Nurikabe puzzle

Contents

[1. Giới thiệu về Nurikabe puzzle 1](#_Toc191721144)

[2. Luật chơi 1](#_Toc191721145)

[3. Mục tiêu 2](#_Toc191721146)

[4. Giới thiệu về thuật toán Hill Climbing 2](#_Toc191721147)

[4.1. Ý tưởng cơ bản 2](#_Toc191721148)

[4.2. Cách hoạt động 2](#_Toc191721149)

[4.3. Các biến thể chính 2](#_Toc191721150)

[4.4. Ưu điểm 3](#_Toc191721151)

[4.5. Nhược điểm 3](#_Toc191721152)

[4.6. Ứng dụng 3](#_Toc191721153)

[4.7. Cách khắc phục nhược điểm 3](#_Toc191721154)

[4.8. So sánh với các thuật toán khác 3](#_Toc191721155)

[5. Hiện thực giải thuật Hill Climbing cho Nurikabe 4](#_Toc191721156)

[6. Đánh giá sự chiếm dụng tài nguyên máy tính khi chạy giải thuật Hill Climbing cho Nurikabe 10](#_Toc191721157)

1. Giới thiệu về Nurikabe puzzle

Nurikabe là một loại câu đố logic xuất xứ từ Nhật Bản, được phát triển bởi nhà xuất bản Nikoli – nơi cũng nổi tiếng với việc phổ biến Sudoku. Tên "Nurikabe" bắt nguồn từ một yokai (linh hồn) trong văn hóa dân gian Nhật Bản, nghĩa là "tường gạch" hoặc "tường vô hình", ám chỉ một bức tường cản đường người đi bộ vào ban đêm. Trong lĩnh vực câu đố, Nurikabe còn được biết đến với các tên gọi khác như "Cell Structure" hoặc "Islands in the Stream".

1. Luật chơi

Nurikabe được chơi trên một lưới hình chữ nhật gồm các ô vuông, trong đó một số ô có chứa số. Người chơi phải tô màu từng ô thành đen (biển) hoặc trắng (đảo), tuân theo các quy tắc sau:

1. Mỗi ô có số là một phần của đảo trắng: Con số trong ô biểu thị kích thước của đảo (số ô trắng liền kề theo chiều ngang hoặc dọc, bao gồm cả ô chứa số).
2. Các đảo trắng không được chạm nhau: Các đảo chỉ có thể tiếp xúc ở góc (theo đường chéo), không được tiếp xúc cạnh bên.
3. Tất cả ô đen phải kết nối với nhau: Các ô đen tạo thành một "biển" liên tục, không bị chia cắt.
4. Không có khối đen 2x2: Không được phép có một nhóm 4 ô đen tạo thành hình vuông 2x2 hoặc lớn hơn.
5. Mục tiêu

Mục tiêu của Nurikabe là xác định toàn bộ lưới sao cho mỗi ô được tô đen hoặc trắng, thỏa mãn tất cả các quy tắc trên, đồng thời đảm bảo lời giải duy nhất (đặc trưng của các câu đố logic thuần túy).

1. Giới thiệu về thuật toán Hill Climbing

Thuật toán Hill Climbing (Leo đồi) là một kỹ thuật tối ưu hóa thuộc nhóm thuật toán tìm kiếm cục bộ (local search algorithms), được sử dụng để giải các bài toán tối ưu hóa trong trí tuệ nhân tạo và khoa học máy tính. Nó lấy cảm hứng từ hình ảnh một người leo đồi cố gắng tìm đỉnh cao nhất bằng cách di chuyển từng bước lên cao hơn từ vị trí hiện tại. Thuật toán này đơn giản, dễ triển khai, nhưng có những hạn chế nhất định.

## Ý tưởng cơ bản

Hill Climbing hoạt động bằng cách bắt đầu từ một điểm (giải pháp ban đầu) trong không gian tìm kiếm và liên tục di chuyển đến một điểm lân cận có giá trị tốt hơn (theo hàm mục tiêu), cho đến khi không còn điểm lân cận nào tốt hơn nữa. Nó giống như việc bạn đứng trên một ngọn đồi và chỉ bước tới nơi cao hơn cho đến khi bạn đạt đỉnh.

## Cách hoạt động

Khởi tạo: Chọn một giải pháp ban đầu (có thể ngẫu nhiên hoặc dựa trên một tiêu chí nhất định).

Đánh giá: Tính giá trị của hàm mục tiêu (objective function) tại điểm hiện tại.

Tìm kiếm lân cận: Xem xét các giải pháp lân cận (neighbors) của điểm hiện tại (thường được xác định bởi một quy tắc thay đổi nhỏ).

Di chuyển: Nếu một giải pháp lân cận có giá trị hàm mục tiêu tốt hơn (cao hơn với bài toán tối ưu hóa cực đại, hoặc thấp hơn với tối ưu hóa cực tiểu), chuyển sang giải pháp đó.

Lặp lại: Tiếp tục quá trình cho đến khi không còn giải pháp lân cận nào tốt hơn (đạt cực trị cục bộ - local optimum).

Kết thúc: Trả về giải pháp hiện tại.

## Các biến thể chính

Simple Hill Climbing: Chỉ xem xét giải pháp lân cận đầu tiên tốt hơn và di chuyển ngay lập tức (first-choice).

Steepest-Ascent Hill Climbing: Xem xét tất cả các giải pháp lân cận và chọn giải pháp có giá trị tốt nhất (dốc nhất).

Stochastic Hill Climbing: Chọn ngẫu nhiên một giải pháp lân cận tốt hơn, thay vì luôn chọn cái tốt nhất, để tăng tính linh hoạt.

## Ưu điểm

Đơn giản: Dễ hiểu và dễ cài đặt.

Hiệu quả về bộ nhớ: Chỉ cần lưu trạng thái hiện tại và các trạng thái lân cận, không yêu cầu lưu trữ toàn bộ không gian tìm kiếm.

Nhanh: Phù hợp với các bài toán có không gian tìm kiếm nhỏ hoặc cần giải pháp tạm thời.

## Nhược điểm

Mắc kẹt ở cực trị cục bộ: Nếu gặp một đỉnh nhỏ (local maximum) thay vì đỉnh cao nhất toàn cục (global maximum), thuật toán sẽ dừng lại và không thể tiếp tục.

Phụ thuộc vào điểm khởi đầu: Kết quả thay đổi tùy thuộc vào giải pháp ban đầu.

Không đảm bảo tối ưu toàn cục: Với các bài toán phức tạp, nó thường không tìm được giải pháp tốt nhất.

## Ứng dụng

Tối ưu hóa hàm số: Tìm cực trị của các hàm toán học.

Trí tuệ nhân tạo: Giải các bài toán như xếp lịch, tìm đường đi, hoặc tối ưu hóa tham số trong học máy.

Trò chơi và câu đố: Tìm lời giải gần tối ưu cho các bài toán như Nurikabe hoặc Sudoku.

## Cách khắc phục nhược điểm

Khởi động lại ngẫu nhiên (Random Restart): Chạy Hill Climbing nhiều lần với các điểm bắt đầu khác nhau để tăng cơ hội tìm cực trị toàn cục.

Simulated Annealing: Thêm yếu tố ngẫu nhiên để đôi khi chấp nhận giải pháp tệ hơn, giúp thoát khỏi cực trị cục bộ.

Kết hợp với thuật toán khác: Dùng Hill Climbing như một bước trong các thuật toán phức tạp hơn (ví dụ: Genetic Algorithms).

## So sánh với các thuật toán khác

So với Gradient Descent: Hill Climbing không yêu cầu đạo hàm, phù hợp với các bài toán không liên tục hoặc không có thông tin gradient.

So với A\*: Hill Climbing đơn giản hơn nhưng kém thông minh, vì A\* dùng heuristic để tìm kiếm toàn cục.

1. Hiện thực giải thuật Hill Climbing cho Nurikabe

Các hàm cần thiết để thực hiện giải thuật:

+) def is\_valid\_cell(*self*, *row*, *col*): Hàm để xác định ô mà sắp được đi tới (theo hướng di chuyển lên / xuống, trái / phải 1 ô) thì còn nằm trong ma trận ban đầu không. Hàm trả về True nếu ô đó hợp lệ, ngược lại là False

|  |
| --- |
|  |

+) def evaluate\_state(*self*, *state*): Hàm này dùng để ước lượng độ tốt của trạng thái tiếp theo có tốt hơn trạng thái hiện tại không. Hàm này giá trị score trả về càng nhỏ là càng tốt. Bên trong hàm tiến hành kiểm tra nếu các đảo sai so với size được quy định trong ô đánh số thì score += abs(island\_size - target\_size) \* 2, nếu có vùng nước 2x2 thì score += 15, nếu vùng nước không liên thông thì score += 20. Các số điểm được cộng này thể hiện cho tầm quan trọng của điều kiện ràng buộc. Với điều kiện ràng buộc vùng nước cô lập tuyệt đối không được xuất hiện (quan trọng nhất), quan trọng thứ hai là vùng nước 2x2 không được xuất hiện, còn lại là các trường hợp đảo không đủ kích thước.

|  |
| --- |
|  |

+) def dfs(*self*, *row*, *col*, *state*, *visited*, *cell\_type*): Thuật toán dfs để duyệt qua các ô hợp lệ và cùng loại (ô loại đảo / ô loại biển) và chưa được đi đến và liền kề nhau và trả về size của vùng liền kề đó.

|  |
| --- |
|  |

+) def check\_water\_connectivity(*self*, *state*): Hàm này thực hiện kiểm tra xem vùng nước có phải là một vùng nước liên thông nhau không. Nếu liên thông trả về True ngược lại False

|  |
| --- |
|  |

+) def check\_2x2\_water(*self*, *state*): Kiểm tra trong vùng nước hiện tại có ô nào là 2x2 không. Nếu có trả về là True, ngược lại là False

|  |
| --- |
|  |

+)     def get\_error\_isolated\_water\_regions(*self*, *state*): Hàm này trả về một list các tọa độ các ô mà nằm trong vùng nước 2x2. Dùng cho việc tối ưu việc đi tìm láng giềng tốt hơn. Định nghĩa láng giềng tốt hơn là trạng thái mà tại đó score thấp hơn trạng thái hiện tại. Đạt được láng giềng tốt hơn bằng cách là tập trung đi giải quyết các lỗi sai đang gặp phải mà vùng nước 2x2 là một trường hợp con.

|  |
| --- |
|  |

+)     def get\_false\_size\_island(*self*, *state*): Hàm này trả về một danh sách các đảo sai kích thước. Các tọa độ gốc của đảo sai kích thước được thêm vào danh sách kết quả là các tọa độ của ô được đánh số chỉ định kích thước của đảo. Nếu không có đảo sai kích thước hàm trả về danh sách rỗng.

|  |
| --- |
|  |
|  |

Bên trong đoạn comment là một trường hợp bị lỗi khác của nurikabe khi chạy theo thuật toán chia đảo ra khi đảo sai kích thước nhóm em. Đó là:

+) Trong trường hợp khởi tạo giá trị ngẫu nhiên 3 lần để đạt trạng thái mới

+) Hoặc trong trường hợp đảo bị sai kích thước do 1 đảo sai kích thước thật

+) Hoặc đảo thứ nhất (nhỏ) nối với đảo thứ 2 (nhỏ) mà sai kích thước

+) Hoặc 1 đảo (lớn) nối với 1 đảo (nhỏ) mà sai kích thước

+) Hoặc 2 đảo (lớn) nối với nhau sai kích thước

🡺 Khi cắt đảo ra bằng cách đổi trạng thái L (Land – đảo) sang W (water – nước) thì có thể tạo ra 2 phần đảo một phần (Phần A) là đảo với ô có đánh số chỉ kích thước size đảo. Và phần còn lại (Phần B) có thể không chứa ô đánh số size đảo như vậy phần này có thể gây ra lỗi là đây là vùng đảo mà không chứa ô đánh số size theo yêu cầu bắt buộc.

Ở đây nhóm chỉ comment lại ý tưởng của nhóm. Về hiện thực nhóm vẫn chưa làm được việc đổi trạng thái tất cả các ô đảo trong (Phần B).

Tại sao code của nhóm không kiểm tra điều kiện ràng buộc là hai đảo dính vào nhau theo chiều ngang / dọc (trường hợp mà nhóm gọi là hai đảo liên thông)?

Trả lời:

Vì khi dùng dfs quét qua các vùng đảo với điểm bắt đầu của việc quét là các ô đánh số kích thước đảo. Nhóm đã khởi tạo một ma trận mới cho mỗi lần quét để xem các ô trong mỗi vùng đảo là đã đi qua hay chưa. Việc này giúp nhóm xác định chính xác với một ô đánh số là điểm bắt đầu thì kích thước thật của vùng đảo hiện tại là bao nhiêu. Điều này tránh được tình huống 2 đảo liên thông khi quét đảo 1 rồi thì quét đảo 2 chỉ ra kích thước là 0.

Và khi chỉnh sửa đảo (hoặc lớn sai kích thước, hoặc liên thông, hoặc liên thông sai kích thước) để hết sai bằng cách cắt ô liền kề là đảo với ô đánh số kích thước thì sẽ ngay lập tức giải quyết được tình trạng hai đảo liên thông, và vừa làm giảm kích thước của đảo. Việc hiện thực cắt đảo này được thực hiện trong hàm get\_best\_neighbor().

+)     def get\_error\_2x2\_water\_cells(*self*, *state*): Hàm này trả về danh sách các ô trong vùng nước 2x2. Nếu không có vùng 2x2 trả về rỗng

|  |
| --- |
|  |

+)     def get\_best\_neighbor(*self*): Hàm này trả về láng giềng có trạng thái tốt hơn trạng thái hiện tại. Láng giềng tốt hơn được định nghĩa là láng giềng có điểm trạng thái tại đó nhỏ hơn trạng thái hiện tại. Để tối ưu tìm được láng giềng tốt hơn nhanh nhất thì ta đi tối ưu các vùng có những ô sai quy tắc: Đổi loại ô của các ô trong vùng nước 2x2, các ô trong vùng đảo sai kích thước, các ô trong vùng nước bị cô lập. Nếu không có láng giềng tốt hơn trạng thái hiện tại tiến hành khởi tạo ngẫu nhiên 3 lần để tạo một trạng thái mới. Yêu cầu là trạng thái mới do tạo ngẫu nhiên này phải tốt hơn trạng thái hiện tại thì nhận, mọi trạng thái thấp hơn đều bỏ qua.

|  |
| --- |
|  |
|  |

+) def solve(self): Hàm này dùng để chạy giải thuật hill climbing. Thông qua các bước. Đầu tiên là in ra giá trị ban đầu của bài toán nurikabe ở dạng ma trận. Sau đó đi tìm trạng thái láng giềng tốt hơn. Nếu có trạng thái láng giềng tốt hơn thì gán trạng thái hiện tại bằng láng giềng tốt hơn và in kết quả ra màn hình. Thuật toán chỉ cho phép chạy trong số lần lặp được quy định sẵn để tránh treo máy.

|  |
| --- |
|  |

1. Đánh giá sự chiếm dụng tài nguyên máy tính khi chạy giải thuật Hill Climbing cho Nurikabe

Với đầu vào đơn giản

|  |
| --- |
|  |

Kết quả chạy chương trình:

|  |
| --- |
|  |
|  |

Với đầu vào là hỗn hợp nhiều testcase được đặt trong file test\_nurikabe\_hill.py

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |