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-12 Тарасюк Євгеній Сергійович*
(шифр, прізвище, ім'я, по батькові)

Перевірив *Сопов О. О.*
(прізвище, ім'я, по батькові)

Київ 2022

Зміст

1 Мета лабораторної роботи

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

2 ЗАВДАННЯ

Записати алгоритм розв’язання задачі у вигляді псевдокоду, відповідно до варіанту (таблиця 2.1).

Реалізувати програму, яка розв’язує поставлену задачу згідно варіанту (таблиця 2.1) за допомогою алгоритму неінформативного пошуку **АНП**, алгоритму інформативного пошуку **АІП,** що використовує задану евристичну функцію Func, або алгоритму локального пошуку **АЛП та бектрекінгу,** що використовує задану евристичну функцію Func.

Програму реалізувати на довільній мові програмування.

Увага! Алгоритм неінформативного пошуку **АНП,** реалізовується за принципом «AS IS», тобто так, як є, без додаткових модифікацій (таких як перевірка циклів, наприклад).

Провести серію експериментів для вивчення ефективності роботи алгоритмів. Кожний експеримент повинен відрізнятись початковим станом. Серія повинна містити не менше 20 експериментів для кожного алгоритму. Початковий стан зафіксувати у таблиці експериментів. За проведеними серіями необхідно визначити:

- середню кількість етапів (кроків), які знадобилось для досягнення розв’язку (ітерації);
- середню кількість випадків, коли алгоритм потрапляв в глухий кут (не міг знайти оптимальний розв’язок) – якщо таке можливе;
- середню кількість згенерованих станів під час пошуку;
- середню кількість станів, що зберігаються в пам’яті під час роботи програми.

Передбачити можливість обмеження виконання програми за часом (30 хвилин) та використання пам’яті (1 Гб).

Використані позначення:

- **8-ферзів** – Задача про вісім ферзів полягає в такому розміщенні восьми ферзів на шахівниці, що жодна з них не ставить під удар один одного. Тобто, вони не повинні стояти в одній вертикалі, горизонталі чи діагоналі.
- **8-puzzle** – гра, що складається з 8 однакових квадратних пластинок з нанесеними числами від 1 до 8. Пластинки поміщаються в квадратну коробку, довжина сторони якої в три рази більша довжини сторони пластинок, відповідно в коробці залишається незаповненим одне квадратне поле. Мета гри – переміщаючи пластинки по коробці досягти 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 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** – 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	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

3 Виконання

1.1 Псевдокод алгоритмів

1.1.1 Функція знаходження можливих кроків для зміни стану

```
FUNCTION getPossibleSteps(currState, visitedStates):
    emptyInd <- currState.index(0)
    possibleSteps <- []
    IF emptyInd > 2:
        newState <- currState[:]
        newState[emptyInd], newState[emptyInd - 3] <-
newState[emptyInd - 3], newState[emptyInd]
        IF not newState in visitedStates:
            possibleSteps.append(newState)
        ENDIF
    ENDIF
    IF emptyInd < 6:
        newState <- currState[:]
        newState[emptyInd], newState[emptyInd + 3] <-
newState[emptyInd + 3], newState[emptyInd]
        IF not newState in visitedStates:
            possibleSteps.append(newState)
        ENDIF
    ENDIF
    IF emptyInd % 3 > 0:
        newState <- currState[:]
        newState[emptyInd], newState[emptyInd - 1]<-
newState[emptyInd - 1], newState[emptyInd]
        IF not newState in visitedStates:
            possibleSteps.append(newState)
        ENDIF
    ENDIF
    IF emptyInd % 3 < 2:
        newState <- currState[:]
```

```
        newState[emptyInd], newState[emptyInd + 1] <-
newState[emptyInd + 1], newState[emptyInd]
        IF not newState in visitedStates:
            possibleSteps.append(newState)
        ENDIF
    ENDIF
    RETURN possibleSteps
ENDFUNCTION
```

1.1.2 Неінформативний пошук Breadth-first search

```
FUNCTION SolveBFS():
     finalState <- [1, 2, 3, 4, 5, 6, 7, 8, 0]
     startingState <- GameBoard.numbers
     statesQueue <- [startingState]
     queuedStatesDepths <- [0]
     visitedStates <- []
     deadEnds <- 0
     currState <- statesQueue.pop(0)
     currDepth <- queuedStatesDepths.pop(0)
     depthLimit <- 24
     WHILE currState != finalState AND currDepth < depthLimit:
         newStates <-  getPossibleSteps(currState, visitedStates)
         IF len(newStates) = 0:
             deadEnds += 1
         ENDIF
         statesQueue.extend(newStates)
         queuedStatesDepths.extend([currDepth+1]*len(newStates))
         visitedStates.append(currState)
         currState <- statesQueue.pop(0)
         currDepth <- queuedStatesDepths.pop(0)
     ENDWHILE
ENDFUNCTION
```

1.1.3 Інформативний пошук A*

```
FUNCTION matches(currState):
    matchesNum <- 0
```

```
    FOR i = 0, i < 8:
        IF currState[i] = i + 1:
            matchesNum += 1
        ENDIF
    ENDFOR
    IF currState[8] = 0:
        matchesNum += 1
    ENDIF
    RETURN matchesNum
ENDFUNCTION

FUNCTION solveAStar(self):
    finalState <- [1, 2, 3, 4, 5, 6, 7, 8, 0]
    startingStateArray <- GameBoard.numbers
    statesQueue <- [[startingState],[],[],[],[],[],[],[],[],[]]
    queuedStatesDepths <- [[0],[],[],[],[],[],[],[],[],[]]
    visitedStates <- []
    deadEnds <- 0
    currState <- statesQueue[0].pop(0)
    currDepth <- queuedStatesDepths[0].pop(0)
    WHILE currState != finalState:
        newStates <-  getPossibleSteps(currState, visitedStates)
        IF len(newStates) = 0:
            deadEnds += 1
        ENDIF
        FOR state in newStates:
            ind <- 9 -  matches(state)
            statesQueue[ind].append(state)
            queuedStatesDepths[ind].append(currDepth+1)
        ENDFOR
        visitedStates.append(currState)
        i <- 0
        WHILE len(statesQueue[i]) = 0:
            i += 1
```

```
        ENDWHILE
        currState <- statesQueue[i].pop(0)
        currDepth <- queuedStatesDepths[i].pop(0)
    ENDWHILE
ENDFUNCTION
```

1.2 Програмна реалізація

1.2.1 Вихідний код

EightPuzzle.py

```
import pygame
import Mouse
import Keyboard
import GameScreen

def main():
    pygame.init()
    pygame.font.init()
    scr = pygame.display.set_mode((750, 500))
    pygame.display.set_caption("8-Puzzle")

    keyboard = Keyboard.Keyboard()
    mouse = Mouse.Mouse()
    gameScreen = GameScreen.GameScreen(scr)

    clock = pygame.time.Clock()
    FPS = 20

    while True:

        keyboard.update()
        mouse.update()
        gameScreen.update(keyboard, mouse)
        pygame.display.update()

        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                break
```

```
    if __name__ == '__main__':
        main()
```

