Trường Đại học Khoa học Tự nhiên, ĐHQG-HCM

Khoa Công nghệ thông tin

Môn học: Cơ sở trí tuệ nhân tạo

Lớp: 23TN

Đồ án 2: Hashiwokakero

Sinh viên:
23120039 - Ung Dung Thanh Hạ
23120108 - Vòng Hải Yến
23120145 - Nguyễn Ngọc Duy Mỹ

Giáo viên hướng dẫn:
GS. TS. Lê Hoài Bắc
Thầy Nguyễn Thanh Tình

Tháng 12 năm 2025

Lời cảm ơn

Chúng em xin trân trọng gửi lời cảm ơn đến GS.TS. Lê Hoài Bắc và thầy Nguyễn Thanh Tình vì đã tận tình hướng dẫn, định hướng và hỗ trợ chúng em trong suốt quá trình học tập và thực hiện đồ án môn Cơ sở Trí tuệ Nhân tạo. Những nhận xét chuyên môn và sự tận tâm của thầy đã giúp chúng em hiểu rõ hơn về bản chất của các thuật toán, cũng như cách tư duy và tiếp cận một vấn đề dưới góc nhìn khoa học.

Chúng em cũng xin cảm ơn Khoa Công nghệ Thông tin - Trường Đại học Khoa học Tự nhiên, ĐHQG-HCM đã tạo ra môi trường học thuật nghiêm túc, cơ sở vật chất và nền tảng kiến thức để chúng em có thể nghiên cứu và hoàn thiện đồ án một cách thuận lợi.

Bên cạnh sự hướng dẫn của quý thầy, chúng em ghi nhận tinh thần làm việc trách nhiệm và hỗ trợ lẫn nhau giữa các thành viên trong nhóm. Những buổi trao đổi, phân tích và chỉnh sửa liên tục đã giúp nhóm hoàn thiện đồ án, đồng thời tích lũy thêm nhiều kinh nghiệm quý báu cho các học phần chuyên ngành sau này.

Mặc dù đã nỗ lực hết sức, bài làm khó tránh khỏi những thiếu sót. Chúng em kính mong nhận được sự góp ý của quý thầy để có thể hoàn thiện hơn trong các nghiên cứu và dự án tương lai.

Chúng em xin chân thành cảm ơn.

Mục lục

Danh sách hình vẽ

Danh sách bảng

1 Thông tin chung

1.1 Phân công công việc

Sinh viên	Nhiệm vụ	Mô tả	Hoàn thành
Ung Dung Thanh Hạ	Thuật toán A*	Cài đặt thuật toán	100%
	Brute Force và Backtracking	Phân tích ý tưởng, kiểm tra mã nguồn và xây dựng mã giả	
	Thử nghiệm	Cài đặt thử nghiệm, thu thập kết quả	
	Hoàn thiện báo cáo	Tổng hợp nội dung, rà soát báo cáo, hoàn thiện giới thiệu, kết luận và thông tin khác	
Vòng Hải Yến	Backtracking	Cài đặt thuật toán	100%
	Định nghĩa biến logic	Xây dựng hệ biến logic và ràng buộc CNF	
	Sinh CNF và PySAT	Cài đặt thuật toán và báo cáo về tự động hóa quá trình sinh CNF và giải bằng thư viện PySAT	
	Hoàn thiện mã nguồn	Refactor, tổ chức lại mã và viết README	
Nguyễn Ngọc Duy Mỹ	Test case	Sinh các test input	100%
	Brute Force	Cài đặt thuật toán	
	PySAT và A*	Phân tích ý tưởng, kiểm tra mã nguồn và xây dựng mã giả	
	Trực quan hóa	Lập bảng so sánh; code và vẽ các biểu đồ trực quan hóa; trình bày kết quả và phân tích	
	Video demo	Quay và chỉnh sửa video demo	

Bảng 1: Bảng phân công công việc

1.2 Tự đánh giá

Về phần tự đánh giá, chúng em tự đánh giá rằng nhóm đã hoàn thành đầy đủ các yêu cầu cơ bản của đồ án, bao gồm việc cài đặt mã nguồn và giải thích chi tiết về cách thức hoạt động và ý tưởng đằng sau mỗi thuật toán. Ngoài ra, nhóm cũng đã thực hiện thử nghiệm và chạy thử các thuật toán, kết quả và đánh giá nghiệm thu trong quá trình thực hiện đã được trực quan hoá thông qua các biểu đồ và đồ thị trong báo cáo, kèm theo đó là các nhận xét đã được rút ra về mức độ tối ưu và thời gian chạy trung bình của từng phương pháp.

Mặc dù đã cố gắng hoàn thiện bài làm hết sức có thể, chúng em đôi khi vẫn không tránh khỏi những thiếu sót trong quá trình thực hiện đồ án. Vì vậy, hi vọng có thể nhận được sự đánh giá khách quan và góp ý của thầy để chúng em có thêm kinh nghiệm cho các đồ án trong tương lai.

1.3 Liên kết

Github: `https://github.com/VongHaiYen11/CSAI_Project_2.git`

Video Demo: `https://tinyurl.com/mwcsz3zv`

2 Tổng quan

2.1 Giới thiệu

Hashiwokakero (Tạm dịch: Hãy bắc cầu) là một trò chơi được công bố lần đầu bởi Nikoli - một công ty Nhật Bản chuyên về các dạng trò chơi câu đố - vào năm 1990 [4]. Trò chơi nhanh chóng trở nên nổi tiếng và được phổ biến tới nhiều nước trên thế giới với nhiều tên gọi khác nhau.

Trò chơi bao gồm một số đảo nhất định, mỗi đảo được đánh dấu bởi một số từ một đến tám và nhiệm vụ của người chơi là tìm cách nối các đảo lại với nhau sao cho thỏa mãn một số yêu cầu nhất định như:

- Không thể nối một đảo với chính nó.
- Giữa hai đảo chỉ được nối bằng tối đa hai cây cầu.
- Mỗi đảo chỉ được nối bằng số cầu đúng với số của đảo.
- Các đảo chỉ được nối đến đảo nằm trên cùng một đường thẳng (hàng dọc hoặc hàng ngang).
- Các cầu không được cắt nhau.
- Tất cả các đảo phải được nối thành một nhóm liên thông duy nhất.

Trò chơi không chỉ cuốn hút đối với những người yêu thích giải đố, nó còn trở thành một bài toán trong lĩnh vực máy tính, được mọi người dành thời gian tìm hiểu về cách giải hoặc thuật toán tối ưu để tìm được lời giải cho trò chơi này. Một trong số những cách giải nổi tiếng có thể kể đến việc chuyển bài toán về dạng CNF. Đối với các ngôn ngữ lập trình bậc cao hiện nay, việc giải CNF không còn là một thách thức lớn khi có sự hỗ trợ của các thư viện. Tuy nhiên, tìm hiểu và so sánh với các cách giải CNF khác hoặc thậm chí là các cách giải ngây thơ cũng sẽ giúp chúng ta có một góc nhìn tổng quan hơn về bài toán cũng như cách mà CNF giúp chúng ta giải các bài toán có thể tưởng chừng như không thể.

Dựa trên mục đích đó, đồ án này của chúng em tập trung vào việc phân tích và cài đặt bài toán dưới dạng CNF. Sau đó, bài toán sẽ được giải bằng bốn phương

pháp chính: thư viện `PySAT` của ngôn ngữ lập trình `Python`, thuật toán tìm kiếm A*, thuật toán quay lui (Backtracking) và thuật toán vét cạn (Brute-force). Cụ thể về các thuật toán cũng như cách cài đặt và kết quả thử nghiệm sẽ được lần lượt trình bày trong các phần tiếp theo.

2.2 Biểu diễn bài toán

Để có thể giải bài toán, ta cần phải mô hình hóa nó dưới các dạng biểu diễn trên máy tính để có thể bắt đầu tính toán. Ma trận là dạng biểu diễn được lựa chọn cho bài toán Hashiwokakero. Cụ thể, bài toán được biểu diễn dưới dạng như sau:

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

với các vị trí có số tượng trưng cho các đảo và số của đảo đó, những vị trí có giá trị 0 là vị trí trống.

Cách biểu diễn kết quả cũng có dạng ma trận tương tự:

```
[ "0", "2", "=", "5", "_", "_", "2" ]
[ "0", "0", "0", "$", "0", "0", "|" ]
[ "4", "=", "2", "$", "2", "=", "4" ]
[ "$", "0", "0", "$", "0", "0", "|" ]
[ "$", "1", "_", "5", "=", "2", "|" ]
[ "$", "0", "0", "0", "0", "0", "|" ]
[ "4", "=", "=", "=", "=", "=", "3" ]
```

với `|` và `$` trượng trưng cho một cầu và hai cầu nối theo hàng dọc, `-` và `=` là một cầu và hai cầu nối theo chiều ngang.

3 Xây dựng ràng buộc CNF

3.1 Định nghĩa biến logic

Bài toán Hashiwokakero yêu cầu xác định vị trí và số lượng các cầu nối giữa các đảo sao cho thoả mãn đầy đủ các ràng buộc của trò chơi. Do đó, đối tượng tác động đến lời giải của bài toán và bị ảnh hưởng bởi các ràng buộc cụ thể không phải là các đảo, mà là các cầu mà ta cần đặt để nối các đảo. Vì vậy, việc lựa chọn biến logic biểu diễn cho các cầu có thể được xem là một lựa chọn phù hợp với bản chất của bài toán.

Theo luật chơi Hashiwokakero (đã trình bày ở Mục 1.1), việc đặt cầu phải tuân theo nhiều qui tắc khác nhau, trong đó, có hai ràng buộc ảnh hưởng trực tiếp đến việc lựa chọn cách biểu diễn biến logic, bao gồm:

- Giữa hai đảo chỉ được nối tối đa hai cầu.
- Tổng số cầu nối vào mỗi đảo phải đúng bằng giá trị được ghi trên đảo đó.

Dựa trên các đặc điểm trên, mỗi biến logic được định nghĩa để biểu diễn sự tồn tại của một cầu giữa hai đảo kề nhau theo hàng hoặc cột. Cụ thể, một biến cầu được biểu diễn bằng một bộ ba (u, v, i), trong đó:

- u: đảo xuất phát.
- v: đảo đích có thể nối với u.
- i: chỉ số của cầu, có thể mang giá trị 1 hoặc 2, tương ứng với cầu thứ nhất hoặc thứ hai giữa hai đảo.

Mỗi bộ ba như vậy tương ứng với một biến logic Boolean, nếu biến này là `True`, có nghĩa là có cầu thứ i nối từ đảo u đến đảo v trên ma trận kết quả cuối cùng.

Ta có thể kí hiệu một biến đại diện cho cầu như sau:

$$br\_(r_u, c_u)\_(r_v, c_v)\_i$$

với r là số dòng và c là số cột.

Ngoài ra, cần lưu ý rằng, do các cầu trong bài toán Hashiwokakero tạo thành một đồ thị vô hướng, việc phân biệt thứ tự giữa hai đảo là không cần thiết về mặt ngữ nghĩa. Tuy nhiên, nếu không có quy ước thống nhất, có thể phát sinh các biến logic đối xứng, chẳng hạn như $br\_(r_u, c_u)\_(r_v, c_v)\_i$ và $br\_(r_v, c_v)\_(r_u, c_u)\_i$, dù cả hai đều biểu diễn cùng một cầu giữa hai đảo u và v.

Để tránh hiện tượng này, ta sẽ quy định đảo u luôn được chọn là đảo bắt đầu khi tạo cầu nối với đảo v nếu nó xuất hiện trước v theo thứ tự duyệt ma trận từ trái sang phải và từ trên xuống dưới. Nhờ đó, mỗi cầu khả thi trong bài toán chỉ tương ứng với đúng một biến logic duy nhất, giúp giảm số lượng biến và tránh việc xuất hiện các ràng buộc dư thừa.

Ngoài ra, để thuận tiện cho việc xây dựng công thức CNF và tương thích với SAT solver, các biến logic không được lưu trữ trực tiếp dưới dạng các cấu trúc dữ liệu phức tạp khi lập trình. Thay vào đó, mỗi bộ ba (u, v, i) được ánh xạ sang một số nguyên dương duy nhất (bằng cách sử dụng biến `counter` trong mã nguồn) để phù hợp theo định dạng đầu vào mà SAT solver yêu cầu.

3.2 Ý tưởng xây dựng ràng buộc CNF

3.2.1 Ràng buộc trong quá trình sinh biến

Các ràng buộc đơn giản thuộc nhóm này bao gồm:

1. Không thể nối một đảo với chính nó.
2. Giữa hai đảo chỉ được nối bằng tối đa hai cây cầu.
3. Các đảo chỉ được nối đến đảo nằm trên cùng một đường thẳng.

Bởi vì ta chọn cách biểu diễn biến logic dưới dạng cầu, vì vậy, đối với các ràng buộc trên, ta có thể trực tiếp can thiệp vào việc xây dựng cầu sao cho thỏa mãn các điều kiện trên mà không cần tạo ra các mệnh đề CNF riêng biệt để giới hạn.

3.2.2 Mỗi đảo phải được nối với đúng số lượng cầu bằng với giá trị ghi trên đảo

Đây là một trong những ràng buộc cốt lõi nhất của bài toán Hashiwokakero, ta có thể mô tả ý tưởng sơ lược như sau:

Gọi I là tập các đảo trong bài toán. Với mỗi đảo $u \in I$, ta tạm ký hiệu $B(u)$ là tập các biến logic đại diện cho các cầu kề với đảo u thay vì dùng cách kí hiệu đầy đủ như ở mục 3.1 cho từng cây cầu. Ràng buộc này có thể được phát biểu ngắn gọn như sau:

$$\forall u \in I, \quad \sum_{b \in B(u)} b = \text{matrix}[r_u][c_u]$$

trong đó $\text{matrix}[r_u][c_u]$ là số ghi trên đảo u, và mỗi biến $b \in B(u)$ nhận giá trị 1 nếu cầu tương ứng tồn tại, và 0 trong trường hợp ngược lại.

Giả sử một đảo u có n cầu khả thi kề với nó, tương ứng với tập các biến logic

$$B(u) = \{b_1, b_2, \ldots, b_n\}$$

trong đó mỗi biến b_i nhận giá trị `True` nếu cầu tương ứng được chọn. Giá trị ghi trên đảo là k. Khi đó, yêu cầu đặt ra là phải có đúng k biến trong tập $B(u)$ nhận giá trị `True`.

Một cách tiếp cận trực tiếp là xem mỗi tổ hợp lựa chọn k biến là một cấu hình hợp lệ, và nghiệm phải thuộc ít nhất một trong các cấu hình này. Ta xem xét ví dụ sau:

Giả sử một đảo có tập các biến logic kề với nó là $B = \{b_1, b_2\}$, và giá trị ghi trên đảo là $k = 1$. Các lời giải hợp lệ tương ứng với điều kiện này có thể dễ dàng được rút ra như sau: $(b_1), \quad (b_2)$.

Mỗi lời giải trên có thể được biểu diễn bằng một mệnh đề liên hợp, trong đó biến được chọn mang giá trị `True` và biến còn lại mang giá trị `False`:

$$(b_1 \wedge \neg b_2), \quad (\neg b_1 \wedge b_2)$$

Để đảm bảo lời giải sinh ra thoả mãn đúng một trong hai tổ hợp trên, ta lấy phép

`OR` của các mệnh đề tương ứng và thu được ràng buộc:

$$C_{\text{island}} = (b_1 \wedge \neg b_2) \ \vee \ (\neg b_1 \wedge b_2)$$

Ngoài ra, ta cũng có thể xây dựng ràng buộc thông qua việc kết hợp hai điều kiện cần và đủ, đó là xây dựng cận dưới và cận trên đối với số biến k nhận giá trị `True` như sau:

1. Ràng buộc cận dưới (ít nhất k biến `True`). Xét mọi tập con gồm $(n-k+1)$ biến bất kỳ trong $B(u)$. Với mỗi tập con như vậy, ta tạo một mệnh đề dạng:

   $$b_{i_1} \vee b_{i_2} \vee \cdots \vee b_{i_{n-k+1}}$$

   Ràng buộc này đảm bảo rằng không thể tồn tại đồng thời $(n-k+1)$ biến nhận giá trị `False`. Do đó, số biến `True` trong toàn bộ tập $B(u)$ không thể nhỏ hơn k.

2. Ràng buộc cận trên (không quá k biến `True`). Xét mọi tập con gồm $(k+1)$ biến bất kỳ trong $B(u)$. Với mỗi tập con, ta tạo một mệnh đề dạng:

   $$\neg b_{j_1} \vee \neg b_{j_2} \vee \cdots \vee \neg b_{j_{k+1}}$$

   Ràng buộc này ngăn không cho $(k+1)$ biến cùng nhận giá trị `True` đồng thời, từ đó đảm bảo số biến `True` không vượt quá k.

Kết hợp hai nhóm ràng buộc trên, ta thu được điều kiện rằng số cầu được chọn kề với đảo u phải bằng đúng k.

3.2.3 Các cầu không được cắt nhau

Ở đây, có thể đưa ra một nhận xét rằng, mỗi cầu chỉ có thể được vẽ giữa các đảo nằm trên một đường thẳng, hiện tượng cắt nhau chỉ có thể xảy ra giữa một cầu ngang và một cầu dọc. Vì vậy, để giảm bớt số cặp cầu cần kiểm tra khi chạy thuật toán (do hai cầu cùng 1 nhóm chắc chắn không giao nhau), ta sẽ trực tiếp chia các biến logic cầu thành 2 nhóm (Ngang và Dọc) để thuật toán hiệu quả hơn.

Trong thực tế, ta sẽ thực hiện xem xét tất cả các cặp cầu khả thi (1 ngang, 1 dọc) trong miền giá trị và thực hiện tạo ràng buộc. Ở đây, ta sẽ xem xét một trường hợp tổng quát như sau:

- Cầu dọc b_{ver} nối hai đảo $u_{ver}(r_{v1}, c_v)$ và $v_{ver}(r_{v2}, c_v)$, với $r_{v1} < r_{v2}$,
- Cầu ngang b_{hor} nối hai đảo $u_{hor}(r_h, c_{h1})$ và $v_{hor}(r_h, c_{h2})$, với $c_{h1} < c_{h2}$.

Hai cầu b_v và b_h sẽ cắt nhau nếu và chỉ nếu thỏa mãn đồng thời hai điều kiện:

$$\begin{cases} c_{h1} < c_v < c_{h2} \\ r_{v1} < r_h < r_{v2} \end{cases}$$

Nói cách khác, với mỗi cặp cầu ngang và cầu dọc có khả năng cắt nhau, ta thêm một mệnh đề đảm bảo rằng hai cầu không cùng nhau xuất hiện trong lời giải cuối cùng, mệnh đề được viết như sau:

$$\neg b_{\text{ver}} \vee \neg b_{\text{hor}}$$

3.2.4 Tất cả các đảo phải được nối thành một nhóm liên thông duy nhất

Thông thường, đối với các yêu cầu ràng buộc, ta cần phải tạo ra các ràng buộc bằng các literal. Tuy nhiên, việc xây dựng ràng buộc để kiểm tra liên thông trong trường hợp có nhiều đảo là rất phức tạp và tốn nhiều chi phí thời gian. Do đó, chúng em chọn cách kiểm tra tính liên thông trực tiếp trên từng lời giải sinh ra. Cụ thể, sau khi sinh ra một lời giải bất kỳ, cần kiểm tra xem các cầu đã vẽ có tạo thành một đồ thị liên thông hay không. Nói cách khác, từ một đảo bất kỳ, có thể đi đến tất cả các đảo còn lại thông qua các cầu đã nối. Nếu điều kiện này thỏa mãn, lời giải được coi là hợp lệ theo yêu cầu của bài toán, ngược lại, ta sẽ thêm các clause để loại bỏ các lời giải vi phạm.

3.2.5 Ràng buộc phụ về sự phụ thuộc giữa các cầu

Mỗi cặp đảo u và v bất kì chỉ có thể nối tối đa hai cầu, được đánh số thứ tự $i = 1, 2$ như đã trình bày ở phần 3.1. Để tránh các clause trùng lặp về mặt ngữ nghĩa do

việc chọn cầu thứ 2 trước cầu thứ 1, ta bổ sung một ràng buộc phụ bên cạnh những yêu cầu khác của bài toán, rằng cầu thứ 2 chỉ có thể tồn tại nếu cầu thứ 1 đã được chọn, ràng buộc đó có thể được biểu diễn như sau:

$$b_1 \vee \neg b_2$$

Trong đó: b_1 và b_2 là các cây cầu nối giữa cùng một cặp đảo.

3.2.6 Lưu ý

Trong quá trình xây dựng các ràng buộc CNF, có thể phát sinh nhiều mệnh đề trùng lặp nhau. Do đó, cần thực hiện loại bỏ các ràng buộc trùng nhau nhằm giảm kích thước công thức CNF và cải thiện hiệu quả của quá trình tìm kiếm nghiệm.

3.3 Ví dụ

Để minh họa, ta sẽ thực hiện xây dựng bộ ràng buộc CNF với input sau đây:

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

Cần lưu ý rằng ở đây chỉ đơn thuần là xây dựng ràng buộc để giải được bài toán, chứ không phải nêu ý tưởng tương ứng với việc lập trình ràng buộc. Bên cạnh đó, như đã trình bày, về ràng buộc liên quan đến mục 3.2.4, ta sẽ áp dụng thuật toán để kiểm tra sau khi đã sinh ra một lời giải nên ta không xét đến các CNF liên quan đến điều kiện này.

3.3.1 Sinh biến Logic

Đầu tiên, ta sẽ thực hiện tạo ra tất cả các cầu hợp lệ thỏa các ràng buộc trong quá trình sinh biến (mục 3.2.1) thông qua phương pháp thủ công và thu được các biến logic như sau:

1. $br\_(0,1)\_(0,3)\_1$
2. $br\_(0,1)\_(0,3)\_2$
3. $br\_(0,3)\_(0,6)\_1$
4. $br\_(0,3)\_(0,6)\_2$
5. $br\_(2,0)\_(2,2)\_1$
6. $br\_(2,0)\_(2,2)\_2$
7. $br\_(2,2)\_(2,6)\_1$
8. $br\_(2,2)\_(2,6)\_2$
9. $br\_(6,0)\_(6,6)\_1$
10. $br\_(6,0)\_(6,6)\_2$
11. $br\_(0,1)\_(4,1)\_1$
12. $br\_(0,1)\_(4,1)\_2$
13. $br\_(0,6)\_(2,6)\_1$
14. $br\_(0,6)\_(2,6)\_2$
15. $br\_(2,6)\_(6,6)\_1$
16. $br\_(2,6)\_(6,6)\_2$
17. $br\_(2,0)\_(6,0)\_1$
18. $br\_(2,0)\_(6,0)\_2$

Cần lưu ý, trong phần trình bày này, nhằm giữ cho logic được ngắn gọn và dễ hiểu ở một số phần, ta sẽ biểu diễn các biến trên dưới dạng br_i, với i là số thứ tự của biến logic trong danh sách trên.

3.3.2 Ràng buộc số lượng cầu nối vào đảo

Để tạo các ràng buộc ở mục 3.2.2, ta có thể thực hiện xây dựng các ràng buộc theo một ví dụ cụ thể dưới đây.
Xét đảo (0, 3) trong ma trận trên, có 4 cầu khả thi (br_1, br_2, br_3, br_4) và giá trị ghi trên đảo là 3. Các tổ hợp hợp lệ sẽ là:

- (br_1, br_2, br_3)
- (br_1, br_2, br_4)
- (br_1, br_3, br_4)
- (br_2, br_3, br_4)

Mỗi tổ hợp này tương ứng với một cách chọn 3 cầu để thoả điều kiện số cầu bằng số ghi trên đảo. Tuy nhiên cần lưu ý rằng ở đây chưa xét đến giá trị của các cầu lân cận, chẳng hạn như tổ hợp (1) chắc chắn sẽ không xảy ra do đảo ở vị trí (0, 1) chỉ mang giá trị 1, nhưng những ràng buộc liên quan đến thông tin này sẽ được thêm vào ở bước khác.
Từ những tổ hợp hợp lệ trên, ta có thể xây dựng ràng buộc bằng cách dùng phép `AND` như sau:

- $(br_1 \wedge br_2 \wedge br_3 \wedge \neg br_4)$
- $(br_1 \wedge br_2 \wedge br_4 \wedge \neg br_3)$
- $(br_1 \wedge br_3 \wedge br_4 \wedge \neg br_2)$
- $(br_2 \wedge br_3 \wedge br_4 \wedge \neg br_1)$

Để đảm bảo nghiệm thoả mãn điều kiện ít nhất một tổ hợp hợp lệ, ta dùng phép OR giữa các tổ hợp này và thu được một ràng buộc như sau:

$$\begin{aligned} C_{\text{island\_(0,3)}} = &(br_1 \wedge br_2 \wedge br_3 \wedge \neg br_4) \\ &\vee (br_1 \wedge br_2 \wedge br_4 \wedge \neg br_3) \\ &\vee (br_1 \wedge br_3 \wedge br_4 \wedge \neg br_2) \\ &\vee (br_2 \wedge br_3 \wedge br_4 \wedge \neg br_1) \end{aligned}$$

Tương tự, với mỗi đảo còn lại trong ma trận, ta xây dựng ràng buộc số cầu theo cùng một nguyên tắc như trên.

3.3.3 Ràng buộc các cầu cắt nhau

Để xây dựng các ràng buộc này, trước hết ta cần phải phân nhóm những biến logic biểu diễn cầu thành nhóm Cầu ngang (Horizontal Bridge) và Cầu dọc (Vertical Bridge), cụ thể như sau:

- Nhóm cầu ngang (Horizontal Bridges)

  Đây là các cầu nối giữa hai đảo nằm trên cùng một hàng. Với input đã cho, các biến logic thuộc nhóm này bao gồm các biến: $br_1, br_2, ..., br_{10}$.

- Nhóm cầu dọc (Vertical Bridges)

  Đây là các cầu nối giữa hai đảo nằm trên cùng một cột. Các biến logic thuộc nhóm này bao gồm các biến còn lại: $br_{11}, br_{12}, ..., br_{18}$.

Sau khi kiểm tra toàn bộ các cặp cầu có nguy cơ cắt nhau, ta có thể nhận thấy rằng đối với input này, chỉ có các cầu tạo bởi các đảo có toạ độ $(0, 1)$ với $(4, 1)$ (bao gồm br_{11} và br_{12}) có thể cắt các cầu tạo bởi các đảo có toạ độ $(2, 0)$ và $(2, 2)$ (bao gồm br_5 và br_6). Ta có thể liệt kê các ràng buộc như sau:

- $\neg b_{11} \vee \neg b_5$
- $\neg b_{11} \vee \neg b_6$
- $\neg b_{12} \vee \neg b_5$
- $\neg b_{12} \vee \neg b_6$

3.3.4 Ràng buộc phụ về sự phụ thuộc giữa các cầu

Như đã được trình bày ở mục 3.2.5, đối với mỗi cặp cầu, ta cần thêm ràng buộc để đảm bảo rằng trong tất cả trường hợp, không bao giờ cầu thứ 2 được vẽ mà chưa có cầu 1, có thể được liệt kê như sau:

- $br_1 \vee \neg br_2$
- $br_3 \vee \neg br_4$
- $br_5 \vee \neg br_6$
- $br_7 \vee \neg br_8$
- $br_9 \vee \neg br_{10}$
- $br_{11} \vee \neg br_{12}$
- $br_{13} \vee \neg br_{14}$
- $br_{15} \vee \neg br_{16}$
- $br_{17} \vee \neg br_{18}$

3.4 Cài đặt chương trình

3.4.1 Các nguyên tắc cơ bản về CNF

Khi xây dựng các ràng buộc CNF trong lập trình, cần lưu ý một số nguyên tắc cơ bản sau [5]:

- Mỗi clause thường được biểu diễn dưới dạng: `literal1 literal2 ... 0`, trong đó số 0 biểu thị kết thúc một clause.
- Để biểu diễn một biến logic âm (negative literal), ta sử dụng dấu -.
- Ý nghĩa logic của clause trong ngữ cảnh CNF khi lập trình là:

$$\text{literal1} \vee \text{literal2} \vee \dots$$

  Tức là clause trả về giá trị `True` nếu ít nhất một literal trong clause là `True`.

- Tất cả các clause được kết hợp bằng phép `AND`. Khi giải CNF, mục tiêu là tìm một tập hợp các literal (lời giải) sao cho tất cả các clause đều thỏa mãn (trả về `True`).
- Dựa trên nguyên tắc này, các biểu thức CNF cơ bản có thể được xây dựng theo hai dạng:
  - `-literal1 v -literal2`: clause này chỉ sai (`False`) khi cả hai literal cùng `True`. Nó tương ứng với mệnh đề logic: "Hai biến logic không thể cùng được chọn".
  - `literal1 v literal2`: clause này chỉ sai (`False`) khi cả hai literal cùng `False`. Nó tương ứng với mệnh đề logic: "Phải chọn ít nhất một trong hai biến logic (hoặc cả hai)".

Do sự hạn chế trong việc biểu diễn, nên trong quá trình sinh các ràng buộc, ta cần khéo léo tận dụng hai dạng trên để tạo ra các ràng buộc phù hợp thỏa mãn yêu cầu bài toán.

3.4.2 Sinh biến logic

Đối với 3.2.1, trong quá trình lập trình, các cầu được tạo bằng cách duyệt qua tất cả các cặp đảo theo nguyên tắc sau:

- Với mỗi đảo, chỉ bắt cặp với các đảo có tọa độ khác nó, đảm bảo điều kiện (1).
- Với mỗi đảo, chỉ bắt cặp với các đảo nằm ở bên phải (trên cùng hàng) và bên dưới nó (trên cùng cột), đảm bảo điều kiện (3).

Đối với điều kiện (2), do biến i chỉ nhận giá trị 1 hoặc 2, nên trong trường hợp cả hai biến tương ứng đều `true`, số cầu giữa hai đảo không vượt quá hai, đảm bảo ràng buộc tối đa.

Trong chương trình, phần mã nguồn tạo ra các cầu hợp lệ được cài đặt trong hàm `find_potential_bridges()`. Sau đó, ta sẽ sử dụng hàm `create_variables()` để tạo các biến logic và ánh xạ literal đó thành dạng số nguyên tương ứng để phù hợp với định dạng đầu vào khi đưa vào thuật toán A* hoặc SAT solver.

Mã giả của hai hàm trên được trình bày như sau:

Algorithm 1: Sinh ra tất cả các cầu hợp lệ

Input : Ma trận *grid*, *islands*

Output: Danh sách các cầu *bridges*

```
1  bridges ← ∅;
2  rows ← số hàng của grid;
3  cols ← số cột của grid;
4  foreach start_node ∈ islands do
5      r, c ← start_node.r, start_node.c;
6      for nc ← c + 1 to cols − 1 do
7          if grid[r][nc] ≠ 0 then
8              end_node ← đảo tại vị trí (r, nc) trong islands;
9              Thêm cầu nối đảo start và end vào bridges;
10             break;
11     for nr ← r + 1 to rows − 1 do
12         if grid[nr][c] ≠ 0 then
13             end_node ← đảo tại vị trí (nr, c) trong islands;
14             Thêm cầu nối đảo start_node và end_node vào bridges;
15             break;
16 return bridges;
```

Algorithm 2: Khởi tạo biến logic

Input : $bridges$

Output: $var\_map, reverse\_map$, số lượng biến

```
var_map ← ∅;
reverse_map ← ∅;
counter ← 1;
foreach (u, v, direction) ∈ bridges do
    for i ← 1 to 2 do
        var_map[(u, v, i)] ← counter;
        reverse_map[counter] ← (u, v, i, direction);
        counter ← counter + 1;
return var_map, reverse_map, counter − 1;
```

3.4.3 Tự động sinh ràng buộc CNF

Sau khi đã có các biến logic hợp lệ, việc sinh ra các clause theo ý tưởng được trình bày ở mục 3.2.2 và 3.2.3 sẽ được thực hiện theo các bước sau:

1. **Tạo ràng buộc số cầu**

   Do ta không thể trực tiếp biểu diễn điều kiện so sánh giữa số ghi trên đảo với số cầu nối dưới dạng CNF. Chúng ta có thể tự lập trình để tạo ra các clause dựa trên các nguyên tắc toán học hoặc sử dụng thư viện hỗ trợ. Tuy nhiên, để đơn giản hoá quá trình tạo CNF thoả điều kiện này, ta sẽ sử dụng phương thức `PBEnc.equals` từ thư viện PySAT để tự sinh các clause phù hợp. Trong chương trình, hàm được sử dụng để sinh ra ràng buộc cho số cầu là `generate_capacity_constraints()`, có mã giả như sau:

Algorithm 3: Sinh ràng buộc giá trị đảo

Input : $islands$, $bridges$, var_map, top_id

Output: Tập các mệnh đề CNF, $current\_max\_id$ mới

```
1  cnf_clauses ← ∅;
2  current_max_id ← top_id;
3  foreach island ∈ islands do
4      connected_vars ← ∅;
5      foreach (u, v, direction) ∈ bridges do
6          if u = island or v = island then
7              Thêm cầu 1, 2 vào connected_vars
8      pb ← PBEnc.equals(lits = connected_vars, bound =
           island.val, top_id = current_max_id);
9      Thêm pb.clauses vào cnf_clauses;
10     Cập nhật lại current_max_id;
11 return cnf_clauses, current_max_id;
```

Trong mã giả, có một biến tương đối lạ, chưa từng được đề cập trước đây, đó là `current_max_id`, biến này được cài đặt với mục đích theo dõi chỉ số biến lớn nhất đã được sử dụng, nhằm đảm bảo rằng các biến phụ sinh ra trong quá trình mã hoá ràng buộc pseudo-Boolean là duy nhất và không trùng với các biến đã tồn tại trong công thức CNF, tránh dẫn tới sự sai lệch về kết quả khi đưa vào solver của PySAT (sẽ được nhắc đến ở phần sau)

2. **Tạo ràng buộc không giao nhau**

   Để sinh các ràng buộc CNF, chúng ta xét tất cả các cặp cầu khả thi $(b_{\text{hor}}, b_{\text{ver}})$ gồm một cầu ngang và một cầu dọc theo ý tưởng được trình bày ở mục 3.2.3

   Trong trường hợp này, khi lập trình, ta thêm clause CNF đối với mỗi cặp cầu được xác định là cắt nhau, ví dụ như sau:

$$\neg b_{\text{hor}} \vee \neg b_{\text{ver}}$$

   để đảm bảo ít nhất một trong hai cầu không xuất hiện trong lời giải cuối cùng.

Trong chương trình, phần mã nguồn cho công việc này được cài đặt ở hàm `generate_crossing_constraints()`, có mã giả như dưới đây:

Algorithm 4: Sinh ràng buộc các cầu không cắt nhau

Input : Danh sách các cầu *bridges*, bản đồ biến *var_map*

Output: Tập các mệnh đề *clauses* cấm cắt nhau

```
clauses ← ∅;
horizontal_bridges ← {b ∈ bridges | b.direction = 'H'};
vertical_bridges ← {b ∈ bridges | b.direction = 'V'};
foreach h ∈ horizontal_bridges do
    foreach v ∈ vertical_bridges do
        if hai cầu h và v cắt nhau then
            Thêm mệnh đề {−h, −v} vào clauses;
return clauses;
```

3. **Giới hạn thứ tự cầu**

Ý tưởng của phần này đã được nêu rõ ở mục 3.2.5, và yêu cầu này được biểu diễn dưới dạng clause như sau:

$$\neg br\_(r_u, c_u)\_(r_v, c_v)\_1 \vee br\_(r_u, c_u)\_(r_v, c_v)\_2$$

Dễ thấy, việc lập trình điều kiện này sao cho phù hợp với nguyên tắc tạo clause ở phần 3.4.1 là đơn giản, vì nó chỉ bao gồm các phép `OR`. Trong mã nguồn, nó được cài đặt thông qua hàm `generate_bridge_dependency_constraints()` với ý tưởng như sau:

Algorithm 5: Sinh ràng buộc phụ thuộc cầu

Input : *bridges*, bản đồ biến *var_map*

Output: Tập các mệnh đề *clauses* đảm bảo thứ tự cầu

```
clauses ← ∅;
foreach (u, v, direction) ∈ bridges do
    v1 ← var_map[(u, v, 1)];
    v2 ← var_map[(u, v, 2)];
    Thêm mệnh đề {−v2, v1} vào clauses;
return clauses;
```

Cần lưu ý rằng, trong quá trình sinh các ràng buộc, có thể dẫn đến những sự trùng lặp giữa các clause, gây ra các chi phí dư thừa trong quá trình tìm kiếm lời giải và lưu trữ. Vì vậy, trước khi trả về các clause CNF hợp lệ, ta sẽ tiến hành loại bỏ các clause trùng lặp thông qua hàm `remove_duplicate_clauses()`, cụ thể là theo mã giả dưới đây:

Algorithm 6: Loại bỏ mệnh đề trùng lặp

Input : Tập các mệnh đề *cnf_clauses*

Output: Tập các mệnh đề duy nhất *unique_clauses*

```
seen ← ∅;
unique_clauses ← ∅;
foreach cl ∈ cnf_clauses do
    cl_sorted ← tuple(sort(cl));
    if cl_sorted ∉ seen then
        Thêm cl_sorted vào seen;
        Thêm cl vào unique_clauses;
return unique_clauses;
```

Như đã trình bày ở mục 3.2.4, việc kiểm tra tính liên thông sẽ được thực hiện đối với từng bộ lời giải, và các clause ràng buộc thêm sẽ được thêm vào sau nếu cần thiết và phần này sẽ được cài đặt riêng ở từng thuật toán. Cụ thể, đối với trường hợp đồ thị không liên thông, nghiệm hiện tại bị loại bỏ bằng cách thêm vào một Blocking clause phủ định toàn bộ model. Trong đó, công thức Blocking Clause được

mô tả như sau: Với một nghiệm S cụ thể, để loại bỏ nghiệm này khỏi không gian tìm kiếm, ta thêm vào hệ CNF một mệnh đề phủ định của S, gọi là *blocking clause*:

$$C_{\text{block}} = \neg S = \bigvee_{e \in E} \text{change}(e, S_e)$$

Hiểu một cách đơn giản, để có thể "chặn"được nghiệm không hợp lệ trên, ít nhất một biến trong nghiệm đó phải bị đảo ngược để nghiệm này không còn được chấp nhận, vì vậy, ta sử dụng phép `OR`.

Trong bối cảnh đồ án này, ta chỉ sử dụng cách thức thêm clause này đối với chương trình giải bài toán Hashiwokakero sử dụng Solver của thư viện PySAT. Ngoài ra, việc kiểm tra liên thông áp dụng trong toàn bộ đồ án được cài đặt bằng cách áp dụng thuật toán BFS (tìm kiếm theo chiều rộng).

3.4.4 Mã giả

Ở các phần phân tích trên, các mã giả để cài đặt từng bước trong quá trình sinh CNF đã được trình bày, để tổng hợp lại quá trình sinh CNF và trả về kết quả cuối cùng, ta có thể thực hiện từng bước theo mã giả dưới đây:

Algorithm 7: Quy trình tổng quát sinh CNF cho bài toán Hashi

Input : Ma trận $grid$

Output: Tập mệnh đề CNF_Final

$Islands \leftarrow$ ExtractIslands($grid$);

$Bridges \leftarrow$ FindPotentialBridges($grid, Islands$);

$(VarMap, RevMap, TopID) \leftarrow$ CreateVariables($Bridges$);

$CNF \leftarrow \emptyset$;

$CNF \leftarrow$

$CNF \cup$ GenerateBridgeDependencyConstraints($Bridges, VarMap$);

$CNF \leftarrow CNF \cup$ GenerateCrossingConstraints($Bridges, VarMap$);

$(CapacityClauses, NewTopID) \leftarrow$

GenerateCapacityConstraints($Islands, Bridges, VarMap, TopID$);

$CNF \leftarrow CNF \cup CapacityClauses$;

$CNF\_Final \leftarrow$ RemoveDuplicateClauses(CNF);

return CNF_Final;

4 Phương pháp và thuật toán

4.1 Thư viện PySAT

4.1.1 Giới thiệu thư viện PySAT trong bài toán Hashiwokakero

Sau khi các luật của bài toán Hashiwokakero đã được chuyển đổi thành hệ ràng buộc CNF (Conjunctive Normal Form), chúng em sử dụng thư viện PySAT để giải quyết hệ này. PySAT là thư viện Python hỗ trợ giải bài toán SAT (Satisfiability Problem) một cách hiệu quả, tích hợp các solver hiện đại như Glucose3 [7] - một Conflict-Driven Clause Learning. Trong bài toán này, PySAT đóng vai trò là bộ giải lõi, chịu trách nhiệm tìm assignment cho các biến logic đại diện cho cầu giữa các đảo sao cho toàn bộ các clause đều satisfiable.

Đặc biệt, PySAT hỗ trợ incremental solving [6], cho phép thêm clause động mà không cần khởi tạo lại solver. Điều này rất hữu ích khi model satisfiable về CNF nhưng không liên thông: chúng em block model đó bằng clause phủ định và gọi `solve()` lại.

4.1.2 Định dạng input và output

Input cho PySAT Solver

Đầu vào của PySAT solver là hệ ràng buộc logic của bài toán Hashiwokakero, được biểu diễn dưới dạng Conjunctive Normal Form (CNF) (được trình bài trong Mục 2). CNF được lưu dưới dạng một danh sách các clause, trong đó mỗi clause là một danh sách các số nguyên (literal).

Mỗi biến logic được đánh số bắt đầu từ 1 và đại diện cho một cầu tiềm năng giữa hai đảo hợp lệ. Literal dương biểu thị biến được gán giá trị TRUE (cầu tồn tại), trong khi literal âm biểu thị biến được gán giá trị FALSE (cầu không tồn tại).

Ví dụ: Clause $[5, -7]$ tương ứng với mệnh đề logic:

$$(x_5 \vee \neg x_7)$$

Nghĩa là hoặc cầu số 5 tồn tại, hoặc cầu số 7 không tồn tại. Trong một test case kích thước 7x7, hệ CNF thường gồm khoảng 150-250 clause và 80-120 biến logic,

tương ứng với các cầu đơn và cầu đôi tiềm năng giữa các đảo.
Toàn bộ hệ CNF này được sinh tự động từ dữ liệu đầu vào của bài toán và được sử dụng trực tiếp làm input cho PySAT solver.

Output cho PySAT Solver

Sau khi giải hệ CNF, PySAT solver trả về hai loại kết quả:

- Satisfiable (SAT): tồn tại ít nhất một nghiệm thỏa mãn toàn bộ CNF.
- Unsatisfiable (UNSAT): không tồn tại nghiệm hợp lệ.

Trong trường hợp SAT, solver trả về một model SAT, là một danh sách các số nguyên có dấu, biểu diễn trạng thái TRUE/FALSE của từng biến logic. Ví dụ: model trả về:

$$[-1, 2, -3, 4, 5, -6, ..., 85, -86]$$

Trong model trên, các biến $2, 4, 5, 85$ được gán giá trị TRUE, nghĩa là các cầu tương ứng được chọn trong nghiệm.
Diễn giải model SAT thành kết quả Hashiwokakero: Để chuyển model SAT thành lời giải Hashiwokakero, các literal dương trong model được lọc ra và ánh xạ ngược về thông tin cầu thông qua cấu trúc dữ liệu `reverse_map`. Mỗi biến TRUE cho biết:

- Hai đảo được nối
- Số lượng cầu (1 hoặc 2)
- Hướng cầu (ngang hoặc dọc)

Các cầu giữa cùng một cặp đảo được gom nhóm để xác định đó là cầu đơn hay cầu đôi. Dựa trên hướng và số lượng cầu, ký tự tương ứng được điền vào các ô trống giữa hai đảo trên lưới:

- -: cầu đơn ngang
- =: cầu đôi ngang
- |: cầu đơn dọc

- $: cầu đôi dọc

Kết quả cuối cùng được xuất ra dưới dạng ma trận chuỗi, đúng theo định dạng đầu ra của đề bài.

4.1.3 Quy trình triển khai PySAT

Trọng tâm của quy trình là sử dụng SAT solver để tìm nghiệm thỏa mãn các ràng buộc logic, sau đó kiểm tra và lọc nghiệm theo yêu cầu đặc thù của bài toán Hashiwokakero. Quy trình tổng thể gồm 4 bước chính.

Bước 1: Chuẩn bị dữ liệu CNF

Hệ ràng buộc CNF được sinh ra thông qua lớp `CNFGenerator`, trả về tập clause cùng các cấu trúc hỗ trợ như `reverse_map, islands, bridges, var_map` và `num_vars`. Sinh CNF sẽ được thiết kế theo hướng module hóa, trong đó các ràng buộc exactly-k được mã hóa bằng PBEnc với cơ chế quản lý biến phụ động để tránh trùng lặp.

Bước 2: Khởi tạo Solver

SAT solver được khởi tạo bằng `Glucose3(bootstrap_with=clauses)` để nạp toàn bộ hệ CNF ban đầu. Việc sử dụng Glucose3 cho phép áp dụng incremental SAT solving dựa trên CDCL, giúp solver có thể bổ sung clause mới trong quá trình giải mà không cần khởi tạo lại, từ đó tối ưu hiệu suất.

Bước 3: Vòng lặp giải và kiểm tra nghiệm

Solver được gọi trong vòng lặp `while solver.solve()`. Với mỗi model thu được, các biến TRUE được ánh xạ thành các cầu và kiểm tra tính liên thông bằng BFS trên đồ thị đảo. Nếu nghiệm không liên thông, một blocking clause được thêm vào để loại nghiệm hiện tại, tận dụng cơ chế học clause của solver.

Bước 4: Xử lý kết quả

Thời gian thực thi được đo bằng `time.perf_counter()` để phục vụ đánh giá hiệu suất. Nếu tìm được nghiệm hợp lệ, hàm trả về model cùng các cấu trúc ánh xạ cần

thiết để xây dựng lưới kết quả; ngược lại, trả về `None` nếu không tồn tại nghiệm thỏa mãn.

4.1.4 Phân tích và đánh giá

Sử dụng PySAT là lời giải trong bài toán Hashiwakakero mang lại nhiều ưu điểm rõ rệt. Điểm mạnh lớn nhất của PySAT là khả năng xử lý hiệu quả các ràng buộc logic phức tạp thông qua mô hình hóa CNF, giúp giải bài toán Hashiwokakero mà không cần duyệt trực tiếp toàn bộ không gian trạng thái. Nhờ cơ chế học xung đột và incremental solving, SAT solver có thể loại bỏ sớm các cấu hình không hợp lệ và tìm nghiệm nhanh hơn so với các phương pháp tìm kiếm truyền thống. Tuy nhiên, việc sử dụng PySAT cũng đòi hỏi chi phí mô hình hóa lớn, đòi hỏi thiết kế ánh xạ biến và hệ ràng buộc cẩn thận; nếu không kiểm soát tốt, số lượng biến và clause có thể tăng nhanh và gây khó khăn trong việc diễn giải nghiệm.

4.2 Thuật toán tìm kiếm A*

4.2.1 Giới thiệu

A* là thuật toán tìm kiếm có thông tin trên không gian trạng thái, kết hợp giữa chi phí đã đi $g(n)$ và hàm heuristic $h(n)$ để đánh giá mức độ "hứa hẹn" của mỗi trạng thái thông qua hàm [3]:

$$f(n) = g(n) + h(n)$$

Trong đồ án này, A* được áp dụng để giải bài toán Hashiwokakero theo hướng tìm kiếm assignment cho các biến logic trong hệ CNF, thay vì sử dụng SAT solver chuyên biệt như PySAT. Thay vì kiểm tra tất cả các tổ hợp (như brute-force), A* sử dụng heuristic để ưu tiên các trạng thái hứa hẹn, giảm không gian tìm kiếm.
Cách áp dụng A* vào CNF: Không gian trạng thái được biểu diễn như một cây quyết định, nơi mỗi nút là một partial assignment cho các cặp cầu giữa đảo (0, 1, hoặc 2 cầu). Mỗi bước, A* mở rộng trạng thái bằng cách gán giá trị cho một cặp cầu tiếp theo, kiểm tra tính nhất quán với CNF, và sử dụng heuristic dựa trên tính liên thông để hướng dẫn tìm kiếm đến goal (assignment đầy đủ, satisfiable CNF, và đồ thị đảo liên thông).

Để tối ưu hóa mã nguồn, chúng em sử dụng thêm các module tách biệt như cnf cho việc sinh CNF, connectivity cho kiểm tra liên thông, và is_intersect cho kiểm tra giao cắt, giúp code dễ bảo trì và mở rộng hơn.

4.2.2 Hàm Heuristic

Hàm heuristic $h(x)$ trong A* được sử dụng để ước lượng chi phí còn lại từ trạng thái hiện tại đến trạng thái mục tiêu, đóng vai trò định hướng quá trình tìm kiếm. Để đảm bảo tính tối ưu của A*, heuristic cần thỏa mãn hai tính chất quan trọng là admissible (không đánh giá vượt quá chi phí thực tế) và consistent.

Trong đồ án này, heuristic được thiết kế dựa trên mức độ liên thông của đồ thị đảo tại mỗi trạng thái. Cụ thể, với mỗi trạng thái x, số thành phần liên thông $components(x)$ của đồ thị đảo được tính bằng BFS. Hàm heuristic được định nghĩa như sau:

$$h(x) = (components(x) - 1) * 10$$

Ý tưởng chính: để nối k thành phần liên thông thành một đồ thị duy nhất, cần ít nhất $k - 1$ cầu, tương tự bài toán Minimum Spanning Tree. Do đó, heuristic này không vượt quá chi phí thực tế cần thiết để đạt trạng thái goal, đảm bảo tính admissible. Hệ số nhân 10 được sử dụng nhằm tăng mức ưu tiên cho các trạng thái gần liên thông hơn, giúp A* tập trung mở rộng các trạng thái hứa hẹn mà không loại bỏ sớm các nhánh hợp lệ.

Ngoài ra còn có $g(x)$, $g(x)$ là hàm chi phí đã đi từ trạng thái ban đầu đến trạng thái hiện tại x. Trong implement này, $g(x)$ được tính bằng số cặp cầu đã gán (tức depth của trạng thái trong cây tìm kiếm, tăng 1 mỗi khi mở rộng nhánh).

4.2.3 Quá trình tìm kiếm trạng thái và chiến lược cắt tỉa

Quá trình tìm kiếm của thuật toán A* trong bài toán Hashiwokakero có thể được mô hình hóa dưới dạng một cây tìm kiếm, trong đó mỗi nút biểu diễn một trạng thái tương ứng với một partial assignment cho các cặp cầu tiềm năng. Từ mỗi trạng thái, thuật toán mở rộng sang các trạng thái con bằng cách gán số cầu 0, 1 hoặc 2 cho cặp cầu tiếp theo, tạo ra cây tìm kiếm với hệ số phân nhánh là 3.

Để giảm kích thước không gian tìm kiếm vốn rất lớn, nhóm sử dụng `chiến lược cắt tỉa nhánh (pruning)` trong quá trình sinh trạng thái. Cụ thể, một trạng thái sẽ bị loại bỏ ngay lập tức nếu vi phạm một trong các điều kiện cục bộ sau:

- Giao nhau (Crossing): nếu việc gán cầu mới làm phát sinh cầu cắt ngang với một cầu đã được xây dựng trước đó, trạng thái này bị cắt bỏ vì vi phạm ràng buộc hình học của bài toán.
- Quá tải cục bộ (Partial Degree Exceeded): nếu tổng số cầu nối vào một trong hai đảo đầu mút vượt quá giá trị yêu cầu của đảo đó, trạng thái tương ứng sẽ bị loại bỏ ngay mà không cần mở rộng thêm.

Nhờ các chiến lược cắt tỉa này, nhiều nhánh không hợp lệ bị loại bỏ sớm, giúp giảm đáng kể số trạng thái cần đánh giá so với việc duyệt toàn bộ không gian 3^N. Sau khi vượt qua bước pruning, các trạng thái hợp lệ mới được đưa vào hàng đợi ưu tiên và tiếp tục được đánh giá bằng hàm heuristic trong A*.

4.2.4 Mã giả

Algorithm 8: Thuật toán A* giải bài toán Hashiwokakero

Input: Puzzle P (ma trận grid)

Output: Trạng thái S_{goal} hoặc *null*

Preprocessing: Sinh CNF dùng $CNFGenerator$, tạo $bridge\_pairs$, $island\_incident$, $crossing\_pairs$ dùng $is\_intersect$;
$S_{init} \leftarrow$ Trạng thái ban đầu ($index = 0$, components = số đảo);
$PQ \leftarrow$ Hàng đợi ưu tiên (min-heap);
$PQ.push(S_{init}, f = h(S_{init}), g = 0)$;
while *PQ không rỗng* **do**
    $S \leftarrow PQ.pop()$; `// Lấy trạng thái có f nhỏ nhất`
    if $S.index \geq$ *số bridge_pairs* ***và*** $CheckConnected(S)$ **then**
        return S ; `// Tìm thấy giải pháp`
    else
        if $S.index \geq$ *số bridge_pairs* **then**
            continue;
    for $val \in \{0, 1, 2\}$ **do**
        $S_{new} \leftarrow S.copy()$ + gán val cho $bridge\_pairs[S.index]$;
        if $CheckCrossing(S_{new})$ ***hoặc*** $CheckPartialDegreeExceeded(S_{new})$ **then**
            continue ; `// Cắt tỉa nhánh`
        $h_{new} \leftarrow (num\_components(S_{new}) - 1) \times 10$;
        $g_{new} \leftarrow g(S) + 1$;
        $f_{new} \leftarrow g_{new} + h_{new}$;
        $PQ.push(S_{new}, f_{new})$;
return null;

4.2.5 Phân tích và đánh giá

Việc áp dụng thuật toán A* cho bài toán Hashiwokakero cho thấy đây là một phương pháp mạnh về mặt tư duy tìm kiếm và phù hợp để minh họa rõ ràng cách kết hợp giữa ràng buộc logic và heuristic trong AI. A* cho phép kiểm soát trực tiếp quá trình mở rộng trạng thái, nhờ đó dễ dàng tích hợp các chiến lược cắt tỉa và kiểm tra tính hợp lệ ngay trong quá trình tìm kiếm. Điều này giúp thuật toán tránh được nhiều nhánh không khả thi và cải thiện hiệu suất so với các phương pháp duyệt mù. Tuy nhiên, hạn chế lớn nhất của A* nằm ở khả năng mở rộng. Khi kích thước bài toán tăng, số lượng trạng thái trung gian cần lưu trữ trong hàng đợi ưu tiên có thể tăng nhanh, dẫn đến chi phí bộ nhớ đáng kể. Ngoài ra, hiệu quả của A* phụ thuộc mạnh vào chất lượng heuristic; nếu heuristic chưa đủ tốt, lợi thế của A* so với các phương pháp tìm kiếm khác sẽ giảm rõ rệt.

Tổng thể, A* là một lựa chọn phù hợp cho các instance nhỏ và trung bình, đồng thời có giá trị cao về mặt học thuật và minh họa thuật toán. Trong bối cảnh đồ án, A* đóng vai trò như một phương pháp đối chứng có kiểm soát, giúp làm nổi bật ưu thế về hiệu suất và mức độ tự động hóa của các SAT solver chuyên biệt.

4.3 Thuật toán quay lui (Backtracking)

4.3.1 Ý tưởng

Ý tưởng của thuật toán này là sử dụng đệ quy, vừa xây dựng lời giải vừa kiểm tra việc thỏa mãn các yêu cầu của bài toán để có thể đưa ra lời giải cuối cùng cho bài toán, nếu một lựa chọn dẫn đến trạng thái không hợp lệ, thuật toán sẽ quay lui về trạng thái trước đó để tiếp tục tìm kiếm lời giải khác [2]. Đặt vào bối cảnh bài toán, trước khi thử đặt cầu, thuật toán sẽ kiểm tra liệu cầu này có vi phạm luật của bài toán không thông qua các hàm tự cài đặt. Nếu có vi phạm xảy ra, thuật toán sẽ không tiếp tục đào sâu mà trực tiếp bỏ qua cây cầu đó để tìm kiếm các lối đi khác. Ngược lại, nếu một cây cầu được xây thỏa mãn các yêu cầu bài toán, thuật toán sẽ tiếp tục đi theo hướng đó, tìm kiếm sâu hơn các đường đi tiếp theo. Nếu đi đến cuối đường mà vẫn không thể tìm ra được kết quả, cây cầu vừa đặt sẽ được xóa và các phương án tiếp theo sẽ được tiếp tục thử nghiệm. Không như thư viện `PySAT` hay

thuật toán tìm kiếm A*, quay lui được thực hiện trực tiếp trên mã trận biểu diễn bài toán thay vì thông qua việc giải CNF.

4.3.2 Mã giả

Algorithm 9: Thuật toán Backtracking giải bài toán Hashiwokakero

Input: Puzzle P (ma trận grid)

Output: Ma trận kết quả hoặc *null*

```
1  Preprocessing: Xác định I, tạo danh sách cạnh E, khởi tạo
     current_degree = 0;
2  Function Backtrack(index):
3      if index = |E| then
4          if CheckAllDegreesSatisfied(I) và CheckConnected(I, E) then
5              return True ;          // Đã điền đủ và đồ thị liên thông
6          return False;
7      else
8          edge ← E[index];
9          for val ∈ {0, 1, 2} do
               // Kiểm tra điều kiện cắt tỉa (Pruning)
10             if CheckPartialDegreeExceeded(edge, val) hoặc (val > 0 và
                 CheckCrossing(edge)) then
11                 continue;
12             PlaceBridge(edge, val);
13             UpdateDegrees(edge, val);
14             if Backtrack(index + 1) then
15                 return True ;      // Tìm thấy giải pháp từ nhánh con
16             RemoveBridge(edge, val) ;                    // Backtrack
17             RevertDegrees(edge, val);
18         return False;
19 if Backtrack(0) then
20     return BuildOutput(P, E);
21 else
22     return null;
```

4.4 Thuật toán vét cạn (Brute-force)

4.4.1 Ý tưởng

Đây là thuật toán ngây thơ nhất để giải bài toán. Tương tự như backtracking, brute-force được thực hiện trên ma trận biểu diễn bài toán. Tuy nhiên, ý tưởng cốt lõi của nó là sinh ra toàn bộ các lời giải (duyệt toán bộ không gian tìm kiếm) và kiểm tra từng cái cho đến khi tìm được lời giải chính xác cho bài toán [1]. Vì thuật toán không quan tâm những trạng thái mà nó sinh ra có đang hợp lệ hay không mà chỉ kiểm tra tính hợp lệ sau khi một đường đi được thiết lập nên thuật toán sẽ hoàn toàn không học được gì từ những sai lầm trước đó. Đây cũng chính là điểm khiến brute-force kém hiệu quả hơn so với backtracking, vốn sẽ dừng ngay lập tức nếu phát hiện một cây cầu sai, thay vì tiếp tục tìm kiếm với cây cầu đó như thuật toán vét cạn. Vì độ phức tạp của thuật toán là rất lớn nên khi cài đặt, thuật toán được cài đặt để kích thước đầu vào lớn nhất mà nó có thể nhận là 9×9.

4.4.2 Mã giả

Algorithm 10: Thuật toán Brute Force giải bài toán Hashiwokakero

Input: Puzzle P (ma trận grid)

Output: Ma trận kết quả hoặc *null*

Preprocessing: Xác định danh sách đảo I, tạo danh sách các cạnh có thể E (chỉ xét hướng phải và dưới);
$N \leftarrow$ Số lượng cạnh trong E;
$TotalConfigs \leftarrow 3^N$;
for $config \in \{0, 1, 2\}^N$ **do**
    `// Duyệt qua tất cả cấu hình (0, 1, 2 cầu)`
    Assign $config$ cho E ; `// Gán số lượng cầu cho từng cạnh`
    if *CheckDegreeValid*(I, E) ***và*** *!CheckCrossing*(E) **then**
        if *CheckConnected*(I, E) **then**
            return BuildOutput(P, E) ; `// Tìm thấy giải pháp hợp lệ`
return null ; `// Không tìm thấy giải pháp sau khi duyệt hết`

5 Thử nghiệm và kết quả

5.1 Thử nghiệm

5.1.1 Tiêu chí đánh giá mức độ hiệu quả

Để đánh giá mức độ hiệu quả đối với bài toán Hashiwokakero, các phương pháp không chỉ được đánh giá dựa trên việc nó có tìm ra được đáp án chính xác không mà còn phải xem xét liệu phương pháp đó tốn bao nhiêu thời gian để tìm được kết quả. Thời gian chạy của các thuật toán được đo bằng thư viện `time` của ngôn ngữ lập trình `Python`. Một đơn vị khác cũng sẽ được so sánh ở trong phần tiếp theo là mức độ tiêu thụ bộ nhớ của các thuật toán.

Đối với những trường hợp bài toán có kích thước quá lớn hoặc quá phức tạp, thời gian thực thi của các thuật toán có thể quá lớn để tìm được đáp án. Do đó, các phương pháp đều được dùng chung một giới hạn thời gian. Nếu để tìm được đáp án mà phương pháp đó lại cần nhiều hơn mức thời gian đã quy định, điều đó đồng nghĩa với việc thuật toán có thể không quá hiệu quả và đồng thời ngay lập tức dừng thuật toán, tránh lãng phí nhiều thời gian.

5.1.2 Tập dữ liệu đầu vào và đầu ra

Để đảm bảo sự công bằng, các phương pháp sẽ lần lượt được chạy thử trên mười bộ dữ liệu kiểm thử (bộ test) khác nhau. Các bộ kiểm thử được đặt chung trong thư mục `Inputs` và các tệp được đặt tên theo dạng `input-xx.txt` với `xx` là số thứ tự của bộ test đó. Ví dụ: `input-01.txt`, `input-02.txt`, `input-03.txt`,... Tương ứng với từng đầu vào, ta thiết lập một thư mục `Solutions` gồm các tệp `solution-xx.txt` chứa lời giải chính xác tương ứng của câu hỏi trong tệp `input-xx.txt`. Cuối cùng, ta có một thư mục `Outputs` chứa kết quả các thuật toán chạy được, kết quả của từng thuật toán được đặt vào thư mục với tên của thuật toán. Quy định đặt tên của các tệp trong thư mục này cũng tuân thủ như các tệp trước đó, có dạng `output-xx.txt`. Việc so sánh kết quả chạy ra trong tệp `output` với `solution` giúp xác minh độ chính xác của thuật toán.

Dữ liệu biểu diễn trong các file `input`, `solution` hay `output` đều được biểu diễn

theo dạng đã được đề cập tại phần 2.2. Tuy nhiên, trong các phương pháp tốn nhiều hơn thời gian giới hạn mà vẫn chưa tìm được kết quả, file `ouput` của sẽ in ra vấn đề gặp phải, ví dụ: `NO SOLUTION: Timeout (> 60s)`.

Để có thể thử nghiệm trên nhiều trường hợp, các bộ test có kích thước và số lượng đảo khác nhau. Chi tiết các bộ test được thể hiện trong bảng 2.

Tên tệp	**Kích thước**	**Số lượng đảo**
input-01.txt	7×7	12
input-02.txt	7×7	18
input-03.txt	7×7	17
input-04.txt	7×7	16
input-05.txt	9×9	26
input-06.txt	11×11	32
input-07.txt	11×11	39
input-08.txt	13×13	59
input-09.txt	17×17	97
input-10.txt	20×20	126

Bảng 2: Bộ dữ liệu kiểm thử

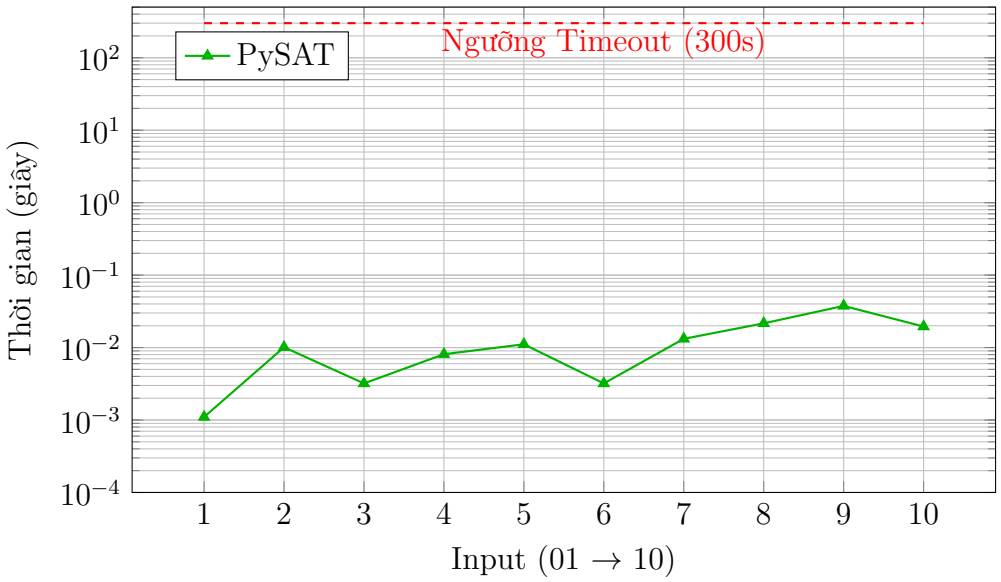
5.2 Kết quả và phân tích

5.2.1 Thư viện PySAT

Kết quả thực nghiệm giải bài toán Hashiwokakero dùng thư viện PySAT được thể hiện qua Bảng 3 và Hình 1

Input	Trạng thái	Thời gian (s)	Bộ nhớ (MB)
input-01	CORRECT	0.0011	0.07
input-02	CORRECT	0.0101	0.15
input-03	CORRECT	0.0032	0.14
input-04	CORRECT	0.0081	0.12
input-05	CORRECT	0.0111	0.33
input-06	CORRECT	0.0032	0.5
input-07	CORRECT	0.0132	0.74
input-08	CORRECT	0.0216	1.71
input-09	CORRECT	0.0376	4.66
input-10	CORRECT	0.0195	7.88

Bảng 3: Bảng kết quả dùng thư viện PySAT



Hình 1: Thời gian chạy của thư viện PySAT theo từng input

Nhận xét: Ta có thể thấy thời gian chạy của PySAT tăng nhẹ theo input nhưng vẫn dao động ở mức rất thấp (khoảng mili-giây), cho thấy hiệu năng ổn định. Tất cả các input đều cách rất xa ngưỡng timeout 300s, chứng tỏ bài toán được xử lý rất hiệu quả. Sự dao động nhỏ giữa các input phản ánh độ phức tạp cấu trúc bài toán hơn là sự suy giảm hiệu năng của solver.

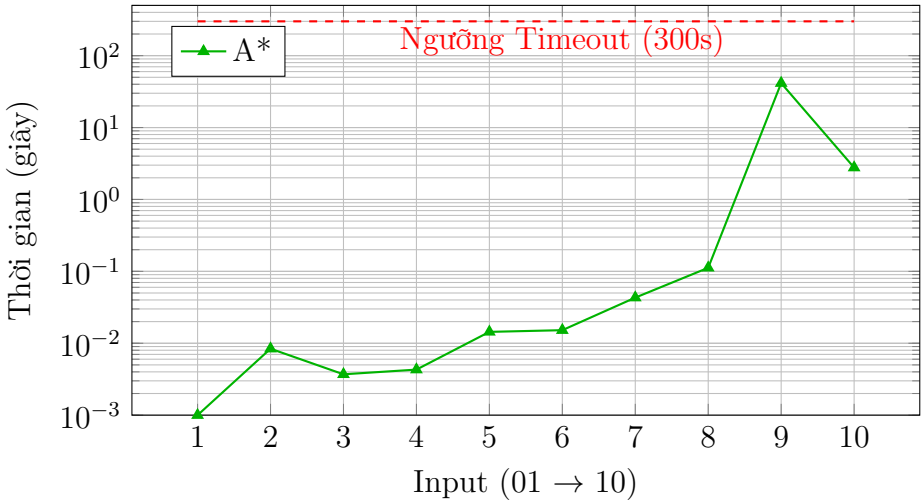
5.2.2 Thuật toán tìm kiếm A*

Kết quả thực nghiệm giải bài toán Hashiwokakero dùng thuật toán A* được thể hiện qua Bảng 4 và Hình 2

Input	**Trạng thái**	**Thời gian (s)**	**Bộ nhớ (MB)**
input-01	CORRECT	0.001	0.14
input-02	CORRECT	0.0084	0.32
input-03	CORRECT	0.0037	0.29
input-04	CORRECT	0.0043	0.26
input-05	CORRECT	0.0144	0.68
input-06	CORRECT	0.0152	1.02
input-07	CORRECT	0.0431	1.52
input-08	CORRECT	0.1121	3.48
input-09	CORRECT	41.4991	9.41
input-10	CORRECT	2.7741	15.88

Bảng 4: Bảng kết quả thuật toán tìm kiếm A*

Nhận xét: Ta có thể thấy thấy thời gian chạy của thuật toán A* tăng nhanh khi độ phức tạp của input lớn dần, đặc biệt ở các input có số lượng cầu tiềm năng cao. Mặc dù A* vẫn tìm được nghiệm trong thời gian cho phép, sự gia tăng đột biến ở



Hình 2: Thời gian chạy của thuật toán tìm kiếm A* theo từng input

các input lớn phản ánh chi phí tìm kiếm và lưu trữ trạng thái của thuật toán. Điều này cho thấy A* phù hợp hơn với các instance nhỏ và trung bình, trong khi hiệu suất giảm rõ rệt khi kích thước bài toán tăng.

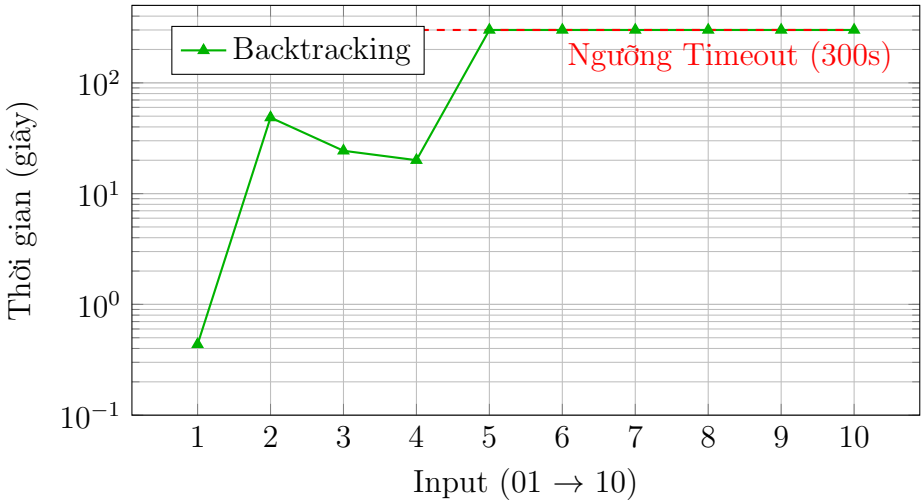
5.2.3 Thuật toán Backtracking

Kết quả thực nghiệm giải bài toán Hashiwokakero dùng thuật toán Backtracking được thể hiện qua Bảng 5 và Hình 3

Input	**Trạng thái**	**Thời gian (s)**	**Bộ nhớ (MB)**
input-01	CORRECT	0.4358	0.12
input-02	CORRECT	48.5978	0.18
input-03	CORRECT	24.3637	0.17
input-04	CORRECT	20.0300	0.16
input-05 -> 10	**Timeout**	>300.00	-

Bảng 5: Bảng kết quả thuật toán Backtracking

Nhận xét: Kết quả cho thấy thuật toán Backtracking chỉ hoạt động hiệu quả với các input nhỏ, trong khi thời gian chạy tăng rất nhanh khi độ phức tạp bài toán



Hình 3: Thời gian chạy của thuật Backtracking theo từng input

tăng. Nhiều input trung bình và lớn không thể hoàn thành trong giới hạn thời gian 300 giây, phản ánh sự bùng nổ không gian tìm kiếm của phương pháp này. Điều này cho thấy Backtracking thiếu khả năng mở rộng và không phù hợp để giải bài toán Hashiwokakero ở các instance lớn.

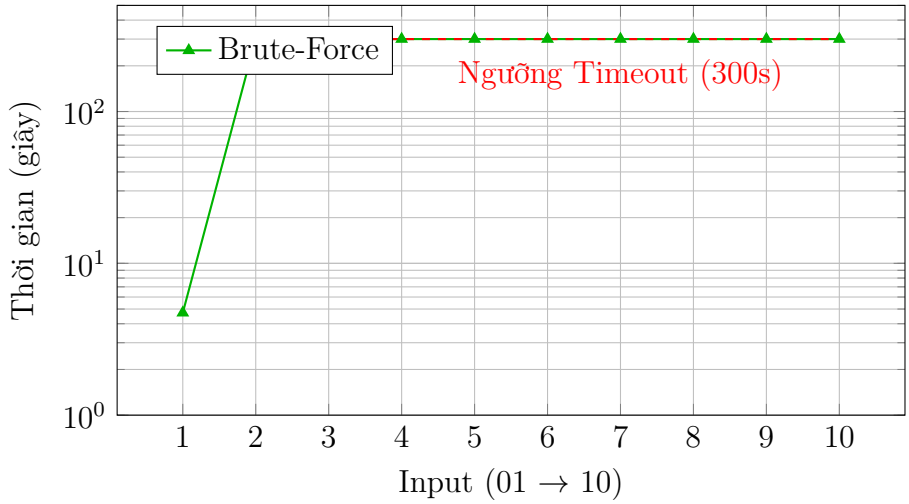
5.2.4 Thuật toán Brute-Force

Kết quả thực nghiệm giải bài toán Hashiwokakero dùng thuật toán Brute-Force được thể hiện qua Bảng 6 và Hình 4

Input	**Trạng thái**	**Thời gian(s)**	**Bộ nhớ (MB)**
input-01	CORRECT	4.7252	0.01
input-01 -> 10	**Timeout**	>300.00	-

Bảng 6: Bảng kết quả thuật toán Brute-Force

Nhận xét: Ta có thể thấy Brute-Force chỉ giải được input nhỏ nhất, trong khi hầu hết các input còn lại đều vượt quá ngưỡng timeout 300 giây. Thời gian chạy tăng đột biến ngay từ các instance nhỏ, phản ánh sự bùng nổ tổ hợp khi duyệt toàn bộ không gian nghiệm. Điều này khẳng định Brute-Force không có khả năng mở



Hình 4: Thời gian chạy của thuật toán Brute-Force theo từng input

rộng và chỉ mang ý nghĩa đối chứng, không phù hợp để giải bài toán Hashiwokakero trong thực tế.

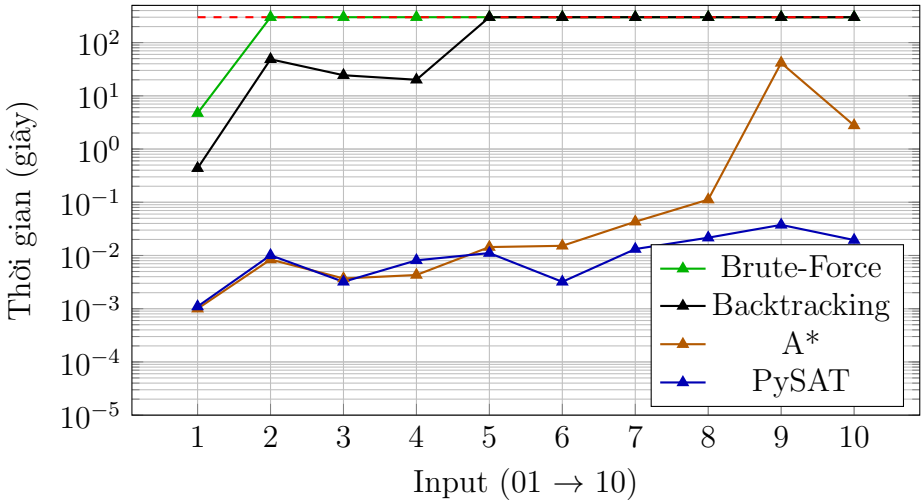
5.2.5 So sánh tổng hợp

Bảng 7 miêu tả số lượng test case giải thành công theo từng thuật toán ở từng mốc thời gian khác nhau.

Thuật toán	**< 1 phút**	**< 2 phút**	**< 5 phút**
PySAT	10	10	10
A*	10	10	10
Backtracking	4	4	4
Brute-Force	1	1	1

Bảng 7: Bảng thống kê số lượng Test case giải thành công theo mốc từng thời gian

Nhận xét: Bảng 7 và biểu đồ 5 cho thấy PySAT và A* đạt hiệu suất vượt trội, giải thành công toàn bộ 10 test case trong thời gian dưới 1 phút, trong khi Backtracking chỉ giải được 4 case và Brute-Force chỉ 1 case, phản ánh rõ sự hạn chế của các phương pháp tìm kiếm không thông tin hoặc không có pruning tối ưu.

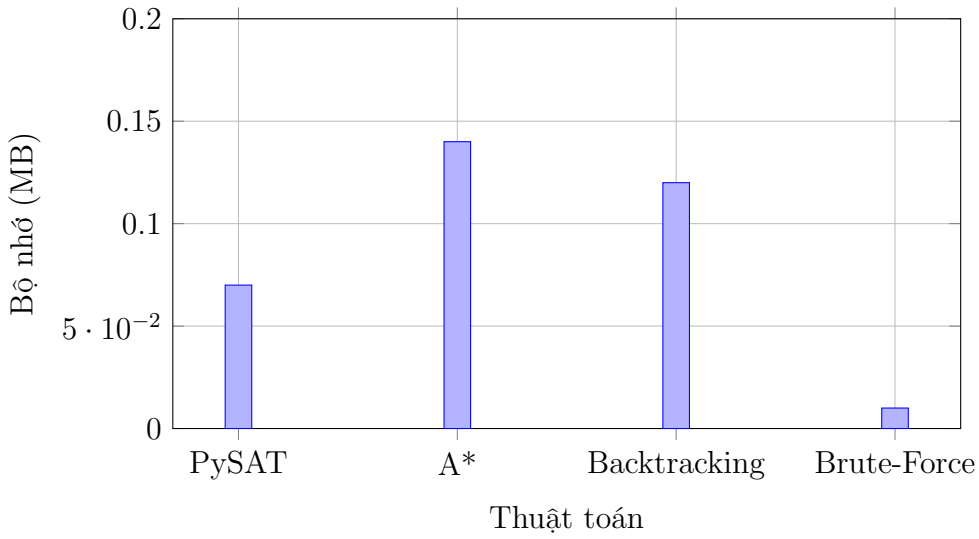


Hình 5: So sánh thời gian thực thi của từng thuật toán

Đặc biệt, biểu đồ minh họa PySAT thường ổn định và nhanh nhất trên đa số input (thời gian 10^{-2} đến 10^{-1} giây), nhưng A* vượt trội hơn ở test case thứ 4 (nhanh hơn PySAT), có thể do heuristic connectivity và pruning direct phù hợp hơn với cấu trúc puzzle cụ thể này.

Tóm lại, kết quả khẳng định ưu thế của PySAT (SAT-based) và A* (heuristic search) so với backtracking/brute-force về tốc độ và khả năng mở rộng, giúp nhóm so sánh rõ ràng giữa hai cách tiếp cận khác nhau.

5.2.6 So sánh mức độ tiêu thụ bộ nhớ



Hình 6: Bộ nhớ sử dụng trong Test input-01

Do sự khác biệt lớn về khả năng mở rộng, chỉ test case nhỏ nhất được tất cả các thuật toán giải thành công. Vì vậy, thay vì so sánh định lượng trên các test lớn, nhóm thực hiện phân tích định tính mức sử dụng bộ nhớ dựa trên đặc điểm hoạt động của từng thuật toán.

Thuật toán Brute-Force có mức tiêu thụ bộ nhớ tức thời thấp do chỉ duyệt tuần tự các cấu hình mà không lưu trữ trạng thái, nhưng không khả thi với bài toán lớn vì không gian tìm kiếm tăng theo hàm mũ. Backtracking sử dụng thêm bộ nhớ cho ngăn xếp đệ quy và trạng thái trung gian, tuy nhiên mức tăng được kiểm soát nhờ cơ chế cắt tỉa.

Ngược lại, A* tiêu tốn nhiều bộ nhớ nhất do phải lưu trữ nhiều trạng thái trong hàng đợi ưu tiên và sao chép trạng thái trong quá trình tìm kiếm. Trong khi đó, PySAT cho thấy khả năng sử dụng bộ nhớ ổn định hơn nhờ biểu diễn bài toán dưới dạng CNF và cơ chế suy diễn của SAT solver, không cần liệt kê toàn bộ không gian trạng thái.

6 Kết luận

Từ những thử nghiệm đối với các test case, ta có thể thấy sử dụng thư viện `PySAT` phương pháp tối ưu nhất trong những phương pháp được nêu ra ở 4 để giải quyết bài toán Hashiwokakero. Sự tối ưu của `PySAT` không chỉ được thể hiện ở thời gian giải từng bộ kiểm thử mà còn là ở mức tiêu thụ bộ nhớ hợp lý. Điều này khá dễ hiểu khi đây là một thư viện chính thức của ngôn ngữ lập trình `Python`, chuyên dùng cho việc giải các bài toán CNF.

Ngay sau `PySAT`, thuật toán A* là thuật toán tối ưu tiếp theo khi thành công giải được tất cả các bộ test với thời gian trung bình không quá lớn. Dù mức tiêu thụ bộ nhớ của A* có phần nhỉnh hơn các phương pháp còn lại nhưng nhìn chung, nó vẫn không phải là một vấn đề quá lớn.

Ngược lại với hai phương pháp vừa được nêu, brute-force và backtracking hoạt động không quá hiệu quả đối với những đầu vào quá phức tạp. Dù tiêu tốn lượng bộ nhớ ít nhất nhưng brute-force đã không thể cho ra đáp án ngay từ `input-02` trong khoảng thời gian giới hạn. Điều này cũng xảy ra tương tự đối với backtracking tại `input-05.txt` khi trong cả ba mốc thời gian: 1 phút, 2 phút và 5 phút, thuật toán đều không thể tìm được kết quả.

Cuối cùng, ta có thể kết luận, đối với bài toán Hashiwokakero, phương pháp tối ưu nhất là `PySAT`, tiếp theo là A*, backtracking và thuật toán kém tối ưu nhất là brute-force.

Tài liệu

[1] GeeksforGeeks. *Brute Force Approach and Its Pros and Cons*. 2025. URL: https://www.geeksforgeeks.org/dsa/brute-force-approach-and-its-pros-and-cons/ (**urlseen** 21/12/2025).

[2] GeeksforGeeks. *Introduction to Backtracking*. Truy cập: 2025-12-21. 2025. URL: https://www.geeksforgeeks.org/dsa/introduction-to-backtracking-2/.

[3] Peter E Hart, Nils J Nilsson **and** Bertram Raphael. "A formal basis for the heuristic determination of minimum cost paths". **in***IEEE Transactions on Systems Science and Cybernetics*: 4.2 (1968), **pages** 100–107.

[4] *Hashiwokakero*. https://en.wikipedia.org/wiki/Hashiwokakero. Accessed: 2025-12-17. 2025.

[5] Timo Junttila. *Solving SAT*. Accessed: 2025-12-21. Aalto University. 2022. URL: https://users.aalto.fi/~tjunttil/2022-DP-AUT/notes-sat/solving.html.

[6] PySAT Development Team. *PySAT — Python Toolkit for Prototyping with SAT*. Accessed: 2025-12-21. PySAT. 2025. URL: https://pysathq.github.io/.

[7] Laurent Simon **andothers**. *Glucose SAT Solver*. Accessed: 2025-12-21. LaBRI, University of Bordeaux. 2025. URL: https://www.labri.fr/perso/lsimon/research/glucose/.