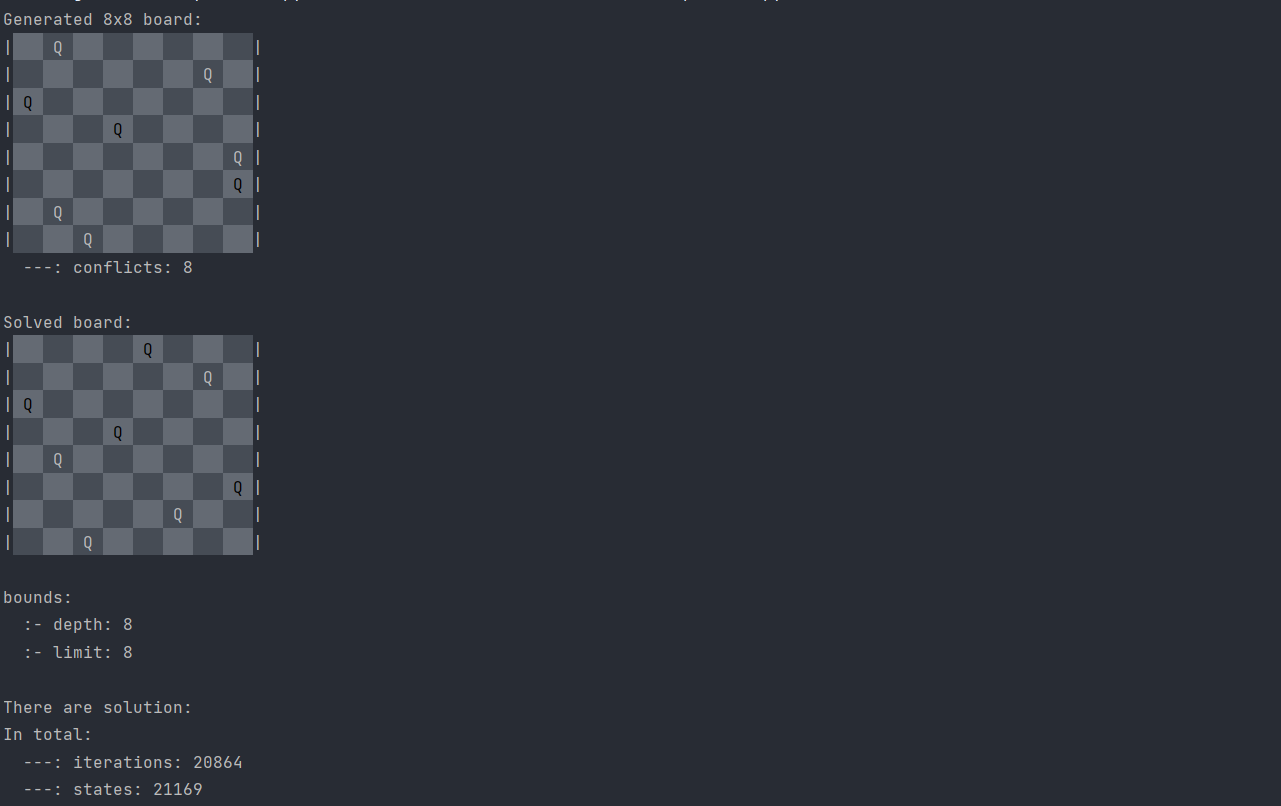
**from** queue **import** PriorityQueue  
**from** typing **import** Optional  
**from** node **import** Node  
**from** board **import** Board  
**from** Logger **import** NQLogger  
  
  
**class** Nqueens:  
 **def** \_\_init\_\_(self, queens: int, board: Optional[Board] = **None**) -> **None**:  
  
 *# for report info* 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-Star Algorithm \*\*\*"**)  
  
 *# priority queue that uses heuristic function that defined in node file* opened: PriorityQueue[Node] = PriorityQueue()  
 closed: set[Board] = set()  
  
 *# Root into queue* opened.put(self.root)  
  
 NQLogger.info(**"Placed root into queue"**)  
  
 **while not** opened.empty():  
 top = opened.get()  
  
 **if** top.is\_solved():  
 NQLogger.info(**"\*\*\* A-Star 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"**)  
  
 *# move the queen to a new slot* 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(**"In total:"**)  
 print(**f" ---: iterations: {**self.iter**}"**)  
 print(**f" ---: states: {**self.total\_states**}"**)  
 print(**f" ---: memory states: {**self.memory\_states**}\n"**)

***main.py***

**from** argparse **import** ArgumentParser, ArgumentTypeError  
**from** Queens **import** Nqueens  
  
*# for analysis***from** timer **import** Timer  
**from** Logger **import** log\_file  
  
  
**def** val\_int(val):  
 tmp = int(val)  
 **if** tmp < 0:  
 **raise** ArgumentTypeError(**"must provide non-negative value"**)  
 **return** tmp  
  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 argparser = ArgumentParser()  
 argparser.add\_argument(**'-q'**, type=val\_int, default=8)  
 argparser.add\_argument(**'-l'**, type=str, default=**'info\_about\_alg.log'**)  
  
 *# get number of queens from cl args* queens: int = argparser.parse\_args().q  
 log\_path: str = argparser.parse\_args().l  
  
 *# Set up the logger* log\_file(log\_path)  
  
 *# Create root node's board* NQ\_Astar = Nqueens(queens)  
  
 *# Print the root node's board* NQ\_Astar.root.board.print(pre=**f"Generated {**queens**}x{**queens**} board:"**, end=**''**)  
 print(**f" ---: conflicts: {**NQ\_Astar.root.board.conflict\_number()**}\n"**)  
  
  
 **def** solve():  
 **with** Timer():  
 NQ\_Astar.AStar()  
  
  
 solve()  
  
 print(**f"\n\*\* logged to {**log\_path**} \*\*"**)

