**TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN**

**ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH**

**KHOA CÔNG NGHỆ THÔNG TIN**

**---🙠🕮🙢---**

**A blue and white logo

Description automatically generated**

**MÔN HỌC CƠ SỞ TRÍ TUỆ NHÂN TẠO**

**NĂM HỌC 2024 – 2025**

**HỌC KỲ II**

**BÁO CÁO BÀI TẬP THỰC HÀNH**

**GEM HUNTER**

**Họ và tên:** Nguyễn Minh Thiện

**MSSV:** 22120344

**Lớp:** CQ2022/5

**Email:** [22120344@student.hcmus.edu.vn](mailto:22120344@student.hcmus.edu.vn)

**SĐT:** 0949455062

**TPHCM, 30/03/2025**

**MỤC LỤC**

[I. MÔ TẢ GIẢI PHÁP: 1](#_Toc198470045)

[II. PHƯƠNG PHÁP SINH CNF TỰ ĐỘNG: 4](#_Toc198470046)

* [Hàm generate\_cnf(): 4](#_Toc198470047)

[III. PHƯƠNG PHÁP GIẢI CNF SỬ DỤNG PYSAT: 4](#_Toc198470048)

[IV. PHƯƠNG PHÁP GIẢI CNF SỬ DỤNG BRUTE-FORCE: 5](#_Toc198470049)

[V. PHƯƠNG PHÁP GIẢI CNF SỬ DỤNG BACKTRACKING: 5](#_Toc198470050)

[VI. PHÂN TÍCH VÀ THỰC NGHIỆM: 5](#_Toc198470051)

[VII. KẾT LUẬN: 6](#_Toc198470052)

# MÔ TẢ GIẢI PHÁP:

1. **Giới thiệu bài toán:**

* Gem Hunter là một trò chơi logic trên lưới ô vuông, trong đó
* Các ô có thể chứa bẫy (Trap - T) hoặc đá quý (Gem - G)
* Các ô có số từ 1-9 thể hiện số lượng bẫy trong 8 ô xung quanh
* Mục tiêu là xác định vị trí của tất cả các bẫy và đá quý trên lưới.

1. **Tiếp cận giải quyết bằng CNF (Conjunctive Normal Form):** Bài toán được chuyển thành bài toán thỏa mãn ràng buộc logic (SAT) sử dụng Conjunctive Normal Form (CNF)
2. *Biến logic cho mỗi ô:*

* Gán một biến logic cho mỗi ô trên lưới.
* Biến = True (T): Ô chứa bẫy.
* Biến = False (G): Ô chứa đá quý.

1. *Ràng buộc dạng CNF:*

* Đối với mỗi ô chứa số k:
* Lấy danh sách các ô lân cận (tối đa 8 ô).
* Đếm số bẫy đã có (**trap\_count**) trong các ô lân cận.
* Tính **remaining = k - trap\_count**: Số bẫy cần đặt trong các ô lân cận trống.
* Với các ô lân cận trống (có **unknown\_count** ô), cần đúng **remaining** ô là bẫy. Ràng buộc này được chia thành:
* Ít nhất remaining bẫy: Đảm bảo có ít nhất remaining biến True
* Nhiều nhất **remaining** bẫy: Đảm bảo không có quá **remaining** biến True.

1. **Biểu diễn CNF:**

* Cho một ô có số k, tập biến lân cận trống X = {x₁, x₂,..., xₙ} (n = unknown\_count = 8):

1. *Ràng buộc "Ít nhất* ***remaining*** *bẫy":*

* Tạo mệnh đề cho mọi tổ hợp n - remaining + 1 biến: (x1 ∨ x2 ∨ ... ∨ xm)

với m = n - remaining + 1.

Ví dụ: Để có ít nhất 2 biến trong tập 4 biến {x₁, x₂, x₃, x₄}, ta cần các mệnh đề:

(x₁ ∨ x₂ ∨ x₃) - nếu cả 3 biến này đều False, thì không thể có 2 biến True

(x₁ ∨ x₂ ∨ x₄)

(x₁ ∨ x₃ ∨ x₄)

(x₂ ∨ x₃ ∨ x₄)

1. *Ràng buộc "Nhiều nhất* ***remaining*** *bẫy":*

* Tạo mệnh đề cho mọi tổ hợp remaining + 1 biến: (¬xj1 ∨ ¬xj2 ∨ ... ∨ ¬xjk)

với k = remaining + 1

Ví dụ: Để có nhiều nhất 2 biến True trong {x₁, x₂, x₃, x₄}, chúng ta cần:

(¬x₁ ∨ ¬x₂ ∨ ¬x₃) - không thể có cả 3 biến này là True

(¬x₁ ∨ ¬x₂ ∨ ¬x₄)

(¬x₁ ∨ ¬x₃ ∨ ¬x₄)

(¬x₂ ∨ ¬x₃ ∨ ¬x₄)

1. **Trường hợp đặc biệt:**

* remaining = 0: Tất cả 8 ô xung quanh đều là đá quý (G)
* remaining = 8: Tất cả 8 ô xung quanh đều là bẫy (T)
* Ô đã biết giá trị: Không cần tạo biến cho ô đó

1. **Biểu diễn CNF cho test case cụ thể:**

A screenshot of a computer

AI-generated content may be incorrect.

**Hình ảnh:** Ví dụ là test case 1 trong bài code.

***Ví dụ 1:*** Ô có giá trị 2 (góc trên bên trái)

* Ô (0,0) = 2, nghĩa là cần có đúng 2 bẫy trong các ô lân cận.
* Lân cận: (0,1)=\_, (1,0)=\_, (1,1)=5.
* Biến: x₁ = (0,1), x₂ = (1,0)
* k = 2, trap\_count = 0, remaining = 2, unknown\_count = 2
* Ràng buộc: [x₁], [x₂] (tất cả là T)
* Nếu đặt biến:
* x₁ = ô (0,1)
* x₂ = ô (1,0)
* x₃ = ô (1,1)
* Cần đảm bảo có đúng 2 biến trong tập {x₁, x₂, x₃} là True.
* Ràng buộc "Ít nhất 2":
  + n = 3, k = 2
  + Chọn (n - k + 1) = 2 biến: {x₁, x₂}, {x₁, x₃}, {x₂, x₃}
  + Mệnh đề: (x₁ ∨ x₂), (x₁ ∨ x₃), (x₂ ∨ x₃)
  + Ý nghĩa: Không thể có 2 biến cùng False (vì nếu có 2 biến False thì chỉ có 1 biến True)
* Ràng buộc "Nhiều nhất 2":
  + Chọn (k+1) = 3 biến: {x₁, x₂, x₃}
  + Mệnh đề: (¬x₁ ∨ ¬x₂ ∨ ¬x₃)
  + Ý nghĩa: Không thể có cả 3 biến đều True

***Ví dụ 2:*** Ô có giá trị 5 (ở vị trí (1,1))

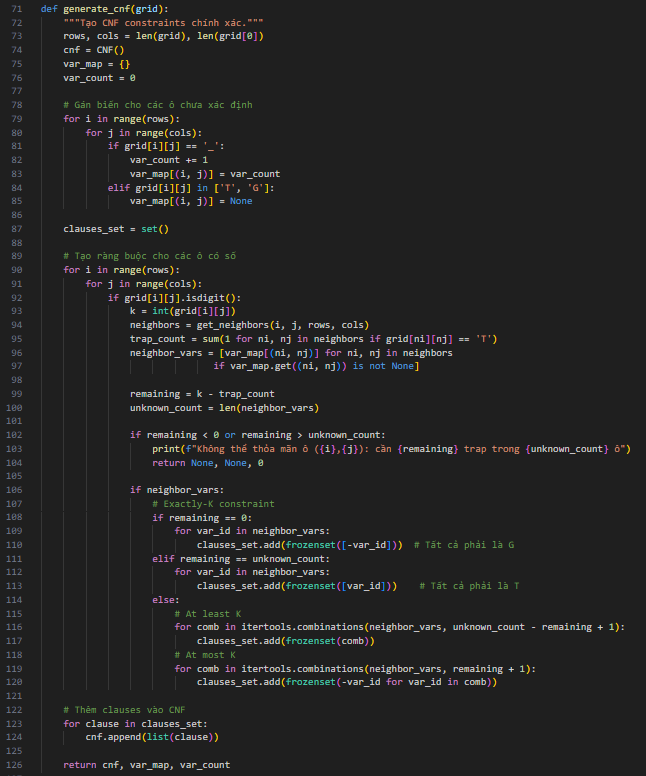
* Ô (1,1) có giá trị 5, nghĩa là cần có đúng 5 bẫy trong các ô lân cận.
* Lân cận: (0,0)=2, (0,1)=\_, (0,2)=\_, (1,0)=\_, (1,2)=4, (2,0)=3, (2,1)=\_, (2,2)=\_.
* k = 5, trap\_count = 0, remaining = 5, unknown\_count = 5
* Giả sử đã biết giá trị của một vài ô:
* (0,0) = 2 (là số, không phải bẫy/đá quý)
* (1,0) = chưa biết (đặt biến là x₁)
* (0,1) = chưa biết (đặt biến là x₂)
* (2,0) = 3 (là số, không phải bẫy/đá quý)
* (0,2) = chưa biết (đặt biến là x₃)
* (2,1) = chưa biết (đặt biến là x₄)
* (1,2) = 4 (là số, không phải bẫy/đá quý)
* (2,2) = chưa biết (đặt biến là x₅)
* Ở đây, có 5 biến không biết giá trị {x₁, x₂, x₃, x₄, x₅} và cần đúng 5 bẫy trong 8 ô lân cận. Vì có 3 ô là số (không phải bẫy), nên tất cả 5 ô còn lại phải là bẫy.
* Ràng buộc "Ít nhất 5":
  + Vì cần đúng 5 bẫy trong 5 biến, nên tất cả các biến phải là True
  + Mệnh đề: (x₁), (x₂), (x₃), (x₄), (x₅)
  + Nghĩa là: x₁ phải True, x₂ phải True, ...
* Ràng buộc "Nhiều nhất 5":
  + Trường hợp này được thỏa mãn tự động vì chỉ có 5 biến

