Rabin-Karp Algorithm

String pattern matching algorithms are used all the time in your everyday life. From hitting Ctrl+F and searching a page for a word online to making sure this paper isn't plagiarized. These algorithms generally have a large display of text and must search through this display looking for matches to a given pattern. A very simple but ineffective way to do this is to go through extracting parts of the text of length m, the length of the pattern, and checking to see if it equals the pattern. This however would result in a horrible worst case runtime of O(nm). Two other string pattern matching algorithms are the Knuth-Morris-Pratt algorithm (KMP) and the Boyer-Moore algorithm. KMP provides a fast O(n) runtime at a cost of space. The Boyer-Moore algorithm has an ever worse runtime with a worst case of O(nm) and therefore I decided to create my own string pattern matching algorithm. My newest invention is a brilliantly developed Monte Carlo string pattern matching algorithm, called the Rabin-Karp algorithm. This algorithm uses the hashed code of the pattern to check against the hashed code of m length portions of the text, to see if the portion of the text matches the pattern. This means we need to make n (the length of the text) hashes, so at a constant time to hash, which we can implement using a rolling hash, I can provide an O(n) worst case run time for this algorithm. This algorithm provides a much faster runtime compared to the other 3 algorithms stated and does not require much memory overhead.

The Monte Carlo Rabin-Karp algorithm I have developed uses hashing as a way of speeding up checks for patterns in text and allowing for the fast runtime. The algorithm starts by taking in a string of both the pattern being searched for and a string of the text being searched through. The next trivial pieces of the algorithm calculate the length of the pattern and text, and hash the pattern to store its hash value to be compared with. The hashing function used in the beginning is a simple implementation of Horner's method. By taking advantage of a character's ASCII value, we can create a hash value just like any other number. We start by taking the "ones" place of the string and multiplying it by the size of

the alphabet, R, raised to the 0 power. We then add the value of the "tens" place character times R raised to the 1st power. This pattern repeats taking each character in the string, multiplying it by R raised to continually increasing powers and then adding to the total value. Once this number is received, to hash the value we modulus this value with a large enough prime number to reduce the amount of collisions possible created by this hashing function. For example to hash the pattern "abcd," it would look something like h = (97(ascii value of a) * 256(extended ascii table size)^3 + 98*256^2 + 99*256^1 + 100*256^0) % Q(large prime number). Determining the length of the pattern and text, and hashing the pattern can all be done in constant time and therefore don't asymptotically impact the overall runtime of this algorithm. The search then begins by extracting the first portion of the total text that is of m length, and comparing it to the hashed value of the pattern. If these two values match, the pattern has been found at the beginning of the text and the function can return a 1, meaning found at the first index. From this point we could continually extract m length portions of the text starting at the previous index plus 1 and continue to hash and check for each extraction. This however will be detrimental to our overall runtime as each hash of the extraction is done in O(m) time and thus the algorithm would be decreased to a O(nm) runtime which we simply can not have. The algorithm I have created incorporates a neat replacement for the hash function and instead uses a rolling hash. This is very easily done once the first extraction has been hashed. Instead of moving over 1 index and rehashing the whole string again, I can simply manipulate the previous equation to return a new hash value much faster. This can be done by subtracting out the first value in the hash, 97*256^3 in the example above, multiplying the equation by R and then adding the ASCII value of the new character. For example if "abcd" is read in first, the next pattern read in would be "bcde," then all you have to do is subtract out the value for a, multiply the equation by R so that b can be multiplied by R^3 now, and then technically adding the ASCII value of e plus R^0, which is just the ASCII value of e. This rolling hash function allows for a constant time to hash instead of the O(m) time stated previously. This move over 1 in the text, perform the rolling hash to acquire the new hash value of the extracted text, and

check with patterns hashed value is completed continuously until the end of the text is reached. This results in a beautiful runtime of only O(n) and requires only space for the array that backs the hash table used to store the hashed value for the pattern.

The two other string pattern matching algorithms stated before were KMP and the Boyer-Moore algorithm. KMP uses a deterministic finite state automata to store information about the pattern that can be used while traversing the text. It does this by using states to determine how much of the pattern has been matched and when a match is found. When each character from the text is read, it is sent into the dfa along with the current state, and then the new state is outputted. If the final state is outputted from the dfa, we can reason that a match has been found in the text. This allows us to avoid checking the same character in the text twice and gives it a runtime of O(n) as it simply traverses the text once. The downside of this algorithm is the space complexity. The DFA can be stored as a two dimensional array containing with a row size of R, and column size m. For each character in the pattern, a value is stored in each column to symbolize what state the DFA should change to based on what character is read and what the current state is. This allows the DFA to know how much of a string has been matched and when a pattern has been found. The space complexity required here is O(mR) and is thus a downside of this algorithm compared to Rubin-Karp. The Boyer Moore algorithm uses a very different approach and begins by comparing extractions of the text starting with their last letters. This allows the algorithm to skip ahead m positions when no match is found. For example if the pattern were “abcd” and the beginning of the text was “abce,” once e and d were shown to not be equal, the next comparison can be made using the string of length m that starts right after the e in this text. This would allow for a best case runtime of O(n/m) but also comes with the assumption that with a large alphabet, you will often come across characters not in the pattern a lot. On the other hand the worst case runtime for this algorithm would be O(nm) which is the same as brute force, which was stated earlier. While Boyer-Moore does not have any space concerns, the fact that its runtimes vary so much is a deterrent and a reason to choose Rabin-Karps O(n) time over it. KMP has the same runtime as Rabin-Karp but its

high space complexity makes it a less efficient string pattern matching algorithm. Boyer Moore also has a decent average run time, but its worst case of O(nm) makes it undesirable.

The Rabin-Karp algorithm uses a rolling hash to simply read through a piece of text once and efficiently check for pattern matches. The approach uses Horner's method to give a value to a string of characters and then we simply modulus that value to give it a unique hash code. This code is then created for each extraction of length m of the text and checked with the initial value to see if they are equal. This allows for a fast runtime of O(n) and only a space complexity equal to the size of the array backing the hash table. This allows for a more efficient way to check for pattern matches compared to KMP and Boyer Moore. On the other hand this Monte Carlo implementation of Rabin-Karp could also use some work. The reason this implementation is Monte Carlo is because it only checks the hashed value of the two patterns, which while unlikely could result in a match even when the patterns are not the same. If I were to implement a Las Vegas approach, I would need to do a character by character check of the extraction of the text and the pattern to make sure they are a true match. This would in turn decrease the runtime to an atrocious O(nm), which is the same as brute force. The implementation of the Monte Carlo Rabin Karp allows for a much faster completion time at the risk of the match being incorrect. Therefore one improvement that could be implemented in the future would be a faster way to check if a match is really a match instead of having to do a character by character check. This could be done by trying to eliminate all collisions that would occur within the hash table because then a match would be guaranteed when the hash values are equal. Another improvement could be made to try and make Rabin-Karp more like Boyer Moore in the fact that it can eliminate whole portions of text at once. If Rabin-Karp could skip over portions of the text, it would allow for fewer hashes and an improved average run time. Overall my invention is very useful in todays society as many people use string pattern matching algorithms all the time in their everyday life and should be adopted by everyone.