Imperial College London

Mastermind Report

EEE1-06 | Software Engineering 1: Introduction to Computing

Arya, Arie
3-21-2019

Table of Contents

Introduction

The algorithms discussed in this report are based principally from a combination of Donald E. Knuth's *Five-Guess* algorithm (1977) and Peter F. Swaszek's *the Mastermind Novice* algorithm (1999-2000). However, due to the O(n) linear time complexity of these algorithms, as well as expansive memory usage for larger combinations of codelengths and symbols, variants of these algorithms are used. Fundamentally, the minimax strategy comprising these two algorithms have been disregarded due to its $O(b^m)$ time complexity [1], where b is the number of possible moves at each point and m is the maximum depth of the tree investigated. For large combinations, it will not be possible to traverse the depth of the tree within the 10 seconds specified timeframe, hence the minimax strategy is not utilized in this code.

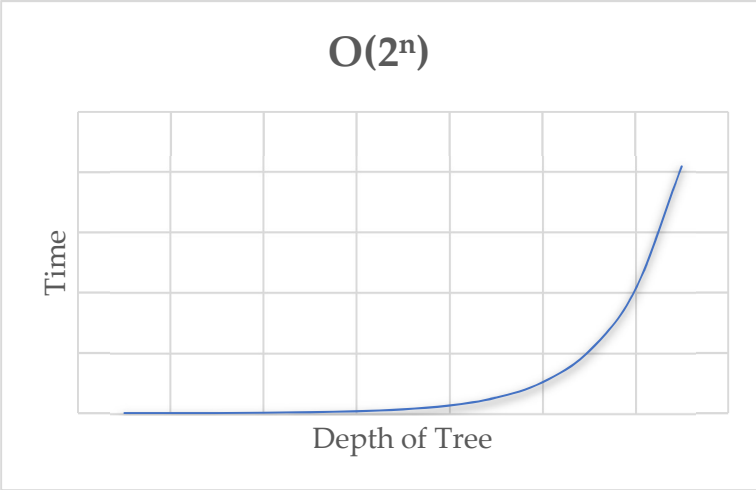


Figure 1: Time Complexity of $O(2^n)$. $O(b^m)$ of the minimax strategy would generally have an even steeper time-complexity curve.

The algorithms used in this code are set under three main categories: Knuth, Swazsek, and the Split algorithm. Note again that Knuth and Swaszek for the purposes of this code does not employ the minimax strategy; though traditionally, minimax forms an essential part of these two algorithms.

The attempt efficiency of these three algorithms go by the order Knuth, Swaszek, and Split. However, the runtime efficiency runs the opposite way; that is, from Split, to Swaszek, to Knuth. For this reason, Knuth is used wherever possible for lower combinations, before switching to Swaszek and eventually Split for larger combinations of codelengths and symbols. The table below shows the regions occupied by the three algorithms in the 15 by 15 space.

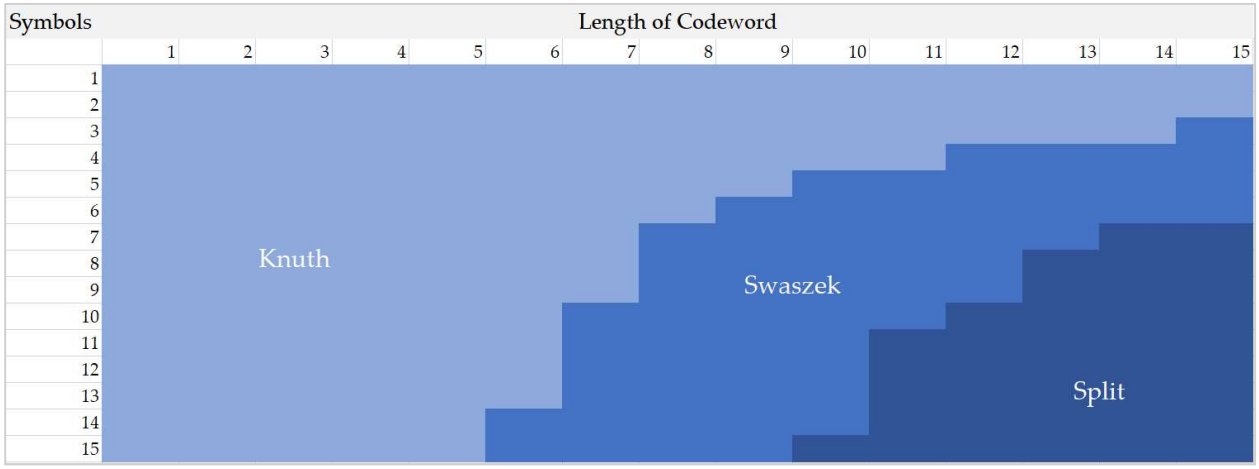


Figure 2: Regions occupied by Knuth, Swazsek, and Split in the 15 by 15 space

Beyond this space - for much larger combinations - the Split algorithm will be most predominantly used; though the curved pattern shown above still remains. The criteria for switching to different algorithms will be explained in detail in its individual sections; but beforehand, a brief explanation of the mandatory give feedback function is provided, as it in itself forms an imperative part of each algorithm, and more crucially, the indexing system.

Give Feedback Function

The give feedback function has the purpose of calculating the number of black and white hits for a given codeword and attempt. It is within the struct mm_code_maker, and is called after every formal attempt made by the code. The feedback obtained from this function will then be used by the algorithm to decide the next best attempt.

The method to perform this function is described in Donald E. Knuth's *The Computer as a Mastermind* [2]. Firstly, the total number of black hits is searched for. Let a_i and b_i be the i-th symbol of the codeword and test pattern respectively; and let num and length represent the number of symbols used and length of the codeword. Traversing the length of the code, the number of occurrences where $a_i = b_i$ gives the total number of black hits n_b. Now, let m_i and m'_i be the number of times a symbol i appears in the codeword and the test pattern respectively. The total number of hits can be found by

$$\min(m_1, m'_1) + \min(m_2, m'_2) + \ldots + \min(m_{length}, m'_{length})$$

For example, let the codeword be 114253 and the test pattern be 123412. The total number of hits n_t (both black and white) is given by

$$\min(0, 0) + \min(2, 2) + \min(1, 2) + \min(1, 1) + \min(1, 1) + \min(1, 0) = 5 \text{ hits}$$

Symbol:	0	1	2	3	4	5

Indeed, by inspection we have 1 black hit and 4 white hits, giving a total of 5 hits. The white hits is obtained by subtracting the total hits n_t by the number of black hits n_b, giving us

$$n_w = n_t - n_b$$

The function to implement this is relatively simple and consists of two for loops. The first traverses the attempt and sequence vectors to search for black hits. For total hits, two vectors "attemptchecker" and "sequencechecker" are introduced with size num and initialized to zero. In traversing the codeword and test code from index i = 0 to i = length – 1 (because it is 0 indexed), given that the i-th symbol of the codeword and test code is x_i and y_i respectively, the x_i-th element of sequencechecker and y_i-th element of attemptchecker is incremented by 1.

```
41        for(int i = 0; i < length; i++){
42          if(sequence[i] == attempt[i]){
43            black_hits++; // black hits determined here
44          }
45          sequencechecker[sequence[i]]++;
46          attemptchecker[attempt[i]]++;
47        }
```

A minimum comparison between these two vectors is then made at each index, and summing these minimum values gives the total hits, stored in the variable "whitetemp".

```
48        for(int i = 0; i < num; i++){
49          if(sequencechecker[i] <= attemptchecker[i]){
50            whitetemp = whitetemp + sequencechecker[i];
51          }
52          else{
53            whitetemp = whitetemp + attemptchecker[i];
54          }
55        }
```

Subtracting this total hits value by the black hits gives the white hits, hence completing the give feedback function.

```
56        white_hits = whitetemp - black_hits;
```

Knuth Algorithm

The Knuth Algorithm for a 6 by 4 system traditionally begins with a predetermined test code of 1122, as it is proven that this code eliminates the maximum number of code possibilities. However, because of the varying code and symbol lengths, this will not be implemented. The Knuth Algorithm is based on creating a vector of all possible codewords and using the white and black hits from the give feedback function to eliminate remaining possible codewords from this vector, before finally reaching the real codeword. Firstly, start off with the vector of all possible codewords; and since the codeword itself is a vector, a vector of vectors is used.

```
75    std::vector<std::vector<int>> possiblevalues;
```

Fundamentally, the general process is as such. Firstly, fill the vector of possible codewords with all possible combinations of code (00000…000 → nnnnn…nnn). Next, send out a random test code and receive a feedback of black hits (n_b) and white hits (n_w). Then, compare all codes in this vector with the test code with an internal give feedback function. If the comparison does not give exactly n_b and n_w, it cannot be the real code, so it is removed from the vector. Next, randomly select a code from the remaining possible codes as a test code and receive feedback. Repeat the process until the real code is obtained. Traditionally, Knuth's algorithm would utilize the minimax strategy to select test codes that would eliminate the maximum number of possible codes, however, as mentioned previously, this will not be implemented in this code.

