

GitHub Repository: <https://github.com/Alireza-Ezaz/Optimal-Edit-Metric-Codes>

In this assignment, we want to establish **bounds** on the size of **optimal edit metric error-correcting codes** using a chosen computational approach.

Assumptions:

- We will work with **fixed-length**  $(n, M, d)_q$  **edit metric codes**.
- These are sets of  $M$   $q$ -ary codewords of length  $n$  where every pair of codewords is at least  $d$  edit distance apart.
- We want to determine the largest possible value of  $M$  for a given  $n$ ,  $d$ , and  $q$  such that there exists an  $(n, M, d)_q$  code.

We start with the **simplest possible** (maybe the worst in terms of computational time 😞) approach and then try to improve it step by step. For the simplest approach to solve this problem (**exhaustive search**), we can:

1. Generate all possible  $q$ -ary codewords of length  $n$
2. Create all possible combination sets of those codewords
3. Check whether all pairs of codewords in each set have an edit distance  $\geq d$  and if they do, they are valid.
4. Keep the valid set with the greatest number of codewords as an optimal edit metric error-correcting code with  $M$  codewords.

In the **simplest.py** file, you will find my code following the above steps as below:

```
43 def find_optimal_code(n, q, d):
44     all_codewords = generate_codewords(n, q) step 1
45     # initialize the optimal code to be the empty set
46     optimal_code = set()
47     for r in range(1, len(all_codewords) + 1):
48         # "itertools.combinations" returns all possible combinations of the given iterable (in this case, the codewords)
49         for codeword_set in itertools.combinations(all_codewords, r): step 2, 3
50             if is_code_valid(codeword_set, d) and len(codeword_set) > len(optimal_code):
51                 optimal_code = codeword_set step 4
52     return optimal_code
53
54
```

Notes:

- The *“generate\_codewords”* function generates all possible  $q$ -ary codewords of length  $n$
- The *“is\_code\_valid”* function checks whether the given code is a valid code with minimum edit distance  $\geq d$  between any two codewords. For that purpose, it uses the *“calculate\_edit\_distance”* function implementing the algorithm in [slide 13- week7](#) and using a [dynamic programming technique](#).

### Analysis of the simplest approach

The “*find\_optimal\_code*” function finds the optimal code of length  $n$  and size  $q$  with minimum edit distance  $d$  between any two codewords by exhaustively checking all the possible sets of codewords.

In this approach, we must check the space of  $2^{(q^n)}$  sets of codewords. (where  $q^n$  is the number of possible codewords and in that case, we have  $2^{(q^n)}$  subsets )

For each set, we need to check the edit distance between every pair of codewords. In the worst case, this is  $\binom{P}{2}$  comparisons per set, where  $P$  is the maximum number of codewords in a set.

Each comparison takes  $O(m * n)$  time, where  $m$  is the length of codeword1 and  $n$  is the length of codeword2. (In our assumptions we know that we are working on **fixed-length** ( $n, M, d$ ) **q edit metric codes**. So, we can say each comparison takes  $O(n^2)$  time)

As a result, the total time complexity is  $O(2^{(q^n)} * \binom{P}{2} * n^2)$  which obviously needs improvements. Let's see how far we can go with this approach and find the optimal codes with different parameters.

**q = 2**

N \ d	1	2	3	4	5
1	2	1	1	1	1
2	4	2	1	1	1
3	8	4	2	1	1
4	16	8	2	2	1
5	?	?	?	?	?

- For  $n = 4$ ,  $d = 1$  it took about a second
- For  $n = 4$ ,  $d = 2$  it took 1.36s
- For  $n = 5$ ,  $d = 1$  it took too much time :(

**q = 3**

N \ d	1	2	3	4	5
1	3	1	1	1	1
2	9	3	1	1	1
3	?	?	?	?	?

- For  $n = 2$ ,  $d = 1$  it took 0.02s
- For  $n = 3$ ,  $d = 1$  it took too much time :(

**q = 4**

N \ d	1	2	3	4	5
1	4	1	1	1	1
2	16	4	1	1	1
3	?	?	?	?	?

- For  $n = 2$ ,  $d = 1$  it took 10.71s
- For  $n = 2$ ,  $d = 2$  it took 0.5s
- For  $n = 3$  it took too much time :(

**Conclusion:** It is obvious that we cannot go beyond  $n = 4$  or  $n = 5$  in a reasonable amount of time. So, we start improving our approach.

To improve our approach, we can use the **backtracking algorithm** introduced in [week5slides](#). Backtracking is a more efficient approach compared to a simple exhaustive search. By using backtracking, we can eliminate large portions of the search space that cannot possibly contain a solution, thereby reducing computation time significantly.

In the **Backtracking.py** you will find the previous functions in addition to 3 new functions enabling us to reduce the search space.

- **`is_code_valid_after_addition_of_new_codeword(current_set, new_codeword, d)`**: function (which is a modified version of `is_code_valid` in previous file) returns True if we can add the given `new_codeword` to an already valid existing code and false otherwise. This function will be used in the **`backtrack`** function.

```
# This function checks if the new codeword can be added to the current set of codewords while maintaining the minimum edit distance d
1 usage  ▲ Alireza Ezaz
def is_code_valid_after_addition_of_new_codeword(current_set, new_codeword, d):
    return all(calculate_edit_distance(new_codeword, cw) >= d for cw in current_set)
```

- **`backtrack(codewords, n, q, d, current_set, optimal_set)`**: function implements the backtracking algorithm proposed in [week5slides](#).

```
47 # This is the basic backtracking algorithm that we discussed in class
2 usages  ▲ Alireza Ezaz *
48 def backtrack(codewords, n, q, d, current_set, optimal_set):
49     # If the current set is a valid code with a larger size than the optimal code, we update the optimal code
50     if len(current_set) > len(optimal_set[0]):
51         optimal_set[0] = current_set.copy()
52
53     for codeword in codewords:
54
55         if is_code_valid_after_addition_of_new_codeword(current_set, codeword, d):
56             # We add the codeword to the current set if it still satisfies the minimum edit distance d
57             current_set.append(codeword)
58             # We only need to consider codewords that come after the current codeword in the list
59             codewords_for_next_level = codewords[codewords.index(codeword) + 1:]
60             backtrack(codewords_for_next_level, n, q, d, current_set, optimal_set)
61             current_set.pop()
62
```

- **Challenge**: It took me some time to debug my backtrack implementation as I wasn't calculating the **`codewords_for_next_level`** at first, passing the whole set of codewords which would result in extra computational time. By the way, I finally solved it 😊.
- Finally the last added function is **`find_optimal_code_with_backtracking(n, q, d)`** which just calls the `backtrack` function and also computes the execution time for our analysis.

```
2 usages  ▲ Alireza Ezaz
54 def find_optimal_code_with_backtracking(n, q, d):
55     start_time = time.time()
56     codewords = generate_codewords(n, q) # Could be optimized for on-the-fly generation
57     optimal_set = [[]]
58     backtrack(codewords, n, q, d, current_set=[], optimal_set)
59     ⚡ end_time = time.time()
60     return optimal_set[0], end_time - start_time
```

### Analysis of the Backtracking approach (improvement of exhaustive search with backtracking)

Technically, the time complexity remains the same but in practice, we discarded the large portion of the search space and as a result, we are now capable of finding bound for a larger amount of  $n$  and  $q$  in a reasonable amount of time.

**q = 2**

N \ d	1	2	3	4	5
1	2	1	1	1	1
2	4	2	1	1	1
3	8	4	2	1	1
4	16	8	2	2	1
5	*	16	4	2	2
6	*	*	7	4	2
7	*	*	12	5	2
8	*	*	*	*	4

- For  $n = 5$ ,  $d = 2$  it took 31.68s
- For  $n = 5$ ,  $d = 3, 4, 5$  it took almost nothing
- For  $n = 5$ ,  $d = 1$  it again took too much time
- For  $n = 6$ ,  $d = 3$  it took 8.53s
- For  $n = 6$ ,  $d = 4$  it took 0.1s
- For  $n = 7$ ,  $d = 4$  it took 9.5s
- For  $n = 8$ ,  $d = 5$  it took 15s

**Example:** For  $n = 7$ ,  $d = 4$  the code was: ['0000000', '0001111', '0110011', '0111100', '1010101']

**q = 3**

N \ d	1	2	3	4	5
1	3	1	1	1	1
2	9	3	1	1	1
3	*	9	3	1	1
4	*	*	7	3	1
5	*	*	*	5	3

- For  $n = 2$ ,  $d = 1$  it took 0.004s < 0.02s without backtracking
- For  $n = 3$ ,  $d = 2$  it took 0.59s
- For  $n = 3$ ,  $d = 3$  it took almost nothing
- For  $n = 4$ ,  $d = 3$  it took 7.74s
- For  $n = 5$ ,  $d = 4$  it took 67.96s

**Example:** For  $n = 5$ ,  $d = 4$  the code was: ['00000', '01112', '10222', '12011', '22120']

**q = 4**

N \ d	1	2	3	4	5
1	4	1	1	1	1
2	16	4	1	1	1
3	*	*	4	1	1

- For  $n = 2$ ,  $d = 1$  it took 1.3s < 10.71s before
- For  $n = 3$ ,  $d = 3$  it took 0.26s
- For  $n = 3$ ,  $d = 4$  it took 0.01s

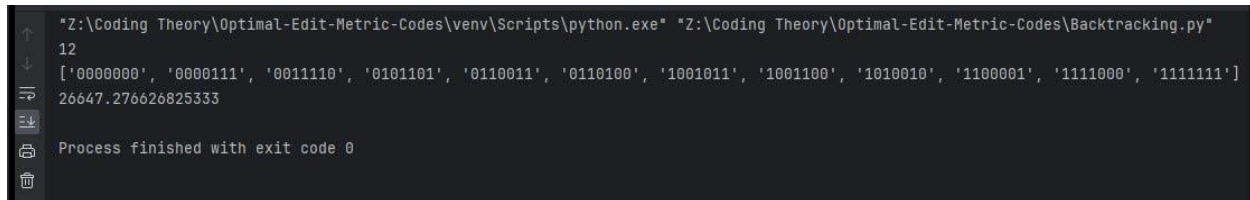
**Conclusion:** Using Backtracking we can go further (green cells) **specifically** when  $d$  is also a greater number and find bounds in a reasonable time. For example, for  $n = 9$   $q = 2$  and  $d = 6$  we get the following result in only 38s: ['000000000', '000111111', '111000111', '111111000']

```
"Z:\Coding Theory\Optimal-Edit-Metric-Codes\venv\Scripts\python.exe" "Z:\Coding Theory\Optimal-Edit-Metric-Codes\Backtracking.py"
4
['000000000', '000111111', '111000111', '111111000']
38.38008155678166
Process finished with exit code 0
```

We still Cannot find the optimal code for something like  $n = 8$   $q = 2$  and  $d = 3$  in a reasonable time. However, we are observing that we can go further for greater amounts diagonally (as we increase the  $n$ , we must increase  $d$  as well).

Example:  $n = 11$   $q = 2$   $d = 10$  takes 72s and finds ['00000000000', '01111111111']

Note: The most difficult problem my code was able to solve was  $n=7$ ,  $q = 2$ ,  $d = 3$  which took 26647 seconds equal to **7 hours and 40 minutes to find  $M = 12$ !** (I left the lab computer to run it and came back tomorrow to check the result :))



```
"Z:\Coding Theory\Optimal-Edit-Metric-Codes\venv\Scripts\python.exe" "Z:\Coding Theory\Optimal-Edit-Metric-Codes\Backtracking.py"
12
['0000000', '0000111', '0011110', '0101101', '0110011', '0110100', '1001011', '1001100', '1010010', '1100001', '1111000', '1111111']
26647.276626825333
Process finished with exit code 0
```

### Optimizations on the Backtracking approach

In the **Optimized.py** you will find the final version with every improvement possible.

First, we change our backtrack function a little bit to take **M** (Maximum length of the code) and **level** as inputs. By taking **M** we are at least able to find some lower bounds for larger amounts of  $n$  for which we were not able to find the optimal code. the **level** also helps us to implement an optimization mentioned in [slide 9 – Week5Slides](#). The optimization follows the following: **The candidates for a level should only contain words starting with an allowed symbol.**

A potential approach could be to start with a more relaxed restriction and gradually make it stricter based on the depth of the search (level). I did this. However, the slides, use the following (I had challenges when applying the slides' bounds and this is probably because the backtracking algorithm in the slides aims to find all codes and is a bit different from our problem here):

- Symbol 0 is allowed at any level
- Symbol 1 is allowed only when  $\text{level} \geq \lceil M/q \rceil + 1$
- Symbol 2 is allowed only when  $\text{level} \geq |C_0| + \lceil (M - |C_0|) / (q-1) \rceil + 1$  (Etc.)

Anyway, I implemented the **is\_starting\_symbol\_allowed\_in\_the\_level(codeword, level, q):** function like the following (The numbers came from trial and error and we might need to change the thresholds according to  $M$  to see the effects):

```

1 usage  S.Alireza Ezaz +1
70 def is_starting_symbol_allowed_in_the_level(codeword, level, q):
71     first_symbol = int(codeword[0])
72     # We define thresholds for each symbol
73     # result = math.ceil(M / q) + 1 as slides approach
74     threshold_for_1 = 2 # symbol '1' is allowed from level 2 onwards
75     threshold_for_2 = 4 # symbol '2' is allowed from level 4 onwards
76     threshold_for_3 = 6 # symbol '3' is allowed from level 6 onwards
77
78     if first_symbol == 1 and level < threshold_for_1:
79         return False
80     elif first_symbol == 2 and level < threshold_for_2:
81         return False
82     elif first_symbol == 3 and level < threshold_for_3:
83         return False
84     return True
85

```

Another improvement can be first calculating Hamming Distance using the XOR (only for binary we can use xor) operation which is efficient:

- if hamming distance < d → no need for calculating edit distance as it won't satisfy edit distance
- if hamming distance >= d → calculate edit distance.
- *calculate\_hamming\_distance* function takes care of this integrated with modified *is\_code\_valid\_after\_addition\_of\_new\_codeword(current\_set, new\_codeword, d)* function

```

48 def is_code_valid_after_addition_of_new_codeword(current_set, new_codeword, d):
49     # We iterate over each codeword in the current set
50     for cw in current_set:
51         # First, calculate the Hamming distance between the new codeword and the current codeword
52         hamming_dist = calculate_hamming_distance(new_codeword, cw)
53
54         # If the Hamming distance is already less than d, the new codeword is not valid
55         if hamming_dist < d:
56             return False
57
58         # If the Hamming distance is equal to or greater than d, we calculate the edit distance
59         edit_dist = calculate_edit_distance(new_codeword, cw)
60
61         # If the edit distance is also less than d, the new codeword is not valid
62         if edit_dist < d:
63             return False
64
65     # If the new codeword passes all checks, it is valid
66     return True
67

```

```
# This function calculates the Hamming distance between two given codewords
1 usage
def calculate_hamming_distance(codeword1, codeword2):
    # We first check if codewords are binary
    if set(codeword1).issubset({'0', '1'}) and set(codeword2).issubset({'0', '1'}):
        # Convert binary strings to integers and calculate Hamming distance using bitwise XOR
        int1, int2 = int(codeword1, 2), int(codeword2, 2)
        xor_result = int1 ^ int2
        return bin(xor_result).count('1')
    else:
        # For non-binary strings, we use the standard method
        return sum(c1 != c2 for c1, c2 in zip(codeword1, codeword2))
```

We can even precompute some distances in advance, to make another step of improvement.

### Analysis of the Optimizations on the Backtracking Approach

The best way to analyze the optimization results is to compare the computational time for the same parameters in **Optimized.py** and **Backtracking.py**.

**n = 6, q = 2, d = 3**

- Exhaustive search → cannot find in a reasonable amount of time
- Simple Backtracking → 8.60s

```
"Z:\Coding Theory\Optimal-Edit-Metric-Codes\venv\Scripts\python.exe" "Z:\Coding Theory\Optimal-Edit-Metric-Codes\Backtracking.py"
7
['000001', '001100', '011011', '100111', '101010', '110000', '111101']
8.600191595170166
Process finished with exit code 0
```

- Optimized Backtracking (M = 7) → 5.90s

```
"Z:\Coding Theory\Optimal-Edit-Metric-Codes\venv\Scripts\python.exe" "Z:\Coding Theory\Optimal-Edit-Metric-Codes\Optimized.py"
7
['000001', '001100', '011011', '100111', '101010', '110000', '111101']
5.900764465332031
Process finished with exit code 0
```

**n = 4, q = 3, d = 3**

- Exhaustive search → cannot find in a reasonable amount of time
- Simple Backtracking → 7.51s

```
"Z:\Coding Theory\Optimal-Edit-Metric-Codes\venv\Scripts\python.exe" "Z:\Coding Theory\Optimal-Edit-Metric-Codes\Backtracking.py"
7
['0001', '0110', '0222', '1020', '1211', '2012', '2200']
7.511336088180542
Process finished with exit code 0
```

- Optimized Backtracking (M = 7) → 1.16s

```
"Z:\Coding Theory\Optimal-Edit-Metric-Codes\venv\Scripts\python.exe" "Z:\Coding Theory\Optimal-Edit-Metric-Codes\Optimized.py"
7
['0001', '0110', '0222', '1020', '1211', '2012', '2200']
1.1687984466552734
Process finished with exit code 0
```

**Conclusion:** With these optimizations, the algorithm will work for slightly bigger parameter values and slightly better for the same parameters as compared above.

### Overall Conclusion

My initial approach using exhaustive search highlighted the limitations of this method, especially for larger values of  $n$  (codeword length) and  $q$  ( $q$ -ary). The computational time increases exponentially, making it impractical for larger parameters.

The introduction of backtracking significantly improved the computational efficiency. This approach helped us to explore larger parameter spaces (higher values of  $n$  and  $q$ ) within a reasonable time, as demonstrated by my results.

The additional optimizations, such as using Hamming Distance where applicable and adjusting the search strategy (thresholds) based on the depth of the search (level), further enhanced the performance. These optimizations allowed the algorithm to work for slightly larger parameter values more efficiently.

### Future Improvements

- First, I think Considering more advanced algorithmic techniques, specifically those using AI Approaches, heuristics, or machine learning approaches, such as genetic algorithms or 3 heuristics provided in the paper on the Brightspace, can help us improve solving this problem way more efficiently.
- Second, implementing parallel computing strategies could significantly reduce computation time. By distributing the workload across multiple processors or machines, we can explore larger search spaces more efficiently.

### References:

- <https://www.youtube.com/watch?v=We3YDTzNXEk>
- Provided paper
- Lecture slides
- ChatGPT for helping understand the problem and giving the general idea.