KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
227 Nguyễn Văn Cừ, Phường 4, Quận 5, TP.HCM
Điện thoại: (08) 38.354.266 – Fax: (08) 38.350.096

Cơ sở trí tuệ nhân tạo

Báo cáo lab 2

Gem hunter

Giảng viên hướng dẫn:

Thầy Bùi Duy Đăng

Thầy Nguyễn Thanh Tình

Họ và tên : Nguyễn Thanh Phong

MSSV: 22120265

Học kỳ: 2

Năm học : 2024-2025

Thành phố Hồ Chí Minh, tháng 05, năm 2025

Mục lục

I. Bài toán

1. Mô tả bài toán

Gem hunter là bài toán lập trình trong lĩnh vực trí tuệ nhân tạo, yêu cầu phát triển trò chơi tìm kiếm đá quý trên lưới ô vuông sử dụng logic dựa trên Conjunctive Normal Form – CNF. Trong trò chơi, người chơi sẽ khám phá một lưới ô vuông để tìm kiếm các viên đá quý ẩn (G) trong khi tránh các bẫy (T). Một số ô trên lưới ô vuông chứa một số nguyên từ 1 – 8, biểu thị số lượng bẫy xung quanh ô đó. Nhiệm vụ chính là xây dựng các ràng buộc CNF, từ đó áp dụng thư viện PySAT hoặc thuật toán brute-force, backtracking để xác định vị trí đá quý và bẫy.

2. Mục tiêu

- Xây dựng các ràng buộc CNF dựa trên thông tin số bẫy xung quanh một ô.
- Tự động sinh CNF từ lưới đầu vào.
- Sử dụng thư viện PySAT để tìm lời giải.
- Cài đặt thuật toán brute-force, backtracking, so sánh hiệu suất (thời gian thực thi) của chúng so với sử dụng thư viện PySAT.
- Phân tích và đánh giá hiệu quả của các phương pháp qua các test case với lưới ô vuông các kích thước khác nhau (5x5, 11x11, 20x20).

II. Xử lý sinh CNF

1. Nguyên lý xây dựng ràng buộc

Lưới ô vuông được biểu diễn dưới dạng các ràng buộc logic trong dạng chuẩn tắc kết hợp (CNF). Mỗi ô trên lưới được gán một biến logic: True nếu ô đó chứa bẫy, False nếu ô đó chứa đá quý. Các ràng buộc CNF được xây dựng dựa trên các thông tin sau:

- Ô chứa số: chứa số tự nhiên k (từ 1 – 8) biểu thị có đúng k ô chứa bẫy trong các ô xung quanh (Exactly k True):

$$Exactly\ K\ true\ \Leftrightarrow (No\ more\ than\ K\ \wedge At\ least\ K)$$

- Ô chứa bẫy (T): Biến logic tương ứng với ô này nhận giá trị True.
- Ô chứa đá quý (G): Biến logic tương ứng với ô này nhận giá trị False.

2. Các bước xây dựng ràng buộc CNF

Quá trình xây dựng CNF bao gồm các bước sau:

- Gán một biến logic duy nhất cho mỗi ô (i, j) trên lưới: sử dụng hàm get_var_id(i, j, width) từ utils.py để trả về một số nguyên duy nhất.
- Xác định các ô lân cận bằng hàm get_neighbors(i, j, height, width) từ utils.py để trả về danh sách các tọa độ của 8 ô xung quanh.
- Tạo các mệnh đề CNF dựa trên giá trị của ô (số, T, G, _) và thông tin từ các ô lân cận.

3. Sinh CNF tự động

a. Gán biến logic cho mỗi ô

- Mỗi ô trên lưới được gán một biến logic $var\_id_{i,j}$, trong đó (i, j) là tọa độ của ô trên lưới ô vuông.
- Với lưới có kích thước mxn thì có tổng cộng m.n biến logic.

- Ví dụ: Lưới có kích thước 3x3 ta có 3.3 = 9 ô ⇒ có 9 biến logic
  2, _, 1
  _, _, 1
  1, _, 2

b. Sinh các mệnh đề CNF

Duyệt qua toàn bộ ô vuông (grid), kiểm tra giá trị của nó:

- Nếu grid(i, j) = 'T', ta thêm mệnh đề [var_id] để đảm bảo biến logic đó nhận giá trị True.
- Nếu grid(i, j) = 'G', ta thêm mệnh đề [-var_id] để đảm bảo biến logic đó nhận giá trị False.
- Nếu grid(i, j) = k:
  - Lấy danh sách các ô lân cận bằng get_neighbors(i, j, n_rows, n_columns).
  - Lấy danh sách các ô chứa bẫy (n_traps)
  - Tạo danh sách các biến logic cho các ô trống (unknown_neighbors).
  - Số bẫy còn lại cần xác định là remaining = k – len(n_traps).
  - Kiểm tra tính khả thi: nếu số bẫy còn lại (remaining) nhỏ hơn 0 hoặc lớn hơn số ô chưa xác định lân cận (unknown neighbors) thì bài toán không có lời giải, nếu số bẫy còn lại bằng 0 ta cần thêm mệnh đề [-x *for* x *in* unknown_neighbors] để đảm bảo không có ô trống nào là bẫy, nếu số bẫy còn lại bằng đúng với số ô chưa xác định lân cận thì ta cần thêm mệnh đề [x *for* x *in* unknown_neighbors] để đảm bảo tất các các ô trống còn lại là bẫy.
  - Tạo các ràng buộc "at least k true".
  - Tạo các ràng buộc "at most k true".

4. Áp dụng thư viện PySAT để giải CNF

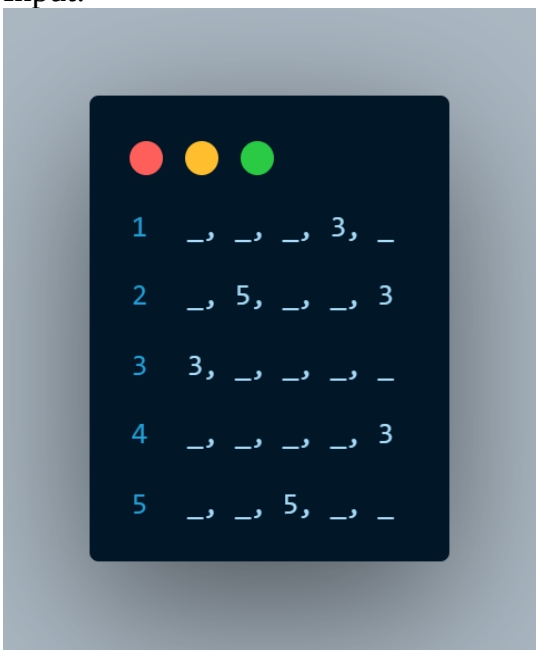
- Sau khi tạo CNF, ta áp dụng PySAT để tìm nghiệm.
- Nghiệm là tập hợp giá trị (True or False) cho các biến $var\_id_{i,j}$, tương ứng với việc gán T hoặc G cho các ô trống.
- Hàm solve_cnf() sử dụng PySAT:

```python
from pysat.solvers import Glucose3
from utils import *

def solve_cnf(cnf):
    solver = Glucose3()

    for clause in cnf:
        solver.add_clause(clause)

    if solver.solve():
        model = solver.get_model()
        return model
    else:
        return None
```

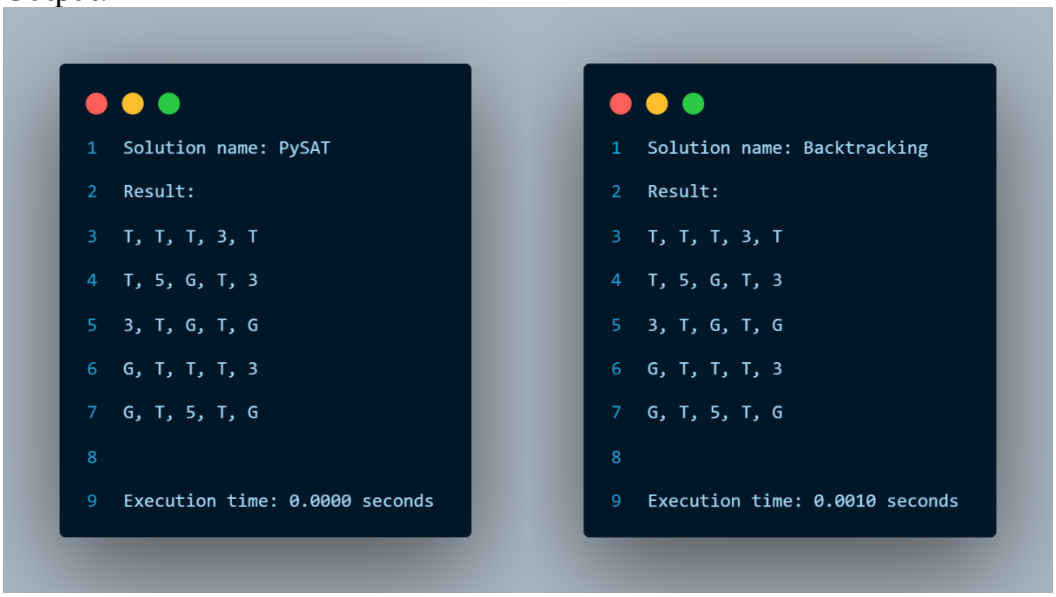
III. Thực nghiệm

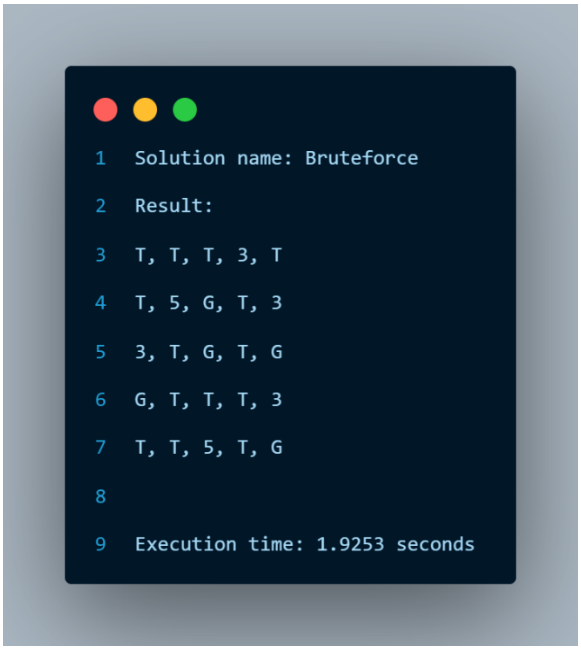
1. Lưới 5x5

- Input:



- Output:

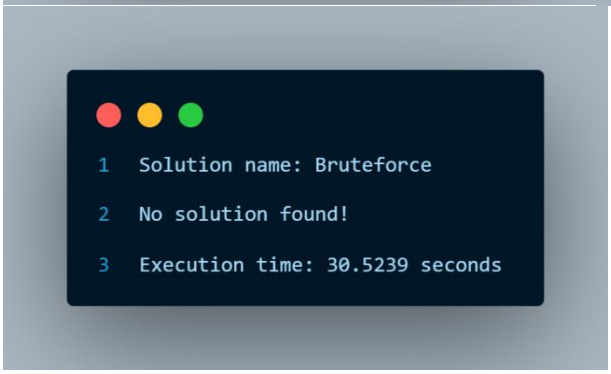




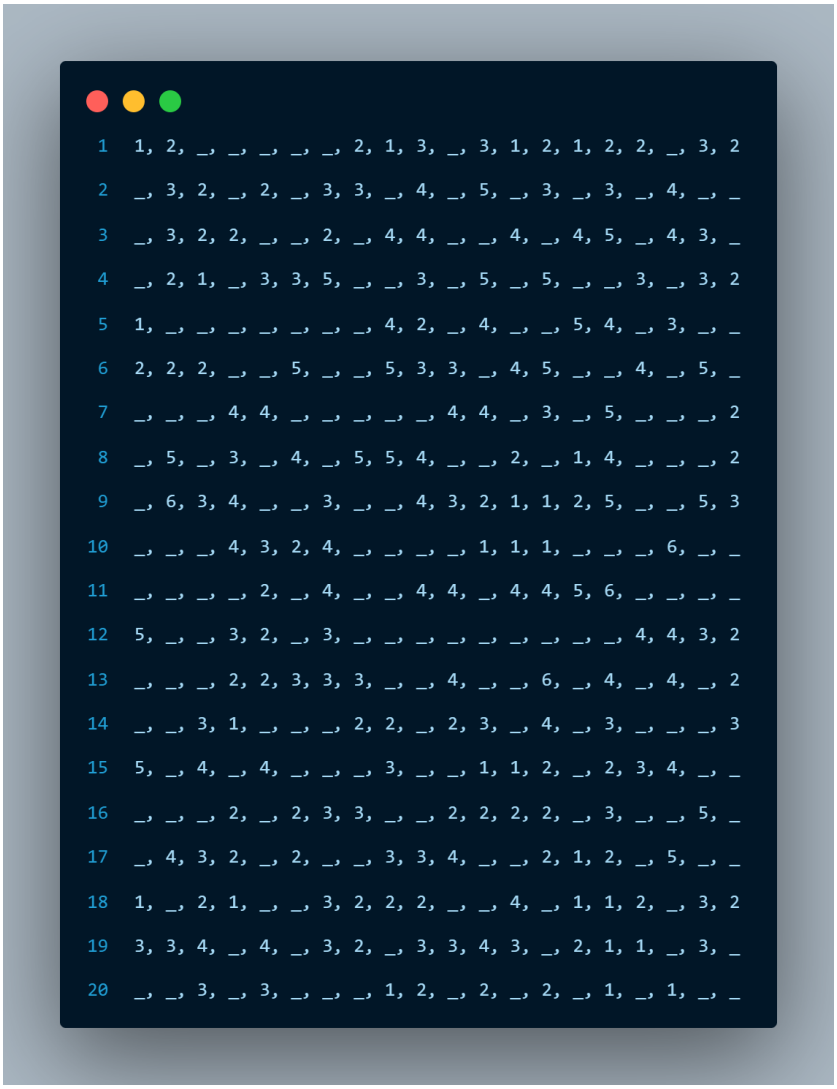
2. Lưới 11x11

- Input:

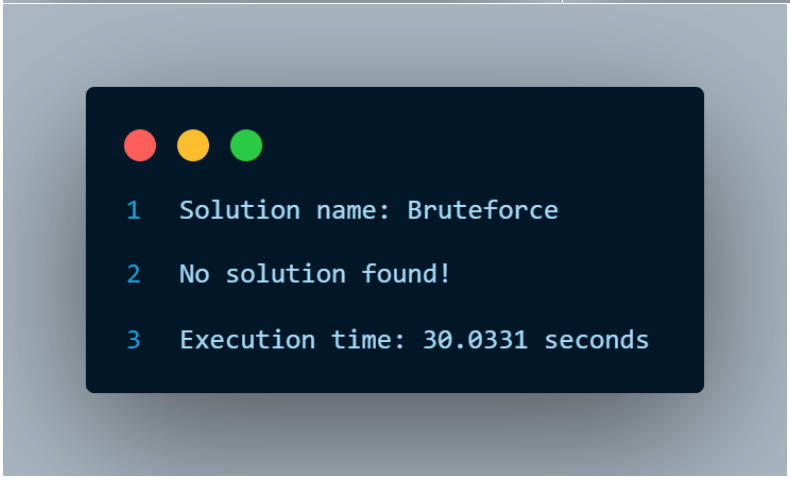
- Output:



3. **Lưới 20x20:**
   - Input:


1 1, 2, _, _, _, _, _, 2, 1, 3, _, 3, 1, 2, 1, 2, 2, _, 3, 2
2 _, 3, 2, _, 2, _, 3, 3, _, 4, _, 5, _, 3, _, 3, _, 4, _, _
3 _, 3, 2, 2, _, _, 2, _, 4, 4, _, _, 4, _, 4, 5, _, 4, 3, _
4 _, 2, 1, _, 3, 3, 5, _, _, 3, _, 5, _, 5, _, _, 3, _, 3, 2
5 1, _, _, _, _, _, _, _, 4, 2, _, 4, _, _, 5, 4, _, 3, _, _
6 2, 2, 2, _, _, 5, _, _, 5, 3, 3, _, 4, 5, _, _, 4, _, 5, _
7 _, _, _, 4, 4, _, _, _, _, _, 4, 4, _, 3, _, 5, _, _, _, 2
8 _, 5, _, 3, _, 4, _, 5, 5, 4, _, _, 2, _, 1, 4, _, _, _, 2
9 _, 6, 3, 4, _, _, 3, _, _, 4, 3, 2, 1, 1, 2, 5, _, _, 5, 3
10 _, _, _, 4, 3, 2, 4, _, _, _, _, 1, 1, 1, _, _, _, 6, _, _
11 _, _, _, _, 2, _, 4, _, _, 4, 4, _, 4, 4, 5, 6, _, _, _, _
12 5, _, _, 3, 2, _, 3, _, _, _, _, _, _, _, _, _, 4, 4, 3, 2
13 _, _, _, 2, 2, 3, 3, 3, _, _, 4, _, _, 6, _, 4, _, 4, _, 2
14 _, _, 3, 1, _, _, _, 2, 2, _, 2, 3, _, 4, _, 3, _, _, _, 3
15 5, _, 4, _, 4, _, _, _, 3, _, _, 1, 1, 2, _, 2, 3, 4, _, _
16 _, _, _, 2, _, 2, 3, 3, _, _, 2, 2, 2, 2, _, 3, _, _, 5, _
17 _, 4, 3, 2, _, 2, _, _, 3, 3, 4, _, _, 2, 1, 2, _, 5, _, _
18 1, _, 2, 1, _, _, 3, 2, 2, 2, _, _, 4, _, 1, 1, 2, _, 3, 2
19 3, 3, 4, _, 4, _, 3, 2, _, 3, 3, 4, 3, _, 2, 1, 1, _, 3, _
20 _, _, 3, _, 3, _, _, _, 1, 2, _, 2, _, 2, _, 1, _, 1, _, _

- Output:



4. Nhận xét

- Brute-force: Hiệu quả thấp, không khả thi cho lưới vừa và lớn do tính chất của thuật toán vét cạn, thời gian thực thi rất lâu.
- Backtracking: Hiệu quả trung bình, phù hợp với lưới có kích thước vừa phải.
- PySAT: Hiệu quả cao nhất, thời gian thực thi nhanh (vài giây đến vài phút) với lưới có kích thước lớn.

5. Link video demo

6. Tự đánh giá

Yêu cầu	Điểm	Hoàn thành
1. Solution description: Describe the correct logical principles for generating CNFs,	20%	90%
2. Generate CNFs automatically	10%	100%
3. Use PySAT library to solve CNFs correctly.	10%	100%
4. Program brute-force algorithm to compare with using library(speed).	10%	100%
5. Program backtracking algorithm to compare with using library (speed).	10%	100%
6. Documents and other resources that you need to write and analysis in your report: Thoroughness in analysis and experimentation. Give at least 3 test cases with different sizes (5x5, 11x11, 20x20) to check your solution Comparing result and performance	40%	90%

IV. Tham khảo

[1] https://github.com/pysathq/pysat
[2] https://gist.github.com/davefernig/e670bda722d558817f2ba0e90ebce66f
[3] https://github.com/bgrohman/Brute-Force-Sudoku-Solver
[4] Slide in this course
[5] Claude AI