**TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN, ĐHQG-HCM**

**KHOA CÔNG NGHỆ THÔNG TIN**

🙡&🙣

A blue and white logo

Description automatically generated

**BÁO CÁO LAB: GEM HUNTER**

**Môn học:** Cơ sở trí tuệ nhân tạo

**Mã lớp:** CQ2022/3

**GVLT:** Bùi Duy Đăng

**GVTH:** Nguyễn Trần Duy Minh

**SV thực hiện:** 22120448 – Bùi Đoàn Thúy Vy

*Thành phố Hồ Chí Minh, ngày 15 tháng 05 năm 2025*

*Mục lục*

[**I.** **Giới thiệu** 3](#_Toc198253034)

[**II.** **Mô tả bài toán** 3](#_Toc198253035)

[**III.**  **Giải pháp và triển khai** 4](#_Toc198253036)

[**Mô hình hóa logic:** 4](#_Toc198253037)

[**Sinh ràng buộc(CNF):** 4](#_Toc198253038)

[**Giải bằng PySAT:** 5](#_Toc198253039)

[**Cài đặt Brute-force:** 6](#_Toc198253040)

[**Cài đặt backtracking:** 7](#_Toc198253041)

[**IV.**  **Thực hiện** 9](#_Toc198253042)

[**V.** **Đánh giá mức độ hoàn thành** 9](#_Toc198253043)

[**VI. Link demo** 9](#_Toc198253044)

[**VII. Tham khảo** 9](#_Toc198253045)

# **Giới thiệu**

**Gem Hunter** là một bài toán lập trình mang tính logic, được xây dựng như một trò chơi mô phỏng việc truy tìm viên ngọc (gem) trên một lưới ô vuông. Trò chơi có nhiều nét tương đồng với **Minesweeper (Dò mìn)**, nhưng được mở rộng với những biến thể thú vị. Trong game, người chơi sẽ khám phá từng ô trong lưới để tìm kiếm các viên ngọc (ký hiệu **G**), đồng thời phải tránh rơi vào các bẫy (**T**). Mỗi ô trên lưới có thể là một con số, một bẫy (**T**), hoặc một viên ngọc (**G**), và người chơi cần suy luận dựa trên thông tin hiện có để đưa ra bước đi chính xác.

# **Mô tả bài toán**

* Mỗi ô trong lưới có thể là:
  + Số (từ 1–9): chỉ số lượng bẫy (T) xung quanh 8 ô lân cận.
  + Trống: ô chưa biết (có thể là T hoặc G)
  + T: ô bẫy (trap)
  + G: ô chứa đá quý (gem)
* **Mục tiêu:** - Xác định chính xác các ô là T hay G dựa trên ràng buộc từ các số.

- Các ô chứa số tạo ra các ràng buộc logic: Số lượng bẫy trong 8 ô lân cận phải đúng bằng số được cho.

- Bài toán được diễn đạt dưới dạng logic mệnh đề, sau đó chuyển thành CNF để sử dụng các công cụ giải SAT (như pysat).

# **III. Giải pháp và triển khai**

## **Mô hình hóa logic:**

* **Biến Boolean:**  
  Mỗi ô (i, j) trên lưới được gán một biến Boolean duy nhất thông qua hàm:

|  |
| --- |
| def varnum(i, j, cols):  return i \* cols + j + 1 |

* Biến này biểu diễn cho ô (i, j), với giá trị:
* **True**: ô chứa bẫy (T)
* **False**: ô chứa đá quý (G)
* **Ràng buộc logic dựa trên ô chứa số:**  
  Với mỗi ô có giá trị số nguyên k, ta có ràng buộc:  
  **Số lượng ô lân cận chứa bẫy (True) phải bằng đúng k.**
* **Xác định ô lân cận:**  
  Hàm neighbors(i, j, rows, cols) lấy tất cả các ô lân cận (tối đa 8 ô xung quanh) hợp lệ, loại trừ chính ô (i, j):

|  |
| --- |
| def neighbors(i, j, rows, cols):  for dx in [-1, 0, 1]:  for dy in [-1, 0, 1]:  if dx == 0 and dy == 0:  continue  ni, nj = i + dx, j + dy  if 0 <= ni < rows and 0 <= nj < cols:  yield (ni, nj) |

Hàm này cho phép tìm ra chính xác các ô lân cận mà ràng buộc về số lượng bẫy cần được áp dụng.

