

ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH



NHẬP MÔN TRÍ TUỆ NHÂN TẠO (CO3061)

BÁO CÁO BÀI TẬP LỚN SỐ 1 GIẢI THUẬT TÌM KIẾM N-PUZZLE, SUDOKU VÀ PATH FINDING

Giảng viên hướng dẫn: Vương Bá Thịnh
Lớp: L01
SV thực hiện: Bùi Tiến Dũng – 2113055
Hoàng Minh Hải Đăng – 2110120
Nguyễn Hồ Nhật Hà – 2111118
Trương Thành Long - 2211909
Nguyễn Thái Sơn - 2112198

Tp. Hồ Chí Minh, Tháng 11/2024



Mục lục

1	Bài toán N-puzzle	3
1.1	Giới thiệu bài toán	3
1.2	Xác định các yếu tố cho bài toán tìm kiếm	3
1.2.1	Định nghĩa trạng thái bài toán	3
1.2.2	Xác định trạng thái khởi đầu và trạng thái mục tiêu	4
1.2.3	Xác định luật di chuyển	4
1.2.4	Hàm sinh trạng thái khởi đầu	5
1.2.5	Hàm sinh trạng thái mục tiêu	7
1.3	Định dạng testcase kiểm thử	7
1.3.1	Input kiểm tra giải thuật	7
1.3.2	Output ghi nhận kết quả	7
1.4	Giải thuật DFS	8
1.4.1	Hiện thực giải thuật DFS	8
1.4.2	Đo đặc thời gian và bộ nhớ tiêu tốn	9
1.4.2.a	Đối với bài toán 8-puzzle	9
1.4.2.b	Đối với bài toán 15-puzzle	10
1.4.3	Nhận xét	10
1.5	Giải thuật IDS	10
1.5.1	Hiện thực giải thuật IDS	10
1.5.2	Đo đặc thời gian bộ nhớ tiêu tốn	11
1.5.2.a	Đối với bài toán 8-puzzle	11
1.5.2.b	Đối với bài toán 15-puzzle (đơn giản)	11
1.5.2.c	Đối với bài toán 15-puzzle (phức tạp)	12
1.5.3	Nhận xét	12
1.6	So sánh kết quả 2 giải thuật	13
1.7	Giao diện trò chơi	13
2	Bài toán Sudoku	14
2.1	Giới thiệu bài toán	14
2.2	Xác định các yếu tố cho bài toán tìm kiếm	15
2.2.1	Không gian trạng thái	15
2.2.2	Định nghĩa trạng thái bài toán	15
2.2.3	Trạng thái khởi đầu và trạng thái mục tiêu	15
2.2.4	Luật di chuyển	15
2.2.5	Hàm lượng giá trạng thái	15
2.2.6	Hàm sinh trạng thái khởi đầu	16
2.3	Giải bài toán Sudoku bằng giải thuật di truyền	17
2.3.1	Giải thuật di truyền	17
2.3.2	Ứng dụng giải bài toán Sudoku	17
2.4	Kết quả thực thi và đánh giá hiệu năng giải thuật	21
2.4.1	Mức độ dễ	21
2.4.2	Mức độ trung bình	22
2.4.3	Mức độ khó	23
2.4.4	Thống kê kết quả và nhận xét	24
2.5	Giao diện trò chơi	24



3	Bài toán Path Finding	26
3.1	Giới thiệu bài toán	26
3.2	Xác định các yếu tố cho bài toán tìm kiếm	26
3.2.1	Không gian trạng thái	26
3.2.2	Định nghĩa trạng thái bài toán	27
3.2.3	Trạng thái khởi đầu và trạng thái mục tiêu	28
3.2.4	Xác định luật di chuyển	28
3.2.5	Hàm sinh trạng thái khởi đầu	28
3.2.6	Hàm sinh trạng thái mục tiêu	29
3.2.7	Hàm lượng giá trạng thái	30
3.2.8	Hiện thực giải thuật A*	31
3.3	Định dạng testcase kiểm thử	33
3.3.1	Input kiểm tra giải thuật	33
3.3.2	Kết quả	34
3.4	Đo đặc thời gian và bộ nhớ tiêu tốn	34
3.4.1	Test-case 1: Trường hợp kích thước bản đồ 20 * 20	34
3.4.2	Test-case 2: Trường hợp kích thước bản đồ 100 * 100	36
3.4.3	Test-case 3: Trường hợp kích thước bản đồ 2000 * 2000	37
3.4.4	Test-case 4: Trường hợp kích thước bản đồ 5000 * 5000	38
3.4.5	Test-case 5: Trường hợp kích thước bản đồ 10000 * 10000	38
3.4.6	Thống kê kết quả và nhận xét	38
	Tài liệu tham khảo	39



1 Bài toán N-puzzle

1.1 Giới thiệu bài toán

N-puzzle là một trò chơi xếp hình cổ điển và cũng là một bài toán kinh điển trong lĩnh vực trí tuệ nhân tạo. Trò chơi bao gồm một bảng vuông được chia thành $N + 1$ ô vuông nhỏ hơn. Trong đó, có một ô trống và các ô còn lại chứa các số từ 1 đến N .

12	14	6	4
2	5	7	1
	3	8	15
10	9	11	13

Hình 1: Một đề bài của bài toán N-Puzzle với $N = 15$

Mục tiêu của trò chơi là sắp xếp các số trên bảng theo một thứ tự nhất định, thường là theo thứ tự tăng dần từ trái sang phải và từ trên xuống dưới, bằng cách di chuyển ô trống.

1.2 Xác định các yếu tố cho bài toán tìm kiếm

1.2.1 Định nghĩa trạng thái bài toán

Trạng thái của bài toán biểu diễn một tình huống có thể xảy ra trong quá trình trò chơi diễn ra thông qua một danh sách có kích thước N tương ứng với số ô vuông trong bài toán. Các phần tử trong danh sách nhận giá trị x sao cho $x \in 0, \dots, N$ và các phần tử này đều lần lượt chứa các giá trị x đôi một khác nhau. Trong đó, phần tử có giá trị bằng 0 biểu diễn cho vị trí của ô trống trong bài toán N-Puzzle và đây cũng là phần tử duy nhất được phép di chuyển trong bài toán này.

12 14 6 4
2 5 7 1
10 9 11 13

Hình 2: Danh sách biểu diễn trạng thái của bài toán ở Hình 1



1.2.2 Xác định trạng thái khởi đầu và trạng thái mục tiêu

Trạng thái khởi đầu là trạng thái đầu tiên mà đề bài đưa ra. Ví dụ danh sách ở hình 2 chính là trạng thái khởi đầu cho bài toán 15-Puzzle ở hình 1.

Trạng thái mục tiêu của bài toán N-Puzzle là trạng thái mà các số trên bảng được sắp xếp theo thứ tự tăng dần từ trái sang phải, từ trên xuống dưới và ô trống sẽ nằm ở vị trí cuối cùng trong bảng. Trạng thái mục tiêu được biểu diễn dưới dạng danh sách có thứ tự $[1, 2, \dots, N, 0]$.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Hình 3: Trạng thái mục tiêu của bài toán 15 puzzle

1.2.3 Xác định luật di chuyển

Luật di chuyển cho phép thực hiện chuyển từ một trạng thái này sang một trạng thái tiếp theo của trò chơi. Trong trò chơi N-Puzzle, người chơi có thể thay đổi trạng thái của trò chơi bằng cách thay đổi vị trí của ô trống trên bảng. Như vậy, luật di chuyển sẽ là tập hợp các quy tắc sao cho việc di chuyển ô trống trong hình vuông là hợp lệ, các quy tắc bao gồm:

1. Nếu ô trống đang ở hàng trên cùng của hình vuông, nó sẽ không thể di chuyển theo hướng đi lên.
2. Nếu ô trống đang ở sát lề bên phải của hình vuông, nó sẽ không thể di chuyển sang phải.
3. Nếu ô trống đang ở sát lề bên trái của hình vuông, nó sẽ không thể di chuyển sang trái.
4. Nếu ô trống đang ở hàng dưới cùng của hình vuông, nó sẽ không thể di chuyển xuống dưới.

Bên cạnh đó, để giải thuật tìm kiếm hoạt động hiệu quả hơn, chúng ta sẽ thêm vào một số luật di chuyển bổ sung để tránh việc sinh ra các trạng thái đã thực hiện trước đó. Các luật di chuyển bổ sung này sẽ không cho phép ô trống di chuyển theo hướng ngược lại để trở về trạng thái trước đó. Các luật này bao gồm:

1. Nếu trạng thái hiện tại được tạo thành từ việc di chuyển ô trống sang bên trái, ô trống hiện tại sẽ không được di chuyển sang bên phải.
2. Nếu trạng thái hiện tại được tạo thành từ việc di chuyển ô trống sang bên phải, ô trống hiện tại sẽ không được di chuyển sang bên trái.
3. Nếu trạng thái hiện tại được tạo thành từ việc di chuyển ô trống lên trên, ô trống hiện tại sẽ không được di chuyển xuống dưới.



4. Nếu trạng thái hiện tại được tạo thành từ việc di chuyển ô trống xuống dưới, ô trống hiện tại sẽ không được di chuyển lên trên.

Các luật di chuyển này sẽ được hiện thực thông qua phương thức `available_moves` trong class `State` như sau:

```
# Xác định hành động di chuyển hợp lệ cho các trạng thái hiện tại
def available_moves(self, x):
    moves = ['Left', 'Right', 'Up', 'Down']
    size = self.size
    # 0 trong không thể di sang trái
    if x % size == 0:
        moves.remove('Left')
    # 0 trong không thể di sang phải
    if x % size == size - 1:
        moves.remove('Right')
    # 0 trong không thể di lên trên
    if x - size < 0:
        moves.remove('Up')
    # 0 trong không thể di xuống dưới
    if x + size >= size * size:
        moves.remove('Down')

    # Các luật di chuyển hỗ trợ tìm kiếm nhanh hơn
    # Loại bỏ các hành động ngược lại của hành động trước đó
    if self.action == 'Left' and 'Right' in moves:
        moves.remove('Right')
    elif self.action == 'Right' and 'Left' in moves:
        moves.remove('Left')
    elif self.action == 'Up' and 'Down' in moves:
        moves.remove('Down')
    elif self.action == 'Down' and 'Up' in moves:
        moves.remove('Up')

    return moves
```

1.2.4 Hàm sinh trạng thái khởi đầu

Để đảm bảo bài toán N-puzzle được đưa ra là có thể giải được (có tồn tại lời giải), ta phải hiện thực hàm sinh trạng thái khởi đầu cho bài toán này. Hàm sinh được hiện thực dựa trên ý tưởng như sau:

1. Sinh ngẫu nhiên một trạng thái của bài toán N-puzzle với kích thước k ($N = k*k - 1$) được chỉ định.
2. Kiểm tra xem với trạng thái ban đầu đó, bài toán có tồn tại lời giải hay không. Nếu có, trả về trạng thái này làm trạng thái ban đầu của bài toán, nếu không thì thực hiện lại bước 1.

Để xác định một bài toán N-puzzle (ví dụ 8-puzzle, 15-puzzle) có lời giải hay không, ta cần xem xét hai yếu tố chính:

- **Số lần nghịch đảo (inversions):** Trong một cấu hình của N-puzzle, "nghịch đảo" xảy ra khi một số lớn hơn nằm trước một số nhỏ hơn (ngoại trừ ô trống) khi các ô được đọc



theo thứ tự từ trên xuống dưới và từ trái qua phải. Để kiểm tra bài toán có thể giải được hay không, ta cần tính tổng số nghịch đảo trong cấu hình ban đầu của puzzle.

- **Vị trí của ô trống:** Đối với puzzle có kích thước $k \times k$, vị trí của ô trống cũng ảnh hưởng đến tính giải được của bài toán.

Quy tắc xác định một puzzle là có thể giải được hay không như sau:

- **Đối với puzzle có kích thước k lẻ** (ví dụ 8-puzzle): Puzzle có thể giải được nếu tổng số nghịch đảo là số chẵn.
- **Đối với puzzle có kích thước k chẵn** (ví dụ 15-puzzle): Puzzle có thể giải được nếu:
 - Số nghịch đảo là **lẻ** và ô trống nằm trên một hàng **chẵn** tính từ dưới lên.
 - Số nghịch đảo là **chẵn** và ô trống nằm trên một hàng **lẻ** tính từ dưới lên.

