

深さ優先探索による 塗りつぶし



探索とは

- 探索とは、可能性を調べながら、解を探す方法です

深さ優先探索とは

- 最も基礎的な探索の方法です

本スライドで学ぶこと

- 深さ優先探索による「塗りつぶし」
- 深さ優先探索の再帰関数による実装
- 深さ優先探索のスタックによる実装

問題

入力

- 右図のような迷路
- . が通れる

出力

- s から g までいけるか？

```

s . . . . .
#####.
# . . . . . #.
# . #####. #.
## . . . . #. #.
#####. #. #.
g . #. #. #. #.
# . #. #. #. #.
# . #. #. #. #.
# . . . . . #. . .

```



塗りつぶし

Flood-fill

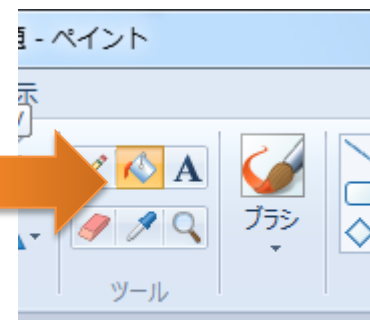
問題の解き方

1. **s** から行ける場所を全部調べる
2. **g** に行けてれば “yes”, 行けなかったら “no”

「**s** から行ける場所を全部調べる」

- これを**塗りつぶし (flood-fill)** と呼びます
- とても良く使います

このイメージ



s.
#####.
#. #.
#. . #####. #.
##. . . . #. #.
#####. #. #.
g. #. ####. #.
#. #. #. #. #.
####. #. #. #.
#. #. . .



s.
#####.
#. #.
#. . #####. #.
##. . . . #. #.
#####. #. #.
g. #. ####. #.
#. #. #. #. #.
####. #. #. #.
#. #. . .

s から行ける領域 ↑

深さ優先探索による 塗りつぶし

基本的な考え方

- 今居るところから隣に行こうとしてみる
- まだ行ってなかったら行く

ちゃんと塗りつぶすために

- 全箇所から4方向への移動を試しつくす
- そのために、試しつくしていない場所を覚えておき、戻ってくる

基本：今いる所の隣を塗る

#####. #. ##

#####. #. ##

#**S**.

#####. #####

#####. #####

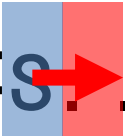
最初は s からスタート

基本：今いる所の隣を塗る

#####. #. ##

#####. #. ##

#s



#####. #####

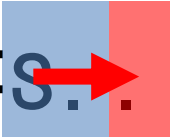
#####. #####

右隣へ移動（赤の矢印はsからの経路）

基本：今いる所の隣を塗る

#####. #. ##

#####. #. ##



#####. #####

#####. #####

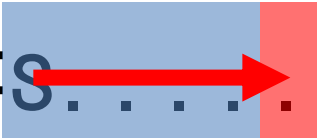
右隣へ移動（赤の矢印は s からの経路）

基本：今いる所の隣を塗る

#####. #. ##

#####. #. ##

#s.....

A diagram showing a grid with a blue highlighted area and a red arrow pointing right from 's'. The blue area covers the first four dots of the sequence 's.....'. The red arrow starts at the 's' and points to the fifth dot. The rest of the sequence '.....' is in black.

#####. #####

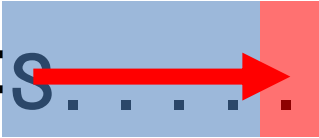
#####. #####

右隣へ移動（赤の矢印はsからの経路）

基本：今いる所の隣を塗る

#####. #. ##

#####. #. ##



#####. #####

#####. #####

分かれ道だけど，とりあえず気にせず進む

基本：今いる所の隣を塗る

#####. #. ##

#####. #. ##



#####. #####

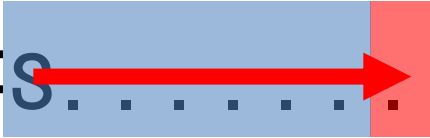
#####. #####

分かれ道だけど，とりあえず気にせず進む

基本：今いる所の隣を塗る

#####. #. ##

#####. #. ##



#####. #####

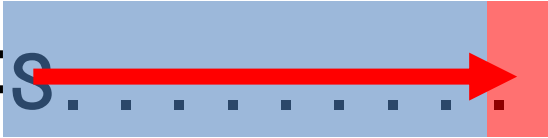
#####. #####

分かれ道だけど，とりあえず気にせず進む

基本：今いる所の隣を塗る

#####. #. ##

#####. #. ##



#####. #####

#####. #####

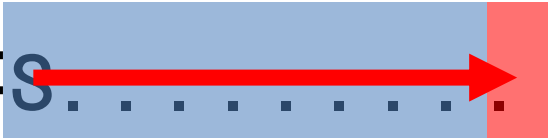
分かれ道だけど，とりあえず気にせず進む

進めるところが無くなったら戻る

#####. #. ##

#####. #. ##

#s.....



#####. #####

#####. #####

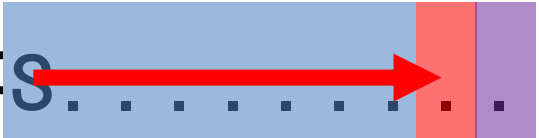
もう進めるところがない！

進めるところが無くなったら戻る

#####. #. ##

#####. #. ##

#s.....



#####. #####

#####. #####

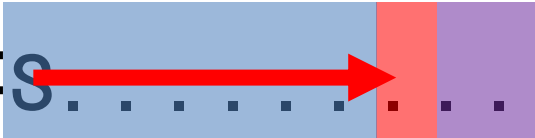
なので戻ります

進めるところが無くなったら戻る

#####. #. ##

#####. #. ##

#s.....



#####. #####

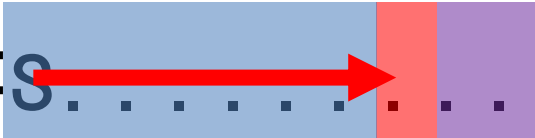
#####. #####

また戻ります

戻ってきて別のところに行けたらそっちに行く

#####. #. ##

#####. #. ##



#####. #####

#####. #####

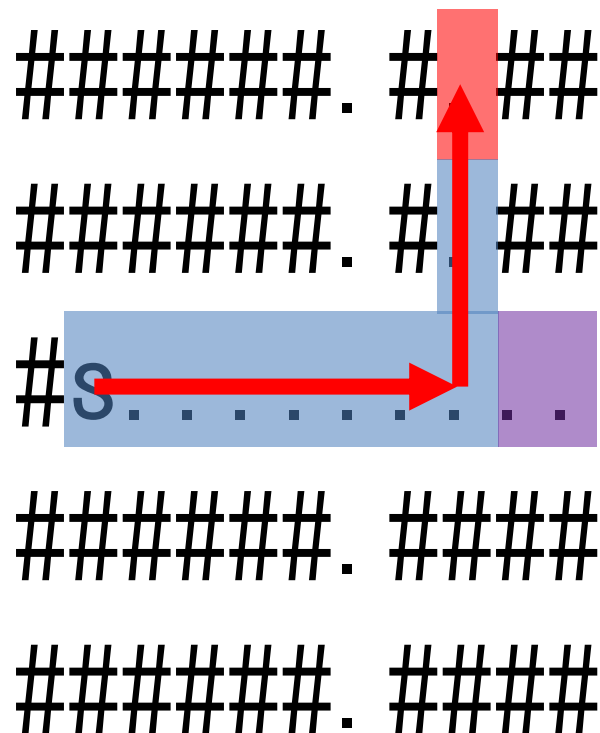
ここはまだ上に行っていない！

戻ってきて別のところに行けたらそっちに行く

#####. #. ##
#####. # ##
#S.....
#####. #####
#####. #####

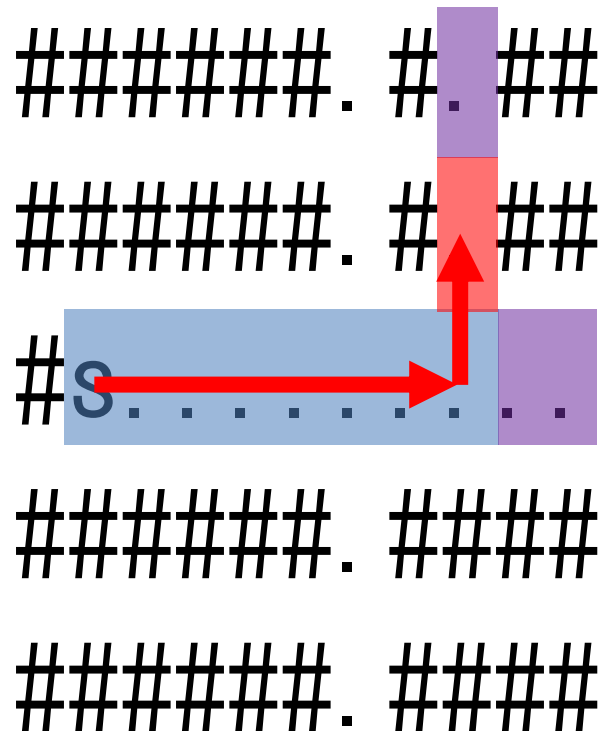
なので今度は上に行ってみます

戻ってきて別のところに行けたらそっちに行く



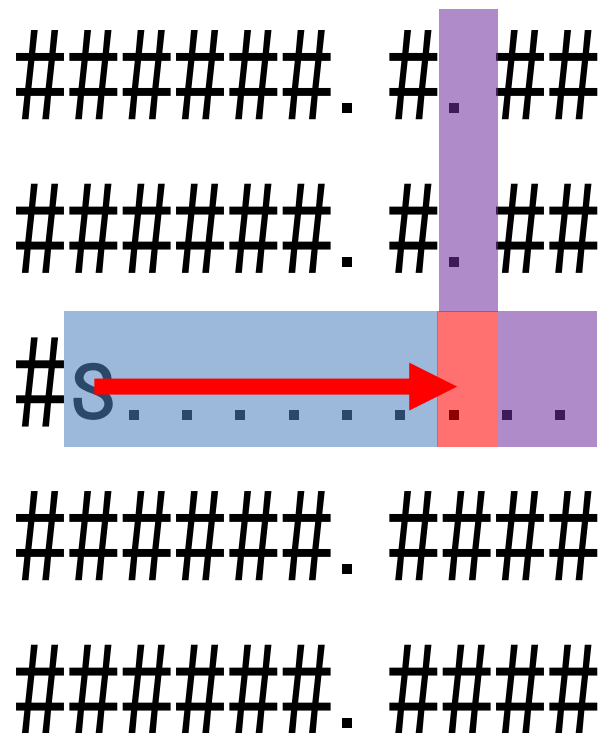
一番上まで行ったらまた戻ります

戻ってきて別のところに行けたらそっちに行く



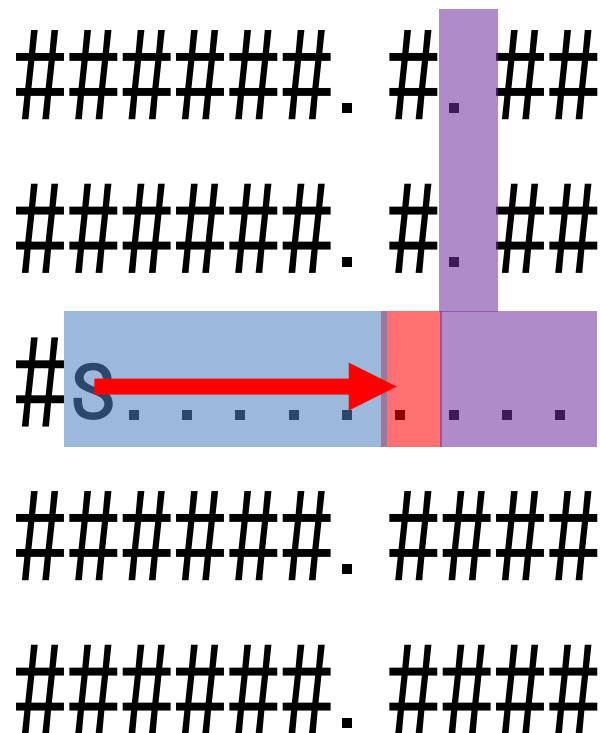
一番上まで行ったらまた戻ります

戻ってきて別のところに行けたらそっちに行く



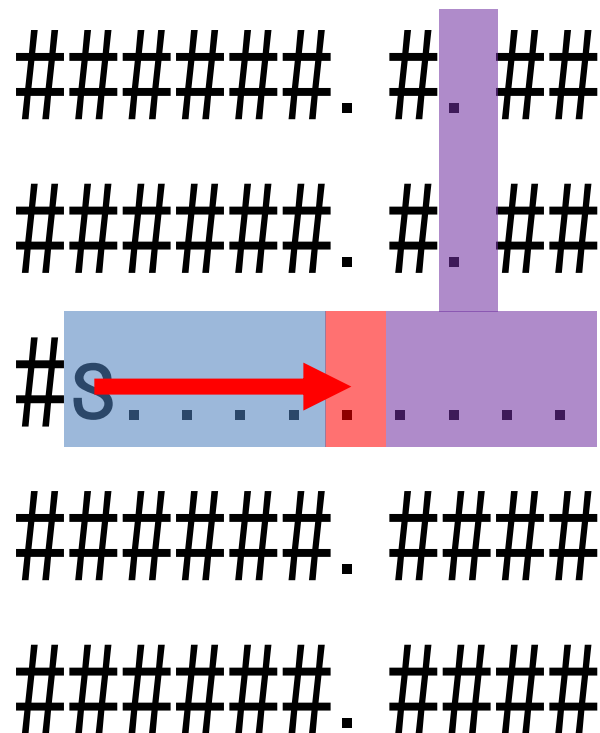
一番上まで行ったらまた戻ります

戻ってきて別のところに行けたらそっちに行く



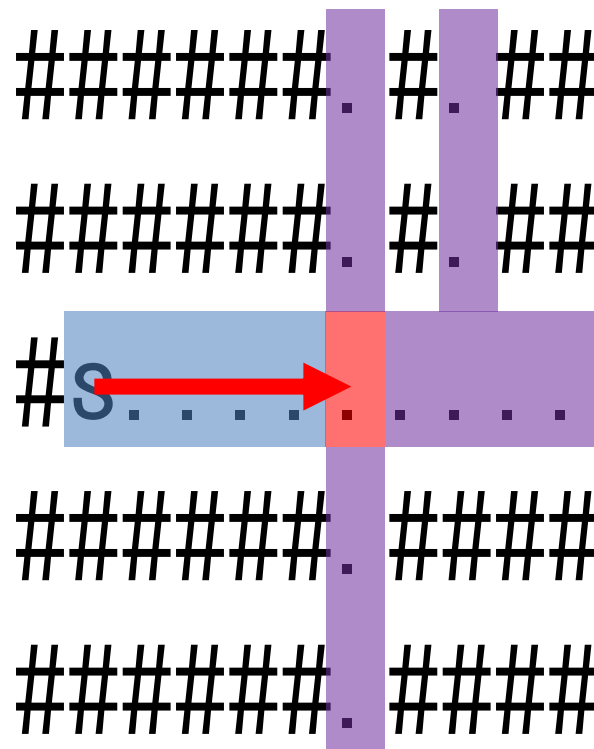
もう行く所がないのでさらに戻ります

戻ってきて別のところに行けたらそっちに行く



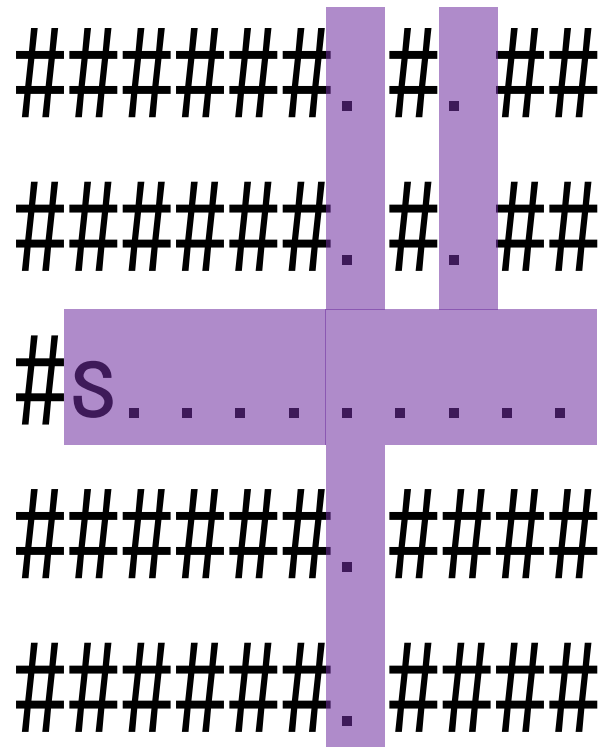
もう一個戻って

戻ってきて別のところに行けたらそっちに行く



上と下も行って帰ってきて (省略)

戻ってきて別のところに行けたらそっちに行く



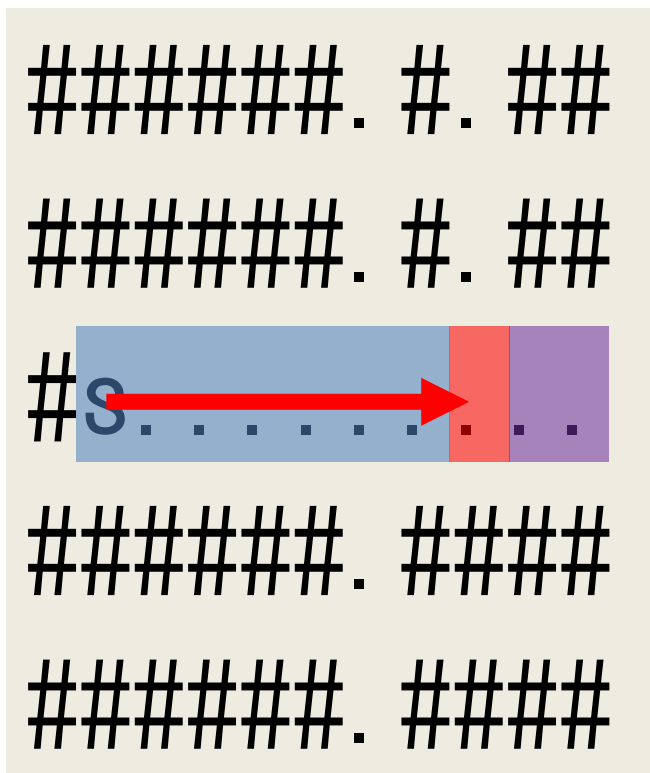
s まで戻ったら完了！

何故これが深さ優先探索？

まだ4方向を試し尽くしていない場所の中で、一番「深い」場所から探索を広げるから

- 試し尽くしていない場所 = 青 or 赤
- 深い = s から遠い
- 一番「深い」場所 = 赤

今後紹介する「幅優先探索」等との対比でよくわかると思います



再帰関数による 深さ優先探索

考え方

- 位置を引数にした再帰関数を使う
- 自分から 4 方向への呼び出しを行う

何故これで深さ優先探索になる？

- 再帰関数なので呼び出した後戻ってくる
- 戻ってきて違う方向へまた呼び出す

```
関数 search(x, y) {  
    if (場所 (x, y) が壁か迷路の外) return  
    if (既に (x, y) に一度到達している) return  
    (x, y) に到達したということを記録
```

```
// 4 方向を試す
```

```
search(x + 1, y) // 右
```

```
search(x - 1, y) // 左
```

```
search(x, y + 1) // 下
```

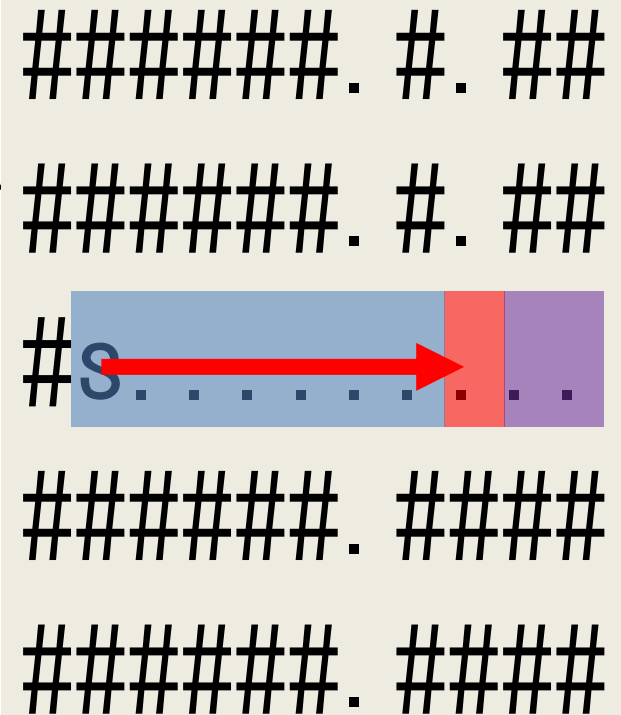
```
search(x, y - 1) // 上
```

```
}
```