Mouse.py

```
import pygame

class Mouse:
    def __init__(self):
        """create a mouse object to record mouse coords and state"""
        self.x : int
        self.y : int
        self.isDown : bool

    def update(self):
        """get new data for the next frame"""
        self.x = pygame.mouse.get_pos()[0]
        self.y = pygame.mouse.get_pos()[1]
        if pygame.mouse.get_pressed()[0]:
            if self.isDown == False:
                self.isDown = True
            elif self.isDown:
                self.isDown = None
        else:
            self.isDown = False

    def intersects(self, coords):
        """check if mouse pos is in [x1, y1, x2, y2]"""
        if (self.x >= coords[0]) and (self.x <= coords[2]) and (self.y >=
coords[1]) and (self.y <= coords[3]):
            return True
        else:
            return False
```

Keyboard.py

```
import pygame

class Keyboard:
    def __init__(self):
        """create a keyboard object to record pressed keys"""
        self.up : bool      # UpArrow / W
        self.left : bool    # LeftArrow / A
        self.down : bool    # DownArrow / S
        self.right : bool   # RightArrow / D
```

```
        self.reset : bool       # R
        self.solveAStar : bool  # Q
        self.solveBFS : bool    # F
        self.new : bool         # E


    def update(self):
        """get new data for the next frame"""
        pressed = pygame.key.get_pressed()
        isDown = False # enforce only one key at a time

        if (pressed[pygame.K_UP] or pressed[pygame.K_w]) and (isDown == False):
            isDown = True
            if self.up == False:
                self.up = True
            elif self.up:
                self.up = None
        else:
            self.up = False

        if (pressed[pygame.K_LEFT] or pressed[pygame.K_a]) and (isDown == False):
            isDown = True
            if self.left == False:
                self.left = True
            elif self.left == True:
                self.left = None
        else:
            self.left = False

        if (pressed[pygame.K_DOWN] or pressed[pygame.K_s]) and (isDown == False):
            isDown = True
            if self.down == False:
                self.down = True
            elif self.down:
                self.down = None
        else:
            self.down = False

        if (pressed[pygame.K_RIGHT] or pressed[pygame.K_d]) and (isDown == False):
            isDown = True
```

```
        if self.right == False:
            self.right = True
        elif self.right:
            self.right = None
    else:
        self.right = False

    if (pressed[pygame.K_r]) and (isDown == False):
        isDown = True
        if self.reset == False:
            self.reset = True
        elif self.reset:
            self.reset = None
    else:
        self.reset = False

    if (pressed[pygame.K_q]) and (isDown == False):
        isDown = True
        if self.solveAStar == False:
            self.solveAStar = True
        elif self.solveAStar:
            self.solveAStar = None
    else:
        self.solveAStar = False

    if (pressed[pygame.K_f]) and (isDown == False):
        isDown = True
        if self.solveBFS == False:
            self.solveBFS = True
        elif self.solveBFS:
            self.solveBFS = None
    else:
        self.solveBFS = False

    if (pressed[pygame.K_e]) and (isDown == False):
        isDown = True
        if self.new == False:
            self.new = True
        elif self.new:
            self.new = None
    else:
```

```
            self.new = False
```

Button.py

```
import pygame

class Button:
    def __init__(self, scr, coords, text = "", textSize = 25, frameColor = (255,
255, 255), textColor = (255, 255, 255)):
        self.coords = coords
        self.scr = scr

        self.text = text
        self.textSize = textSize
        self.textColor = textColor
        self.textObj = self.createText(text, (coords[0] + coords[2]) // 2,
(coords[1] + coords[3]) // 2, textSize, textColor)

        self.frameColor = frameColor
        self.defaultFrameColor = frameColor

    @staticmethod
    def createText(text, x, y, textSize, textColor):
        myFont = pygame.font.SysFont("Bahnschrift", textSize)
        text = str(text)
        text = myFont.render(text, True, textColor)
        textRect = text.get_rect()
        textRect.center = (x, y)
        return [text, textRect]

    def update(self, isPressed = False, isHovered = False, text = None, coords =
None):
        if text != None:
            self.text = text
        if coords != None:
            self.coords = coords

        if isPressed:
            self.frameColor = (0, 127, 255) #lightblue
        elif isHovered:
            self.frameColor = (0, 255, 0) #green
        else:
            self.frameColor = self.defaultFrameColor
```

```
        pygame.draw.rect(self.scr, (0, 0, 0), (self.coords[0], self.coords[1],
self.coords[2]-self.coords[0], self.coords[3]-self.coords[1]))
        pygame.draw.rect(self.scr, self.frameColor, (self.coords[0],
self.coords[1], self.coords[2]-self.coords[0], self.coords[3]-self.coords[1]), 1)
        self.textObj = self.createText(self.text, (self.coords[0] +
self.coords[2]) // 2, (self.coords[1] + self.coords[3]) // 2, self.textSize,
self.textColor)
        self.scr.blit(self.textObj[0], self.textObj[1])
```

GameScreen.py

```
import Mouse
import Keyboard
import GameBoard
import Button

class GameScreen:
    def __init__(self, scr):
        self.title = Button.Button(scr, coords=(250, 0, 500, 100),
text="8-Puzzle", textSize=30, frameColor=(0, 0, 0))
        self.gameBoard = GameBoard.GameBoard(scr)
        self.generateButton = Button.Button(scr, coords=(9*50, 2*50, 13*50,
2*50+60), text="New board (E)")
        self.resetButton = Button.Button(scr, coords=(9*50, 2*50+75, 13*50,
2*50+135), text="Reset puzzle (R)")
        self.solveButtonBFS = Button.Button(scr, coords=(9*50, 2*50+150, 13*50,
2*50+210), text="BFS solve (F)")
        self.solveButtonAStar = Button.Button(scr, coords=(9*50, 2*50+225, 13*50,
2*50+285), text="A* solve (Q)")
        self.messageBox = Button.Button(scr, coords=(25, 8*50+20, 725, 10*50-20),
text="", textSize=25, frameColor=(0, 0, 0))

    def update(self, keyboard, mouse):
        self.title.update(False, False)
        self.gameBoard.update(keyboard, mouse)

        if keyboard.new or (mouse.isDown and
mouse.intersects(self.generateButton.coords)):
            self.generateButton.update(True, False)
            self.gameBoard.generate()
            self.messageBox.update(False, False, "New board created")
        elif mouse.intersects(self.generateButton.coords):
```

```
            self.generateButton.update(False, True)
        else:
            self.generateButton.update(False, False)

        if keyboard.reset or (mouse.isDown and
mouse.intersects(self.resetButton.coords)):
            self.resetButton.update(True, False)
            self.gameBoard.reset()
            self.messageBox.update(False, False, "Current board reset")
        elif mouse.intersects(self.resetButton.coords):
            self.resetButton.update(False, True)
        else:
            self.resetButton.update(False, False)

        if keyboard.solveBFS or (mouse.isDown and
mouse.intersects(self.solveButtonBFS.coords)):
            self.solveButtonBFS.update(True, False)
            message = self.gameBoard.solveBFS()
            self.messageBox.update(False, False, message)
        elif mouse.intersects(self.solveButtonBFS.coords):
            self.solveButtonBFS.update(False, True)
        else:
            self.solveButtonBFS.update(False, False)

        if keyboard.solveAStar or (mouse.isDown and
mouse.intersects(self.solveButtonAStar.coords)):
            self.solveButtonAStar.update(True, False)
            message = self.gameBoard.solveAStar()
            self.messageBox.update(False, False, message)
        elif mouse.intersects(self.solveButtonAStar.coords):
            self.solveButtonAStar.update(False, True)
        else:
            self.solveButtonAStar.update(False, False)
```

