AN IMPROVED LOSSLESS IMAGE COMPRESSION ALGORITHM BASED ON HUFFMAN CODING

İlayda Akınet - Yahya Çakıcı

INTRODUCTION

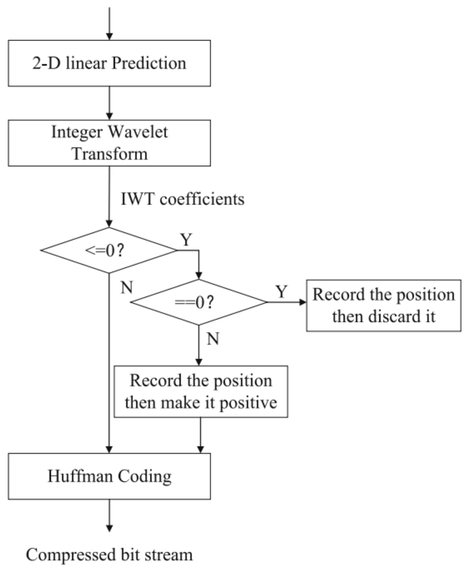
In this study, we aim to evaluate and implement an improved lossless image compression algorithm by combining the following methods:

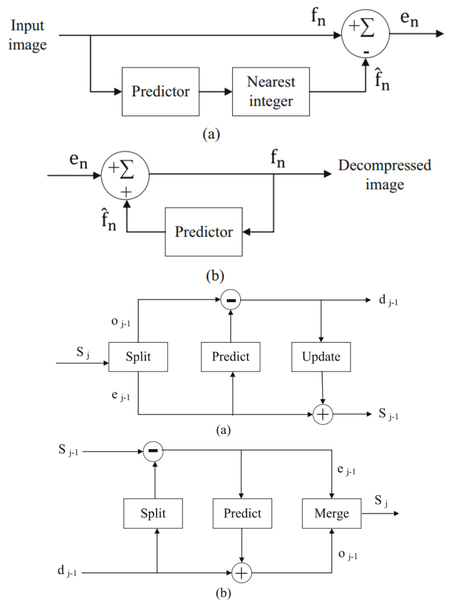
- Linear Prediction for reducing spatial redundancy
- Integer Wavelet Transform for efficient data representation
- Huffman Coding for entropy-based compression

Our goal is to test the proposed approach on various datasets, compare its compression performance, and analyze its suitability for lossless image compression tasks in different domains.

After our tests and observations, we hope to see the proposed algorithms efficiency in image compression, and understand the possible parameters that can affect that efficiency.

METHODOLOGY





The proposed lossless image compression algorithm integrates three core techniques to achieve high compression efficiency without compromising data integrity. These steps are as follows:

- **Linear Prediction**

Linear prediction reduces spatial redundancy in the image by estimating pixel values based on their neighbors. For a given pixel, the prediction is calculated as the average of its neighboring pixel values. The difference between the actual and predicted values (residuals) is stored, significantly reducing the entropy of the image.

Predicted Value = (Neighbor1 + Neighbors2) / 2

Residual = Actual Value - Predicted Value

- **Integer Wavelet Transform (IWT)**

The IWT is applied to the residuals obtained from the linear prediction step. This transformation decomposes the image into approximation and detail coefficients, providing a more compact representation of the data. Positive values are kept as is, while negative values are transformed into their absolute values, and their positions are encoded separately. This step further minimizes the entropy of the data.

- **Huffman Coding**

Huffman coding is used to encode the transformed data efficiently. It assigns shorter codes to more frequent values and longer codes to less frequent values, ensuring minimal storage requirements. A binary tree is constructed based on the frequencies of the data, and the final encoded bitstream is generated.

DATA AND EXPERIMENTAL SETUP

All experiments were conducted on the same computer to ensure consistency. A computer with sufficient hardware specifications was used to handle the computational requirements of the image compression tasks.

For the experimental datasets, **images of varying sizes and resolutions** were utilized. This allowed us to observe how the algorithm's performance changes based on input size and complexity. The datasets included small images as well as larger, high-resolution images to test the scalability and efficiency of the proposed approach.

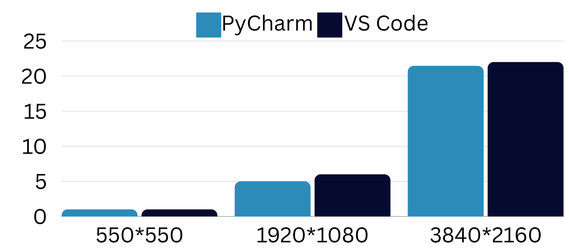
The algorithm was implemented in both **C++** and **Python**. This provided an opportunity to compare the execution time and performance of the algorithm across these programming environments. Observations were made regarding the runtime differences, with C++ generally offering faster execution due to its lower-level nature, while Python demonstrated ease of implementation.

RESULTS

We conducted a series of experiments to evaluate the performance of the proposed algorithm under different conditions:

1-Interpreter Performance:

- The algorithm was first tested using Python on two different interpreters – **PyCharm and Visual Studio Code** (VS Code). While the results were really close, we observed some difference in execution time.



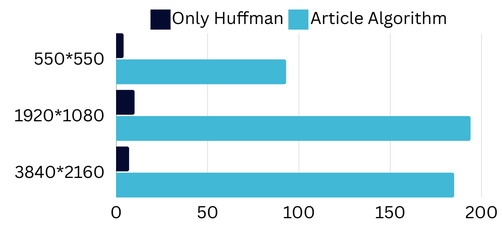
2-C++ vs Python Comparison:

- Next, we implemented the algorithm in both C++ and Python using VS Code. The results showed that the C++ implementation outperformed Python in terms of execution time.
  - Reason: This is likely due to the fact that C++ is a compiled language, optimized for performance, whereas Python is an interpreted language, which introduces runtime overhead.

Item	Size	c++ Ratio	Python Ratio	c++ Time	Python Time
1	686*386	1.55	1.57	0.013	1.93
2	1200*630	2.54	2.57	0.038	6.12
3	1920*1080	2.98	2.94	0.104	15.29
4	3840*2160	2.89	2.85	0.41	60.69
5	550*550	1.85	1.85	0.014	2.43
6	800*500	1.86	1.86	0.021	2.82
7	1500*1000	1.42	1.42	0.073	11.21
8	1073*605	2.08	2.08	0.032	4.98
9	128*128	1.47	1.47	0.001	0.13
10	2048*1635	2.6	2.59	0.156	23.62
11	3840*2160	2.01	2.01	0.404	60.71
12	7680*4320	4.86	4.89	1.803	229.35

3-Comparative Analysis of Huffman Coding:

- To further understand the strength of the algorithm, we tested two additional Python scripts:
  - One without Huffman coding.
  - Another using only Huffman coding for compression.
  - The results revealed that the compression ratio significantly decreased when Huffman coding was either omitted or used alone.



CONCLUSION

Based on our observations, we conclude that the compression algorithm presented in the studied paper is a powerful approach. Its effectiveness is further enhanced when implemented in **C++**, delivering superior performance in terms of speed and efficiency.

However, to further improve this already effective algorithm, enhancements can be made to address its processing time and hardware resource usage. Optimization techniques can be explored to reduce computational overhead and achieve faster performance. For example, **parallel processing** can be a way to optimize and enhance the algorithm.

The algorithm's complexity has been calculated as **O(M×N)**, where M and N represent the rows and columns of the image, respectively. Accordingly, as the size of the tested data increases, the time complexity grows. This is why larger images took significantly longer to execute compared to smaller ones.

REFERENCES

- Wang, Zhen, et al. "An improved lossless image compression algorithm based on Huffman coding." Multimedia Tools and Applications, vol. 81, no. 18, 2022, pp. 4781-4795., Springer