

Báo cáo BTL 1: Trò chơi Shikaku

1. Giới Thiệu trò chơi Shikaku

Shikaku là một trò chơi giải đố logic xuất xứ từ Nhật Bản, còn được gọi là "Những hình chữ nhật". Mục tiêu của trò chơi là chia lưới ô vuông thành các hình chữ nhật hoặc hình vuông sao cho mỗi vùng chứa đúng một số, và số đó biểu thị diện tích của vùng đó.

Luật chơi:

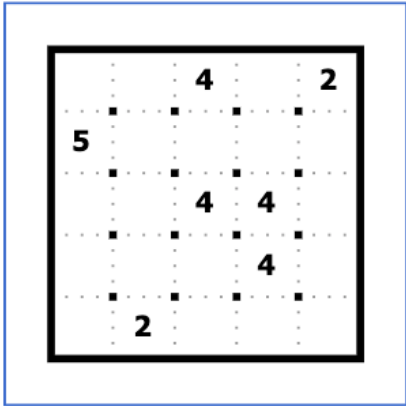
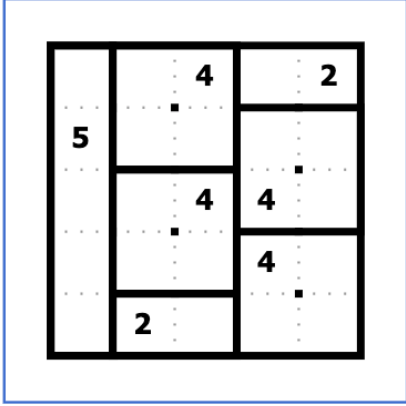
- Phân chia lưới:** Chia lưới thành các hình chữ nhật hoặc hình vuông.
- Số trong mỗi vùng:** Mỗi vùng phải chứa chính xác một số.
- Diện tích vùng:** Diện tích (số ô) của mỗi vùng phải bằng số nằm trong vùng đó.

Cách chơi:

- Vẽ hình chữ nhật:** Nhấn và giữ chuột trái tại một ô, sau đó kéo để tạo hình chữ nhật mong muốn.
- Xóa hình chữ nhật:** Nhấn và giữ chuột phải, hoặc giữ phím Ctrl/Shift và nhấn chuột trái, sau đó kéo để xóa vùng đã vẽ.

Chiến lược giải đố:

- Bắt đầu từ các số nhỏ:** Các số nhỏ thường có ít khả năng phân chia, giúp xác định nhanh chóng các vùng tương ứng.
- Sử dụng loại trừ:** Dựa vào vị trí của các số và kích thước lưới, loại trừ các khả năng không hợp lý để thu hẹp phạm vi lựa chọn.
- Quan sát các số gần nhau:** Xem xét mối quan hệ giữa các số liền kề để xác định ranh giới giữa các vùng.

	
Câu đố Shikaku 5x5 (Ban đầu)	Lời giải cho Shikaku 5x5

2. Mô hình bài toán

Cho một lưới kích thước $R \times C$, trong đó một số ô chứa số nguyên dương n . Bài toán yêu cầu phân chia lưới thành các hình chữ nhật sao cho:

- Mỗi hình chữ nhật chứa chính xác một số duy nhất.
- Diện tích của mỗi hình chữ nhật phải bằng giá trị số trong ô đó.
- Các hình chữ nhật không được chồng lên nhau.
- Toàn bộ lưới phải được bao phủ hoàn toàn, không bỏ sót ô nào.

Các bước giải quyết bài toán

Bước 1: Đọc và khởi tạo dữ liệu đầu vào

- Nhập số hàng R và số cột C .
- Xây dựng ma trận ban đầu, trong đó các ô chứa số được đánh dấu, các ô còn lại để trống.

Bước 2: Xác định các cách chia hợp lệ cho từng số

- Với mỗi số n trong lưới, tìm tất cả các cặp (h, w) thỏa mãn: $h \times w = n$
 - Trong đó, h là số hàng và w là số cột của hình chữ nhật.
 - Các cặp này xác định tất cả các cách có thể đặt hình chữ nhật xung quanh số n .