***Ví dụ 3:*** Ô có giá trị 1 (ở vị trí (0,3))

* Ô (0,3) có giá trị 1, nghĩa là cần có đúng 1 bẫy trong các ô lân cận.
* Lân cận: (0,2)=\_, (0,4)=\_, (1,2)=4, (1,3)=2, (1,4)=\_
* k = 1, trap\_count = 0, remaining = 1, unknown\_count = 3
* Giả sử:
* (0,2) = chưa biết (đặt biến là x₁)
* (0,4) = chưa biết (đặt biến là x₂)
* (1,2) = 4 (là số, không phải bẫy/đá quý)
* (1,3) = 2 (là số, không phải bẫy/đá quý)
* (1,4) = chưa biết (đặt biến là x₃)
* Ràng buộc "Ít nhất 1":
  + Mệnh đề: (x₁ ∨ x₂ ∨ x₃)
  + Ý nghĩa: Ít nhất một trong các biến phải là True
* Ràng buộc "Nhiều nhất 1":
  + Mệnh đề: (¬x₁ ∨ ¬x₂), (¬x₁ ∨ ¬x₃), (¬x₂ ∨ ¬x₃)
  + Ý nghĩa: Không thể có 2 biến nào đồng thời là True

# PHƯƠNG PHÁP SINH CNF TỰ ĐỘNG:

## Hàm generate\_cnf():



* Sử dụng frozenset và set để đảm bảo không có mệnh đề trùng, giảm kích thước CNF
* Xử lý các trường hợp đặc biệt
* remaining = 0 *=> Thêm mệnh đề đơn vị [-var\_id] (tất cả là G)*
* remaining = unknown\_count *=> Thêm mệnh đề đơn vị [var\_id] (tất cả là T)*
* Giảm độ phức tạp của CNF
* Sử dụng itertools.combinations tạo ra các tổ hợp cần thiết cho ràng buộc "Ít nhất k" và "Nhiều nhất k".
* Nếu remaining < 0 hoặc remaining > unknown\_count, trả về None để báo lỗi lưới không khả thi

# PHƯƠNG PHÁP GIẢI CNF SỬ DỤNG PYSAT:

1. **Mô Tả:**

* Sử dụng thư viện PySAT với bộ giải Glucose3 để giải công thức CNF.
* Hàm solve\_with\_pysat:
* Tiền xử lý lưới để giảm số ô trống.
* Tạo CNF bằng generate\_cnf.
* Khởi tạo solver và giải.
* Ánh xạ lời giải về lưới, ghi kết quả ra tệp output\_x.txt.

1. **Triển khai:**

* Tiền xử lý: Hàm *preprocess\_grid* gán T hoặc G cho các ô bắt buộc dựa trên ràng buộc số.
* Kiểm tra hợp lệ: Hàm *is\_grid\_valid* đảm bảo lưới khả thi.
* Giải CNF:
* Khởi tạo Glucose3(bootstrap\_with=cnf.clauses).
* Nếu solver.solve() trả về True, lấy model và gán T (nếu biến True) hoặc G (nếu biến False) cho ô trống.
* Đo thời gian: dùng time.perf\_counter() để đo hiệu suất..

# PHƯƠNG PHÁP GIẢI CNF SỬ DỤNG BRUTE-FORCE:

1. **Mô Tả:**

* Hàm optimized\_brute\_force liệt kê tất cả tổ hợp giá trị (True/False) cho các biến logic.
* Tối ưu bằng:
* Tiền xử lý để giảm số biến.
* Sắp xếp biến theo tần suất xuất hiện (VSIDS heuristic).
* Giới hạn số biến ≤ 90 để tránh thời gian chạy quá lâu..

1. **Triển Khai:**

* Tiền xử lý: Gọi preprocess\_grid và is\_grid\_valid.
* Tạo CNF: dùng generate\_cnf để lấy danh sách biến.
* Sắp xếp biến: Biến xuất hiện nhiều trong mệnh đề được ưu tiên.
* Duyệt tổ hợp: Sử dụng itertools.product để tạo tổ hợp giá trị.
* Kiểm tra hợp lệ: Hàm is\_valid kiểm tra mỗi gán thỏa mãn tất cả mệnh đề.
* Kết quả: Trả về lưới đầu tiên thỏa mãn hoặc None nếu không tìm thấy

# PHƯƠNG PHÁP GIẢI CNF SỬ DỤNG BACKTRACKING:

1. **Mô Tả:**

* Hàm optimized\_backtracking sử dụng backtracking với các tối ưu:
* MRV (Minimum Remaining Values): Chọn biến xuất hiện nhiều trong mệnh đề (VSIDS).
* LCV (Least Constraining Value): Thử True trước False.
* Forward Checking: Kiểm tra tính nhất quán và áp dụng unit propagation.

1. **Triển Khai:**

* Tiền xử lý: Gọi preprocess\_grid và is\_grid\_valid.
* Tạo CNF: Sử dụng generate\_cnf, xây dựng watch\_list và clause\_db.
* Backtracking:
* Chọn biến bằng select\_unassigned\_variable (VSIDS).
* Kiểm tra tính nhất quán bằng is\_consistent.
* Lưu gán vào assignments, quay lui nếu thất bại.
* Kết quả: Ánh xạ assignments về lưới hoặc trả về None nếu không tìm thấy

# PHÂN TÍCH VÀ THỰC NGHIỆM:

1. **Bộ test cases:**
2. Test case 1: 5x5

A screenshot of a computer

AI-generated content may be incorrect.

1. Test case 2: 11x11

A screenshot of a computer program

AI-generated content may be incorrect.

1. Test case 3: 20x28

A black screen with white lines

AI-generated content may be incorrect.

1. Test case 4: 6x6

A screenshot of a computer

AI-generated content may be incorrect.

1. **Kết quả:**
2. Test case 1: 5x5

A screenshot of a computer

AI-generated content may be incorrect.

1. Test case 2: 11x11

A screenshot of a computer

AI-generated content may be incorrect.

1. Test case 3: 20x28

A screenshot of a computer screen

AI-generated content may be incorrect.

1. Test case 4: 6x6

A screenshot of a computer

AI-generated content may be incorrect.

1. **So sánh hiệu năng:** Thời gian thực thi được đo bằng time.perf\_counter()

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Phương pháp** | **Test 1 (5x5)** | **Test 2 (11x11)** | **Test 3 (20x28)** | **Test 4 (6x6)** |
| PySAT | 0.000526s | 0.001810s | 0.003778s | 0.000919s |
| Brute-force | 0.000145s | Không khả thi | Không khả thi | 95.292531s |
| Backtracking | 0.000133s | 0.079489s | 4.001633s | 0.014295s |

* **Nhận xét:**
* PySAT: Nhanh nhất, nhờ bộ giải Glucose3 tối ưu.
* Brute-force: Chỉ khả thi với lưới nhỏ (≤ 40 biến), do độ phức tạp O(2^n).
* Backtracking: Nhanh hơn brute-force nhờ tối ưu, nhưng chậm hơn PySAT với lưới lớn.

1. **Phân tích kết quả và So sánh về thuật toán:**

* PySAT:
* Ưu điểm: Hiệu quả với mọi kích thước lưới, tận dụng thuật toán DPLL và CDCL.
* Nhược điểm: Phụ thuộc vào thư viện bên ngoài.
* Brute-force:
* Ưu điểm: Đơn giản, đảm bảo tìm lời giải nếu tồn tại.
* Nhược điểm: Không khả thi với số biến lớn.
* Backtracking:
* Ưu điểm: Hiệu quả hơn brute-force nhờ MRV, LCV, và forward checking.
* Nhược điểm: Conflict analysis giản lược, kém hiệu quả với lưới lớn so với PySAT...

# KẾT LUẬN:

* PySAT là phương pháp hiệu quả nhất, phù hợp với mọi kích thước lưới.
* Brute-force chỉ phù hợp với lưới nhỏ, trong khi Backtracking là lựa chọn trung gian.
* Các kỹ thuật tiền xử lý, loại bỏ trùng lặp, và heuristic (VSIDS, MRV, LCV) cải thiện đáng kể hiệu suất.

# LINK YOUTUBE: