

INF 365 Théorie de l'Information Projet

Compression de LZW

Avec Correction de Code de Huffman (15,11)

Emir Kaan Öz / 19401847

INTRODUCTION

Ce projet se compose de 4 parties : Compression, Décompression, Code Correction et Bilan. Étant donné que le projet est réalisé à l'aide des algorithmes LZW et Huffman, je les expliquerai dans les trois premières parties. Enfin, j'expliquerai les méthodes que j'ai utilisées pour calculer différents aspects de l'efficacité du code tels que le gain l'utilisation de la capacité du canal et le taux de compression.

COMPRESSION ET DECOMPRESSION

Pour effectuer la compression, j'ai décidé d'utiliser l'algorithme LZW, car j'ai découvert qu'il était également utilisé dans la compression des GIF. Ma première intention était d'utiliser la compression GIF pour ce projet, mais je ne me faisais pas suffisamment confiance pour le construire et le terminer, alors j'ai décidé d'en choisir une plus simple. Comme vous le verrez plus tard dans ce rapport de projet, j'expliquerai ma méthode de compression de texte avec l'algorithme LZW.

Tout d'abord, je veux commencer par expliquer brièvement le fonctionnement de l'algorithme LZW. L'algorithme LZW (Lempel-Ziv-Welch) est un algorithme de compression de données sans perte. Il fonctionne en remplaçant des séquences répétitives de caractères par des codes plus courts. Voici un aperçu succinct de son fonctionnement :

1. Initialisation du dictionnaire : L'algorithme LZW commence par créer un dictionnaire initial qui contient tous les caractères possibles du jeu de données à compresser, chacun associé à un code unique.
2. Lecture de la séquence de données : L'algorithme lit la séquence de données à compresser caractère par caractère.
3. Construction du dictionnaire : Lors de la lecture de la séquence, l'algorithme commence à construire le dictionnaire en ajoutant de nouvelles séquences de caractères rencontrées au fur et à mesure dans le dictionnaire. Chaque nouvelle séquence est associée à un code unique.
4. Recherche des séquences répétitives : Lorsque l'algorithme rencontre une séquence déjà présente dans le dictionnaire, il continue à lire la séquence suivante et à former une séquence plus longue jusqu'à atteindre une séquence qui n'est pas présente dans le dictionnaire.
5. Encodage : Lorsqu'une séquence est trouvée qui n'est pas dans le dictionnaire, l'algorithme émet le code correspondant à la séquence précédente (qui est présente

dans le dictionnaire) et ajoute la séquence actuelle avec un nouveau code unique dans le dictionnaire.

6. Répétition : Les étapes 3 à 5 sont répétées jusqu'à ce que toute la séquence de données ait été lue.

Puisque nous avons vu comment l'algorithme fonctionne en théorie, je vais passer pour expliquer comment je l'ai implémenté. Tout d'abord, j'ai commencé par rechercher un moyen de l'implémenter et j'ai cherché des vidéos YouTube l'expliquant. Heureusement, j'en ai trouvé un qui a été expliqué et également implémenté en Java. Malheureusement, je cherchais celui qui était implémenté en C #, car le projet exige également une interface graphique. Je prévoyais de l'implémenter dans Unity et Unity a besoin de scripts C # pour fonctionner. Depuis que j'ai de l'expérience dans le développement de jeux dans Unity et la conception d'interfaces dessus, j'ai pensé que ce serait la meilleure approche pour moi. J'ai donc transformé le script java en C# et terminé les deux premières parties des projets qui sont la compression et la décompression.

J'ai utilisé le code devant pour tester l'algorithme de LZW. J'ai essayé de trouver un string qui consiste des caractères répétant pour tester mieux si l'algorithme fonctionne.

