Dynamic Programming Matrix vs. Skiena’s Dynamic Programming Report

Aidan Sherr

Southern New Hampshire University

**Executive Summary**

**Purpose**

The purpose of this project was to demonstrate the differences between dynamic programming using a matrix to find a solution, versus Skiena’s dynamic programming which involves a recursive approach with minimal recursive steps. The problem used to demonstrate these differences was to detect whether a string of characters is an in-order jumble of two other strings. The comparison between the two solutions consists of runtime, amount of effort for programming, and the resource demands of each individual approach.

**Summary**

The initial approach was with the traditional matrix based dynamic programming. This takes the strings, and builds a Boolean matrix that stores true or false based on whether the current character in string one or string two matches the current character in string three, and whether or not the previous result for the string found was true. If there is a matching character and the previous result was true, a true is stored at that position in the matrix, otherwise a false is stored. The final solution to the problem will be in the last index of the matrix.

The next attempt was with Skiena’s dynamic programming approach. This is a recursive algorithm that checks at each stage of the recursion to check if current state has at least one string’s current character matching the jumbled string’s current character. If not, then it is not a true jumble and the algorithm exits without recursing to the end. If the algorithm is able to recurse to the point where there are no characters left, then it is a true jumble and that result is returned.

**Conclusion**

The intent behind this project was to find the differences between these two algorithms. Each approach successfully solved the problem, but there were significant differences found between the two. First, through my experience, I found Skiena’s approach to be much simpler and easier to program. The matrix approach’s logic required the previous check to be considered, whereas the recursive approach would only continue if the previous check was found to be true. Second, the matrix algorithm ran longer than Skiena’s algorithm and ran at a consistent time, even if the result was false. In my experiment, I found that Skiena’s solution exited sooner if no jumble was found, as it did not rely on the entire matrix to be solved to determine a solution. My third observation was that Skiena’s approach required more data to be placed on the stack to determine a solution, compared to the matrix approach which only required a copy of each string, its size, and a 2D matrix. Overall, I found Skiena’s approach to be faster and simpler, but could pose some problems on the demand of resources.

# **Overview**

The purpose of this project was to demonstrate the differences between dynamic programming using a matrix to find a solution, versus Skiena’s dynamic programming which involves a recursive approach with minimal recursive steps. The problem used to demonstrate these differences was to detect whether a string of characters is an in-order jumble of two other strings. If the string is a jumble, the two combined strings will be interleaved with their characters in the correct left to right order. The comparison between the two approaches consists of runtime, amount of effort for programming, and the demands of each individual approach in terms of resources.

The initial approach was with the traditional matrix based dynamic programming. This was done by taking in the character arrays and their sizes in the constructor and creating and initializing the dynamic programming matrix to the size of size1+1 x size2+1 where size1 and size2 are the two smaller strings being searched for in the jumbled string. Once the matrix is created, isJumble() is called. isJumble() starts with two nested for loops to iterate through string one and string two and check if each character is equal to the current character in string three. If one is found to be equal and the previous check was true, then a true is stored at that location in the matrix, otherwise a false will be stored. Once all strings have been completely exhausted, the last index in the matrix will contain the solution to the problem.

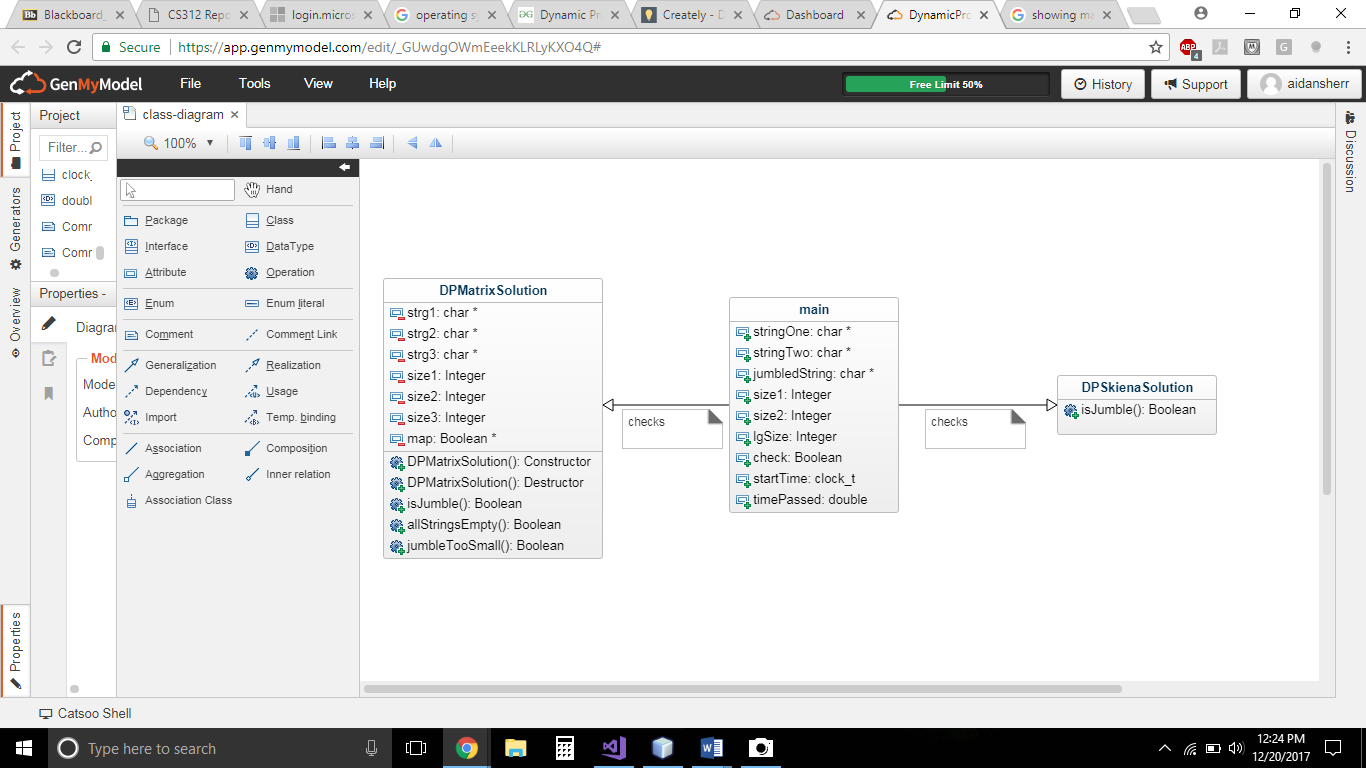
The next attempt was with Skiena’s dynamic programming approach. This did not require any initialization, meaning I was able to jump right to calling isJumble(). isJumble() is a recursive algorithm that checks at each call, whether strings are empty or whether neither character matches. If the strings are empty, that means the check for a jumble was true and the method returns. If neither character matches, the check for jumble is false and the method returns. This allows for a quick exit at each check. If these two cases are not found, then the method checks whether the two smaller strings have the same character at the current position, and if so, recurses one level beyond to check which character gives a true result. If neither gives a true result, then the check for a jumble is false and the method returns. If one is found to be true, the method continues by advancing one character in that string and the jumbled string and recursing. The only other two cases are just string one matches string 3 at the current position, or just string 2 matches string 3 at the current position, in which case the method would recurse in the same way by advancing one character ahead in each matching string.

I compared each approach’s runtime by running both solutions on the same strings 100,000 times (both algorithms were too quick to be detected if they were only run once), both a true jumble and a false jumble and recorded the resulting runtime. I then found the average of each algorithms runtime after 5 attempts both a true and false jumble.

# **Class Structure**

I implemented a class for each algorithm. The first class was the DPMatrixSolution class which had private variables that held the matrix, each string, and each strings size. This was all initialized through the class’s constructor. The class also had two private function. One checked if all strings were empty, which would mean the jumble was true since there were no strings to compare. The other checked if the jumble was not the same size as size of string one plus size of string two, which would mean the jumble was false. The only public function for this class was to check for the jumble, which built the matrix based on each character position and returned the result in the last index of the matrix.

The second class was DPSkienaSolution, which did not store any variables, and only consisted of the public function to check for a jumble. This function calls itself recursively and passes strings of characters based on the current evaluation of the problem. If the function gets to a point where each string is null, then it returns true, as it has found the larger string to be a jumble of the other two strings.

The main function handles all user input and output, creates the objects for each algorithm and runs the check, as well as creating a timer to compare each algorithms runtime. Here you can see a UML diagram explaining each class and how it relates to the entire program.

# **Language, Libraries, and IDE Chosen**

My program was written in C++. I chose this language because it is the language I am most comfortable using. I used the Visual Studio 2017 IDE for development. This IDE gives me robust tools for debugging and it is the IDE I have the most experience programming in with C++. For libraries I used iostream for input and output with the user, string for the initial user input, algorithm to convert the string to all lowercase characters, and ctime for the runtime measurement.

**Results**

In my experiment, I found that both Skiena’s solution and the matrix solution solved the problem effectively. Skiena’s algorithm exited sooner if no jumble was found and was overall faster. I used the ctime library to measure the amount of time it took for each algorithm in milliseconds. For the same strings, Skiena’s solution took an average of 95 milliseconds compared to an average of 370 milliseconds for the matrix solution when the jumble was true. With a false jumble, the matrix solution took the same amount of time, but Skiena’s solution took only an average of 63 milliseconds.

# **Conclusion**

The intent behind this project was to find the differences between these two algorithms. Each approach successfully solved the problem, but there were significant differences found between the two. First, through my experience, I found Skiena’s approach to be much simpler and easier to program. The matrix approach’s logic required the previous check to be considered, whereas the recursive approach would only continue if the previous check was found to be true. Second, the matrix algorithm ran longer than Skiena’s algorithm and ran at a consistent time, even if the result was false. In my experiment, I found that Skiena’s solution exited sooner if no jumble was found, as it did not rely on the entire matrix to be solved to determine a solution. My third observation was that Skiena’s approach required more data to be placed on the stack to determine a solution, compared to the matrix approach which only required a copy of each string, its size, and a 2D matrix. Overall, I found Skiena’s approach to be faster and simpler to implement, but could pose some problems on the demand of resources. This result is unexpected for me because I tend to use the iterative approach over the recursive approach more often, but for this problem I found Skiena’s way to be much simpler. The matrix solution, while long-winded, provided a very logical solution and a clear path to show how the problem was solved. For longer problems, I could see the matrix solution being more clear and less demanding for stack space.