BỘ GIÁO DỤC VÀ ĐÀO TẠO
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

BÁO CÁO
Lab: Gem hunter

Môn: Cơ sở trí tuệ nhân tạo

Năm học: HKII - 2024-2025

Sinh viên: 22120418 – Huỳnh Trần Ty

GEM HUNTER

Họ tên: Huỳnh Trần Ty
MSSV: 22120418
Lớp: Trí tuệ nhân tạo 2022/1

Ngày 19 tháng 5 năm 2025

Mục lục

1 Giới thiệu

1.1 Tổng quan về bài toán

Trong báo cáo này, chúng ta sẽ triển khai một trò chơi Gem Hunter sử dụng dạng chuẩn tắc hội (Conjunctive Normal Form - CNF). Trò chơi này đòi hỏi người chơi khám phá một lưới để tìm các viên đá quý ẩn giấu trong khi tránh các bẫy. Mỗi ô có số đại diện cho số lượng bẫy xung quanh nó.

1.2 Mục tiêu

Mục tiêu chính của bài tập này là:

- Mô hình hóa bài toán dưới dạng các ràng buộc CNF
- Triển khai việc tạo CNF tự động
- Sử dụng thư viện pysat để giải quyết các CNF
- Lập trình thuật toán Brute-force và Backtracking để so sánh hiệu suất
- Phân tích và đánh giá kết quả

2 Đánh giá mức độ hoàn thành

Hạng mục	Trạng thái	Phần trăm
Sinh CNF	**Đã hoàn thành**	100%
Sử dụng PySAT, Backtracking, Brute-force	**Đã hoàn thành**	100%
So sánh PySat, Brute-force và Backtracking	**Đã hoàn thành**	100%
Viết báo cáo và phân tích kết quả	**Đã hoàn thành**	100%
Quay video chạy chương trình	**Đã hoàn thành**	100%

Bảng 1: Bảng đánh giá mức độ hoàn thành lab

3 Video Demo

- Link: Video demo gem hunter

4 Mô tả nguyên lý giải bài toán và biểu diễn dưới dạng CNF

4.1 Mô tả bài toán

Bài toán "Gem Hunter" yêu cầu phát hiện ra các **traps (bẫy)** và **gems (ngọc)** trong một lưới 2 chiều, dựa vào các ô có chứa số. Mỗi ô số cho biết **số lượng bẫy xung quanh nó**. Mỗi ô có thể chứa:

- Ký tự "_": ô chưa biết.
- Số nguyên từ 1–9: biểu thị số lượng bẫy xung quanh.
- "T": là bẫy (trap).
- "G": là gem (không phải bẫy).

Để giải bài toán này, ta cần xây dựng các **ràng buộc logic dưới dạng CNF (Conjunctive Normal Form)** để đưa vào bộ giải SAT (`pysat`) nhằm suy luận ra các vị trí chính xác của bẫy và ngọc.

4.2 Biến logic

Mỗi ô (i, j) trong lưới được ánh xạ thành một biến logic duy nhất:

- **Biến đúng (True):** ô này là bẫy.
- **Biến sai (False):** ô này là gem.

Việc ánh xạ từ tọa độ (i, j) sang biến số nguyên được thực hiện trong hàm `setup_variable_maps()`: Kết quả là hai bảng ánh xạ:

- `var_map`: từ tọa độ sang số biến.
- `reverse_map`: từ số biến về tọa độ.

4.3 Sinh ràng buộc CNF

1. Các ràng buộc CNF được sinh tự động trong hàm `generate_cnf()`.

   Nguyên lý: Với mỗi ô chứa số k, các ô xung quanh chưa biết sẽ được gom vào danh sách `unknowns`. Ta cần ràng buộc rằng **chính xác** k **ô** trong số đó là bẫy.

   Cách biểu diễn ràng buộc "chính xác k biến đúng":

   - Dựa trên kỹ thuật tổ hợp (`itertools`), ta xây dựng hai loại mệnh đề:
     - **Tối thiểu** k **biến đúng** (At least): Dạng OR của các biến – mỗi tổ hợp có $(n - k + 1)$ phần tử $\rightarrow$ biểu diễn rằng ít nhất k biến là đúng.
     - **Tối đa** k **biến đúng** (At most): Dạng OR phủ định các biến – mỗi tổ hợp có $(k + 1)$ phần tử $\rightarrow$ biểu diễn rằng không quá k biến là đúng.