Hàm sinh trạng thái ban đầu cho bài toán N-puzzle và các hàm hỗ trợ của nó được viết trong ngôn ngữ lập trình Python như sau:

```
# Phương thức hỗ trợ tính nghịch đảo của puzzle
def count_inversions(arr):
    inversions = 0
    flattened = [num for num in arr if num != 0] # Bỏ ô trống
    for i in range(len(flattened)):
        for j in range(i + 1, len(flattened)):
            if flattened[i] > flattened[j]:
                inversions += 1
    return inversions

# Kiểm tra xem puzzle có thể giải được hay không
def is_solvable(puzzle, k):
    inversions = count_inversions(puzzle.flatten())
    if k % 2 == 1:
        # Kích thước lẻ: Số lần nghịch đảo phải là chẵn
        return inversions % 2 == 0
    else:
        # Kích thước chẵn: Số lần nghịch đảo
        # và hàng của ô trống (tính từ dưới lên) phải có tổng là lẻ
        blank_row = np.where(puzzle == 0)[0][0]
        return (inversions + blank_row) % 2 == 1

# Hàm sinh trạng thái ban đầu
def generate_puzzle(k):
    # Sinh ngẫu nhiên ma trận k*k và kiểm tra tính khả giải
    while True:
        puzzle = np.arange(k * k)
        np.random.shuffle(puzzle)
        puzzle = puzzle.reshape((k, k))
        if is_solvable(puzzle, k):
            return puzzle
```



1.2.5 Hàm sinh trạng thái mục tiêu

Trạng thái mục tiêu của bài toán N-Puzzle là trạng thái mà các số trên bảng được sắp xếp theo thứ tự tăng dần từ trái sang phải, từ trên xuống dưới và ô trống sẽ nằm ở vị trí cuối cùng trong bảng. Trong bài tập lớn này, nhóm em hiện thực một class State để đại diện cho một trạng thái của bài toán và trạng thái mục tiêu sẽ được thiết lập là một thuộc tính của class State nhằm giúp cho việc kiểm tra trạng thái hiện tại có phải là trạng thái mục tiêu chưa dễ dàng hơn. Đoạn mã sinh ra trạng thái mục tiêu cho bài toán dựa trên kích thước bài toán (k) và hàm kiểm tra trạng thái mục tiêu như sau:

```
class State:
    def __init__(self, state, parent, action, depth, size):
        self.state = state
        self.parent = parent
        self.action = action
        self.depth = depth
        self.size = size
        # Goal state: [1,2,...,k*k - 1,0]
        self.goal = list(range(1, size * size)) + [0]

    def check(self):
        # Kiểm tra trạng thái hiện tại có phải goal state chưa
        return self.state == self.goal
```

1.3 Định dạng testcase kiểm thử

1.3.1 Input kiểm tra giải thuật

Bên cạnh việc tự sinh ngẫu nhiên một trạng thái khởi đầu có lời giải để thực hiện giải thuật, chương trình cũng cho phép nhận đầu vào qua file text "input.txt" chứa một hoặc nhiều đề bài, định dạng file "input.txt" như sau:

- Dòng đầu tiên là một số nguyên thể hiện số lượng đề bài trong file.
- Các dòng tiếp theo thể hiện cho các đề bài.
- Dòng đầu tiên của mỗi đề bài là một số nguyên thể hiện cho kích thước k của bài toán.
- k dòng kế tiếp là các dãy chứa k số biểu diễn cho mỗi hàng của puzzle cần giải.
- Tiếp theo là các thông tin cho các đề bài khác (nếu có) với định dạng tương tự.

1.3.2 Output ghi nhận kết quả

Lời giải trò chơi sẽ được ghi nhận lại trong một file text có tên "output.txt", chứa số lời giải tương ứng với số lượng đề bài của file Input. Mỗi lời giải bao gồm các thông tin như sau:

- Path to the goal state: Bao gồm các bước di chuyển ô trống mà người chơi cần thực hiện để đạt được trạng thái mục tiêu.
- Numbers of action: Số hành động cần thực hiện để đạt được trạng thái mục tiêu.
- Number of explored nodes: là số trạng thái mà giải thuật đã duyệt qua trong quá trình tìm kiếm.



- Execution time: Là thời gian tiêu tốn mà giải thuật sử dụng để tìm ra lời giải cho đề bài đó.

```
# input.txt
1 2
2 3
3 1 8 3
4 5 7
5 6 2 0
6 3
7 1 2 8
8 7 5 6
9 4 3 0
10

# output.txt
1 Action: ['Up', 'Left', 'Down', 'Right', 'Up', 'Up', 'Left',
2 'Down', 'Down', 'Left', 'Up', 'Up', 'Right', 'Right', 'Down',
3 'Down', 'Left', 'Up', 'Up', 'Left', 'Down', 'Down', 'Right',
4 'Up', 'Right', 'Down']
5 Number of actions: 26
6 Number of explored nodes is: 28538
7 Time: 0.2179 second
8 -----
9 Action: ['Left', 'Up', 'Right', 'Down', 'Left', 'Up', 'Left',
10 'Up', 'Right', 'Right', 'Down', 'Down', 'Left', 'Up', 'Up',
11 'Left', 'Down', 'Down', 'Right', 'Up', 'Right', 'Down']
12 Number of actions: 22
13 Number of explored nodes is: 14290
14 Time: 0.3316 second
15 -----
16
```

Hình 4: File input gồm 2 đề bài và file output tương ứng

1.4 Giải thuật DFS

1.4.1 Hiện thực giải thuật DFS

```
def DFS(given_state, size):
    # Khởi tạo nút gốc
    root = State(given_state, None, None, 0, size)

    # Kiểm tra xem trạng thái đầu có phải là trạng thái mục tiêu không
    if root.check():
        return root.solution(), 0

    # Tạo ngăn xếp (stack) cho các nút sẽ duyệt
    stack = [root]
    visited = set()

    # Duyệt qua stack
    while stack:
        current_node = stack.pop()
        if current_node.check():
            return current_node.solution(), len(visited)
        state_hash = hash(tuple(current_node.state))
        visited.add(state_hash)

        # Sinh ra các trạng thái kế tiếp từ trạng thái hiện tại
        children = current_node.expand()

        for child in children:
            child_hash = hash(tuple(child.state))
            # Kiểm tra xem trạng thái đã được duyệt qua chưa
            # nếu chưa thì thêm vào stack
            if child_hash not in visited:
                stack.append(child)

    # Nếu không có lời giải, trả về None và số trạng thái đã duyệt
    return None, len(visited)
```



Giải thuật được hiện thực thông qua hàm DFS có cơ chế hoạt động như sau:

1. Bắt đầu từ nút gốc giải thuật sẽ tiến hành so sánh với trạng thái mục tiêu. Nếu đã thỏa mãn trạng thái mục tiêu thì dừng tìm kiếm, ngược lại thì sinh ra các nút tiếp theo và đưa các nút đó vào stack.
2. Tiếp tục pop từng nút trong stack để tiến hành kiểm tra như ở bước 1 cho đến khi đạt được trạng thái mục tiêu hoặc khi không còn trạng thái nào để kiểm tra nữa.

1.4.2 Đo đặc thời gian và bộ nhớ tiêu tốn

Việc tính toán thời gian thực thi của giải thuật được hỗ trợ thông qua thư viện time của Python. Tương tự, việc tính toán bộ nhớ tiêu tốn cần đến thư viện memory_profiler, các hàm tìm kiếm cần được gói vào hàm profile của thư viện.

1.4.2.a Đối với bài toán 8-puzzle

Test case:

1
3
1 8 3
4 5 7
6 2 0

Output:

Number of actions: 114520

Number of explored nodes is: 139261

Time: 0.7197 second

Line #	Mem usage	Increment	Occurrences	Line Contents
83	47.7 MiB	47.7 MiB	1	@profile
84				def DFS(given_state, size):
85				# Khởi tạo nút gốc
86	47.7 MiB	0.0 MiB	1	root = State(given_state, None, None, 0, size)
87				
88				# Kiểm tra xem trạng thái hiện tại có phải là trạng thái mục tiêu hay không, nếu phải thì trả về lời giải bài toán
89	47.7 MiB	0.0 MiB	1	if root.check():
90				return root.solution(), 0
91				
92				# Tạo ngăn xếp (stack) cho các nút sẽ duyệt
93	47.7 MiB	0.0 MiB	1	stack = [root]
94	47.7 MiB	0.0 MiB	1	visited = set()
95				
96				# Duyệt qua stack
97	142.3 MiB	0.0 MiB	139429	while stack:
98	142.3 MiB	0.0 MiB	139429	current_node = stack.pop()
99	142.3 MiB	0.0 MiB	139429	if current_node.check():
100	143.2 MiB	1.0 MiB	1	return current_node.solution(), len(visited)
101	142.3 MiB	0.0 MiB	139428	state_hash = hash(tuple(current_node.state))
102	142.3 MiB	5.1 MiB	139428	visited.add(state_hash)
103				
104				# Tạo các nút con
105	142.3 MiB	78.3 MiB	139428	children = current_node.expand()
106				
107	142.3 MiB	10.5 MiB	385440	for child in children:
108	142.3 MiB	0.1 MiB	246012	child_hash = hash(tuple(child.state))
109				#kiểm tra xem nút này đã duyệt hay chưa, nếu chưa thì thêm vào stack
110	142.3 MiB	0.6 MiB	246012	if child_hash not in visited:
111	142.3 MiB	0.1 MiB	213133	stack.append(child)
112				
113				#trả về không tìm thấy lời giải và số nút đã duyệt
114				return None, len(visited)

Hình 5: Chi tiết sử dụng bộ nhớ của testcase 8 puzzle với giải thuật DFS

Do output trên có số lượng bước di chuyển quá lớn (114520 bước) nên báo cáo chỉ trình bày số bước hành động, số trạng thái được duyệt qua và thời gian thực thi của giải thuật. Với kích thước của 8-puzzle, DFS đã dành ra khoảng 0.7 giây để tìm ra đáp án với bộ nhớ sử dụng là $142.3 - 47.7 = 94.6 \text{ (MiB)} = 94.6 * 2^{20} = 99,195,289.6 \text{ (bytes)}$



1.4.2.b Đối với bài toán 15-puzzle

Test case:

1
4
1 0 7 4
6 3 10 2
5 11 14 8
9 13 15 12

Name	Status	62% CPU	95% Memory	97% Disk	0% Network
Visual Studio Code (20)		20.3%	2,406.5 MB	125.3 MB...	0 Mbps
Python		20.3%	2,357.9 MB	124.7 MB...	0 Mbps

Hình 6: Chi tiết sử dụng bộ nhớ của testcase 15 puzzle với giải thuật DFS

Đối với test case 15-puzzle, máy tính cá nhân của sinh viên không đủ khả năng để hỗ trợ giải thuật DFS tìm ra đáp án. Nguyên nhân là do số trạng thái của bài toán 15-puzzle nhiều hơn gấp $16!/9! = 57657600$ lần so với bài toán 8-puzzle. Sau hơn 1 giờ thực thi, chương trình vẫn chưa thể tìm ra được kết quả của bài toán trên. Tuy nhiên thông qua Task manager, có thể tương đối biết được rằng bộ nhớ tiêu tốn trong trường hợp này ít nhất là $2,357.9 \text{ (MB)} = 2,357.9 * 10^6 = 2,357,900,000 \text{ (bytes)}$, gấp 23.77 lần so với test case 8-puzzle.

1.4.3 Nhận xét

Có thể thấy, giải thuật DFS chỉ có thể giải được các bài toán N-puzzle với $N = 8$, với các bài toán có kích thước lớn hơn thì số trạng thái sẽ tăng nhanh chóng theo cấp số nhân nên việc giải quyết bằng giải thuật DFS trên máy tính cá nhân là không khả thi. Hơn nữa, do giải thuật DFS hoạt động theo cơ chế duyệt theo chiều sâu, nghĩa là khi không có giới hạn về độ sâu thì nó sẽ duyệt mãi theo một nhánh cho tới khi đạt đến trạng thái mục tiêu. Việc này dẫn đến lời giải của giải thuật DFS cho bài toán N-puzzle chứa số lượng bước di chuyển là rất lớn (lên đến hàng trăm ngàn bước) và không khả thi khi áp dụng tìm lời giải cho các bài toán ngoài thực tế.

Vì vậy, ở phần tiếp theo nhóm em sẽ xem xét hiện thực giải bài toán N-puzzle bằng giải thuật cải tiến của DFS đó là IDS. Giải thuật này giúp giới hạn độ sâu tìm kiếm để lời giải tìm được chứa số bước di chuyển ít hơn rất nhiều so với DFS và có thể chấp nhận được. Hơn nữa IDS sẽ lặp lại giải thuật với nhiều độ sâu giới hạn khác nhau cho đến khi tìm thấy lời giải ở một độ sâu thích hợp nên có thể tránh được việc lặp vô hạn trong những bài toán có số trạng thái khổng lồ như 15-puzzle.

1.5 Giải thuật IDS

1.5.1 Hiện thực giải thuật IDS

Giải thuật được hiện thực thông qua hàm IDS có cơ chế hoạt động như sau:

- Thực thi giải thuật DLS (DFS có giới hạn độ sâu) với độ sâu tăng dần từ 1 tới độ sâu tối đa.
- Nếu tìm thấy lời giải, trả về lời giải và kết thúc thuật toán. Nếu không, tiếp tục thực hiện DLS với độ sâu tăng lên 1.



```
def IDS(given_state, size, max_depth):  
    # Tầng do sau tìm kiếm từ 1 -> max_depth  
    for depth in range(1, max_depth + 1):  
        solution, nodes_visited = DLS(given_state, size, depth)  
        # Nếu tại độ sâu depth có lời giải bài toán, trả về lời giải đó và kết thúc hàm  
        # Tra về lời giải và kết thúc giải thuật  
        if solution is not None:  
            return solution, nodes_visited  
    return None, nodes_visited
```

1.5.2 Đo đặc thời gian bộ nhớ tiêu tốn

1.5.2.a Đối với bài toán 8-puzzle

Test case:

1
3
1 8 3
4 5 7
6 2 0

Output:

Action: ['Up', 'Left', 'Down', 'Right', 'Up', 'Up', 'Left', 'Down', 'Down', 'Left', 'Up', 'Up', 'Right', 'Right', 'Down', 'Down', 'Left', 'Up', 'Up', 'Left', 'Down', 'Down', 'Right', 'Up', 'Right', 'Down']

Number of actions: 26

Number of explored nodes is: 20538

Time: 0.2160 second

Line #	Mem usage	Increment	Occurrences	Line Contents
206	47.6 MiB	47.6 MiB	1	@profile
207				def IDS(given_state, size, max_depth):
208				# Tầng độ sâu tìm kiếm từ 1 -> max_depth
209	49.2 MiB	0.0 MiB	26	for depth in range(1, max_depth + 1):
210	49.2 MiB	1.5 MiB	26	solution, nodes_visited = DLS(given_state, size, depth)
211				# Nếu tại độ sâu depth có lời giải bài toán, trả về lời giải đó và kết thúc hàm
212	49.2 MiB	0.0 MiB	26	if solution is not None:
213	49.2 MiB	0.0 MiB	1	return solution, nodes_visited
214				return None, nodes_visited

Hình 7: Chi tiết sử dụng bộ nhớ của testcase 8 puzzle với giải thuật IDS

Có thể thấy, giải thuật IDS hiệu quả hơn hẳn khi số bước di chuyển cần thiết để giải bài toán được giảm chỉ còn 26 bước, giảm đi 4404.6 lần so với giải thuật DFS. Với kích thước của 8-puzzle, IDS đã dành ra khoảng 0.2 giây để tìm ra đáp án với bộ nhớ sử dụng là $49.2 - 47.6 = 1.6$ (MiB) $= 1.6 * 2^{20} = 1,677,721.6$ (bytes)

1.5.2.b Đối với bài toán 15-puzzle (đơn giản)

Test case:

1
4
1 0 7 4
6 3 10 2
5 11 14 8
9 13 15 12



Output:

Action: ['Down', 'Right', 'Right', 'Down', 'Down', 'Left', 'Up', 'Left', 'Up', 'Right', 'Up', 'Left', 'Down', 'Left', 'Down', 'Down', 'Right', 'Right', 'Right']

Number of actions: 19

Number of explored nodes is: 489145

Time: 5.3269 second

Line #	Mem usage	Increment	Occurrences	Line Contents
206	47.2 MiB	47.2 MiB	1	@profile
207				def IDS(given_state, size, max_depth):
208				# Tăng độ sâu tìm kiếm từ 1 -> max_depth
209	50.3 MiB	-3.5 MiB	25	for depth in range(1, max_depth + 1):
210	50.3 MiB	-1.3 MiB	25	solution, nodes_visited = DLS(given_state, size, depth)
211				# Nếu tại độ sâu depth có lời giải bài toán, trả về lời giải đó và kết thúc hàm
212	50.3 MiB	-4.3 MiB	25	if solution is not None:
213	49.5 MiB	-0.8 MiB	1	return solution, nodes_visited
214				return None, nodes_visited

Hình 8: Chi tiết sử dụng bộ nhớ của testcase 15 puzzle với giải thuật IDS

Có thể thấy, giải thuật IDS hiệu quả khi có thể giải được bài toán 15-puzzle trên với số bước di chuyển là 19 bước trong khi giải thuật DFS không thể tìm ra được lời giải. Với kích thước của 15-puzzle, IDS đã dành ra khoảng 5.5 giây để tìm ra đáp án với bộ nhớ sử dụng là $50.3 - 47.2 = 3.1$ (MiB) $= 3.1 * 2^{20} = 3,250,585.6$ (bytes)

1.5.2.c Đối với bài toán 15-puzzle (phức tạp)

Test case

1
4
15 6 8 14
10 9 4 2
11 1 13 7
0 3 12 5

Name	Status	60% CPU	85% Memory	12% Disk	0% Network
Visual Studio Code (20)		39.8%	1,683.7 MB	0.1 MB/s	0 Mbps
Python		39.8%	1,397.8 MB	0 MB/s	0 Mbps

Hình 9: Chi tiết sử dụng bộ nhớ của testcase 15 puzzle (phức tạp) với giải thuật IDS

Đối với test case 15-puzzle phức tạp, máy tính cá nhân của sinh viên không đủ khả năng để hỗ trợ giải thuật IDS tìm ra đáp án. Sau hơn 1 giờ thực thi, chương trình vẫn chưa thể tìm ra được kết quả của bài toán trên. Tuy nhiên thông qua Task manager, có thể tương đối biết được rằng bộ nhớ tiêu tốn trong trường hợp này ít nhất là $1,397.8$ (MB) $= 1,397.8 * 10^6 = 1,397,800,000$ (bytes), gấp 430 lần so với test case 15-puzzle cơ bản.

1.5.3 Nhận xét

Có thể thấy, giải thuật IDS đã có thể giải được các bài toán N-puzzle với $N = 8$, với các bài toán có kích thước lớn hơn như 15-puzzle thì IDS đã có thể giải được những bài toán mức độ dễ với số bước di chuyển nhỏ. Hơn nữa, do giải thuật IDS hoạt động theo cơ chế tăng dần độ sâu giới hạn, lời giải mà nó tìm ra được chứa số bước ít hơn rất nhiều lần so với giải thuật DFS nên cho kết quả khả thi hơn khi ứng dụng vào thực tế. Nhìn chung, IDS cho hiệu suất tốt hơn so với DFS tuy nhiên vẫn chưa đủ tốt nếu cần giải các bài toán N-Puzzle phức tạp.



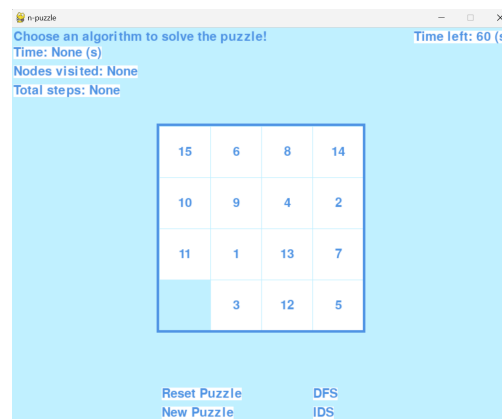
1.6 So sánh kết quả 2 giải thuật

Testcase	DFS			IDS		
	Time (s)	Memory (byte)	Num moves	Time (s)	Memory (byte)	Num moves
8-puzzle	0.7197	$99.19 * 10^6$	114520	0.216	$1.67 * 10^6$	26
15-puzzle	-	$2.35 * 10^9$	-	5.3269	$3.25 * 10^6$	19
15-puzzle (phức tạp)	-	-	-	-	$1.39 * 10^9$	-

Từ bảng so sánh trên, có thể thấy giải thuật IDS hiệu quả hơn so với giải thuật DFS về mọi mặt. Tuy nhiên do giải thuật IDS vẫn là giải thuật trên tìm kiếm mù theo chiều sâu nên khi không gian trạng thái bài toán càng lớn thì nó cũng tỏ ra không hiệu quả và không thể giải được trong thời gian giới hạn. Vì vậy, để giải được các bài toán N-Puzzle phức tạp hơn (ví dụ như 15-Puzzle, 24-Puzzle) nhóm đề xuất người dùng nên thử thực hiện các giải thuật tìm kiếm heuristic điển hình như A*.

1.7 Giao diện trò chơi

Ta có thể thực thi file UI.py để hiển thị ra giao diện trò chơi, trạng thái ban đầu của trò chơi sẽ được đọc từ file input.txt có định dạng như đã trình bày ở trên. Người dùng có thể chọn New Puzzle và nhập vào kích thước k từ Terminal để trò chơi sinh ra một trạng thái ban đầu mới thỏa mãn kích thước người dùng đã nhập và có thể giải được từ hàm sinh đã được hiện thực bên trên.



Hình 10: Giao diện trò chơi

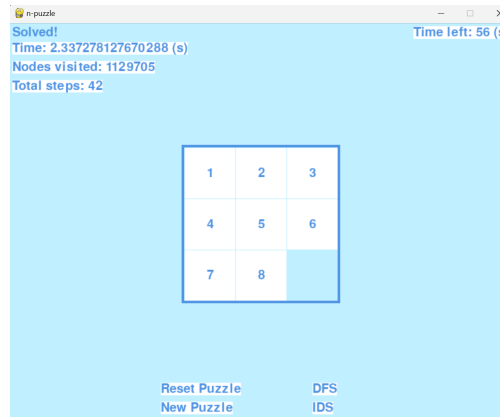
Trên giao diện, người dùng có 4 lựa chọn là:

- New Puzzle: Tạo game mới dựa trên kích thước k mà người dùng nhập vào.
- Reset Puzzle: Reset bài toán hiện tại về trạng thái ban đầu.
- DFS: Giải bài toán bằng giải thuật DFS.
- IDS: Giải bài toán bằng giải thuật IDS

Sau khi bài toán được giải xong, trò chơi sẽ tự động di chuyển các ô theo lời giải bài toán và hoàn thành sắp xếp theo thứ tự. Đồng thời, bên góc trái trên cùng cũng sẽ hiển thị các thông



tin liên quan đến lời giải như thời gian thực thi, số trạng thái đã duyệt, số bước di chuyển đã thực hiện.



Hình 11: Giao diện sau khi giải xong một bài toán

2 Bài toán Sudoku

2.1 Giới thiệu bài toán

Sudoku là 1 dạng trò chơi giải đố, với mục tiêu là điền hết tất cả các ô trong lưới 9×9 sao cho mỗi hàng, mỗi cột, mỗi khối con 3×3 đều chứa tất cả các số nguyên từ 1 đến 9 và mỗi số chỉ xuất hiện đúng một lần. Với mỗi bài toán hợp lệ thì chỉ có duy nhất một lời giải được tìm ra. Độ khó của trò chơi phụ thuộc vào số lượng các số đã được cho trước.

		4				7	2	1
3				1				
	7				8		5	
			4		1	6		5
9		8		2		3		4
5		1	3		9			
	1		5				8	
				7				6
7	2	5				1		

Hình 12: Ví dụ về đề bài của một bài toán Sudoku mức độ dễ

Bài toán Sudoku được liệt kê vào loại **NP-complete**, có nghĩa là không có một giải thuật cụ thể nào có thể đảm bảo việc giải một sudoku bất kì trong một khoảng thời gian ước lượng được. Đối với Sudoku khi ở trạng thái bắt đầu, chúng ta có thể sinh ra được khoảng 6.67×10^{21} nước đi mà chỉ tồn tại một nước đi hợp lệ trong đó. Về số lượng các ô cho sẵn, nếu có ít hơn hoặc bằng 16 ô cho trước thì sẽ không thể đảm bảo được việc chỉ tồn tại duy nhất một lời giải.



2.2 Xác định các yếu tố cho bài toán tìm kiếm

2.2.1 Không gian trạng thái

Không gian trạng thái của bài toán Sudoku bao gồm tất cả các bảng 9x9, với mỗi ô trong bảng có thể chứa một số từ 1 đến 9.

Kích thước của không gian trạng thái này là rất lớn, vì có 9^{81} trạng thái có thể có. Tuy nhiên, giải thuật di truyền sẽ chỉ làm việc với một tập hợp con nhỏ của không gian trạng thái này ở mỗi thế hệ, được gọi là quần thể (population)

2.2.2 Định nghĩa trạng thái bài toán

Một trạng thái của bài toán Sudoku được biểu diễn bằng một bảng 9x9, trong đó mỗi ô chứa một số từ 1 đến 9. Một bảng đầy đủ sẽ có 81 ô đã được điền đầy đủ, và có thể hợp lệ (đúng theo quy tắc Sudoku) hoặc không hợp lệ.

Trong bài toán này, một trạng thái được biểu diễn bằng một ma trận 9x9.

2.2.3 Trạng thái khởi đầu và trạng thái mục tiêu

Trạng thái khởi đầu của bài toán là bảng Sudoku đã cho trong đề bài, trong đó một số ô đã được điền giá trị và các ô trống được đánh dấu bằng số 0. Các ô đã điền từ trước được xem là "cố định" và không được phép thay đổi trong suốt quá trình tiến hóa.

Trạng thái mục tiêu là một bảng Sudoku đầy đủ và hợp lệ, trong đó mỗi hàng, cột và khối 3x3 đều chứa các số từ 1 đến 9 mà không có số trùng lặp theo quy luật của Sudoku.

2.2.4 Luật di chuyển

Luật di chuyển trong bài toán Sudoku là cách điền một số từ 1 đến 9 vào một ô trống trong bảng theo luật của Sudoku sao cho mỗi hàng, cột và khối 3x3 đều chứa các số từ 1 đến 9 mà không có số trùng lặp.

2.2.5 Hàm lượng giá trạng thái

Hàm lượng giá đánh giá mức độ phù hợp của mỗi cá thể bằng cách đo lường mức độ gần đạt đến một trạng thái hợp lệ của Sudoku. Hàm lượng giá được tính bằng cách đếm số lượng các số duy nhất từ 1 đến 9 trong mỗi hàng, cột và khối 3x3:

- Tổng điểm của tất cả các hàng, cột và khối sẽ được chuẩn hóa trong khoảng từ 0 đến 1.
- Với mỗi hàng, cột và khối 3x3, nếu có đủ các số từ 1 đến 9 mà không bị trùng lặp, cá thể sẽ đạt điểm tối đa cho phần đó. Công thức tính điểm cho mỗi hàng/cột/khối 3x3 như sau:

$$score = \frac{1.0/N}{9}$$

Với N là số lượng các ký tự số duy nhất trong hàng/cột/khối tương ứng. Khi này mỗi phần sẽ có điểm tối đa là 1/9 và tổng điểm tối đa tất cả hàng, cột và khối có thể đạt được là 1.0

Điểm lượng giá (fitness) của một cá thể sẽ bằng tích tổng điểm của các hàng, cột và khối 3x3 trong bảng Sudoku của cá thể đó. Một cá thể có điểm lượng giá đạt giá trị 1.0 nghĩa là bảng Sudoku của cá thể đó hoàn toàn hợp lệ, không có số nào bị lặp lại trong các hàng, cột hoặc khối 3x3.



2.2.6 Hàm sinh trạng thái khởi đầu

Để đảm bảo bài toán Sudoku được đưa ra là có thể giải được (có tồn tại lời giải), ta phải hiện thực hàm sinh trạng thái khởi đầu cho bài toán này. Hàm sinh được hiện thực dựa trên ý tưởng như sau bằng kỹ thuật heuristic kết hợp với việc giảm dần số lượng lựa chọn của các ô để tạo bảng Sudoku hoàn chỉnh và sử dụng thuật toán backtracking để tạo ra bảng Sudoku khiếm khuyết (câu đố sudoku):

1. Khởi tạo danh sách các ô trong bảng Sudoku và tạo một bảng Sudoku trống.
2. Duyệt qua từng ô và chọn một ô có ít giá trị khả thi nhất, dựa trên số lượng giá trị còn lại có thể điền vào ô đó.
3. Lựa chọn ngẫu nhiên một giá trị hợp lệ và điền vào ô đó. Sau khi điền, loại bỏ giá trị này khỏi các ô liên quan (cùng hàng, cột hoặc ô 3x3).
4. Tiếp tục cho đến khi tất cả các ô trong bảng đều được điền giá trị hợp lệ.
5. Sau đó tiến hành xóa đi các ô trống dựa trên cấp độ, đồng thời xác định bài toán chỉ có duy nhất một lời giải.

Để đảm bảo bảng Sudoku sinh ra là hợp lệ, quá trình kiểm tra và điền số vào bảng phải tuân theo các nguyên tắc sau:

- Mỗi số trong một hàng, cột và khối 3x3 phải là duy nhất, không được lặp lại.
- Đảm bảo rằng việc lựa chọn ô có ít lựa chọn nhất sẽ giảm thiểu khả năng gặp phải các tình huống không thể điền được giá trị sau này.
- Đảm bảo chỉ có duy nhất một lời giải cho bài toán Sudoku.

Hàm sinh trạng thái ban đầu cho bài toán Sudoku và các hàm hỗ trợ của nó đã được hiện thực bằng ngôn ngữ lập trình Python tại file **sudoku_gen.py**. Hàm hỗ trợ sinh ra đề bài Sudoku theo độ khó được hiện thực như sau:

```
def generate_sudoku(sudoku, difficulty):  
    if difficulty == "EASY":  
        num_holes = random.randint(30, 35)  
    elif difficulty == "MEDIUM":  
        num_holes = random.randint(36, 45)  
    elif difficulty == "HARD":  
        num_holes = random.randint(46, 55)  
    else:  
        raise ValueError("Invalid difficulty level")  
  
    remove_cells(sudoku, num_holes, difficulty)  
    return sudoku
```

Đề bài Sudoku được sinh ra sẽ được phân độ khó theo ba cấp độ:

- Dễ (Easy): Bảng Sudoku chứa 30-35 ô trống.
- Trung bình (Medium): Bảng Sudoku chứa 36-45 ô trống.
- Khó (Hard): Bảng Sudoku chứa 46-55 ô trống.



2.3 Giải bài toán Sudoku bằng giải thuật di truyền

2.3.1 Giải thuật di truyền

Giải thuật di truyền (Genetic Algorithm - GA) là một kỹ thuật tối ưu hóa được lấy cảm hứng từ quá trình tiến hóa trong tự nhiên. Nó áp dụng các nguyên tắc tiến hóa như di truyền, chọn lọc tự nhiên, đột biến để tìm kiếm qua không gian giải pháp nhằm tìm ra giải pháp tối ưu. Trình tự hoạt động của giải thuật di truyền như sau:

1. **Khởi tạo quần thể ban đầu:** Bắt đầu giải thuật bằng cách tạo ra một quần thể ban đầu gồm nhiều cá thể. Mỗi cá thể được biểu diễn bằng một chuỗi gen hoặc vectơ, đại diện cho một giải pháp tiềm năng.
2. **Đánh giá sự thích nghi (fitness):** Đánh giá mức độ thích nghi của mỗi cá thể trong quần thể bằng cách tính toán giá trị thích nghi của nó thông qua một hàm lượng giá (fitness function). Hàm fitness này đánh giá mức độ tốt của giải pháp được đại diện bởi cá thể.
3. **Lựa chọn (Selection):** Chọn lọc các cá thể tốt nhất để tiếp tục vào thế hệ tiếp theo. Các cá thể có điểm fitness cao hơn có khả năng được chọn cao hơn, nhưng các cá thể khác cũng có cơ hội được chọn.
4. **Lai ghép (Crossover):** Tạo ra các cá thể mới bằng cách kết hợp gen của các cá thể được chọn lọc từ thế hệ hiện tại. Quá trình này tạo ra sự đa dạng gen và kết hợp các đặc điểm tốt từ các cá thể cha mẹ.
5. **Đột biến (Mutation):** Ngẫu nhiên thay đổi một số gen trong các cá thể mới để giữ cho quần thể không bị rơi vào một vùng lặp lại quá sớm và để giữ cho sự đa dạng gen.
6. **Điều kiện dừng:** Thuật toán tiếp tục tạo ra các thế hệ mới bằng cách lặp lại các bước trên cho đến khi một điều kiện dừng được đáp ứng như là số lượng thế hệ tối đa, đạt giá trị fitness mong muốn hoặc không có sự cải thiện qua một số thế hệ liên tiếp. Thuật toán kết thúc bằng cách trả về giải pháp tốt nhất mà nó tìm thấy nếu đạt fitness mong muốn, ngược lại thì trả về kết quả không có lời giải được tìm thấy.

2.3.2 Ứng dụng giải bài toán Sudoku

Để giải bài toán Sudoku bằng giải thuật di truyền, ta sẽ định nghĩa một số lớp (class) hỗ trợ như sau:

- **Candidate:** Lớp biểu diễn một ứng viên trong quần thể.
- **GivenPuzzle:** Lớp con của Candidate, đại diện cho bảng Sudoku đã cho ở đề bài với một số ô được điền trước.
- **Population:** Lớp biểu diễn cho một quần thể gồm nhiều ứng viên.
- **Tournament:** Lớp biểu diễn cho một vòng đấu trong quần thể để lựa chọn ứng viên cho quá trình lai ghép.
- **Crossover:** Lớp thực hiện việc lai ghép giữa các ứng viên để tạo ra ứng viên mới.

Bài toán được giải quyết thông qua lớp Sudoku là lớp chính của chương trình. Lớp này gồm có các thuộc tính như sau:

- **given_puzzle:** Bảng Sudoku ban đầu (đề bài) mà chương trình cần giải.



- **population:** Quần thể tại một thời điểm trong quá trình tiến hóa gồm các ứng viên.
- **best_candidate:** Ứng viên tốt nhất tại một thời điểm trong quá trình tiến hóa.
- **size:** Kích thước của quần thể, biểu diễn số lượng ứng viên của quần thể tại mỗi thế hệ.
- **generations:** Số thế hệ tối đa cho quá trình tiến hóa.
- **mutation_rate:** Xác suất xảy ra đột biến.
- **num_elites:** Số lượng ứng viên ưu tú được giữ lại từ thế hệ này sang thế hệ khác.
- **reseeding_threshold:** Ngưỡng để tái khởi động một phần hoặc toàn bộ quần thể sau một số lượng thế hệ liên tiếp không có sự cải thiện.

Ngoài ra, lớp Sudoku còn chứa các thuộc tính khác dùng để điều chỉnh xác suất đột biến trong quá trình tiến hóa theo quy tắc $\frac{1}{5}$ của Richenberg và có hai phương thức bao gồm:

- **load():** sử dụng để nạp bảng Sudoku đã cho ở đề bài vào chương trình để chạy giải thuật.
- **solve():** triển khai giải thuật di truyền để giải bài toán Sudoku.

Hiện thực của phương thức load() và hàm khởi tạo các thuộc tính của lớp Sudoku trong chương trình bằng ngôn ngữ Python như sau:

```
class Sudoku:
    def __init__(self, size = 1000, generations = 1000, mutation_rate =
        mutation_rate_default):
        self.given_puzzle = None
        self.population = None
        self.best_candidate = None
        self.size = size
        self.generations = generations
        self.mutation_rate = mutation_rate

        self.num_elites = int(0.05*self.size)
        self.reseeding_threshold = 100

        # Dung dieu chinh xac suat dot bien
        # Su dung quy tac 1/5 cua Richenberg
        self.num_mutations = 0 # So dot bien da xay ra
        self.phi = 0 # So dot bien thanh cong
        self.sigma = 1

        self.cycle_crossover = Crossover()
        self.tournament = Tournament()

        # Phuong thuc dung de load bang Sudoku da cho tu de bai
    def load_puzzle(self, puzzle):
        self.given_puzzle = GivenPuzzle(puzzle)
```

Phương thức solve() dùng để triển khai giải thuật di truyền theo các bước của giải thuật đã được trình bày ở trên được hiện thực như sau:



```
def solve(self):
    #Khởi tạo quần thể ban đầu
    self.population = Population()
    self.population.seed(self.size, self.given_puzzle)

    stale = 0 # Biến đếm số lượng thế hệ liên tiếp không có sự cải thiện
    for generation in range(0, self.generations):
        print("Generation ", generation)

        # Trong mỗi thế hệ, chọn ra ứng viên tốt nhất và kiểm tra có phải là lời giải hay chưa
        best_fitness = 0.0
        for candidate in range(0, self.size):
            fitness = self.population.candidates[candidate].fitness
            if fitness == 1:
                print("Solution found at generation ", generation)
                print(self.population.candidates[candidate].puzzle)
                return self.population.candidates[candidate]

            if fitness > best_fitness:
                best_fitness = fitness
                self.best_candidate = self.population.candidates[candidate]

        print("Best fitness: ", best_fitness)
        print("Stale: ", stale)

        # Tạo quần thể kế tiếp
        next_population = []
        # Lựa chọn các thế hệ ưu tú để giữ lại
        self.population.sort()
        elites = []
        for idx in range(0, self.num_elites):
            elite = Candidate()
            elite.puzzle = np.copy(self.population.candidates[idx].puzzle)
            elites.append(elite)

        # Tạo các ứng viên con lai bằng lai ghép
        for _ in range(self.num_elites, self.size, 2):
            parent1 = self.tournament.compete(self.population.candidates)
            parent2 = self.tournament.compete(self.population.candidates)
            child1, child2 = self.cycle_crossover.crossover(parent1, parent2, crossover_rate=0.9)

            # Đốt biến các cá thể con
            old_fitness = child1.fitness
            mutate = child1.mutate(self.mutation_rate, self.given_puzzle)
            child1.update_fitness()
```



```
        if mutate:
            self.num_mutations += 1
            if child1.fitness > old_fitness: # Dung theo doi ty
                le thanh cong cua dot bien
                self.phi += 1

        old_fitness = child2.fitness
        mutate = child2.mutate(self.mutation_rate, self.
            given_puzzle)
        child2.update_fitness()
        if mutate:
            self.num_mutations += 1
            if child2.fitness > old_fitness:
                self.phi += 1

        next_population.append(child1)
        next_population.append(child2)

    for elite in range(0, self.num_elites):
        next_population.append(elites[elite])

    # Chon the he ke tiep
    self.population.candidates = next_population
    self.population.update_fitness()

    # Dieu chinh ty le dot bien
    if self.num_mutations == 0:
        self.phi = 0 # Tranh chia cho 0
    else:
        self.phi = self.phi / self.num_mutations

    if self.phi > 0.2:
        self.sigma = self.sigma / 0.998
    elif self.phi < 0.2:
        self.sigma = self.sigma * 0.998

    self.mutation_rate = abs(np.random.normal(loc=0.0, scale=self
        .sigma, size=None))
    self.num_mutations = 0
    self.phi = 0

    # Kiem tra quan the bi "stale"
    if self.population.candidates[0].fitness != self.population.
        candidates[1].fitness:
        stale = 0
    else:
        stale += 1
    # Neu vuot nguong, quan the duoc tai khoi dong lai
    if stale >= self.reseeding_threshold:
        print("The population has gone stale. Re-seeding ...")
        self.population.seed(self.size, self.given_puzzle)
        stale = 0
        self.sigma = 1
```



```
self.phi = 0
self.num_mutations = 0
self.mutation_rate = mutation_rate_default

print("No solution found.")
return None
```

Phương thức solve() của lớp Sudoku hoạt động như sau:

1. Đầu tiên, giải thuật khởi tạo quần thể ban đầu bằng cách sử dụng phương thức seed() của lớp Population.
2. Tiếp đến, bắt đầu vòng lặp qua từng thế hệ với số thế hệ tối đa được giới hạn bởi thuộc tính generations của class Sudoku.
3. Trong mỗi thế hệ, lựa chọn ra ứng viên tốt nhất (dựa trên điểm fitness) và kiểm tra xem đó có phải là lời giải của bài toán hay không. Nếu phải, trả về lời giải bài toán và kết thúc chương trình.
4. Nếu không phải là giải pháp, tiến hành tạo ra thế hệ kế tiếp bằng cách lựa chọn, lai ghép và đột biến. Quá trình này được lặp lại cho đến khi tìm được lời giải bài toán hoặc đã vượt quá số thế hệ cho phép.
5. Trong quá trình lặp, tỷ lệ đột biến được điều chỉnh dựa trên quy tắc 1/5 của Rechenberg.
6. Nếu quần thể trở nên "stale" (nghĩa là không có sự thay đổi, cải thiện trong quần thể sau một số thế hệ liên tiếp), giải thuật sẽ tạo ra một quần thể mới bằng cách thực hiện lại quá trình khởi tạo quần thể ban đầu.
7. Cuối cùng, nếu không có lời giải nào được tìm thấy sau khi đã thực hiện một số lượng thế hệ nhất định, phương thức sẽ trả về không có lời giải nào được tìm thấy.

2.4 Kết quả thực thi và đánh giá hiệu năng giải thuật

Để kiểm thử chương trình, ta sẽ tiến hành chạy giải thuật di truyền trên đề bài Sudoku được sinh ra từ hàm sinh và các hàm hỗ trợ trong file **sudoku_gen.py**. Việc kiểm thử sẽ được thực hiện với độ khó đề bài tăng dần từ dễ, trung bình đến khó.

