ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA

BÀI TẬP LỚN NHẬP MÔN TRÍ TUỆ NHÂN TẠO CO3061

SỬ DỤNG GIẢI THUẬT BLIND SEARCH VÀ HEURISTIC ĐỂ GIẢI BÀI TOÁN TRONG GAME SUDOKU VÀ MINESWEEPER

Lớp:	CN01	
GVHD:	Vương Bá Thịnh	
Khoa:	Khoa học và Kỹ thuật máy tính	
SV thực hiện:	Bùi Hồ Hải Đăng	2153289
	Nguyễn Hữu Nhật Minh	2153575
	Đặng Thành Anh	2153153

TP. Hồ Chí Minh, Tháng 9/2023

Table of content

1 Danh sách thành viên

STT	Họ và tên	MSSV	Phân chia công việc
1	Đặng Thành Anh	2153153	Giải thuật cho Sodoku
2	Bùi Hồ Hải Đăng	2153289	Viết Report, tổng hợp link
3	Nguyễn Hữu Nhật Minh	2153575	Giải thuật cho Minesweeper

2 Mô tả đề tài

Có rất nhiều bài toán được dùng để giới thiệu các vấn đề trong môn Trí tuệ nhân tạo, như: Block world, Water-jug, N-puzzle... Trong học kì này, sinh viên được yêu cầu hiện thực một số giải thuật tìm kiếm (bằng Python) để giải một số bài toán cụ thể.

3 Cơ sở lý thuyết

3.1 Thuật toán suy diễn lùi (BackTracking)

Thuật toán suy diễn lùi (Backtracking) là một dạng của Thuật toán tìm kiếm theo chiều sâu (DFS). Thuật toán suy diễn lùi là một kỹ thuật giải quyết vấn đề được sử dụng để tìm kiếm giải pháp cho các bài toán tìm kiếm và tối ưu hóa. Nó hoạt động bằng cách xây dựng dần dần một giải pháp từ các lựa chọn khả thi, và quay lại khi không còn lựa chọn nào khả thi nữa. **Cách thức hoạt động:**

- Khởi tạo: Bắt đầu với một tập hợp các lựa chọn khả thi.
- Lựa chọn: Chọn một lựa chọn từ tập hợp các lựa chọn khả thi.
- Kiểm tra: Kiểm tra xem lựa chọn được chọn có dẫn đến giải pháp hay không.
- Nếu có giải pháp: Dừng lại.
- Nếu không có giải pháp:
  - Loại bỏ lựa chọn: Loại bỏ lựa chọn được chọn khỏi tập hợp các lựa chọn khả thi.
  - Quay lại: Quay lại bước 2.

Ưu điểm:

- Đơn giản: Dễ dàng hiểu và triển khai.
- Hiệu quả: Có thể tìm ra giải pháp cho nhiều loại bài toán khác nhau.
- Hoàn chỉnh: Đảm bảo tìm thấy giải pháp nếu có, miễn là thuật toán có thể chạy đủ lâu.

Nhược điểm:

- Có thể tốn kém về mặt tính toán cho các bài toán lớn: Thuật toán có thể thử nhiều khả năng không cần thiết trước khi tìm thấy giải pháp.
- Có thể gặp vấn đề về backtracking: Thuật toán có thể bị kẹt trong vòng lặp vô hạn nếu không có chiến lược thoát phù hợp.

Ứng dụng:

- Giải các bài toán tổ hợp: Tìm kiếm các tập con, hoán vị, tổ hợp của một tập hợp.
- Giải các bài toán tối ưu hóa: Tìm kiếm giá trị tối ưu cho một hàm mục tiêu với các ràng buộc nhất định.
- Giải các bài toán logic: Kiểm tra tính khả thi của các mệnh đề logic.
- Chơi game: Tìm kiếm các nước đi hợp lệ trong các trò chơi như cờ vua, cờ tướng.

3.2 Thuật toán Heuristic

Heuristic là các quy tắc hoặc chiến lược được sử dụng để giải quyết vấn đề một cách hiệu quả. Nó dựa trên kiến thức chuyên môn và kinh nghiệm để đưa ra phán đoán thông minh về cách thức thực hiện một nhiệm vụ nhất định. Heuristic không đảm bảo giải pháp tối ưu, nhưng nó thường giúp tiết kiệm thời gian và tài nguyên so với các phương pháp tìm kiếm đầy đủ.
Thuật giải Heuristic là một sự mở rộng khái niệm thuật toán. Nó thể hiện cách giải bài toán với các đặc tính sau:

- Thường tìm được lời giải tốt (nhưng không chắc là lời giải tốt nhất)
- Giải bài toán theo thuật giải Heuristic thường dễ dàng và nhanh chóng đưa ra kết quả hơn so với giải thuật tối ưu, vì vậy chi phí thấp hơn.
- Thuật giải Heuristic thường thể hiện khá tự nhiên, gần gũi với cách suy nghĩ và hành động của con người.

Có nhiều phương pháp để xây dựng một thuật giải Heuristic, trong đó người ta thường dựa vào một số nguyên lý cơ sở như sau:

- Nguyên lý vét cạn thông minh: Trong một bài toán tìm kiếm nào đó, khi không gian tìm kiếm lớn, ta thường tìm cách giới hạn lại không gian tìm kiếm hoặc thực hiện một kiểu dò tìm đặc biệt dựa vào đặc thù của bài toán để nhanh chóng tìm ra mục tiêu.
- Nguyên lý tham lam (Greedy): Lấy tiêu chuẩn tối ưu (trên phạm vi toàn cục) của bài toán để làm tiêu chuẩn chọn lựa hành động cho phạm vi cục bộ của từng bước (hay từng giai đoạn) trong quá trình tìm kiếm lời giải.
- Nguyên lý thứ tự: Thực hiện hành động dựa trên một cấu trúc thứ tự hợp lý của không gian khảo sát nhằm nhanh chóng đạt được một lời giải tốt.

Hàm Heuristic: Trong việc xây dựng các thuật giải Heuristic, người ta thường dùng các hàm Heuristic. Đó là các hàm đánh giá thô, giá trị của hàm phụ thuộc vào trạng thái hiện tại của bài toán tại mỗi bước giải. Nhờ giá trị này, ta có thể chọn được cách hành động tương đối hợp lý trong từng bước của thuật giải.

3.3 Thuật toán tìm kiếm A*

Thuật toán A* (phát âm là "A-star") là một trong những giải thuật Heuristic. A* một thuật toán duyệt đồ thị và tìm đường đi được sử dụng trong nhiều lĩnh vực khoa học máy tính do tính hoàn chỉnh, tối ưu và hiệu quả tối ưu của nó. Cho một biểu đồ có trọng số , một nút nguồn và một nút mục tiêu, thuật toán sẽ tìm đường đi ngắn nhất (đối với các trọng số đã cho) từ nguồn đến mục tiêu. **Nguyên tắc hoạt động:**

- Giả định: Mỗi cạnh trong đồ thị có một trọng số được gán cho nó.
- Mục tiêu: Tìm đường đi ngắn nhất từ điểm bắt đầu đến điểm kết thúc.
- Cách thức: Sử dụng hai giá trị để đánh giá mỗi nút trong đồ thị:
  - f(n): Tổng chi phí ước tính từ điểm bắt đầu đến điểm kết thúc, bao gồm chi phí đã đi đến nút hiện tạig(n): Chi phí thực tế đã đi từ điểm bắt đầu đến nút hiện tại. (g(n)) và chi phí ước tính còn lại (h(n)).
  - g(n): Chi phí thực tế đã đi từ điểm bắt đầu đến nút hiện tại.
  - h(n): Chi phí ước tính từ nút hiện tại đến điểm kết thúc.

Thuật toán duy trì hai tập:

- Tập mở: Chứa các nút chưa được khám phá nhưng có thể khám phá trong tương lai.
- Tập đóng: Chứa các nút đã được khám phá.

Thuật toán lặp lại các bước sau:

- Chọn nút có f(n) nhỏ nhất từ tập mở.
- Xóa nút đó khỏi tập mở và thêm nó vào tập đóng.
- Kiểm tra tất cả các nút lân cận của nút đã chọn:
  - Nếu nút lân cận chưa được khám phá, hãy thêm nó vào tập mở và cập nhật g(n) và f(n) của nó.
  - Nếu nút lân cận đã được khám phá nhưng f(n) mới nhỏ hơn f(n) cũ, hãy cập nhật g(n) và f(n) của nó và di chuyển nó từ tập đóng sang tập mở.
- Lặp lại các bước 1-3 cho đến khi tìm thấy nút kết thúc hoặc tập mở rỗng.

Ưu điểm:

- Hoàn chỉnh: Đảm bảo tìm thấy đường đi ngắn nhất nếu có.
- Tối ưu: Tìm thấy đường đi ngắn nhất trong số tất cả các đường đi có thể.
- Hiệu quả: Sử dụng heuristics để giảm số lượng nút cần khám phá.

Nhược điểm:

- Có thể tốn kém về mặt tính toán cho các đồ thị lớn.
- Heuristics cần được lựa chọn cẩn thận để đảm bảo tính hoàn chỉnh và tối ưu.

