Midterm Project Documentation

Avery Peiffer

Aep65

To solve this problem, I first implemented a solution in Java, which is included in these files. You will notice that my solution includes a lot of extra code to visually walk through the steps of each round. I felt that this was extremely useful for debugging my high-level solution, and it would help later when I was debugging my MIPS solution with the tests included in the project files. For the Java solution, I started with the first test case (3 candidates, 5 voters). When I verified that the results were correct, I then added in the code for the second test case, and so on.

I think that one of the biggest challenges in programming a complex algorithm in MIPS is keeping track of which registers were available to use in a given step and which could not be changed. Working from essentially a top-down perspective using my high-level implementation meant that I could allocate registers as needed before I began a given step. Instead of writing code for the first test, making sure it was correct, and moving on, I instead wrote the same code that had been in my Java implementation. It took longer to finish the code, and there were plenty of bugs that occurred along the way, but the code worked for all four test cases when finished.

I first validated my code using my high-level implementation and the test cases included in the project files. Since my Java code allowed me to see all the intermediate steps in each round of voting, such as the number of ones in each row, I was able to trace the entire way through my MIPS implementation to find exactly where and when an error was occurring.

The structure of my solution is a large network of nested loops. The outer loop (*beginround*) runs up to n – 1 times, where n is the number of candidates in the election. Inside this outer loop, there are four main sub-blocks of code. First, *foreachrow* and *foreachcol* count the number of ones in each row by jumping through the data as needed. I created an array in the .data section called *numofones* which is an array of eight zeroes. This is reflective of the number of candidates that the solution is expected to handle. I was researching ways to dynamically allocate memory for an array in MIPS, as this would be preferred to creating a fixed array, but I felt that changing my solution to do this was less important than creating a working solution. *Foreachrow* and *foreachcol* work to assign the number of ones in each row to *numofones*. These nested loops have *O(mn)* complexity, where *m* is the number of voters and *n* is the number of candidates.

The next set of for loops, starting with *findwinnerandloser*, walks through the *numofones* array to find the winner and loser of each round. This sub-segment has complexity of *O(n),* where n is the number of candidates. *Putinresults* starts the next sub-segment, which puts the winner and loser into the *results* array. This block has complexity of *O(1).* *Deleteloser* starts the last sub-segment. This block goes through the array and deletes the value of the round loser for each voter. Then, it reassigns votes as necessary for that voter. This block has complexity of *O(mn),* where m is the number of voters and n is the number of candidates. Therefore, my solution has an overall complexity of *O(mn2).*

I formulated a few tests in addition to those given in the project files to make sure my code could handle as many corner cases as possible. In the first test, labeled *unpopularcandidates.asm*, two of the five candidates had zero first-place votes. However, all four rounds were required to decide a winner. I wanted to make sure that my code could correctly handle a candidate having zero ones since each of the test cases had at least one first-place vote for each candidate.

My next test was one in which a three-way tie occurred, labeled *threewaytie.asm*. This test required me to be more careful when finding the winner and loser of the round. At first, the winner and loser were both initially assigned to be the first value in the *numofones* array. When another value was strictly less than or greater than that first value, the winner and loser would be reassigned accordingly. This does no good if no other values are strictly less than or strictly greater than the first value – both the winner and the loser will be the first candidate by this method. To rectify this, I had my code ‘manually’ set the round loser to be the latter of the two.

My final test was similar to Test 4, in that it combined concepts from the three-way tie test with the other tests in the project files. With eight candidates and eight voters, it forced my code to handle an eight-way tie scenario, similar to that described in the above paragraph, and then deal with other scenarios throughout the algorithm before producing a winner.