2. Hàm `add_exact_k_constraint()` thực hiện nhiệm vụ sinh các tổ hợp này.

3. **Loại bỏ trùng lặp:** Tập mệnh đề CNF được lọc để **loại bỏ các mệnh đề trùng nhau** bằng cách dùng `set()`.

4. Tóm lại: Tại sao làm vậy?

   - CNF không thể biểu diễn trực tiếp "chính xác k đúng".
   - Nhưng ta có thể biểu diễn bằng tổ hợp của 2 ràng buộc:
     - Ít nhất k đúng đảm bảo không thiếu số bẫy.
     - Không quá k đúng đảm bảo không dư số bẫy.
   - Kết hợp lại, ta ép được chính xác k ô là bẫy.

4.4 Sinh CNF tự động

Hàm `generate_cnf(grid, var_map)` có nhiệm vụ tự động sinh các mệnh đề CNF từ tiles. Mỗi ô trên tiles có thể là:

- Một số nguyên k biểu thị số lượng bẫy xung quanh,
- Một ô chưa biết, ký hiệu là dấu gạch dưới `'_'`.

Ý tưởng chính:

- Duyệt qua tất cả các ô trên tiles. Với mỗi ô có chứa số k, xác định tập các ô chưa biết (`unknowns`) xung quanh nó.
- Với tập các ô chưa biết này, cần sinh các mệnh đề logic để biểu diễn rằng **chính xác k ô trong số đó là bẫy**.
- Để biểu diễn ràng buộc "chính xác k ô là bẫy" trong ngôn ngữ CNF, ta kết hợp hai loại ràng buộc:
  - Ràng buộc **tối thiểu k biến đúng (At least k)**: đảm bảo có ít nhất k ô là bẫy.
  - Ràng buộc **tối đa k biến đúng (At most k)**: đảm bảo không nhiều hơn k ô là bẫy.
- Các mệnh đề được sinh từ các tổ hợp tương ứng và được đưa vào tập CNF.
- Để tránh lặp lại các mệnh đề giống nhau, mỗi mệnh đề được chuẩn hóa và kiểm tra trước khi thêm vào CNF.

Kết quả đầu ra là một danh sách các mệnh đề CNF, sau đó được chuyển đổi thành đối tượng `CNF()` của thư viện `pysat` và các thuật toán backtracking và brute_force để giải quyết vấn đề.

5 Thuật toán

5.1 PySAT (SAT Solver)

1. **Ý tưởng:**
   - Mô hình hóa bài toán dưới dạng tập các mệnh đề logic (CNF).
   - Mỗi ô chưa biết là một biến boolean (bẫy hoặc không).
   - Dựa vào các con số gợi ý, sinh ra các ràng buộc "chính xác k bẫy" rồi giải bằng SAT solver (như MiniSAT).
2. **Chi tiết hoạt động:**
   (a) Với mỗi ô số k, tìm các ô xung quanh chưa biết → lập danh sách `unknowns`.
   (b) Dựa trên tổ hợp, sinh ràng buộc dạng CNF để biểu diễn điều kiện: *"chính xác k ô trong số đó là bẫy"*:
   - **Tối thiểu** k: với mỗi tổ hợp $(n - k + 1)$ ô, sinh một mệnh đề OR.

- **Tối đa** k: với mỗi tổ hợp $(k+1)$ ô, sinh một mệnh đề phủ định OR.

(c) Gom tất cả ràng buộc vào một đối tượng CNF.

(d) Đưa CNF vào SAT solver (ví dụ MiniSAT).

(e) Nếu solver trả về một nghiệm (satisfiable), ta có thể giải mã nghiệm này để biết ô nào là bẫy.

Ví dụ: Với ô số 2 và 3 ô xung quanh chưa biết: A, B, C. Sinh ràng buộc sao cho **chính xác 2 trong 3** là bẫy.

3. **Ưu điểm:**
   - Hiệu suất rất cao, kể cả với bài toán lớn.
   - Bảo đảm tìm được nghiệm đúng nếu tồn tại.
   - Dễ dàng mở rộng sang các biến thể logic khác.
4. **Nhược điểm:**
   - Cần hiểu rõ về biểu diễn logic và CNF.
   - Việc sinh ràng buộc ban đầu có thể phức tạp với người mới học.