### Приклади роботи

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



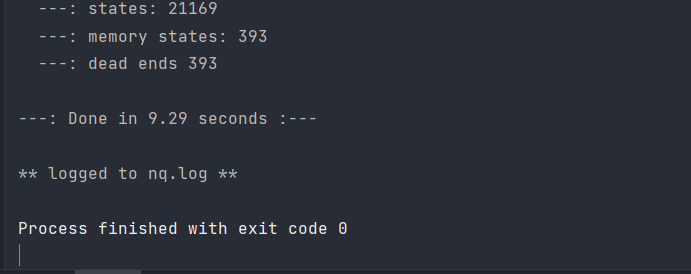


Рисунок 3.1 – Алгоритм LDFS

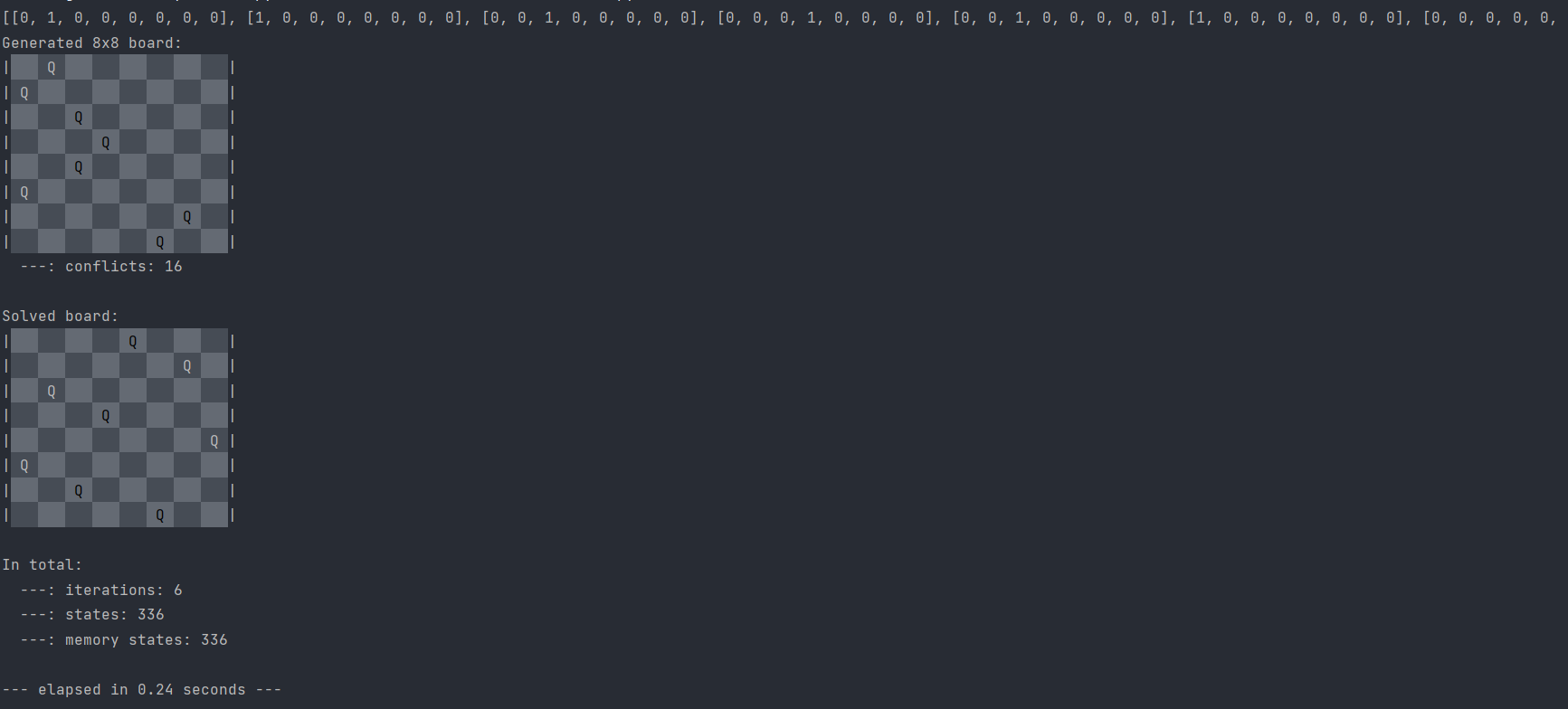


Рисунок 3.2 – Алгоритм A\*

## Дослідження алгоритмів

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

Таблиця 3.1 – Характеристики оцінювання LDFS

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Початкові стани | Ітерації | К-сть глухих кутів | Всього станів | Всього станів у пам’яті |
|  | 42701 | 42693 | 43009 | 393 |
|  | 20002 | 19994 | 20329 | 393 |
|  | 560022 | 560014 | 560337 | 393 |

Продовження таблиці 3.1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Початкові стани | Ітерації | К-сть глухих кутів | Всього станів | Всього станів у пам’яті |
|  | 563032 | 563024 | 563361 | 393 |
|  | 211709 | 211701 | 212017 | 393 |
|  | 55685 | 55677 | 56001 | 393 |

Продовження таблиці 3.1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Початкові стани | Ітерації | К-сть глухих кутів | Всього станів | Всього станів у пам’яті |
|  | 7355 | 7347 | 7673 | 393 |
|  | 1131852 | 1131844 | 1132153 | 393 |
|  | 751188 | 751180 | 751521 | 393 |

Продовження таблиці 3.1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Початкові стани | Ітерації | К-сть глухих кутів | Всього станів | Всього станів у пам’яті |
|  | 1004823 | 1004815 | 1005089 | 393 |
|  | 1132149 | 1132141 | 1132433 | 393 |
|  | 772141 | 772133 | 772409 | 393 |
|  | 72413 | 72405 | 72689 | 393 |

Продовження таблиці 3.1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Початкові стани | Ітерації | К-сть глухих кутів | Всього станів | Всього станів у пам’яті |
|  | 397075 | 397067 | 397377 | 393 |
|  | 579531 | 579523 | 579825 | 393 |
|  | 18011 | 18003 | 18313 | 393 |

Продовження таблиці 3.1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Початкові стани | Ітерації | К-сть глухих кутів | Всього станів | Всього станів у пам’яті |
|  | 1333374 | 1333366 | 1333641 | 393 |
|  | 59474 | 59466 | 59753 | 393 |
|  | 1304045 | 1304037 | 1304353 | 393 |
|  | 1317181 | 1317173 | 1317457 | 393 |

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

Таблиця 3.2 – Характеристики оцінювання А\*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Початкові стани | Ітерації | К-сть глухих кутів | Всього станів | Всього станів у пам’яті |
|  | 32 | 0 | 1792 | 1792 |
|  | 8 | 0 | 448 | 448 |
|  | 4 | 0 | 224 | 224 |

Продовження таблиці 3.2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Початкові стани | Ітерації | К-сть глухих кутів | Всього станів | Всього станів у пам’яті |
|  | 4 | 0 | 224 | 224 |
|  | 91 | 0 | 5096 | 5096 |
|  | 12 | 0 | 672 | 672 |
|  | 5 | 0 | 224 | 224 |

Продовження таблиці 3.2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Початкові стани | Ітерації | К-сть глухих кутів | Всього станів | Всього станів у пам’яті |
|  | 26 | 0 | 1456 | 1456 |
|  | 3 | 0 | 168 | 168 |
|  | 6 | 0 | 336 | 336 |
|  | 11 | 0 | 616 | 616 |

Продовження таблиці 3.2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Початкові стани | Ітерації | К-сть глухих кутів | Всього станів | Всього станів у пам’яті |
|  | 160 | 0 | 8960 | 8960 |
|  | 12 | 0 | 672 | 672 |
|  | 70 | 0 | 3920 | 3920 |
|  | 80 | 0 | 4480 | 4480 |

Продовження таблиці 3.2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Початкові стани | Ітерації | К-сть глухих кутів | Всього станів | Всього станів у пам’яті |
|  | 159 | 0 | 8904 | 8904 |
|  | 6 | 0 | 336 | 336 |
|  | 53 | 0 | 2968 | 2868 |

В таблиці 3.3 наведені середні значення характеристик оцінювання алгоритмів LDFS і А\*, які були визначені серіями із 20 дослідів (таблиці 3.1-3.2).

Таблиця 3.3 – Середні значення характеристик оцінування алгоритмів

LDFS і А\*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Назва алгоритму | Ітерації | К-сть глухих кутів | Всього станів | Всього станів у пам’яті |
| LDFS | 772141 | 772133 | 772409 | 393 |
| A\* | 26 | 0 | 1456 | 1456 |

Висновок

При виконанні даної лабораторної роботи було розглянуто алгоритм неінформативного пошуку (АНП) – Limited Depth First Search (LDFS), а також алгоритм інформатичного пошуку (АІП) – A\*, з використанням евристичної функції – **F2** – кількість пар ферзів, які б’ють один одного без урахування видимості.

Алгоритми були розглянуті на прикладі задачі про 8 ферзів. В результаті виконання лабораторної роботи я отримав практичні навички роботи з цими алгоритмами, а саме записав псевдокод алгоритмів, виконав їх програмну реалізацію, а також було проведено дослідження на основі 20 станів для кожного.

Програмна реалізація і псевдокод алгоритму LDFS виконувалась «AS IS», тобто без додаткових модифікацій алгоритму. LDFS працює аналогічно класичному алгоритму DFS (пошук в глибину), але однією важливою відмінністю – з обмеженням максимальної глибини. Тобто вузол з максимально допустимою глибиною не розгортається. Було розглянуто алгоритм на з вхідною дошкою яка містить на кожному рядку лише одну королеву ( для зменшення часу необхідного на пошук вирішення проблеми) .Також для алгоритму було обрано максимально глибину 8, тобто кількість ферзів, оскільки будь-яку задачу можна вирішити, якщо поставити кожен з 8 ферзів на правильну клітинку у своєму рядку. В загальному навіть виконавши спрощення алгоритм LDFS все ще є повільним .

Тестування роботи алгоритмів LDFS і A\* проводилось на спрощеному варіанті( варіанті коли на дошці в одному рядку знаходиться лише одна королева) задачі про 8 ферзів, для порівняння ефективності їх виконання в однакових умовах. У результаті виконання можна зробити висновок що алгоритми A\* працює значно швидше, генеруючи менше станів, виконуючи менше ітерацій та цей алгоритм немає глухих кутів. Проте A\* має більше станів, що зберігаються в пам’яті. Це зумовлено тим, що LDFS – це алгоритм АНП, алгоритм що “наосліп” виконує пошук, а A\* – це АІП, алгоритм, що використовує додаткову характеристику для обрання наступного вузла, який буде розглянуто.

Єдиною перевагою LDFS перед А\* – є використання пам’яті. В середньому: А\* зберігає 1456, а LDFS – 393.

Просторова складність LDFS дорівнює , де

b – кількість нащадків у вузла, а L – максимальна глибина пошуку.

Простора складність А\* дорівнює , де

d – це глибина найбільш поверхневого рішення.

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%.