## **Sinh ràng buộc(CNF):**

Hàm chính sinh ràng buộc là gen\_cnf\_from\_grid(grid). Quy trình sinh ràng buộc:

* Lặp qua từng ô (i, j) trong lưới:
  + Nếu ô có giá trị số nguyên cell, ta tiến hành sinh ràng buộc cho ô đó.
* Lấy danh sách các biến vars\_ đại diện cho các ô lân cận chưa biết (tất cả ô lân cận đều được lấy biến Boolean bằng hàm varnum).
* Sinh hai tập ràng buộc để đảm bảo số lượng biến đúng (bẫy) trong các ô lân cận là đúng bằng cell:
  + 1. **Ràng buộc "ít nhất cell biến True":**  
       Tương đương không có tổ hợp nào gồm len(vars\_) - cell + 1 biến là False cùng lúc.  
       Ta sinh clause dương từ các tổ hợp biến kích thước này, ví dụ:

|  |
| --- |
| for combo in combinations(vars\_, len(vars\_) - cell + 1):  clause = tuple(sorted(combo))  clause\_set.add(clause) |

* + 1. **Ràng buộc "không quá cell biến True":**  
       Tương đương không có tổ hợp nào gồm cell + 1 biến là True cùng lúc.  
       Ta sinh clause phủ định (biến âm) từ các tổ hợp biến kích thước cell + 1, ví dụ:

|  |
| --- |
| for combo in combinations(vars\_, cell + 1):  clause = tuple(sorted([-v for v in combo]))  clause\_set.add(clause) |

* **Loại bỏ trùng lặp clause:**  
  Tất cả các clause được lưu trong một tập set (clause\_set) để tránh thêm nhiều lần các mệnh đề giống nhau.
* Cuối cùng, tạo đối tượng CNF từ tập các clause đã sinh:

|  |
| --- |
| cnf = CNF()  for clause in clause\_set:  cnf.append(list(clause))  return cnf |

* **Ý nghĩa logic của ràng buộc:**
  + Ràng buộc đầu đảm bảo **ít nhất** cell ô trong lân cận phải là bẫy.
  + Ràng buộc sau đảm bảo **không quá** cell ô trong lân cận là bẫy.  
    => Tổng hợp lại, số bẫy chính xác bằng cell.

## **Giải bằng PySAT:**

* Hàm solve\_with\_pysat(grid) thực hiện:

1. Gọi gen\_cnf\_from\_grid(grid) để tạo CNF từ lưới đầu vào.
2. Khởi tạo solver Glucose3:

|  |
| --- |
| solver = Glucose3()  solver.append\_formula(cnf.clauses) |

1. Thực hiện giải bài toán SAT:

|  |
| --- |
| if solver.solve():  model = solver.get\_model() |

* + Nếu có nghiệm, model là danh sách các biến với giá trị dương (True) hoặc âm (False).

1. Duyệt qua từng ô trong lưới, với các ô không phải số:
   * Lấy biến v = varnum(i, j, cols)
   * Nếu v trong model (biến dương) => ô là bẫy 'T'
   * Nếu -v trong model (biến âm) => ô là đá quý 'G'
   * Nếu không xác định => '?'
2. Trả về ma trận kết quả có các ký tự 'T', 'G', số nguyên ban đầu hoặc '?'.
3. Nếu không tìm được nghiệm (solver.solve() trả về False), trả về None.

## **Cài đặt Brute-force:**

* Hàm solve\_bruteforce(grid, cnf) thực hiện giải bài toán bằng cách thử mọi tổ hợp gán giá trị cho các biến xuất hiện trong CNF.
  1. Lấy danh sách tất cả các biến xuất hiện trong CNF

|  |
| --- |
| vars\_in\_cnf = set(abs(lit) for clause in cnf.clauses for lit in clause) |

**Mục đích**: Chỉ lấy các biến thực sự quan trọng — tức là **có mặt trong mệnh đề CNF**.

* + Mỗi biến là một số nguyên đại diện cho ô (i,j).
  + CNF có thể chứa literal âm hoặc dương nên dùng abs(lit).
  1. Duyệt lưới, lấy ra các ô chưa biết có mặt trong CNFTạo ánh xạ từ biến sang chỉ số tổ hợp

|  |
| --- |
| unknowns = []  vars\_used = []  for i in range(rows):  for j in range(cols):  if not isinstance(grid[i][j], int):  v = i \* cols + j + 1  if v in vars\_in\_cnf:  unknowns.append((i, j))  vars\_used.append(v) |

**Mục đích**:

* Xác định các ô chưa biết cần xét.
* Tạo danh sách biến tương ứng (vars\_used) theo thứ tự.

v = i \* cols + j + 1 là cách ánh xạ (i,j) → số nguyên duy nhất đại diện cho biến.

* 1. Tạo ánh xạ biến → vị trí trong danh sách tổ hợp

|  |
| --- |
| var\_to\_idx = {var: idx for idx, var in enumerate(vars\_used)} |

**Mục đích**:

* Khi xét tổ hợp các bit (True/False), dễ dàng tra nhanh vị trí của một biến cụ thể trong mảng bits.
  1. Định nghĩa hàm kiểm tra mệnh đề có thỏa không

|  |
| --- |
| def clause\_satisfied(clause, bits):  ... |

**Cách hoạt động**:

* Với mỗi clause (một danh sách literal), hàm này kiểm tra xem có **ít nhất một literal đúng** không.
* Nếu mệnh đề chứa biến không có trong tổ hợp (không ánh xạ được), giả định là **thỏa mãn** (không thể kiểm tra được nên coi là không vi phạm).
  1. Thử từng tổ hợp gán True/False cho các biến

|  |
| --- |
| for bits in product([False, True], repeat=len(vars\_used)): |

**Mục đích**:

* Sinh ra mọi khả năng gán (bit vector) cho các ô chưa biết.
* Tổng số tổ hợp: 2^n với n là số ô chưa biết (trong vars\_used).
  1. Kiểm tra xem tổ hợp hiện tại có thỏa mãn mọi mệnh đề CNF không

|  |
| --- |
| if all(clause\_satisfied(clause, bits) for clause in cnf.clauses): |

Nếu tổ hợp này **thỏa mãn toàn bộ mệnh đề**, thì đây là một nghiệm hợp lệ → đi đến bước tạo kết quả.

* 1. Gán giá trị 'T' hoặc 'G' lại cho các ô tương ứng

|  |
| --- |
| if idx is not None:  val = bits[idx]  row\_res.append('T' if val else 'G') |

Dựa vào kết quả của tổ hợp:

* Nếu bit tương ứng là True → đặt ô là **bẫy ('T')**
* Nếu là False → đặt ô là **đá quý ('G')**

Nếu ô không nằm trong CNF → gán mặc định (thường là 'T' hoặc có thể tùy chỉnh).

## **Cài đặt backtracking:**

* Hàm solve\_backtracking(grid, cnf) thực hiện việc **tìm nghiệm thỏa mãn CNF bằng thuật toán quay lui (backtracking)**, kết hợp với kiểm tra điều kiện an toàn của trò chơi và ràng buộc logic từ CNF.
  + 1. Khởi tạo assignment ban đầu

|  |
| --- |
| assignment = [[cell if isinstance(cell, int) else '' for cell in row] for row in grid]  empty\_cells = [(i, j) for i in range(rows) for j in range(cols) if not isinstance(grid[i][j], int)] |

Tạo bảng assignment từ grid:

* Các ô là số giữ nguyên.
* Các ô chưa biết được gán là '' (chưa có giá trị).

Tạo danh sách empty\_cells: các ô cần gán ('T' hoặc 'G').

