# GAME SUDOKU

## 1. Giới thiệu

Trong dự án này, nhóm Chúng em đã lựa chọn bài toán Sudoku từ danh sách các bài toán Logic Puzzles để hiện thực hai giải thuật tìm kiếm:

**Blind Search (DFS – Depth-First Search):**  
Phương pháp duyệt đệ quy với kỹ thuật backtracking nhằm thử tất cả các khả năng gán số cho các ô trống trong bảng Sudoku cho đến khi tìm được lời giải hợp lệ.

**Heuristic Search (A):\***Áp dụng A\* kết hợp với heuristic MRV (Minimum Remaining Values) để ưu tiên chọn các ô có khả năng ít lựa chọn nhất, giúp giảm không gian tìm kiếm và tăng hiệu quả giải.

Chúng em cũng hiện thực giao diện trực quan (demo step-by-step) sử dụng Pygame để trực quan hóa quá trình giải bài Sudoku tự động cũng như cung cấp giao diện cho người chơi có thể sử dụng.

## 2. Luật chơi

**Quy tắc Sudoku:**  
Sudoku là trò chơi logic trên bảng 9x9 với mục tiêu điền các số từ 1 đến 9 sao cho:

* Mỗi hàng chứa đủ các số từ 1 đến 9 mà không có số lặp.
* Mỗi cột chứa đủ các số từ 1 đến 9 mà không có số lặp.
* Mỗi khối 3x3 (9 khối trên bảng) chứa đủ các số từ 1 đến 9 mà không có số lặp.

**Cách chơi trên giao diện:**

* Người chơi có thể chọn ô trống bằng cách nhấp chuột.
* Nhập số (bằng bàn phím số hoặc bàn phím số trên bàn phím phụ) để tạo ghi chú (candidate) hiển thị ở góc trái ô.
* Nhấn Enter để xác nhận số đó, nếu số nhập đúng (theo đáp án), số được hiển thị ở giữa ô (với màu xanh); nếu sai, thông báo lỗi được hiển thị.
* Bạn có thể sai tối đa 5 lần
* Game cũng cho phép chuyển động giữa các ô bằng phím mũi tên.
* Ngoài ra, có chức năng giải tự động DFS khi nhấn phím ‘d’ hoặc A\* khi nhấn phím ‘a’ trên bàn phím để trực quan hóa quá trình giải bài, và người chơi có thể dừng (quit) bất kỳ lúc nào.

## 3. Quy trình tìm hiểu và hiện thực

### 3.1. Nghiên cứu thuật toán DFS và A\*

**DFS (Backtracking):**  
Chúng em tìm hiểu cách duyệt đệ quy bảng Sudoku bằng việc tìm ô trống, thử gán số từ 1 đến 9 vào ô đó nếu thỏa mãn các ràng buộc (không trùng số trên hàng, cột, khối 3x3). Nếu bước gán không dẫn đến lời giải, thuật toán thực hiện backtracking bằng cách xóa số vừa gán.

**A với MRV:\***Ở phương pháp A\*, các trạng thái của bảng Sudoku được biểu diễn dưới dạng các cấu trúc bất biến (tuple) để lưu trữ trong dictionary. Hàm heuristic là số ô trống còn lại. Ngoài ra, MRV được sử dụng để chọn ô có số khả năng gán ít nhất trước, giúp giảm không gian tìm kiếm.

### 3.2. Hiện thực mã nguồn

Các file mã nguồn chính bao gồm:

### 3.1. File ****utils.py****

***find\_empty\_cell(board)***Duyệt qua bảng (9x9) và trả về vị trí (row, col) của ô đầu tiên có giá trị 0 (ô trống). Nếu không còn ô trống, trả về None.

***is\_valid(board, row, col, num)***Kiểm tra xem việc gán số num vào ô (row, col) có hợp lệ hay không bằng cách:

* + Duyệt hàng row và cột col để đảm bảo không có số nào trùng num.
  + Tính vị trí của khối 3x3 chứa ô đó và kiểm tra không có số nào trùng num.

***mrv\_cell(board)***Sử dụng heuristic MRV để tìm ô trống có ít lựa chọn (candidate) nhất. Hàm duyệt tất cả các ô trống và trả về một tuple (i, j, options) với options là danh sách các số có thể gán vào ô đó.