5.2 Backtracking (Quay lui)

1. **Ý tưởng:**
   - Duyệt thử tất cả các cách gán giá trị cho các ô chưa biết.
   - Sau mỗi bước gán, kiểm tra xem nó có hợp lý không dựa vào các con số gợi ý.
   - Nếu hợp lý thì tiếp tục, ngược lại thì quay lui.
2. **Chi tiết hoạt động:**

(a) Tạo danh sách tất cả các ô chưa biết.

(b) Thử gán giá trị (bẫy hoặc không) cho ô đầu tiên trong danh sách.

(c) Sau mỗi bước gán, kiểm tra ràng buộc: các ô số xung quanh có còn thỏa điều kiện không?

(d) Nếu hợp lệ, tiếp tục đệ quy gán cho ô tiếp theo.

(e) Nếu sai, quay lui để thử lựa chọn khác.

(f) Khi đến ô cuối và tất cả ràng buộc đều đúng, ta đã tìm được một lời giải hợp lệ.

Ví dụ: Thử A là bẫy, kiểm tra. Nếu sai, thử A không phải bẫy. Cứ thế với B, C,...

3. **Ưu điểm:**
   - Dễ hiểu, dễ cài đặt.
   - Có thể tối ưu thêm bằng các chiến lược heuristic (ưu tiên ô dễ trước).
4. **Nhược điểm:**
   - Hiệu suất thấp với bài toán lớn do không tránh được tổ hợp.
   - Tăng nhanh theo số lượng ô chưa biết.

5.3 Brute Force (Duyệt toàn bộ)

1. **Ý tưởng:**
   - Sinh tất cả các tổ hợp có thể của các ô chưa biết.
   - Kiểm tra từng tổ hợp xem có thỏa mãn tất cả các gợi ý hay không.
2. **Chi tiết hoạt động:**
   (a) Xác định tất cả các ô chưa biết.
   (b) Sinh **mọi tổ hợp nhị phân** (bẫy/không) cho các ô này.
   (c) Với mỗi tổ hợp:
      - Gán thử lên lưới.
      - Kiểm tra toàn bộ các ô số xem có thỏa điều kiện không (số bẫy xung quanh đúng).
   (d) Nếu có một tổ hợp thỏa mãn tất cả điều kiện, lưu lại làm lời giải.

   Ví dụ: Với 3 ô A, B, C $\rightarrow$ có $2^3 = 8$ tổ hợp. Thử từng tổ hợp: [000], [001], [010], ..., [111].
3. **Ưu điểm:**
   - Rất dễ cài đặt.
   - Dùng để kiểm chứng độ chính xác nếu số ô chưa biết rất ít.
4. **Nhược điểm:**
   - Không dùng được cho bài toán thực tế với nhiều ô chưa biết.
   - Thời gian chạy tăng theo cấp số mũ (exponential time).

6 Thực nghiệm và phân tích hiệu năng

6.1 Các bộ test

- Test 1: Bản đồ 5x5 với 16 ô chưa giải
- Test 2: Bản đồ 10x10 với 67 ô chưa giải
- Test 3: Bản đồ 15x15 với 129 ô chưa giải
- Test 4: Bản đồ 20x20 với 235 ô chưa giải

6.2 So sánh thời gian chạy và độ chính xác của từng thuật toán

1. Các bảng so sánh từ thí nghiệm

Kích thước	SL ô cần giải	PySAT (ms)	Backtracking (ms)	Brute Force (ms)
5×5	16	1.001	0.994	329.076
10×10	67	2.429	66.541	*Long*
15×15	129	2.122	832.893	*Long*
20×20	235	3.094	6342.932	*Long*

Bảng 2: So sánh thời gian chạy giữa các phương pháp giải

Thuật toán	Độ chính xác
PySAT	100%
Backtracking	100%
Brute-force	Không ổn định, có thể < 100%

Bảng 3: So sánh độ chính xác của các thuật toán

2. PySAT và Backtracking Kết quả giống nhau 100 % trên tất cả các bộ test.
   Lý do:
   - Cả hai thuật toán đều dựa trên logic và ràng buộc chặt chẽ từ các con số gợi ý (ô chứa số).
   - Chúng đảm bảo rằng số lượng mìn (bẫy) xung quanh mỗi ô đều đúng như yêu cầu.