2.4.1 Mức độ dễ

Testcase:

```
0 0 1 9 3 4 0 2 6
4 2 6 0 8 5 1 9 3
3 8 9 0 0 0 0 0 4
9 1 7 0 4 0 5 0 8
0 0 0 0 1 9 6 0 2
0 6 3 8 5 0 0 4 0
0 4 0 0 0 0 0 0 0
6 3 5 2 9 0 4 0 7
0 9 0 4 7 8 3 6 0
```

Output:

Sudoku solution:
5 7 1 9 3 4 8 2 6



4 2 6 7 8 5 1 9 3
3 8 9 1 2 6 7 5 4
9 1 7 6 4 2 5 3 8
8 5 4 3 1 9 6 7 2
2 6 3 8 5 7 9 4 1
7 4 8 5 6 3 2 1 9
6 3 5 2 9 1 4 8 7
1 9 2 4 7 8 3 6 5

Execution time: 29.0696 seconds

Line #	Mem usage	Increment	Occurrences	Line Contents
309	58.8 MiB	58.8 MiB	1	@profile
310				def solve(self):
311				# Khởi tạo quần thể ban đầu
312	58.8 MiB	0.0 MiB	1	self.population = Population()
313	59.4 MiB	0.6 MiB	1	self.population.seed(self.size, self.given_puzzle)
314				
315				# Lặp qua các thế hệ
316	59.4 MiB	0.0 MiB	1	stale = 0 # Biến đếm số thế hệ liên tiếp mà quần thể không cải thiện về fitness
317	59.4 MiB	0.0 MiB	1	for generation in range(0, self.generations):
318	59.4 MiB	0.0 MiB	1	print("Generation ", generation)
319				
320				# Trong mỗi thế hệ, chọn ra ứng viên tốt nhất và kiểm tra đã tìm thấy lời giải hay chưa
321	59.4 MiB	0.0 MiB	1	best_fitness = 0.0
322	59.4 MiB	0.0 MiB	24	for candidate in range(0, self.size):
323	59.4 MiB	0.0 MiB	24	fitness = self.population.candidates[candidate].fitness
324	59.4 MiB	0.0 MiB	24	if fitness == 1:
325	59.4 MiB	0.0 MiB	1	print("Solution found at generation ", generation)
326	59.4 MiB	0.0 MiB	1	print(self.population.candidates[candidate].puzzle)
327	59.4 MiB	0.0 MiB	1	return self.population.candidates[candidate]
328				
329				# Tìm ứng viên tốt nhất
330	59.4 MiB	0.0 MiB	23	if fitness > best_fitness:
331	59.4 MiB	0.0 MiB	3	best_fitness = fitness
332	59.4 MiB	0.0 MiB	3	self.best_candidate = self.population.candidates[candidate]

Hình 13: Chi tiết sử dụng bộ nhớ của bài toán Sudoku mức độ dễ

Có thể thấy, giải thuật di truyền đã tốn khoảng 29 giây để tìm ra lời giải cho đề bài Sudoku mức độ dễ với bộ nhớ sử dụng là $59.4 - 58.8 = 0.6$ (MiB) $= 0.6 * 2^{20} = 629,145.6$ (bytes)

2.4.2 Mức độ trung bình

Testcase:

3 8 9 2 7 0 6 0 4
7 0 6 9 1 0 2 8 5
0 0 0 8 4 0 0 3 0
6 0 4 3 0 2 0 0 0
0 1 8 0 0 7 0 0 2
0 0 0 6 0 0 0 5 0
0 0 0 7 0 0 1 0 6
1 0 0 0 0 8 0 0 0
0 6 3 0 0 0 5 7 0

Output:

Sudoku solution:

3 8 9 2 7 5 6 1 4
7 4 6 9 1 3 2 8 5
2 5 1 8 4 6 7 3 9
6 7 4 3 5 2 8 9 1
5 1 8 4 9 7 3 6 2
9 3 2 6 8 1 4 5 7
8 9 5 7 3 4 1 2 6
1 2 7 5 6 8 9 4 3
4 6 3 1 2 9 5 7 8



Execution time: 78.0562 seconds

Line #	Mem usage	Increment	Occurrences	Line Contents
309	58.5 MiB	58.5 MiB	1	@profile
310				def solve(self):
311				# Khởi tạo quần thể ban đầu
312	58.5 MiB	0.0 MiB	1	self.population = Population()
313	59.1 MiB	0.6 MiB	1	self.population.seed(self.size, self.given_puzzle)
314				
315				# Lập qua các thế hệ
316	59.1 MiB	0.0 MiB	1	stale = 0 # Biến đếm số thế hệ liên tiếp mà quần thể không cải thiện về fitness
317	60.4 MiB	0.0 MiB	11	for generation in range(0, self.generations):
318	60.4 MiB	0.0 MiB	11	print("Generation ", generation)
319				
320				# Trong mỗi thế hệ, chọn ra ứng viên tốt nhất và kiểm tra đã tìm thấy lời giải hay chưa
321	60.4 MiB	0.0 MiB	11	best_fitness = 0.0
322	60.4 MiB	0.0 MiB	10173	for candidate in range(0, self.size):
323	60.4 MiB	0.0 MiB	10163	fitness = self.population.candidates[candidate].fitness
324	60.4 MiB	0.0 MiB	10163	if fitness == 1:
325	60.4 MiB	0.0 MiB	1	print("Solution found at generation ", generation)
326	60.5 MiB	0.0 MiB	1	print(self.population.candidates[candidate].puzzle)
327	60.5 MiB	0.0 MiB	1	return self.population.candidates[candidate]
328				
329				# Tìm ứng viên tốt nhất
330	60.4 MiB	0.0 MiB	10162	if fitness > best_fitness:
331	60.4 MiB	0.0 MiB	75	best_fitness = fitness
332	60.4 MiB	0.0 MiB	75	self.best_candidate = self.population.candidates[candidate]

Hình 14: Chi tiết sử dụng bộ nhớ của bài toán Sudoku mức độ trung bình

Có thể thấy, giải thuật di truyền đã tốn khoảng 78 giây để tìm ra lời giải cho đề bài Sudoku mức độ trung bình với bộ nhớ sử dụng là $60.4 - 59.1 = 1.3$ (MiB) $= 1.3 * 2^{20} = 1,363,148.8$ (bytes)

2.4.3 Mức độ khó

Testcase:

```
0 6 8 0 0 3 4 0 0
0 0 2 0 8 7 9 0 0
0 0 0 0 5 0 6 8 3
6 0 0 4 0 9 1 3 0
0 0 4 0 0 0 5 0 0
0 1 9 0 0 0 0 0 0
0 7 0 0 9 0 0 5 6
0 0 0 0 2 0 0 4 0
9 0 6 0 4 0 0 0 0
```

Output:

Sudoku solution:

```
5 6 8 9 1 3 4 2 7
3 4 2 6 8 7 9 1 5
1 9 7 2 5 4 6 8 3
6 8 5 4 7 9 1 3 2
7 3 4 1 6 2 5 9 8
2 1 9 5 3 8 7 6 4
4 7 3 8 9 1 2 5 6
8 5 1 7 2 6 3 4 9
9 2 6 3 4 5 8 7 1
```

Execution time: 190.6612 seconds

Có thể thấy, giải thuật di truyền đã tốn khoảng 191 giây để tìm ra lời giải cho đề bài Sudoku mức độ trung bình với bộ nhớ sử dụng là $60.6 - 59.2 = 1.4$ (MiB) $= 1.4 * 2^{20} = 1,468,006.4$ (bytes)



Line #	Mem usage	Increment	Occurrences	Line Contents
309	58.4 MiB	58.4 MiB	1	@profile
310				def solve(self):
311				# Khởi tạo quần thể ban đầu
312	58.4 MiB	0.0 MiB	1	self.population = Population()
313	59.2 MiB	0.7 MiB	1	self.population.seed(self.size, self.given_puzzle)
314				
315				# Lặp qua các thế hệ
316	59.2 MiB	0.0 MiB	1	stale = 0 # Biến đếm số thế hệ liên tiếp mà quần thể không cải thiện về fitness
317	60.6 MiB	-0.1 MiB	63	for generation in range(0, self.generations):
318	60.6 MiB	-0.1 MiB	63	print("Generation ", generation)
319				
320				# Trong mỗi thế hệ, chọn ra ứng viên tốt nhất và kiểm tra đã tìm thấy lời giải hay chưa
321	60.6 MiB	-0.1 MiB	63	best_fitness = 0.0
322	60.6 MiB	-74.3 MiB	62864	for candidate in range(0, self.size):
323	60.6 MiB	-74.3 MiB	62802	fitness = self.population.candidates[candidate].fitness
324	60.6 MiB	-74.3 MiB	62802	if fitness == 1:
325	60.6 MiB	-0.0 MiB	1	print("Solution found at generation ", generation)
326	60.6 MiB	0.0 MiB	1	print(self.population.candidates[candidate].puzzle)
327	60.6 MiB	0.0 MiB	1	return self.population.candidates[candidate]
328				
329				# Tìm ứng viên tốt nhất
330	60.6 MiB	-74.2 MiB	62801	if fitness > best_fitness:
331	60.6 MiB	-0.4 MiB	307	best_fitness = fitness
332	60.6 MiB	-0.4 MiB	307	self.best_candidate = self.population.candidates[candidate]

Hình 15: Chi tiết sử dụng bộ nhớ của bài toán Sudoku mức độ khó

2.4.4 Thống kê kết quả và nhận xét

	Giải thuật di truyền		
Level	Time (second)	Memory (MiB)	Result
Easy	29.0696	0.6	Success
Medium	78.0562	1.3	Success
Hard	190.6612	1.4	Success

Từ bảng thống kê kết quả trên có thể thấy giải thuật di truyền đã giải quyết thành công bài toán Sudoku ở cả ba cấp độ khó (Easy, Medium, Hard). Thời gian thực thi tăng đáng kể theo độ khó, từ 29 giây (Easy) đến gần 191 giây (Hard), phản ánh sự gia tăng độ phức tạp của đề bài. Tuy nhiên, mức sử dụng bộ nhớ rất thấp (chỉ từ 0.6 MiB đến 1.4 MiB), cho thấy giải thuật hiệu quả về mặt tài nguyên. Nhìn chung, thuật toán hoạt động khá tốt nhưng cần cải thiện thêm về mặt thời gian để giải quyết các bài toán khó hơn.

2.5 Giao diện trò chơi

Ta có thể thực thi file UI.py để hiển thị ra giao diện trò chơi. Các bước tương tác với giao diện trò chơi Sudoku như sau:

1. Lựa chọn độ khó bằng cách sử dụng các phím mũi tên lên xuống trên bàn phím. Giao diện hiển thị có ba mức độ khó khác nhau cho phép người sử dụng lựa chọn gồm Easy, Medium và Hard. Để lựa chọn một mức độ, người dùng cần sử dụng phím mũi tên trên bàn phím để chọn vào độ khó tương ứng vào bấm phím **Enter**.
2. Sau khi lựa chọn độ khó, trò chơi sẽ sinh ra đề bài tự động theo độ khó người dùng lựa chọn và hiển thị lên giao diện. Để giải đề bài Sudoku bằng giải thuật di truyền, người dùng nhấn vào phím **Space** trên bàn phím.
3. Sau khi giải thuật di truyền được thực hiện xong và có lời giải, giao diện trò chơi sẽ được tự động điền vào các số còn thiếu trên bảng Sudoku dựa vào lời giải và hiển thị thời gian mà giải thuật đã dùng để tìm ra lời giải.



Sudoku

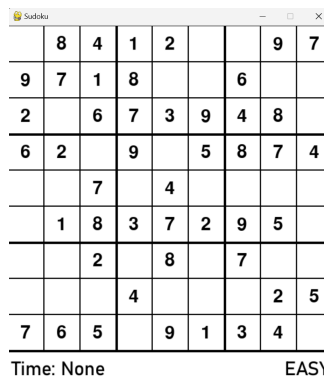
Sudoku

EASY

MEDIUM

HARD

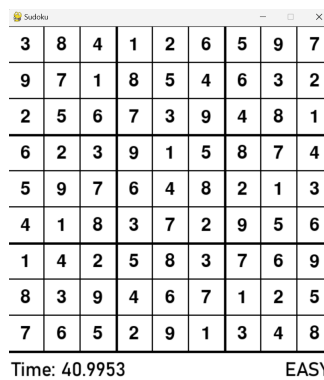
Hình 16: Giao diện lựa chọn mức độ khó của game Sudoku



	8	4	1	2			9	7
9	7	1	8			6		
2		6	7	3	9	4	8	
6	2		9		5	8	7	4
		7		4				
	1	8	3	7	2	9	5	
		2		8		7		
			4				2	5
7	6	5		9	1	3	4	

Time: None EASY

Hình 17: Giao diện đề bài Sudoku được sinh ra sau khi chọn độ khó



3	8	4	1	2	6	5	9	7
9	7	1	8	5	4	6	3	2
2	5	6	7	3	9	4	8	1
6	2	3	9	1	5	8	7	4
5	9	7	6	4	8	2	1	3
4	1	8	3	7	2	9	5	6
1	4	2	5	8	3	7	6	9
8	3	9	4	6	7	1	2	5
7	6	5	2	9	1	3	4	8

Time: 40.9953 EASY

Hình 18: Giao diện sau khi hoàn tất giải bài toán Sudoku

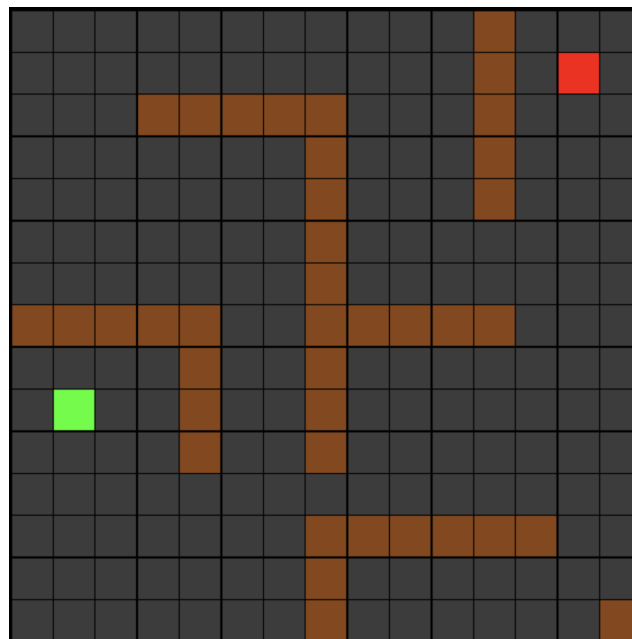


3 Bài toán Path Finding

3.1 Giới thiệu bài toán

Bài toán tìm đường là một trong những vấn đề cơ bản trong lập trình, trí tuệ nhân tạo và khoa học máy tính. Mục tiêu của nó là tìm ra đường đi ngắn nhất hoặc có hiệu quả nhất (chi phí thấp nhất) từ điểm xuất phát đến điểm đích trong một không gian nhất định. Thông thường, không gian này được biểu diễn dưới dạng một lưới ô vuông hoặc đồ thị.

