Javier Ortega

Sebastian Gionfriddo

Elias Mosco

Final project documentation

We found while analyzing the code that there are 3 major possibilities: Either there was too much water and there was no way to prevent any regions from being flooded, there was not enough water to prevent every region from being in a drought, or there is enough water for every region. The last scenario was divided further into two subsections: either there was enough water for every region to not be in a drought, but not enough to meet every need, or there is enough water to meet every need. Initially our group divided the work evenly into each scenario based on how much water is available. If there was too much water, it would run the “Flood” function, if there was too little, it would run the “Drought” function, and if there was not enough it would run the “Perfect” function.

The idea was to try to get as many points as possible based on water availability. In a flood, we would flood 1 region to cause an overflow, then distribute the remaining water to get it out of a flooded state. We would lose some points due to overflow, but it allowed the region to eventually get solved. In a drought, we would choose one region to be stuck in a drought and deliver water to the other regions to try to get them to be out of a drought, and if possible, also meet their needs. Since there was guaranteed to be at least 1 region in a drought in this situation, we found that it was best to use the water elsewhere. In a “perfect” scenario, we would try to meet as many needs as possible and ensure that nothing is in a drought.

Initially, this seemed like the best way to approach the situation, but while trying to solve each scenario, we began running into lots of bugs (many of which are explained later in the document). This overcomplicated things and created code that was confusing and difficult to follow, with tons of if-else statements. We eventually decided to switch to a new solution that is much more simplified. We began by looking at a map that we drew to help us see how the regions and canals are organized.

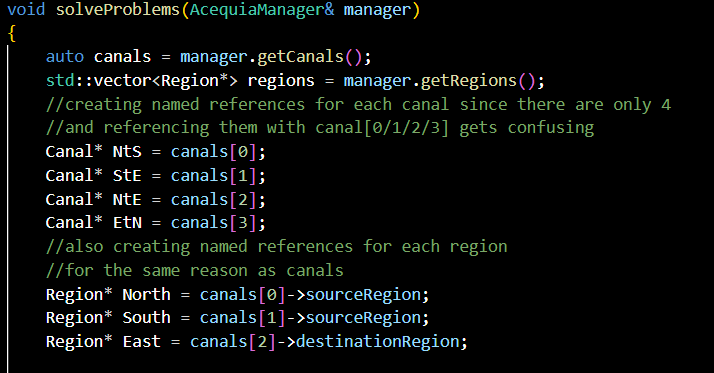
North Region

East Region

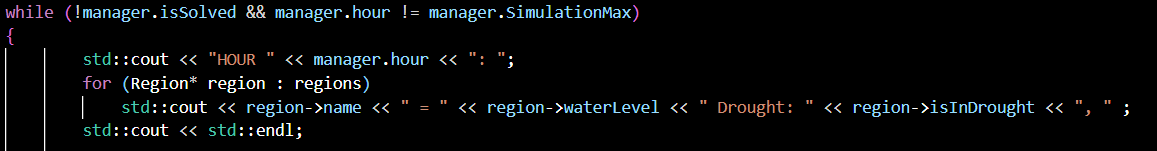
South Region

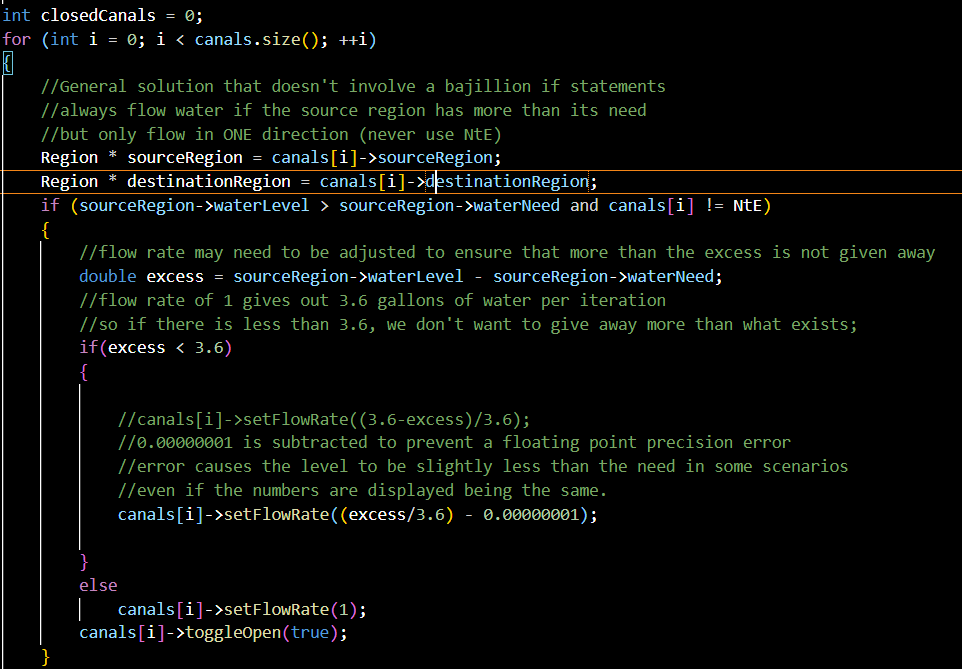
We would first check that the source region of each canal was above its need, and if it was, we would open depending on how much it was above its need, however we always kept the North to east canal closed so it would have the water flowing in one direction. We created simplified solutions that would prevent us from losing points by dropping the water level below regions need or putting a region into a drought.

The top of the function creates named references to each canal and region. This makes accessing them in certain cases easier, especially during debugging.

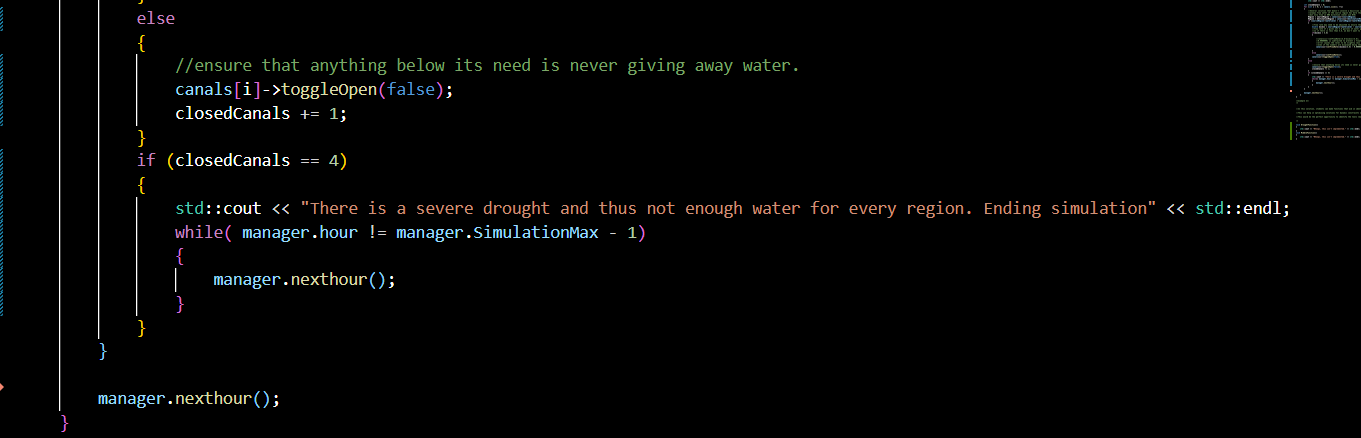


Next, we start the main loop. At the top of the loop, we output the hour and how much water is in each region to help us understand what is going on. This was particularly useful during debugging, as this effectively creates a sort of log for us to look at.



Next, we declare a counter variable and loop through each canal. The canal only flows if it has more than the source region’s water need. We also exclude the canal going from North to East, for the reasons previously stated. The flow rate is determined by how much a source region is above its need. A flow rate of 1 (The maximum allowed) gives an output of 3.6 gallons per hour. If we have less than that, we divide the actual amount by 3.6 to ensure we don’t dip below the need. We found that in some scenarios, we were being penalized despite meeting the water need. After investigation found that it was due to a floating point rounding error and subtracted a very small amount from the flow rate to compensate without interfering with most other cases.

Finally, if the canal in question did not meet the previous conditions for being opened, we close it and add to our counter. If the counter variable is at 4, that means every canal is closed and no changes will occur. This will only occur if there is not enough water for every region, because otherwise the solution will eventually get solved. A message is output stating that there is an extreme drought and not enough water, and the simulation is rushed ahead using manager.nextHour() in a while loop to prevent the screen from being covered in pointless output statements for every hour that it would otherwise be forced to run due to being unsolvable.



In the cases where there is enough water for the region to be solved, the solution tends to work very quickly. During our experimentation we did not find any scenarios where it took more than 10 hours to solve.

A screenshot of a computer

AI-generated content may be incorrect.

Initial Issues:

We found when working on the project that there were a couple of edge cases that were left ignored. For example, you had the possibility of Draining excess water from an empty zone, allowing you to pull infinite water. Along with that, the grading made our project quite difficult. The random values offered no guarantees that we would have enough water, or even that we would have enough water to fill up the zones at all. There were also several times where misconceptions created whole chunks of code that were later deleted. For example, we misunderstood the way that penalties were scored. This meant that we could spawn in situations where we are cursed to infinite negative points, regardless of what code we wrote, which was rather demotivating. Without knowing how final scoring will be measured it made it hard not to abuse holes in the code like infinite water to make up for it (impossible answer for impossible problem).

We also noticed that total drought was decided based off the amount of total capacity it had, not total need. This did not make sense logically, because no matter how you cut it, droughts are based off the amount of water you NEED, not the water you “Could hold”, so if removed the limit of capacity, we could have a drought even after reaching Need, if need was too far below capacity. To follow that, in the current program if the water needs spawns equal to capacity (100=100), then to reach need you MUST flood, and there is no safeguards for these situations. These issues played an important role in our final decision, as many of our solutions had holes in them that would pop if the wrong prompt was added.