   → Độ chính xác: 100%

3. Brute-force Kết quả không chính xác hoàn toàn, có một số ô xác định sai.
   Lý do:
   - Brute-force tạo tất cả các tổ hợp khả dĩ của mìn và kiểm tra xem tổ hợp nào hợp lệ.
   - Nhưng trong nhiều trường hợp có nhiều lời giải thỏa mãn, Brute-force chỉ lấy ra một lời giải bất kỳ (thường là lời giải đầu tiên tìm được).
   - Nếu yêu cầu là xác định chắc chắn ô nào luôn luôn là mìn hoặc luôn luôn an toàn trong mọi lời giải, thì brute-force cần phân tích tất cả lời giải hợp lệ, điều mà nó không làm được hoàn chỉnh.

   → Độ chính xác: $< 100\%$, giảm dần khi số ô chưa biết tăng.

7 Kết luận

- Ưu điểm của việc sử dụng SAT solver so với cách giải truyền thống.
- Vai trò của logic và biểu diễn tri thức trong AI.
- Khả năng mở rộng bài toán với input lớn hơn.

7.1 Nhận xét chi tiết

1. **Thuật toán CNF + pysat:**
   - Hiệu suất vượt trội với thời gian chạy rất nhanh ngay cả trên bảng kích thước lớn.
   - Độ phức tạp thời gian tăng gần như tuyến tính theo kích thước bài toán.
   - Lý do chính: Các bộ giải SAT hiện đại sử dụng nhiều kỹ thuật tối ưu hóa, cắt tỉa và học hỏi từ xung đột (conflict-driven clause learning).
2. **Thuật toán brute-force:**
   - Hiệu suất kém nhất, chỉ khả thi với bài toán kích thước nhỏ (5×5).
   - Không thể hoàn thành trong thời gian hợp lý với bài toán kích thước 11×11 trở lên.
   - Độ phức tạp thời gian là $O(2^n)$ với n là số lượng ô chưa biết, dẫn đến sự bùng nổ tổ hợp.
3. **Thuật toán backtracking:**
   - Hiệu suất tốt hơn đáng kể so với brute-force nhờ vào việc cắt tỉa các nhánh tìm kiếm sớm.
   - Vẫn chậm hơn nhiều so với phương pháp dựa trên CNF + pysat.
   - Có thể giải quyết được bài toán kích thước 20×20 nhưng thời gian chạy tăng đáng kể.

7.2 Kết luận so sánh

Từ kết quả thực nghiệm, chúng ta có thể rút ra các kết luận sau:

1. Phương pháp dựa trên SAT (CNF + pysat) là lựa chọn tối ưu cho bài toán Gem Hunter, với hiệu suất vượt trội và có thể mở rộng tốt cho các bài toán kích thước lớn.
2. Thuật toán brute-force không phù hợp cho bài toán này do độ phức tạp tổ hợp cao, chỉ phù hợp cho các bài toán có số lượng ô chưa biết rất nhỏ.
3. Thuật toán backtracking là một cải tiến đáng kể so với brute-force, nhưng vẫn không thể cạnh tranh với phương pháp dựa trên SAT về hiệu suất.
4. Thời gian chạy của thuật toán backtracking tăng nhanh theo kích thước bảng, trong khi thuật toán CNF + pysat tăng khá chậm, cho thấy tính mở rộng tốt hơn.

Về lý do tại sao thuật toán CNF + pysat hiệu quả hơn nhiều:

- Các bộ giải SAT hiện đại đã được tối ưu hóa cao với nhiều kỹ thuật tiên tiến như CDCL (Conflict-Driven Clause Learning), heuristic điều hướng tìm kiếm thông minh, và cắt tỉa không gian tìm kiếm hiệu quả.

- Biểu diễn bài toán dưới dạng CNF cho phép bộ giải SAT phân tích cấu trúc ràng buộc và áp dụng các kỹ thuật lan truyền ràng buộc (constraint propagation) hiệu quả.
- Nhiều bộ giải SAT hiện đại được triển khai song song, tận dụng sức mạnh của nhiều nhân CPU.

8 Tài liệu tham khảo

- PySAT: `https://pysathq.github.io`
- Slides bài giảng môn Trí tuệ nhân tạo