Trong bài tập lớn này, nhóm em sử dụng lưới ô vuông (kích thước $N \times N$) để biểu diễn bài toán tìm đường vì tính trực quan của nó. Mỗi ô trong lưới sẽ đại diện cho một vị trí có thể đi qua hoặc không thể đi qua (vị trí của vật cản). Các ô liền kề theo các hướng nhất định (trên, dưới, trái phải) sẽ biểu diễn các đường đi khả thi giữa các vị trí trong không gian bài toán.



Hình 19: Hình ảnh trực quan của của một bài toán tìm đường

Ở trong hình ảnh này, các ô màu đen là các vị trí có thể đi qua được, các ô màu nâu là các vị trí không thể đi qua (vị trí của vật cản). Vị trí bắt đầu là ô màu xanh và vị trí mục tiêu là ô màu đỏ. Ở mỗi một lượt đi, ta chỉ di chuyển một ô và chi phí di chuyển giữa 2 ô cạnh nhau là 1 đơn vị.

Bài toán cần giải là tìm được đường đi ngắn nhất (hoặc có chi phí thấp nhất) từ vị trí bắt đầu đến vị trí mục tiêu.

3.2 Xác định các yếu tố cho bài toán tìm kiếm

3.2.1 Không gian trạng thái

Bài toán diễn ra trên một lưới ô vuông có kích thước $N \times N$, tức có N hàng và N cột (chỉ số bắt đầu từ 0 đến $N - 1$). Trong không gian này là một tập các ô vuông C được phân thành 2 loại không trùng lên nhau gồm:



- Tập các ô có thể di chuyển được gọi là E .
- Tập các ô không thể di chuyển (vị trí có vật cản) được gọi là W (Wall).

Nhóm sử dụng phương pháp Cellular Automaton để tạo ra bản đồ cho bài toán. Trạng thái bắt đầu và trạng thái kết thúc sẽ được đặt ở rìa hai cạnh đối diện nhau của bản đồ.

```
def generate_maze(rows: int, cols: int, birth: List[int], survival: List[int]):  
    grid = create_grid(rows, cols)  
    start, end = initialize_start_end(grid)  
    initialize_random(grid)  
  
    iterations = 0  
    while iterations < 300:  
        grid = update_grid(grid, birth, survival)  
        iterations += 1  
        if is_solvable(grid, start, end):  
            print(f"Me cung duoc tao sau {iterations} lan lap.")  
            return grid, start, end  
  
    print("Khong the tao me cung kha thi sau 300 lan lap.")  
    return None, None, None
```

Mô tả các hàm trong hàm generate_maze

- `create_grid(rows, cols)`: Tạo lưới mê cung mới với kích thước `rows` x `cols`, các ô mặc định là trống (Empty).
- `initialize_start_end(grid)`: Chọn điểm bắt đầu và kết thúc trong mê cung, đánh dấu chúng là Start và End.
- `initialize_random(grid)`: Tạo tường ngẫu nhiên trong mê cung với xác suất cố định% (Nhóm thiết lập tỉ lệ là 50%).
- `update_grid(grid, birth, survival)`: Cập nhật trạng thái của các ô theo quy tắc Cellular Automaton: ô trống có thể thành tường nếu có đủ số hàng xóm trống (theo quy tắc `birth`), và ô trống sẽ thành tường nếu không có đủ số hàng xóm trống (theo quy tắc `survival`).
- `is_solvable(grid, start, end)`: Kiểm tra xem có lối đi từ điểm bắt đầu đến điểm kết thúc trong mê cung hay không bằng thuật toán tìm kiếm theo chiều sâu.

Các hàm này phối hợp để tạo và tối ưu mê cung sao cho có thể giải quyết được (có lối đi từ bắt đầu đến kết thúc) và có cấu trúc tường hợp lý.

3.2.2 Định nghĩa trạng thái bài toán

Trạng thái hợp lệ của bài toán được biểu diễn bởi vị trí hiện tại trong không gian tìm kiếm, là một tọa độ trong lưới (x, y) thỏa mãn:

- Nằm trong không gian trạng thái, tức là $0 \leq x < N$ và $0 \leq y < N$.
- Không trùng với vị trí của vật cản, tức là $(x, y) \notin W$.



3.2.3 Trạng thái khởi đầu và trạng thái mục tiêu

Trạng thái khởi đầu Là một trạng thái (ngẫu nhiên hoặc được xác định cụ thể) đảm bảo nằm trong không gian trạng thái của bài toán và không trùng với vị trí của vật cản.

Trạng thái khởi đầu được ký hiệu là $(x_{\text{start}}, y_{\text{start}})$, là một vị trí $(x_{\text{start}}, y_{\text{start}})$ trên lưới ô vuông, đảm bảo sao cho:

- $0 \leq x_{\text{start}} < N$,
- $0 \leq y_{\text{start}} < N$,
- $(x_{\text{start}}, y_{\text{start}}) \notin W$

Trong đó N là kích thước của lưới và W là tập các ô có vật cản.

Trạng thái mục tiêu Là một trạng thái (ngẫu nhiên hoặc được xác định cụ thể) đảm bảo nằm trong không gian trạng thái của bài toán và không trùng với vị trí của vật cản và không trùng với trạng thái bắt đầu.

Trạng thái kết thúc được ký hiệu là $(x_{\text{end}}, y_{\text{end}})$, là một vị trí $(x_{\text{end}}, y_{\text{end}})$ trên lưới ô vuông, đảm bảo sao cho:

- $0 \leq x_{\text{end}} < N$,
- $0 \leq y_{\text{end}} < N$,
- $(x_{\text{end}}, y_{\text{end}}) \notin W$,
- $(x_{\text{end}}, y_{\text{end}}) \neq (x_{\text{start}}, y_{\text{start}})$.

3.2.4 Xác định luật di chuyển

Trong không gian trạng thái, nhóm áp dụng hệ trục tọa độ với gốc $(0,0)$ ở góc trên cùng bên trái. Mỗi bước di chuyển là từ một ô đến một ô lân cận chỉ theo một trong các hướng:

- Lên trên: $(x, y) \rightarrow (x, y - 1)$, thỏa mãn $y - 1 \geq 0$ và $(x, y - 1) \notin W$.
- Xuống dưới: $(x, y) \rightarrow (x, y + 1)$, thỏa mãn $y + 1 < N$ và $(x, y + 1) \notin W$.
- Sang phải: $(x, y) \rightarrow (x + 1, y)$, thỏa mãn $x + 1 < N$ và $(x + 1, y) \notin W$.
- Sang trái: $(x, y) \rightarrow (x - 1, y)$, thỏa mãn $x - 1 \geq 0$ và $(x - 1, y) \notin W$.

Nếu không thỏa mãn các điều kiện trên, bước di chuyển đó sẽ không được thực hiện.

Chi phí di chuyển mỗi lượt là 1 đơn vị.

3.2.5 Hàm sinh trạng thái khởi đầu

Hàm này sẽ sinh ra một trạng thái khởi đầu $(x_{\text{start}}, y_{\text{start}})$ (một cách ngẫu nhiên)

```
def get_random_start(grid: list[list[Cell]]) -> tuple[int, int]:
    """Chọn một điểm ngẫu nhiên không phải là CellType.Wall tu luoi.

    Args:
        grid (list[list[Cell]]): Lưới các ô (Cell) với các vật cản và ô
            trong.

    Returns:
```



```
tuple[int, int]: Toa do (x, y) cua diem khong phai la CellType.Wall.

Error:
    Game co the treo neu khong co bat ky vi tri trong nao de chon lam
        trang thai bat dau.
"""
width = len(grid)
height = len(grid[0])

while True:
    # Chon mot toa do ngau nhien (x, y)
    x = random.randint(0, width - 1)
    y = random.randint(0, height - 1)

    # Kiem tra neu o khong phai la CellType.Wall
    if grid[x][y].type != CellType.Wall:
        return (x, y)
```

Hàm trên sẽ sinh một trạng thái bắt đầu ngẫu nhiên hợp lệ với luật của trò chơi. Tuy nhiên, hàm này có nhược điểm là sẽ khiến game bị treo nếu bản đồ không có vị trí trống để chọn làm trạng thái bắt đầu.

3.2.6 Hàm sinh trạng thái mục tiêu

Hàm này sẽ sinh ra một trạng thái mục tiêu ($x_{\text{end}}, y_{\text{end}}$) (một cách ngẫu nhiên)

```
def get_random_end(grid: list[list[Cell]], start: tuple[int, int]) ->
tuple[int, int]:
    """Chon mot diem ngau nhien khong phai la CellType.Wall va khong
        trung voi diem bat dau.

    Args:
        grid (list[list[Cell]]): Luoi cac o (Cell) voi cac vat can va o
            trong.
        start (tuple[int, int]): Toa do cua diem bat dau.

    Returns:
        tuple[int, int]: Toa do (x, y) cua diem ket thuc khong phai la
            CellType.Wall va khong trung voi diem bat dau.

    Error:
        Game co the treo neu khong co bat ky vi tri trong nao de chon lam
            trang thai bat dau.
    """
    width = len(grid)
    height = len(grid[0])

    while True:
        # Chon mot toa do ngau nhien (x, y)
        x = random.randint(0, width - 1)
        y = random.randint(0, height - 1)
```



```
# Kiểm tra nếu ô không phải là CellType.Wall và không trùng với  
điểm bắt đầu  
if grid[x][y].type != CellType.Wall and (x, y) != start:  
    return (x, y)
```

Hàm trên sẽ sinh một trạng thái mục tiêu ngẫu nhiên hợp lệ với luật. Tuy nhiên, hàm này có nhược điểm là sẽ khiến game bị treo nếu bản đồ không có vị trí trống để chọn làm trạng thái kết thúc.

3.2.7 Hàm lượng giá trạng thái

Ở bài toán này, nhóm sử dụng thuật toán A* để tìm đường đi ngắn nhất.

Thuật toán A* kết hợp ưu điểm của cả thuật toán Dijkstra và Greedy Best-First Search. Nó sử dụng cả:

- **Chi phí thực tế** từ trạng thái bắt đầu (x_{start}, y_{start}) đến trạng thái hiện tại $(x_{current}, y_{current})$ (ký hiệu là $g(x, y)$),
- **Chi phí ước lượng** từ trạng thái hiện tại $(x_{current}, y_{current})$ đến trạng thái kết thúc (x_{end}, y_{end}) (ký hiệu là $h(x, y)$).

Tổng chi phí ước tính $f(x, y)$ được tính bằng công thức:

$$f(x, y) = g(x, y) + h(x, y)$$

Trong đó:

- $g(x, y)$ là chi phí thực tế từ trạng thái bắt đầu (x_{start}, y_{start}) đến trạng thái hiện tại $(x_{current}, y_{current})$.
- $h(x, y)$ là chi phí ước tính từ trạng thái hiện tại $(x_{current}, y_{current})$ đến trạng thái kết thúc (x_{end}, y_{end}) .

A* sẽ ưu tiên mở rộng các trạng thái có tổng chi phí $f(x, y)$ nhỏ nhất (kết hợp cả chi phí thực tế và ước tính).

A* là thuật toán hoàn chỉnh và tối ưu, nghĩa là nó sẽ luôn tìm được đường đi ngắn nhất nếu hàm heuristic $h(x, y)$ là hợp lệ (tức là không ước tính quá cao so với chi phí thực tế).

Bằng cách kết hợp chi phí thực tế và heuristic, A* mở rộng khám phá các hướng có tiềm năng và vẫn đảm bảo tìm ra đường đi ngắn nhất, làm cho nó hiệu quả hơn so với Dijkstra trong nhiều trường hợp (tức không cần phải khám phá, đánh giá toàn bộ các trạng thái có thể có).

Hàm lượng giá $h(x, y)$ có thể được định nghĩa theo một trong các cách sau:

- **Khoảng cách Manhattan (Manhattan Distance):**

$$h(x, y) = |x_{end} - x| + |y_{end} - y|$$

Đây là khoảng cách tổng thể giữa vị trí hiện tại (x, y) và điểm kết thúc (x_{end}, y_{end}) , chỉ cho phép di chuyển ngang và dọc. Hàm này thích hợp khi chỉ cho phép di chuyển theo các hướng lên, xuống, trái, phải.

```
def manhatan_distance(a: tuple[int, int], b: tuple[int, int]) -> int:  
    # Hàm tính khoảng cách Manhattan giữa 2 điểm  
    return abs(a[0] - b[0]) + abs(a[1] - b[1])
```



- Khoảng cách Euclidean (Euclidean Distance):

$$h(x, y) = \sqrt{(x_{\text{end}} - x)^2 + (y_{\text{end}} - y)^2}$$

Đây là khoảng cách thẳng từ trạng thái hiện tại (x, y) đến trạng thái kết thúc $(x_{\text{end}}, y_{\text{end}})$. Khoảng cách này cho phép di chuyển theo mọi hướng, kể cả chéo. Hàm này phù hợp hơn khi bài toán cho phép di chuyển tự do trong không gian 2D.

```
def euclidean_distance(a: tuple[int, int], b: tuple[int, int]) -> float:
    # Hàm tính khoảng cách Euclidean giữa 2 điểm
    return math.sqrt((a[0] - b[0]) ** 2 + (a[1] - b[1]) ** 2)
```

3.2.8 Hiện thực giải thuật A*

```
def a_star(grid: CellGrid) -> int:
    """
    Hàm thực hiện thuật toán A*.

    Mục đích là để tìm số bước tối đa nhưng vẫn hiển thị theo số bước
    hiện tại.

    Parameters:
        grid (CellGrid): Lưới chứa các ô và thông tin vị trí bắt đầu và
        kết thúc.

    Returns:
        int: Tổng số bước tối đa để thực hiện qua trình tìm kiếm.
    """

    start, end = grid.get_start(), grid.get_end() # Vị trí bắt đầu và vị
    trí kết thúc
    start.hidden = 0 # Trong an của mỗi ô, tương đương với count nhưng
    an
    start.update_cell(
        0, None, heuristic(end.pos, start.pos)
    ) # Update giá trị của ô bắt đầu
    max_steps = 0 # Biến đếm số bước tối đa

    frontier = PriorityQueue() # Hàng đợi ưu tiên
    frontier.put((0, start)) # Thêm vị trí bắt đầu vào hàng đợi
    visited = set() # Các ô đã duyệt
    visited.add(start.pos) # Thêm vị trí bắt đầu vào Set đã duyệt
    while not frontier.empty(): # Duyệt đến khi hàng đợi rỗng
        _, current = frontier.get()
        # Lấy ô đầu tiên từ hàng đợi, tức ô có độ ưu tiên cao nhất (tức
        chi phí lương gia thấp nhất),
        # độ ưu tiên được tính bằng hàm tổng của Số bước từ ô bắt đầu +
        hàm lương gia từ ô hiện tại đến ô kết thúc

        if current.pos == end.pos: # Nếu ô hiện tại là ô kết thúc thì
            dừng
```




```
        break

    for next in grid.get_neighbors(current.pos):
        # Duyệt qua các ô lân cận của ô hiện tại
        # Lưu ý: thu tu duyệt của ô lân cận không có dinh ma se thay doi
        # tuy thuộc vào vị trí của ô hiện tại
        # 0 hiện tại có tổng hoành độ và tung độ là số Chẵn: thì sẽ duyệt theo thu tu: trên, trái, dưới, phải.
        # Lẻ thì theo thu tu: phải, dưới, trái, trên
        # Điều này giúp ưu tiên tìm theo đường chéo thay vì theo hàng ngang hoặc hàng dọc
        new_cost = current.hidden + 1
        if next.pos not in visited or new_cost < next.hidden:
            visited.add(next.pos)
            heuristic_value = heuristic(end.pos, next.pos)
            priority = new_cost + heuristic_value
            # Nếu ô lân cận chưa duyệt hoặc có trọng số mới nhỏ hơn trọng số cũ
            # thì cập nhật trọng số mới và thêm vào hàng đợi ưu tiên với độ ưu tiên dựa vào
            # số bước từ ô bắt đầu + hàm lượng giá từ ô hiện tại đến ô kết thúc
            next.hidden = new_cost
            frontier.put((priority, next))
            next.update_cell(new_cost, current, heuristic_value)
        max_steps += 1 # Cập nhật số bước đã duyệt (số ô duyệt qua)

    return max_steps
```

Đặc điểm của hàm `a_star`

- **Mô tả hàm:** Hàm thực hiện thuật toán A*. Hàm sẽ thực hiện việc tìm kiếm và trả về số bước tối đa bằng cách sử dụng trọng số ẩn của mỗi ô. Trọng số ẩn (`hidden`) sẽ tương tự như trọng số thật (`count`) nhưng không hiển thị trên lưới. Mục đích là để tìm số bước tối đa nhưng vẫn hiển thị theo số bước hiện tại.

- **Tham số:**

- `grid (CellGrid)`: Lưới chứa các ô và thông tin vị trí bắt đầu và kết thúc.

- **Giá trị trả về:**

- `int`: Tổng số bước tối đa để thực hiện quá trình tìm kiếm.

Các bước hoạt động của giải thuật A*

1. Khởi tạo:

- Thêm ô bắt đầu vào hàng đợi ưu tiên (*Priority Queue*) với giá trị f ban đầu là 0.
- Thiết lập trọng số ẩn (`hidden`) cho ô bắt đầu bằng 0.

2. Lặp qua hàng đợi ưu tiên:

- Lấy ô có f nhỏ nhất ra khỏi hàng đợi (ô hiện tại).



- Nếu ô hiện tại là ô kết thúc, giải thuật dừng.

3. Duyệt các ô lân cận:

- Xét từng ô lân cận của ô hiện tại.
- Tính chi phí mới g từ ô bắt đầu đến ô lân cận.
- Nếu ô lân cận chưa được xét hoặc chi phí mới nhỏ hơn chi phí cũ:
 - Cập nhật chi phí g và hàm lượng giá h .
 - Tính tổng $f = g + h$.
 - Thêm ô lân cận vào hàng đợi ưu tiên.

4. Cập nhật trạng thái:

- Đánh dấu ô hiện tại đã được xét.
- Nếu cần hiển thị trạng thái, cập nhật trên giao diện (nếu có).

5. Lặp lại:

- Tiếp tục xử lý các ô trong hàng đợi cho đến khi tìm được đường đi hoặc hàng đợi rỗng.

6. Truy vết:

- Khi tìm được ô kết thúc, truy vết lại từ ô kết thúc đến ô bắt đầu để xây dựng đường đi tối ưu.

Hàm truy vết để lấy được đường đi ngắn nhất khi đạt được trạng thái kết thúc

```
def backtrack_to_start(end: Cell) -> list[tuple[int, int]]:

    current = end
    path = []

    while current is not None:
        path.append(current.pos)
        current = current.path_from

    path.reverse()
    return path
```

3.3 Định dạng testcase kiểm thử

3.3.1 Input kiểm tra giải thuật

Bên cạnh việc tự sinh ngẫu nhiên một trạng thái khởi đầu có lời giải để thực hiện giải thuật, chương trình cũng cho phép nhận đầu vào qua file text "input.txt" chứa một hoặc nhiều đề bài, định dạng file "input.txt" như sau:

- Dòng đầu tiên là một số nguyên thể hiện kích thước của bản đồ.
- Dòng thứ hai là một danh sách các vị trí của vật cản. Theo mẫu sau:
(x1,y1),(x2,y2),(x3,y3),...(xn,yn)



- Dòng thứ ba là vị trí của trạng thái bắt đầu.
- Dòng thứ ba là vị trí của trạng thái kết thúc.

Ví dụ đầu vào là một file **input1.text** có nội dung như sau:

```
10
(1,1) , (2,2) , (3,3) , (4,4) , (5,5) , (6,6) , (7,7) , (8,8) , (9,9)
(0,9)
(9,0)
```

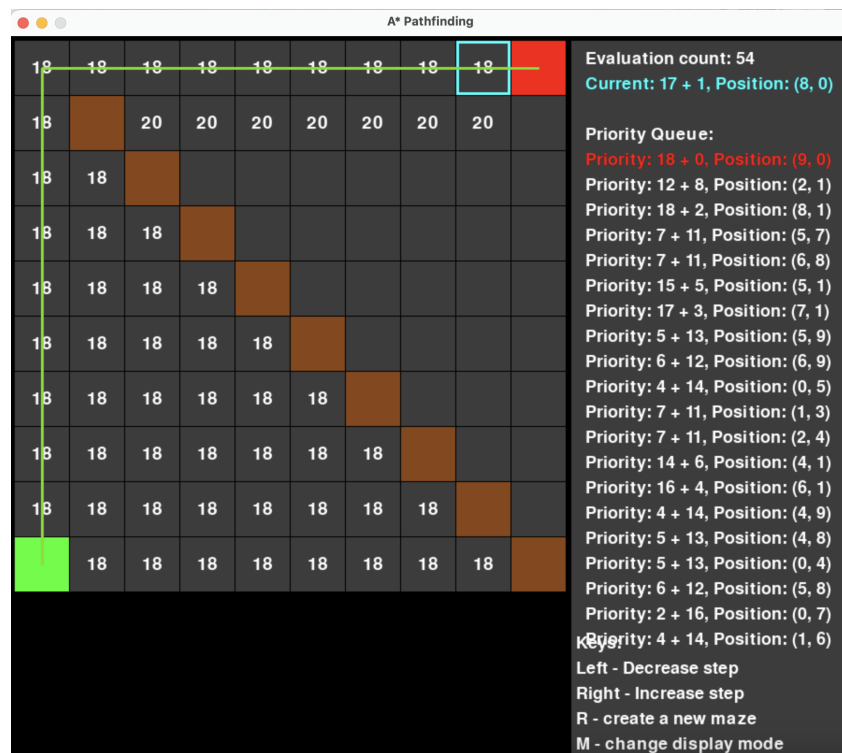
3.3.2 Kết quả

Đường đi ngắn được tìm thấy và lưu vào file **output1.text**

```
(0,9) , (0,8) , (0,7) , (0,6) , (0,5) , (0,4) , (0,3) , (0,2) , (0,1) , (0,0) ,
(1,0) , (2,0) , (3,0) , (4,0) , (5,0) , (6,0) , (7,0) , (8,0) , (9,0)
```

Nhóm có sử dụng giao diện để hiển thị kết quả một cách trực quan.

Đường nối màu xanh từ điểm bắt đầu đến điểm kết thúc là đường đi ngắn nhất từ điểm bắt đầu đến điểm kết thúc.



Hình 20: Hình ảnh trực quan kết quả của một bài toán tìm đường

3.4 Đo đặc thời gian và bộ nhớ tiêu tốn

3.4.1 Test-case 1: Trường hợp kích thước bản đồ 20 * 20

Test case (file "input1.text")



20

```
(1,1),(1,3),(1,4),(1,5),(1,9),(1,12),(1,13),(1,14),(1,16),(1,18),(2,2)
,(2,3),(2,4),(2,8),(2,9),(2,11),(2,12),(2,13),(2,14),(2,17),(2,18)
,(3,4),(3,7),(3,8),(3,9),(3,10),(3,11),(3,12),(3,13),(3,15),(3,16)
,(3,17),(4,1),(4,2),(4,6),(4,7),(4,8),(4,14),(4,15),(4,18),(5,4),(5,9)
,(5,12),(5,16),(6,1),(6,2),(6,5),(6,7),(6,11),(6,13),(6,14),(6,15)
,(6,16),(6,17),(7,3),(7,6),(7,7),(7,8),(7,9),(7,11),(7,13),(7,17)
,(8,2),(8,3),(8,4),(8,10),(8,11),(8,12),(8,13),(8,15),(8,18),(9,1)
,(9,5),(9,9),(9,11),(9,12),(9,13),(9,17),(9,18),(10,3),(10,7),(10,8)
,(10,9),(10,12),(10,13),(10,16),(10,17),(11,1),(11,2),(11,3),(11,6)
,(11,9),(11,12),(11,13),(11,14),(12,4),(12,10),(12,11),(12,12),(12,15)
,(12,16),(12,17),(12,18),(13,2),(13,8),(13,10),(13,11),(13,16),(13,17)
,(14,1),(14,7),(14,13),(15,1),(15,2),(15,5),(15,12),(15,13),(15,15)
,(15,18),(16,1),(16,4),(16,7),(16,8),(16,10),(16,14),(16,15),(16,17)
,(17,3),(17,4),(17,6),(17,7),(17,9),(17,10),(17,15),(18,1),(18,4)
,(18,8),(18,9),(18,12),(18,14),(18,16),(18,18)
(0,2)
(19,6)
```

Kết quả (file "output1.text")

```
(0,2),(0,1),(0,0),(1,0),(2,0),(3,0),(4,0),(5,0),(6,0),(7,0),(8,0),(9,0)
,(10,0),(11,0),(12,0),(13,0),(14,0),(15,0),(16,0),(17,0),(18,0),(19,0)
,(19,1),(19,2),(19,3),(19,4),(19,5),(19,6)
```

Đo đạc thời gian và bộ nhớ

Time: 0.00092983 seconds

Number of Evaluated nodes: 82

Chi tiết sử dụng bộ nhớ sử dụng công cụ @Profile

Line #	Mem usage	Increment	Occurrences	Line Contents
11	77.0 MiB	77.0 MiB	1	@profile
12				def a_star(grid: CellGrid) -> int:
13	77.0 MiB	0.0 MiB	1	start, end = grid.get_start(), grid.get_end()
14	77.0 MiB	0.0 MiB	1	start.hidden = 0
15	77.0 MiB	0.0 MiB	1	start.update_cell(0, None, heuristic(end.pos, start.pos))
16	77.0 MiB	0.0 MiB	1	max_steps = 0
17				
18	77.0 MiB	0.0 MiB	1	frontier = PriorityQueue()
19	77.0 MiB	0.0 MiB	1	frontier.put((0, start))
20	77.0 MiB	0.0 MiB	1	visited = set()
21	77.0 MiB	0.0 MiB	1	visited.add(start.pos)
22	77.1 MiB	0.0 MiB	55	while not frontier.empty():
23	77.1 MiB	0.0 MiB	55	_, current = frontier.get()
24				
25	77.1 MiB	0.0 MiB	55	if current.pos == end.pos:
26	77.1 MiB	0.0 MiB	1	break
27				
28	77.1 MiB	0.0 MiB	224	for next in grid.get_neighbors(current.pos):
29	77.1 MiB	0.0 MiB	170	new_cost = current.hidden + 1
30	77.1 MiB	0.0 MiB	170	if next.pos not in visited or new_cost < next.hidden:
31	77.1 MiB	0.0 MiB	82	visited.add(next.pos)
32	77.1 MiB	0.0 MiB	82	heuristic_value = heuristic(end.pos, next.pos)
33	77.1 MiB	0.0 MiB	82	priority = new_cost + heuristic_value
34	77.1 MiB	0.0 MiB	82	next.hidden = new_cost
35	77.1 MiB	0.0 MiB	82	frontier.put((priority, next))
36	77.1 MiB	0.0 MiB	82	next.update_cell(new_cost, current, heuristic_value)
37	77.1 MiB	0.0 MiB	54	max_steps += 1
38				
39	77.1 MiB	0.0 MiB	1	return max_steps