***possible\_candidates(board, row, col)***Trả về danh sách các số từ 1 đến 9 mà khi gán vào ô (row, col) không vi phạm quy tắc Sudoku (sử dụng is\_valid).

***zero\_number(board)***Đếm số ô trống (giá trị 0) trong bảng.

***is\_right\_number(board, row, col, num, goal\_board)***Kiểm tra xem số num có phải là đáp án chính xác cho ô (row, col) so với goal\_board không (kết hợp kiểm tra tính hợp lệ).

### 3.2. File ****solve\_dfs.py****

* ***solve\_dfs(board)***
  + Tìm ô trống đầu tiên bằng ***find\_empty\_cell(board).***
  + Nếu không còn ô trống, trả về True (bài đã giải).
  + Với mỗi số từ 1 đến 9, nếu ***is\_valid(board, row, col, num)*** là True, gán số đó vào ô, gọi đệ quy ***solve\_dfs(board)***.
  + Nếu đệ quy trả về True, trả về True; nếu không, thực hiện backtracking bằng cách gán lại ô thành 0 và tiếp tục thử số khác.

### 3.3. File ****solve\_astar.py****

***board\_to\_tuple(board)***Chuyển bảng Sudoku (list of lists) thành tuple các tuple để có thể dùng làm key trong dictionary (vì tuple là immutable).

***solve\_astar(initial\_board)***

* Tạo một bản sao của bảng ban đầu (initial) và chuyển nó thành tuple (initial\_tuple).
* Khởi tạo gScore và fScore:
  + - gScore: Chi phí từ trạng thái ban đầu đến trạng thái hiện tại (ban đầu là 0).
    - fScore: Ước tính chi phí từ trạng thái hiện tại đến lời giải, tính bằng gScore + zero\_number(initial).
* Sử dụng priority queue (openSet) chứa các tuple (fScore, gScore, state).
* Trong vòng lặp, lấy trạng thái có fScore thấp nhất, nếu bảng không còn ô trống (zero\_number(state)==0), cập nhật bảng ban đầu và trả về True.
* Sử dụng MRV để chọn ô có ít candidate nhất và thử gán các số hợp lệ, cập nhật gScore và fScore của trạng thái mới, và đẩy trạng thái đó vào openSet nếu chưa được duyệt.
* Nếu không tìm được lời giải, trả về False.

### 3.4. File ****gui.py****

Giao diện được xây dựng trên Pygame với các chức năng:

**Vẽ lưới:**  
Hàm ***draw\_grid()*** vẽ các đường phân chia ô, với đường dày hơn mỗi 3 ô.

**Vẽ số:**  
Hàm ***draw\_numbers(board, font, color)*** hiển thị các số trên bảng, với số ban đầu (trong initial\_board) vẽ màu đen và số người chơi nhập (sau khi xác nhận) vẽ màu xanh.

**Vẽ ghi chú:**  
Hàm ***draw\_number\_mark(number\_mark, number\_font)*** hiển thị các candidate nhập vào (chưa xác nhận) ở góc trái của ô.

**Highlight ô được chọn:**  
Hàm ***highlight\_selected(cell)*** vẽ viền đỏ cho ô được chọn.

**Các thông báo:**  
Các hàm ***display\_congratulation\_msg(), display\_wrong\_msg(), display\_lose\_msg()*** hiển thị thông báo chiến thắng, nhập sai hoặc thua.

**Giải thuật trực quan:**

* + ***solve\_dfs\_visual(initial\_board)*** cập nhật giao diện mỗi khi gán số vào ô hoặc backtracking xảy ra.
  + ***solve\_astar\_visual(initial\_board)*** sẽ tương tự, sử dụng giải thuật A\*.

**Xử lý sự kiện:**  
Giao diện cho phép chọn ô bằng chuột, nhập số bằng bàn phím, di chuyển ô bằng mũi tên, và gọi giải thuật tự động bằng phím A (A\*) hoặc D (DFS).

## Đánh giá hiệu năng dựa trên input đầu vào

**Input:**

sudoku\_board = [

    [0, 0, 0, 0, 0, 8, 0, 0, 0],

    [0, 0, 0, 1, 3, 0, 5, 9, 0],

    [0, 0, 0, 0, 2, 7, 3, 4, 0],

    [0, 2, 0, 0, 0, 3, 0, 0, 0],

    [0, 6, 9, 0, 0, 2, 0, 0, 0],

    [3, 0, 8, 5, 4, 0, 0, 2, 0],

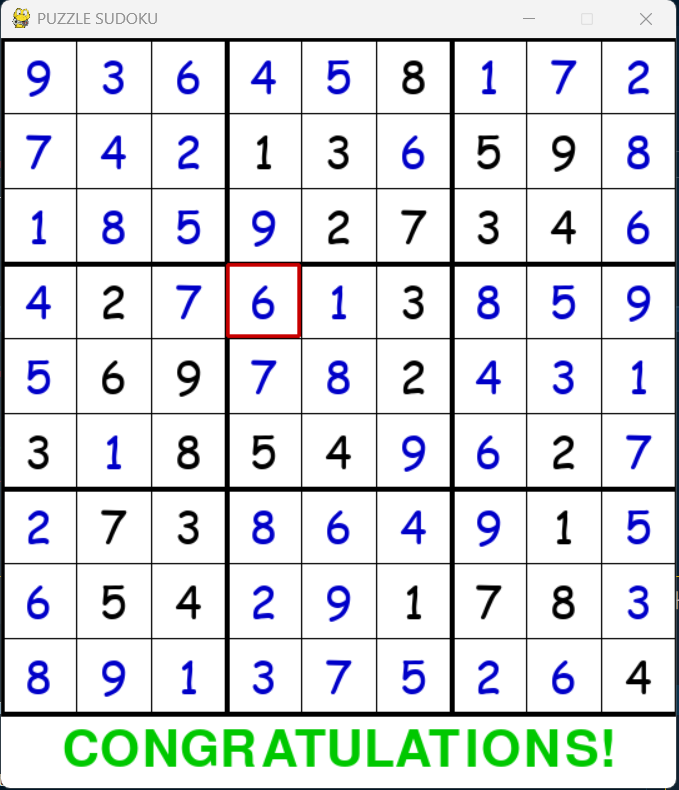
    [0, 7, 3, 0, 0, 0, 0, 1, 0],

    [0, 5, 4, 0, 0, 1, 7, 8, 0],

    [0, 0, 0, 0, 0, 0, 0, 0, 4]

]

**Output:**



### 4.1. Bảng số liệu thời gian và bộ nhớ tiêu tốn

|  |  |  |  |
| --- | --- | --- | --- |
| **Thuật toán** | **Thời gian chạy (s)** | **Bộ nhớ sử dụng (KB)** | **Kết luận** |
| DFS | 16 - 17 | 2 - 3 | Phù hợp với Sudoku 9x9, nhưng có thể chậm với input phức tạp hơn. |
| A\* | 0.15 - 0.16 | 17 – 18 | Hiệu quả hơn DFS về thời gian, nhưng bộ nhớ tiêu thụ cao hơn do lưu trữ các trạng thái. |

Lưu ý: Các số liệu trên là ước tính dựa trên các bài kiểm tra với input Sudoku 9x9. Khi tăng kích thước bài toán hoặc input phức tạp, hiệu năng của từng thuật toán sẽ thay đổi.

### 4.2. Phân tích độ phức tạp

**DFS (Backtracking):**

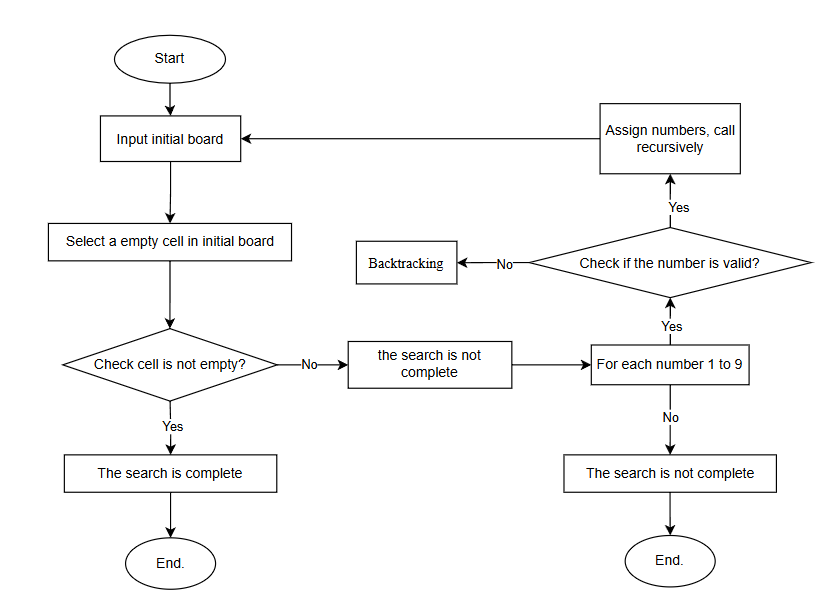
* + **Time Complexity:**  
    Trong trường hợp xấu nhất, DFS có độ phức tạp là O(9^(n)) với n là số ô trống. Tuy nhiên, đối với Sudoku 9x9, các ràng buộc giúp giảm đáng kể số trường hợp cần duyệt.
  + **Space Complexity:**  
    Độ phức tạp không gian chủ yếu là O(n) cho đệ quy (với n là số ô trống), khá hạn chế với Sudoku 9x9.