It is perhaps easiest to explain this using an example. Say, for a 6 by 6 system, the codeword is 535213. First, a vector of all possible codewords is made, containing vectors 000000 to 555555. Now a random test code is selected, say 533324. The feedback gives 2 black hits and 2 white hits. Next, traverse the whole list of possible codewords and eliminate codes where it is not consistent with this feedback. For example, comparing 111111 to 533324, this scores 0 black hits and 0 white hits; it is not consistent (with 2 black hits and 2 white hits), so this is removed from the vector. If it is consistent, for example 333500 with 533324 gives 2 black hits and 2 white hits, we keep this. After repeating a number of times, the real code will eventually be obtained.

However, this whole process can be heavily time-optimized. This is especially important as it requires to traverse a vector of size num^{length} every initial stage, which reaches over 16 million elements in an 8 by 8 system. To optimize this, first begin by looking at the indexing system.

Introduction to the Indexing System

The indexing system is a method of bypassing redundancies, which can be quite significant as it reaches higher values. For the general Knuth implementation, it allows it to solve a 6 by 15 system barely under 10s (but because of the risk of it reaching over 10s, I use the Swaszek algorithm to perform a 6 by 15), whereas a normal Knuth implementation could only reach up to

5 by 15. This may not seem like a lot, but 5 by 15 and 6 by 15 accounts for a difference of more than 10 million vector elements. The indexing system, as shall be seen, forms an imperative and an even more important part of the Swaszek Algorithm, which will be explained in the next section.

The general Knuth process starts from 00000…000 and increments by 1 each time to traverse the whole possible code.

00000…000 → 00000…001 → 0000…002 → … → nnnnn…nnn

The indexing system allows an index to be selected from the code being examined, and this index is incremented (instead of incrementing the last index all the time as shown above). This could potentially eliminate large numbers of combinations. The example below shows the indexing system in action (in a 9 by 9 system), with the up arrow pointing at the index.

000000000 → 010000000
↑

This one index from the example above eliminates a total of 9^7 = 4.7 million vectors; that is,

00xxxxxxx

Where x can be any number from 0 to 9 (in this example). The number of different combinations of these x values corresponds to 9^7 combinations. This example only shows a single index increment occurrence. In a single traversal of the possible codewords vector, multiple indexes can be expected to be incremented, meaning many redundant vectors will be bypassed.

The general index incrementing process is explained, but how are these indexes determined in the first place? There are two ways of determining the index, one is by examining the black hits at every index, and the other by examining the white hits at every index. The important general question being asked is "does it matter what the values in the next indexes are for deciding whether the code is valid or not?" If not, then why bother? That will therefore be the index to be incremented.

For example (in an 8 by 8 system), say the codeword is 42142160 and our first test code is 33300000, giving 1 black hit and 0 white hits. We increment one by one and reach 33000000. Here, we notice that the first two indexes (33) of the test code already returns 2 blacks (which is inconsistent to the 1 black received from the feedback earlier). From here, we realize that whatever the value that follows, e.g. (33)402415, (33)213525, (33)002345, or (33)xxxxxx (x can be any number between 0 and 7 in this example) must all be wrong, so why bother going through all of that? Simply increment the identified index to bypass 8^6 = 262,000 combinations. We therefore increment the second index and get 34000000, and move on. The code to perform this operation will be explained in the next subsection.

Implementation of the Time-Optimized Knuth Algorithm

Firstly, the condition to enter the Knuth Algorithm is to have $num^{length} < 5{,}000{,}000$; that is, the maximum number of vector elements of all possible codewords must be less than 5 million. This will set a bool called "knuthmode" to true, meaning throughout the code, only the Knuth implementation will be considered.

```
98        if(std::pow(num, length) < 5000000){
99          knuthmode = true;
100         std::vector<int> lengthinitializertemp(length, 0);
101         std::vector<int> numinitializertemp(num, 0);
102         freevector = lengthinitializertemp;
103         numinitializer = numinitializertemp;
104       }
```

This stage will also introduce two vectors "freevector" and "numinitializer". "freevector" is a free variable vector which maps all the vectors from 00000…000 to nnnnn…nnn by incrementing its indexes. "numinitializer" will be used to initialize or reset certain vectors after it is used in certain operations.

Now, instead of immediately creating a vector of all possible codewords, a random attempt is first shot out.

```
165       if(initialattempt){
166         for(int i = 0; i < length; i++){
167             attempt.push_back(randn(num));
168         }
169         initialattempt = false;
170       }
```

The feedback from this random attempt will be used to create a smaller vector using the more efficient indexing system. As mentioned previously, instead of traversing the list of all possible codewords to see if each code is consistent with the feedback, we can use the indexing system to remove redundancies.

This step utilizes a function called "nextinitial", which increments "freevector" by the correct index until a possible code is reached.

```
268           while(nextinitial(black_hits, white_hits, attempt)){
269             possiblevalues.push_back(freevector);
270           }
```

As shown above, when "freevector" reaches a code that is consistent with the feedback (i.e. returns the same number of black and white hits), we store this code into the list of possible codewords ("possiblevalues").

Incrementing the "freevector" efficiently, however, can be quite challenging. Within the "nextinitial" function, there is another function called "nextinitialcompare", which is effectively the give feedback function, but returns the index of the freevector that should be incremented.

The method of identifying the index to increment is done by observing both the current black and white hits.

```
446         for(int i = 0; i < length; i++){
447           lengthfromend--;
448           if(freevector[i] == attempt[i]){
449             currentblackpegs++;
450           }
451           if(currentblackpegs > black_hits || currentblackpegs + lengthfromend < black_hits){
452             index = i;
453             return false;
454           }
```

For example, if the current black hits measured (in the for loop) is higher than the black hits given from the feedback, we can simply set the index as the value of i and increment that index. This is because whatever the value is that follows after this index, it will not matter, and will always be wrong (recall the earlier example from *Introduction to the Indexing System*). For example, let the codeword be 100134, and we have the test code 001351, giving us 1 black hit.

Test code: 0 | 01351

Free vector: 0 | 00000

1 black hit

Test code: 00 | 1351

Free vector: 00 | 0000

2 black hits

Dotted line represents the for loop comparing one layer at a time (and moves to the right as we move on to the next index)

Because 2 black hits > 1 black hit (from feedback), set this as the index to increment.

000000 → 010000
↑

The other condition for black hits, as shown in the above code snippet, is when the current black hits plus the remaining length of symbols in the code, equates to less than the real black hits. This means that, given that all of the next remaining symbols all score black hits, it will still not be enough to satisfy the black hits from the feedback, and is therefore redundant to continue checking the remaining combinations.

The other set of conditions for the indexing process come from looking at the white hits.

```
468         if(whitetemp - currentblackpegs > white_hits || whitetemp - currentblackpegs + lengthfromend*2 < white_hits){
469           index = i;
470           return false;
471         }
```

The first white hit condition is rather self-explanatory. If the current white hits is greater than the white hits from the feedback, the remaining combination must be incorrect, so we increment this index (similar to the first black hit condition).

The second condition is more complicated (especially the *2 term). Here, we follow a similar approach with the second black hit condition. If current white hits plus the remaining length of symbols in the code is less than the real white hits, it will be wrong, so we increment this index. But what about the *2? This comes from the rare occasion where as we move to the next layer of the code, we get an increase of 2 white hits. For example, consider the test code 00532 and 45423, and look at the transition between the fourth and fifth layer:

Test code: 0053 | 2

Free vector: 4542 | 3

1 white hit

Test code: 00532 |

Free vector: 45423 |

3 white hits

As can be seen, there is an increase in 2 white hits from a single transition in layer, hence the *2 term is necessary when identifying the index.

When an index is found, the function “nextinitialcompare” will return false, and will continue inside the while loop within the “nextinitial” function (as shown below).

```
418         while(nextinitialcompare(attempt, index, black_hits, white_hits) == false){
419           freevector[index]++;
420           if(freevector[index] == num){
421             while(freevector[index] == num && index > 0){
422               freevector[index] = 0;
423               freevector[index - 1]++;
424               index--; // move to previous index
425             }
426             if(index == 0 && freevector[index] == num){
427               freevectorchecker = numinitializer;
428               attemptchecker = numinitializer;
429               return false; // this indicates end of possibilities
430             }
431           }
432           index = 0;
433           freevectorchecker = numinitializer;
434           attemptchecker = numinitializer;
435         }
436         freevectorchecker = numinitializer;
437         attemptchecker = numinitializer;
438         return true;
439       }
```

Firstly, the parts with the "numinitializer" can be ignored, this is to keep resetting the vectors after it is used in the "nextinitialcompare" function. Although it is possible to define the vectors "freevectorchecker" and "attemptchecker" within the scope of "nextinitialcompare" (so that resetting is not necessary), I have decided to make them a member data of the mm_solver as these vectors are used in all three algorithms (so they could all use the same name - as they serve the same purpose).

The next steps can be summarized as such. When an index is found, we increment the freevector in that particular index (line 419). If that particular index is overflowing, i.e. if it increments to a number greater than the symbols provided (num), we simply increment the previous index by 1, and set the current index to 0 (line 421 to 425). For example, for a 5 by 5 system

00040 → 00100
↑

or 14440 → 20000
↑

This process is repeated until "nextinitialcompare" returns true, which occurs when the freevector resembles a possible code. This will break the while loop, and, as we have seen before, the code of the freevector will be added to the list of possible codes.

```
268           while(nextinitial(black_hits, white_hits, attempt)){
269             possiblevalues.push_back(freevector);
270           }
```

We increment the free vector by 1 (line 405 to 412), and repeat this process over and over until the whole list is exhausted. This is identified when the index is pointing at 0 (lowest index), and the value of the free vector at that index is equal to num. For example, in a 7 by 7 system

6231620 → return false
↑

As can be seen above, the index points at index 0, and that particular value (at index 0) is equal to num (the highest symbol number). Since combinations containing 7 is not possible, we return false, indicating that we have traversed the entire list, and all possible codes have been found and stored in "possiblevalues". This will allow the code to exit the while loop above. This concludes the indexing process for the Knuth Algorithm – it is used only for generating the initial list of possible codewords, but this process will be re-introduced in the Swaszek Algorithm in more depth.

After this initial list is created, the next steps are relatively straightforward. We choose a random possible code from the list "possiblevalues" as an attempt.

```
172            int random = randn(possiblevalues.size());
173            attempt = possiblevalues[random];
```

Next, we use the feedback from this attempt to reduce the further the list of possible codewords. This can be explained by first looking at the code snippet below.

```
275           for(int i = 0; i < possiblevalues.size(); i++){
276             if(codecomparator(attempt, black_hits, white_hits, i)){
277               newpossiblevalues.push_back(possiblevalues[i]);
278               attemptchecker = numinitializer;
279             }
280             attemptchecker = numinitializer;
281           }
282           possiblevalues = newpossiblevalues;
```

The function "codecomparator" is, again, very similar to the give feedback function, and returns true or false depending on whether or not the code comparison gives a consistent result with the feedback. It returns true if it is consistent, and false otherwise.

Instead of using the vector.erase() function, which is notably time consuming, this method would create a new list of possible codewords called "newpossiblevalues", and will fill this vector with the remaining codes (from "possiblevalues") that satisfy the condition by the new feedback. After filling this new vector, we simply set "possiblevalues" to "newpossiblevalues", removing the need to use vector.erase(). We repeat this process, which will reduce the number of possible codewords stored until the real code is finally found.

Recall again the region of space with which this algorithm operates in Figure 2. Below is shown the average attempt and time of this algorithm for all combinations in which it operates.

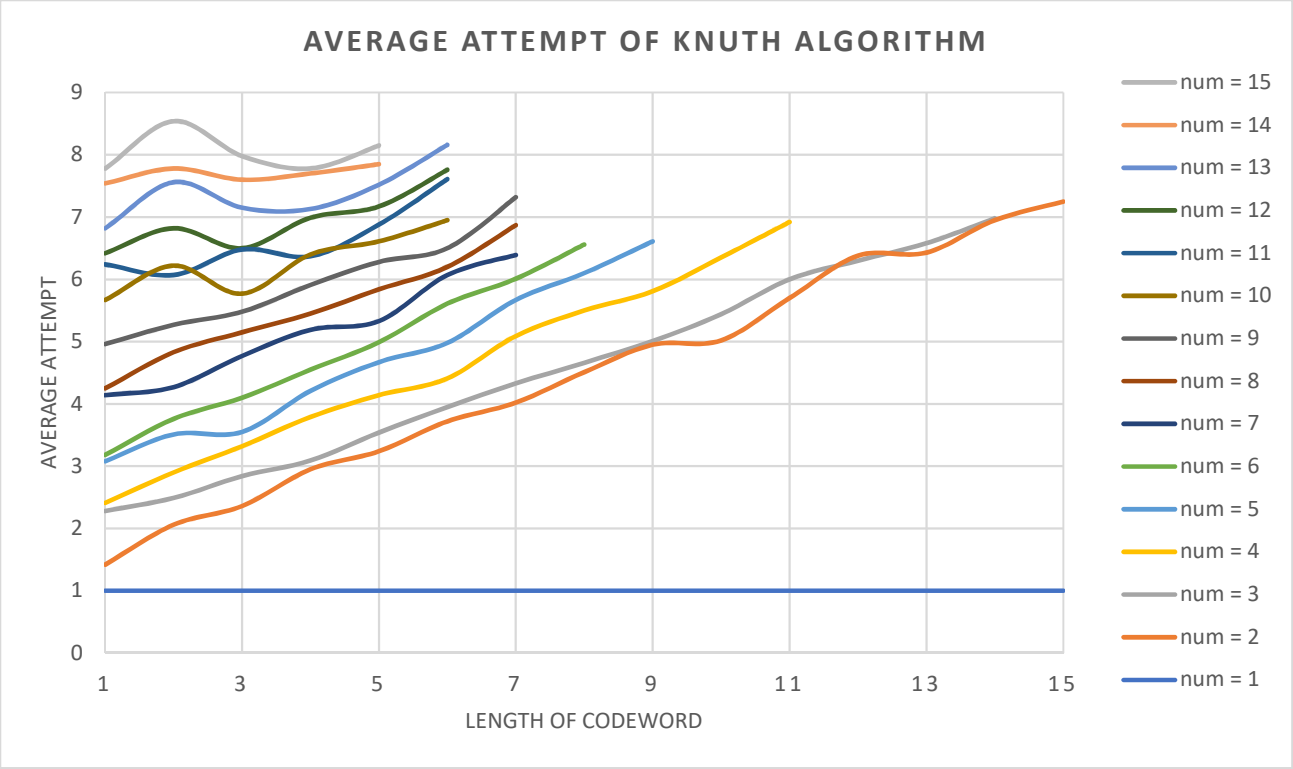


Figure 3: Average Attempt of the Knuth Algorithm for the space it covers.

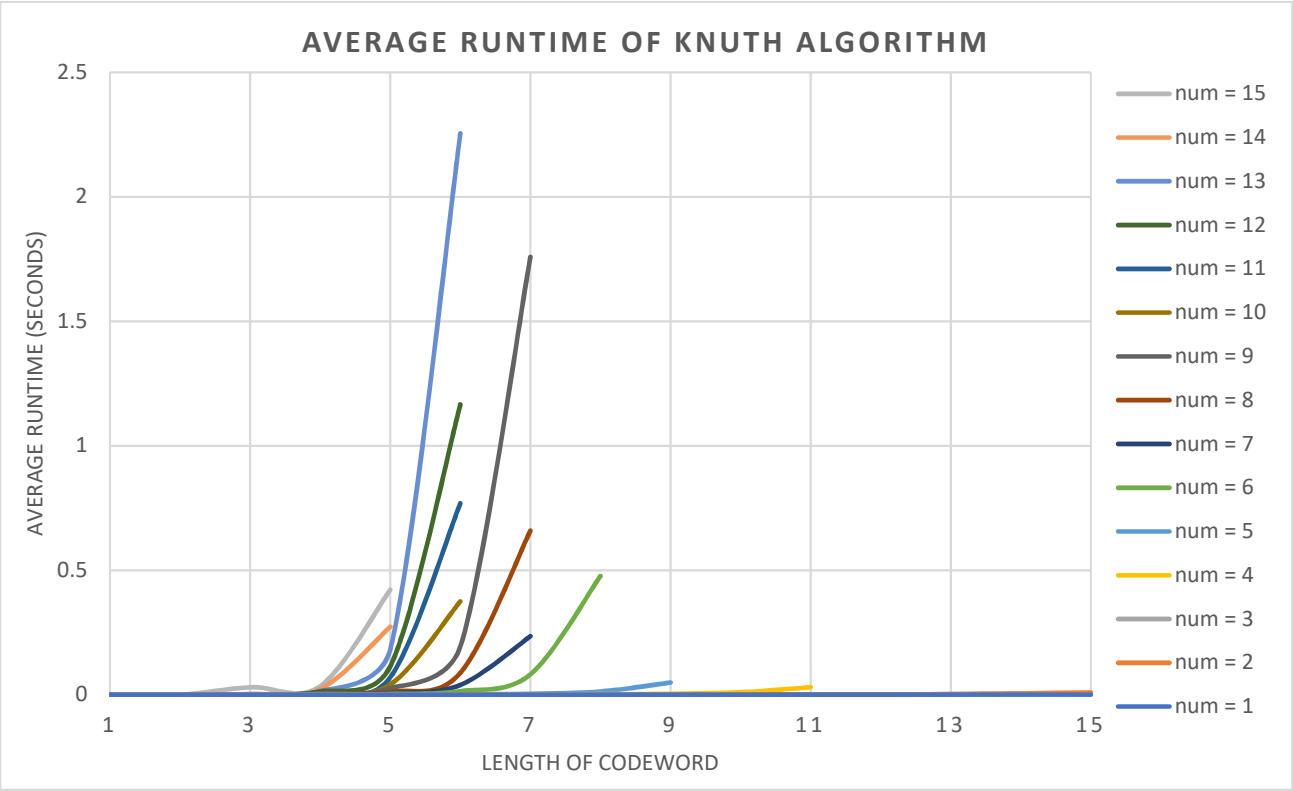


Figure 4: Average runtime of Knuth Algorithm. Beyond this, it will switch to the Swaszek Algorithm.

Modified Swaszek Algorithm

The Swaszek Algorithm [3] is very similar to the Knuth Algorithm (without minimax). The principles of Knuth Algorithm, as we have seen, is to store all possible codes, pick a random code from the list, and reduce that list with the new feedback, and repeat until the real code is obtained. The Modified Swaszek Algorithm is very similar, but instead of storing all possible codes, it always shoots out the first code it deems consistent (with the feedback) as the attempt. It assumes that the operation of storing all possible codes and picking a random one out of the list (not with minimax) is just as attempt efficient as always picking the first possible code as an attempt. In fact, the result is indeed very close. Swaszek scores an average attempt of 4.758 for a 4 by 6, compared to 4.638 with Knuth (without minimax) [4]. However, as will be mentioned in the implementation, the Swaszek Algorithm has a considerably lower runtime, and is therefore used for higher combinations.

Knuth Process:

List of Possible Codes:

00412351538
00513058613
00023521560 → randomly select → attempt = 00513058613
02340151060
02134032501

Swaszek Process:

List of Possible Codes:

00412351538
00513058613
00023521560 → always first code → attempt = 00412351538
02340151060
02134032501

Additionally, the implementation of this Modified Swaszek Algorithm will not be storing a list of all possible codes (as it always picks the first possible code) and is hence more memory efficient than its Knuth counterpart.

Implementation of the Modified Swaszek Algorithm

The general process for this algorithm can be summarized by the diagram below.

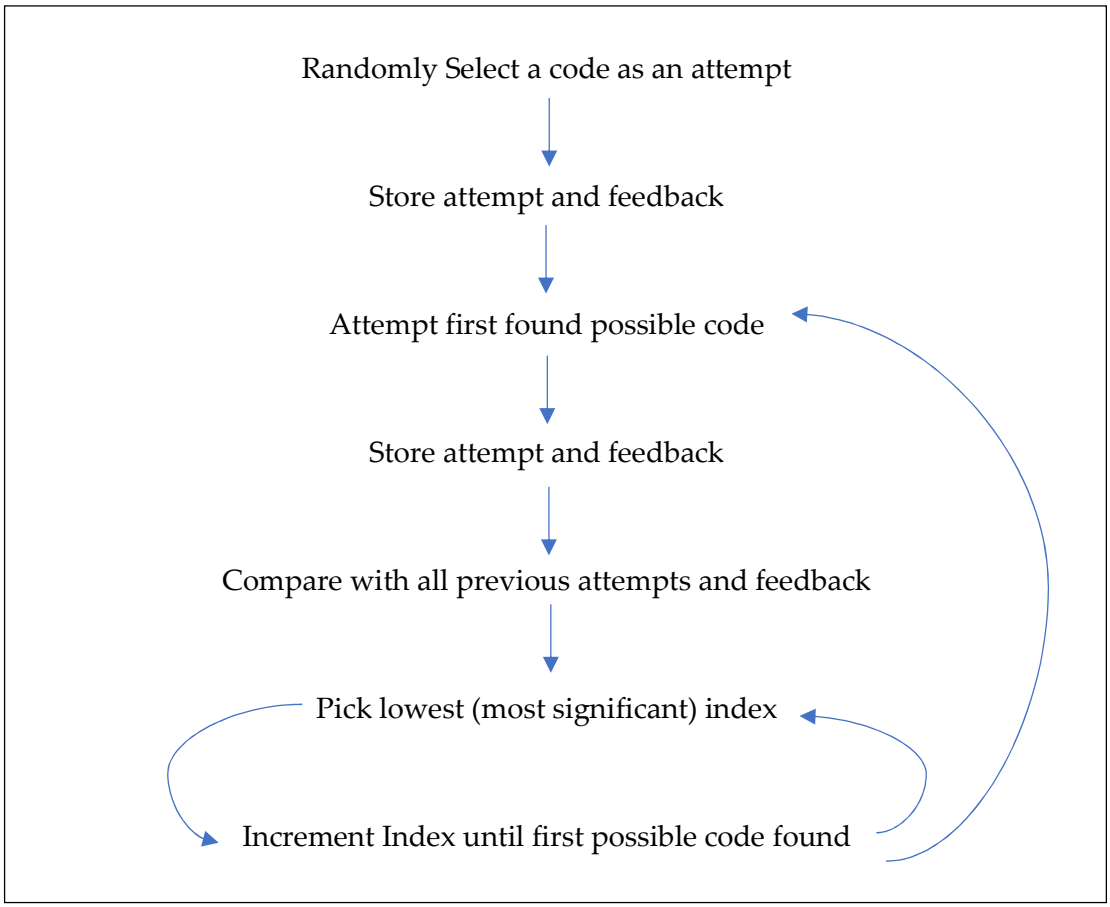


Figure 5: General process of the Swaszek Algorithm

This process will eventually reach the real codeword with an attempt efficiency close to that of Knuth's Algorithm.

Similar to Knuth's Algorithm, it starts off by sending a random code as an attempt.

```
133         if(initialattempt){
134           for(int i = 0; i < length; i++){
135               attempt.push_back(randn(num));
136           }
137           initialattempt = false;
138         }
```

All of the attempt codes and their corresponding feedback will be stored in the same index in two separate vectors "codeusedlist" and "codeusedlistBW", as shown in the snippet below.

```
235             std::vector<int> BWtemp;
236             BWtemp.push_back(black_hits);
237             BWtemp.push_back(white_hits);
238             codeusedlist.push_back(attempt);
239             codeusedlistBW.push_back(BWtemp);
```

After storing the attempt and feedback, the algorithm will look for the next possible code, and does this by calling the function "nextstep" (line 242). This function will map the free vector into the next possible code through the indexing process. The indexing process will be similar to that of the Knuth Algorithm, but instead of comparing the free vector to one (most recent) test code and finding the index, we will compare it to all previously used test codes, and find the lowest (most significant) index from all of the comparison, and increment this.

This will remove considerable amounts of redundancies, which makes the process much more time efficient. This algorithm, for example, could compute the codeword of a length 9 num 15 game in under 5 seconds (consistently) with an average of 20.53, whereas Knuth's Algorithm could only go up to a length 5 and num 15 game within the 10s runtime. To see the scale of the difference, a 9 by 15 game has $15^9 = 38.4$ billion possible code combinations, compared to only $15^5 = 760{,}000$ possible code combinations for a 5 by 15 game.

The process of the function "nextstep" is actually rather identical to that of the Knuth Algorithm. This function will call upon another function called "nextstepcompare", which returns true if the free vector represents a possible code, and false – along with the index – if the free vector is not a possible code. This index will then be incremented in exactly the same way as with Knuth's.

```
301         while(nextstepcompare(index, indexmarker) == false){
302           counter++;
303           freevector[index]++;
304           if(freevector[index] == num){
305             while(freevector[index] == num && index > 0){
306               freevector[index] = 0;
307               freevector[index - 1]++;
308               index--;
309             }
310             if(index == 0 && freevector[index] == num){
311               freevectorchecker = numinitializer;
312               attemptchecker = numinitializer;
313               return false; // indicates end of possible combinations
314             }
315           }
```

"nextstepcompare" works rather the same way as the function "nextinitialcompare", and has the same purpose, which have been explained under the Knuth Algorithm section. "nextstepcompare" differs in that instead of comparing a single test code with the free vector, it compares all used attempts with the free vector. Hence, we use the for loop shown below.

```
330         for(int z = 0; z < codeusedlist.size(); z++){
```

To compare with all previous test codes used, we simply vary the attempt and feedback (black and white hits) according to all the test codes and feedback stored in "codeusedlist" and "codeusedlistBW" using the index z in the for loop.

```
343         std::vector<int> attempt = codeusedlist[z];
344         int black_hits = codeusedlistBW[z][0];
345         if(swaszekmode || usewhite){
346           white_hits = codeusedlistBW[z][1];
347         }
```

The if condition can be ignored for now (it will definitely enter in this case as "swaszekmode" is true). This will be important in the Split algorithm.

The next part of this function will be to find the index of the free vector from each comparison. This has been explained quite rigorously under the *Implementation of the Time-Optimized Knuth Algorithm* section. From all of the comparison, the final index chosen will be the one with the smallest value, i.e. the most significant index. As shown below, the index value will only be updated if the new index (i) is less than the current index.

```
354             if(i < index){
355               index = i;
356             }
```

This lowest index will be the index incremented in the free vector. For example, comparing with 4 previous test codes, we receive these indexes:

02135016847
↑ ↑ ↑ ↑

We simply increment the lowest (most significant) index.

02135016847 → 03135016847
↑

In fact, another important step after incrementing the selected index is to initialize the values following the index to 0. This is necessary so that we do not skip potential possible codes. So, from the example above, we have:

02135016847 → 03000000000
↑

The code to do this is quite self-explanatory and is shown in the snippet below.

```
316         for(int i = index + 1; i < intermediatelength; i++){
317           freevector[i] = 0;
318         }
```

This indexing process will repeat until a possible code is found.

When a possible code is found, "nextstepcompare" will return true, and the code will exit the while loop (shown above). We then equate the vector "nextattempt" to this free vector (this will be the next attempt).

```
258                 nextattempt = freevector;
```

Upon calling "create_attempt" in the main, it will simply equate the "attempt" vector to this "nextattempt".

```
140             attempt = nextattempt;
```

Both the attempt and feedback will be stored, and the process will be repeated until the real code is found.