Bước 3: Tìm kiếm lời giải bằng DFS hoặc BFS

- Sử dụng thuật toán **tìm kiếm theo chiều sâu (DFS)** hoặc **tìm kiếm theo chiều rộng (BFS)** để thử từng cách đặt hình chữ nhật.
- Với mỗi cách chia:
 - Kiểm tra xem hình chữ nhật có nằm trong phạm vi lưới và không chồng lên các vùng đã chọn trước đó.
 - Nếu hợp lệ, đặt hình chữ nhật và tiếp tục thử cho số tiếp theo.
 - Nếu gặp trường hợp không thể chia hợp lệ, thực hiện **backtracking (quay lui)** để thử các phương án khác.

Bước 4: Xác thực lời giải

Sau khi tìm được một cách chia, kiểm tra lại các điều kiện sau:

- Toàn bộ lưới phải được bao phủ bởi đúng một hình chữ nhật tại mỗi ô.
- Không có hai hình chữ nhật nào chồng lên nhau.
- Mỗi vùng chứa đúng một số và diện tích đúng bằng giá trị số đó.

Bước 5: Tối ưu hóa thuật toán

Để giảm không gian tìm kiếm và cải thiện hiệu suất, có thể áp dụng các chiến lược tối ưu sau:

- **Ưu tiên xử lý các số lớn trước:** Vì số lớn có ít cách chia hơn, nên xác định trước sẽ giúp giảm số nhánh cần thử nghiệm.
- **Cắt giảm không gian tìm kiếm:** Nếu một số chỉ có duy nhất một hình chữ nhật hợp lệ, đặt nó ngay lập tức để giảm số lựa chọn cần thử.
- **Loại bỏ phương án không hợp lệ sớm:** Nếu một hình chữ nhật không thể được đặt hợp lệ, loại bỏ ngay thay vì tiếp tục kiểm tra các bước sau.

Độ phức tạp tính toán

- Trong trường hợp xấu nhất, thuật toán có thể phải thử tất cả các cách chia hình chữ nhật, dẫn đến độ phức tạp **xấp xỉ $O(2^N)$** .
- Tuy nhiên, bằng cách áp dụng các chiến lược tối ưu hóa, độ phức tạp có thể giảm đáng kể trong các trường hợp thực tế.

Kết luận

Bài toán Shikaku đòi hỏi sự kết hợp giữa các kỹ thuật **tìm kiếm**, **backtracking**, và **heuristic** để tìm lời giải tối ưu. Việc áp dụng các chiến lược tối ưu giúp giảm số lần thử nghiệm và cải thiện hiệu suất thuật toán, giúp giải quyết bài toán một cách hiệu quả hơn.

3. Trình bày mã nguồn và giải thích

Bước 1: Khởi tạo và đọc dữ liệu

```
from typing import List, Tuple, Set
from dataclasses import dataclass
import glob
import time
import tracemalloc
from math import sqrt
from collections import defaultdict

@dataclass
Đức Anh Mai, 3 days ago | 1 author (Đức Anh Mai)
class PuzzleData:
    rows: int
    cols: int
    puzzle: List[List[int]]
    location_data: List[Tuple[int, int, int]]
    state: List[List[int]]
    factors_cache: List[List[int]]
    count: List[int]
    last_cells: List[List[Tuple[int, int]]]

Đức Anh Mai, 3 days ago | 1 author (Đức Anh Mai)
class PuzzleSolver:
    def __init__(self):
        self.data = None

    def read_puzzle(self, input_filename: str) -> None:
        """Read puzzle data from file and initialize puzzle structure."""
        with open(input_filename, "r") as input_file:
            rows = int(input_file.readline())
            cols = int(input_file.readline())
            puzzle = [[-1] * cols for _ in range(rows)]
            location_data = []

            for row, line in enumerate(input_file):
                for col, symbol in enumerate(line.split()):
                    value = -1 if symbol == "-" else int(symbol)
                    puzzle[row][col] = value
                    if value != -1:
                        location_data.append((row, col, value))

            self.data = PuzzleData(
                rows=rows,
                cols=cols,
                puzzle=puzzle,
                location_data=location_data,
                state=[[-1] * cols for _ in range(rows)],
                factors_cache=[],
                count=[0] * len(location_data),
                last_cells=[[] for _ in range(len(location_data))]
            )
```

Thư viện và kiểu dữ liệu:

- typing: Sử dụng để định nghĩa các kiểu danh sách (List) và bộ giá trị (Tuple).
- dataclasses: Cung cấp decorator @dataclass để tạo lớp dữ liệu đơn giản.

