1. Giới thiệu về thuật toán BFS

BFS (Breadth-First Search) hay còn gọi là thuật toán tìm kiếm theo chiều ngang sẽ duyệt từng các node theo hàng trong Tree từ node gốc tới node lá bằng queue theo FIFO (First In First Out).

Nguyên lý hoạt động:

* Queue bắt đầu tại Node gốc .
* Nếu Node chưa phải là kết quả cần tìm đưa các node con vào hàng đợi.
* Qua trình tiếp tục cho đến khi tìm được kết quả hoặc duyệt hết toàn bộ cây.

1. Hiện thực Nurikabe bằng thuật toán BFS
   1. Tiến trình thuật toán:

Đầu vào thuật toán sẽ là 1 ma trận vuông kích thước tùy chỉnh dữ liệu dạng chuỗi. Trong đó:

* Các khu vực ghi số là diện tích của đảo đó.
* Còn lại sẽ là cái dấu ‘?’ với ý nghĩa các ô đó chưa xác định.

Đầu ra thuật toán tương tự đầu vào là 1 ma trân vuông kích thước tùy chỉnh với các ô đã được xác định (nếu tìm thấy) hoặc ma trận ban đầu (không có kết quả). Trong đó:

* Các dấu ‘.’ được xem như là các ô nước.
* Các dấu ‘#’ được xem như là các ô đất.
* Các số là diện tích đảo.

Bắt đầu thuật toán, hàm solve\_puzzle() sẽ bắt đầu tiền xử lí dữ liệu ô đầu vào để tối ưu khả năng tìm node, sau đó hàm tạo 1 danh sách tìm tọa độ của tất cả các dấu ‘?’ khả dĩ và bắt đầu vào thuật toán BFS.

Thuật toán BFS bắt đầu với node đầu tiên là tọa độ dấu ‘?’ thứ nhất các node con là các tọa độ tiêp theo với tọa độ cũ được gán dữ liệu là nước hoặc đất.

Tại mỗi node, hàm sẽ kiểm tra 1 phần luật lệ của bài toán để nhanh chóng loại các ma trận chắc chắn sai.

Tại các node với tọa độ ‘?’ là cuối cùng trong danh sách hàm sẽ tiến hành kiểm tra bằng toàn bộ luật để đưa ra đầu ra đúng.

* 1. Chức năng các hàm:
     1. Các hàm kiểm tra luật

Để đảm bảo các đầu ra với thuật toán này là đúng với yêu cầu các hàm bao gồm như sau:

* def is\_water\_connected(self): kiểm tra các ô nước có liên thông không.
* def is\_islands\_connected(self): kiểm tra các đảo có dính nhau không
* def has\_2x2\_water\_square(self): kiểm tra có vùng nước nào 2x2 ô không.
* def is\_island\_area\_correct(self): kiểm tra kích thước đảo có đúng với yêu cầu không.
* def has\_free\_island(self): kiểm tra có đảo “vô danh” nào không (không liên thông với ô trọng số).
* def is\_island\_area\_correct2(self): kiểm tra diện tích đảo có lớn hơn yêu cầu không.
  + 1. Các hàm tổng hợp luật

Bao gồm có 2 hàm Rule() và partly\_rule() để kiểm tra 1 phần hoặc toàn bộ hàm:

* Rule() bao gồm các hàm:
  + def is\_water\_connected(self)
  + def is\_islands\_connected(self)
  + def has\_free\_island(self)
* partly\_rule() bao gồm các hàm:
  + def is\_islands\_connected(self)
  + def has\_2x2\_water\_square(self)
  + def is\_island\_area\_correct2(self)

Tất cả các ma trận ở các lá sẽ được kiểm them bởi Rule() trong khi tất cả Node sẽ được kiểm tra 1 phần bởi partly\_rule() để tỉa nhánh.

* + 1. Hàm đặt ID

Hàm def set\_ID\_begin(self): có tác dụng đánh dấu các tọa độ chứ số với ID độc nhất để đảm bảo quá trình sinh cây không bị đè dữ liệu lên tọa độ chứa số, đảm quả các đảo không dính nhau, đảm bảo diện tích đảo và hỗ trợ tiền xử lí dữ liệu.

