Algorithm to generate Maps

Description and Proof of Correctness

Surname, First name:	SCHNEIDER, Fabian
Matriclenumber:	11709041
E-Mail:	faebl.taylor@pm.me
Date:	December 13, 2020
Version:	Draft
Subject:	Software Engineering 1

universität wien

Contents

1 Introduction

As part of the implementation of a client for a game, the client should generate a map. The map produced is constraint and should only be used when correct. The following constraints and definitions were given.

- The map to be generated should be 4x8 fields
- a field is either a water, grass or mountain field
- there must be at least 3 mountain fields, 15 grass fields and 4 water fields on a map
- there may be no isles
- the long side may maximally contain 3 water fields
- the short side may maximally contain 1 water field
- at least five fields of the map must be different on two different maps (randomness)

To achieve these goals, I devised an algorithm that computes a completely random map that conforms to the specification and proved its' correctness.

2 Description of the Algorithm

The Algorithm consists of three parts:

- The first part computes a random map without any isles on it
- The second part validates a map without isles for the other constraints
- The third part generates a stream of maps and filers them using the second part until a valid map is returned.

The algorithm makes heavy use of the functional map-reduce-filter approach.

2.1 An Isle-free map

2.1.1 Definitions

Definition of a Map
A map is any two dimensional matrix of Terrains.

Definition of Terrain
Let $x \in \{ WATER, GRASS, MOUNTAIN \}$ be called a Terrain

Definition of an Isle
An Isle of x in a matrix be an area that is horizontally and vertically bounded by either

- The elements in the first or last row or column of a parent matrix
- elements x known as boundary elements
- a combination of the two options above

Definition of an area
an area is a smaller matrix that can be embedded (meaning it has the same components in the part where it can be embedded) in its' parent matrix.

Using this definitions there are 4 types of isles:

Type 1 Isles
A Type 1 Isle is an area of a Matrix whose outermost elements are of type x. The corners can be of any type.
an NxM Matrix A isan Isle in P if

- it is an area of the parent matrix P
- $\left\{ \begin{matrix} A_{[1..N-2][0]} = x & A_{[0][1..M-2]} = x \\ A_{[1..N-2][M-2]} = x & A_{[N-1][1..M-2]} = x \end{matrix} \right\}$
- and all other elements are arbitrary.

e.g.
$$A = \begin{bmatrix} \lambda_I & x & x & x & \lambda_{II} \\ x & \lambda & \cdots & & x \\ & & & & \\ x & \vdots & \ddots & & x \\ \lambda_{III} & x & x & x & \lambda_{IIII} \end{bmatrix} \quad P_A = \begin{bmatrix} \mu_i & \cdots & \mu_{ii} \\ \vdots & A & \vdots \\ \mu_{iii} & \cdots & \mu_{iiii} \end{bmatrix}$$

Type 2 Isles
A Type 2 Isle is an area encompassed by the boundary element x on three sides and the elements of the first or last row or column of the parent matrix on the remaining side. The edges of the isle that are not deleimited by x are of any type.
a NxM matrix A is an Isle in P with boundary element x if

- A is an area of P
- $\left\{ \begin{matrix} A_{[1..N-1][0]} = x & A_{[0][1..M-2]} = x \\ A_{[1..N-1][1..M-2]} = P_{[\lambda+N-1][\mu+1..\mu+M-2]} \end{matrix} \right\}$ where λ and μ are the index of the upper left corner of the area A in P.
- a variation of the above using another boundary in P

e.g.
$$A = \begin{bmatrix} \lambda_I & x & x & x & \lambda_{II} \\ x & \lambda & \cdots & & x \\ & & & & \\ x & \vdots & \ddots & & x \\ x & P_1 & P_2 & P_3 & x \end{bmatrix} \quad P_A = \begin{bmatrix} \mu_i & \cdots & \mu_{ii} \\ \vdots & A & \vdots \end{bmatrix}$$

Type 3 Isles
a variation of Type 2 using two borders in P
e.g.
$$A = \begin{bmatrix} x & x & x & x & \lambda_{II} \\ P_4 & \lambda & \cdots & & x \\ & & & & \\ P_5 & \vdots & \ddots & & x \\ P_6 & P_1 & P_2 & P_3 & x \end{bmatrix} \quad P_A = \begin{bmatrix} \mu_i & \cdots & \mu_{ii} \\ A & \ddots & \vdots \end{bmatrix}$$

Type 4 Isles
a variation of Type 2 using three borders in P
e.g.

$$A = \begin{bmatrix} P_7 & P_8 & P_9 & P_{10} & x \\ P_4 & \lambda & \cdots & & x \\ P_5 & \vdots & \ddots & & x \\ P_6 & P_1 & P_2 & P_3 & x \end{bmatrix} \quad P_A = \begin{bmatrix} A & \cdots \end{bmatrix}$$

Note that this is a special case and can also be defined as follows:
If one row or column of P only consists of the border element x there are two Type 4 isles, each one delimited by three boundaries of P and the boundary created by x.
e.g.

$$P = \begin{bmatrix} \lambda_i & \cdots & x & \lambda & \cdots \\ \vdots & \ddots & x & \vdots & \ddots \\ \lambda & \cdots & x & \lambda & \cdots \end{bmatrix}$$