* + 1. Tạo ánh xạ từ biến → vị trí (i,j)

|  |
| --- |
| for i in range(rows):  for j in range(cols):  v = i \* cols + j + 1  var\_to\_pos[v] = (i, j) |

Biến v là ánh xạ từ vị trí (i,j) sang biến đại diện trong CNF.  
→ Giúp kết nối giữa biến trong mệnh đề logic và tọa độ trong lưới.

* + 1. Hàm is\_safe(grid, assignment, i, j)

Hàm này **kiểm tra một bước gán có hợp lệ hay không** với lưới ban đầu, theo luật trò chơi:

* Duyệt từng ô số trong lưới.
* Đếm số bẫy 'T' xung quanh.
* Kiểm tra nếu:
  + số\_bẫy > giá\_trị\_ô hoặc
  + số\_bẫy + số\_ô\_chưa\_biết < giá\_trị\_ô  
    → Trả về False vì không thể thỏa.

Điều này giúp tránh các nhánh sai sớm.

* + 1. Hàm clause\_satisfied(clause, assignment, ...)

Kiểm tra một mệnh đề trong CNF có thỏa mãn không với trạng thái assignment hiện tại:

* Với mỗi literal trong clause, ánh xạ biến → (i,j).
* Nếu ô chưa được gán → bỏ qua.
* Nếu literal dương và gán 'T', hoặc literal âm và gán khác 'T' → mệnh đề thỏa mãn.
  + 1. Hàm is\_cnf\_satisfied(...)

Duyệt qua tất cả các mệnh đề trong cnf.clauses.

* Nếu **tất cả mệnh đề đều thỏa mãn**, trả về True.
  + 1. Hàm backtrack(...)

|  |
| --- |
| if idx == len(empty\_cells):  if is\_cnf\_satisfied(...):  return assignment  return None |

Đây là hàm chính dùng để **duyệt quay lui các tổ hợp gán 'T'/'G' cho các ô chưa biết**:

* Nếu đã gán hết empty\_cells → kiểm tra toàn bộ CNF.
* Nếu còn ô chưa gán:
  + Thử gán 'T' rồi 'G'.
  + Nếu is\_safe(...) đúng → tiếp tục quay lui sâu hơn.
  + Nếu thất bại → quay lui.
    1. Trả về kết quả

Hàm solve\_backtracking(...):

* Gọi hàm backtrack(...) từ vị trí ban đầu.
* Nếu tìm được nghiệm thỏa mãn: trả về assignment với các ô đã được điền 'T'/'G'.
* Nếu không có nghiệm: trả về None.

# **IV. Thực hiện**

1. **Matrix 5x5**

2, \_, \_, 1, \_

\_, 5, 4, 2, \_

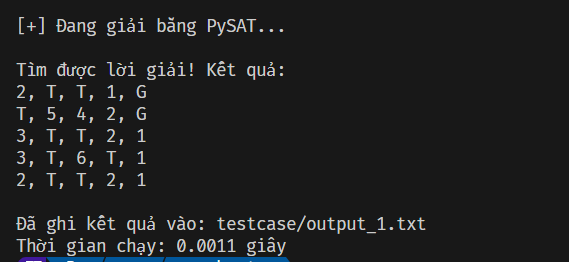
3, \_, \_, 2, 1

3, \_, 6, \_, 1

2, \_, \_, 2, 1

* + - * **Output**

**+ Pysat**

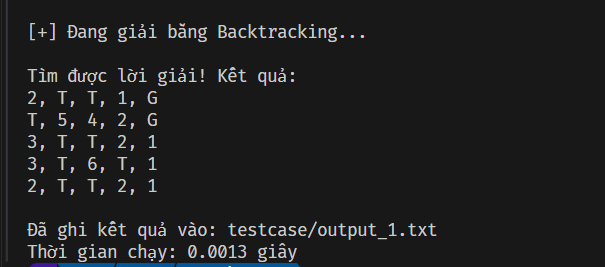
****

**+ Brute-force**

**A screen shot of a computer

AI-generated content may be incorrect.**

**+ Backtracking**

****

* + - * **Hiệu suất**

|  |  |  |
| --- | --- | --- |
| **Pysat** | **Brute force** | **Backtracking** |
| Thời gian: 0.0011 giây. Nhanh nhất trên lưới nhỏ nhờ khả năng mã hóa ràng buộc tốt và SAT solver tối ưu. | Thời gian: 0.0073 giây. Chậm nhất do phải duyệt toàn bộ tổ hợp, không có tối ưu hóa. Hoàn toàn không khả thi trên lưới lớn vì thời gian tăng theo cấp số mũ. | Thời gian: 0.0013 giây. Gần đạt hiệu suất cao nhất trên lưới nhỏ nhờ cắt tỉa nhánh hiệu quả. |

1. **Matrix 11x11**

2, \_, \_, 1, \_, \_, 2, \_, \_, 1, \_

\_, 4, \_, \_, 3, \_, \_, 2, \_, \_, 1

\_, \_, 3, \_, \_, 2, \_, \_, 3, \_, \_

1, \_, \_, 3, \_, \_, 2, \_, \_, 2, 1

\_, 2, \_, \_, 4, \_, \_, 3, \_, \_, 2

\_, \_, 3, \_, \_, 4, \_, \_, 3, \_, \_

1, 2, \_, \_, 3, \_, \_, 4, \_, \_, 2

\_, \_, 2, \_, \_, 3, \_, \_, 2, \_, \_

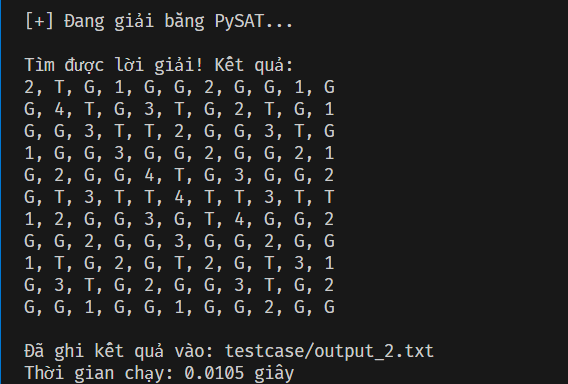
1, \_, \_, 2, \_, \_, 2, \_, \_, 3, 1

\_, 3, \_, \_, 2, \_, \_, 3, \_, \_, 2

\_, \_, 1, \_, \_, 1, \_, \_, 2, \_, \_

* + - * **Output**

**+ Pysat**

****

**+ Brute-force**

**+ Backtracking**

**A screen shot of a computer

AI-generated content may be incorrect.**

* + - * **Biệu suất**

|  |  |  |
| --- | --- | --- |
| **Pysat** | **Brute force** | **Backtracking** |
| Thời gian: 0.0105 giây. Nhanh nhất trên lưới 11x11 nhờ khả năng mã hóa ràng buộc tốt và SAT solver tối ưu. | Không áp dụng được vì có tới 2⁷⁸ tổ hợp — vượt quá khả năng xử lý. | Thời gian: 0.0266 giây. Gần đạt hiệu suất cao nhất trên lưới nhỏ nhờ cắt tỉa nhánh hiệu quả. Tuy nhiên, hiệu suất sẽ giảm nếu số ô chưa biết tăng nhiều. |

1. **Matrix 20x20**

\_, \_, 2, \_, \_, \_, 3, \_, \_, \_, 2, \_, \_, 1, \_, \_, \_, 2, \_, \_

\_, 3, \_, \_, 2, \_, \_, \_, \_, 3, \_, \_, \_, \_, 2, \_, \_, \_, \_, 2

\_, \_, \_, 2, \_, \_, \_, 3, \_, \_, \_, 2, \_, \_, \_, \_, \_, 2, \_, \_

2, \_, \_, \_, \_, 3, \_, \_, \_, 2, \_, \_, 2, \_, \_, \_, 3, \_, \_, \_

\_, \_, 2, \_, \_, \_, \_, 2, \_, \_, \_, \_, 3, \_, \_, 2, \_, \_, 3, \_

\_, \_, \_, 3, \_, \_, \_, \_, 2, \_, \_, \_, \_, 2, \_, \_, \_, 3, \_, \_

\_, 2, \_, \_, \_, 2, \_, \_, \_, 3, \_, \_, \_, \_, \_, 3, \_, \_, \_, \_

3, \_, \_, \_, \_, \_, \_, 2, \_, \_, \_, 3, \_, \_, \_, \_, \_, 2, \_, \_

\_, \_, 3, \_, \_, 2, \_, \_, \_, \_, \_, \_, 2, \_, \_, 2, \_, \_, 3, \_

\_, \_, \_, 2, \_, \_, \_, 3, \_, \_, \_, \_, \_, 3, \_, \_, \_, 2, \_, \_

2, \_, \_, \_, \_, 3, \_, \_, 2, \_, \_, \_, 2, \_, \_, \_, \_, \_, \_, 2

\_, 3, \_, \_, \_, \_, 2, \_, \_, \_, 3, \_, \_, \_, \_, \_, 3, \_, \_, \_

\_, \_, \_, 3, \_, \_, \_, \_, \_, 2, \_, \_, \_, \_, 2, \_, \_, \_, 2, \_

\_, 2, \_, \_, \_, 2, \_, \_, \_, \_, \_, \_, 3, \_, \_, \_, \_, \_, \_, \_

3, \_, \_, \_, \_, \_, \_, 2, \_, 2, \_, \_, \_, 2, \_, \_, \_, 3, \_, \_

\_, \_, 3, \_, \_, \_, \_, 3, \_, \_, \_, 2, \_, \_, \_, 2, \_, \_, \_, \_

\_, \_, \_, 2, \_, \_, \_, \_, 2, \_, \_, \_, \_, 3, \_, \_, \_, \_, 3, \_

\_, 2, \_, \_, \_, 3, \_, \_, \_, \_, \_, \_, 2, \_, \_, 2, \_, \_, \_, \_

\_, \_, \_, 4, \_, \_, \_, \_, \_, 2, \_, \_, \_, \_, 3, \_, \_, \_, \_, \_

\_, 2, \_, \_, \_, 2, \_, 3, \_, \_, \_, 2, \_, \_, \_, \_, \_, \_, 3, \_

* + - * **Output**

**+ Pysat**

**A screen shot of a computer

AI-generated content may be incorrect.**

**+ Brute-force**

**+ Backtracking**

**A screen shot of a computer

AI-generated content may be incorrect.**

* + - * **Hiệu suất**

|  |  |  |
| --- | --- | --- |
| **Pysat** | **Brute force** | **Backtracking** |
| Thời gian: 0.0109 giây.  Nhanh và ổn định nhờ mã hóa ràng buộc hiệu quả và sử dụng SAT solver mạnh. | Không khả thi được do không gian tìm kiếm quá lớn (2^400 tổ hợp). | Thời gian: 9.4839 giây.  Có thể giải được nhưng rất chậm; hiệu suất giảm mạnh do số lượng ô chưa biết lớn và cây tìm kiếm rộng. |

# **Kết luận**

Brute Force: Hiệu quả thấp, không khả thi cho lưới lớn do thời gian chạy quá dài (O(kn)O(k^n)O(kn)).

Backtracking: Hiệu quả khá, khả thi cho lưới vừa, nhưng chậm với lưới lớn (vài phút đến vài giờ).

PySAT: Hiệu quả cao nhất, nhanh (vài giây đến vài phút), phù hợp lưới lớn, dù cần mã hóa phức tạp.

=> Dùng PySAT cho hiệu suất tối ưu cho đồ án hơn, Backtracking cho lưới nhỏ và đơn giản, tránh Brute Force.

# **Đánh giá mức độ hoàn thành**

|  |  |
| --- | --- |
| Công việc | Mức độ hoàn thành |
| Mô tả nguyên lý logic | 100% |
| Tạo CNF tự động | 100% |
| Sử dụng thư viện Pysat để giải CNFs | 100% |
| Triền khai brute-force | 100% |
| Triền khai backtracking | 100% |
| Báo cáo và phân tích | 100% |

# **VI. Link demo**

# **VII. Tham khảo**