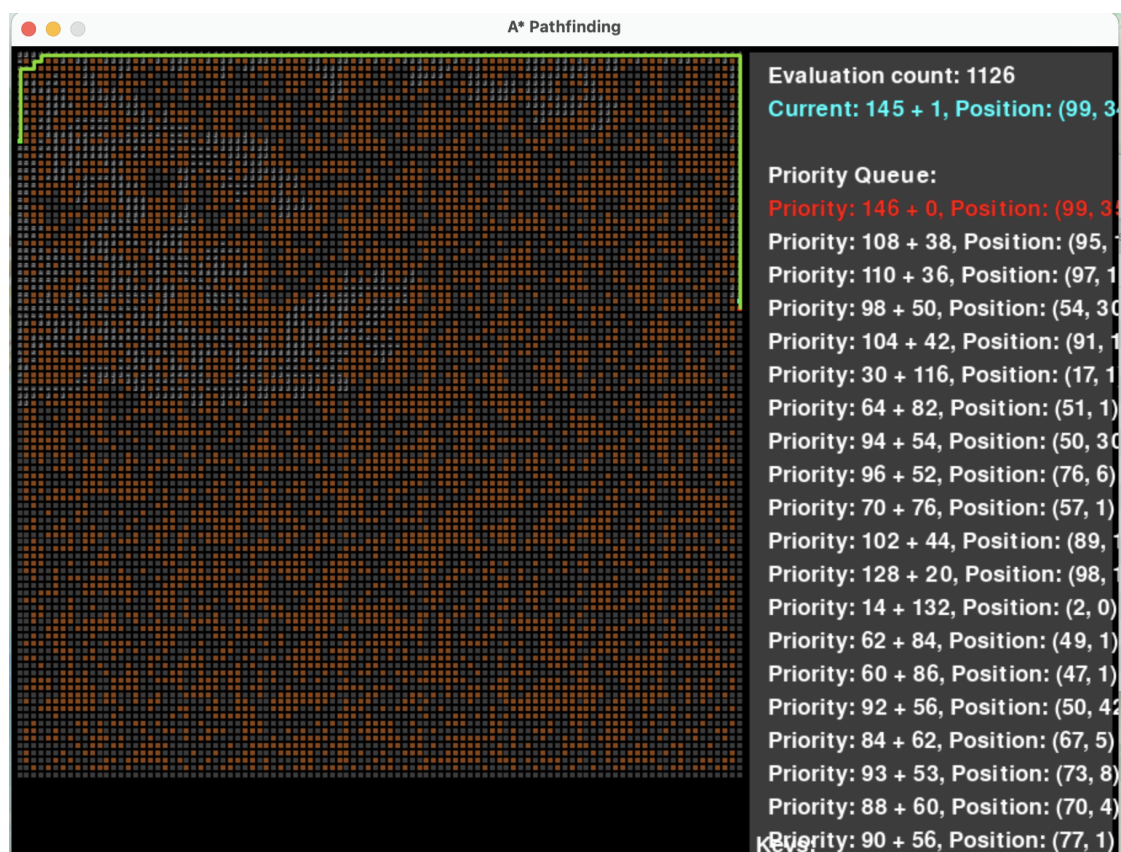
Hình 21: Hình ảnh chi tiết sử dụng bộ nhớ của test case 1



3.4.2 Test-case 2: Trường hợp kích thước bản đồ 100 * 100

Test case được lưu vào file "input2.text" và kết quả đường đi ngắn nhất được lưu vào file "output2.text".

Dưới đây là hình ảnh trực quan của test-case 2.



Hình 22: Hình ảnh bản đồ 100 * 100 và đường đi ngắn nhất được tìm thấy

Đo đặc thời gian và bộ nhớ

Time: 0.00799584 seconds

Number of Evaluated nodes: 1126

Chi tiết sử dụng bộ nhớ sử dụng công cụ @Profile



Line #	Mem usage	Increment	Occurrences	Line Contents
10	57.0 MiB	57.0 MiB	1	@profile
11				def a_star(grid: CellGrid) -> int:
12	57.0 MiB	0.0 MiB	1	start, end = grid.get_start(), grid.get_end()
13	57.0 MiB	0.0 MiB	1	start.hidden = 0
14	57.0 MiB	0.0 MiB	1	start.update_cell(0, None, heuristic(end.pos, start.pos))
15	57.0 MiB	0.0 MiB	1	max_steps = 0
16				
17	57.0 MiB	0.0 MiB	1	frontier = PriorityQueue()
18	57.0 MiB	0.0 MiB	1	frontier.put((0, start))
19	57.0 MiB	0.0 MiB	1	visited = set()
20	57.0 MiB	0.0 MiB	1	visited.add(start.pos)
21	57.1 MiB	0.0 MiB	1127	while not frontier.empty():
22	57.1 MiB	0.0 MiB	1127	_, current = frontier.get()
23				
24	57.1 MiB	0.0 MiB	1127	if current.pos == end.pos:
25	57.1 MiB	0.0 MiB	1	break
26				
27	57.1 MiB	0.0 MiB	3936	for next in grid.get_neighbors(current.pos):
28	57.1 MiB	0.0 MiB	2810	new_cost = current.hidden + 1
29	57.1 MiB	0.0 MiB	2810	if next.pos not in visited or new_cost < next.hidden:
30	57.1 MiB	0.0 MiB	1241	visited.add(next.pos)
31	57.1 MiB	0.0 MiB	1241	heuristic_value = heuristic(end.pos, next.pos)
32	57.1 MiB	0.0 MiB	1241	priority = new_cost + heuristic_value
33	57.1 MiB	0.0 MiB	1241	next.hidden = new_cost
34	57.1 MiB	0.0 MiB	1241	frontier.put((priority, next))
35	57.1 MiB	0.0 MiB	1241	next.update_cell(new_cost, current, heuristic_value)
36	57.1 MiB	0.0 MiB	1126	max_steps += 1
37				
38	57.1 MiB	0.0 MiB	1	return max_steps

Hình 23: Hình ảnh chi tiết sử dụng bộ nhớ của test case 2

Từ kết quả đo đặc bộ nhớ trong ảnh trên, có thể thấy bộ nhớ sử dụng cho giải thuật để giải bài toán với testcase 2 là 0.1 MiB.

3.4.3 Test-case 3: Trường hợp kích thước bản đồ 2000 * 2000

Test case được lưu vào file "input3.text" và kết quả đường đi ngắn nhất được lưu vào file "output3.text".

Đo đặc thời gian và bộ nhớ

Time: 1.39688968 seconds

Number of Evaluated nodes: 159590

Chi tiết sử dụng bộ nhớ sử dụng công cụ @Profile

Line #	Mem usage	Increment	Occurrences	Line Contents
10	1052.9 MiB	1052.9 MiB	1	@profile
11				def a_star(grid: CellGrid) -> int:
12	1052.9 MiB	0.0 MiB	1	start, end = grid.get_start(), grid.get_end()
13	1052.9 MiB	0.0 MiB	1	start.hidden = 0
14	1052.9 MiB	0.0 MiB	1	start.update_cell(0, None, heuristic(end.pos, start.pos))
15	1052.9 MiB	0.0 MiB	1	max_steps = 0
16				
17	1052.9 MiB	0.0 MiB	1	frontier = PriorityQueue()
18	1052.9 MiB	0.0 MiB	1	frontier.put((0, start))
19	1052.9 MiB	0.0 MiB	1	visited = set()
20	1052.9 MiB	0.0 MiB	1	visited.add(start.pos)
21	1069.1 MiB	-31.8 MiB	159591	while not frontier.empty():
22	1069.1 MiB	-31.6 MiB	159591	_, current = frontier.get()
23				
24	1069.1 MiB	-31.9 MiB	159591	if current.pos == end.pos:
25	1069.1 MiB	0.0 MiB	1	break
26				
27	1069.1 MiB	-110.3 MiB	543431	for next in grid.get_neighbors(current.pos):
28	1069.1 MiB	-75.8 MiB	383841	new_cost = current.hidden + 1
29	1069.1 MiB	-78.5 MiB	383841	if next.pos not in visited or new_cost < next.hidden:
30	1069.1 MiB	-27.6 MiB	162534	visited.add(next.pos)
31	1069.1 MiB	-30.9 MiB	162534	heuristic_value = heuristic(end.pos, next.pos)
32	1069.1 MiB	-31.0 MiB	162534	priority = new_cost + heuristic_value
33	1069.1 MiB	-34.2 MiB	162534	next.hidden = new_cost
34	1069.1 MiB	-34.1 MiB	162534	frontier.put((priority, next))
35	1069.1 MiB	-34.2 MiB	162534	next.update_cell(new_cost, current, heuristic_value)
36	1069.1 MiB	-31.8 MiB	159590	max_steps += 1
37				
38	1069.1 MiB	0.0 MiB	1	return max_steps

Evaluation node: 159590

Hình 24: Hình ảnh chi tiết sử dụng bộ nhớ của test case 3



Từ kết quả đo đặc bộ nhớ trong ảnh trên, có thể thấy bộ nhớ sử dụng cho giải thuật để giải bài toán với testcase 3 là 16.2 MiB.

3.4.4 Test-case 4: Trường hợp kích thước bản đồ 5000 * 5000

Test case được lưu vào file "input4.text" và kết quả đường đi ngắn nhất được lưu vào file "output4.text".

Đo đặc thời gian và bộ nhớ

Time: 16.5969858 seconds

Number of Evaluated nodes: 1822956

Chi tiết sử dụng bộ nhớ sử dụng công cụ @Profile

Line #	Mem usage	Increment	Occurrences	Line Contents
10	5164.8 MiB	5164.8 MiB	1	@profile
11				def a_star(grid: CellGrid) -> int:
12	5164.8 MiB	0.0 MiB	1	start, end = grid.get_start(), grid.get_end()
13	5164.8 MiB	0.0 MiB	1	start.hidden = 0
14	5164.8 MiB	0.0 MiB	1	start.update_cell(0, None, heuristic(end.pos, start.pos))
15	5164.8 MiB	0.0 MiB	1	max_steps = 0
16				
17	5164.8 MiB	0.0 MiB	1	frontier = PriorityQueue()
18	5164.8 MiB	0.0 MiB	1	frontier.put((0, start))
19	5164.8 MiB	0.0 MiB	1	visited = set()
20	5164.8 MiB	0.0 MiB	1	visited.add(start.pos)
21	5170.3 MiB	-2884392049.6 MiB	1822957	while not frontier.empty():
22	5170.3 MiB	-2884392385.5 MiB	1822957	_, current = frontier.get()
23				
24	5170.3 MiB	-2884392513.2 MiB	1822957	if current.pos == end.pos:
25	2864.4 MiB	-2306.0 MiB	1	break
26				
27	5170.3 MiB	-9782518738.3 MiB	6183467	for next in grid.get_neighbors(current.pos):
28	5170.3 MiB	-6898127232.2 MiB	4360511	new_cost = current.hidden + 1
29	5170.3 MiB	-6898127270.3 MiB	4360511	if next.pos not in visited or new_cost < next.hidden:
30	5170.3 MiB	-2898312273.3 MiB	1830157	visited.add(next.pos)
31	5170.3 MiB	-2898312378.6 MiB	1830157	heuristic_value = heuristic(end.pos, next.pos)
32	5170.3 MiB	-2898312452.6 MiB	1830157	priority = new_cost + heuristic_value
33	5170.3 MiB	-2898312578.0 MiB	1830157	next.hidden = new_cost
34	5170.3 MiB	-2898312866.1 MiB	1830157	frontier.put((priority, next))
35	5170.3 MiB	-2898313000.1 MiB	1830157	next.update_cell(new_cost, current, heuristic_value)
36	5170.3 MiB	-2884391901.5 MiB	1822956	max_steps += 1
37				
38	2864.4 MiB	0.0 MiB	1	return max_steps

Evaluation node: 1822956
Thời gian chạy: 1219.3802411556244 giây

Hình 25: Hình ảnh chi tiết sử dụng bộ nhớ của test case 4

Từ kết quả đo đặc bộ nhớ trong ảnh trên, có thể thấy bộ nhớ sử dụng cho giải thuật để giải bài toán với testcase 4 là 2305.9 MiB

3.4.5 Test-case 5: Trường hợp kích thước bản đồ 10000 * 10000

Test case được lưu vào file "input5.text" và kết quả đường đi ngắn nhất được lưu vào file "output5.text".

Đo đặc thời gian và bộ nhớ

Time: 118.74473977088928 seconds

Number of Evaluated nodes: 4816334

3.4.6 Thống kê kết quả và nhận xét

Khi phân tích test-case 4, với giải thuật A* trên bản đồ 5000x5000 có tỉ lệ vật cản 0.5 và 1,822,956 node được lượng giá trong thời gian chạy 16.60 giây, ta thấy rằng tỉ lệ node được lượng giá là khoảng 14.58%. Điều này chứng tỏ A* chỉ lượng giá những khu vực có thể dẫn đến đích, giúp giảm thiểu số lượng node cần kiểm tra.



Map Size	Time (Seconds)	Explored Nodes	Memory Usage
10x10	0.0009298	82 (16.4%)	≈ 0 MiB
100x100	0.0079958	1126 (11.26%)	0.1 MiB
2000x2000	1.3968897	159590 (15.96%)	16.2 MiB
5000x5000	16.5969858	1822956 (14.58%)	2305.9 MiB
10000x10000	118.7447398	4816334 (9.63%)	rất lớn

Kết quả này cho thấy giải thuật A* không chỉ tìm ra đường đi ngắn nhất mà còn tối ưu quá trình tìm kiếm, giảm thiểu số lượng tính toán không cần thiết.

Nhóm rút ra kết luận rằng A* hoạt động hiệu quả trong việc tìm đường trên bản đồ lớn, đặc biệt khi chỉ duyệt qua những ô tiềm năng thay vì duyệt toàn bộ bản đồ, giúp tiết kiệm thời gian và bộ nhớ.

Tài liệu

- [1] Paria Sarzaeim *N-Puzzle Solver with Search Algorithms*, 2020. [Đường dẫn truy cập](#)
- [2] *Iterative Deepening Search(IDS) or Iterative Deepening Depth First Search(IDDFS)*, 2024. [Đường dẫn truy cập](#)
- [3] *How to check if an instance of 15 puzzle is solvable?* , 2023. [Đường dẫn truy cập](#)
- [4] David D. *Solving Sudoku puzzles with Genetic Algorithm*, 2019. [Đường dẫn truy cập](#)
- [5] Christian T. Rodriguez Jacobs *Sudoku Genetic Algorithm*, 2020. [Đường dẫn truy cập](#)
- [7] Red Blob Games *Introduction to the A* Algorithm*, 2014 - 2023. [Đường dẫn truy cập](#)