* + 1. Hàm tiền xử lí dữ liệu

Hàm def pre\_solve(self) sẽ duyệt qua đầu vào và đặt trước 1 số điểm đảo và nước sao cho tại đó nó chắc chắn đúng.

Theo đó, có các quy luật để đặt điểm:

1. Xung quanh ô ‘1’ luôn là nước
2. Nếu 1 ô ở giữa 2 ô số, ô đó luôn là nước.
3. Nếu ô số (trừ số ‘1’) bị bao bọc bởi 3 ô nước, ô còn lại chắc chắn là ô đất.
4. Tại các góc ma trận, nếu 2 ô cạnh là nước thì ô tại góc chắc chắn là nước.
   * 1. Hàm xử lí

Như đã đề cập ở mục “Tiến trình thuật toán” hàm thực hiện theo đúng tiến trình yêu cầu và đầu ra đảm bảo chính xác.

* 1. Các thông tin trong quá trình xử lí

Vì BFS xử lí 1 lượng lớn BFS, các ma trận sẽ được in ra trong 1 log với bước nhảy 200. Thông tin trong log bao gồm:

* Ma trận qua tiền xử lí dữ liệu.
* Thông tin ma trận ở các bước Iteration
* Thông tin ma trận kết quả
* Các thông số bao gồm số Iteration, số Iteration đã bỏ qua bởi hàm luật và số Iteration thực tế.
* Nếu không có ma trận đầu ra, sau khi kết thúc viết thông tin ma trận ở quá trình, file sẽ có dòng “No solution found”.

1. Hiệu suất thuật toán

Tại bài toán được xử lí với ma trận có kích thước nxn dữ liệu. Theo đó, ta có các thông số như sau:

|  |  |  |  |
| --- | --- | --- | --- |
| Kích thước | Thời gian trung bình | Bộ nhớ chiếm dụng trung bình | Số lương ma trận khả dĩ ( |
| 2x2 | 1 ms | 10 Kb | 16 |
| 3x3 | 3.75 ms | 20 Kb | 512 |
| 4x4 | 0.11 s | 429 Kb | 65,536 |
| 5x5 | 5.8 s | 34,2 Mb | 33,554,432 |

Kích thước bài toán sẽ tăng theo cấp số mũ với 1 ô có 2 trạng thái (nước, đất), ta nhận xét mức độ phức tạp của bài toán ở trường hợp tệ nhất là

Không chỉ vậy thời gian xử lí bài toán cũng tăng lên với kích thước ma trận với 2x2 là 0.001s, 3x3 là 0.003s, 5x5 là 5.8s,… tiếp tục tăng mạnh với các kích thước cao hơn.

Bộ nhớ chiếm dụng cũng tăng theo số lượng node chứa trong queue hay số node tối đa là node. Khiến số lượng Iteration tăng mạnh dù đầu vào không ở mô hình tệ nhất (quá ít thông tin và rời rạc hoặc đầu vào lỗi) như ở 3x3 số lượng Iteration đạt tới trung bình 110-200 trong khi 5x5 con số từ khoảng 500 (trường hợp tốt) tới 200000 (trường hợp số lớn, rời rạc, và ít thông tin).

Có một số kích thước lớn hơn yêu cầu lượng bộ nhớ khổng lồ (lên tới GB) khi tăng lên 6x6,7x7,… vì thế nhìn chung các test case chi sử dụng kiểm tra với ma trận kích thước nhỏ hơn 5x5.

Nhìn chung, mặc dù lượng tài nguyên tiêu tốn theo cấp số mũ phụ thuộc vào kích thước đầu vào, các đầu vào đã được xử lí 1 phần đề cố gắng giảm thiểu số ‘?’ cần tìm giúp giảm kích thước tài nguyên. Không chỉ thể, hàm cho thấy tính chính xác khi chắc chắn tìm ra kết quả đúng nếu đầu vào là hợp lệ.