2.1.2 Algorithm for isle free maps

The following algorithm is used to compute NxM isle free maps. 'G' is used for arbitrary terrain elements, 'W' is used to denote the water boundary element.

```
map = [[]]
lt = []
for(i = 0; i < N, i++){
    for (j = 0; j < M, j++){
        character = randomCharacter('G', 'W');
        if(character == 'G') {
            lt += 'G'
        } else {
            if (i > 0 & j > 0 & j < M-1
                    & (map[i-1][j-1] = 'W' || map [i-1][j+1] = 'W')) {
// 1
                lt += 'G'
            } else if(i > 0 & j = 0 & map[i-1][j+1] = 'W') {
// 2
                lt += 'G'
            } else if(i > 0 & j = M-1 & map[i-1][j-1] = 'W') {
// 3
                lt += 'G'
            } else if (i = N-1 & map[i-1][j] = 'W') {
// 4
                lt += 'G'
            } else if (j = M-1 & map[i][j-1] = 'W') {
// 5
                lt += 'G'
            } else {
// 6
                lt += 'W'
            }
        }
    }
}
return map
```

2.1.3 Proof of correctness for the isle-free-maps algorithm

Theorem generating NxM Matrices using the described algorithm results in isle free maps.
I will show this by disproving thta each isle type can occur if a map is generated using the described algorithm. Only this step is needed, because simple rectangular isles make up more complex isles in a map. By disallowing simple islands, complex islands are automatically impossible[1].

[1] This is not in the scope of this proof.

Lemma 1: the algorithm produces maps that are free of Type 1 Isles.
Proof
Assuming f generated a map containing a Type 1 isle, there would need to be a way to

- generate a series of horizontally adjacant x
- generate a series of vertically connected x at posititon $M_{[\mu+1][\lambda-1]}$ where μ is the row index of the first series and λ is the column index of the first series.

Assuming the previous row consists just of W (except for the boundaries so $\begin{bmatrix}\lambda & \cdots & & \lambda \\ \lambda & W & \cdots & \lambda\end{bmatrix}$) generating the next row we will have to check 1, 2 and 3.

- if we want to produce an x on the boundary, 2 recognizes the first W in the line and would place a G.
- placing it somewhere in the middle results in 1 recognizing the W in front or behind this position on the previous row, placing a G.
- 3 prevents a W being places at the end of the row.

Since we cannot construct a necessary edge E for the Isle $E = \begin{bmatrix}\lambda & W \\ W & \lambda\end{bmatrix}$ there can be no Type 1 isles in the result matrix.[2]

Lemma 2: the algorithm produces maps thta are free of type 2 isle.
Proof
The proof follows the same steps as the proof for Lemma 1.
since we canot construct an edge in the for of $\begin{bmatrix}\lambda & W \\ W & \lambda\end{bmatrix}$ or $\begin{bmatrix}W & \lambda \\ \lambda & W\end{bmatrix}$ that could be the upper or lower left and right edges of a type 2 isle, the algorithm will not produce a matrix containing type 2 isles.

Lemma 3 the algorithm produces maps that are free of type 3 isles.
Proof
This again follows the same logic as Lemma 1 and 2.
for a type 3 isle, we need at least one edge that is not adjacent to the boundaries of the parent matrix. since we cannot construct this (see above), the algorithm cannot produce maps with type 3 isles.

Lemma 4 The algorithm produces maps free of type 4 isles.
Proof
Type 4 isles arecharacterized by a horizontal or vertical series of x that span the parent. assuming $A = \begin{bmatrix}\lambda & \cdots & \lambda \\ \vdots & \ddots & \vdots \\ w & w & x\end{bmatrix}$ or $A' = \begin{bmatrix}\lambda & \cdots & w & \lambda \\ \vdots & \ddots & w\vdots & \\ \lambda & \cdots & x & \lambda\end{bmatrix}$
where x is the element currently being generated, 4 and 5 are relevant:

- 5 prevents us from inserting W into A
- 6 prevents us from inserting W into A'

[2] In simpler words: we cannot construct the edges of a Type 1 isle.

Since there are no other type sof isles and these types cannot be generated, we can conclude, that the described algorithm produces isle free matrices. Q.E.D.

2.2 Converting specifications into filter constraints

We finished the first part of the algorithm according to the specifications:

- The map to be generated should be 4x8 fields (DONE)
- a field is either a water, grass or mountain field (DONE)
- there must be at least 3 mountain fields, 15 grass fields and 4 water fields on a map
- there may be no isles (DONE)
- the long side may maximally contain 3 water fields
- the short side may maximally contain 1 water field
- at least five fields of the map must be different on two different maps (randomness)

The complete algorithm now creates a stream of isle free maps and filters them using the following filters to establish correctness of the map:

```
frequency(Mountain in map) >= 3          // minimum count of mountains
frequency(Grass in map) >= 15            // minimum count of grass
frequency(Water in map) >= 4             // minimum count of water
frequency(Water in first column) <= 1    // water on small edge
frequency(Water in last column) <= 1     // water on small edge
frequency(Water in first row) <= 3       // water on big edge
frequenct(Water in last row) <= 3        //water on big edge
```

Where $freuqency$ is a function that counts the occureence of a terrain field in the given list or matrix of fields.

2.2.1 A Java implementation

For an implementation of the algorithm using Java streams and filters consult the map generation implementation found in the game implementation on GitLab.