**A với MRV:**

* + **Time Complexity:**  
    Thời gian chạy phụ thuộc vào chất lượng của hàm heuristic. Với hàm heuristic (số ô trống) đơn giản, độ phức tạp không thể tránh khỏi trường hợp xấu nhất là O(9^(n)), nhưng thực tế A\* tìm được lời giải nhanh hơn DFS nhờ MRV lựa chọn ô có ít lựa chọn nhất.
  + **Space Complexity:**  
    A\* tiêu tốn nhiều không gian hơn DFS do phải lưu trữ các trạng thái mở (openSet) và đóng (closeSet). Độ phức tạp không gian là O(b^d) trong trường hợp xấu nhất, với b là bậc của cây tìm kiếm và d là độ sâu lời giải.

## 5. Diagram

### 5.1. Diagram luồng thuật toán DFS



### 5.2. Diagram luồng thuật toán A\*

## 6. Kết luận

**DFS**:  
Mặc dù thuật toán DFS với backtracking có độ phức tạp thời gian xấu nhất rất cao, nhưng nhờ các ràng buộc của Sudoku, DFS có thể giải bài toán 9x9 một cách hiệu quả. Ưu điểm của DFS là triển khai đơn giản và tiêu thụ không gian bộ nhớ hạn chế.

**A**\*:  
Thuật toán A\* với heuristic MRV cho thấy hiệu quả vượt trội về thời gian nhờ khả năng ưu tiên các trạng thái tiềm năng. Tuy nhiên, nó tiêu tốn bộ nhớ cao hơn so với DFS do lưu trữ các trạng thái mở và đóng.

**Trực quan hóa**:  
Việc hiện thực tính năng demo step-by-step bằng Pygame giúp người dùng hiểu được quá trình giải thuật (các bước gán số và backtracking) một cách trực quan, đồng thời hỗ trợ việc kiểm tra và debug dễ dàng.

Trong báo cáo này, Chúng em đã trình bày chi tiết quá trình tìm hiểu, hiện thực và đánh giá hiệu năng của hai giải thuật DFS và A\* cho bài toán Sudoku. Các số liệu về thời gian và bộ nhớ cũng đã được thu thập và phân tích để so sánh ưu, nhược điểm của từng phương pháp.

## 7. Tài liệu tham khảo

1. [Sudoku #1: Depth-First Search - Blog](https://kevinkle.in/posts/2021-02-28-sudoku_dfs/)

2.[techwithtim.net/tutorials/python-programming/sudoku-solver-backtracking](https://www.techwithtim.net/tutorials/python-programming/sudoku-solver-backtracking)

3.[Solving Sudoku with A\* | Matej++](http://blog.matejzavrsnik.com/solving_sudoku_with_a.html)

4.[sudoku-solver/sudokuSolver.py at main · erickfunier/sudoku-solver](https://github.com/erickfunier/sudoku-solver/blob/main/sudokuSolver.py)

5.[A\* search algorithm - Wikipedia](https://en.wikipedia.org/wiki/A*_search_algorithm)

6.[Depth-first search - Wikipedia](https://en.wikipedia.org/wiki/Depth-first_search)

7.[Sudoku - online puzzle game](https://www.puzzle-sudoku.com/)

8.[Building and visualizing Sudoku Game Using Pygame - GeeksforGeeks](https://www.geeksforgeeks.org/building-and-visualizing-sudoku-game-using-pygame/)

9.[How to Build a Sudoku Game with Python - The Python Code](https://thepythoncode.com/article/make-a-sudoku-game-in-python)