Project 3 Report: The Gridiron Guru

Administrative

- The Gainesville ‘87ers: Jett Nguyen, Zach Ostroff, and Wiliam Shaoul
- GitHub usernames: JettNguyen, zachostroff, and wshaoul
- GitHub repo: https://github.com/JettNguyen/GridironGuru
- Link to video demo: https://youtu.be/fsc1PRHfNWw

Extended and Refined Proposal

Determining the ideal play calls during different situations in an American football game can be tough, especially during the pressuring moments of a game. There needs to be an easy and accurate way to decide what kind of offensive play your team should make. This problem exists as these crucial situations can be the difference between winning and losing a game. Coaches may be putting the team at a disadvantage by making these decisions without the proper statistical background information. When a repertoire of past events combines to make a statistically successful play for a football team, a visible change can be seen in their performance by analyzing their increase in first downs, touchdowns, field goals, and two-point conversions.

The most useful information for a program of this type will be play-by-play data. By collecting individual features of every single play made over 10 years’ worth of football seasons, calculations can be done to find the play that is most likely to succeed or the play that has the best outcome in any given circumstance. Multiple .csv files borrowed from https://nflsavant.com/about.php were put together and curated by our team. Each of the combined 485,462 datapoints represents a play with its attributes included in the columns. However, not all categories are needed because they either do not add statistical significance or result from the specific team’s performance. For example, anything to do with a challenge or a penalty can be blamed on a team’s attitude or technical mistake, so that data is not necessary in the calculations of a successful play. Non-numerical information such as the description, play types, and formations are handy pieces of information for the user so that they can call the play with full intention and no second-guessing.

To get started solving the problem, the first choice is determining the tools, languages, APIs, and libraries that will be needed in the process. For the language, we chose C++ because it is what we have familiarized ourselves with when learning to code various data structures and

algorithms. It is second nature to us at this point since we all have at least a year's worth of experience in the language. Using our own IDEs, we can share code using a GitHub repository. Additionally, within C++, we will need to use libraries such as <fstream> and <sstream> to read the .csv file and additional libraries to implement our data structures.

As a group, we have chosen that the best ways to organize and suggest plays would be through max heaps and hash tables. Firstly, max heaps offer easy organization in a tree fashion. If a functor is constructed to calculate the weightage of how successful two plays are and compare them, the max heap can be constructed with the best plays at the top. We don't want to compare a play that is completely different from the input situation, so a temporary heap is built from the original heap to only include plays that are like the user's circumstances. From there, likelihoods of a first down, touchdown, two-point conversion, and/or field goal are calculated and presented in the console alongside the best historical play. On the other hand, the hash table offers organization in a way that more evenly and fairly distributes a large amount of data. By using hash functions and rehashing, an organized and developed table can be made from a vector of linked lists. Each index would contain a list of similar situations and based on the hash code for the input situation, it would find that index and retrieve all of the plays from the list.

As for the distribution of responsibility and roles, each group member worked hard. Jett focused on the outline and max heap structure the most. Zach worked on the hash table, data gathering, and research. Will also worked on the hash table and making a cohesive program. Each of the group members worked almost equally on the report and evaluation. Everything followed close to plan.

Analysis

Following the proposal, no major changes were made. We decided to stick with our desired data structures, which were the Max Heap and Hash Table. Regarding how we set up our algorithms, however, we decided once we began implementing our structures to use another max heap to organize the plays that were most similar in order by our play rating formula found in ComparePlay.cpp.

Big O worst case time complexity analysis of the major functions/features you implemented

The major functions within this program come from the files PlayHashTable.cpp and PlayMaxHeap.cpp. Both files include functions for reading the data and suggesting the data.

readDataAndPushIntoHeap: O(s*n logn) where n is the number of lines from the .csv file, s is the longest string. It goes through each play in the sheet and within each play, it is pushed into the heap with O(log n).

readDataAndPushIntoHashMap: O(n*s*(m + h*L)) where n is the number of lines from the .csv file, s is the longest string, m is the number of primes, h is the capacity, and L is the length of the LinkedList with the most plays (nodes). It goes through each play in the sheet and within each play, a hash function is called which goes through an undefined number of primes then in the worst case could rehash every time using the capacity as a variable. For each iteration in the rehash function, another iteration through the container of LinkedLists occurs, too.

suggestPlayFromHeap: O(n^2): Each if statement and line before the while loop has a time-complexity of O(1). In the while loop, it goes over every single play, n. In the worst case, every single play will be added into the temp heap. There are 4 possibilities of using brackets for the priority_queue which has O(log n). Then the next thing to take time is an iterator of the new map which in the worst case would be O(n logn) because of the insert function. Then there is a function to help sort with O(n^2), then it may order everything again with O(n). It is overall only O(n^2) because it would combine to be O(nlogn + nlogn + n^2 + n) where n is the number of plays and n^2 overpowers.

suggestPlayFromHashTable: O(n^2): This is almost exactly the same as the previous function. It just gets the data from the linked list and sorts it the exact same way. Each if statement and line before the while loop has a time-complexity of O(1). In the while loop, it goes over every single play, n. In the worst case, every single play will be added into the temp heap. There are 4 possibilities of using brackets for the priority_queue which has O(log n). Then the next thing to take time is an iterator of the new map which in the worst case is O(n logn) because of the insert function. Then there is a function to help sort with O(n^2), then it may order everything again with O(n). It is overall only O(n^2) because it would combine to be O(nlogn + nlogn + n^2 + n) where n is the number of plays and n^2 overpowers.

Reflection

This project was a long process for us as a group, but overall was incredibly rewarding for us to have full control as software engineers. From the start, it was fantastic for us to be able to choose a topic which we were interested in and to work on something that we wanted to work on. With all of us being huge football fans, we were proud to work on and have a project on our portfolios that went alongside our interests.

Dividing ourselves into specific roles initially was done somewhat blindly, as we did not know exactly how much each aspect of the project would demand. While the data and visualization portions were done in somewhat short order, the logic and structural portion required a lot more, meaning that everyone had to step up and aid in that.

A major issue we ran into as a group came in the hash map structure when we started to run into a hash overflow error. We were passing too much data into our arrays after already programming the entire hash function, which caused us to have to scrap most of that work which was already done. We decided to essentially restart that section from nothing after analyzing what caused our errors and how we would be able to fix it. That process took a very long time and was a major hurdle that we ran into as a team.

If we were to restart this project as a team, we would work harder at first to initially get a head start with how we want our code to be structured to not run into the late scares which we had. This way we could spend more time on initial brainstorming of our concepts and leave time for testing between each step.

Each group member has specific takeaways from the project that will help in their future endeavors as a software engineer. Jett's biggest takeaway was learning how these data structures can be used for real-world applications as he outlined two structures to complete the same job. Will learned about how to properly write his own custom hash function. He made sure to keep track of a list of prime numbers to modulus the hash codes, understanding that this would leave the least amount of room for unnecessary collisions to happen in the case that two codes' remainders are factors of each other. Zach's takeaway from the project was learning to understand the programming cycle, and the true value of pseudo coding before starting to write code.

References

https://stackoverflow.com/questions/2340281/check-if-a-string-contains-a-string-in-c

https://www.geeksforgeeks.org/measure-execution-time-function-cpp/

https://www.educative.io/answers/how-to-sort-a-map-by-value-in-cpp

https://www.geeksforgeeks.org/descending-order-map-multimap-c-stl/