Ứng dụng:

- Lập bản đồ và định vị: Tìm đường đi ngắn nhất giữa hai vị trí trên bản đồ.
- Trò chơi: Tìm đường đi ngắn nhất cho nhân vật trong trò chơi.
- Mạng máy tính: Tìm đường đi hiệu quả nhất để truyền dữ liệu.
- Robot học: Lập kế hoạch di chuyển cho robot.

4 Sudoku

Sudoku là một trò chơi trí tuệ nổi tiếng, thu hút nhiều người tham gia thuộc nhiều tầng lớp, độ tuổi khác nhau. Sudoku ra đời ở Nhật và không lâu sau đã đó đã nhanh chóng lan rộng trên toàn thế giới. Ngày này, sudoku co nhieu bản thể khác nhau: 9x9, 3x3, 4x4, 6x6, 5x5, 7x7, 8x8, 16x16, 12x12, 25x25, ... Đối với đề tài này, chỉ áp dụng cho sudoku dạng chuẩn 9x9 Một đề sudoku là một hình vuông, mỗi chiều có 9 ô nhỏ, hợp thành 9 cột, 9 hàng và được chia thành 9 ô lớn 3x3. Một vài ô nhỏ được đánh số, đó là những manh mối duy nhất để bạn tìm lời giải. Tuỳ theo mức độ nhiều hay ít của các manh mối, các câu đố được xếp loại dễ, trung bình, khó hay cực khó.

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

Cách chơi Sudoku khá đơn giản là điền số từ 1 đến 9 vào những ô trống sao cho mỗi cột dọc, mỗi hàng ngang, mỗi phân vùng nhỏ có đủ các từ 1 đến 9 mà không được lặp lại.

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

4.1 Ứng dụng để giải bài toán sudoku bằng python3

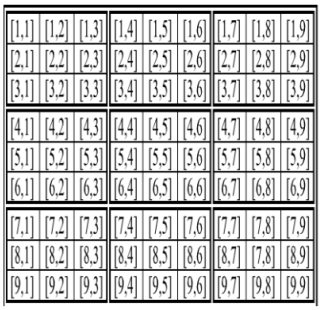
4.1.1 Input và Output cho bài toán

Input: Để Sudoku do người dùng nhập hoặc đã cho trước dưới dạng file *.sdk hoăc *.txt. Một file *.sdk hoặc *.txt sẽ lưu một đề sudoku dưới dạng một ma trận vuông(Ví dụ: file board.txt)
Ví dụ:
309000605
000009000
800507003
061030700
000102000
004050310
200801007
000600000
103000406
Hoặc một chuỗi các số:
003050100060900020002001308000000030500060000090000600300600901070003080001070200
Ví dụ như trong file easier_puzzles.txt và file harder_puzzles.txt **Trong đó:**

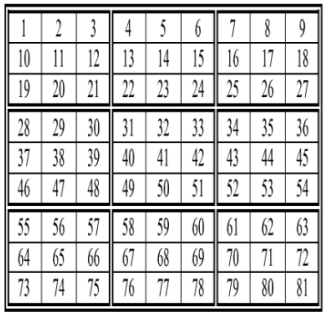
- Số 0: là vị trí các ô trống trong đề sudoku
- Số khác 0: là vị trí các ô số mà đề cho.

Output: Đáp án sudoku. Có thể có nhiều đáp án, hoặc không có đáp án. Giải thuật kết thuất khi không còn số 0 nào trong Input cả nghĩa là tất cả ô trống trong Sudoku được lấp đầy bới các số từ 1 đến 9 và tuân thủ đúng luật của trò chơi Sudoku **Cách để xác định dữ liệu đầu vào:**

- Xác định bằng [hàng, cột], với cách xác định này, ta dễ dàng duyệt theo cột và theo hàng



- Xác định theo số thứ tự từ 1 đến 81. Cách này ta sẽ sử dụng chủ yếu để duyệt toàn bộ 81 ô, hay để lấy đề bài, xuất kết quả ra giao diện người dùng.



Nhưng để thống nhất trong code thì chúng ta sẽ đổi dạng chuỗi số sang dạng mảng bằng lệnh input_board_auto(output, source) trong file retrieve_board.py

```
def input_board_auto(output, source):
    done = False
    puzzles = split_data(source)
    with open(output, 'a') as file:
        for puzzle in puzzles:
            count = 0
            for i in range(81):
                if count == 8:
                    file.write(puzzle[i])
                    count = 0
                    file.write('\n')
                    continue
                file.write(puzzle[i] + "␣")
                count += 1
            file.write('\n')

def split_data(source):
    with open(source, 'r') as src:
        all_puzzles = src.readlines()
        return all_puzzles
```

4.1.2 Ứng dụng giải thuật Backtracking

Chương trình giải dựa trên giải thuật Backtracking. Tư tưởng của giải thuật này là chia nhỏ bài toán lớn thành các bài toán phần tử, giải các bài toán phần tử, ứng với mỗi trường hợp giải được của bài toán phần tử đó, ta tìm lời giải cho bài toán phần tử tiếp theo cho đến khi bài toán lớn được giải quyết.

```
<Khởi tạo các thông số cần thiết>
Void Try (int i)
{
        <Nếu cấu hình hiện tại đáp ứng đủ yêu cầu thì xuất ra và>
        <Duyệt các khả năng có thể có ở vị trí i>
        {
                <Đánh dấu là đã duyệt ở vị trí i>
                <Gọi hàm Try (i+1)>
                <Hủy đánh dấu ở trên>
        }
}
```

```python
def find_empty(board):
    for i in range(len(board)):
        for j in range(len(board[0])):
            if board[i][j] == 0:
                return (i, j)
    return None

def valid(board, num, pos):
    # Check row
    for i in range(len(board[0])):
        if board[pos[0]][i] == num and pos[1] != i:
            return False
    # Check column
    for i in range(len(board)):
        if board[i][pos[1]] == num and pos[0] != i:
            return False
    # Check box
    box_x = pos[1] // 3
    box_y = pos[0] // 3
    for i in range(box_y*3, box_y*3 + 3):
        for j in range(box_x * 3, box_x*3 + 3):
            if board[i][j] == num and (i,j) != pos:
                return False
    return True
```

Đầu tiên ta khởi tạo các thông số cần thiết như:

- find_empty(board): Dùng để xác định các vị trí trống trong Sudoku hàm nhận vào một mảng 9x9 và trả về vị trí của ô nếu ô đó là ô trống và trả về None nếu không còn ô trống
- valid(board, num, pos): Hàm kiểm tra thử số ở ô vừa điền có hộp lên không bằng cách kiểm tra trên hàng, cột và trong vùng 3x3 xem thừ có số nào khác trừng với số vừa điền không nếu không trả về True ngược lại trả về False

```python
#  Backtracking
def solveDFS(board):
    find = find_empty(board)
    if not find:
        return True
    else:
        # if have a empty cell, get the row and column index
        row, col = find
    for i in range(1,10):
        # verify if is a valid number, considering sudoku rules
        if (valid(board, i , (row,col))):
            # insert the valid value to board
            board[row][col] = i

            # call solveDFS recursive to continue this board
            # if the result is true, the sudoku has been solved
            if (solveDFS(board)):
                return True
            # if not, change the actual valid value to 0
            # and continue testing next value
            board[row][col] = 0
    return False
```

Đến với giải thuật Backtracking ta có hàm solveDFS(board) với đầu vào là biến Board là một mảng 9x9. Các bước thực hiện như đã giới thiệu phái trên:

- Kiểm tra có ô nào còn trống không
  - Nếu không trả về True và kết thúc giải thuật
  - Nếu không lấy vị trí ô còn trống
- Chọn các số từ 1 đến 9 để điền vào ô trống. Nếu như số điền là hợp lệ ta thì ta xét tiếp vị trí trống khác bằng cách sử dụng đệ quy.
  - Nếu như hàm đệ quy trả về True nghĩa là đã hoàn thành bài toán ta trả về True.
  - Nếu như không thể điền tiếp vị trí tiếp theo không hợp lệ thì ta xóa số vừa điền vào bằng cách trả về 0(Nghĩa là trả về ô trống) rồi điền số khác
- Làm như vậy đến khi không có bất kì trường hợp nào trả về True nghĩa là không tìm ra lời giải cho bài toán ta trả về False

4.1.3 Ứng dụng giải thuật Heuristic: A*

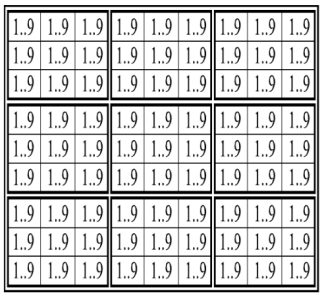
Ta thấy rằng khi đang xét ở vị trí i, ta đưa ra các phương án để tiếp tục xét vị trí i+1, có những phương án sẽ làm cho vị trí i+1, i+2 ... có thể tìm các phương án và dẫn tới kết quả cuối

cùng (gọi là phương án khả thi) và có những phương án sẽ không có kết quả (gọi là phương án bất khả thi). Do đó, để chương trình chạy nhanh, ta cần phải loại bỏ các phương án bất khả thi càng nhiều càng tốt.

Giải quyết vấn đề: Ta đánh dấu các khả năng điền số và tìm cách loại bỏ các khả năng bất khả thi.

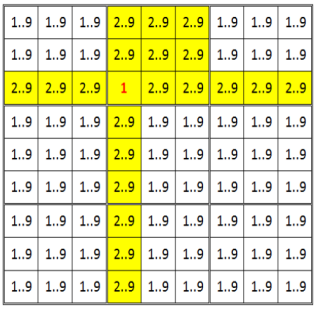
Giải sử ở mỗi vị trí ta lưu thêm khả năng mà ô đố có thể điền.

Ví dụ:



Và khi có một ô đã được điền số thì các ô còn lại trên cùng một cột hoặc một hàng hoặt một vùng sẽ có khả năng điền được không chứa số mà ô này có.

Ví dụ:



```
    #  AStar
def solveAStar(board):
    # create a array of possible board solutions
    solutions = []

    # loop to run in all board
    for i in range(9):
        for j in range(9):
            # create a list with all posibilities
            x = [1, 2, 3, 4, 5, 6, 7, 8, 9]

            # check if cell is empty
            if (board[i][j] == 0):
```

```
                # loop to verify and remove incorrect numbers from row,
                #column and quadrant
                for k in range(9):
                    if (board[i][k] != 0 and board[i][k] in x):
                        x.remove(board[i][k])
                    if (board[k][j] != 0 and board[k][j] in x):
                        x.remove(board[k][j])

                quad_x = j // 3 #used integer division to get
                quad_y = i // 3 #the integer value from positions
                for m in range(quad_y * 3, quad_y * 3 + 3):
                    for n in range(quad_x * 3, quad_x * 3 + 3):
                        if (board[m][n] != 0 and board[m][n] in x
                        and (m, n) != (i,j)):
                            x.remove(board[m][n])
                tmp = []
                tmp.append(i)
                tmp.append(j)
                tmp.append(len(x))
                tmp.append(x)

                # insert ordered, lowest first
                index = 0
                for k in range(len(solutions)):
                    if (solutions[k][2] > tmp[2] and tmp[2] > 0):
                        index = k
                        break
                    else:
                        index+=1
                solutions.insert(index, tmp)

    if (len(solutions) > 0):
        row = solutions[0][0]
        col = solutions[0][1]
        # loop in array of solutions of current recursion
        for i in solutions[0][3]:
            # insert the valid value to board
            board[row][col] = i
            # if not find empty cell print result or call recursive to next cell
            if (not find_empty(board)):
                return True
            elif solveAStar(board):
                return True
            # if not, change the actual valid value to 0 and continue testing next value
            board[row][col] = 0
```

```
59          return False
```

Đến với giải thuật A* ta có hàm solveAStar(board) với đầu vào là biến Board là một mảng 9x9. Cách thức thực hiện hàm:

- Đầu tiên ta sẽ khai báo một danh sách solutions mỗi phần tử trong danh sách sẽ gồm 2 thành phần:
  - index (thể hiện thứ tự các giải pháp khả thi) trong đó index = 0 là trường hợp khả thi mà khả năng số có thể điền tại vị trí đó là thấp nhất
  - tmp(là giải pháp khả thi) nó gồm 4 thành phần:
    - Trong đó hai thành phần đầu lần lượt là vị trí hàng và cột
    - Thành phần thứ ba đó là số các số khả thi mà ô đó có thể điền
    - Thành phần thứ tư là danh sách các số khả thi mà ô đó có thể điền
- Tiếp đến ta duyệt qua tất cả các ô. Lập ra danh sách x là các số khả thi mà một ô có thể điền với gia trị ban đầu có chín số từ 1 đến 9
  - Duyệt qua ô trống bất kì mà đề cho ta lần lượt xét trên các hàng các cột và các vùng của nó để tìm ra danh sách khả thi (solutions)
- Sau khi kết thúc việc duyệt qua các tất cả các ô ta đã có được danh sách khả năng khả thi (solutions) của các ô còn trống. Ta tới bước tiếp theo là điền số vào những ô còn trống:
  - Ta xét nếu như danh sách khả thi (solution) có số phần tử lớn hơn 0 ta xét vị trí index = 0(trường hợp khả thi mà khả năng số có thể điền tại vị trí đó là thấp nhất). Lấy vị trí của trường hợp này.
  - Sau đó chọn một số trong các số khả thi có thể điền tại vị trí này:
    - Nếu như không còn ô trong nữa ta trả về True kết thúc giải thuật
    - Ta gọi đệ quy để xét đến vị trí tiếp theo. Nếu hàm đệ quy trả về True ta trả về True và kết thúc giải thuật. Nếu không ta trả vị trí này về 0 (tức trả về ô trống)
- Nếu như không có trường hợp nào trả về True nghĩa là không tìm ra lời giải cho bài toán ta trả về False

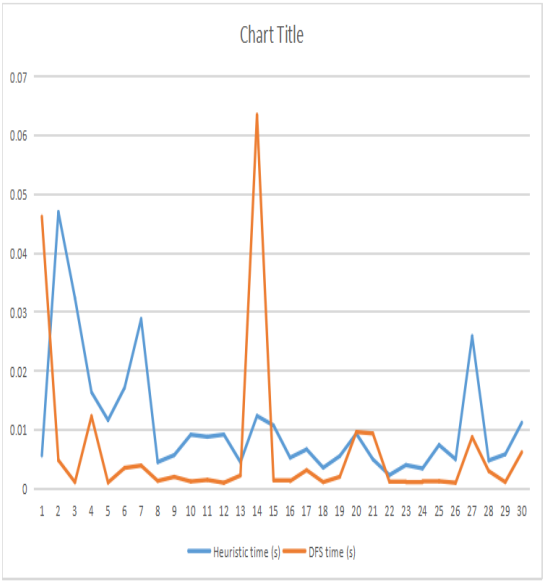
4.1.4 DEMO

So sánh hai giải thuật Dưới đây là bảng thống kê thời gian và sự tiêu tốn bộ nhớ của từng giải thuật:

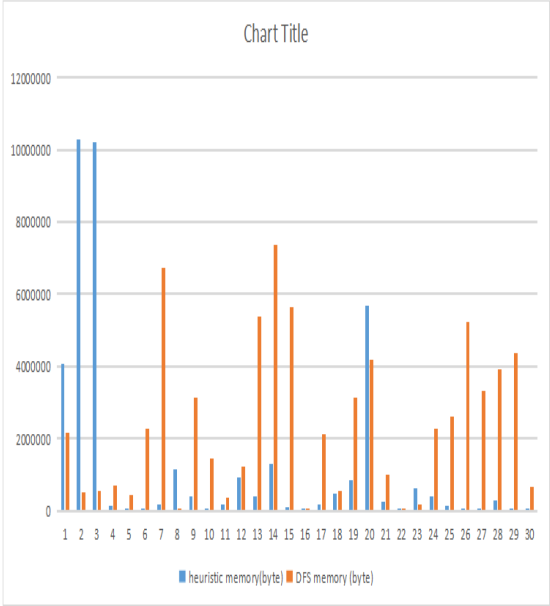
STT	Board	Heuristic time (s)	heuristic mer	DFS time (s)	DFS memory (byte)
1	[5, 3, 0, 0, 7, 0, 0, 0, 0],[6, 0, 0, 1, 9, 5, 0, 0, 0],[0, 9, 8, 0, 0, 0, 0, 6, 0],[8, 0, 0, 0, 6, 0, 0, 0, 3],[4, 0, 0, 8, 0, 3, 0, 0, 1],[7, 0, 0, 0, 2, 0, 0, 0, 6],[0, 6, 0, 0, 0, 0, 2, 8, 0],[0, 0, 0, 4, 1, 9, 0, 0, 5],[0, 0, 0, 0, 8, 0, 0, 7, 9]	0.005637884	4083712	0.046178341	2162688
2	[9, 0, 0, 0, 0, 0, 0, 0, 0],[6, 0, 1, 0, 2, 5, 3, 0, 0],[0, 0, 7, 0, 3, 0, 0, 0, 0],[0, 5, 0, 0, 4, 0, 0, 0, 0],[8, 0, 4, 2, 5, 0, 7, 0, 6],[0, 7, 0, 0, 6, 0, 4, 5, 0],[2, 0, 3, 5, 4, 0, 0, 0, 0],[0, 0, 0, 0, 7, 0, 2, 0, 3],[0, 0, 0, 0, 0, 0, 0, 1, 0]	0.046957731	10297344	0.004826546	524288
3	[1, 0, 0, 0, 9, 0, 0, 3, 0],[9, 0, 0, 0, 0, 0, 2, 0, 0],[3, 2, 8, 7, 4, 1, 6, 5, 9],[0, 8, 0, 1, 0, 3, 0, 0, 4],[5, 0, 0, 2, 0, 4, 0, 8, 0],[0, 7, 4, 0, 0, 0, 0, 2, 1],[6, 0, 7, 0, 0, 0, 0, 0, 0],[0, 0, 0, 0, 3, 0, 1, 0, 8],[2, 6, 0, 7, 5, 0, 2, 9, 5],[0, 5, 7, 0, 0, 8, 0, 4, 0]	0.032378197	10211328	0.001102209	548864
4	[0, 2, 0, 0, 0, 5, 3, 0, 0],[0, 0, 0, 1, 0, 0, 8, 0, 2],[0, 1, 4, 0, 8, 2, 9, 0, 0],[0, 0, 0, 0, 0, 6, 0, 0, 0],[7, 3, 6, 4, 2, 0, 0, 9, 0],[0, 0, 5, 0, 6, 8, 1, 4, 0],[0, 6, 9, 2, 0, 7, 3, 0, 0],[0, 0, 7, 3, 0, 0, 0, 6, 0],[0, 0, 1, 0, 2, 0, 0, 0, 0],[0, 0, 0, 0, 0, 0, 0, 0, 0]	0.016376495	135168	0.012298346	708608
5	[0, 0, 0, 7, 5, 0, 1, 3, 2],[4, 0, 0, 0, 2, 0, 0, 0, 0],[0, 0, 0, 0, 0, 0, 0, 0, 0],[0, 0, 4, 0, 0, 0, 7, 0, 0],[0, 3, 0, 8, 0, 0, 7, 9, 5],[0, 7, 4, 0, 9, 3, 1, 0, 8],[0, 8, 0, 0, 5, 6, 0, 0, 0],[0, 5, 0, 0, 0, 0, 0, 3, 1],[4, 0, 8, 5, 0, 0, 0, 9, 5],[1, 0, 1, 3, 7, 5, 0, 0, 0]	0.011639833	57344	0.001041651	430080
6	[0, 9, 8, 5, 3, 0, 6, 0, 2],[2, 0, 0, 0, 9, 0, 8, 3, 0],[3, 0, 6, 8, 2, 0, 0, 9, 0],[0, 0, 2, 7, 6, 3, 9, 5, 0],[5, 6, 0, 0, 0, 0, 8, 7, 0],[1, 0, 3, 9, 0, 0, 0, 0, 0],[0, 0, 1, 7, 0, 0, 0, 6, 0],[8, 0, 5, 3, 5, 0, 0, 2, 4]	0.017141104	57344	0.003519535	2289664
7	[0, 7, 0, 4, 0, 2, 6, 0, 0],[5, 0, 0, 0, 0, 0, 2, 0, 0],[8, 0, 1, 5, 0, 6, 0, 3, 0],[6, 1, 0, 2, 0, 3, 0, 0, 0],[0, 0, 0, 0, 0, 0, 0, 1, 3],[3, 5, 0, 8, 0, 9, 7, 6, 0],[0, 6, 0, 0, 0, 0, 0, 5, 0],[0, 0, 0, 0, 0, 0, 0, 0, 0],[0, 0, 0, 0, 0, 0, 0, 2, 6],[5, 7, 0, 8, 0, 0, 0, 0, 0]	0.028782368	163840	0.003918886	6729728
8	[0, 6, 2, 1, 0, 0, 0, 7, 5],[0, 0, 0, 0, 0, 0, 0, 0, 0],[0, 4, 0, 0, 2, 0, 1, 7, 0],[2, 0, 0, 0, 0, 0, 8, 4, 0],[4, 0, 0, 5, 0, 2, 6, 0, 0],[8, 9, 0, 4, 6, 3, 2, 0, 5],[0, 2, 0, 5, 7, 0, 0, 0, 0],[0, 7, 0, 0, 0, 5, 1, 0, 2],[6, 0, 5, 0, 0, 4, 7, 1, 3]	0.004511595	1134592	0.001319408	81920
9	[7, 2, 0, 9, 0, 8, 0, 0, 0],[0, 0, 0, 1, 4, 0, 2, 0, 7],[0, 1, 5, 0, 0, 0, 0, 4, 9],[0, 0, 7, 0, 0, 1, 0, 8, 0],[3, 7, 9, 0, 0, 5, 0, 2, 0],[0, 0, 0, 2, 6, 0, 0, 0, 3],[5, 0, 0, 0, 7, 0, 1, 0, 0],[2, 7, 0, 0, 5, 0, 0, 0, 0],[4, 0, 0, 0, 2, 1, 0, 0, 0]	0.005698442	389120	0.001995325	3145728
10	[5, 4, 2, 7, 3, 6, 8, 9, 0],[0, 0, 0, 0, 0, 1, 6, 0, 7],[1, 0, 6, 0, 0, 0, 2, 0, 5],[0, 0, 3, 4, 5, 7, 0, 0, 0],[8, 0, 5, 1, 0, 0, 4, 0, 0],[7, 3, 0, 6, 0, 0, 5, 1, 2],[9, 0, 7, 0, 0, 0, 4, 5, 0],[0, 1, 0, 0, 0, 0, 9, 0, 0],[6, 0, 0, 8, 0, 0, 0, 7, 0]	0.009161949	69632	0.001223326	1458176
11	[0, 0, 2, 3, 0, 6, 7, 4, 0],[7, 4, 0, 0, 2, 0, 0, 5, 9],[0, 0, 0, 5, 0, 0, 2, 0, 0],[3, 0, 6, 5, 0, 0, 2, 0, 0],[0, 0, 1, 8, 0, 0, 0, 0, 3],[6, 7, 0, 0, 5, 4, 8, 9, 1],[5, 0, 0, 6, 3, 0, 0, 0, 2],[1, 5, 6, 0, 3, 2, 0, 4, 0],[0, 0, 9, 0, 1, 0, 0, 0, 0],[2, 0, 0, 0, 0, 8, 5, 1, 0]	0.008805513	176128	0.001479149	376832
12	[9, 5, 1, 0, 7, 8, 0, 4, 0],[2, 0, 3, 0, 1, 9, 5, 0, 0],[0, 0, 0, 4, 0, 0, 1, 2, 9],[0, 0, 7, 3, 0, 0, 6, 0, 2],[0, 2, 0, 7, 4, 5, 9, 0, 0],[5, 0, 9, 0, 2, 0, 4, 7, 0],[6, 9, 5, 0, 8, 2, 7, 0, 0],[1, 0, 8, 5, 0, 0, 3, 0, 0],[3, 0, 2, 0, 0, 0, 0, 0, 0]	0.009181499	925696	0.000996828	1224704
13	[0, 6, 0, 0, 7, 0, 0, 0, 5],[2, 0, 0, 0, 0, 0, 0, 0, 1],[8, 0, 5, 9, 2, 1, 0, 3, 7],[0, 0, 4, 0, 6, 2, 0, 5, 0],[0, 0, 0, 0, 3, 1, 0, 0, 8],[4, 0, 6, 5, 3, 0, 0, 7, 0],[0, 4, 0, 0, 0, 0, 7, 0, 0],[7, 0, 0, 6, 0, [0, 1, 5, 0, 0, 0, 0, 0, 0],[7, 0, 0, 0, 5, 0, 0, 4, 0],[0, 0, 0, 0, 0, 0, 0, 0, 0]	0.00450182	397312	0.002236128	5361664
14	[0, 4, 0, 0, 0, 0, 3, 0, 7],[0, 7, 0, 0, 0, 0, 0, 0, 0],[0, 0, 0, 0, 0, 0, 7, 5, 0],[3, 0, 5, 0, 0, 2, 0, 0, 0],[0, 0, 0, 0, 0, 0, 2, 5, 0],[5, 6, 3, 2, 0, 1, 0, 0, 0],[0, 0, 0, 0, 0, 0, 9, 0, 5],[0, 3, 0, 1],[0, 0, 0, 0, 0, 0, 7, 0, 6],[0, 5, 0, 8, 3, 5, 0, 0, 2],[1, 2, 6, 4, 0, 0, 5, 0, 0]	0.012347221	1302528	0.063445091	7372800
15	[6, 0, 7, 0, 9, 4, 2, 3, 5],[4, 0, 9, 0, 0, 8, 0, 0, 6],[3, 0, 0, 6, 0, 7, 0, 4, 0],[5, 7, 0, 9, 0, 2, 3, 0, 0],[0, 0, 2, 0, 1, 6, 0, 0, 7],[0, 6, 1, 0, 5, 0, 4, 8, 2],[1, 0, 8, 0, 0, 5, 0, 2, 0],[0, 4, 3, 2, 0, 0, 8, 1, 0],[0, 5, 0, 8, 4, 1, 0, 5, 0]	0.010731936	90112	0.001407862	5644288
16	[0, 3, 8, 1, 6, 0, 0, 0, 0],[9, 5, 0, 0, 7, 0, 3, 6, 0],[0, 4, 0, 3, 0, 2, 7, 0, 5],[0, 1, 2, 0, 0, 0, 9, 0, 0],[8, 0, 0, 0, 1, 6, 0, 0, 7],[0, 6, 5, 9, 8, 0, 1, 3, 4],[3, 2, 9, 7, 0, 0, 5, 0, 0],[0, 0, 0, 5, 0, 0, 0, 0, 2],[9, 0, 1, 0, 15, 0, 4, 6, 0],[8, 9, 0, 0, 0]	0.005255699	20480	0.001278162	77824
17	[6, 0, 5, 0, 0, 8, 0, 7, 0],[0, 0, 0, 2, 0, 0, 0, 0, 5],[0, 4, 0, 0, 9, 5, 3, 0, 2],[9, 7, 0, 2, 0, 0, 3, 0, 1],[1, 0, 0, 0, 0, 7, 0, 0, 0],[0, 0, 0, 0, 1, 9, 5, 8, 0],[4, 9, 0, 0, 0, 0, 5, 3, 0],[0, 0, 0, 0, 0, 0, 0, 2, 0],[0, 0, 0, 0, 0, 0, 0, 0, 0]	0.006665707	163840	0.003140926	2129920
18	[3, 0, 0, 6, 0, 0, 0, 8, 9],[0, 6, 5, 0, 0, 3, 0, 0, 2],[4, 9, 0, 1, 8, 5, 3, 7, 0],[7, 5, 9, 0, 0, 6, 8, 0, 0],[1, 2, 0, 0, 0, 7, 9, 6, 0],[0, 0, 0, 8, 9, 1, 7, 2, 0],[9, 0, 4, 3, 2, 5, 1, 7, 0],[5, 0, 1, 0, 0, 0, 0, 0, 0],[0, 0, 0, 5, 0, 0, 0, 4, 0]	0.003554821	466944	0.001109838	548864
19	[0, 0, 7, 0, 9, 0, 2, 3, 5],[4, 0, 9, 0, 0, 0, 0, 0, 6],[0, 0, 0, 6, 0, 7, 0, 0, 0],[5, 0, 9, 0, 0, 3, 0, 0, 0],[0, 0, 2, 0, 1, 6, 0, 0, 7],[0, 6, 0, 0, 5, 0, 0, 8, 2],[1, 0, 8, 0, 0, 6, 2, 0, 0],[0, 0, 3, 2, 0, 0, 9, 0, 1],[0, 0, 0, 3, 2, 0, 0, 9, 1],[0, 9, 0, 8, 0, 1, 0, 0, 0]	0.005528688	835584	0.002010584	3145728
20	[0, 0, 8, 9, 0, 0, 0, 3, 5],[3, 0, 0, 6, 2, 0, 8, 1, 0],[5, 0, 0, 0, 8, 0, 0, 0, 0],[8, 0, 1, 0, 5, 0, 0, 0, 2],[7, 0, 2, 0, 0, 0, 0, 2, 0],[0, 0, 7, 0, 2, 0, 0, 0, 0],[0, 3, 0, 6, 8, 1, 0, 0, 0],[0, 0, 7, 0, 0, 0, 6, 0, 0],[0, 0, 0, 0, 0, 0, 0, 9, 5],[0, 0, 0, 0, 0, 0, 0, 0, 0]	0.00934267	5693440	0.009610415	4194304
21	[0, 0, 0, 0, 0, 1, 0, 2, 0],[2, 0, 0, 8, 0, 5, 1, 0, 6],[1, 5, 7, 3, 0, 0, 0, 0, 4],[5, 0, 1, 2, 6, 0, 0, 0, 8],[0, 9, 0, 7, 0, 0, 0, 0, 5],[0, 0, 6, 0, 0, 0, 4, 7, 3],[0, 3, 2, 0, 0, 0, 0, 0, 0],[0, 0, 0, 0, 0, 0, 0, 0, 0],[0, 0, 0, 0, 0, 0, 0, 0, 0]	0.004981518	266240	0.00937438	1011712
22	[0, 0, 3, 8, 4, 0, 6, 2, 0],[2, 0, 4, 5, 0, 0, 0, 7, 8],[0, 0, 5, 0, 7, 2, 4, 0, 0],[0, 0, 7, 6, 1, 0, 5, 8, 2],[8, 0, 1, 2, 0, 5, 0, 6, 0],[0, 0, 6, 0, 0, 7, 0, 0, 1],[0, 0, 9, 0, 0, 3, 5, 6, 7],[7, 5, 2, 3, 5, 4, 0, 0, 0],[0, 0, 8, 0, 5, 0, 0, 0, 0],[0, 0, 0, 0, 0, 0, 0, 0, 0]	0.002304316	49152	0.001186132	12288
23	[7, 9, 5, 4, 0, 0, 1, 8, 0],[6, 0, 0, 1, 9, 0, 0, 0, 2],[1, 0, 0, 7, 0, 5, 0, 9, 3],[4, 5, 0, 1, 0, 3, 5, 0, 0],[0, 7, 0, 0, 0, 0, 0, 1, 4],[0, 0, 0, 4, 6, 0, 2, 0, 0],[0, 7, 0, 0, 4, 2, 6, 1, 5],[5, 6, 0, 0, 0, 0, 0, 4, 7],[2, 0, 9, 6, 7, 1, 0, 0, 0]	0.00400281	610304	0.001106501	159744

Link đầy đủ của bảng dữ liệu với 30 Input:

https://docs.google.com/spreadsheets/d/1Vcx7EpBHK0-VuFvJ2wLxzMPVq5rumU2s/edit?usp=drive_link&ouid=101528424145030177469&rtpof=true&sd=true



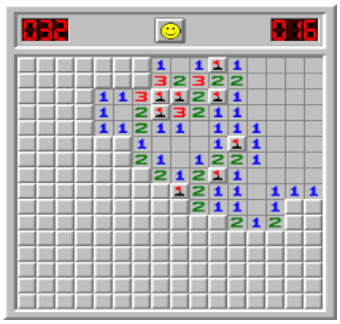
Về thời gian của của hai giải thuật ta thấy không chênh lệch quá nhiều, giải thuật quay lùi có nhiều lần thời gian nhanh hơn giải thậu A* một chút. Một phần là các Input có độ khó tương đối dễ. Nếu đặt trong các chương trình có nhiều tính năng như trong game hoặc là Input có độ khó cao thì giải thuật A* sẽ có lợi thế về thời gian hơn.



Về sự tiêu tốn bộ nhớ của hai giải thuật thì ta thấy giải thuật quay lùi có nhiều lần có sự tiêu tốn bộ nhớ lớn hơn vì giải thuật quay lùi sử dụng đệ quy khá nhiều. Giải thuật A* cũng có sử dụng đệ quy nhưng vì đã loại bỏ bớt các trường hợp khả thi nên thường tốn dung lượng ít hơn

5 Minesweeper

Minesweeper là một trò chơi giải đố trên máy tính dành cho một người chơi. Trò chơi bao gồm một "bãi mìn" là những ô vuông có thể chứa "mìn", và người chơi cần phải dựa vào những con số thể hiện số mìn xung quanh để mở hết tất cả những ô vuông trống mà không kích nổ quả mìn nào. Trò chơi được xây dựng như một chương trình giải trí cài đặt trên hệ điều hành Microsoft Windows.



Trong Dò mìn, người chơi phải mở được tất cả các ô không có mìn trên một bảng ô vuông, đồng thời không được kích nổ bất cứ một quả mìn nào. Trò chơi được xếp hạng bằng thời gian hoàn thành, vì vậy việc hoàn thành trò chơi trong thời gian sớm nhất cũng là một mục tiêu quan trọng đối với người chơi đã thành thạo.

5.1 Ứng dụng để giải bài toán sudoku bằng python3

5.1.1 Input và Output cho bài toán

Input: Cho một mảng 2D arr[][] có kích thước N*M, đại diện cho ma trận trò chơi Minesweeper, trong đó mỗi ô chứa một số nguyên trong khoảng [0, 9], đại diện cho số lượng mìn trong ô đó và tám ô xung quanh, nhiệm vụ là giải trò chơi Minesweeper và phơi lên tất cả các mìn trong ma trận. In 'X' cho ô chứa mìn và '_' cho tất cả các ô trống khác. Nếu không thể giải trò chơi Minesweeper, in "-1". **Cách tạo input:** Để giải ma trận Minesweeper cho trước arr[][], nó phải là một đầu vào hợp lệ, tức là ma trận Minesweeper phải có thể giải được. Do đó, ma trận đầu vào được tạo trong hàm generateMineField(). Thực hiện các bước sau để tạo ma trận Minesweeper đầu vào:

- Các đầu vào cho việc tạo ra là kích thước của mảng N và M và cũng xác suất P (hoặc mật độ) của mảng mìn.
- Một xác suất P bằng 0 nếu không có mìn nào trong mảng mìn và P bằng 100 nếu tất cả các ô đều là mìn trong mảng mìn.
- Một số ngẫu nhiên được chọn cho mỗi ô và nếu số ngẫu nhiên nhỏ hơn P, một mìn được gán cho lưới và một mảng boolean mines[][] được tạo ra.
- Dữ liệu đầu vào cho bộ giải ràng buộc là trạng thái của mỗi ô đếm số mìn của chính nó và tám ô xung quanh.

```
Generated Input:
0 1 1 1 1 1 1
1 3 3 3 2 2 1
1 2 2 2 2 2 1
1 2 2 2 1 1 0
1 2 1 1 1 1 1
1 3 2 2 1 1 1
1 3 2 2 1 1 1
```

Time Complexity: O(ROW*COL)
Auxiliary Space: O(ROW*COL)

```
    def __init__(self, height=8, width=8, mines=8):

        # Set initial width, height, and number of mines
        self.height = height
        self.width = width
        self.mines = set()

        # Initialize an empty field with no mines
        self.board = []
        for i in range(self.height):
            row = []
            for j in range(self.width):
                row.append(False)
            self.board.append(row)

        # Add mines randomly
        while len(self.mines) != mines:
            i = random.randrange(height)
            j = random.randrange(width)
            if not self.board[i][j]:
                self.mines.add((i, j))
                self.board[i][j] = True

        # At first, player has found no mines
        self.mines_found = set()
```

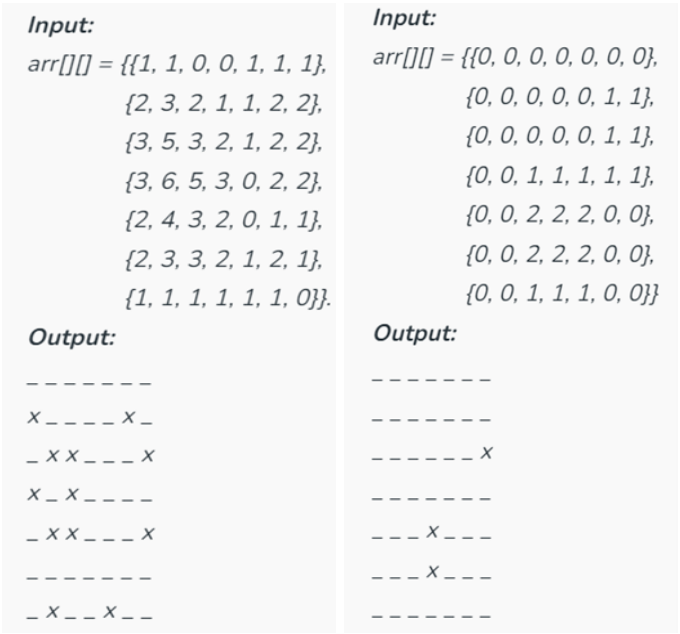
5.1.2 Ứng dụng giải thuật Backtracking

Vấn đề cho trước có thể được giải bằng Backtracking. Ý tưởng là lặp qua từng ô của ma trận, dựa trên thông tin có sẵn từ các ô láng giềng, gán một mìn cho ô đó hoặc không.

- Khởi tạo một ma trận, gọi là grid[][], và visited[][] để lưu trữ lưới kết quả và theo dõi các ô đã được thăm khi duyệt lưới. Khởi tạo tất cả các giá trị lưới là sai.
- Khai báo một hàm đệ quy solveMineSweeper() để chấp nhận các mảng arr[][], grid[][], và visited[][] làm tham số.
  - Nếu tất cả các ô đã được thăm và một mìn được gán cho các ô thỏa mãn lưới đầu vào cho trước, thì trả về true cho cuộc gọi đệ quy hiện tại.
  - Nếu tất cả các ô đã được thăm nhưng giải pháp không thỏa mãn lưới đầu vào, trả về false cho cuộc gọi đệ quy hiện tại.
  - Nếu hai điều kiện trên được tìm thấy là sai, thì tìm một ô chưa được thăm (x, y) và đánh dấu (x, y) là đã thăm.
  - Nếu một mìn có thể được gán cho vị trí (x, y), thực hiện các bước sau:

    * Đánh dấu grid[x][y] là true.
    * Giảm số lượng mìn của các ô láng giềng của (x, y) trong ma trận arr[][] đi 1.
    * Gọi đệ quy cho solveMineSweeper() với (x, y) có mìn và nếu nó trả về true, thì một giải pháp tồn tại. Trả về true cho cuộc gọi đệ quy hiện tại.
    * Nếu không, đặt lại vị trí (x, y) tức là đánh dấu grid[x][y] là false và tăng số lượng mìn của các ô láng giềng của (x, y) trong ma trận arr[][] lên 1.
  - Nếu hàm solveMineSweeper() với (x, y) không có mìn, trả về true, thì có nghĩa là một giải pháp tồn tại. Trả về true từ cuộc gọi đệ quy hiện tại.
  - Nếu cuộc gọi đệ quy trong bước trên trả về false, có nghĩa là giải pháp không tồn tại. Do đó, trả về false từ cuộc gọi đệ quy hiện tại.

- Nếu giá trị trả về của hàm solveMineSweeper(grid, arr, visited) là true, thì một giải pháp tồn tại. In ma trận grid[][] là giải pháp cần thiết. Ngược lại, in "-1".



Time Complexity: O(2N * M * N * M)
Auxiliary Space: O(N * M)

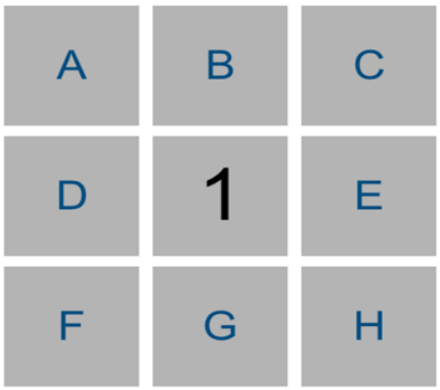
```
    def SolveMinesweeper(grid, arr, visited, N, M):
    # Function call to check if each cell is visited and the solved grid is satisfying the
    done = isDone(arr, visited)

    # If the solution exists and all cells are visited
    if done:
        return True

```

```
    x, y = findUnvisited(visited)

    # Function call to check if all the cells are visited or not
    if x == -1 and y == -1:
        return False

    # Mark cell (x, y) as visited
    visited[x][y] = True

    # Function call to check if it is safe to assign a mine at (x, y)
    if isSafe(arr, x, y, N, M):
        # Mark the position with a mine
        grid[x][y] = True

        # Recursive call with (x, y) having a mine
        if SolveMinesweeper(grid, arr, visited, N, M):
            # If solution exists, then return true
            return True

        # Reset the position x, y
        grid[x][y] = False
        for i in range(9):
            if isValid(x + dx[i], y + dy[i], N, M):
                arr[x + dx[i]][y + dy[i]] += 1

    # Recursive call without (x, y) having a mine
    if SolveMinesweeper(grid, arr, visited, N, M):
        # If solution exists then return true
        return True

    # Mark the position as unvisited again
    visited[x][y] = False

    # If no solution exists
    return False

# Function to perform generate and solve a minesweeper
def minesweeperOperations(arr, N, M):
    # Stores the final result
    grid = np.zeros((N, M), dtype=bool)

    # Stores whether the position (x, y) is visited or not
    visited = np.zeros((N, M), dtype=bool)

    # If the solution to the input minesweeper matrix exists
```

```
    if SolveMinesweeper(grid, arr, visited, N, M):
        # Function call to print the grid[][]
        printGrid(grid)
    # No solution exists
    else:
        print("No solution exists")
```

5.1.3 Ứng dụng giải thuật Heuristic: Logic mệnh đề

Mục tiêu của bạn trong dự án này sẽ là xây dựng một trí tuệ nhân tạo có thể chơi trò chơi Minesweeper. Một cách chúng ta có thể biểu diễn kiến thức của một trí tuệ nhân tạo về một trò chơi Minesweeper là bằng cách làm cho mỗi ô là một biến phát biểu mà đúng nếu ô chứa một mìn và sai nếu không.



Thông tin chúng ta có hiện tại là gì? Dường như bây giờ chúng ta biết rằng một trong tám ô lân cận là một mìn. Do đó, chúng ta có thể viết một biểu thức logic như dưới đây để chỉ ra rằng một trong các ô lân cận là một mìn.

```
Or(A, B, C, D, E, F, G, H)
```

Nhưng thực tế, chúng ta biết nhiều hơn so với những gì biểu thức này nói. Câu logic trên biểu diễn ý tưởng rằng ít nhất một trong tám biến đó là đúng. Nhưng chúng ta có thể đưa ra một tuyên bố mạnh mẽ hơn: chúng ta biết rằng chính xác một trong tám biến đó là đúng. Điều này đưa cho chúng ta một câu logic phát biểu như dưới đây.

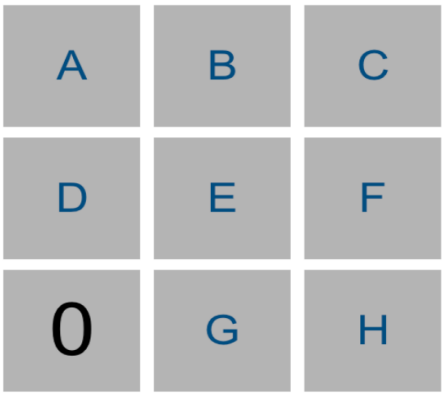
```
Or(
And(A, Not(B), Not(C), Not(D), Not(E), Not(F), Not(G), Not(H)),
And(Not(A), B, Not(C), Not(D), Not(E), Not(F), Not(G), Not(H)),
And(Not(A), Not(B), C, Not(D), Not(E), Not(F), Not(G), Not(H)),
And(Not(A), Not(B), Not(C), D, Not(E), Not(F), Not(G), Not(H)),
And(Not(A), Not(B), Not(C), Not(D), E, Not(F), Not(G), Not(H)),
And(Not(A), Not(B), Not(C), Not(D), Not(E), F, Not(G), Not(H)),
```

```
    And(Not(A), Not(B), Not(C), Not(D), Not(E), Not(F), G, Not(H)),
    And(Not(A), Not(B), Not(C), Not(D), Not(E), Not(F), Not(G), H)
)
```

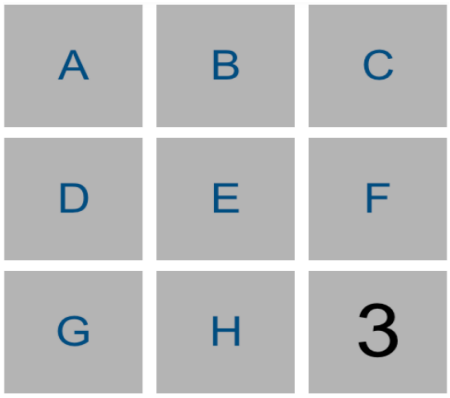
Đó là một biểu thức khá phức tạp! Và đó chỉ là để biểu diễn ý nghĩa của việc một ô có số 1. Nếu một ô có số 2 hoặc 3 hoặc một giá trị khác, biểu thức có thể còn dài hơn nữa.
Cố gắng thực hiện kiểm tra mô hình trên loại vấn đề này cũng sẽ nhanh chóng trở nên không thể giải quyết: trên một lưới 8x8, kích thước mà Microsoft sử dụng cho cấp độ Beginner, chúng ta sẽ có 64 biến, và do đó có 2^{64} mô hình có thể kiểm tra - quá nhiều để máy tính tính toán trong bất kỳ thời gian hợp lý nào. Chúng ta cần một cách biểu diễn kiến thức tốt hơn cho vấn đề này.
Biểu diễn kiến thức: Thay vào đó, chúng ta sẽ biểu diễn mỗi câu của kiến thức của trí tuệ nhân tạo của chúng ta như dưới đây.

```
{A, B, C, D, E, F, G, H} = 1
```

Mỗi câu logic trong biểu diễn này có hai phần: một tập hợp các ô trên bảng mà câu này liên quan đến, và một số lượng, đại diện cho số lượng ô đó là mìn. Câu logic trên nói rằng trong các ô A, B, C, D, E, F, G và H, chính xác 1 trong số chúng là một mìn. Chúng ta sử dụng cách này vì, Một phần, nó thích hợp cho một số loại suy luận. **Ví dụ như:**



Sử dụng kiến thức từ số ở góc dưới bên trái, chúng ta có thể xây dựng câu $\{D, E, G\} = 0$ để chỉ ra rằng trong các ô D, E và G, chính xác 0 trong số chúng là mìn. Theo cách suy luận tự nhiên, chúng ta có thể suy ra từ câu đó rằng tất cả các ô phải là an toàn. Mở rộng ra, bất cứ khi nào chúng ta có một câu mà số lượng là 0, chúng ta biết rằng tất cả các ô của câu đó phải là an toàn.
Ví dụ khác như:



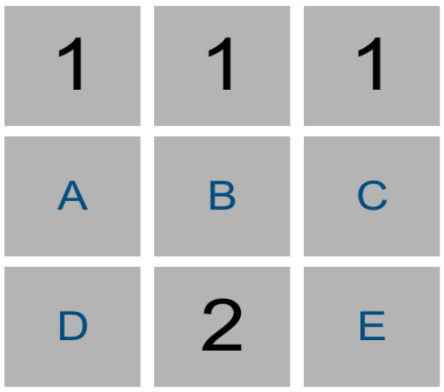
Trí tuệ nhân tạo của chúng tôi sẽ xây dựng câu $\{E, F, H\} = 3$. Theo cách suy luận tự nhiên, chúng ta có thể suy ra rằng tất cả các ô E, F và H đều là mìn. Nói chung, mỗi khi số lượng ô bằng với số lượng, chúng ta biết rằng tất cả các ô của câu đó phải là mìn.

Nói chung, chúng tôi chỉ muốn các câu của chúng tôi liên quan đến các ô vẫn chưa biết là an toàn hoặc là mìn. Điều này có nghĩa là, một khi chúng tôi biết một ô có phải là mìn hay không, chúng tôi có thể cập nhật các câu của mình để đơn giản hóa chúng và có thể đưa ra các kết luận mới.

Ví dụ, nếu trí tuệ nhân tạo của chúng tôi biết câu $\{A, B, C\} = 2$, chúng ta vẫn chưa có đủ thông tin để kết luận điều gì. Nhưng nếu chúng ta được cho biết rằng C là an toàn, chúng ta có thể loại bỏ C khỏi câu đó hoàn toàn, để lại câu $\{A, B\} = 2$ (điều này, tình cờ, cho phép chúng ta đưa ra một số kết luận mới.)

Tương tự, nếu trí tuệ nhân tạo của chúng tôi biết câu $\{A, B, C\} = 2$, và chúng ta được cho biết rằng C là mìn, chúng ta có thể loại bỏ C khỏi câu và giảm giá trị của số lượng (vì C là mìn đã đóng góp vào số lượng đó), đưa chúng ta đến câu $\{A, B\} = 1$. Điều này là hợp lý: nếu hai trong số A, B và C là mìn, và chúng ta biết rằng C là mìn, thì phải có một trong số A và B là mìn.

Nếu chúng ta thông minh hơn một chút, có một loại suy luận cuối cùng chúng ta có thể thực hiện.



Hãy xem xét chỉ hai câu mà trí tuệ nhân tạo của chúng ta sẽ biết dựa trên ô ở giữa trên

và ô ở giữa dưới. Từ ô ở giữa trên, chúng ta có $\{A, B, C\} = 1$. Từ ô ở giữa dưới, chúng ta có $\{A, B, C, D, E\} = 2$. Logic cho thấy, chúng ta có thể suy ra một kiến thức mới, là $\{D, E\} = 1$. Cuối cùng, nếu hai trong số A, B, C, D và E là mìn, và chỉ có một trong số A, B và C là mìn, thì có lý là chính xác một trong số D và E phải là mìn khác.

Nói chung, bất cứ khi nào chúng ta có hai câu set1 = count1 và set2 = count2 trong đó set1 là tập con của set2, chúng ta có thể xây dựng câu mới set2 - set1 = count2 - count1. Hãy xem xét ví dụ trên để đảm bảo bạn hiểu tại sao điều đó đúng.

Vì vậy, khi sử dụng phương pháp biểu diễn kiến thức này, chúng ta có thể viết một tác nhân trí tuệ nhân tạo có thể thu thập kiến thức về bảng Minesweeper.

```
    def mark_mine(self, cell):
        self.mines.add(cell)
        for sentence in self.knowledge:
            sentence.mark_mine(cell)

    def mark_safe(self, cell):
        self.safes.add(cell)
        for sentence in self.knowledge:
            sentence.mark_safe(cell)

    def add_knowledge(self, cell, count):
        # 1) mark the cell as a move that has been made
        self.moves_made.add(cell)

        # 2) mark the cell as safe
        self.mark_safe(cell)

        # 3) add a new sentence to the AI's knowledge base
        #    based on the value of `cell` and `count`
        cells = set()

        # Loop over all cells within one row and column
        for i in range(cell[0] - 1, cell[0] + 2):
            for j in range(cell[1] - 1, cell[1] + 2):

                # Ignore the cell itself
                if (i, j) == cell:
                    continue

                # Add to the cell collection if the cell is not yet explored
                # and is not the mine already none
                if 0 <= i < self.height and 0 <= j < self.width:
                    if (i, j) not in self.moves_made and (i, j) not in self.mines:
                        cells.add((i, j))
                    # when excluding a known mine cell, decrease the count by 1
```

```
                elif (i, j) in self.mines:
                    count -= 1
        self.knowledge.append(Sentence(cells, count))

        # 4) mark any additional cells as safe or as mines
        #    if it can be concluded based on the AI's knowledge base
        for sentence in self.knowledge:
            safes = sentence.known_safes()
            if safes:
                for cell in safes.copy():
                    self.mark_safe(cell)
            mines = sentence.known_mines()
            if mines:
                for cell in mines.copy():
                    self.mark_mine(cell)

        # 5) add any new sentences to the AI's knowledge base
        #    if they can be inferred from existing knowledge
        for sentence1 in self.knowledge:
            for sentence2 in self.knowledge:
                if sentence1 is sentence2:
                    continue
                if sentence1 == sentence2:
                    self.knowledge.remove(sentence2)
                elif sentence1.cells.issubset(sentence2.cells):
                    new_knowledge = Sentence(
                        sentence2.cells - sentence1.cells,
                        sentence2.count - sentence1.count)
                    if new_knowledge not in self.knowledge:
                        self.knowledge.append(new_knowledge)
```

5.1.4 DEMO

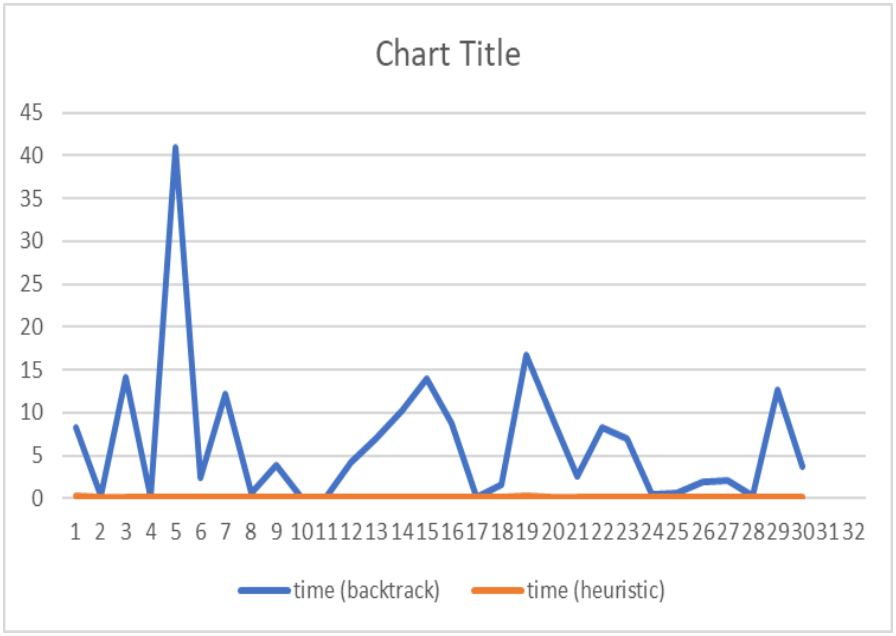
So sánh hai giải thuật

Dưới đây là bảng thống kê thời gian và sự tiêu tốn bộ nhớ của từng giải thuật:

test case	board	time (backtrack)	memory (backtr	Lost	time (heuristic)	memory (heuristic)	lost
1	[[False, False, False, False, False, False, T	8.27911	44731	0	0.247	28467	11
2	[[False, False, True, False, False, False, Fal	0.28133	44551	0	0.09673	28104	0
3	[[False, False, True, False, False, False, Fal	14.03118	43115	0	0.14081	28883	1
4	[[False, False, False, False, True, True, Fals	0.13186	42365	0	0.09737	23643	3
5	[[False, False, False, False, False, False, Fa	40.99005	43307	0	0.13516	23676	0
6	[[False, True, False, True, False, False, Fals	2.40328	44203	0	0.16228	28212	4
7	[[False, False, False, True, False, False, Fal	12.13081	43179	0	0.1104	29244	2
8	[[True, False, False, True, False, False, Fals	0.66811	43499	0	0.10596	28131	1
9	[[False, False, False, False, False, False, Fa	3.81061	44803	0	0.14808	26648	0
10	[[True, False, False, False, False, False, Fal	0.19888	43797	0	0.13088	26820	0
11	[[True, False, True, False, True, False, True]	0.18067	43069	0	0.16809	28964	1
12	[[True, False, False, False, False, False, Fal	4.10807	45059	0	0.12871	29043	1
13	[[True, False, False, False, False, True, Fals	6.87456	44803	0	0.14378	28884	1
14	[[False, False, False, False, False, False, T	10.14636	44803	0	0.17321	30701	13
15	[[False, False, True, True, True, True, True],	13.99937	44203	0	0.09794	27080	0
16	[[False, True, True, False, False, False, True	8.69676	44803	0	0.17799	27971	7
17	[[False, False, False, True, False, True, True	0.15443	43005	0	0.12503	32420	3
18	[[True, False, False, True, False, False, Fals	1.51218	44803	0	0.18407	26091	2
19	[[False, True, False, False, True, False, Fals	16.75946	43051	0	0.23781	28852	5
20	[[True, False, False, False, False, False, Fal	9.3908	44803	0	0.17766	24932	2
21	[[False, False, False, False, False, False, Fa	2.5378	44803	0	0.15115	27667	2
22	[[False, False, False, False, False, True, Fal	8.24999	45059	0	0.15912	33938	6
23	[[False, False, False, False, True, False, Fal	7.02522	45059	0	0.16788	27844	1

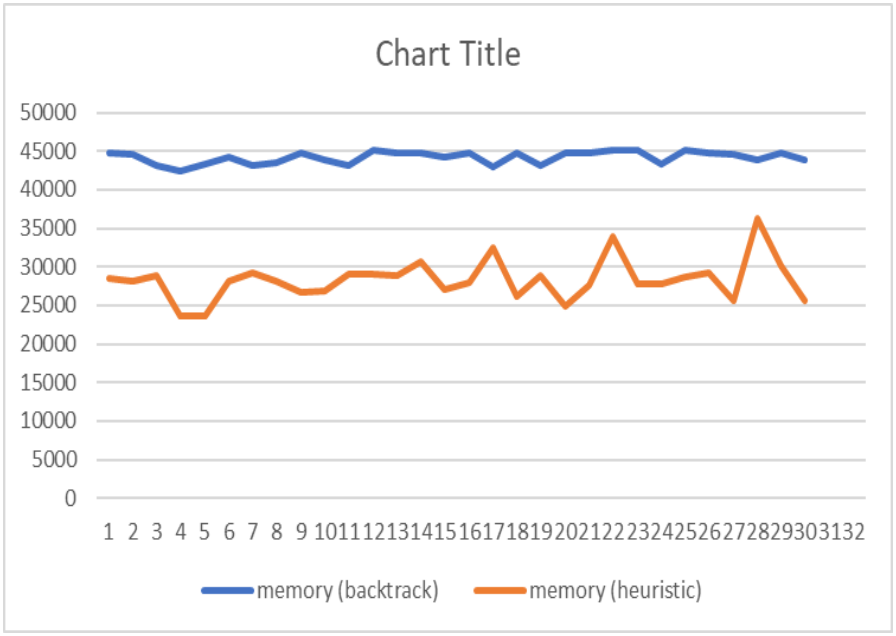
Link đầy đủ của bảng dữ liệu với 30 Input:

https://docs.google.com/spreadsheets/d/1eJ-xKbT3axfHDZTnL-rN9bXSWqk4IOs1/edit?usp=drive_link&ouid=101528424145030177469&rtpof=true&sd=true

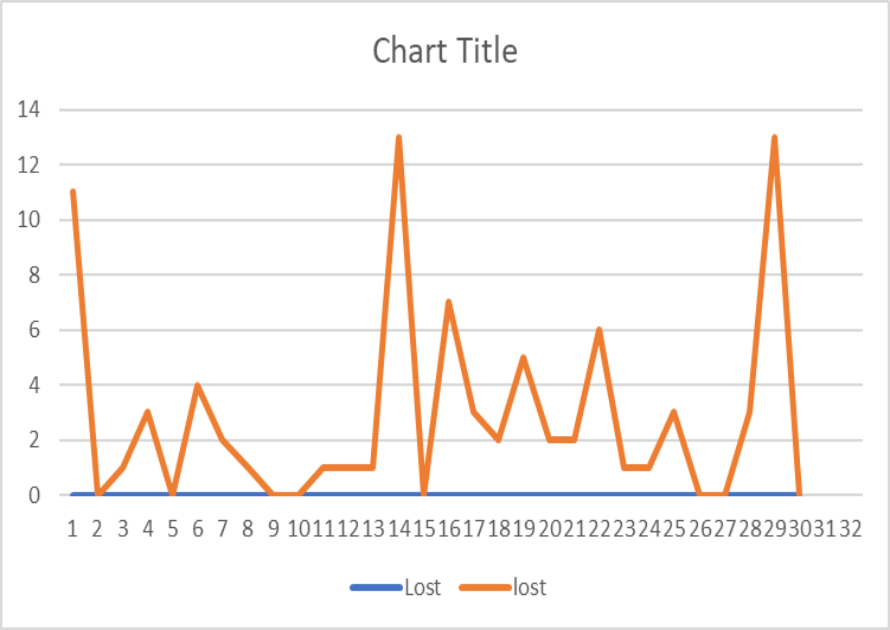


Về thời gian: Ta thấy có sự chênh lệch lớn giữa hai bên:

- Backtrack: Thời gian có sự chênh lệch lớn giữa các test case. Lí dó là vì Backtrack sẽ thử mọi trường hợp khả thi dẫn đến khi gặp trường hợp phức tạp thì Backtrack sẽ phải duyệt qua rất nhiều trường hợp sai.
- Heuristic: Thời gian có sự ổn định hơn Backtrack.



Về Bộ nhớ: Ta thấy có sự ổn định giữa các test case, bộ nhớ sử dụng của Backtrack nhiều hơn Heuristic vì Backtrack sử dụng phương pháp đệ quy nên sẽ tiêu tốn nhiều bộ nhớ hơn



Về độ chính xác: Có sự chệnh lệch lớn trong tỉ lệ thắng thua giữa hai bên:

- Backtrack: Hoàn toàn không có khả năng thua, lí do là vì Backtrack nhận đầu vào và tính toán hết tất cả các bước khả thi cho tới khi ra được kết quả rồi mới chọn.
- Heuristic: có sự chênh lệch về số lượng thua trong các test case, lí do là vì Heuristic sử dụng logic mệnh đề, vừa chọn vừa tính toán. Đặc biệt là trong lần chọn đầu tiên có khả năng chọn trúng mìn và có những trường hợp không thể tìm ra một giải pháp an toàn nên phải chọn ngẫu nhiên.

6 Repository's Link

https://github.com/dangbui03/Intro-To-AI

7 Thống kê và Video Demo's link:

https://drive.google.com/drive/folders/1EJvAQyXFe-5UCmjUVwuVAFJbjmkJSSiM?usp=drive_link

8 References:

1. "Propositional Logic." CS50 AI. https://cs50.harvard.edu/ai/2020/projects/1/minesweeper/propositional-logic (Truy cập vào ngày 22 tháng 4 năm 2024).

2. "A* search algorithm." Wikipedia. https://en.wikipedia.org/wiki/A* search algorithm (Truy cập vào ngày 22 tháng 4 năm 2024).

3. "Depth-first search." Wikipedia. https://en.wikipedia.org/wiki/Depth-first search (Truy cập vào ngày 22 tháng 4 năm 2024).

4. "Sudoku." Wikipedia tiếng Việt. https://vi.wikipedia.org/wiki/Sudoku (Truy cập vào ngày 22 tháng 4 năm 2024).

5. "Dò mìn (trò chơi)." Wikipedia tiếng Việt. https://vi.wikipedia.org/wiki/Dò mìn (trò chơi) (Truy cập vào ngày 22 tháng 4 năm 2024).

6. "Minesweeper Solver." GeeksforGeeks. https://www.geeksforgeeks.org/minesweeper-solver/ (Truy cập vào ngày 22 tháng 4 năm 2024).