再帰関数なので、1つの方向を試し終わった後、帰ってきて次の方向を試す
→ 全ての位置から4方向を試し尽くせる

関数 search をスタートの座標から呼び出す

```
関数 search(x, y) {  
    if (場所 (x, y) が壁か迷路の外) return  
    if (既に (x, y) に一度到達している) return  
    (x, y) に到達したということを記録  
  
    // 4 方向を試す  
    search(x + 1, y) // 右  
    search(x - 1, y) // 左  
    search(x, y + 1) // 下  
    search(x, y - 1) // 上  
}
```



- 色が付いている所 = 「到達した」と記録されている所
- 青 = 再帰関数で呼び出し中の（後で戻ってくる）場所
- 赤 = 再帰関数で今訪れている場所
- 紫 = 既に再帰関数から抜けた（もう戻ってこない）場所

```
int W, H; // 横幅と縦幅
char maze[MAX_W][MAX_H]; // 迷路
bool reached[MAX_W][MAX_H]; // 到達できる？

void search(int x, int y) {
    // 迷路の外側か壁の場合は何もしない
    if (x < 0 || W <= x || y < 0 || H <= y || maze[x][y] == '#' ) return;
    // 以前に到達していたら何もしない
    if (reached[x][y]) return;

    reached[x][y] = true; // 到達したよ

    // 4 方向を試す
    search(x + 1, y); // 右
    search(x - 1, y); // 左
    search(x, y + 1); // 下
    search(x, y - 1); // 上
}
```

関数 search をスタートの
座標から呼び出す

スタックによる 深さ優先探索

考え方

- 試すべき位置をスタックで管理
- 再帰呼び出しの代わりに, スタックへ push

アルゴリズム

- 最初はスタート地点をスタックに投入
- **繰り返す**: スタックから位置を取り出して, まだ訪れていない隣があれば push

今回は詳細な解説は省略します.

今後アップロード予定の幅優先探索の解説にて, 深さ優先探索のこの方針での実装についても言及します.

-
- 再帰関数が深くなりすぎると「スタックオーバーフロー」が起こって実行時エラーになることがあります.
 - スタックを用いたこっちの実装法ならば回避できます.

おわりに

塗りつぶしの他の出番

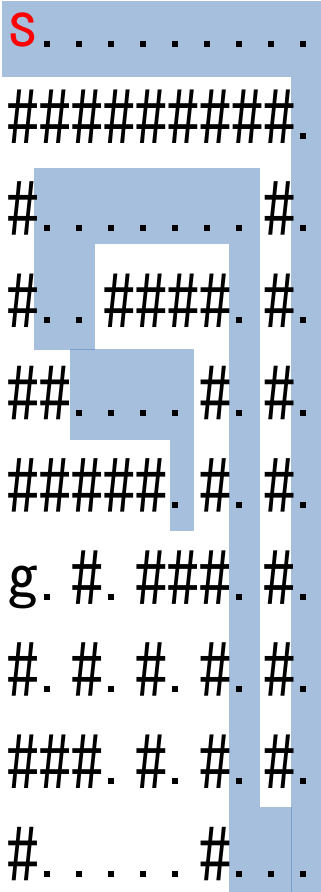
- 行ける領域の大きさを計算する
- 連結なグループの個数を数える
- などなど.....

その他の塗りつぶしアルゴリズム

- 今回は深さ優先探索による方法を紹介
- 塗りつぶしを行う方法は多数存在

詳しくは以下を参照

http://en.wikipedia.org/wiki/Flood_fill



```
S . . . . . . . . . .
#####.
# . . . . . #.
# . #####. #.
## . . . #. #.
#####. #. #.
g. #. ####. #.
#. #. #. #. #.
####. #. #. #.
#. . . . #. . .
```

深さ優先探索の他の出番

- 組合せを全て試す**全探索**
- その高速化である**枝刈り探索**

深さ以外を優先する探索

- 幅優先探索
- 最良優先探索 (→ Dijkstra / Prim のアルゴリズム)

塗りつぶしは幅優先探索などでやっても正しく行える。
深さ優先探索を使う利点は、再帰関数で簡単に書けること。