GameBoard.py

```
import random
import array
import time
import Button

class GameBoard(object):
    """description of class"""
```

```
    def __init__(self, scr):
        self.scr = scr
        self.startingState : list
        self.numbers : list
        self.cells : list
        self.generate()

    def generate(self):
        self.startingState = list(range(9))
        self.startingState.append(self.startingState.pop(0))
        for i in range(100):
            self.startingState =
random.choice(self.getPossibleSteps(self.startingState, []))
        self.reset()

    def reset(self, fromStart = True):
        if fromStart:
            self.numbers = self.startingState[:]
        self.cells = []
        for i in range(9):
            currCoords = (102+i%3*100, 102+i//3*100, 198+i%3*100, 198+i//3*100)
            if self.numbers[i] == 0:
                self.cells.append(Button.Button(self.scr, currCoords))
            else:
                self.cells.append(Button.Button(self.scr, currCoords,
self.numbers[i], 40))

    def move(self, keyboard, mouse):
        """swap elements in self.numbers and self.cells according to inputs"""
        emptyInd = self.numbers.index(0)

        if emptyInd > 2:
            upNeighborInd = emptyInd - 3
        else:
            upNeighborInd = None

        if emptyInd < 6:
            downNeighborInd = emptyInd + 3
        else:
            downNeighborInd = None
```

```
        if emptyInd % 3 > 0:
            leftNeighborInd = emptyInd - 1
        else:
            leftNeighborInd = None

        if emptyInd % 3 < 2:
            rightNeighborInd = emptyInd + 1
        else:
            rightNeighborInd = None

        if downNeighborInd != None: #swap empty and down
            if keyboard.up or
(mouse.intersects(self.cells[downNeighborInd].coords) and mouse.isDown):
                self.numbers[emptyInd], self.numbers[downNeighborInd] =
self.numbers[downNeighborInd], self.numbers[emptyInd]
                self.cells[emptyInd], self.cells[downNeighborInd] =
self.cells[downNeighborInd], self.cells[emptyInd]

        if rightNeighborInd != None: #swap empty and right
            if keyboard.left or
(mouse.intersects(self.cells[rightNeighborInd].coords) and mouse.isDown):
                self.numbers[emptyInd], self.numbers[rightNeighborInd] =
self.numbers[rightNeighborInd], self.numbers[emptyInd]
                self.cells[emptyInd], self.cells[rightNeighborInd] =
self.cells[rightNeighborInd], self.cells[emptyInd]

        if upNeighborInd != None: #swap empty and up
            if keyboard.down or
(mouse.intersects(self.cells[upNeighborInd].coords) and mouse.isDown):
                self.numbers[emptyInd], self.numbers[upNeighborInd] =
self.numbers[upNeighborInd], self.numbers[emptyInd]
                self.cells[emptyInd], self.cells[upNeighborInd] =
self.cells[upNeighborInd], self.cells[emptyInd]

        if leftNeighborInd != None: #swap empty and left
            if keyboard.right or
(mouse.intersects(self.cells[leftNeighborInd].coords) and mouse.isDown):
                self.numbers[emptyInd], self.numbers[leftNeighborInd] =
self.numbers[leftNeighborInd], self.numbers[emptyInd]
                self.cells[emptyInd], self.cells[leftNeighborInd] =
self.cells[leftNeighborInd], self.cells[emptyInd]
```

```
        return emptyInd

    def update(self, keyboard, mouse):
        movedCellInd = self.move(keyboard, mouse)

        for i in range(9):
            currCoords = (102+i%3*100, 102+i//3*100, 198+i%3*100, 198+i//3*100)
            if movedCellInd == i:
                pressed = True
            else:
                pressed = False
            if mouse.intersects(currCoords):
                hovered = True
            else:
                hovered = False
            self.cells[i].update(pressed, hovered, None, currCoords)

#record in 20 experiments:
# - avg amount of steps needed
# - avg number of dead ends
# - avg number of generated states
# - avg amount of states saved in memory

    @staticmethod
    def getPossibleSteps(currState, visitedStates):
        emptyInd = currState.index(0)

        possibleSteps = []

        if emptyInd > 2:
            newState = currState[:]
            newState[emptyInd], newState[emptyInd - 3] = newState[emptyInd - 3],
newState[emptyInd]
            if not newState in visitedStates:
                possibleSteps.append(newState)

        if emptyInd < 6:
            newState = currState[:]
            newState[emptyInd], newState[emptyInd + 3] = newState[emptyInd + 3],
newState[emptyInd]
```

```
            if not newState in visitedStates:
                possibleSteps.append(newState)

        if emptyInd % 3 > 0:
            newState = currState[:]
            newState[emptyInd], newState[emptyInd - 1] = newState[emptyInd - 1],
newState[emptyInd]
            if not newState in visitedStates:
                possibleSteps.append(newState)

        if emptyInd % 3 < 2:
            newState = currState[:]
            newState[emptyInd], newState[emptyInd + 1] = newState[emptyInd + 1],
newState[emptyInd]
            if not newState in visitedStates:
                possibleSteps.append(newState)

        return possibleSteps

    def solveBFS(self):
        finalStateList = [1, 2, 3, 4, 5, 6, 7, 8, 0]
        finalState = []
        finalState = array.array('h')
        finalState.fromlist(finalStateList)
        startingStateArray = []
        startingStateArray = array.array('h')
        startingStateArray.fromlist(self.numbers) #fromlist(self.startingState)
        statesQueue = [startingStateArray]
        queuedStatesDepths = [0]
        visitedStates = []
        deadEnds = 0

        currState = statesQueue.pop(0)
        currDepth = queuedStatesDepths.pop(0)
        depthLimit = 24
        while currState != finalState and currDepth < depthLimit:

            newStates = self.getPossibleSteps(currState, visitedStates)
            if len(newStates) == 0:
                deadEnds += 1
            statesQueue.extend(newStates)
```

```
                queuedStatesDepths.extend([currDepth+1]*len(newStates))
                visitedStates.append(currState)

                currState = statesQueue.pop(0)
                currDepth = queuedStatesDepths.pop(0)

            if currState == finalState:
                message = "Solved with BFS. See 'BFSreport.txt'"
            else:
                message = f"{depthLimit} Depth limit reached. See 'BFSreport.txt'"

            outfile = open('BFSreport.txt', 'a')
            outfile.write(f'---------------\n')
            outfile.write(f'Max depth reached: {currDepth}\n')
            outfile.write(f'Reached depth limit (failed to find a solution):
{currDepth == depthLimit}\n')
            outfile.write(f'Dead ends encountered: {deadEnds}\n')
            outfile.write(f'States visited: {len(visitedStates)}\n')
            outfile.write(f'Unvisited states left in memory: {len(statesQueue)}\n')
            outfile.write(f'States generated: {len(visitedStates) +
len(statesQueue)}\n')
            outfile.write(f'States stored in memory: {len(visitedStates) +
len(statesQueue)}\n')
            outfile.write(f'---------------\n')
            outfile.close()

            self.numbers = currState.tolist()
            self.reset(False)

            return message

    @staticmethod
    def matches(currState):
        matchesNum = 0
        for i in range(8):
            if currState[i] == i + 1:
                matchesNum += 1
        if currState[8] == 0:
            matchesNum += 1
        return matchesNum
```

```
    def solveAStar(self):
        finalStateList = [1, 2, 3, 4, 5, 6, 7, 8, 0]
        finalState = []
        finalState = array.array('h')
        finalState.fromlist(finalStateList)
        startingStateArray = []
        startingStateArray = array.array('h')
        startingStateArray.fromlist(self.numbers) #fromlist(self.startingState)

        statesQueue = [[startingStateArray], [], [], [], [], [], [], [], [], []] #
0, 1, 2, 3, 4, 5, 6, 7, 8, 9 off
        queuedStatesDepths = [[0], [], [], [], [], [], [], [], [], []]

        visitedStates = []
        deadEnds = 0

        currState = statesQueue[0].pop(0)
        currDepth = queuedStatesDepths[0].pop(0)
        depthLimit = 31

        while currState != finalState:
            newStates = self.getPossibleSteps(currState, visitedStates)
            if len(newStates) == 0:
                deadEnds += 1
            for state in newStates:
                ind = 9 - self.matches(state)
                statesQueue[ind].append(state)
                queuedStatesDepths[ind].append(currDepth+1)
            visitedStates.append(currState)

            i = 0
            while len(statesQueue[i]) == 0:
                i += 1

            currState = statesQueue[i].pop(0)
            currDepth = queuedStatesDepths[i].pop(0)

        if currState == finalState:
            message = "Solved with A*. See 'ASTARreport.txt'"
        else:
            message = f"{depthLimit} Depth limit reached. See 'ASTARreport.txt'"
```

```
        statesLeft = sum(len(statesQueue[i]) for i in range(10))
        outfile = open('ASTARreport.txt', 'a')
        outfile.write(f'---------------\n')
        outfile.write(f'Max depth reached: {currDepth}\n')
        outfile.write(f'Reached depth limit (failed to find a solution):
{currDepth >= depthLimit}\n')
        outfile.write(f'Dead ends encountered: {deadEnds}\n')
        outfile.write(f'States visited: {len(visitedStates)}\n')
        outfile.write(f'Unvisited states left in memory: {statesLeft}\n')
        outfile.write(f'States generated: {len(visitedStates) + statesLeft}\n')
        outfile.write(f'States stored in memory: {len(visitedStates) +
statesLeft}\n')
        outfile.write(f'---------------\n')
        outfile.close()

        self.numbers = currState.tolist()
        self.reset(False)

        return message
```

1.2.2 Приклади роботи

На рисунках 3.1-3.8 показані приклади роботи програми для різних алгоритмів пошуку.

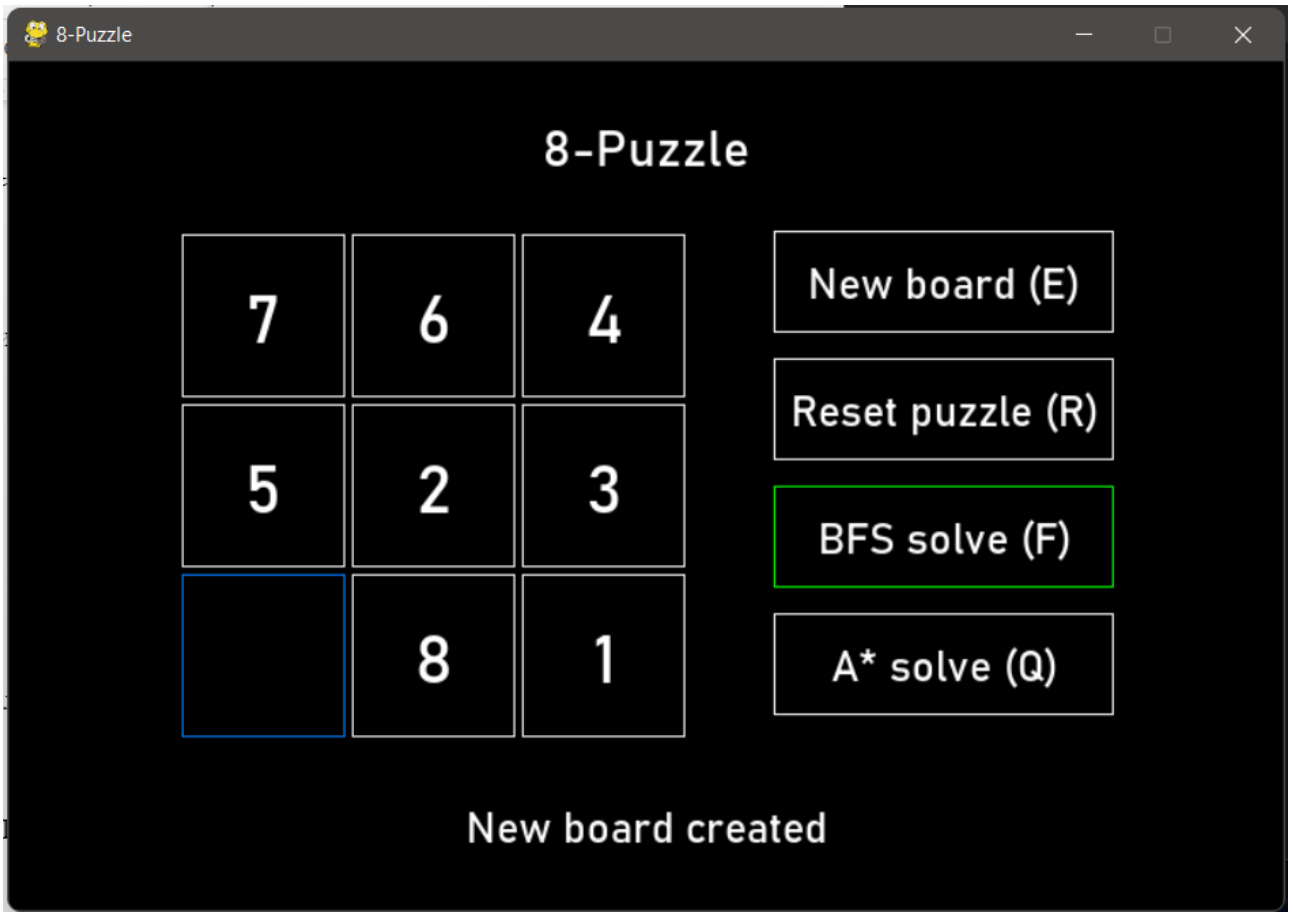

Рисунок 3.1 – Створено нове ігрове поле

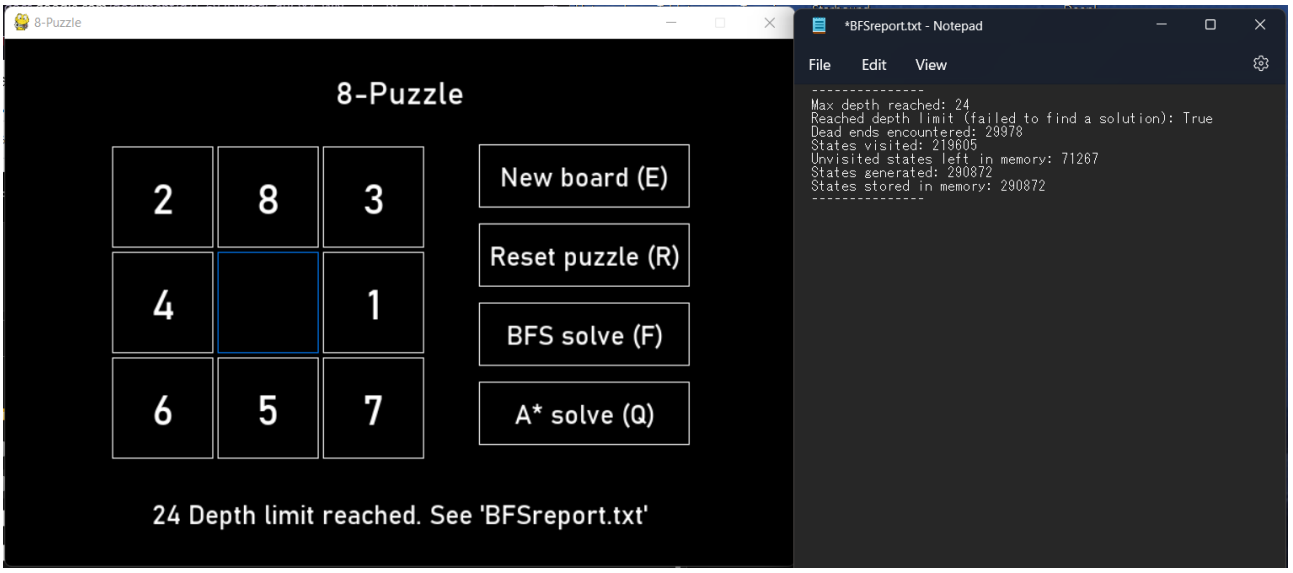

Рисунок 3.2 – Алгоритм BFS дійшов до ліміту глибини

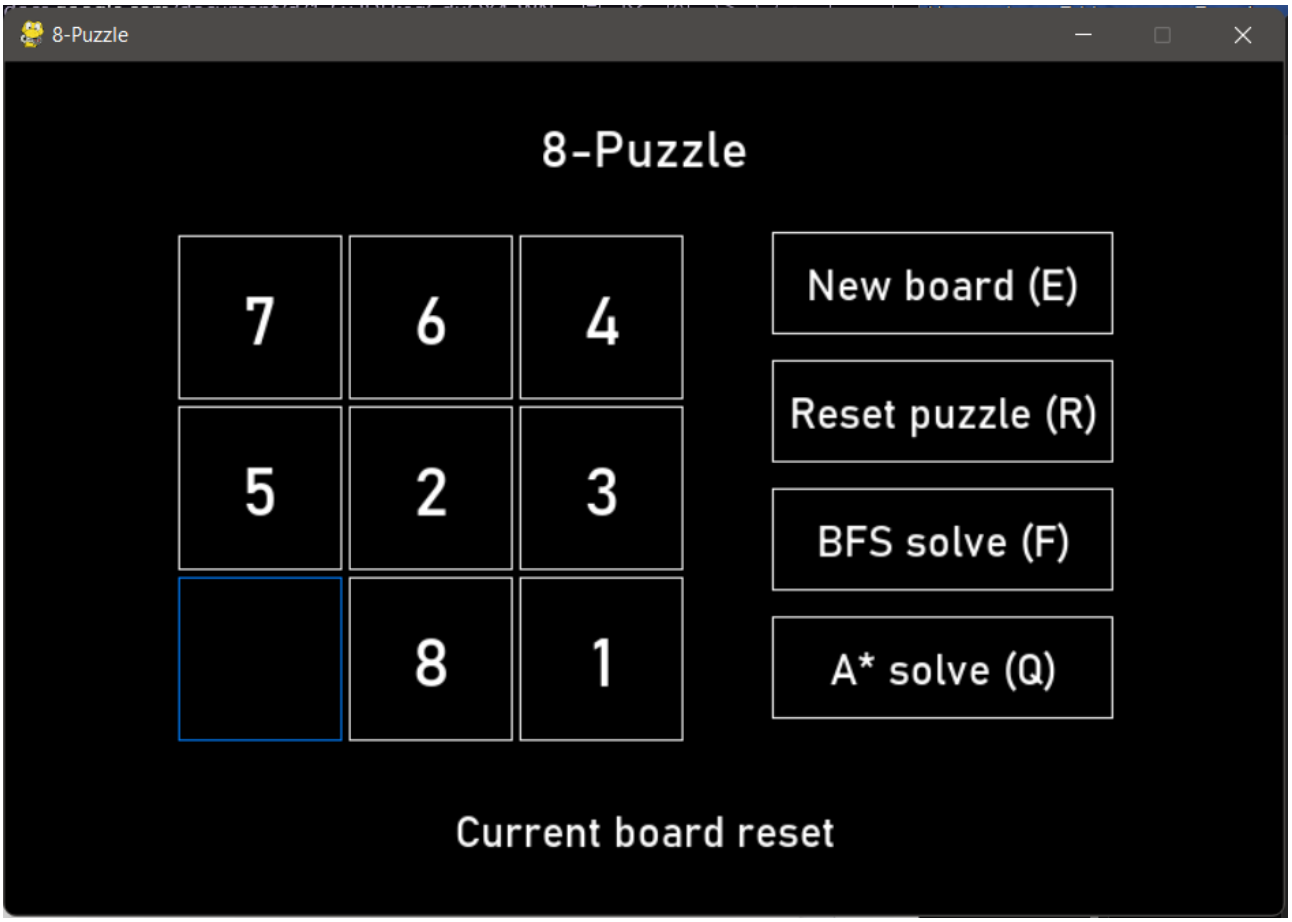

Рисунок 3.3 – Поле повернено до початкового стану

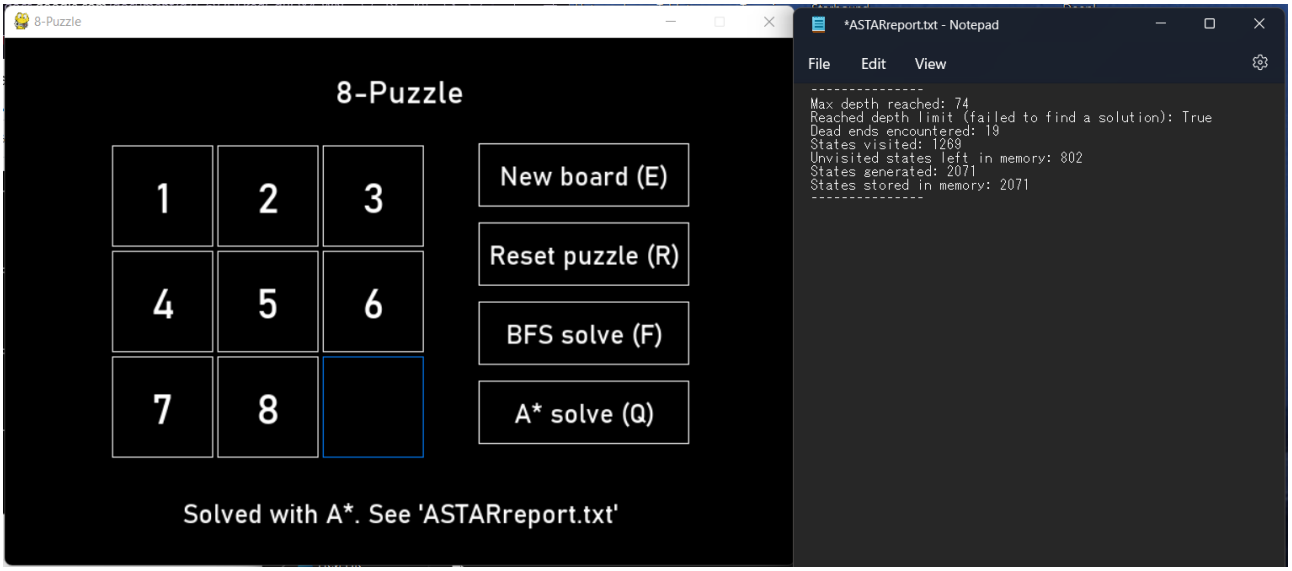

Рисунок 3.4 – Алгоритм A*

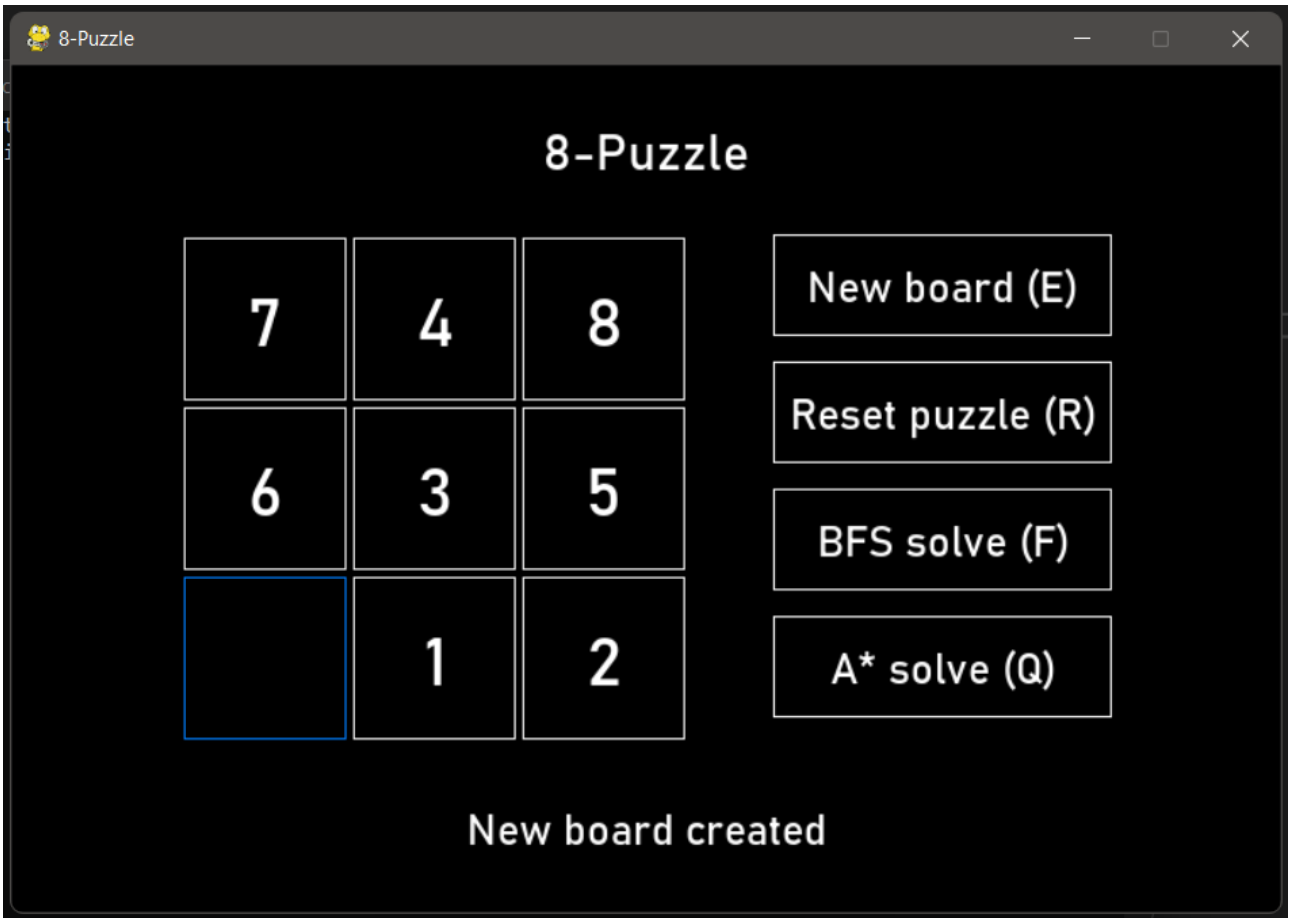

Рисунок 3.5 – Створено нове ігрове поле

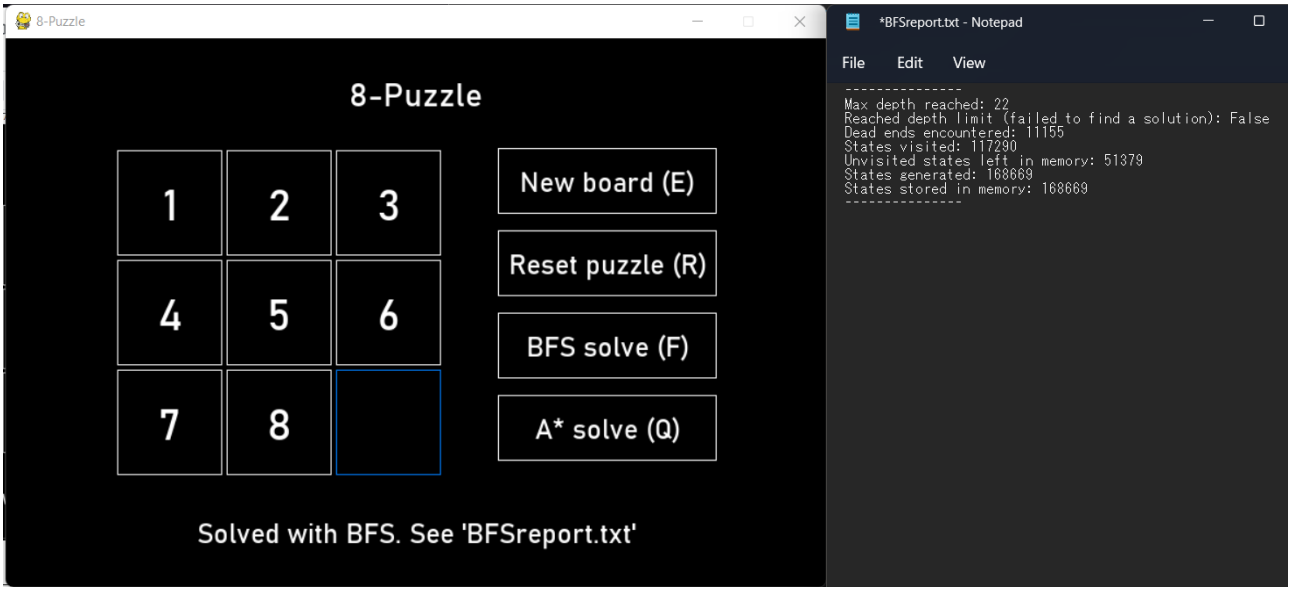

Рисунок 3.6 – Алгоритм BFS упорався з завданням

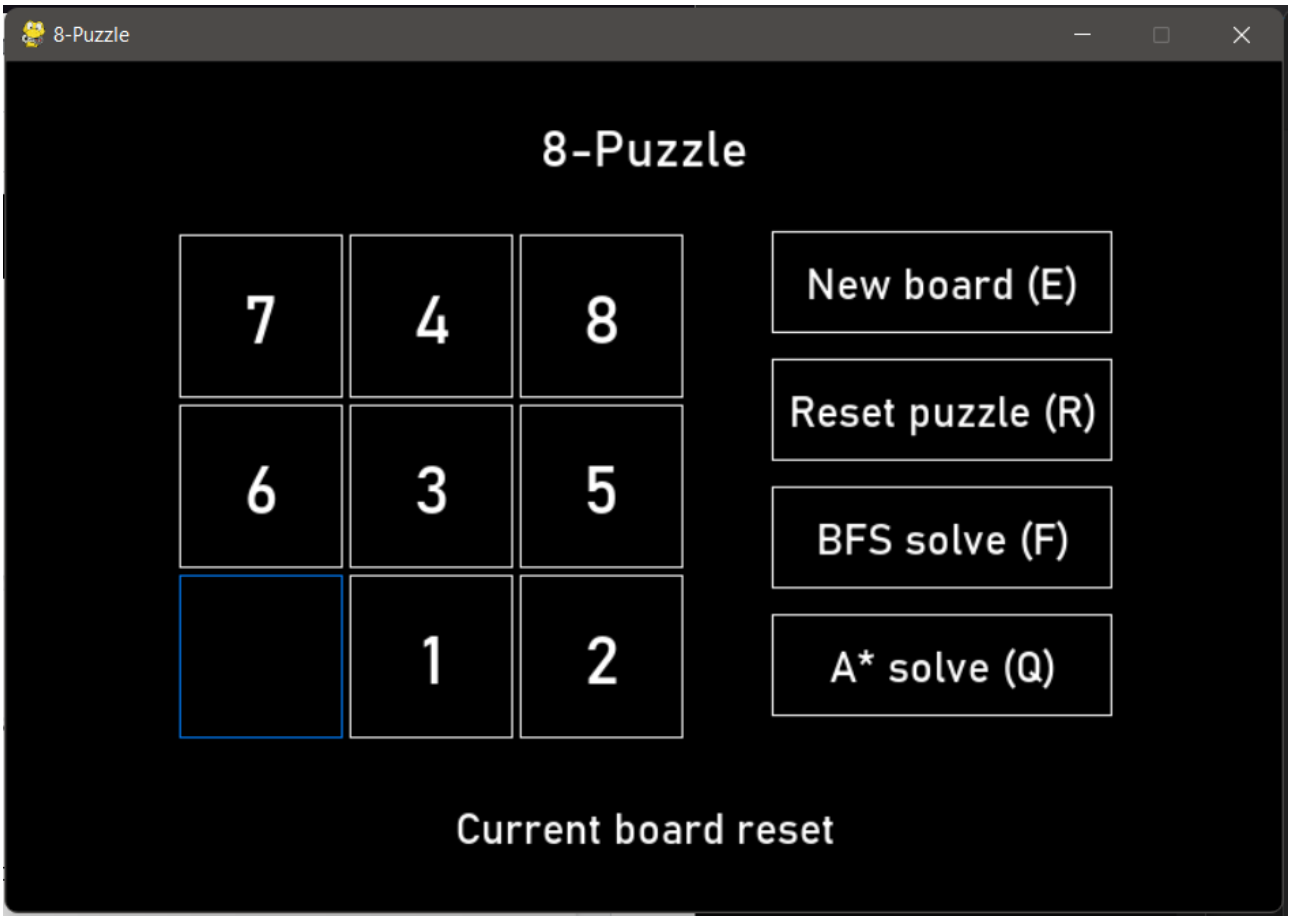

Рисунок 3.7 – Поле повернено до початкового стану

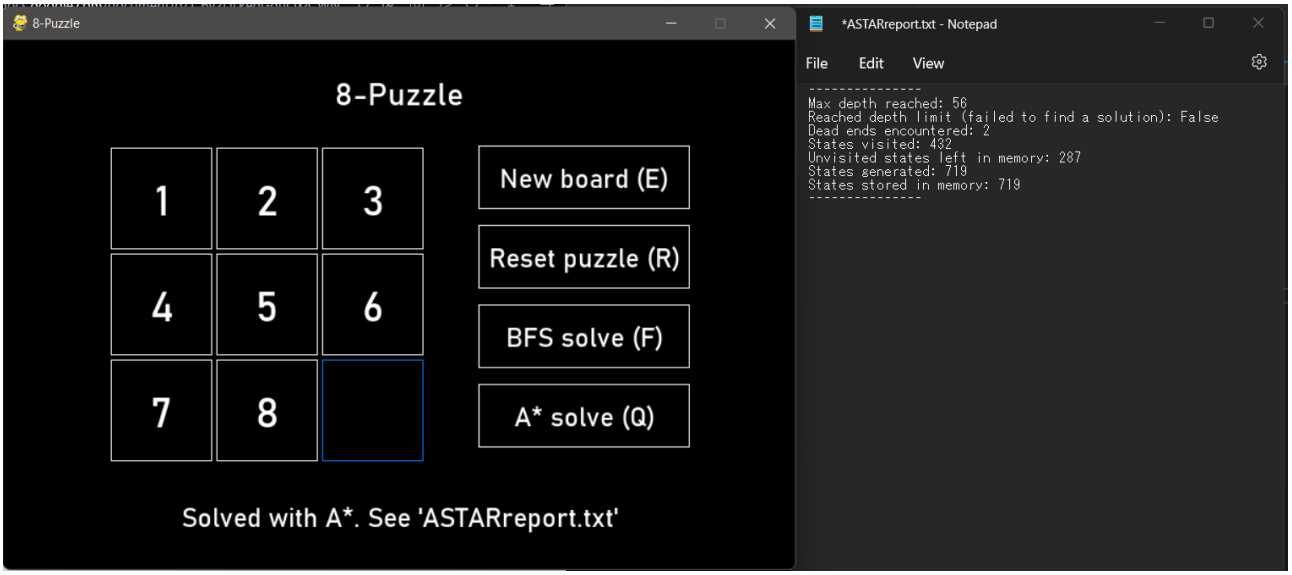

Рисунок 3.8 – Алгоритм A* знайшов розв’язок швидше, але використав більше кроків

1.3 Дослідження алгоритмів

В таблиці 3.1 наведені характеристики оцінювання алгоритму Breadth-first search, задачі 8-puzzle для 20 початкових станів.

Таблиця 3.1 – Характеристики оцінювання алгоритму Breadth-first search, задачі 8-puzzle для 20 початкових станів.

Початкові стани	Знайдено розв’язок	Найбільша глибина	К-сть гл. кутів	Всього станів пройдено	Всього станів у пам'яті
Стан 1	Ні	24	39184	255781	329332
Стан 2	Так	18	966	23317	37470
Стан 3	Так	20	5282	65043	95934
Стан 4	Ні	24	29978	219605	290872
Стан 5	Так	22	29978	163437	222137
Стан 6	Так	8	8	292	485
Стан 7	Так	10	16	596	977
Стан 8	Так	18	889	28298	45440
Стан 9	Ні	24	29978	219605	290872
Стан 10	Так	20	7931	79023	111697
Стан 11	Ні	24	39184	255781	329332
Стан 12	Так	20	6338	70608	102198
Стан 13	Так	20	3754	56998	86799
Стан 14	Ні	24	39184	255781	329332
Стан 15	Так	20	7590	77151	109517
Стан 16	Так	20	3523	55939	85636
Стан 17	Так	18	2180	32762	49297
Стан 18	Так	18	803	22331	36258
Стан 19	Так	16	662	16872	26068
Стан 20	Так	12	23	1355	2279
Середнє значення	75% Так 25% Ні	19	12372.55	95028.75	129096

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

Таблиця 3.2 – Характеристики оцінювання оцінювання алгоритму A* search, задачі 8-puzzle для 20 початкових станів.

Початкові стани	Знайдено розв’язок	Найбільша глибина	К-сть гл. кутів	Всього станів пройдено	Всього станів у пам'яті
Стан 1	Так	36	6	743	1221
Стан 2	Так	26	3	237	391
Стан 3	Так	32	2	254	425
Стан 4	Так	66	20	1636	2654
Стан 5	Так	48	21	1464	2383
Стан 6	Так	42	3	590	975
Стан 7	Так	50	11	891	1454
Стан 8	Так	46	13	879	1447
Стан 9	Так	56	19	1449	2373
Стан 10	Так	50	11	1159	1900
Стан 11	Так	38	2	400	665
Стан 12	Так	68	8	839	1371
Стан 13	Так	46	20	1479	2419
Стан 14	Так	54	25	2417	3913
Стан 15	Так	48	22	1559	2537
Стан 16	Так	48	12	1167	1911
Стан 17	Так	62	42	2150	3488
Стан 18	Так	24	4	534	879
Стан 19	Так	18	1	177	298
Стан 20	Так	44	6	841	1360
Середнє значення	100% Так	45.1	12.55	1043.25	1703.2

Висновок

При виконанні даної лабораторної роботи було розглянуто роботу алгоритмів пошуку для розв’язання задачі 8-puzzle: неінформативний алгоритм пошуку в ширину (Breadth-first search) та інформативний пошук A* за кількістю фішок не на своєму місці.

Алгоритм пошуку в ширину є повним та одразу знаходить найкоротший шлях розв’язання, але працює повільно (складність алгоритму експоненціальна), через що потрібно було встановити обмеження на глибину пошуку. В результаті рішення було не знайдено у 25% випадків при обмеженні у 24 кроки (для розв’язку 8-puzzle необхідно зробити максимум 31 крок).

Використана варіація алгоритму пошуку A* зупиняється на першому знайденому рішенні, яке часто не є найкоротшим. Натомість швидкість роботи цього алгоритму значно краща, ніж у повних алгоритмів.

До алгоритмів було додано модифікацію: пройдені стани додаються до списку станів, які програма вважатиме недоцільними для перевірки, що частково дозволило уникнути перевірку однакових станів та суттєво покращило швидкість роботи.

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%.