BỘ GIÁO DỤC VÀ ĐÀO TẠO

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

CƠ SỞ TRÍ TUỆ NHÂN TẠO

BÁO CÁO MÔN HỌC

PROJECT 02 – Hashiwokakero Logic

Giảng viên hướng dẫn : Lê Hoài Bắc
Lê Nhựt Nam

Nhóm thực hiện : Nhóm 9

Lớp : 23TNT1

Nhóm sinh viên thực hiện :

Bàng Mỹ Linh | 23122009
Lại Nguyễn Hồng Thanh | 23122018
Phan Huỳnh Châu Thịnh | 23122019
Nguyễn Trọng Hòa | 23122029

Tháng 12 năm 2025

Mục lục

1 NỘI DUNG BÁO CÁO

1.1 Mô hình hóa bài toán về CNF

1.1.1 Biến logic

Để chuyển đổi bài toán Hashiwokakero thành một bài toán SAT (CNF) trước hết là xác định các biến mệnh đề. Thay vì gán biến cho từng ô trong ma trận, nhóm quyết định tập trung vào các cạnh tiềm năng giữa các cặp đảo. Vì có thể giúp phản ánh đúng bản chất của bài toán, giảm đáng kể số biến và số mệnh đề, đồng thời giúp đơn giản hóa các ràng buộc, dễ kiểm tra tính liên thông khi giải.

Xác định tập cạnh tiềm năng: Từ ma trận đầu vào ta có danh sách các đảo **islands** có dạng *(r, c, value)*. Ta tiến hành tìm các cạnh tiềm năng (E) trong hàm `find_aligned_pairs`: Với mỗi cặp đảo, chúng phải cùng nằm trên cùng một hàng hoặc cùng một cột và giữa chúng phải không có đảo nào khác. Nếu thỏa thì sẽ có một cạnh tiềm năng với chiều là H (cùng hàng) và V (cùng cột).

Định nghĩa biến: Với mỗi cạnh tiềm năng $e = (u, v) \in E$, số lượng cầu có thể là $0, 1, 2$. Vì vậy, để biểu diễn 3 trạng thái này bằng logic nhị phân (True/ False), nhóm sử dụng 2 biến logic cho mỗi cạnh $(x_{uv,1}, x_{uv,2})$, với việc định danh chúng bằng một cặp số ID `(var_counter, var_counter + 1)` trong hàm `create_logic_vars`. Khi mô hình giải, biến $x_{uv,1}$ sẽ là *True* nếu có ít nhất 1 cầu nối giữa hai đảo u, v và biến $x_{uv,2}$ sẽ là *True* nếu có đúng 2 cầu nối.
Khi đó, việc kết nối giữa hai đảo u, v được định nghĩa theo quy tắc sau:

- **Không có cầu**: $x_{uv,1} = False, x_{uv,2} = False$
- **Cầu đơn**: $x_{uv,1} = True, x_{uv,2} = False$
- **Cầu đôi**: $x_{uv,1} = True, x_{uv,2} = True$

và tổng số lượng biến logic cần sử dụng cho bài toán là $2 \times |E|$.

1.1.2 Các ràng buộc

Ràng buộc 1: Double Bridge Constraint

Mô tả ràng buộc: Với mỗi đảo có cầu đôi xây trên một cạnh hợp lệ của nó thì chắc chắn cũng xây được cầu đơn trên ứng với cạnh này.

Triển khai: Hàm `generate_cnf_double_bridges(var_map)`: Với `var_map` là dictionary ánh xạ mỗi cạnh hợp lệ sang cặp literals (e_1, e_2), với $e_1 = 1 \Leftrightarrow$ có thể xây cầu đơn trên cạnh; $e_2 = 1 \Leftrightarrow$ có thể xây cầu đôi trên cạnh.

Khi đó ta có ràng buộc $(e_2 \Rightarrow e_1) \Leftrightarrow (\neg e_2 \vee e_1)$ (CNF).

Ràng buộc 2: Số lượng cầu trên đảo

Mô tả ràng buộc: Với cạnh hợp lệ ứng với 1 đảo trên đồ thị, số lượng cầu xây trên cạnh này phải đúng bằng số ghi trên đảo.

Ý tưởng giải quyết: Ta chia thành hai điều kiện con:

1. Không tồn tại đảo nào có số lượng cầu vượt quá số ghi trên đảo
2. Không tồn tại đảo nào có số lượng cầu nhỏ hơn số ghi trên đảo

Triển khai: Ta triển khai hai hàm phụ trợ:

- `at_least_k(lits, k)`: Với `k` là giá trị số cầu tối thiểu, `lits` là danh sách các literals ứng với đảo đang xét - đại diện cho các cầu có thể xây trên các cạnh hợp lệ nối với đảo này.
  Cơ chế như sau: Nếu kết hợp với ràng buộc cầu đôi, ta luôn có:

  $$\sum_{x \in lits} x = \sum_{e} (x_{1e} + x_{2e}) \quad \text{với e là một cạnh hợp lệ đến đảo}$$

  Tức là tổng giá trị của các literals đúng bằng số cầu có thể xây đến đảo. Số cầu không ít hơn k tương đương có ít nhất k biến bằng 1 trong lits. Suy ra có nhiều nhất n - k biến bằng 0. Điều này tương đương với ràng buộc: phép nối rời của mỗi tổ hợp n - k + 1 biến bất kì của lits phải true. Vậy ta có thêm C_n^{n-k+1} CNF.

- `at_most_k(lits, k)`: Hoàn toàn tương tự hàm at least, với `k` là giá trị số cầu tối đa. Số cầu không vượt quá k tương đương có nhiều nhất k biến bằng 1 trong lits. Suy ra mỗi tổ hợp k + 1 biến bất kì của lits luôn tồn tại ít nhất 1 biến là false. Do đó phép nối rời của **phủ định** k + 1 biến bất kỳ luôn true. Vậy ta có thêm C_n^{k+1} CNF.

Ràng buộc 3: Không cắt nhau

Mô tả ràng buộc: Không thể cùng xây cầu trên hai cạnh hợp lệ cắt nhau.

Triển khai: Hàm `generate_cnf_no_cross` thực hiện kiểm tra hai cạnh ngang - dọc có cắt nhau không dựa vào tọa độ hình học của các đảo. Với mỗi cặp cạnh ngang - dọc được xác định là cắt nhau, ta thực hiện ánh xạ các cạnh này với các literals tương ứng của chúng. Cụ thể, giả sử với cạnh ngang e_A và cạnh dọc e_B, 2 cặp literals tương ứng lần lượt là (a_1, a_2) và (b_1, b_2). Khi đó ta có các CNF cần thỏa mãn là:

$$[\neg a_1 \vee \neg b_1] \quad \text{(không xây 2 cầu đơn cho 2 cạnh)}$$
$$[\neg a_1 \vee \neg b_2] \quad \text{(không xây cầu đơn - cầu đôi)}$$
$$[\neg a_2 \vee \neg b_1] \quad \text{(không xây cầu đơn - cầu đôi)}$$
$$[\neg a_1 \vee \neg b_1] \quad \text{(không xây 2 cầu đôi cho 2 cạnh)}$$

1.2 Các giải thuật

1.2.1 Solver1: PySAT

Mục tiêu chính: Giải quyết bài toán bằng cách sử dụng bộ giải SAT hiện đại. Các ràng buộc cục bộ được giải quyết bởi Glucose3, trong khi ràng buộc toàn cục về tính liên thông được kiểm tra và xử lý theo cơ chế lặp.

Chiến lược Ràng buộc trễ
Việc mã hóa điều kiện "đồ thị liên thông"trực tiếp sang dạng chuẩn CNF đòi hỏi số lượng mệnh đề rất lớn, gây bùng nổ bộ nhớ và giảm hiệu năng. Do đó, Solver1: PySAT áp dụng kỹ thuật Lazy Constraints, cụ thể:

1. Khởi tạo bộ giải chỉ với các ràng buộc cục bộ (3 ràng buộc trên)
2. Tìm kiếm nghiệm khả dĩ thỏa mãn các ràng buộc này.
3. Kiểm tra tính liên thông ở bước hậu xử lý. Nếu vi phạm, hệ thống sẽ thêm ràng buộc mới vào bộ giải để loại bỏ nghiệm này và tiếp tục tìm kiếm.

Quy trình hoạt động chi tiết
Thuật toán trong hàm `solve_with_pysat` thực hiện quy trình lặp như sau:

1. Khởi tạo và nạp dữ liệu: Toàn bộ các ràng buộc về số lượng cầu nối và giao cắt cạnh được chuyển đổi thành các mệnh đề CNF và nạp vào `Glucose3`.
2. Vòng lặp tìm kiếm Tại mỗi bước lặp, thuật toán thực hiện:
   - Truy vấn SAT: Gọi hàm `solver.solve()` để tìm một phép gán biến (model) thỏa mãn tập mệnh đề hiện tại. Nhờ cơ chế Conflict-Driven Clause Learning, bộ giải có thể nhanh chóng định hướng không gian tìm kiếm đến các vùng khả thi.
   - Giải mã và Kiểm tra: Phép gán trả về được chuyển đổi thành cấu trúc đồ thị. Hàm `is_connected` kiểm tra xem đồ thị này có tạo thành một thành phần liên thông duy nhất hay không.
   - Xử lý kết quả:
     - Nếu đồ thị liên thông: Đây là nghiệm hợp lệ. Thuật toán dừng và trả về kết quả.
     - Nếu đồ thị không liên thông: Đây là nghiệm cục bộ sai. Thuật toán chuyển sang bước tạo mệnh đề chặn.

Cơ chế Mệnh đề chặn
Khi phát hiện một nghiệm cục bộ sai (thỏa mãn số cạnh nhưng bị ngắt quãng), thuật toán cần đảm bảo bộ giải không lặp lại nghiệm này trong tương lai. Ta xây dựng một mệnh đề chặn (blocking clause) bằng cách phủ định lại toàn bộ phép gán hiện tại.

Giả sử phép gán hiện tại là tập các literal $L = \{l_1, l_2, \ldots, l_k\}$, mệnh đề chặn được thêm vào là:

$$C_{block} = \neg l_1 \vee \neg l_2 \vee \cdots \vee \neg l_k$$

Mệnh đề này buộc bộ giải phải thay đổi giá trị của ít nhất một biến trong lần giải tiếp theo, từ đó lái quá trình tìm kiếm sang một cấu hình khác.

Điều kiện dừng
Thuật toán kết thúc khi gặp một trong ba trường hợp:

1. Tìm được nghiệm thỏa mãn tính liên thông (SAT).
2. Không còn phép gán nào thỏa mãn tập ràng buộc (UNSAT), chứng tỏ bài toán vô nghiệm.
3. Thời gian thực thi vượt quá ngưỡng cho phép (Timeout 30s).

1.2.2 Solver2: A* Search

Mục tiêu chính: Dùng A* giải toàn bộ các mệnh đề CNF cho bài toán Hashiwokakero để tìm một phép gán hợp lệ.

Định nghĩa không gian trạng thái

Một trạng thái trong A* được biểu diễn bởi:

- Một phép gán `assignment` cho một số biến (một biến được gán khi nó là biến đầu tiên mang giá trị none hoặc nó là literal cuối cùng trong 1 mệnh đề → nó phải mang giá trị sao cho mệnh đề này được thỏa mãn)
- Heurisic h: đo mức độ chưa thỏa mãn của trạng thái, được tính bằng số literals chưa thỏa mãn.
- Chi phí g: thể hiện số lần mở rộng trạng thái kể từ trạng thái gốc.

Giải thích hàm chi phí và Heuristic

Hàm chi phí :

Trong A* chuẩn ta có công thức:

$$f(s) = g(s) + h(s).$$

Chi phí thực tế $g(s)$: Trong bối cảnh SAT, chi phí được định nghĩa là **số lần mở rộng trạng thái**, tức mỗi lần thuật toán lấy một trạng thái ra khỏi hàng đợi ưu tiên (open list) và thực hiện sinh trạng thái con:

$$g(s_{\text{child}}) = g(s_{\text{parent}}) + 1.$$

Heuristic:

Heuristic được sử dụng là *số mệnh đề chưa được thỏa mãn* dưới phép gán hiện tại:

$$h(s) = |\{C \in \text{CNF} \mid C \text{ chưa thỏa trong } s\}|.$$

Trạng thái càng tốt khi càng ít mệnh đề chưa thỏa. Heuristic này thỏa mãn tính chất không âm và xem như ước lượng mức "xa" so với trạng thái mục tiêu.

Unit propagation

Sau mỗi lần gán một biến ở bước mở rộng, ta áp dụng **lan truyền đơn vị (unit propagation)**:

- Nếu một mệnh đề chỉ còn một literal chưa gán, literal đó bắt buộc phải được gán để tránh xung đột.
- Nếu xuất hiện mệnh đề rỗng sau khi lan truyền, nhánh hiện tại bị loại bỏ.

Kỹ thuật này giúp thu hẹp mạnh không gian tìm kiếm và phát hiện các nhánh vi phạm sớm.

Cơ chế mở rộng trạng thái

Hàm `expand_state` thực hiện:

- Chọn biến chưa gán đầu tiên theo thứ tự (các biến được sắp xếp theo tần suất xuất hiện để ưu tiên biến có nhiều ràng buộc nhất).
- Tạo hai trạng thái con tương ứng với hai quyết định gán `True` và `False`.
- Áp dụng `unit propagation` cho mỗi trạng thái con - unit propagation là hệ quả bắt buộc khi thực hiện gán, nói cách khác việc gán và unit prop được tính là một hành động. Do đó cost không tăng khi thực hiện unit prop.

Điều kiện dừng

Một trong hai điều kiện sau:

1. Khi h = 0 và tất cả mệnh đề đều thỏa và mọi biến đã được gán.
2. Thời gian chạy vượt quá một ngưỡng nhất định (mặc định 30s) ⇒ Xem như không tìm được nghiệm

1.2.3 Solver3: Backtracking

Mục tiêu chính: Giải bài toán thỏa mãn ràng buộc bằng thuật toán tìm kiếm theo chiều sâu (DFS), kết hợp với các chiến lược suy diễn và lựa chọn biến thông minh để giảm thiểu không gian tìm kiếm.

Cơ chế hoạt động

Nếu A* lưu trữ nhiều trạng thái trong hàng đợi, thì Backtracking chỉ duy trì một đường đi cục bộ tại mỗi thời điểm. Thuật toán thử gán giá trị cho một biến, sau đó lan truyền ràng buộc. Nếu gặp mâu thuẫn với các ràng buộc thì thuật toán sẽ quay lui (backtrack) để thử gán giá trị khác.

Quy trình đệ quy:

1. **Lan truyền**: Suy diễn các biến bắt buộc từ phép gán hiện tại.

2. **Kiểm tra**: Nếu mâu thuẫn thì trả về thất bại và quay lui. Nếu đã gán hết biến thì trả về kết quả.
3. **Lựa chọn**: Chọn biến tiếp theo để gán (dùng Heuristic).
4. **Mở rộng**: Thử lần lượt các giá trị (True/False) và đệ quy.

Các chiến lược heuristic

Để tối ưu hóa tốc độ và tránh bùng nổ tổ hợp, thuật toán áp dụng hai chiến lược heuristic quan trọng dựa trên cấu trúc của CNF:

1. Lựa chọn biến: Most Constrained Variable (MCV)

Hàm `select_most_constrained_variable` chọn biến chưa được gán mà xuất hiện nhiều nhất trong các mệnh đề *chưa thỏa mãn*.

$$Var_{next} = \arg\max_{v \in V_{unassigned}} (\text{tần suất } v \text{ trong } \{C \in \text{CNF} \mid C \text{ chưa thỏa}\})$$

Ý nghĩa: Fail-first principle - Chọn biến khó thỏa mãn nhất để xử lý trước. Nếu nhánh này dẫn đến bế tắc, ta muốn phát hiện điều đó sớm nhất có thể để lược bỏ.

2. Lựa chọn giá trị: Least Constraining Value (LCV)

Hàm `choose_value_order` quyết định nên thử gán `True` hay `False` trước. Nó đếm xem mỗi giá trị sẽ giúp thỏa mãn bao nhiêu mệnh đề ngay lập tức. *Ý nghĩa: Succeed-first principle* - Ưu tiên giá trị có khả năng dẫn đến lời giải cao hơn, giúp tìm ra nghiệm (nếu có) nhanh hơn mà không cần duyệt hết không gian.

Lan truyền đơn vị (Unit Propagation)

Tương tự như A*, Backtracking sử dụng hàm `unit_propagate` tại mỗi bước đệ quy.

- **Cơ chế:** Nếu một mệnh đề $C = l_1 \vee l_2 \cdots \vee l_k$ có tất cả literal đều sai ngoại trừ l_i chưa gán, thì l_i bắt buộc phải là `True`.
- **Tác dụng:** Biến đổi bài toán từ *dự đoán* sang *suy diễn logic*. Thay vì phải đoán mò giá trị của biến, thuật toán tự động điền các giá trị bắt buộc, giúp giảm độ sâu của cây tìm kiếm đi đáng kể.

Cắt tỉa nhánh (Pruning) và Điều kiện dừng

- **Cắt tỉa:** Ngay khi `unit_propagate` trả về `None` (phát hiện mâu thuẫn), hàm đệ quy `dfs` lập tức dừng nhánh đó và quay lui. Đây là yếu tố then chốt giúp Backtracking chạy nhanh hơn Brute-force hàng nghìn lần.
- **Timeout:** Trong quá trình đệ quy, cứ sau mỗi 1000 node được mở rộng, thuật toán kiểm tra thời gian thực:

  $$t_{current} - t_{start} > 30s \Rightarrow \text{Dừng và trả về trạng thái tốt nhất (hoặc None).}$$

  Điều này đảm bảo chương trình không bị treo vô hạn với các bản đồ lớn (20×20).

1.2.4 Solver4: Brute-Force

Mục tiêu chính: Tìm kiếm nghiệm bằng cách thử tất cả các cấu hình có thể của các cầu nối giữa các đảo.

Nguyên lý hoạt động

Thuật toán Brute-force giải bài toán Hashiwokakero bằng cách:

1. Liệt kê tất cả các cạnh tiềm năng (potential edges) giữa các cặp đảo thẳng hàng
2. Với mỗi cạnh, thử tất cả 3 khả năng: 0 cầu, 1 cầu, hoặc 2 cầu
3. Kiểm tra từng cấu hình xem có thỏa mãn tất cả các ràng buộc hay không
4. Dừng ngay khi tìm được cấu hình hợp lệ đầu tiên

Không gian tìm kiếm

Với E là số cạnh tiềm năng, không gian tìm kiếm có kích thước:

$$\text{Search Space} = 3^E$$

Đây là lý do chính khiến Brute-force chỉ khả thi với các bài toán có số cạnh nhỏ. Ví dụ:

- Với $E = 10$ cạnh: $3^{10} = 59,049$ cấu hình
- Với $E = 20$ cạnh: $3^{20} \approx 3.5$ tỷ cấu hình
- Với $E = 30$ cạnh: $3^{30} \approx 2 \times 10^{14}$ cấu hình

Các bước kiểm tra cho mỗi cấu hình

Thuật toán sử dụng ba hàm kiểm tra để loại bỏ sớm các cấu hình không hợp lệ:

1. Kiểm tra bậc đỉnh (Degree Check) Hàm `_check_degrees(config)` kiểm tra xem tổng số cầu nối vào mỗi đảo có đúng bằng số ghi trên đảo hay không:

$$\sum_{\text{cạnh nối với đảo } i} \text{số cầu trên cạnh} = \text{giá trị đảo } i$$

Đây là bước kiểm tra nhanh nhất và loại bỏ được nhiều cấu hình không hợp lệ ngay từ đầu.

2. Kiểm tra không cắt nhau (No-Cross Check) Hàm `_check_no_cross(sol)` kiểm tra từng cặp cạnh trong cấu hình:

- Lấy tất cả các cạnh có cầu (số cầu > 0)
- Với mỗi cặp cạnh, gọi hàm `cross_check` để kiểm tra giao nhau
- Nếu tìm thấy bất kỳ cặp cạnh nào cắt nhau $\Rightarrow$ loại bỏ cấu hình

3. Kiểm tra tính liên thông (Connectivity Check) Sử dụng hàm `is_connected(islands, sol)` để đảm bảo tất cả các đảo được kết nối thành một đồ thị liên thông duy nhất.

Cơ chế tối ưu hóa

Mặc dù là thuật toán đơn giản, code đã áp dụng một số kỹ thuật tối ưu:

- **Early termination**: Kiểm tra degree trước (nhanh nhất), sau đó mới kiểm tra no-cross và connectivity. Điều này giúp loại bỏ nhiều cấu hình không hợp lệ sớm mà không cần thực hiện các kiểm tra phức tạp hơn.
- **Timeout mechanism**: Dừng sau một khoảng thời gian nhất định (mặc định 60 giây) để tránh chạy vô hạn trên các bài toán lớn.
- **Periodic checking**: Chỉ kiểm tra timeout sau mỗi 1000 cấu hình để giảm overhead từ việc gọi `time.time()` quá thường xuyên.

Độ phức tạp

- **Thời gian**: $O(3^E \times (V + E^2))$
  - 3^E: số cấu hình cần thử
  - V: chi phí kiểm tra degree cho V đảo
  - E^2: chi phí kiểm tra no-cross cho mọi cặp cạnh
- **Không gian**: $O(V + E)$ - chỉ lưu một cấu hình tại một thời điểm

Ưu và nhược điểm

Ưu điểm:

- Đơn giản, dễ hiểu và dễ cài đặt
- Đảm bảo tìm được nghiệm nếu nghiệm tồn tại (complete)
- Sử dụng ít bộ nhớ - chỉ lưu trữ một cấu hình tại một thời điểm
- Không cần cấu trúc dữ liệu phức tạp

Nhược điểm:

- Độ phức tạp hàm mũ $O(3^E)$ - không khả thi cho bài toán lớn
- Không có chiến lược thông minh để tìm kiếm - thử mọi khả năng một cách mù quáng
- Thời gian chạy tăng cực kỳ nhanh khi số cạnh tăng
- Không tận dụng được các ràng buộc để cắt tỉa không gian tìm kiếm hiệu quả

Kết luận về Brute-force

Thuật toán Brute-force phù hợp để:

- Kiểm chứng tính đúng đắn của các thuật toán phức tạp hơn trên các bài toán nhỏ
- Làm baseline để so sánh hiệu năng
- Giải các puzzle rất nhỏ (grid $\leq 5 \times 5$, ít cạnh)

Tuy nhiên, Brute-force hoàn toàn không khả thi cho các bài toán thực tế có kích thước từ 7×7 trở lên do bùng nổ tổ hợp.

1.3 Thực nghiệm và kết quả

1.3.1 Mô tả thực nghiệm

Cấu trúc file input

Mỗi file input được tổ chức dưới dạng ma trận các số nguyên, trong đó:

- **Số 0**: Đại diện cho ô trống (không có đảo)
- **Số từ 1-8**: Đại diện cho đảo với giá trị tương ứng là số cầu cần nối với đảo đó
- Các giá trị được phân cách bởi dấu phẩy (,)
- Mỗi hàng của ma trận được ghi trên một dòng riêng

Ví dụ cấu trúc file input-01.txt:

```
0,2,0,0,3,0,2
2,0,0,3,0,1,0
0,0,0,0,1,0,3
0,2,0,6,0,2,0
2,0,1,0,0,0,0
0,0,0,3,0,0,3
2,0,3,0,1,0,0
```

Ma trận này biểu diễn một lưới 7×7 với 18 đảo. Ví dụ, đảo có giá trị 6 ở vị trí (3,3) cần có tổng cộng 6 cầu nối vào nó.

Bộ dữ liệu thử nghiệm

Nhóm đã thiết kế bộ test case bao gồm 34 file input với các kích thước và độ phức tạp khác nhau:

Nhóm 1: Grid 7×7 (Files 01-05)

- Số lượng: 5 files
- Kích thước: 7×7
- Số đảo: 18-20 đảo
- Mục đích: Kiểm tra hiệu năng trên các bài toán nhỏ

Nhóm 2: Grid 9×9 **(Files 06-10)**

- Số lượng: 5 files
- Kích thước: 9×9
- Số đảo: 29-32 đảo
- Mục đích: Tăng độ phức tạp, kiểm tra khả năng mở rộng

Nhóm 3: Grid 11×11 **(Files 11-15)**

- Số lượng: 5 files
- Kích thước: 11×11
- Số đảo: 37-41 đảo
- Mục đích: Đánh giá hiệu năng trên bài toán trung bình

Nhóm 4: Grid 13×13 **(Files 16-20)**

- Số lượng: 5 files
- Kích thước: 13×13
- Số đảo: 56-62 đảo
- Mục đích: Đánh giá hiệu năng trên bài toán kích thước trung bình - lớn

Nhóm 5: Grid 17×17 **(Files 21-25)**

- Số lượng: 5 files
- Kích thước: 17×17
- Số đảo: 90-99 đảo
- Mục đích: Thử thách thuật toán với không gian tìm kiếm lớn và mật độ đảo dày đặc.

Nhóm 6: Grid 20×20 **(Files 26-30)**

- Số lượng file: 5 files
- Kích thước lưới: 20×20
- Số đảo: 127-130 đảo
- Mục đích: Kiểm tra giới hạn, đánh giá khả năng mở rộng và hiệu quả quản lý bộ nhớ.

Nhóm 7: Grid nhỏ đặc biệt (Files 31-34)

- Số lượng: 4 files
- Kích thước: 3×3, 5×5, 7×3
- Số đảo: 3-9 đảo
- Mục đích: Kiểm tra Brute-force trên các puzzle rất nhỏ

Chi tiết từng file input

Dưới đây là mô tả chi tiết cho 34 files input:

Bảng 1.1: Thông tin chi tiết các file input (Phần 1: File 01 - 28)

File	Grid	Số đảo	Đặc điểm
01	7×7	18	Puzzle nhỏ, mật độ đảo vừa phải
02	7×7	19	Tương tự file 01, tăng 1 đảo
03	7×7	20	Mật độ đảo cao nhất trong nhóm 7×7
04	7×7	19	Bố trí đảo khác so với file 02
05	7×7	19	Có nhiều đảo giá trị lớn (5)
06	9×9	32	Mật độ đảo cao, nhiều ràng buộc
07	9×9	29	Có đảo giá trị 6 tạo ràng buộc mạnh
08	9×9	30	Bố trí đảo đối xứng
09	9×9	30	Giống hệt file 08 (duplicate)
10	9×9	29	Giống hệt file 07 (duplicate)
11	11×11	37	Puzzle trung bình, có đảo giá trị 6
12	11×11	39	Có đảo giá trị cao (8), nhiều ràng buộc
13	11×11	41	Mật độ đảo cao nhất trong nhóm 11×11
14	11×11	40	Có nhiều đảo giá trị 5-6
15	11×11	40	Có đảo giá trị 7, bài toán khó
16	13×13	60	Puzzle lớn, có đảo giá trị 7
17	13×13	62	Có đảo giá trị 8, ràng buộc rất chặt
18	13×13	56	Puzzle lớn, có đảo giá trị 7
19	13×13	60	Mật độ đảo cao hơn file 18, có đảo giá trị 8
20	13×13	56	Số lượng đảo trung bình, đảo max 7
21	17×17	91	Kích thước rất lớn, đảo max 6
22	17×17	93	Mật độ cao, có đảo giá trị 8
23	17×17	98	Số lượng đảo lớn, khó với đảo giá trị 8
24	17×17	99	Mật độ cao nhất nhóm 17×17
25	17×17	95	Phức tạp, nhiều đảo giá trị 8
26	20×20	130	Kích cỡ cực đại, số lượng đảo cao nhất
27	20×20	123	Kích thước rất lớn, nhiều đảo giá trị 7
28	20×20	130	Kích cỡ cực đại, nhiều đảo giá trị 8, mật độ rất cao

Bảng 1.2: Thông tin chi tiết các file input (Phần 2: File 29 - 34)

File	Grid	Số đảo	Đặc điểm
29	20×20	129	Kích cỡ cực đại, độ khó cao với nhiều đảo giá trị 7
30	20×20	127	Kich cỡ rất lớn, nhiều đảo giá trị 5-6-7
31	3×3	3	Kích cỡ siêu nhỏ, đơn giản
32	3×3	4	Kích cỡ siêu nhỏ, 4 đảo ở 4 góc
33	7×3	8	Kích thước chữ nhật hẹp, đảo 2 bên, cách 1 ô
34	5×5	9	Kích cỡ nhỏ, hình vuông, đảo trải đều cách 1 ô

Phương pháp đánh giá

1. **Khả năng giải (Success/Failed)**:
   - Success: Tìm được nghiệm hợp lệ
   - Failed: Không tìm được nghiệm thỏa các ràng buộc và tính liên thông trong thời gian cho phép (Timeout)
2. **Thời gian chạy (Time) (s)**:
   - Timeout: 30 giây cho A*, backtracking, 60 giây cho Brute-force
   - Độ chính xác: 4-5 chữ số thập phân

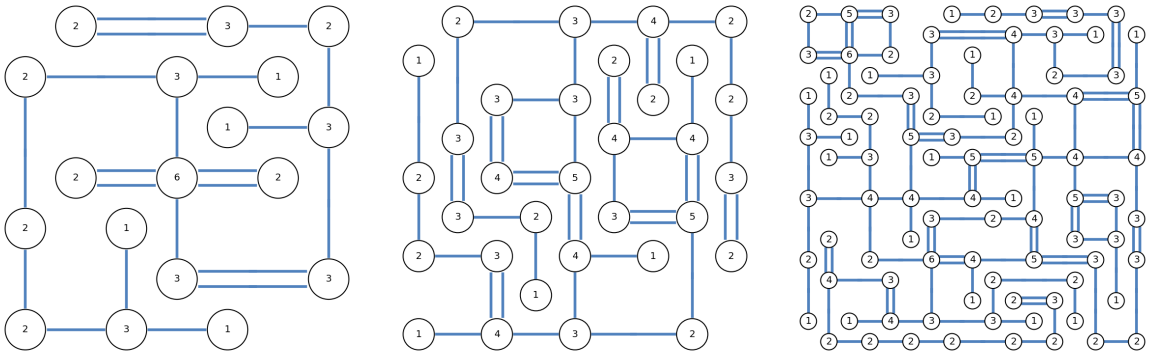
Môi trường thực nghiệm

Python 3.7+, Thư viện: numpy, python-sat, matplotlib

1.3.2 Kết quả thực nghiệm

Một số ảnh outputs

Sau khi thu được lời giải lưu vào folder `Source/Outputs`, tiến hành vẽ thành các ảnh kết quả để có thể trực quan kết quả và dễ dàng kiểm tra lại.



(a) pysat_01_7x7 (b) pysat_06_9x9 (c) pysat_21_17x17

Hình 1.1: Một số ảnh lời giải lưu tại `Source/visualize`

Bảng 1.3: Bảng kết quả thực nghiệm so sánh các thuật toán (Phần 1).

File	**Grid**	**Đảo**	**Result**	**Thuật toán**			
				pySAT	**A***	**Backtrk**	**Brute**
01	7x7	18	Status	Success	Success	Success	Failed
			Time	0.0000s	0.1019s	0.1446s	60.0016s
02	7x7	19	Status	Success	Success	Success	Failed
			Time	0.0010s	0.0908s	0.7099s	60.0022s
03	7x7	20	Status	Success	Success	Success	Failed
			Time	0.0010s	0.0360s	0.1746s	60.0026s
04	7x7	19	Status	Success	Success	Success	Failed
			Time	0.0010s	0.0335s	0.2122s	60.0007s
05	7x7	19	Status	Success	Success	Success	Failed
			Time	0.0010s	0.0368s	0.2136s	60.0029s
06	9x9	32	Status	Success	Success	Success	Failed
			Time	0.0020s	0.2079s	0.2372s	60.0020s
07	9x9	29	Status	Success	Success	Success	Failed
			Time	0.0020s	0.1953s	3.8307s	60.0040s
08	9x9	30	Status	Success	Success	Success	Failed
			Time	0.0010s	0.5299s	3.0387s	60.0019s
09	9x9	30	Status	Success	Success	Success	Failed
			Time	0.0020s	0.5418s	3.0774s	60.0014s
10	9x9	29	Status	Success	Success	Success	Failed
			Time	0.0010s	0.1893s	3.8384s	60.0013s
11	11x11	37	Status	Success	Success	Success	Failed
			Time	0.0010s	1.3283s	3.5045s	60.0026s
12	11x11	39	Status	Success	Success	Success	Failed
			Time	0.0010s	0.3213s	1.6981s	60.0042s
13	11x11	41	Status	Success	Success	Success	Failed
			Time	0.0010s	19.1077s	30.5536s	60.0014s
14	11x11	40	Status	Success	Success	Success	Failed
			Time	0.0020s	5.9564s	6.1118s	60.0012s
15	11x11	40	Status	Success	Success	Success	Failed
			Time	0.0020s	0.8197s	6.3316s	60.0048s
16	13x13	60	Status	Success	Failed	Failed	Failed
			Time	0.0030s	30.0000s	31.0500s	60.0050s
17	13x13	62	Status	Success	Failed	Failed	Failed
			Time	0.0020s	30.0000s	33.7297s	60.0067s

Bảng 1.4: Bảng kết quả thực nghiệm so sánh các thuật toán (Phần 2).

File	**Grid**	**Đảo**	**Result**	**Thuật toán**			
				pySAT	**A***	**Backtrk**	**Brute**
18	13x13	56	Status	Success	Failed	Failed	Failed
			Time	0.0070s	30.0005s	31.7839s	60.0054s
19	13x13	60	Status	Success	Failed	Failed	Failed
			Time	0.0030s	30.0017s	30.3955s	60.0067s
20	13x13	56	Status	Success	Failed	Failed	Failed
			Time	0.0160s	30.0171s	33.9727s	60.0042s
21	17x17	91	Status	Success	Failed	Failed	Failed
			Time	0.0065s	30.0050s	35.0890s	60.0073s
22	17x17	93	Status	Success	Failed	Failed	Failed
			Time	0.0120s	30.0005s	30.4555s	60.0117s
23	17x17	98	Status	Success	Failed	Failed	Failed
			Time	0.0070s	30.0031s	34.8554s	60.0059s
24	17x17	99	Status	Success	Failed	Failed	Failed
			Time	0.3027s	30.0050s	37.6206s	60.0122s
25	17x17	95	Status	Success	Failed	Failed	Failed
			Time	0.0050s	30.0032s	30.0456s	60.0096s
26	20x20	130	Status	Success	Failed	Failed	Failed
			Time	0.5988s	30.0068s	44.9513s	60.0170s
27	20x20	128	Status	Success	Failed	Failed	Failed
			Time	0.0080s	30.0014s	39.9397s	60.0124s
28	20x20	130	Status	Success	Failed	Failed	Failed
			Time	0.0060s	30.0075s	34.2286s	60.0118s
29	20x20	128	Status	Success	Failed	Failed	Failed
			Time	0.0060s	30.0069s	42.3420s	60.0124s
30	20x20	127	Status	Success	Failed	Failed	Failed
			Time	0.0152s	30.0070s	58.3706s	60.0191s
31	3x3	3	Status	Success	Success	Success	Success
			Time	0.0000s	0.00017s	0.0000s	0.0010s
32	3x3	4	Status	Success	Success	Success	Success
			Time	0.0000s	0.00074s	0.0010s	0.0000s
33	7x3	8	Status	Success	Success	Success	Success
			Time	0.0000s	0.00441s	0.0210s	0.0201s
34	5x5	9	Status	Success	Success	Success	Success
			Time	0.0010s	0.00542s	0.0039s	0.1491s

1.3.3 Đánh giá và phân tích

PySAT

Hiệu quả tổng quát: Dựa trên bộ dữ liệu 34 bài toán, thuật toán PySAT đạt tỉ lệ thành công 100%, hoàn thành với thời gian dưới 0.01s với những bài toán nhỏ và 0.6s với thậm chí những bài toán phức tạp nhất trong bộ dữ liệu. **PySAT gặp khó khăn nhất với bài thứ 24 và 26 (lần lượt tốn 0.3s và 0.6s), lớn hơn đáng kể so với những bài toán cùng kích cỡ.**
Vì sao? Điều này có thể được giải thích bởi khái niệm "Tường Cầu". Trong hai bài toán này, đều tồn tại những cụm dày đặc các đảo có bậc cao, dẫn đến việc chúng thường xuyên được nối lại với nhau (nhằm thỏa mãn ràng buộc bậc), tạo nên những bức tường cầu. Cấu trúc này làm tăng đáng kể số lượng các cầu nối tiềm năng và đặc biệt tăng số lượng các ràng buộc không cắt nhau do nhiều cầu nối dài có thể giao nhau.
Vì sao PySAT lại hiệu quả như vậy?

1. Tính đầy đủ về mặt lý thuyết đảm bảo tìm được kết luận cho bài toán SAT Solving
2. Hầu hết các bài toán Hashiwokakero có thể mã hóa một cách hiệu quả sang SAT.

Grid	Số input	Tỉ lệ thành công	Min time (s)	Avg time (s)	Max time (s)
3x3	2	100.0%	0.0000	0.0000	0.0000
5x5	1	100.0%	0.0010	0.0010	0.0010
7x3	1	100.0%	0.0000	0.0000	0.0000
7x7	5	100.0%	0.0000	0.0008	0.0010
9x9	5	100.0%	0.0010	0.0016	0.0020
11x11	5	100.0%	0.0010	0.0014	0.0020
13x13	5	100.0%	0.0020	0.0062	0.0160
17x17	5	100.0%	0.0050	0.0667	0.3027
20x20	5	100.0%	0.0060	0.1268	0.5988

Bảng 1.5: Thống kê kết quả chạy Pysat theo kích thước lưới

Kết luận về PySAT
Ưu điểm:

- Solver Glucose3 có hiệu suất cực cao, có thể giải mỗi bài toán trong mẫu thử trong vòng chưa tới 1s.
- Xử lý tốt các ràng buộc cục bộ.
- Tính đầy đủ về mặt lý thuyết đảm bảo tồn tại kết luận cuối cùng: miễn đủ thời gian và tài nguyên, thuật toán sẽ luôn luôn cho ra kết quả

Nhược điểm:

- Gặp khó khăn với các ràng buộc liên thông.

A* Search

Hiệu quả tổng quát Dựa trên bộ dữ liệu 34 bài toán, thuật toán A* đạt tỉ lệ thành công 24/34 bài (khoảng 70%), giải được toàn bộ các bài có kích thước lưới từ 7×7 đến 11×11, nhưng thất bại (Timeout 30s) ở toàn bộ các bài từ 13×13 trở lên. Điều này phản ánh rõ đặc điểm của A*: hiệu quả rất cao trên không gian tìm kiếm nhỏ, nhưng không mở rộng tốt khi số biến logic tăng mạnh.

Vì sao A* hoạt động tốt ở các grid-size nhỏ? Có ba nguyên nhân chính:

- **(1) Độ lớn không gian tìm kiếm nhỏ**: Các grid 7×7 và 9×9 có số đảo từ 18–32, dẫn đến số cạnh hợp lệ và số biến CNF ít. Khi số biến ít, mỗi bước gán của A* làm thu hẹp đáng kể không gian.
- **(2) Unit propagation có tác dụng mạnh**: Khi một biến được gán, các ràng buộc CNF từ ba nhóm ràng buộc (tổng cầu, cầu đôi, không cắt nhau) thường xuyên tạo ra các mệnh đề đơn, giúp A* tự động gán thêm nhiều biến mà không cần mở rộng thêm.
- **(3) Heuristic "số mệnh đề chưa thỏa" dẫn hướng tốt với grid-size nhỏ**: Ở bài nhỏ, số lượng clause chưa thỏa thay đổi nhanh theo từng phép gán, giúp A* ưu tiên các trạng thái tốt.

Nhờ ba yếu tố này, thời gian chạy rất nhỏ:

$$\text{Nhanh nhất: } 0.00017s \text{ (input-31)}, \qquad \text{trung bình nhóm nhỏ: } < 0.1s.$$

Vì sao A* thất bại ở grid-size lớn? Khi kích thước lưới vượt 13×13 (tức số đảo 56–130), A* không giải được trong 30s. Điều này không phải lỗi mã nguồn mà xuất phát từ giới hạn bản chất của thuật toán:

- **(1) Heuristic yếu khi số clause nhiều**: Khi số clause lớn, rất nhiều trạng thái có cùng giá trị heuristic $h(s)$. Do đó A* không được "dẫn hướng" rõ ràng, gần như trở thành best-first search mù.
- **(2) Số biến tăng tuyến tính, không gian tìm kiếm tăng theo hàm mũ**: Với 60–130 đảo, số biến logic là vài trăm đến hàng ngàn, dẫn đến số trạng thái cần mở rộng trước khi gặp nghiệm cực kỳ lớn. Bộ nhớ open-list phình nhanh, khiến A* không thể tiến sâu.
- **(3) Unit propagation suy yếu ở bài lớn**: Trong bài nhỏ, mệnh đề đơn xuất hiện thường xuyên. Nhưng trong bài lớn, nhiều literal vẫn chưa bị ràng buộc, propagation không giảm được không gian đáng kể.
- **(4) Bản chất của A* không phù hợp cho SAT lớn**: A* phải lưu toàn bộ open-list, khiến chi phí bộ nhớ và so sánh tăng mạnh.

Kết quả cho thấy tất cả bài từ input-16 đến input-30 đều timeout ở 30 giây.

So sánh các bài cùng grid-size nhưng kết quả khác nhau Ví dụ tiêu biểu là nhóm 11×11:

File	Số đảo	Thời gian A*
input-11	37	1.328s
input-12	39	0.321s
input-13	41	19.107s
input-14	40	5.956s
input-15	40	0.819s

Mặc dù cùng kích thước lưới, thời gian chênh lệch tới ≈ 60 lần. Nguyên nhân:

- **Cấu trúc ràng buộc khác nhau**: Hai bài có cùng số đảo nhưng vị trí đảo và hướng cạnh hợp lệ khác nhau nên độ chặt ràng buộc khác.
- **Số cạnh thẳng hàng (aligned pairs)** khác nhau: Bài có nhiều cạnh hợp lệ nên số biến và clause lớn dẫn đến heuristic yếu hơn.
- **Mức độ dễ của unit propagate khác nhau**: Một bài mà propagation hoạt động mạnh sẽ giảm không gian đáng kể.

Do đó hiệu năng A* phụ thuộc nhiều vào *hình dạng puzzle*, không chỉ kích thước.
Phân tích theo từng loại grid

Grid	Số input	Tỉ lệ thành công	Min time	Avg time	Max time
7×7	6	100%	0.033s	0.060s	0.102s
9×9	5	100%	0.189s	0.330s	0.542s
11×11	5	100%	0.321s	5.10s	19.108s
13×13 trở lên	18	0%	30s	30s	30s

Bảng 1.6: Bảng số liệu thực nghiệm của thuật toán A* theo từng kích thước lưới.

Grid 7×7 A* hoạt động rất hiệu quả. Không gian tìm kiếm nhỏ và các ràng buộc CNF dễ dàng tạo ra nhiều mệnh đề đơn, khiến unit propagation loại bỏ nhanh nhiều nhánh sai. Thời gian chạy cực thấp (trung bình 0.06s). Tỉ lệ thành công 100%.

Grid 9×9 Hiệu năng vẫn rất tốt. Thời gian tăng so với 7×7 nhưng vẫn dưới 1 giây cho mọi input. Propagation vẫn đóng vai trò mạnh và heuristic phân biệt trạng thái tốt. Không có bài nào thất bại.

Grid 11×11 Mặc dù tỉ lệ thành công vẫn 100%, thời gian giải biến thiên rất lớn (0.32s đến 19.10s). Sự khác biệt này đến từ cấu trúc từng bài: số đảo, số cạnh hợp lệ và độ chặt ràng buộc khiến heuristic không còn dẫn hướng tốt như ở grid nhỏ. Tuy nhiên propagation vẫn đủ mạnh để tìm ra nghiệm trước khi timeout.

Grid 13×13 trở lên Tất cả 18 bài đều timeout ở 30 giây. Khi số đảo vượt 55, không gian tìm kiếm tăng theo hàm mũ, trong khi heuristic "số clause chưa thỏa" không đủ mạnh để dẫn hướng. A* phải mở rộng nhiều trạng thái tương đương và không thể tiến đến nghiệm trong thời gian giới hạn.

Đánh giá nghiệm và kiểm tra ràng buộc Trong `main.py`, mỗi nghiệm A* được kiểm tra qua hai bước quan trọng:

1. **Thỏa từng mệnh đề CNF** Các ràng buộc: số lượng cầu, cầu đôi, không cắt nhau đều được mã hóa đúng dạng CNF. A* chỉ được xem là tìm thấy nghiệm khi tất cả clause thỏa.

2. **Kiểm tra tính liên thông của đồ thị** Sau khi thu được model từ A*, đồ thị cầu được xây lại và kiểm tra bởi hàm `is_connected` trong `utils.py` (theo đoạn mã `main.py` tại lệnh gọi `is_connected(...)`). Chỉ nghiệm nào thỏa CNF *và* liên thông mới được xem là hợp lệ.

Điều này đảm bảo nghiệm A* là nghiệm thực sự đúng của bài Hashiwokakero.
Kết luận A* là thuật toán **rất mạnh cho bài toán nhỏ và trung bình**, nhờ:

- khả năng pruning tốt,
- unit propagation giảm mạnh không gian,
- heuristic đơn giản nhưng hiệu quả ở kích thước nhỏ.

A* **không tốt khi mở rộng cho bài toán lớn**. Mặc dù việc cải tiến heuristic có thể giải được các grid-size lớn hơn 13. Tuy nhiên, bản chất thuật toán A* phải lưu toàn bộ open-list khi không gian trạng thái tăng theo hàm mũ. Do vậy ở puzzle từ 20×20 trở lên, A* gần như thất bại vì bùng nổ trạng thái.

Backtracking

Hiệu quả tổng quát Dựa trên bộ dữ liệu 34 bài toán, thuật toán Backtracking (được tối ưu hóa với Unit Propagation và Heuristic MCV/LCV) đạt tỉ lệ thành công 19/34 bài (khoảng 56%). Thuật toán giải quyết tốt các bài toán có kích thước nhỏ và trung bình (từ 7×7 đến 11×11), nhưng bắt đầu gặp khó khăn và Timeout ở các bài toán lớn từ 13×13 trở lên.

Tuy nhiên, so với A*, Backtracking có thời gian giải chậm hơn ở các map trung bình (ví dụ input-13 mất 30s so với A* mất 19s), cho thấy Backtracking tốn nhiều thời gian hơn trong việc thử-sai trên các nhánh sâu.

Vì sao Backtracking hoạt động tốt ở các grid-size nhỏ? Ở các bài toán kích thước nhỏ (7×7, 9×9, và các bài test nhỏ bổ sung input-31 đến 34), Backtracking chạy rất nhanh (thường dưới 1s, thậm chí xấp xỉ 0s như input-31). Có ba nguyên nhân chính:

- **(1) Sức mạnh của Lan truyền đơn vị (Unit Propagation)**: Đây là yếu tố then chốt. Ngay khi thử gán một biến, Unit Propagation lập tức suy diễn ra hàng loạt biến khác bắt buộc phải gán theo. Với không gian nhỏ, chuỗi suy diễn này thường đi đến tận cùng (hoặc tìm ra nghiệm, hoặc phát hiện mâu thuẫn ngay lập tức) mà không cần đệ quy sâu.
- **(2) Heuristic MCV (Most Constrained Variable) hiệu quả**: Chiến lược chọn biến tham gia vào nhiều mệnh đề chưa thỏa mãn nhất giúp thuật toán ưu tiên xử lý các *rào cản* của bài toán. Ở map nhỏ, việc giải quyết đúng các rào cản này thường dẫn ngay đến lời giải.
- **(3) Không tốn chi phí quản lý bộ nhớ**: Khác với A* phải duy trì hàng đợi ưu tiên (Open List) tốn kém, Backtracking chỉ tốn bộ nhớ cho ngăn xếp đệ quy (Stack), giúp các thao tác cơ bản diễn ra cực nhanh.

Vì sao Backtracking thất bại ở grid-size lớn? Khi kích thước lưới tăng lên 13×13 trở đi (Input-16 đến Input-30), Backtracking bắt đầu bị Timeout (vượt quá 30s). Nguyên nhân xuất phát từ bản chất của thuật toán:

- **(1) Bùng nổ tổ hợp ở độ sâu lớn**: Với số lượng biến logic lên tới hàng trăm, cây tìm kiếm trở nên cực kỳ sâu. Dù có cắt tỉa, số lượng nhánh phải duyệt vẫn quá lớn. Nếu một lựa chọn sai lầm xảy ra ở đầu cây, thuật toán có thể mất hàng triệu bước duyệt trong nhánh sai đó trước khi quay lui lại được.
- **(2) Chi phí lặp lại của Unit Propagation**: Tại *mỗi* bước đệ quy, thuật toán phải chạy lại quy trình Unit Propagation trên toàn bộ tập mệnh đề. Với các bài toán lớn có hàng nghìn mệnh đề, chi phí tích lũy này trở nên rất lớn, làm chậm tốc độ duyệt node.
- **(3) Thiếu cái nhìn toàn cục**: Backtracking chỉ nhìn thấy mâu thuẫn cục bộ. Nó không có cơ chế đánh giá tổng thể như hàm $f(n)$ của A* để biết mình đang đi xa hay gần đích, dẫn đến việc lãng phí thời gian vào các nhánh không tiềm năng.

So sánh các bài cùng grid-size nhưng kết quả khác nhau Hiện tượng chênh lệch thời gian cũng xuất hiện rõ rệt ở Backtracking, ví dụ nhóm 11×11:

File	Trạng thái	Thời gian Backtracking
input-12	Success	1.698s
input-11	Success	3.504s
input-14	Success	6.111s
input-15	Success	6.331s
input-13	Success	30.55s (Sát nút Timeout)

Sự chênh lệch lên tới gần 20 lần giữa input-12 và input-13 cho thấy Backtracking rất nhạy cảm với **thứ tự biến**. Ở input-13, heuristic MCV đã vô tình chọn một thứ tự gán không tối ưu ngay từ đầu, khiến thuật toán phải quay lui (backtrack) liên tục ở độ sâu lớn, trong khi ở input-12, các lựa chọn ban đầu dẫn thẳng đến kết quả.

Phân tích theo từng loại grid

Grid	Số input	Tỉ lệ thành công	Min time	Avg time	Max time
7×7	6	100%	0.14s	0.24s	3.83s
9×9	5	100%	3.03s	3.46s	3.84s
11×11	5	100%	1.70s	9.64s	30.55s
13×13 trở lên	15	0%	30s	30s	30s

Bảng 1.7: Bảng số liệu thực nghiệm của thuật toán Backtracking.

Grid 7×7 và Test nhỏ Hoạt động ổn định và chính xác. Tuy nhiên, thời gian trung bình (0.24s) chậm hơn so với A* (0.06s). Điều này cho thấy chi phí gán-rút (assign-undo) của đệ quy tốn kém hơn so với việc mở rộng trạng thái có định hướng của A* ở quy mô nhỏ.

Grid 9×9 Thời gian tăng lên mức 3-4 giây. Đây là mức tăng đáng kể so với 7x7, báo hiệu rằng độ phức tạp đang tăng nhanh. Tuy nhiên, tỉ lệ giải thành công vẫn là 100%.

Grid 11×11 Đây là giới hạn khả năng của Backtracking trong đồ án này. Thời gian dao động mạnh từ 1.7s đến hơn 30s. Input-13 may mắn giải được ở giây thứ 30.5, cho thấy thuật toán bắt đầu chạm ngưỡng quá tải.

Grid 13×13 trở lên Toàn bộ các file từ input-16 đến input-30 đều Timeout. Vì không thể tìm ra nghiệm thỏa mãn hoàn toàn tính liên thông và các ràng buộc logic trong thời gian cho phép.

Kết luận Backtracking là một thuật toán **mạnh mẽ và tiết kiệm bộ nhớ**, phù hợp cho các bài toán Hashiwokakero cỡ vừa và nhỏ.

- **Ưu điểm:** Cài đặt đơn giản hơn A*, không tốn bộ nhớ lưu trạng thái, giải quyết tốt các ràng buộc cục bộ.
- **Nhược điểm:** Dễ bị rơi vào các nhánh sai ở độ sâu lớn, thời gian chạy không ổn định.

So với A*, Backtracking chậm hơn ở các map nhỏ nhưng có khả năng *đào sâu* tốt hơn ở một số trường hợp đặc thù. Tuy nhiên, cả hai đều không thể hơn được PySAT về cả tốc độ và khả năng mở rộng.

Brute-Force

Tổng quan kết quả Brute-force chỉ giải được 4 bài có kích thước rất nhỏ (files 31-34) và thất bại hoàn toàn với tất cả 30 bài còn lại, bao gồm cả những bài nhỏ nhất trong nhóm 7×7.

Bảng 1.8: Tổng hợp kết quả Brute-force theo kích thước grid

Grid	Số input	Thành công	Tỉ lệ	Thời gian TB
3×3	2	2	100%	0.0005s
5×5	1	1	100%	0.1491s
7×3	1	1	100%	0.0201s
7×7	5	0	0%	60.00s (timeout)
9×9	5	0	0%	60.00s (timeout)
11×11	5	0	0%	60.00s (timeout)
13×13	2	0	0%	60.01s (timeout)
Tổng	**21**	**4**	**19%**	–

Vì sao Brute-force thất bại trên hầu hết các bài? Nguyên nhân chính là **bùng nổ tổ hợp** (combinatorial explosion). Khi kích thước grid tăng, số cạnh tiềm năng tăng nhanh, dẫn đến không gian tìm kiếm 3^E tăng theo hàm mũ.

Ví dụ minh họa:

- **File 31** (3×3, 3 đảo): Giả sử có khoảng 3-4 cạnh $\Rightarrow 3^4 = 81$ cấu hình $\Rightarrow$ Giải trong 0.001s
- **File 34** (5×5, 9 đảo): Giả sử có khoảng 12 cạnh $\Rightarrow 3^{12} \approx 531,441$ cấu hình $\Rightarrow$ Giải trong 0.15s

- **File 01** (7×7, 18 đảo): Với 18 đảo, số cạnh tiềm năng có thể lên tới 25-35 cạnh. Giả sử 30 cạnh $\Rightarrow 3^{30} \approx 2 \times 10^{14}$ cấu hình $\Rightarrow$ Timeout sau 60s

Như vậy, chỉ cần tăng từ 12 lên 30 cạnh, không gian tìm kiếm đã tăng gấp **376 triệu lần**!

Phân tích chi tiết theo từng nhóm

Nhóm thành công (3×3, 5×5, 7×3) Bốn bài mà Brute-force giải được có đặc điểm chung:

- **Số đảo rất ít**: 3-9 đảo
- **Số cạnh tiềm năng nhỏ**: Ước tính 3-12 cạnh
- **Không gian tìm kiếm nhỏ**: 3^3 đến 3^{12} cấu hình
- **Thời gian chạy**: Từ gần như tức thời (0.0s) đến 0.15s

File	Grid	Số đảo	Thời gian
32	3×3	4	0.0000s
31	3×3	3	0.0010s
33	7×3	8	0.0201s
34	5×5	9	0.1491s

Quan sát thấy thời gian tăng theo quy luật hàm mũ khi số đảo tăng: từ 0.0s (4 đảo) lên 0.15s (9 đảo).

Nhóm thất bại hoàn toàn (7×7 và lớn hơn) Tất cả 30 bài còn lại đều timeout sau đúng 60 giây. Điều này cho thấy:

1. **Không gian tìm kiếm quá lớn**: Ngay cả với grid 7×7 nhỏ nhất, Brute-force không thể duyệt hết không gian trong 60 giây.
2. **Các tối ưu hóa không đủ mạnh**: Mặc dù có degree check và no-cross check, nhưng các kiểm tra này không loại bỏ đủ nhanh để giảm không gian xuống mức khả thi.
3. **Thời gian timeout nhất quán**: Tất cả đều timeout ở khoảng 60.001-60.019s, chứng tỏ thuật toán chạy hết thời gian mà không tìm được nghiệm.

So sánh với A* trên cùng test cases Để thấy rõ sự khác biệt, ta so sánh trực tiếp: Từ bảng trên, rõ ràng:

- Trên các bài cực nhỏ (3×3): Cả hai đều rất nhanh, chênh lệch không đáng kể
- Trên bài 5×5: A* đã nhanh hơn 27 lần
- Trên bài 7×7: A* nhanh hơn hàng trăm đến hàng nghìn lần, trong khi Brute-force timeout hoàn toàn

Bảng 1.9: So sánh A* và Brute-force trên các bài nhỏ

File	Grid	A* Time	Brute Time	Tỉ lệ
31	3×3	0.00017s	0.0010s	A* nhanh hơn 6×
32	3×3	0.00074s	0.0000s	Tương đương
33	7×3	0.00441s	0.0201s	A* nhanh hơn 5×
34	5×5	0.00542s	0.1491s	A* nhanh hơn 27×
01	7×7	0.1019s	60.0016s	A* nhanh hơn 588×
03	7×7	0.0360s	60.0026s	A* nhanh hơn 1,667×

Vì sao cùng grid-size nhưng đều timeout? Vì sao tất cả 5 files 7×7 đều timeout ở 60s, mặc dù có số đảo khác nhau (18-20)?

Giải thích:

- **Số cạnh không chênh lệch nhiều**: Mặc dù số đảo khác nhau 1-2 đảo, nhưng số cạnh tiềm năng chỉ chênh nhau vài cạnh (ví dụ 28-32 cạnh).
- **Đều nằm trong vùng không khả thi**: Cả 3^{28} và 3^{32} đều quá lớn để Brute-force có thể duyệt hết trong 60 giây.
- **Cơ chế timeout**: Khi đạt 60s, thuật toán dừng ngay lập tức, nên thời gian ghi nhận đều rơi vào khoảng 60.00-60.02s.

Số lượng cấu hình đã kiểm tra Brute-force in ra số cấu hình đã kiểm tra mỗi 1000 lần. Với timeout 60s, ước tính:

- Với các bài 7×7: Brute-force chỉ kiểm tra được khoảng vài triệu đến vài chục triệu cấu hình trong 60s, trong khi tổng không gian có thể là hàng tỷ tỷ cấu hình.
- Tỉ lệ duyệt: $< 0.001\%$ không gian tìm kiếm

Đánh giá nghiệm của Brute-force Với 4 bài thành công (files 31-34), nghiệm được kiểm tra qua:

1. **Degree check**: Đảm bảo mỗi đảo có đúng số cầu
2. **No-cross check**: Không có cầu nào cắt nhau
3. **Connectivity check**: Tất cả đảo liên thông

Điều này được thực hiện trong hàm `solve_brute_force` của `main.py`, tương tự như các solver khác. Do đó, nghiệm Brute-force tìm được là **hoàn toàn hợp lệ**.

Kết luận về Brute-force
Ưu điểm:

- Đơn giản, dễ cài đặt
- Đảm bảo tìm được nghiệm nếu có đủ thời gian

Nhược điểm:

- **Hoàn toàn không khả thi** cho bài toán có kích thước từ 7×7 trở lên
- Độ phức tạp hàm mũ $O(3^E)$ khiến thời gian chạy tăng cực nhanh
- Không có chiến lược thông minh - chỉ thử mù quáng
- Tỉ lệ thành công thấp: 11.8% (4/34)
- Kém hiệu quả gấp hàng trăm đến hàng nghìn lần so với A*

Kết luận: Brute-force chỉ phù hợp làm:

- Baseline để so sánh
- Kiểm chứng tính đúng trên các bài cực nhỏ
- Minh họa cho độ phức tạp hàm mũ

Thuật toán này **không nên được sử dụng** cho bất kỳ bài toán thực tế nào, kể cả những bài nhỏ nhất (7×7). A* vượt trội hoàn toàn về mọi mặt.

1.4 Kết luận

Qua quá trình nghiên cứu và làm bài, nhóm đã hoàn thành việc mô hình hóa bài toán dưới dạng các biến logic và mệnh đề chuẩn tắc hội (CNF), đồng thời cài đặt và kiểm chứng bốn phương pháp giải quyết khác nhau: PySAT, A* Search, Backtracking và Brute-Force. Dựa trên dữ liệu thực nghiệm từ 34 bộ test case với kích thước từ 3×3 đến 20×20, nhóm rút ra các kết luận sau:

1. **Hiệu quả của việc mô hình hóa bài toán:** Việc chuyển đổi các luật chơi của Hashiwokakero (số cầu, cầu đôi, không cắt nhau) sang dạng SAT (Boolean Satisfiability Problem) là bước đi then chốt. Cách tiếp cận này cho phép tận dụng sức mạnh của các bộ giải SAT hiện đại cũng như các kỹ thuật suy diễn logic (Unit Propagation) trong các thuật toán tìm kiếm cổ điển.
2. **Đánh giá hiệu năng các thuật toán:**
    - **PySAT (Glucose3):** Là phương pháp tối ưu nhất trong dự án. Với chiến lược "Lazy Constraints"(xử lý ràng buộc liên thông ở hậu xử lý) và cơ chế học mệnh đề mâu thuẫn (CDCL), PySAT giải quyết thành công 100% các bài toán, bao gồm cả những lưới 20×20 phức tạp nhất, với thời gian trung bình cực thấp (dưới 1 giây). Điều này chứng minh rằng việc quy dẫn về SAT là phương pháp tiếp cận hiệu quả nhất cho dạng bài toán này.
    - **A* Search:** Hoạt động rất tốt và nhanh hơn Backtracking ở các bài toán quy mô nhỏ và trung bình (7×7 đến 11×11) nhờ khả năng cắt tỉa nhánh sớm bằng Heuristic. Tuy nhiên, thuật toán gặp hạn chế lớn về bộ nhớ và khả năng dẫn hướng khi không gian trạng thái bùng nổ ở các lưới lớn (13×13 trở lên), dẫn đến việc không tìm ra nghiệm trong thời gian cho phép.

- **Backtracking:** Là giải thuật ổn định và tiết kiệm bộ nhớ. Việc kết hợp Unit Propagation và Heuristic MCV/LCV giúp thuật toán giải được các bài toán cỡ trung bình. Tuy nhiên, do thiếu cái nhìn toàn cục và chi phí lặp lại của việc suy diễn, Backtracking chậm hơn A* ở các bài nhỏ và cũng thất bại ở các bài toán lớn do độ sâu cây tìm kiếm quá lớn.
- **Brute-Force:** Chỉ mang tính chất tham khảo và kiểm chứng trên các bài toán siêu nhỏ (3×3, 5×5). Sự bùng nổ tổ hợp ($O(3^E)$) khiến phương pháp này hoàn toàn vô hiệu với các bài toán thực tế.

2 PHÂN CÔNG CÔNG VIỆC

MSSV	Họ tên	Công việc	Hoàn thành
23122009	Bàng Mỹ Linh	- Các ràng buộc - A* - Tổng hợp code	95%
23122018	Lại Nguyễn Hồng Thanh	- Thiết kế và mô tả các thực nghiệm - Brute-force - Video demo	95%
23122019	Phan Huỳnh Châu Thịnh	- Biến logic - Backtracking - Video demo	95%
23122029	Nguyễn Trọng Hòa	- pySAT - Thiết kế và mô tả các thực nghiệm - Kết luận	95%

3 PHỤ LỤC

- Video demo: Google Drive hoặc Youtube
- Source code: gitHub/hcmus-csai-Hashiwokakero
- Test cases: Google Drive Inputs

Tài liệu tham khảo

[1] R. F. Malik, R. Efendi, and E. A. Pratiwi, "Solving Hashiwokakero Puzzle Game with Hashi Solving Techniques and Depth First Search," *Bulletin of Electrical Engineering and Informatics*, vol. 1, no. 1, pp. 61–68, 2012.

[2] D. Andersson, "Hashiwokakero is NP-complete," *Information Processing Letters*, vol. 109, no. 19, pp. 1145–1146, 2009.

[3] L. C. Coelho, G. Laporte, A. Lindbeck, and T. Vidal, "Benchmark Instances and Branch-and-Cut Algorithm for the Hashiwokakero Puzzle," arXiv:1905.00973, May 2019.

[4] C. Sinz, "Towards an optimal CNF encoding of boolean cardinality constraints," in *Proc. International Conference on Principles and Practice of Constraint Programming*, pp. 827–831, 2005.

[5] A. Ignatiev, A. Morgado, and J. Marques-Silva, "PySAT: A Python toolkit for prototyping with SAT oracles," in *Proc. International Conference on Theory and Applications of Satisfiability Testing*, pp. 428–437, 2018.