Computing Sea Level Rise

Jake Adicoff, Ethan Zhou
CSCI 3225: Algorithms for GIS

1 Introduction

Simply stated, the problem of sea level rise is to determine the land mass that is flooded when the sea level rises by a specified ammount. In this paper, we provide a time efficient algorithm and implementation details that allow fast computation to find flooded area when sea level is increased. The input for the problem is the GIS standard ASCII (.asc) file that encodes an elevation grid. The importance of sea level rise is far reaching - knowing flooded area allows us to determine land area that is at risk of flooding should the sea level rise due to climate change. Some research suggests that the sea level could rise as much as 6 feet as glaciers begin to melt more rapidly. Our algorithm allows us to visualize areas affected by this change.

2 Algorithm

We present a linear time algorithm to determine flooded area in a terrain for any specified range of values for sea level rise (SLR). Before proceding with the algorithm, we must discuss the data structures we use. First, we read in elevation data and store it in a static array. We will refer to this array as the *Elevation Grid* (`EG`). We maintain a dynamically updated array with the same dimensions as the `EG`. This second array stores the sea level at the corresponding cell in the `EG` is flooded. We initialize this grid to the `NODATA` value (generally -9999). We will refer to this array as the *Flood Level Grid* (`FLG`). We maintain 2 stack (FIFO) structures and we will refer to them as `current_stack` and `next_stack`. The stacks hold a class that we define. Our class is simple and holds values for row, column and elevation of the corresponding grid cell in the `EG`. We use this class to maintain locality when using the stack. Finally, we maintain a dynamically updated unordered set (essentially a hash tabel but instead of a key-value pair, there is only a key). This table hashes instances of our class and we will refer to it as the *Cell Hash* (CH). In section 3 we will discuss the reason for our use of this structure.

Our algorithm is a modification of a *Depth First Search* (DFS) through the `EG`. Instead of a recursive implementation, we choose a stack implementation. We begin the algorithm by determining points that we assume to be ocean. Water in an elevation grid is usually stored as the `NODATA` value (or sometimes 0), and we can make the assumption that if a land mass completely surrounds an area of water, it is not ocean. Thus ocean must be lie on the boarder of the elevation grid and will have elevation of `NODATA` or 0. We traverse the parimiter of our grid and push all points that boarder points that we determine are ocean onto `current_stack`.

Afther this is complete, we begin the DFS. We itearate over values of SLR from 0 to a user specified constant max_SLR. For each value (we call this value the `current_SLR`), we begin with `current_stack`. While `current_stack` is not empty. We pop a cell instance off of the stack and update the corresponding cell in the FTG with the `current_SLR`. We then look at all 8 neighbors of the cell in the `EG`. If the `current_SLR` is greater than or eaqual to a neighbors elevation, or if the neighbor is `NODATA`, we push this neighbor onto the stack. Before doing this however, we check to see that it's corresponding cell in the FTG is not `NODATA` and that it has not already been pushed onto `current_stack`. If the neighbor's elevation exceeds the `current_SLR`, we push the cell onto `next_stack` to be processed later.

after an itearation of this, `current_stack` is empty, and `next_stack` has Cell objects to be processed. We set `current_stack` equal to `next_stack` and then empty `next_stack`. After this, `current_SLR` is incremented, and the process is repeated. When `current_SLR` = `max_SLR`, the algorithm terminates, and we are left with a fully updated FTG that specifies the exact SLR at which a grid cell is flooded. If a cell of the `FLG` is `NODATA`, then it is not flooded by any SLR below `max_SLR`.

Analisys is a bit complicated, but the algorithm runs in $O(n)$ time, where n = the total number of cells (number of rows times number of columns in the `EG`). Every Cell has an initial push event (1 at most), where that cell iseither pushed onto `current_stack`, or the element is pushed onto `next_stack`. If the cell is pushed onto `current_stack`, then it is immediately classified and we do not look at the point again. If a cell is pushed onto `next_stack`, we do not look at its neighbors, so the cell acts as a barrier to more interior points of SHIT WAIT, I THINK OUR ALG IS O(n*`max_SLR`). PLEASE CONFIRM.

3 Implementation

By necessity, we have explained most of the details of implementation in the previous section, but we will fill in some small gaps here. First, in our algorithm, we make checks to see that a grid cell is not already on `next_stack`. The c++ data structure has no function to check for existance of an item in a stack, so in every iteration of our main loop (where we increment the value of `current_SLR`), we make a unordered set of cells (as previously discussed). Before we add any cells to `next_stack`, we hash them in this set, and then push. Before pushing neighobrs onto `next_stack`, we check for that neighbor's

existance in the hash, and if it exists, we do not push it onto the stack, and simply continue. I HAVE A BURNING QUESTION, WHY ISNT THERE A HASH FOR THE CURRENT STACK?

Other minor details, we bullet below:

- For ease of coding and integration with Open GL, we did not make a class, and instead use global variales (heavily).
- To 'switch' `current_stack` and `next_stack`, we use pointers. `current_stack` and `next_stack` are actually stack pointers to stacks `a` and `b` respectively. When we need to switch current and next, if `current_stack` points to `a` we set current equal the address of `b` and next to the address of a. We do the analagous but opposite if current points to `b`.
- We take `increment` as an argument. `ncrement` specifies how much the user would like to see the sea level rise (or decrease) when hitting '+' (or '-'). As a matter of convenience, the increment must evenly divide `max_SLR`.
- We hve additional optional arguments for resolution. The variable `computation_resolution` specifies which neighbors we look at. If resolution is set to 1, we look at neighbors distance 1 to a cell. If set to 5, we look at all 8 neighbors distance 5 from the cell (so 4 cells are skipped altogether by the algorithm and we get speedup). the variable `display_resolution` specifies the resolution you would like for rendering with Open GL. Again, 1 is highest resolution, 5 would skip 4 cells while rendering. If a `computation_resolution` is specified, `display_resolution` will be set equal to it and will remain constant at runtime. If unspecified, both are set to 1.

We bullet key press details below (these apply for rendering only):

'+' Increase the sea level

'-' decrease the sea level

'x' rotate about the x axis

'y' rotate about the y axis

'z' rotate about the z axis

'n' increase `display_resolution`

'm' decrease `display_resolution`

4 Experiments

5 Discussion and Conclusion