- glob, time, tracemalloc: Các thư viện hỗ trợ thao tác với tệp, đo thời gian và theo dõi bộ nhớ.
- sqrt từ math: Dùng để tính căn bậc hai.

Lớp PuzzleData:

- Chứa các thuộc tính mô tả trạng thái của trò chơi, bao gồm kích thước lưới (rows, cols), ma trận ban đầu (puzzle), vị trí và giá trị của các số (location_data), trạng thái hiện tại của lưới (state), bộ nhớ đệm cho các ước số (factors_cache), bộ đếm (count) và danh sách các ô cuối cùng của mỗi vùng (last_cells).

Lớp PuzzleSolver:

- Phương thức __init__: Khởi tạo đối tượng với thuộc tính data ban đầu là None.
- Phương thức read_puzzle: Đọc dữ liệu từ tệp đầu vào, khởi tạo ma trận puzzle và thu thập thông tin về các ô chứa số vào location_data. Các ô trống được đánh dấu bằng -1.

Bước 2: Xử lý và chuẩn bị dữ liệu

```
@staticmethod
def calculate_factors(n: int) -> List[int]:
    """Calculate factors of a number more efficiently."""
    factors = set()
    for i in range(1, int(sqrt(n)) + 1):
        if n % i == 0:
            factors.add(i)
            factors.add(n // i)
    return sorted(list(factors))

def initialize_factors(self) -> None:
    """Initialize factors for all values in location data."""
    self.data.factors_cache = [
        self.calculate_factors(value)
        for _, _, value in self.data.location_data
    ]
```

Phương thức calculate_factors:

- Tính tất cả các ước số của một số n bằng cách duyệt từ 1 đến căn bậc hai của n. Các ước số được lưu trong một tập hợp (set) để loại bỏ trùng lặp, sau đó chuyển thành danh sách và sắp xếp tăng dần.

Phương thức initialize_factors:

- Khởi tạo `factors_cache` bằng cách tính các ước số cho từng giá trị trong `location_data`. Mỗi giá trị đại diện cho diện tích của một hình chữ nhật cần tìm.

Bước 3: Kiểm tra hợp lệ đầu vào

```
@staticmethod
def is_valid_bounds(L: List[List[int]], r1: int, r2: int, c1: int, c2: int) -> bool:
    """Check if the given bounds are valid."""
    if r1 > r2 or c1 > c2 or r1 < 0 or c1 < 0:
        return False
    return r2 < len(L) and c2 < len(L[0])

@staticmethod
def is_valid_region(L: List[List[int]], r1: int, r2: int, c1: int, c2: int,
                    value: int, value2: int) -> bool:
    """Check if the region contains only valid values."""
    return all(L[r][c] in (value, value2)
               for r in range(r1, r2 + 1)
               for c in range(c1, c2 + 1))

@staticmethod
def set_region_value(L: List[List[int]], r1: int, r2: int, c1: int, c2: int,
                     value: int) -> None:
    """Set a value for all cells in the given region."""
    for r in range(r1, r2 + 1):
        for c in range(c1, c2 + 1):
            L[r][c] = value
```

Phương thức `is_valid_bounds`:

- Kiểm tra xem tọa độ của hình chữ nhật (từ $(r1, c1)$ đến $(r2, c2)$) có nằm trong giới hạn của lưới `L` hay không.

Phương thức `is_valid_region`:

- Xác định xem tất cả các ô trong vùng từ $(r1, c1)$ đến $(r2, c2)$ có giá trị bằng `value` hoặc `value2` hay không. Điều này đảm bảo vùng đang xét chưa được sử dụng hoặc chứa giá trị mong muốn.

Phương thức `set_region_value`:

- Đặt giá trị `value` cho tất cả các ô trong vùng từ $(r1, c1)$ đến $(r2, c2)$.

Bước 4: Giải thuật tìm kiếm DFS

```
def verify_solution(self) -> bool:
    """Verify if the current state is a valid solution."""
    for i, (row, col, val) in enumerate(self.data.location_data):
        if self.data.state[row][col] != i:
            return False

        cells = [(r, c) for r in range(self.data.rows)
                 for c in range(self.data.cols)
                 if self.data.state[r][c] == i]

        if len(cells) != val:
            return False

        r_min = min(r for r, _ in cells)
        r_max = max(r for r, _ in cells)
        c_min = min(c for _, c in cells)
        c_max = max(c for _, c in cells)

        if (r_max - r_min + 1) * (c_max - c_min + 1) != len(cells):
            return False

    return True

def dfs(self, next_index: int) -> bool:
    """Depth-first search to solve the puzzle."""
    if next_index >= len(self.data.location_data):
        return True

    e_row, e_col, e_value = self.data.location_data[next_index]

    while self.data.count[next_index] < len(self.data.factors_cache[next_index]):
        fac = self.data.factors_cache[next_index][self.data.count[next_index]]

        for i in range(e_value // fac):
            for j in range(fac):
                r1, r2 = e_row - j, e_row + fac - 1 - j
                c1, c2 = e_col + i - e_value // fac + 1, e_col + i

                if (self.is_valid_bounds(self.data.state, r1, r2, c1, c2) and
                    self.is_valid_region(self.data.state, r1, r2, c1, c2, -1, next_index)):

                    self.set_region_value(self.data.state, r1, r2, c1, c2, next_index)

                    if all(self.data.state[r][c] != -1
                           for r, c in self.data.last_cells[next_index]):
                        if self.dfs(next_index + 1):
                            return True

                    self.set_region_value(self.data.state, r1, r2, c1, c2, -1)
                    self.data.state[e_row][e_col] = next_index

            self.data.count[next_index] += 1

    if next_index > 0:
        self.data.count[next_index] = 0
    return False
```

Phương thức `verify_solution`:

- **Mục đích:** Xác minh xem trạng thái hiện tại của lưới (`self.data.state`) có phải là một giải pháp hợp lệ cho bài toán Shikaku hay không.
- **Quy trình:**
 1. **Kiểm tra vị trí số ban đầu:** Đảm bảo rằng mỗi ô chứa số ban đầu (`location_data`) được gán đúng chỉ số vùng tương ứng trong `state`.
 2. **Thu thập các ô thuộc cùng một vùng:** Dựa trên chỉ số vùng (`i`), thu thập tất cả các ô trong `state` có giá trị bằng `i`.
 3. **Kiểm tra số lượng ô:** Đảm bảo rằng số lượng ô trong vùng bằng với giá trị số ban đầu (`val`).
 4. **Xác minh hình dạng hình chữ nhật:** Tính toán tọa độ tối thiểu và tối đa theo hàng (`r_min`, `r_max`) và cột (`c_min`, `c_max`). Kiểm tra xem số ô trong vùng có bằng diện tích hình chữ nhật được xác định bởi các tọa độ này hay không.
- **Kết luận:** Nếu tất cả các kiểm tra trên đều thỏa mãn, phương thức trả về `True`, ngược lại trả về `False`.

Phương thức `dfs`:

- **Mục đích:** Sử dụng thuật toán tìm kiếm theo chiều sâu (Depth-First Search - DFS) để tìm cách phân chia lưới thành các hình chữ nhật thỏa mãn điều kiện của bài toán.
- **Tham số:** `next_index` đại diện cho chỉ số của số hiện tại trong `location_data` cần được xử lý.
- **Quy trình:**
 1. **Điều kiện dừng:** Nếu `next_index` lớn hơn hoặc bằng độ dài của `location_data`, nghĩa là tất cả các số đã được xử lý, trả về `True`.
 2. **Lấy thông tin số hiện tại:** Tọa độ hàng (`e_row`), cột (`e_col`) và giá trị (`e_value`) của số tại `next_index`.
 3. **Duyệt qua các ước số khả thi:** Sử dụng `factors_cache` để lấy các ước số của `e_value`. Mỗi ước số đại diện cho một chiều (chiều cao hoặc chiều rộng) của hình chữ nhật.
 4. **Thử các vị trí đặt hình chữ nhật:** Với mỗi cặp chiều cao và chiều rộng, tính toán tọa độ góc trên trái (`r1`, `c1`) và góc dưới phải (`r2`, `c2`) của hình chữ nhật.
 5. **Kiểm tra tính hợp lệ:** Sử dụng `is_valid_bounds` để đảm bảo hình chữ nhật nằm trong lưới và `is_valid_region` để đảm bảo vùng này chưa được sử dụng.
 6. **Đặt hình chữ nhật:** Nếu hợp lệ, cập nhật `state` với chỉ số `next_index` cho các ô trong vùng.
 7. **Đệ quy xử lý số tiếp theo:** Gọi đệ quy `dfs` với `next_index + 1`. Nếu tìm được giải pháp (`True`), kết thúc.
 8. **Quay lui:** Nếu không tìm được giải pháp, đặt lại các ô trong vùng về -1 và thử các cấu hình khác.
- **Kết luận:** Nếu không có cấu hình nào hợp lệ cho `next_index`, trả về `False`.

Bước 5: Khởi tạo và tối ưu hoá

```
def initialize(self) -> None:
    """Initialize the puzzle solver state."""
    self.initialize_factors()

    # Set initial state
    for i, (row, col, _) in enumerate(self.data.location_data):
        self.data.state[row][col] = i

    # Calculate last cells
    temp_state = [[-1] * self.data.cols for _ in range(self.data.rows)]
    for i, (row, col, _) in enumerate(self.data.location_data):
        temp_state[row][col] = i

    # Pre-calculate possible positions
    for z, (e_row, e_col, e_value) in enumerate(self.data.location_data):
        for fac in self.data.factors_cache[z]:
            for i in range(e_value // fac):
                for j in range(fac):
                    r1, r2 = e_row - j, e_row + fac - 1 - j
                    c1, c2 = e_col + i - e_value // fac + 1, e_col + i

                    if self.is_valid_bounds(temp_state, r1, r2, c1, c2):
                        self.set_region_value(temp_state, r1, r2, c1, c2, z)

    # Store last cells
    for row in range(self.data.rows):
        for col in range(self.data.cols):
            value = temp_state[row][col]
            if value != -1:
                self.data.last_cells[value].append((row, col))

    # Reset state
    self.data.state = [[-1] * self.data.cols for _ in range(self.data.rows)]
    for i, (row, col, _) in enumerate(self.data.location_data):
        self.data.state[row][col] = i
```

Phương thức initialize:

- **Mục đích:** Chuẩn bị và khởi tạo các cấu trúc dữ liệu cần thiết trước khi bắt đầu quá trình tìm kiếm giải pháp.
- **Quy trình:**
 1. **Khởi tạo các ước số:** Gọi phương thức `initialize_factors` để tính toán và lưu trữ các ước số cho mỗi giá trị trong `location_data`.
 2. **Đánh dấu vị trí các số ban đầu:** Duyệt qua `location_data` và gán chỉ số tương ứng cho các ô chứa số trong `state`.
 3. **Tạo trạng thái tạm thời (temp_state):** Sao chép trạng thái hiện tại của lưới để sử dụng trong quá trình xác định các ô cuối cùng của mỗi vùng.

4. **Xác định các vùng khả thi:** Dựa trên các ước số đã tính, xác định tất cả các hình chữ nhật có thể chứa mỗi số và đánh dấu chúng trong temp_state.
5. **Lưu trữ các ô cuối cùng:** Duyệt qua temp_state để xác định và lưu trữ các ô cuối cùng của mỗi vùng vào last_cells.
6. **Đặt lại trạng thái ban đầu:** Khởi tạo lại state với giá trị -1 cho tất cả các ô và đánh dấu lại các ô chứa số ban đầu với chỉ số tương ứng.

Bước 6: Giải và hiển thị kết quả

```
def solve(self, filename: str) -> Tuple[bool, float, float]:
    """Solve the puzzle and return solution status and time taken."""
    self.read_puzzle(filename)

    print("\nStart Grid:")
    self.print_grid(self.data.puzzle)

    tracemalloc.start()
    start_time = time.time()
    self.initialize()

    success = self.dfs(0)
    end_time = time.time()
    mem_usage = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    return success, end_time - start_time, mem_usage[1] / 1024

    @staticmethod
    def print_grid(grid: List[List[int]]) -> None:
        """Print the grid in a formatted manner."""
        for row in grid:
            print(''.join(f"{str(symbol):>4}" for symbol in row))
```

Phương thức solve:

- **Mục đích:** Đọc dữ liệu từ tệp đầu vào, khởi tạo các cấu trúc dữ liệu, thực hiện giải bài toán và đo lường hiệu suất (thời gian và bộ nhớ).
- **Quy trình:**
 1. **Đọc dữ liệu:** Gọi phương thức read_puzzle để đọc và khởi tạo lưới từ tệp đầu vào.
 2. **Hiển thị lưới ban đầu:** Sử dụng print_grid để in ra lưới ban đầu.
 3. **Khởi động theo dõi bộ nhớ:** Sử dụng tracemalloc.start() để bắt đầu theo dõi việc sử dụng bộ nhớ.
 4. **Ghi lại thời gian bắt đầu:** Lấy thời gian hiện tại bằng time.time().
 5. **Khởi tạo cấu trúc dữ liệu:** Gọi phương thức initialize để chuẩn bị cho quá trình tìm kiếm.

6. **Thực hiện tìm kiếm giải pháp:** Gọi phương thức dfs bắt đầu từ chỉ số 0 để tìm giải pháp.
7. **Ghi lại thời gian kết thúc:** Lấy thời gian sau khi hoàn thành việc tìm kiếm.
8. **Lấy thông tin bộ nhớ:** Sử dụng tracemalloc.get_traced_memory() để lấy thông tin về việc sử dụng bộ nhớ.
9. **Dừng theo dõi bộ nhớ:** Gọi tracemalloc.stop() để kết thúc việc theo dõi.
10. **Trả về kết quả:** Bao gồm trạng thái thành công (success), thời gian thực hiện và bộ nhớ đã sử dụng (tính bằng KB).

Phương thức print_grid:

- **Mục đích:** In ra lưới dưới dạng có định dạng, giúp dễ dàng quan sát trạng thái hiện tại của lưới.
- **Quy trình:** Duyệt qua từng hàng của lưới và in ra các giá trị, mỗi giá trị được căn phải với độ rộng 4 ký tự để đảm bảo căn chỉnh đẹp mắt.

Bước 7: Thực thi chương trình

```
def main():
    solver = PuzzleSolver()
    filenames = sorted(glob.glob("inputSubmit/001.txt"))

    if not filenames:
        print("No input files found in inputSubmit/")
        return

    for filename in filenames:
        print(f"\nProcessing file: {filename}")
        success, time_taken, memory_usage = solver.solve(filename)

        print("\nFinal Grid:")
        solver.print_grid(solver.data.state)
        print(f"\nStatus: {'Solved' if success else 'Not Solved'}")
        print(f"Time taken: {time_taken:.3f} seconds")
        print(f"Memory used: {memory_usage:.2f} KB")
        print("=" * 40)

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

Hàm main:

- **Mục đích:** Đây là điểm bắt đầu của chương trình, chịu trách nhiệm khởi tạo bộ giải, đọc các tệp đầu vào, giải quyết từng bài toán và hiển thị kết quả.

- **Quy trình:**

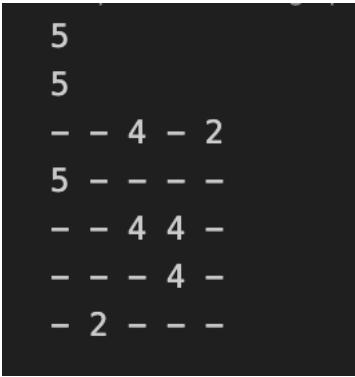
1. **Khởi tạo PuzzleSolver:** Tạo một instance của lớp PuzzleSolver để sử dụng cho việc giải quyết bài toán.
2. **Tìm kiếm tệp đầu vào:** Sử dụng glob.glob để tìm các tệp trong thư mục inputSubmit/ với định dạng tên cụ thể (ở đây là 001.txt). Kết quả được sắp xếp và lưu vào danh sách filenames.
3. **Kiểm tra sự tồn tại của tệp:** Nếu không tìm thấy tệp nào, in thông báo và kết thúc chương trình.
4. **Xử lý từng tệp:** Duyệt qua danh sách filenames:
 - In ra tên tệp đang xử lý.
 - Gọi phương thức solve của solver với tên tệp hiện tại để giải bài toán. Phương thức này trả về:
 - success: Trạng thái giải quyết (thành công hoặc thất bại).
 - time_taken: Thời gian thực hiện giải quyết bài toán.
 - memory_usage: Lượng bộ nhớ đã sử dụng trong quá trình giải.
 - In ra lưới kết quả cuối cùng bằng cách gọi print_grid.
 - In trạng thái giải quyết (Solved hoặc Not Solved), thời gian thực hiện và bộ nhớ đã sử dụng.
 - In một dòng phân cách để dễ dàng đọc kết quả.

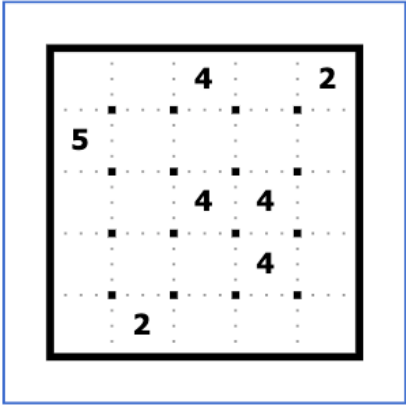
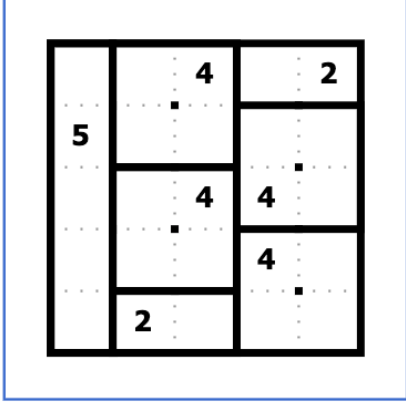
Điểm bắt đầu của chương trình:

- Sử dụng cấu trúc if `__name__ == "__main__"`: để đảm bảo rằng hàm main chỉ được gọi khi tệp này được chạy trực tiếp, không phải khi nó được nhập như một module trong tệp khác.

4. Demo

Sử dụng luôn câu đố ở phần giới thiệu Shikaku. Đây là file input/001.txt

	<p>Thông số:</p> <ul style="list-style-type: none">+ Số hàng: 5+ Số cột: 5 <p>Yêu cầu:</p> <ul style="list-style-type: none">+ Có 2 ô chứa số 2 => Cần tạo các hình chữ nhật có diện tích 2 ô ở vị trí đó+ Có 4 ô chứa số 4 => Cần tạo các hình chữ nhật có diện tích 4 ô ở vị trí đó+ Có 1 ô chứa số 5 => Cần tạo một hình chữ nhật có diện tích 5 ô ở vị trí đó
---	---

<p>Câu đố Shikaku 5x5</p>		<p>Start Grid:</p> <pre> -1 -1 4 -1 2 5 -1 -1 -1 -1 -1 -1 4 4 -1 -1 -1 -1 4 -1 -1 2 -1 -1 -1 </pre>
<p>Lời giải cho Shikaku 5x5</p>		<p>Final Grid:</p> <pre> 2 0 0 1 1 2 0 0 4 4 2 3 3 4 4 2 3 3 5 5 2 6 6 5 5 </pre>
<p>Thời gian và bộ nhớ</p>		<p>Time taken: 0.001 seconds Memory used: 3.30 KB</p>

Link github: <https://github.com/anhmai2605/TTNT-242.git>

Link video: