ĐẠI HỌC QUỐC GIA
THÀNH PHỐ HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

CSC10003: CƠ SỞ TRÍ TUỆ NHÂN TẠO

Báo cáo đồ án 2
Hashiwokakero

Họ và tên sinh viên	**MSSV**
Nguyễn Tấn Huy Khôi	23127071
Nguyễn Hoàng Gia Bảo	23127157
Lương Linh Khôi	23127396
Nguyễn Huỳnh Tiến	23127494

Giảng viên hướng dẫn

Bùi Tiến Lên Lê Nhựt Nam Võ Nhật Tân

TP. Hồ Chí Minh, 8/2025

Mục lục

1 Thông tin chung

1.1 Thông tin thành viên

MSSV	Họ và tên	Email
23127071	Nguyễn Tấn Huy Khôi	nthkhoi23@clc.fitus.edu.vn
23127157	Nguyễn Hoàng Gia Bảo	nhgbao23@clc.fitus.edu.vn
23127396	Lương Linh Khôi	llkhoi23@clc.fitus.edu.vn
23127494	Nguyễn Huỳnh Tiến	nhtien23@clc.fitus.edu.vn

Bảng 1: Bảng thông tin thành viên

1.2 Bảng phân chia công việc

Công việc	Thành viên thực hiện	Trạng thái	%
• Mô tả bài toán. • Mô tả chi tiết giải pháp cho việc cài đặt và thiết lập các CNF từ bài toán ban đầu (có ví dụ minh họa). • Vẽ flowchart của giải pháp. • Tài liệu tham khảo. • Tổng hợp nội dung viết báo cáo bố cục hoàn chỉnh, định dạng đồng bộ.	Nguyễn Tấn Huy Khôi	Hoàn thành	100%
• Tạo tự động các CNF tương ứng với các input. • Dùng thư viện PySAT để giải các CNF. • Viết thuật toán A* để giải CNF mà không dùng thư viện. • Viết report nêu ý tưởng thực hiện, giải thích chi tiết thuật toán, giải thích heuristic, các hàm bổ trợ (nếu có).	Nguyễn Hoàng Gia Bảo	Hoàn thành	100%

• Viết thuật toán Brute-force và Backtracking để giải CNF (có đo thời gian xử lý). • Viết report nêu ý tưởng thực hiện, giải thích chi tiết thuật toán, giải thích các hàm bổ trợ (nếu có). • Thực hiện video demo.	Nguyễn Huỳnh Tiến	Hoàn thành	100%
• Cung cấp 10 test case (có đủ loại 7×7, 9×9, 11×11, 13×13, 17×17, 20×20). • Viết báo cáo thực nghiệm các thuật toán giải CNF trên mỗi loại test case khác nhau (7×7, 9×9, 11×11, 13×13, 17×17, 20×20). • Đánh giá và so sánh các thuật toán. • Viết phần mở rộng (hạn chế và cải tiến).	Lương Linh Khôi	Hoàn thành	100%

Bảng 2: Bảng phân chia công việc

1.3 Video demo và mã nguồn

- Video demo
- Source code

2 Tổng quan về đồ án

2.1 Mô tả đồ án

Hashiwokakero là một trò chơi giải đố trên một ma trận hình chữ nhật chứa các ô được gọi là các "đảo", mỗi đảo được đánh trọng số từ 1 đến 8. Mục tiêu của người chơi là xây dựng hệ thống các "cầu" kết nối các đảo sao cho thoả mãn những quy tắc sau:

- Cầu phải nối hai đảo khác nhau.
- Cầu chỉ được xây theo chiều ngang hoặc dọc.
- Cầu không được cắt ngang bất kỳ cây cầu hay đảo nào khác.
- Giữa mỗi cặp đảo chỉ được xây tối đa 2 cầu.

- Tổng số cầu nối đến một đảo phải bằng đúng trọng số của đảo.
- Tất cả các đảo phải được nối với nhau thành một mạng liên thông duy nhất, tức là có thể di chuyển giữa hai đảo bất kỳ thông qua các cầu.

Trong đồ án này, ta cần cài đặt một chương trình để giải bài toán Hashiwokakero sử dụng Conjunctive Normal Form (CNF) logic. Các bước được yêu cầu thực hiện như sau:

1. Định nghĩa các biến logic cho bài toán.
2. Xây dựng các ràng buộc CNF.
3. Tự động sinh các mệnh đề CNF dựa vào input.
4. Sử dụng thư viện PySAT để giải bài toán.
5. Cài đặt thuật toán tìm kiếm A* để giải bài toán.
6. Cài đặt thuật toán Brute-force và Backtracking để so sánh tốc độ thực thi (thời gian chạy) và hiệu suất với A*.

2.2 Tự đánh giá về kết quả đồ án

STT	**Tiêu chí**	**Tỷ lệ**
1	Mô tả lời giải: Trình bày chính xác logic dùng để sinh công thức CNF từ bài toán Hashiwokakero.	30%
2	Tự động sinh công thức CNF.	10%
3	Sử dụng thư viện PySAT để giải CNF.	10%
4	Cài đặt thuật toán A* để giải CNF mà không sử dụng thư viện giải SAT có sẵn.	10%
5	Cài đặt các thuật toán bổ sung: 1) Thuật toán Brute-force; 2) Thuật toán Backtracking để so sánh tốc độ với A*.	10%
6	Tài liệu và phân tích: 1) Viết báo cáo chi tiết (30%); 2) Phân tích và thực nghiệm kỹ lưỡng; 3) Cung cấp ít nhất 10 bộ test với các kích thước khác nhau (7×7, 9×9, 11×11, 13×13, 17×17, 20×20) để kiểm tra giải pháp; 4) So sánh kết quả và hiệu suất.	30%
Tổng cộng		**100%**

Bảng 3: Bảng tự đánh giá theo tiêu chí

3 Xây dựng CNF

3.1 Mô hình hoá bài toán

Mục tiêu của trò chơi này là tìm một cách đặt các cây cầu sao cho thoả mãn tất cả các quy tắc của trò chơi. Ta có thể chuyển thành một bài toán thoả mãn ràng buộc (**CSP**) như sau: "Với mỗi vị trí có thể xây cầu, cần xác định số lượng cầu được xây sao cho tất cả các ràng buộc của trò chơi được thoả mãn". Khi đó:

- **Biến:** Là tất cả các vị trí có thể xây cầu.
- **Miền giá trị:** Là số lượng cầu được xây ở mỗi vị trí.
- **Ràng buộc:** Là tất cả các quy tắc của trò chơi.

Vì yêu cầu của đồ án, ta có thể chuyển đổi bài toán CSP sang dạng mệnh đề logic CNF để giải như một bài toán **SAT**. Khi này, mỗi biến logic sẽ biểu diễn cho việc có hay không có cầu giữa hai đảo. Các quy tắc của trò chơi sẽ được biểu diễn thành các mệnh đề logic. Từ các mệnh đề logic này, ta sẽ dùng một thuật toán (SAT solver) để gán giá trị cho các biến sao cho nếu tồn tại một phép gán thoả mãn tất cả các mệnh đề thì đó chính là lời giải hợp lệ cho bài toán. Do đó, việc quan trọng là biểu diễn được các quy tắc của trò chơi thành mệnh đề logic CNF.

3.2 Định nghĩa biến logic

Ta gọi V_{ij} là biến logic biểu diễn mệnh đề "Có cầu nối giữa đảo i và đảo j". Tuy nhiên, biểu diễn như vậy thì chưa đầy đủ để giải bài toán, vì cần phải xét thêm số lượng cầu giữa hai đảo (1 hoặc 2 cầu). Do đó, ta tách làm hai biến riêng biệt:

- V_{1ij}: "Có 1 cầu nối giữa đảo i và đảo j".
- V_{2ij}: "Có 2 cầu nối giữa đảo i và đảo j".

Ta chỉ xây dựng các biến logic cho các cặp đảo có thể nối với nhau. Hai đảo i và j có thể xây cầu nối (hay kề nhau) nếu chúng nằm trên cùng một dòng hoặc cột, và giữa chúng không có một đảo nào khác.

Áp dụng hai loại biến logic trên cho mỗi cặp đảo (i, j) kề nhau là đủ để xây dựng các mệnh đề CNF.

3.3 Xây dựng các mệnh đề CNF

Ta sẽ dựa vào các ràng buộc cần thoả mãn trong trò chơi để xây dựng các mệnh đề logic. Tuy nhiên, có một số ràng buộc là hiển nhiên và không cần biểu diễn logic là:

- Cầu nối hai đảo khác nhau.
- Cầu xây theo chiều ngang hoặc dọc.
- Cầu không được cắt đảo.

vì các điều kiện trên đã được kiểm tra và loại bỏ trước khi sinh ra các biến logic. Ta chỉ xem xét xây dựng mệnh đề logic cho các ràng buộc còn lại.

3.3.1 Ràng buộc số cầu giữa mỗi cặp đảo tối đa là 2

Cách định nghĩa biến logic đã bao hàm việc có 0, 1, hoặc 2 cầu. Tuy nhiên, ta phải đảm bảo rằng không thể có đồng thời 1 cầu và 2 cầu. Cho nên với mỗi cặp đảo (i, j) kề nhau, ta có được mệnh đề sau:

$$\neg(V_{1ij} \wedge V_{2ij})$$

Chuyển sang dạng CNF (luật De Morgan):

$$\neg V_{1ij} \vee \neg V_{2ij} \tag{1}$$

Với công thức trên, ta đảm bảo số lượng cầu nối giữa hai đảo không vượt quá 2.

3.3.2 Ràng buộc tổng số cầu nối đến một đảo đúng bằng trọng số của đảo

Giả sử đảo i có trọng số w_i và có thể nối đến n đảo khác là $j_1, j_2, ..., j_n$. Khi đó phải thoả mãn:

$$\sum_{k=1}^{n} (V_{1ij_k} + 2V_{2ij_k}) = w_i$$

(1 khi $V = True$, 0 khi $V = False$).

Xây dựng mệnh đề logic cho điều kiện như trên có vẻ phức tạp vì logic không trực tiếp biểu diễn phép cộng hoặc đếm. Do đó, một cách tiếp cận khác là liệt kê tất cả các trường hợp khả thi sao cho tổng số cầu bằng trọng số.

Trước tiên, ta cần xác định danh sách các biến logic là tất cả các cầu mà từ các đảo kề có thể nối đến đảo i đang xét: Với mỗi đảo kề j, ta xét số lượng cầu có thể xây đến đảo i. Nếu trọng số của đảo j bằng 1 thì chỉ có thể xây 1 cầu (V_{1ij}), nếu trọng số của đảo j lớn hơn 1 thì có thể xây 1 cầu (V_{1ij}) hoặc 2 cầu (V_{2ij}).

Sau khi có được danh sách các biến logic tương ứng với các cầu khả thi, ta sẽ liệt kê tất cả các tổ hợp con của danh sách này sao cho tổng số cầu trong mỗi tổ hợp đúng bằng trọng số w_i. Mỗi tổ hợp như vậy đại diện cho một trường hợp xây cầu hợp lệ, và ta sẽ xây dựng các mệnh đề logic để biểu diễn rằng một trong các tổ hợp hợp lệ đó phải xảy ra:

Giả sử một tổ hợp con hợp lệ là $\mathbf{B} = \{B_1, B_2, ..., B_m\}$ gồm các biến logic biểu diễn các cầu khả thi được chọn. Để tổ hợp này xảy ra thì các biến phải đồng thời đúng. Do đó, ta định nghĩa một biến tổ hợp $\mathbf{T}$ sao cho:

$$\begin{aligned}\mathbf{T} &\equiv \bigwedge \mathbf{B} \\ &\equiv B_1 \wedge B_2 \wedge ... \wedge B_m\end{aligned}$$

Nếu tồn tại n tổ hợp con hợp lệ $\mathbf{T}_1, \mathbf{T}_2, ..., \mathbf{T}_n$ thì ta cần chính xác một trong số đó phải xảy ra, tức là:

1. Ít nhất một tổ hợp xảy ra:

$$\mathbf{T}_1 \vee \mathbf{T}_2 \vee ... \vee \mathbf{T}_n \equiv \left(\bigwedge \mathbf{B}_1\right) \vee \left(\bigwedge \mathbf{B}_2\right) \vee ... \vee \left(\bigwedge \mathbf{B}_n\right) \tag{2}$$

2. Không có hai tổ hợp nào xảy ra đồng thời:

$$\begin{aligned}&\forall i \neq j, \neg\mathbf{T}_i \vee \neg\mathbf{T}_j \\ &\quad\equiv \neg\left(\bigwedge \mathbf{B}_i\right) \vee \neg\left(\bigwedge \mathbf{B}_j\right) \\ &\quad\equiv \left(\bigvee \neg\mathbf{B}_i\right) \vee \left(\bigvee \neg\mathbf{B}_j\right)\end{aligned} \tag{3}$$

Như vậy, với mỗi đảo i, ta có được:

- 1 mệnh đề từ công thức (2).
- $\binom{n}{2}$ mệnh đề từ công thức (3). Trong đó, n là số tổ hợp con hợp lệ.

Ta thấy công thức (3) đã đúng dạng CNF nhưng công thức (2) thì chưa. Do đó, ta cần biến đổi nó về đúng dạng, việc này sẽ sinh thêm nhiều mệnh đề khác.

Tương tự, ta áp dụng các bước trên để xây dựng các mệnh đề của ràng buộc này cho tất cả các đảo còn lại.

3.3.3 Ràng buộc các cầu không cắt nhau

Giả sử một cầu ngang (i, j) và một cầu dọc (p, q) cắt nhau thì hai cầu này không thể cùng tồn tại, tức là:

$$\neg(V_{1ij} \vee V_{2ij}) \vee \neg(V_{1pq} \vee V_{2pq})$$

Qua vài phép biến đổi, ta được:

$$(\neg V_{1ij} \vee \neg V_{1pq}) \wedge (\neg V_{1ij} \vee \neg V_{2pq}) \wedge (\neg V_{2ij} \vee \neg V_{1pq}) \wedge (\neg V_{2ij} \vee \neg V_{2pq}) \tag{4}$$

Công thức trên đảm bảo rằng nếu một trong bốn loại cầu tồn tại thì loại cầu của cặp kia không thể tồn tại. Do đó ở ràng buộc này, với mỗi cặp $((i, j), (p, q))$ cắt nhau, ta sinh được 4 mệnh đề từ công thức (4).

3.3.4 Ràng buộc về liên thông

Trò chơi yêu cầu rằng phải tồn tại một đường đi giữa mọi cặp đảo, tức là đồ thị được hình thành bởi các cầu phải là một đồ thị liên thông. Điều này đòi hỏi kiểm tra toàn bộ cấu trúc đồ thị, khi diễn đạt bằng các mệnh đề logic sẽ rất phức tạp và làm tăng đáng kể chi phí.

Do đó, để đơn giản hơn, ta sẽ kiểm tra riêng biệt ràng buộc này sau khi tìm được một lời giải thỏa mãn các mệnh đề logic còn lại. Nếu đồ thị liên thông thì đó là một lời giải hợp lệ. Ngược lại nếu không liên thông, ta sẽ thêm phủ định của nghiệm vào công thức logic và giải lại đến khi nào tìm được một lời giải liên thông (xem hình 3).

3.3.5 Các ràng buộc bổ sung đặc biệt

Bên cạnh các ràng buộc áp dụng chung cho mọi đảo, ta thêm một số ràng buộc xử lý cho các trường hợp đặc biệt để rút ngắn thời gian giải.

- **Không nối hai đảo có trọng số bằng 1:** Nếu hai đảo i, j đều có trọng số bằng 1 và được nối với nhau thì mỗi đảo đã dùng hết số cầu cho phép và không thể nối tiếp với các đảo khác, vi phạm điều kiện liên thông. Do đó:

$$\neg V_{1ij} \wedge \neg V_{2ij} \tag{5}$$

- **Không đặt cầu đôi giữa hai đảo có trọng số bằng 2:** Tương tự, nếu hai đảo i, j đều có trọng số bằng 2 thì không được xây 2 cầu vì khi đó cũng vi phạm điều kiện liên thông (vẫn có thể xây 1 cầu):

$$\neg V_{2ij} \tag{6}$$

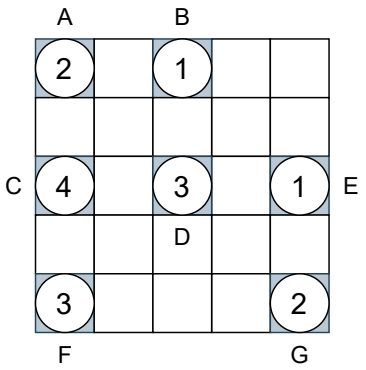
- **Đặt 4 cầu đôi ở đảo có trọng số bằng 8:** Vì mỗi hướng (trên, dưới, trái, phải) chỉ cho xây tối đa 2 cầu, đảo có trọng số bằng 8 buộc phải có 4 cầu đôi nối với các đảo xung quanh để tổng là 8. Giả sử đảo i có trọng số $w_i = 8$ và kề 4 đảo j_1, j_2, j_3, j_4 ở mỗi hướng thì:

$$V_{2ij_1} \wedge V_{2ij_2} \wedge V_{2ij_3} \wedge V_{2ij_4} \tag{7}$$

3.3.6 Tổng hợp các điều kiện

Sau khi có được các mệnh đề logic dạng CNF từ (1) đến (7), ta kết hợp tất cả bằng phép AND ($\wedge$) để được công thức logic. Sau đó ta đưa công thức vào thuật toán để tìm một phép gán thoả mãn.

3.4 Ví dụ minh hoạ



Hình 1: Ma trận trò chơi ban đầu

Ta sẽ xây dựng các mệnh đề logic cho trường hợp trên:

- **Ràng buộc số cầu giữa mỗi cặp đảo tối đa là 2:** (công thức (1))
  - $\neg V_{1AB} \vee \neg V_{2AB}$
  - $\neg V_{1AC} \vee \neg V_{2AC}$
  - $\neg V_{1BD} \vee \neg V_{2BD}$
  - $\neg V_{1CD} \vee \neg V_{2CD}$
  - $\neg V_{1CF} \vee \neg V_{2CF}$
  - $\neg V_{1DE} \vee \neg V_{2DE}$
  - $\neg V_{1EG} \vee \neg V_{2EG}$
  - $\neg V_{1FG} \vee \neg V_{2FG}$
- **Ràng buộc tổng số cầu nối đến một đảo đúng bằng trọng số của đảo:**
  Xét đảo D, ta có danh sách các cầu khả thi có thể nối đến D là: $\{V_{1DB}, V_{1DC}, V_{2DC}, V_{1DE}\}$.
  Các tổ hợp con của danh sách mà tổng số cầu bằng $w_D = 3$ là: $\{(V_{1DB} \wedge V_{1DC} \wedge V_{1DE}), (V_{1DB} \wedge V_{2DC}), (V_{1DE} \wedge V_{2DC})\}$.
  Từ công thức (2), ta có:
  - $(V_{1DB} \wedge V_{1DC} \wedge V_{1DE}) \vee (V_{1DB} \wedge V_{2DC}) \vee (V_{1DE} \wedge V_{2DC}) \equiv ... \equiv \text{CNF}$

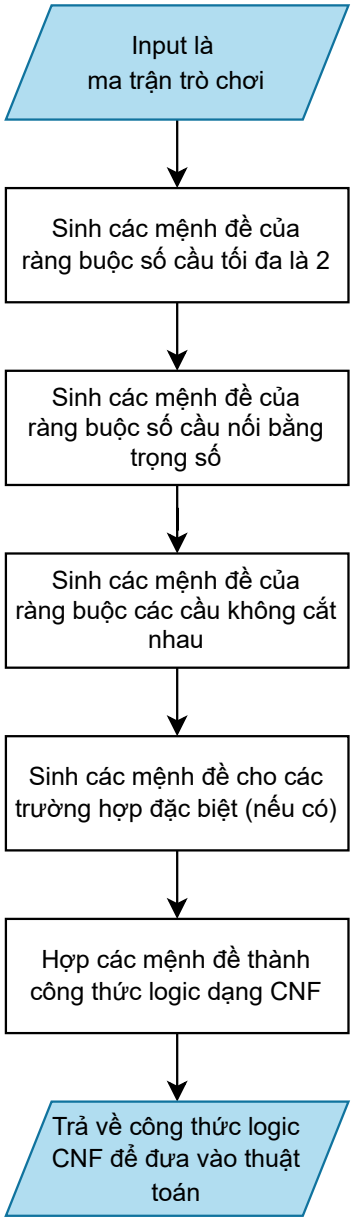
  Từ công thức (3), ta có:
  - $(\neg V_{1DB} \vee \neg V_{1DC} \vee \neg V_{1DE}) \vee (\neg V_{1DB} \vee \neg V_{2DC})$
  - $(\neg V_{1DB} \vee \neg V_{1DC} \vee \neg V_{1DE}) \vee (\neg V_{1DE} \vee \neg V_{2DC})$
  - $(\neg V_{1DB} \vee \neg V_{2DC}) \vee (\neg V_{1DE} \vee \neg V_{2DC})$

  Tương tự cho các đảo còn lại.
- **Ràng buộc các cầu không cắt nhau:** Không có.
- **Các ràng buộc bổ sung đặc biệt:** Không có.

Khi đã có đầy đủ mệnh đề, ta kết hợp tất cả bằng phép AND ($\wedge$) để được công thức logic, từ đó đưa vào thuật toán để giải.

3.5 Flowchart

Hàm `generateCNF()`:

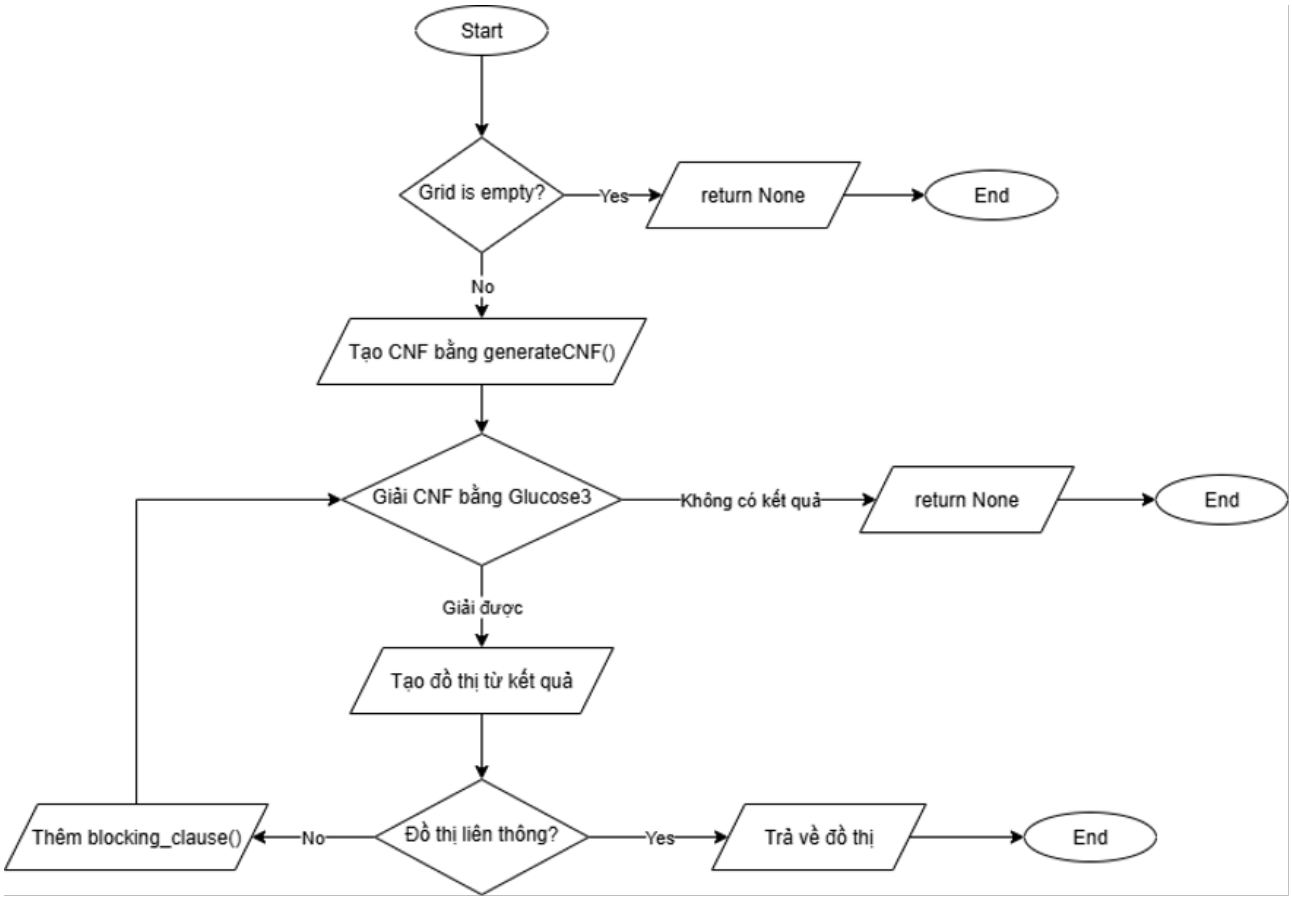


Hình 2: Flowchart hàm tạo CNF

4 Sử dụng thư viện PySAT để giải CNF

Hàm `solve_pysat(grid)`:

- Nhận vào đồ thị từ file input đã được xử lí qua hàm `read_input()`.
- Gọi `generateCNF()` để tự động tạo CNF từ đồ thị nhận vào.
- Dùng SAT Solver là Glucose3 để giải CNF.
- Tạo đồ thị từ kết quả.
- Kiểm tra tính liên thông của kết quả và trả về đồ thị.



Hình 3: Flowchart cài đặt với PySAT

5 Các thuật toán giải CNF

5.1 Thuật toán A*

Hàm `solve_cnf_astar()` được viết nhằm sử dụng thay thế như một SAT Solver, áp dụng thuật toán A* search để giải các CNF sinh ra từ bài toán Hashiwokakero ban đầu.

5.1.1 Mô tả cơ bản

- Không gian trạng thái: Tập hợp tất cả các tổ hợp giá trị cho các biến trong CNF, mỗi tổ hợp giá trị là một trạng thái. Mỗi trạng thái gồm:
  - assignment: các biến đã được gán giá trị.
  - clauses: các mệnh đề chưa đúng với các biến đã được gán giá trị (sau khi lan truyền bằng Unit propagation - loại bỏ mệnh đề đơn và Pure literal elimination - loại bỏ literal thuần ở mục 5.1.3).
- Trạng thái bắt đầu: Tập assignment trống và clauses là CNF sinh ra từ hàm `generateCNF()` sau khi xử lí qua UP và PL.
- Trạng thái kết thúc: Sau khi tập clauses rỗng, tức là đã thỏa mãn tất cả các mệnh đề.
- Hàm chuyển đổi trạng thái:

$$\text{Transition}(\text{assignment}, \text{gán } x_i = v) \rightarrow \text{assignment mới, clauses mới.}$$

- Để mở rộng không gian trạng thái ở mỗi node, thực hiện:
  - Chọn một biến chưa gán giá trị.
  - Thực hiện lần lượt gán biến đó bằng True và False để mở rộng
  - Cập nhật clauses sau khi gán giá trị biến mới và lan truyền và kiểm tra mâu thuẫn.
  - Tạo ra trạng thái mới, tính $f = g + h$ và đưa vào hàng đợi ưu tiên.

5.1.2 Hàm heuristic

Hàm heuristic được thiết kế để ước lượng số bước cần thực hiện để đến trạng thái kết thúc. Trong bài toán này, hàm được tính một cách đơn giản bằng cách cộng hai giá trị:

- Số biến chưa gán (ước lượng chi phí đến đích).
- Số mệnh đề chưa thỏa mãn với giá trị đã gán của các biến hiện tại (ước lượng mức độ vi phạm ràng buộc để tránh các nhánh sai).

$$h(n) = \text{unassigned\_var} + \text{clause\_penalty}$$

Heuristic trên khá đơn giản và đạt hiệu quả tốt trên các đồ thị không quá lớn trong các test case của đồ án này. Tuy nhiên có thể gặp khó khăn trong các đồ thị lớn vì:

- Không ước lượng đến các giá trị khác bên cạnh như số literal còn lại trong các mệnh đề vi phạm (số "cơ hội" để khắc phục mệnh đề).
- Chưa phân biệt mức độ quan trọng của 2 thành phần unassigned_var, clause_penalty. Ví dụ: "một trạng thái gán ít biến hơn nhưng chưa sai mệnh đề nào" (tiềm năng) nên được ưu tiên hơn "một trạng thái gán nhiều biến nhưng có nhiều mệnh đề sai" (dễ thất bại). Do đó clause_penalty nên được đặt nặng hơn so với unassigned_var.

Từ đó một số cải tiến có thể cần áp dụng cho heuristic trên là:

- Đánh giá về số literal còn lại trong các mệnh đề sai.
- Thêm trọng số cho clause_penalty hoặc dùng $(\text{clause\_penalty})^2$ nếu cần thiết.

5.1.3 Lan truyền ràng buộc

Nhằm giảm bớt không gian trạng thái, tăng hiệu suất của thuật toán, ta áp dụng thêm các phương pháp lan truyền ràng buộc:

- **Unit propagation:** Thực hiện tìm và gán giá trị phù hợp cho các mệnh đề đơn (chỉ gồm 1 literal), làm đơn giản danh sách mệnh đề bằng cách loại các mệnh đề chứa literal trên (đã đúng sai khi gán giá trị cho literal đó), loại ¬literal ra khỏi các mệnh đề chứa nó.
- **Pure literal elimination:** Tìm các literal thuần (chỉ có 1 dạng lit hoặc ¬lit trong tất cả các mệnh đề) và gán giá trị phù hợp cho literal thuần. Sau khi gán giá trị các mệnh đề chứa literal này đã thỏa mãn nên loại các mệnh đề này ra khỏi danh sách để làm đơn giản danh sách mệnh đề.

5.1.4 Áp dụng giải Hashiwokakero

Vì thuật toán A* như giải thích ở trên được thiết kế để thay thế một SAT Solver nên hàm `solve_astar()` dùng để giải Hashiwokakero được viết giống như khi dùng thư viện PySAT ở mục 4, chỉ thay dùng SAT Solver Glucose3 bằng hàm `solve_cnf_astar()`.

5.2 Thuật toán Backtracking

5.2.1 Ý tưởng chính

Thuật toán backtracking được sử dụng để giải các câu đố Hashiwokakero dựa trên một phiên bản đơn giản hóa của bộ giải SAT DPLL (Davis–Putnam–Logemann–Loveland). Trước

tiên, câu đố được mã hóa thành dạng CNF, sau đó một quá trình tìm kiếm đệ quy được thực hiện để tìm một phép gán các biến logic thỏa mãn. Áp dụng các suy luận rút gọn mệnh đề đơn và loại biến thuần như ở mục 5.1.3 để loại bỏ sớm các nhánh không hợp lệ.

5.2.2 Giải thích chi tiết

- `unit_propagate(clauses, assignment)`:
  - Hàm này thực hiện suy luận từ các mệnh đề đơn một cách lặp lại.
  - Khi một mệnh đề chỉ còn một biến chưa được gán, biến đó bắt buộc phải nhận giá trị phù hợp để mệnh đề được thỏa mãn.
  - Việc này giúp giảm không gian tìm kiếm và phát hiện mâu thuẫn sớm trong quá trình giải.
- `dpll(clauses, assignment)`:
  - Hàm đệ quy chính của bộ giải SAT.
  - Hàm áp dụng unit propagation và loại bỏ biến thuần để đơn giản hóa công thức trước khi phân nhánh.
  - Với mỗi biến chưa được gán, hàm thử gán cả hai giá trị True và False, đệ quy tiếp tục cho đến khi tìm được một nghiệm thỏa mãn hoặc xác định rằng công thức không thể thỏa mãn.
- `solve_cnf_backtracking(data)`:
  - Hàm này nhận dữ liệu CNF từ `generateCNF()`, sau đó gọi bộ giải `dpll()` để tìm một nghiệm thỏa mãn.
  - Nếu tìm được nghiệm, nó trích xuất các cạnh đang được chọn và kiểm tra xem các đảo có được kết nối không bằng hàm `is_connected()`.
  - Nếu chưa kết nối, hàm thêm một mệnh đề chặn (blocking clause) để loại nghiệm hiện tại và tiếp tục tìm nghiệm khác.
- `solve_backtracking()`:
  - Hàm điều phối tổng thể quá trình giải.
  - Đầu tiên, nó kiểm tra đầu vào grid, sau đó gọi `generateCNF()` để chuyển đổi câu đố thành CNF.
  - Tiếp theo, nó gọi `solve_cnf_backtracking()` để tìm lời giải. Nếu có nghiệm hợp lệ, hàm sẽ in thời gian thực thi và trả về danh sách cạnh là lời giải.

5.3 Thuật toán Brute-force

5.3.1 Ý tưởng chính

Thuật toán Brute-force này thử tất cả các khả năng gán giá trị cho các biến logic. Với mỗi tổ hợp, nó kiểm tra: Có thỏa mãn toàn bộ ràng buộc logic trong CNF không? Có kết nối được tất cả các đảo không? Nếu cả hai điều kiện đều đúng, tổ hợp đó là lời giải.

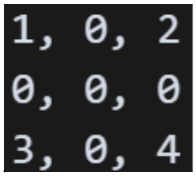
5.3.2 Giải thích chi tiết

- `solve_cnf_bruteforce()`:
  - Hàm này thực hiện thuật toán duyệt toàn bộ để tìm nghiệm cho CNF.
  - Nó sinh ra tất cả các tổ hợp gán giá trị True/False cho các biến, kiểm tra từng tổ hợp xem có thỏa mãn toàn bộ các mệnh đề CNF hay không.
  - Nếu một tổ hợp thỏa mãn được tất cả các mệnh đề, hàm sẽ trích xuất các cạnh tương ứng và kiểm tra tính kết nối giữa các đảo bằng `is_connected()`.
  - Nếu đạt yêu cầu, trả về lời giải. Ngược lại, tiếp tục thử các tổ hợp khác cho đến khi không còn tổ hợp nào.
- `solve_bruteforce()`:
  - Hàm bao ngoài để điều phối quá trình giải bằng Brute-force.
  - Đầu tiên, nó kiểm tra tính hợp lệ của lưới đầu vào (`grid`), sau đó gọi `generateCNF()` để chuyển lưới thành CNF và dữ liệu liên quan.
  - Tiếp theo, gọi `solve_cnf_bruteforce()` để tìm lời giải. Nếu có lời giải hợp lệ, in ra thời gian thực thi và trả kết quả.

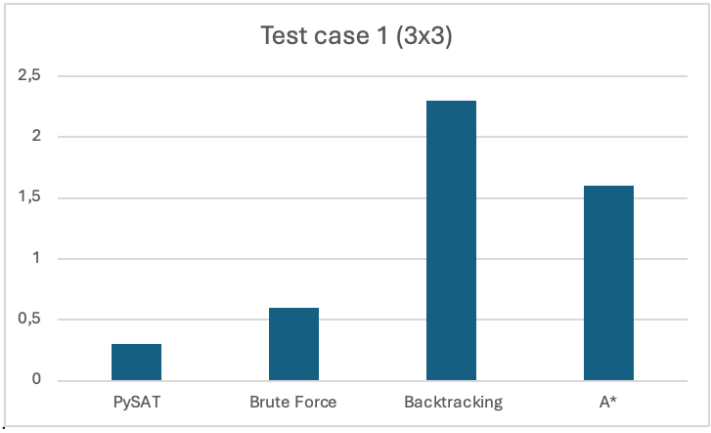
6 Phân tích kết quả thực nghiệm

6.1 Các test case

- **Test case 1:**



Kích cỡ: 3×3.
Số lượng đảo: 4.



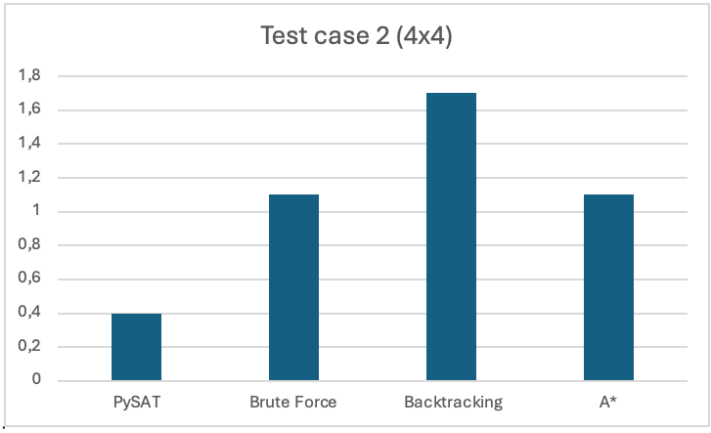
Nhận xét:

- Trong test case có số lượng biến mệnh đề nhỏ như 3×3 thì Brute-force có hiệu năng tốt (0.6ms), đặc biệt là sự cách biệt thời gian xử lý so với khi sử dụng thư viện PySAT là không quá lớn (0.3ms).
- Các hàm Backtracking và A* vì tốn chi phí cho việc xử lý logic phức tạp hơn nên có hiệu năng kém hơn so với Brute-force.

- **Test case 2:**

```
4, 0, 0, 3
0, 0, 0, 0
4, 0, 2, 0
0, 0, 0, 1
```

Kích cỡ: 4×4.
Số lượng đảo: 5.



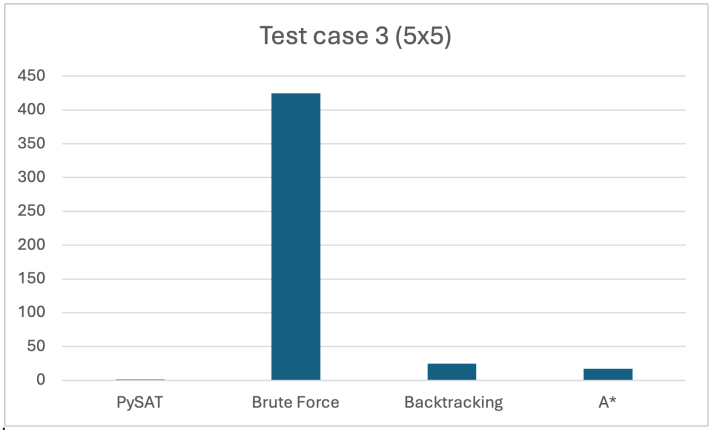
Nhận xét:

- Mặc dù số đảo tăng nhưng thời gian xử lý của Backtracking và A* xê lệch tương đối ít so với test 3×3 do các đảo có phân phối đều hơn và ít đảo nằm ở vị trí góc hơn.
- Brute-force có thời gian xử lý tăng do số lượng đảo tăng dẫn đến số biến mệnh đề tăng.

- **Test case 3:**

```
2, 0, 5, 0, 4
0, 0, 0, 0, 0
0, 0, 4, 0, 3
1, 0, 0, 0, 0
0, 0, 4, 0, 3
```

Kích cỡ: 5×5.
Số lượng đảo: 8.



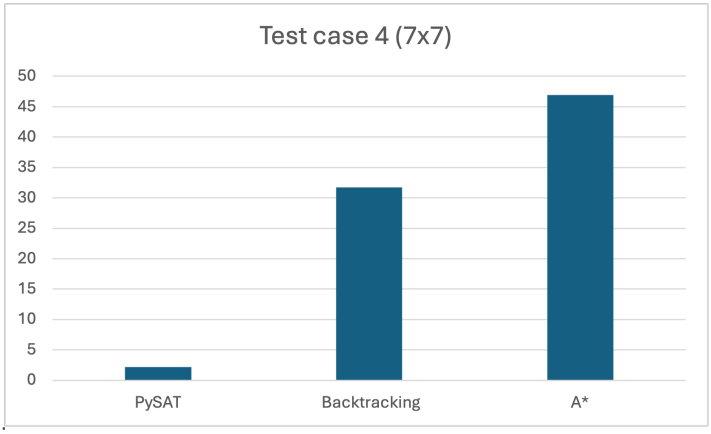
Nhận xét:

- Khi số đảo càng tăng dẫn đến số biến mệnh đề càng tăng, thuật toán Brute-force cho ra hiệu năng có sự cách biệt rất lớn với các thuật toán còn lại (424.5ms), cho thấy sự kém hiệu quả của cách tiếp cận sử dụng tổ hợp các mệnh đề như Brute-force $\rightarrow$ dễ dẫn đến bùng nổ tổ hợp.
- Backtracking và A* có hiệu năng xê lệch không đáng kể.

- **Test case 4:**

```
2, 0, 2, 0, 2, 0, 2
0, 4, 0, 6, 0, 2, 0
2, 0, 0, 0, 0, 0, 0
0, 4, 0, 4, 0, 0, 2
0, 0, 1, 0, 2, 0, 0
1, 0, 0, 0, 0, 0, 0
0, 2, 0, 0, 4, 0, 2
```

Kích cỡ: 7×7.
Số lượng đảo: 17.



Nhận xét:

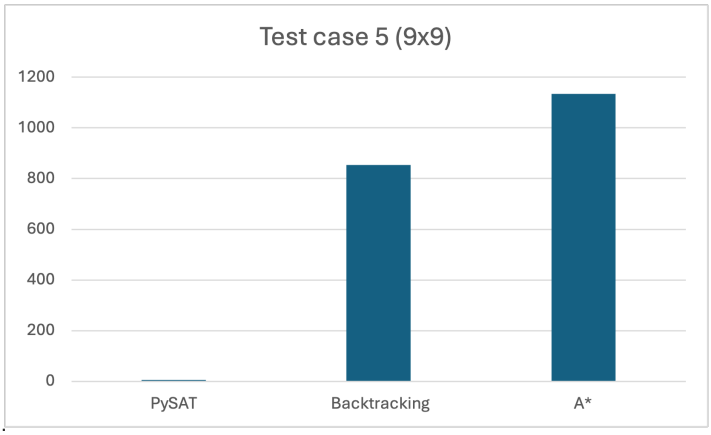
– Brute-force không thể đo được do số lượng tổ hợp đã vượt ngưỡng giới hạn cho việc xử lý tính toán.

– Ở không gian trạng thái rộng hơn, Backtracking dần cho thấy hiệu quả tốt hơn A* nhờ vào việc cắt tỉa đường đi sai sớm trong khi A* phải mở rộng nhiều trạng thái.

- **Test case 5:**

```
3, 0, 3, 0, 4, 0, 3, 0, 3
0, 0, 0, 0, 0, 0, 0, 0, 0
0, 1, 0, 0, 4, 0, 0, 3, 0
3, 0, 3, 0, 0, 2, 0, 0, 0
0, 0, 0, 1, 0, 0, 1, 0, 3
0, 0, 3, 0, 0, 2, 0, 2, 0
2, 0, 0, 0, 0, 0, 2, 0, 2
0, 0, 0, 3, 0, 3, 0, 1, 0
1, 0, 3, 0, 2, 0, 2, 0, 1
```

Kích cỡ: 9×9.
Số lượng đảo: 38.



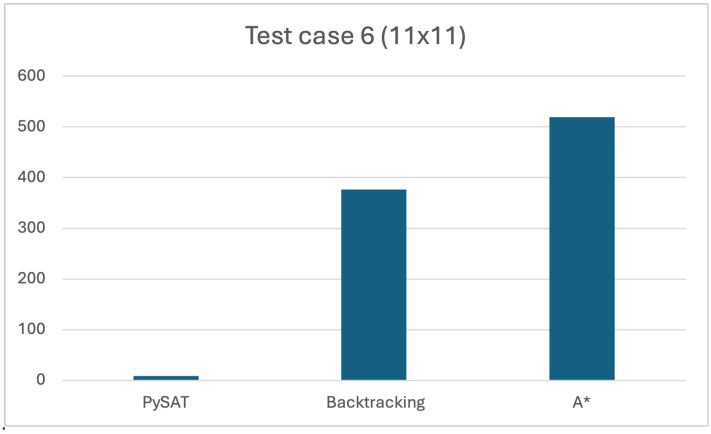
Nhận xét:

- Ở không gian trạng thái rộng hơn, Backtracking dần cho thấy hiệu quả tốt hơn A* nhờ vào việc cắt tỉa đường đi sai sớm trong khi A* phải mở rộng nhiều trạng thái.
- Phân bố đảo đặc thù (có các đảo nằm trong góc, nhiều khoảng trống giữa các đảo) khiến cho thuật toán Backtracking và A* có chi phí xử lý cao hơn tương đối nhiều so với test case trước đó (7×7) và thậm chí gấp đôi thời gian của test (11×11).

- **Test case 6:**

```
2, 0, 0, 2, 0, 0, 3, 0, 0, 3, 0
0, 2, 0, 0, 3, 0, 0, 0, 3, 0, 1
0, 0, 2, 0, 0, 4, 0, 4, 0, 0, 0
2, 0, 0, 0, 0, 0, 1, 0, 0, 2, 0
0, 0, 4, 0, 2, 0, 0, 0, 3, 0, 0
0, 1, 0, 0, 0, 2, 0, 3, 0, 0, 2
3, 0, 6, 0, 4, 0, 2, 0, 0, 2, 0
0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 2
2, 0, 0, 0, 2, 0, 2, 0, 0, 1, 0
0, 0, 3, 0, 0, 3, 0, 0, 5, 0, 3
2, 0, 0, 2, 0, 0, 0, 2, 0, 1, 0
```

Kích cỡ: 11×11.
Số lượng đảo: 61.



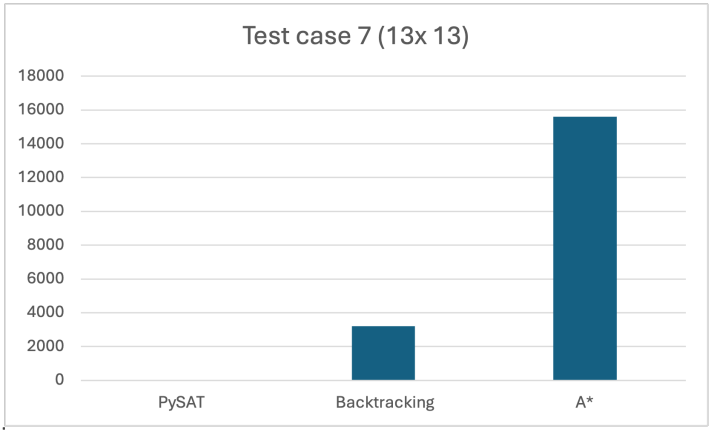
Nhận xét:

- Ở không gian trạng thái rộng hơn, Backtracking dần cho thấy hiệu quả tốt hơn A* nhờ vào việc cắt tỉa đường đi sai sớm trong khi A* phải mở rộng nhiều trạng thái.
- Phân bố đảo có nhiều giá trị thấp 1, 2 (vì nhóm đã thực hiện thêm tối ưu trong việc tạo các mệnh đề CNF ràng buộc cho tình huống này) nên cho dù có số lượng đảo nhiều hơn nhưng thời gian xử lý của A* và Backtracking vẫn nhanh hơn so với test 9×9.

- **Test case 7:**

```
3, 0, 3, 0, 3, 0, 0, 3, 0, 0, 2, 0, 2
0, 2, 0, 3, 0, 0, 2, 0, 0, 4, 0, 4, 0
3, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 2
0, 0, 1, 0, 0, 0, 0, 0, 0, 3, 0, 5, 0
3, 0, 0, 4, 0, 4, 0, 1, 0, 0, 0, 0, 2
0, 0, 2, 0, 0, 0, 1, 0, 3, 0, 0, 3, 0
3, 0, 0, 6, 0, 5, 0, 1, 0, 0, 0, 0, 2
0, 0, 2, 0, 0, 0, 0, 0, 0, 3, 0, 3, 0
0, 2, 0, 3, 0, 3, 0, 0, 2, 0, 0, 0, 2
3, 0, 4, 0, 2, 0, 4, 0, 0, 2, 0, 0, 0
0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0
3, 0, 4, 0, 2, 0, 5, 0, 0, 1, 0, 0, 1
0, 1, 0, 2, 0, 2, 0, 0, 2, 0, 0, 2, 0
```

Kích cỡ: 13×13.
Số lượng đảo: 86.



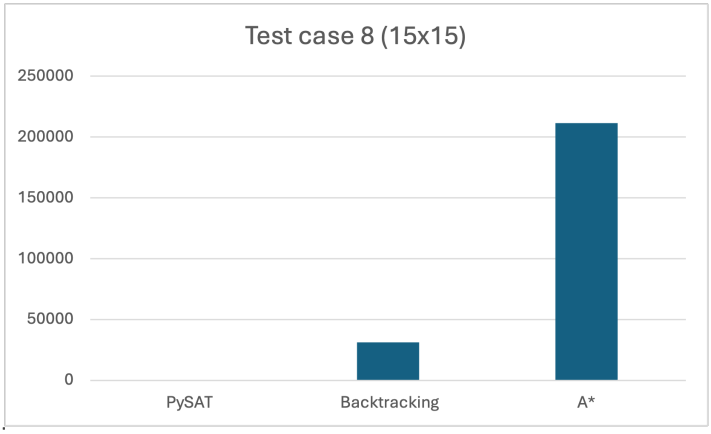
Nhận xét:

– Ở không gian trạng thái rộng lớn như 13 × 13, ta có thể thấy sự khác biệt rõ rệt về thời gian xử lý giữa Backtracking và A*, A* đã phải tốn chi phí lớn cho việc tìm kiếm và duy trì độ ưu tiên và lưu trữ trạng thái cũ → có thể khẳng định rằng A* không phải là giải pháp tối ưu để giải quyết các bài toán có ràng buộc toàn cục.

- **Test case 8:**

```
2, 0, 0, 2, 0, 0, 3, 0, 0, 3, 0
0, 2, 0, 0, 3, 0, 0, 0, 3, 0, 1
0, 0, 2, 0, 0, 4, 0, 4, 0, 0, 0
2, 0, 0, 0, 0, 0, 1, 0, 0, 2, 0
0, 0, 4, 0, 2, 0, 0, 0, 3, 0, 0
0, 1, 0, 0, 0, 2, 0, 3, 0, 0, 2
3, 0, 6, 0, 4, 0, 2, 0, 0, 2, 0
0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 2
2, 0, 0, 0, 2, 0, 2, 0, 0, 1, 0
0, 0, 3, 0, 0, 3, 0, 0, 5, 0, 3
2, 0, 0, 2, 0, 0, 0, 2, 0, 1, 0
```

Kích cỡ: 15 × 15.
Số lượng đảo: 112.



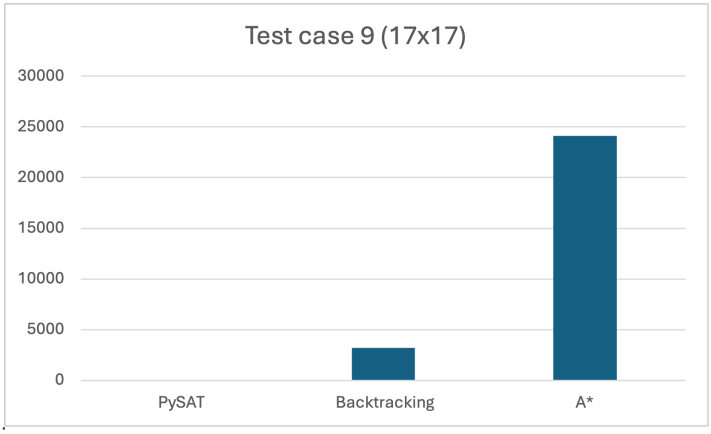
Nhận xét:

– Ở không gian trạng thái rộng lớn như 15 × 15, ta có thể thấy sự khác biệt rõ rệt về thời gian xử lý giữa Backtracking và A*, A* đã phải tốn chi phí lớn cho việc tìm kiếm và duy trì độ ưu tiên và lưu trữ trạng thái cũ → có thể khẳng định rằng A* không phải là giải pháp tối ưu để giải quyết các bài toán có ràng buộc toàn cục.

- **Test case 9:**

```
0, 1, 0, 0, 5, 0, 0, 5, 0, 2, 0, 2, 0, 0, 0, 2, 0
2, 0, 0, 1, 0, 1, 0, 0, 1, 0, 2, 0, 0, 2, 0, 0, 0
0, 0, 1, 0, 3, 0, 0, 3, 0, 2, 0, 1, 0, 0, 0, 0, 1
3, 0, 0, 2, 0, 5, 0, 0, 0, 0, 0, 0, 0, 5, 0, 3, 0
0, 2, 0, 0, 1, 0, 2, 0, 4, 0, 3, 0, 3, 0, 0, 0, 0
3, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 1, 0, 0, 2, 0, 2
0, 2, 0, 0, 0, 4, 0, 0, 2, 0, 1, 0, 3, 0, 0, 2, 0
0, 0, 0, 1, 0, 0, 2, 0, 0, 3, 0, 2, 0, 3, 0, 0, 0
6, 0, 2, 0, 0, 1, 0, 0, 3, 0, 0, 0, 0, 0, 4, 0, 2
0, 1, 0, 4, 0, 0, 4, 0, 0, 0, 3, 0, 2, 0, 0, 2, 0
5, 0, 2, 0, 0, 0, 0, 1, 0, 2, 0, 1, 0, 0, 0, 0, 3
0, 0, 0, 0, 1, 0, 4, 0, 1, 0, 0, 0, 3, 0, 3, 0, 0
0, 0, 0, 3, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 3, 0
5, 0, 4, 0, 1, 0, 3, 0, 0, 2, 0, 0, 2, 0, 2, 0, 3
0, 1, 0, 3, 0, 3, 0, 0, 3, 0, 0, 2, 0, 3, 0, 3, 0
1, 0, 1, 0, 1, 0, 0, 2, 0, 0, 2, 0, 2, 0, 0, 0, 0
0, 2, 0, 3, 0, 2, 0, 0, 2, 0, 0, 2, 0, 3, 0, 0, 2
```

Kích cỡ: 17 × 17.
Số lượng đảo: 117.



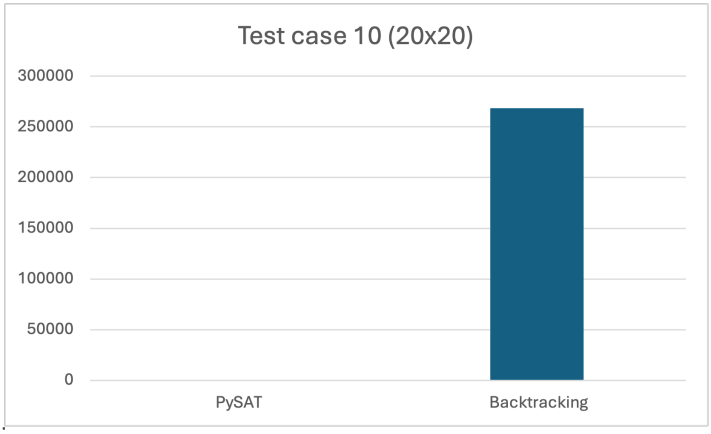
Nhận xét:

– Ở không gian trạng thái rộng lớn như 17 × 17, ta có thể thấy sự khác biệt rõ rệt về thời gian xử lý giữa Backtracking và A*, A* đã phải tốn chi phí lớn cho việc tìm kiếm và duy trì độ ưu tiên và lưu trữ trạng thái cũ → có thể khẳng định rằng A* không phải là giải pháp tối ưu để giải quyết các bài toán có ràng buộc toàn cục.

- **Test case 10:**

```
0, 2, 0, 2, 0, 0, 4, 0, 0, 3, 0, 0, 2, 0, 0, 2, 0, 0, 3, 0
1, 0, 3, 0, 0, 4, 0, 4, 0, 0, 0, 3, 0, 2, 0, 0, 1, 0, 0, 0
0, 0, 0, 0, 0, 0, 2, 0, 3, 0, 3, 0, 4, 0, 3, 0, 0, 2, 0, 1
0, 3, 0, 2, 0, 6, 0, 3, 0, 3, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0
3, 0, 4, 0, 2, 0, 2, 0, 3, 0, 0, 0, 0, 2, 0, 2, 0, 0, 4, 0
0, 4, 0, 0, 0, 0, 0, 2, 0, 4, 0, 0, 3, 0, 3, 0, 0, 4, 0, 2
0, 0, 0, 1, 0, 3, 0, 0, 3, 0, 0, 4, 0, 3, 0, 0, 2, 0, 0, 0
3, 0, 4, 0, 0, 0, 4, 0, 0, 2, 0, 0, 1, 0, 3, 0, 0, 6, 0, 4
0, 3, 0, 0, 2, 0, 0, 0, 3, 0, 0, 6, 0, 3, 0, 0, 4, 0, 0, 0
3, 0, 2, 0, 0, 0, 0, 2, 0, 0, 1, 0, 1, 0, 3, 0, 0, 3, 0, 3
0, 3, 0, 0, 6, 0, 4, 0, 0, 0, 0, 3, 0, 2, 0, 0, 3, 0, 0, 0
4, 0, 0, 0, 0, 0, 0, 0, 5, 0, 5, 0, 2, 0, 3, 0, 0, 3, 0, 0
0, 0, 1, 0, 3, 0, 0, 3, 0, 0, 0, 3, 0, 3, 0, 0, 0, 0, 0, 1
0, 3, 0, 5, 0, 2, 0, 0, 6, 0, 5, 0, 3, 0, 4, 0, 3, 0, 0, 0
0, 0, 0, 0, 3, 0, 0, 3, 0, 0, 0, 2, 0, 2, 0, 2, 0, 6, 0, 4
3, 0, 0, 4, 0, 1, 0, 0, 0, 0, 2, 0, 4, 0, 5, 0, 6, 0, 4, 0
0, 1, 0, 0, 2, 0, 0, 0, 4, 0, 0, 0, 0, 2, 0, 0, 0, 2, 0, 3
4, 0, 0, 3, 0, 2, 0, 4, 0, 4, 0, 3, 0, 0, 2, 0, 0, 0, 0, 0
0, 1, 0, 0, 2, 0, 0, 0, 3, 0, 4, 0, 4, 0, 0, 0, 4, 0, 2, 0
4, 0, 3, 0, 0, 0, 2, 0, 0, 2, 0, 3, 0, 3, 0, 3, 0, 4, 0, 1
```

Kích cỡ: 20 × 20.
Số lượng đảo: 156.



Nhận xét:

– Ở không gian trạng thái rộng lớn và phức tạp như 20 × 20, A* đã không thể cho ra kết quả ở thời gian cho phép đo được, điều này đã khẳng định rằng A* rơi vào bế tắc và tiêu tốn nhiều tài nguyên ở nhưng test rộng lớn và có phân phối đảo phức tạp.

Test Case	**PySAT**	**Brute-force**	**Backtracking**	**A***
Test case 1 (3 × 3)	0.3	0.6	2.3	1.6
Test case 2 (4 × 4)	0.4	1.1	1.7	1.1
Test case 3 (5 × 5)	1.4	424.5	24.7	17.2
Test case 4 (7 × 7)	2.2	–	31.7	46.9
Test case 5 (9 × 9)	4.8	–	853	1134.4
Test case 6 (11 × 11)	8.7	–	376.1	519.2
Test case 7 (13 × 13)	12.9	–	3195.574	15605.44
Test case 8 (15 × 15)	24.4	–	31256.7	211422.6
Test case 9 (17 × 17)	34.6	–	3198.2	24123.84
Test case 10 (20 × 20)	79.2	–	268541	–

Bảng 4: So sánh thời gian thực thi của các thuật toán (Đơn vị: ms)

6.2 Tổng kết

- PySAT vẫn là giải pháp hiệu quả cho bài toán, có thời gian xử lý tăng tuyến tính nhẹ so với số lượng biến mệnh đề, luôn cho kết quả dưới 1s.

- Brute-force cho kết quả tương đối tốt với những test nhỏ (3×3, 4×4) nhưng dần kém hiệu quả khi các test có số lượng biến mệnh đề lớn hơn và dẫn đến bùng nổ tổ hợp ở các test lớn.
- Backtracking nhờ cắt tỉa đường đi sớm nên có phần hiệu quả hơn so với A* nhưng ở các test rộng và phức tạp hơn thì số lần gọi đệ quy tăng lên đáng kể, dẫn đến thời gian xử lý cũng kéo dài (xấp xỉ 4 phút rưỡi cho test 10: 20×20).
- A* cho kết quả gần tương tự với Backtracking ở các test case đầu nhưng khi test case càng rộng và phức tạp thì A* có sự cách biệt rõ ràng so với Backtracking, thậm chí ở test 10: 20×20 thì A* đã không thể cho ra kết quả trong thời gian có thể đo được.

Kết luận:

- Đối với bài toán CNF, các thuật toán như Backtracking và A* có thể được áp dụng tương đối ổn định trong các test case nhỏ và đơn giản. Brute-force, mặc dù vẫn khả thi về mặt lý thuyết, nhưng lại yêu cầu thời gian xử lý quá lâu ở các test có số lượng biến mệnh đề lớn nên không phù hợp để áp dụng trong thực tế.
- Trong khi đó, với các bài toán có không gian trạng thái lớn và phức tạp hơn, PySAT là lựa chọn tối ưu và đáng tin cậy nhất. Nhờ khả năng giải quyết nhanh và hiệu quả, PySAT cho thấy ưu thế rõ rệt về cả tốc độ lẫn tính thực tiễn khi giải các bài toán CNF.

6.3 Mở rộng

6.3.1 Hạn chế

- Heuristic của A* còn đơn giản, có thể không phù hợp với các test lớn và phức tạp.
- Các test case đa số ở độ khó trung bình, có thể chưa phản ánh toàn diện về sự khác biệt của các thuật toán ở không gian trạng thái có độ phức tạp cao.

6.3.2 Cải tiến

- Thêm heuristic (MRV) cho Backtracking để tối ưu thời gian xử lý.
- Thêm các ràng buộc ban đầu khác để tối ưu không gian tìm kiếm.
- Đo thêm các chỉ số về memory usage của mỗi thuật toán, số lượng biến mệnh đề CNF được tạo ra trong mỗi test để cho thấy sự phức tạp của test case và sự tiêu tốn tài nguyên của các thuật toán.

Tài liệu tham khảo

1. Artificial Intelligence: A Modern Approach, 4th Edition
2. AIMA: https://github.com/aimacode/aima-python/blob/master/logic.py
3. Tool tạo test case: https://www.kakuro-online.com/hashi/