An Additional Step

Occasionally, the algorithm will be stuck trying to find a specific value deep within the indexes of the free vector. To overcome this, we set a counter that counts the number of times the indexes are incremented without finding the next possible code. If this counter reaches above a certain value, we can safely assume that it is stuck trying to find this value. To exit this process, we simply shoot out whatever the free vector is as an attempt. This will almost always work, unless the value being searched for is either the real code, or something very close to the real code.

```
382       if(index < indexmarker){ // enter if free vector is not a possible code
383         if(counter > value){
384           if(value > 1500){
385             value = value - 150;
386           }
387           indexbound = index;
388           speedup = true;
389           return true; // return true to indicate possible code is found
390         }
391         return false;
392       }
```

Here, the bool "speedup" will unlock the process by which the free vector is taken and slightly modified before it is used as an attempt. "value" in this case is a predetermined number that, if the counter exceeds, will trigger this speed up process. The selection of this predetermined number will be explained briefly at the end of this section.

```
243             if(speedup){
244               std::vector<int> speeduptemp = freevector;
245               setlength = length;
246               if(!swaszekmode){
247                 setlength = segment;
248               }
249               int upperlimit = setlength - indexbound;
250               int lowerlimit = indexbound;
251               for(int i = 0; i < upperlimit; i++){
252                 speeduptemp[rand() % upperlimit + lowerlimit] = randn(num);
253               }
254               nextattempt = speeduptemp;
255               speedup = false;
256             }
```

Firstly, we equate a new vector "speeduptemp" to the free vector. Next, we will randomize the values of the remaining indexes. This will help more effectively identify the stuck value to leave the process. For example

Free vector: 023761|00000

Speeduptemp: 023761|04251 (randomized after the index)

Lastly, we simply equate the vector "nextattempt" to this modified free vector "speeduptemp", which will then be sent out as the next attempt after "create_attempt" is called. This will allow the program to exit the (stuck) process and continue with its operation.

The value with which if the counter exceeds this speed up process is triggered is set at 10,000.

```
60    int value = 10000; // predetermined, set by experiment
```

This value is chosen from experimentation, and seem to give a near optimal time-efficiency by compromising the attempt-efficiency. Essentially, if we increase this value, the attempt-efficiency will increase, but the time-efficiency drops (perhaps above 10s for some operations). If we reduce this value, the runtime will improve, but the attempt-efficiency will drop. The value 10,000 proves to give an optimal efficiency between the two.

This concludes the Modified Swaszek Algorithm. The charts below show the average attempt and average runtime for this algorithm in the space in which it operates (shown in Figure 2).

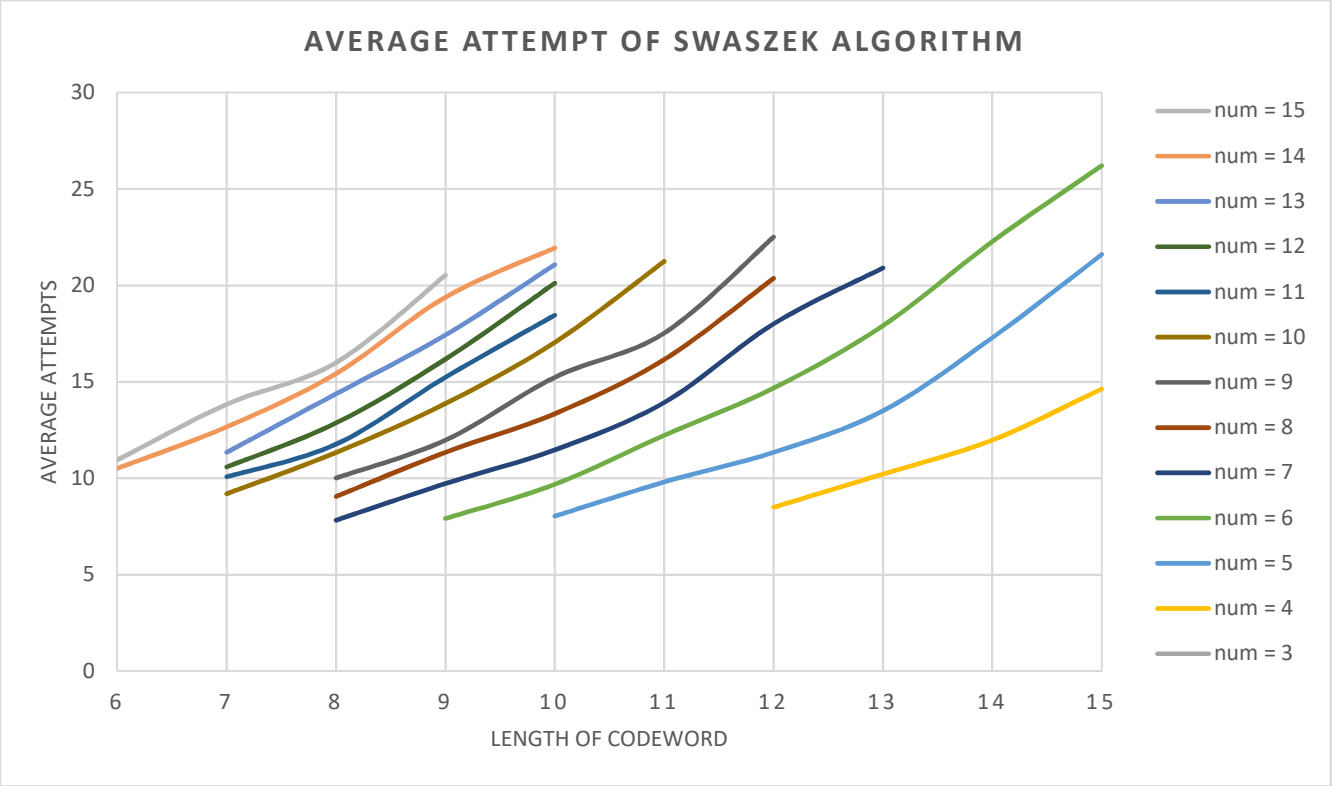


Figure 6: Average Attempts of Swaszek Algorithm

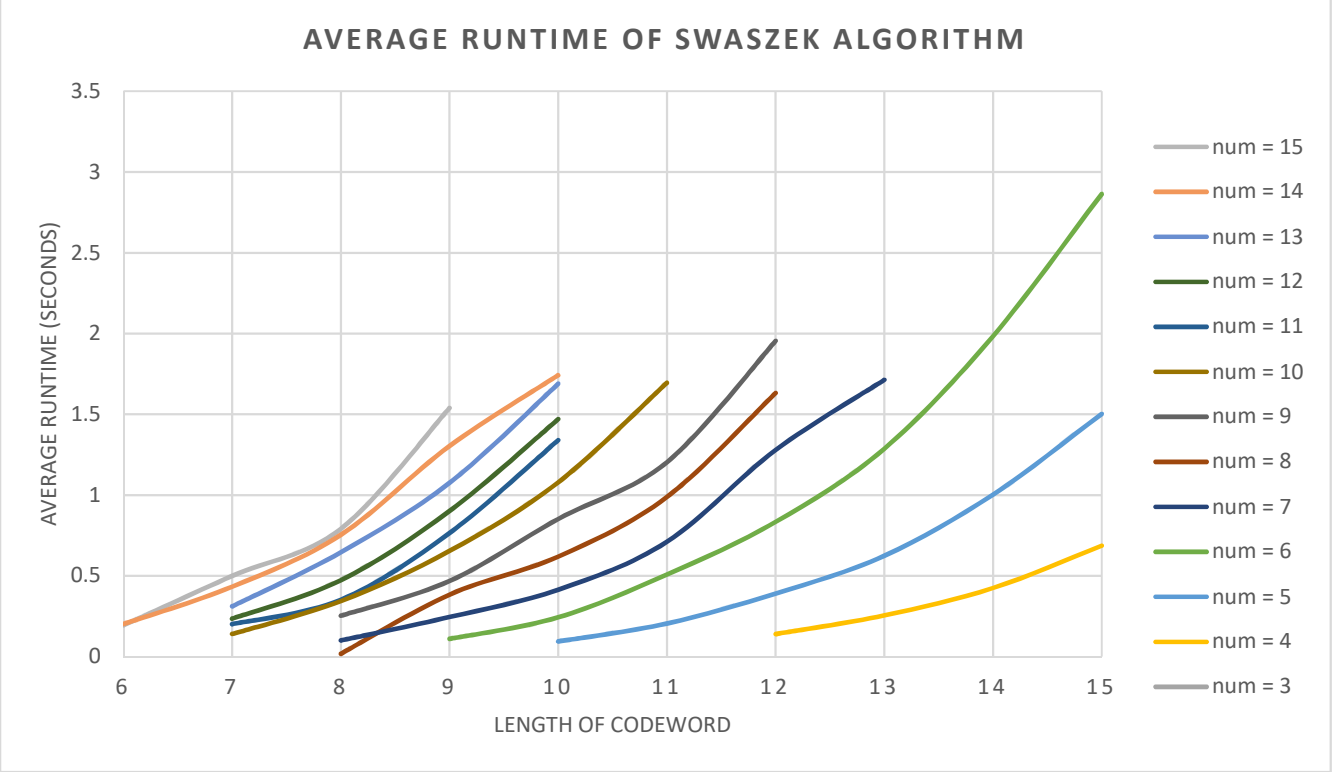


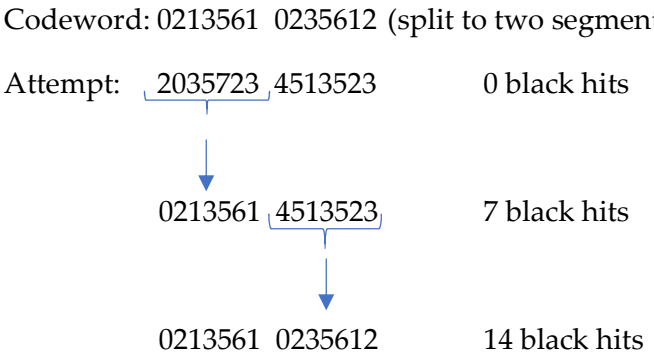
Figure 7: Average Runtime of Swaszek Algorithm. Beyond this, it will switch to the Split Algorithm.

As can be seen in the above average runtime graph for the Swaszek Algorithm, although it displays an exponential time complexity, it is much less extreme than the Knuth Algorithm.

Swaszek Split Algorithm

Although the Modified Swaszek Algorithm is in fact capable of computing codewords for combinations of 15 by 15 and beyond (usually within the runtime), it is often very unreliable and inconsistent. For this reason, another algorithm is used; the Swaszek Split Algorithm. Since this Algorithm is predominantly made of the Modified Swaszek Algorithm, most explanations in this section will only be for code parts unique to this algorithm.

This Algorithm, as its name suggests, is simply the Modified Swaszek Algorithm, but splits the codeword into sections (or segments). Specifically, given the condition to use this algorithm is met, it always splits the code into two sections, and performs Swaszek Algorithm on the two sections one by one.



This is where the Swaszek Algorithm really becomes advantageous, because where a Knuth Algorithm could at best only split a 15 by 15 code to three parts; 5 by 15, 5 by 15, and 5 by 15, the Swaszek Algorithm can split this only to two; 7 by 15 and 8 by 15. This considerably improves attempt-efficiency. For a 15 by 15, a Split algorithm with Knuth's would typically give an average of 42, whereas a split using Swaszek gives an average of 36.5.

The first step of the Split algorithm is to find a test code with exactly 0 black hits. This is important in order to distinguish which black hits come from which segment; mixing these up will give rise to various problems. This is done by simply sending random test codes until this code is found. This typically takes very few attempts, but the average attempt to find a code with 0 black hits increases as length increases and num decreases (this is most problematic at length 15 and num 7, when Swaszek algorithm switches to Split algorithm).

```
146         if(!foundzeroblack){
147           std::vector<int> randomguess;
148           for(int i = 0; i < length; i++){
149             randomguess.push_back(randn(num));
150           }
151           attempt = randomguess;
152         }
```

Once a code with 0 black hits is found, we can proceed with performing the Modified Swaszek Algorithm on the two segments. The two segments are first defined in the "init" member function.

```
109         segment1 = length/2;
110         segment2 = length - segment1;
111         segment = segment1; // initialize to first segment
112         pointer = 0; // start at 0
```

Through experimentation, I found that it is best to always split the segment into two equal halves (where possible); or, if it is an odd number, allow the second segment to have the bigger portion. For example, for a 15 by 15, we would split to: segment 1 = 7 by 15, segment 2 = 8 by 15. This is because the first segment will not consider white hits, whereas the second segment will (and will hence solve that 8 by 15 more efficiently).

The reason why the first segment will not be considering white hits is because that white hit may arise from the second segment. For example

Codeword : 000000 321452

Test code: 315735 070688 0 blacks, 3 whites

The 3 whites clearly do not come from the first segment; hence we cannot use this in our Swaszek Algorithm for the first segment. However, once the first segment is completed, using white hits is allowed for the second segment, as the first segment has no way of affecting the outcome in the second segment.

Codeword: 000000 321452

Test code: 000000 515712 7 black hits (6 from first segment), 2 white hits

The white hits must come from the second segment, so white hits can be considered for the second segment.

This is therefore the function of the bool "usewhite".

```
86      bool usewhite = false;
```

It is initialized to false but is turned on after completing the first segment (line 202).

This segmenting process also uses an int called "pointer", which points at the first term of the segment (pointer in this case is not in the context of arrays). For the first segment, "pointer" will

point at 0, and for the second segment, "pointer" will point at the size of the first segment. For example

Code: 000000 000000; segment 1 = 6, segment 2 = 6 (split by half)

For first segment, we have pointer = 0:

```
000000 000000
↑
```

For second segment, we have pointer = segment 1 = 6

```
000000 000000
       ↑
```

It points at the 7th term because this is 0 indexed (so pointer = 6 points at the 7th term).

The next steps are identical to that of the Modified Swaszek Algorithm, just that the whole code will be split into two segments and solved one by one. Switching from the first segment to the second will be done when the black hits equal to the size of the first segment. For example

Codeword: 0251206 2305216; size of first segment = 7

Test code: 0256023 0267123; 3 black hits

0251206 0267123; 7 black hits, move to second segment

The next steps are very similar to the Swaszek that, in fact, that both algorithms utilize the same "nextstep" and "nextstepcompare" functions. One difference is that there will be a new int called "setlength". If Swaszek is used, "setlength" will simply equal to the length of the code, but if Split is used, "setlength" will equal to the size of the segment. The int "lengthfromend" in the internal give feedback function will also be adjusted accordingly.

```
337         int lengthfromend = length;
338         setlength = length;
339         if(!swaszekmode){
340           lengthfromend = segment;
341           setlength = segment;
342         }
```

Other processes / functions used in this algorithm not mentioned in this section have most probably been explained in the *Modified Swaszek Algorithm* section. These include using the "speedup" process, the storage of all previous attempts and feedback, as well as the indexing system for the Swaszek algorithm.

This therefore concludes the final algorithm in this code. The charts showing the average attempt and time efficiency of this algorithm in the space it is operating is shown below.

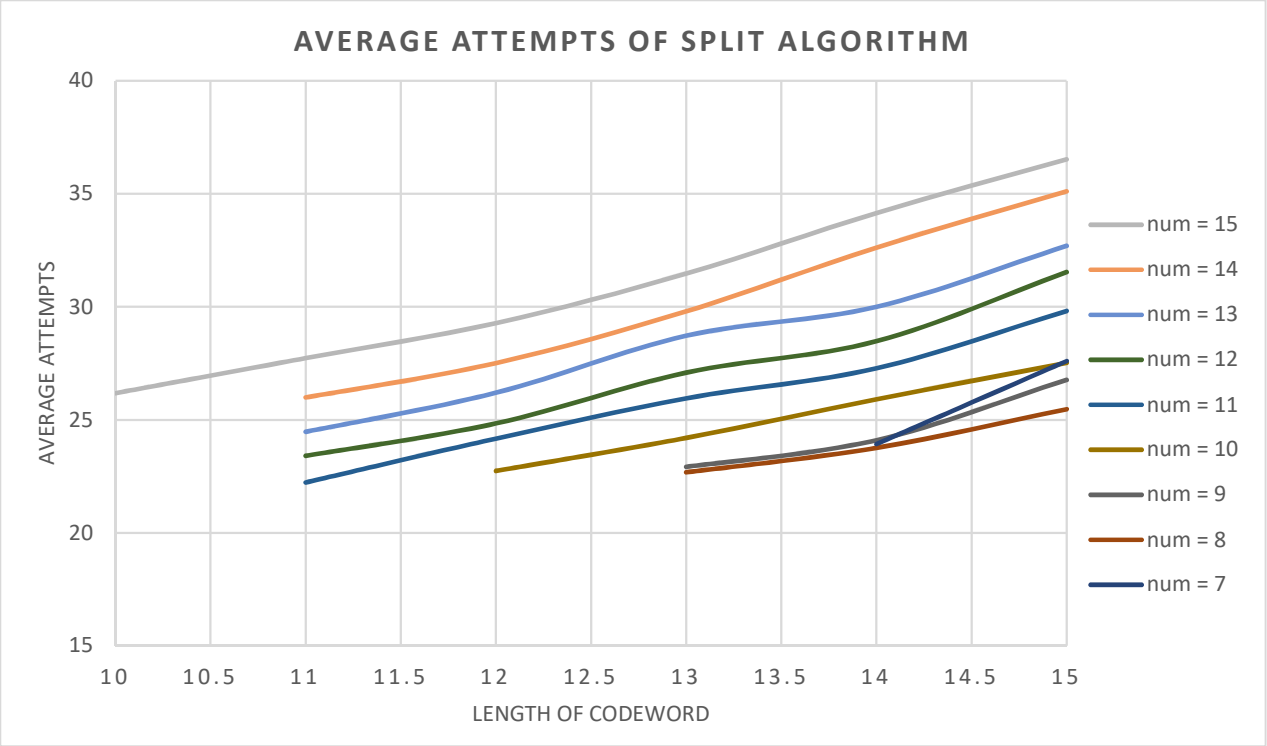


Figure 8: Average Attempts of Swaszek Split Algorithm

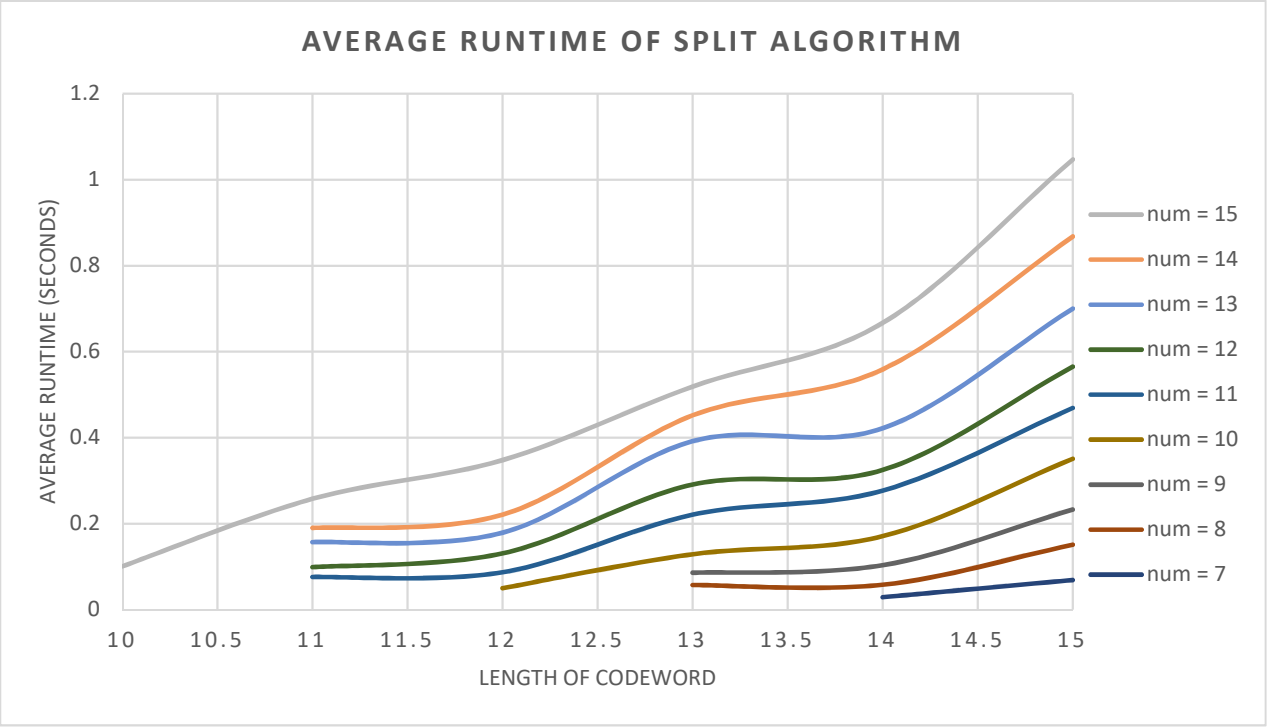


Figure 9: Average Runtime of Swaszek Split Algorithm. Beyond this space, this algorithm will continue to be used as it is most time efficient among the three.

Result

The result of this code is summarized by the following tables below. The numbers and averages are found by iterating the code 100 times.

Attempt Average															
Num	Length														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	1.42	2.06	2.36	2.95	3.24	3.72	4.02	4.51	4.95	5.02	5.7	6.38	6.43	6.95	7.25
3	2.28	2.49	2.84	3.09	3.54	3.95	4.33	4.66	5.01	5.44	6	6.3	6.58	6.98	8.62
4	2.41	2.9	3.32	3.79	4.14	4.41	5.09	5.5	5.81	6.35	6.92	8.5	10.22	11.98	14.63
5	3.08	3.51	3.55	4.21	4.67	4.98	5.67	6.1	6.61	8.04	9.81	11.35	13.5	17.29	21.6
6	3.18	3.76	4.1	4.55	4.99	5.61	6.01	6.56	7.92	9.69	12.23	14.68	17.91	22.27	26.21
7	4.14	4.27	4.77	5.19	5.33	6.07	6.39	7.83	9.73	11.48	13.93	18.01	20.9	23.93	27.6
8	4.25	4.83	5.15	5.45	5.84	6.2	6.87	9.06	11.35	13.35	16.16	20.37	22.68	23.76	25.47
9	4.96	5.27	5.48	5.91	6.28	6.5	7.32	10.02	11.98	15.24	17.53	22.51	22.92	24.1	26.77
10	5.67	6.22	5.77	6.4	6.61	6.95	9.2	11.34	13.88	17.05	21.25	22.74	24.2	25.91	27.52
11	6.24	6.07	6.48	6.37	6.88	7.61	10.08	11.77	15.24	18.46	22.23	24.17	25.95	27.28	29.82
12	6.42	6.82	6.5	6.99	7.17	7.76	10.59	12.87	16.18	20.12	23.41	24.84	27.09	28.48	31.54
13	6.82	7.56	7.15	7.13	7.52	8.16	11.35	14.38	17.43	21.08	24.47	26.2	28.72	30	32.7
14	7.54	7.78	7.6	7.7	7.85	10.52	12.67	15.43	19.39	21.94	25.99	27.51	29.8	32.62	35.11
15	7.78	8.54	7.98	7.78	8.15	10.95	13.84	16	20.53	26.18	27.73	29.28	31.47	34.15	36.53

Figure 10: Attempt Average of the code for the whole 15 by 15 space

Average Runtime (seconds)															
Num	Length														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0.001	0.001	0.002	0.005	0.009
3	0	0	0	0	0	0.001	0.001	0.002	0.004	0.01	0.03	0.099	0.278	0.822	0.157
4	0	0	0	0	0.001	0.001	0.004	0.013	0.049	0.2	0.79	0.14	0.255	0.425	0.686
5	0	0	0	0	0.001	0.003	0.016	0.078	0.385	0.094	0.205	0.391	0.624	1.003	1.502
6	0	0	0	0	0.003	0.014	0.082	0.477	0.11	0.243	0.508	0.834	1.287	1.984	2.864
7	0	0	0	0.001	0.006	0.039	0.235	0.1	0.245	0.414	0.711	1.279	1.713	0.029	0.069
8	0	0	0	0.002	0.012	0.088	0.659	0.0172	0.384	0.619	0.988	1.632	0.0574	0.058	0.151
9	0	0	0.001	0.003	0.026	0.194	1.759	0.253	0.468	0.852	1.204	1.954	0.086	0.10344	0.233
10	0	0	0.001	0.005	0.041	0.375	0.14	0.344	0.654	1.079	1.695	0.05	0.129	0.171	0.351
11	0	0	0.001	0.008	0.07	0.769	0.201	0.352	0.765	1.34	0.076	0.087	0.221	0.277	0.469
12	0	0	0.001	0.013	0.114	1.166	0.235	0.473	0.901	1.47	0.099	0.131	0.291	0.325	0.565
13	0	0	0.002	0.015	0.183	2.255	0.311	0.646	1.076	1.689	0.157	0.179	0.392	0.422	0.7
14	0	0	0.002	0.023	0.273	0.202	0.433	0.754	1.304	1.742	0.19	0.221	0.452	0.559	0.868
15	0	0	0.03	0.034	0.422	0.194	0.499	0.791	1.539	0.101	0.258	0.348	0.519	0.667	1.047

Figure 11: Average runtime of the code for the whole 15 by 15 space

Maximum Attempt															
Num	Length														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	2	3	4	5	6	6	7	8	7	8	8	9	9	9	11
3	3	4	5	5	5	6	7	7	8	8	9	8	9	9	14
4	4	5	5	6	5	6	7	8	8	9	9	17	15	24	26
5	5	6	6	6	7	7	9	8	9	12	15	25	28	33	51
6	6	6	7	6	7	8	8	9	13	17	23	23	36	47	64
7	7	7	8	7	8	8	8	13	15	23	24	39	38	58	62
8	8	8	8	8	8	8	9	14	21	22	29	37	44	56	53
9	9	9	9	8	10	8	9	20	24	28	43	35	48	42	45
10	10	10	8	9	9	10	13	18	23	29	35	40	37	45	41
11	11	11	10	8	10	11	16	17	33	30	38	39	39	38	49
12	12	12	10	12	10	11	14	24	29	32	36	33	38	45	55
13	13	13	13	11	10	11	19	26	29	32	34	35	40	40	42
14	14	14	12	13	12	15	19	25	31	45	33	36	40	49	52
15	15	14	14	13	14	14	21	25	34	36	36	38	44	46	52

Figure 12: Maximum number of attempts of the code for the whole 15 by 15 space

Minimum Attempt															
Num	Length														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	2	2	2	2	3	2	2	3	3
3	1	1	1	1	1	2	3	2	2	3	3	4	3	5	4
4	1	1	1	1	2	3	2	3	3	4	5	5	6	6	7
5	1	1	1	2	2	3	3	3	3	4	6	4	5	8	9
6	1	1	1	2	2	3	4	4	5	5	7	7	9	9	10
7	1	2	2	3	3	4	3	5	6	5	7	9	7	14	16
8	1	1	1	3	3	4	5	5	7	6	7	9	13	15	15
9	1	1	1	2	4	4	5	6	6	8	8	11	14	15	16
10	1	2	1	3	4	3	6	5	5	7	10	15	16	17	18
11	1	1	3	4	4	5	6	7	7	9	14	15	16	20	19
12	1	1	1	4	4	5	7	8	9	10	17	18	18	18	21
13	1	1	3	4	4	6	7	6	9	10	16	18	20	21	24
14	1	1	4	4	4	7	7	9	11	9	18	18	21	22	24
15	1	2	4	3	5	6	9	8	9	17	20	20	23	24	26

Figure 13: Minimum number of attempts of the code for the whole 15 by 15 space

Below shows the chart for the full code in the 15 by 15 space (instead of showing them separately based on the three algorithms)

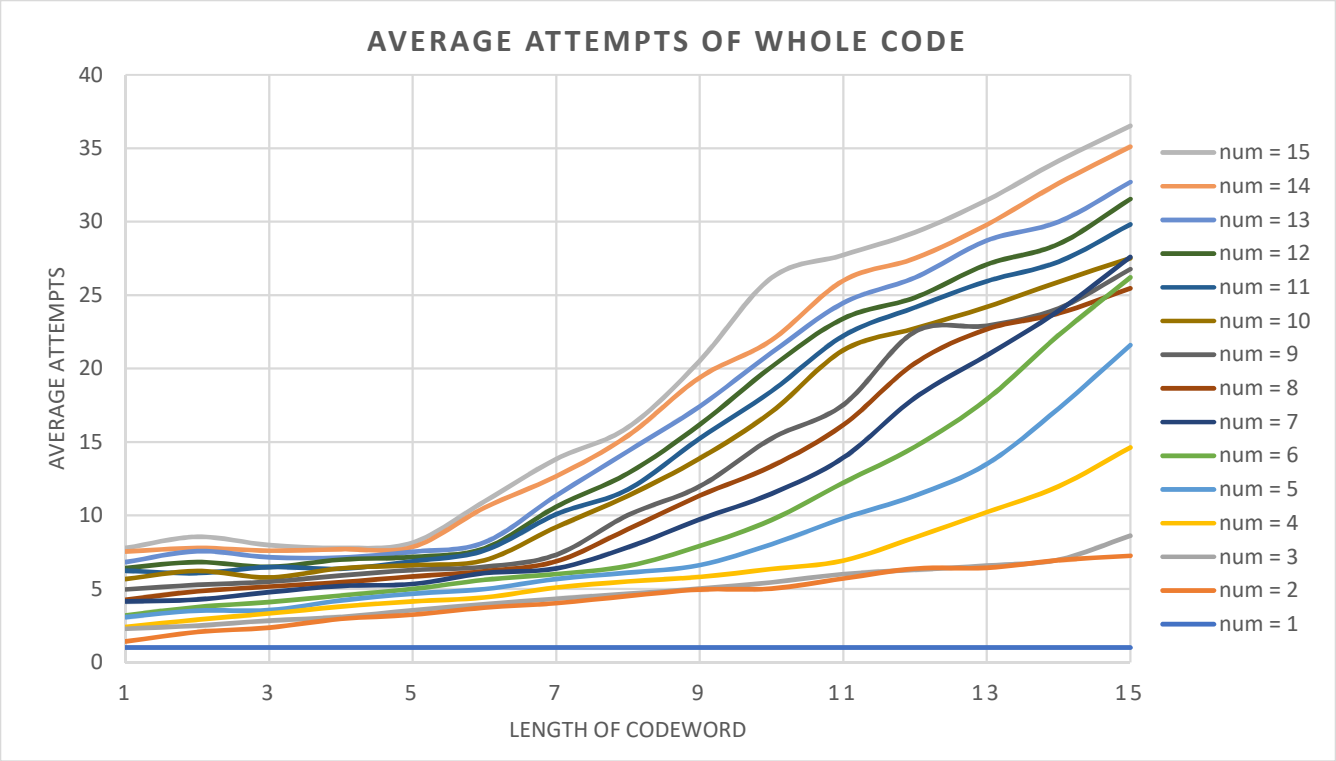


Figure 14: Average attempts of whole code in the 15 by 15 space

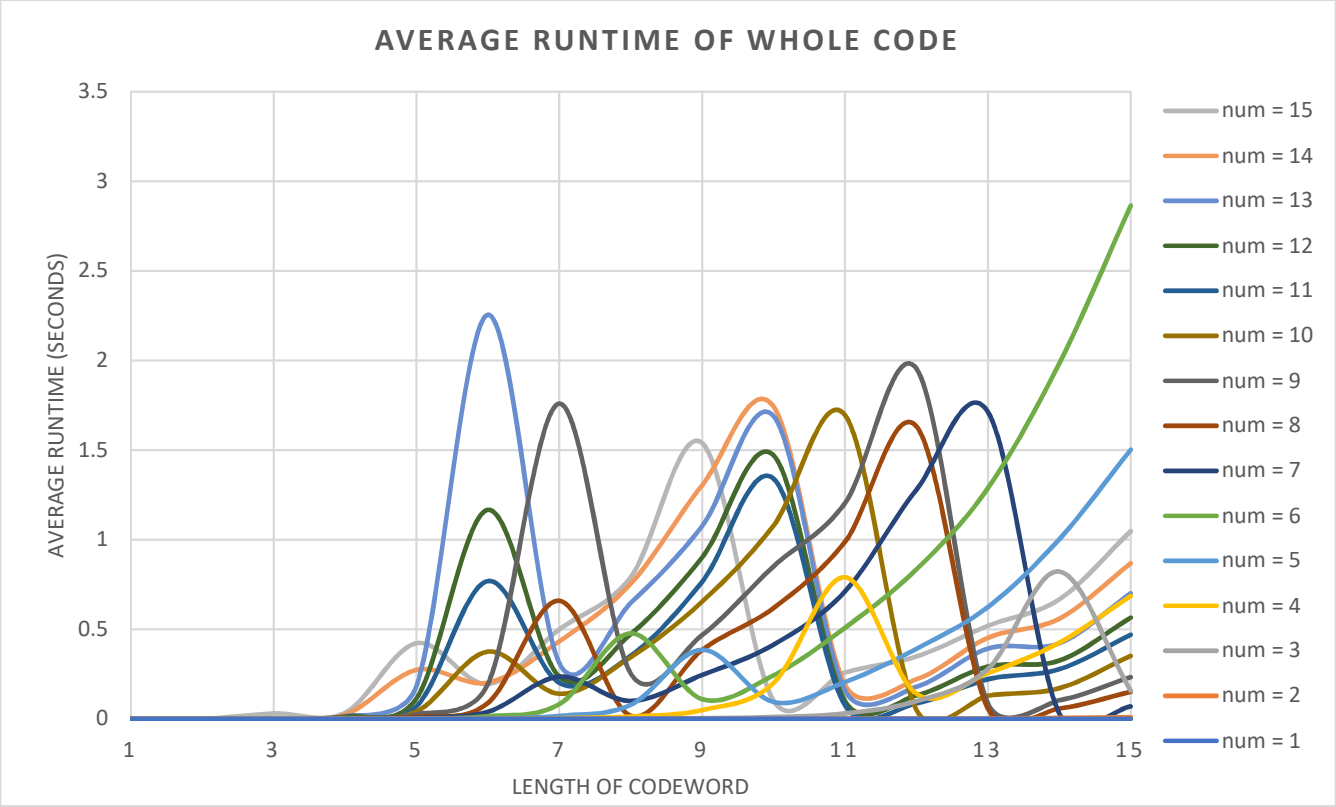


Figure 15: Average runtime of whole code in the 15 by 15 space

The very erratic behavior of the average runtime is due to the switching between the three algorithms. The peak and the drop occur at every switching point between the Knuth and Swaszek, and between Swaszek and Split. The switching points can be determined in Figure 2. The average attempts, on the other hand, is much more stable, with changes in gradient representing the switching between algorithms. The switching from Knuth to Swaszek and Swaszek to Split will reduce attempt-efficiency, but improve time-efficiency, as can be seen from the peaks and drops in the average runtime graph. This will ensure that the code operates within the 10s runtime.

Bibliography

[1] V. Megalooikonomou, "CIS603 S03," *CIS603 Spring 03: Lecture 7*, 2003. [Online]. Available: https://cis.temple.edu/~vasilis/Courses/CIS603/Lectures/l7.html. [Accessed: 21-Mar-2019].

[2] D. E. Knuth, *The Computer as Mastermind*, 1977th-1977th ed., vol. 9(1).

[3] P. F. Swaszek, "The mastermind novice - researchgate.net," *Research Gate*, 2000. [Online]. Available: https://www.researchgate.net/publication/268644635_The_mastermind_novice. [Accessed: 21-Mar-2019].

[4] ORSTAT, K.U.Leuven, "Efficient Solutions for Mastermind using genetic Algorithms". [Online]. Available: http://www.rosenbaum-games.de/3m/p1/Mastermind/2009Berghman01.pdf [Accessed: 21-Mar-2019].