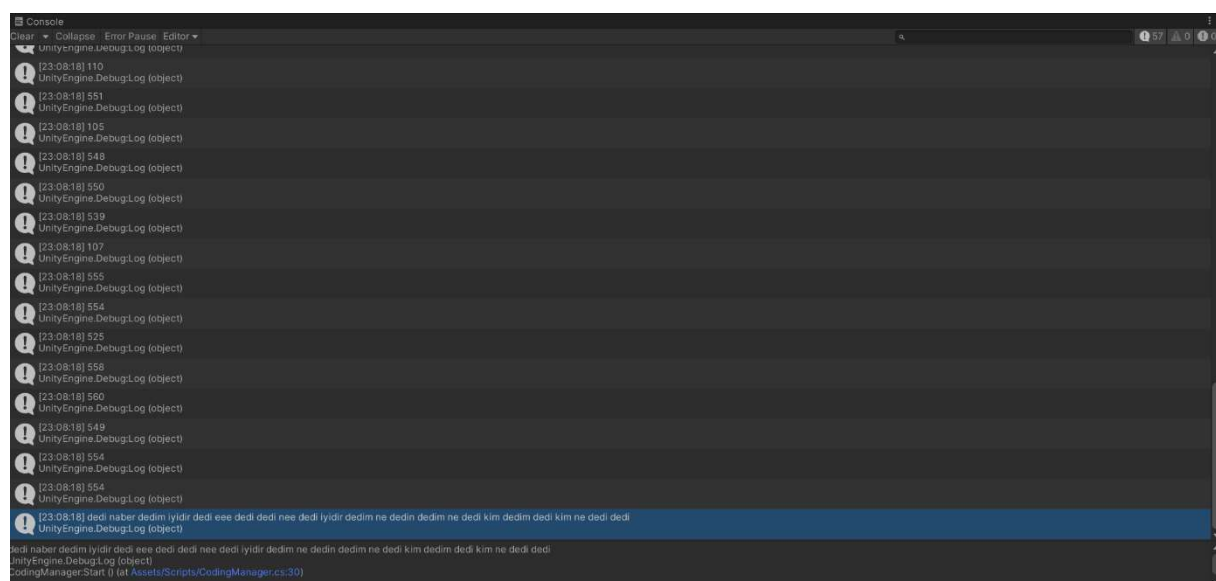
```
#region LZW Test

    List<int> compressed = LZW.Encode("dedi naber dedim iyidir dedi eee dedi
dedi nee dedi iyidir dedim ne dedin dedim ne dedi kim dedim dedi kim ne
dedi dedi");
    foreach (int i in compressed)
    {
        Debug.Log(i);
    }

    String decompressed = LZW.Decode(compressed);
    Debug.Log(decompressed);

#endregion
```

Voici les résultats du code :



```
Console
Clear Collapse Error Pause Editor
[23.08.18] 110 UnityEngine.Debug.Log (object)
[23.08.18] 551 UnityEngine.Debug.Log (object)
[23.08.18] 105 UnityEngine.Debug.Log (object)
[23.08.18] 548 UnityEngine.Debug.Log (object)
[23.08.18] 550 UnityEngine.Debug.Log (object)
[23.08.18] 539 UnityEngine.Debug.Log (object)
[23.08.18] 107 UnityEngine.Debug.Log (object)
[23.08.18] 555 UnityEngine.Debug.Log (object)
[23.08.18] 554 UnityEngine.Debug.Log (object)
[23.08.18] 525 UnityEngine.Debug.Log (object)
[23.08.18] 558 UnityEngine.Debug.Log (object)
[23.08.18] 560 UnityEngine.Debug.Log (object)
[23.08.18] 549 UnityEngine.Debug.Log (object)
[23.08.18] 554 UnityEngine.Debug.Log (object)
[23.08.18] 554 UnityEngine.Debug.Log (object)
[23.08.18] 554 UnityEngine.Debug.Log (object)
[23.08.18] dedi naber dedim iyidir dedi eee dedi dedi nee dedi iyidir dedim ne dedin dedim ne dedi kim dedim dedi kim ne dedi dedi
dedi naber dedim iyidir dedi eee dedi dedi nee dedi iyidir dedim ne dedin dedim ne dedi kim dedim dedi kim ne dedi dedi
UnityEngine.Debug.Log (object)
CodingManager.Start () (at Assets/Scripts/CodingManager.cs:30)
```

CODE CORRECTION

Pour la section de correction de code, j'ai essayé de voir quelles options j'avais donc j'ai fait une recherche avant de commencer. Comme nous n'avons vu que des codes linéaires et une partie de codes polynomiaux dans le cours, je me suis concentré sur ces deux-là. J'ai rapidement abandonné le polynôme car il semblait plus complexe et cette complexité n'était pas nécessaire pour mes besoins.

Pour les codes linéaires, j'ai d'abord fait des recherches approfondies sur leur fonctionnement théorique et j'ai découvert qu'il existe différents types de code de Hamming. Comme Hamming (7,4) ne suffisait pas à mes besoins, j'ai décidé d'aller avec Hamming (15,11). Cela fournirait 2^{11} codes différents à utiliser, ce qui est assez joli pour mon projet.

Le code de Hamming (15,11) est un code de correction d'erreur utilisé pour détecter et corriger les erreurs dans les transmissions de données. Il fonctionne en ajoutant des bits de redondance à un ensemble de données de 11 bits, formant ainsi un mot de code de 15 bits. Voici une explication succincte de son fonctionnement :

1. Données d'origine : Les données d'origine, constituées de 11 bits, sont prises en entrée pour être encodées à l'aide du code de Hamming (15,11).
2. Bits de redondance : Quatre bits de redondance supplémentaires sont ajoutés aux données d'origine pour former un mot de code de 15 bits. Ces bits de redondance sont insérés à des positions stratégiques, correspondant à des puissances de 2 (positions 1, 2, 4 et 8) dans le mot de code.
3. Calcul des bits de redondance : Les bits de redondance sont calculés en fonction des bits de données d'origine. Chaque bit de redondance est déterminé par un calcul binaire effectué sur un sous-ensemble spécifique de bits de données d'origine. Chaque sous-ensemble est défini par la position du bit de redondance dans le mot de code.
4. Transmission des données encodées : Le mot de code de 15 bits, comprenant les données d'origine et les bits de redondance, est transmis.
5. Détection et correction d'erreurs : Lors de la réception des données, le récepteur effectue des calculs de parité sur les différents sous-ensembles de bits du mot de code pour détecter et localiser les erreurs. Si une erreur est détectée, elle peut être corrigée en utilisant les bits de redondance correcteurs d'erreurs.
6. Extraction des données d'origine : Les bits de données d'origine sont extraits à partir du mot de code en ignorant les bits de redondance, fournissant ainsi les données d'origine sans erreur.

Le code de Hamming (15,11) permet la détection et la correction d'une seule erreur bit dans le mot de code. Grâce à l'ajout de bits de redondance et aux calculs de parité, il permet de renforcer la fiabilité des transmissions de données et de garantir une détection d'erreur plus précise et une correction si nécessaire.

J'ai essayé de suivre la même approche que la dernière, en essayant de trouver une vidéo YouTube, mais je n'ai pas eu autant de chance cette fois. Approfondissant mes recherches,

j'ai trouvé un repo de GitHub sur Hamming (15,11) qui était exactement ce dont j'avais besoin.

J'ai donc fait quelques tests pour voir si l'algorithme fonctionne :

```
#region Hamming Test

int errorPosition = 10;
string codeString = "01010101111";

var code = Helpers.prettyStringToBoolArray(codeString);
var encoded = HammingCode.Encode(code);

Debug.Log(Helpers.boolArrayToPrettyString(code));
Debug.Log(Helpers.boolArrayToPrettyString(encoded));

HammingCode.MixinSingleError(encoded, errorPosition);
Debug.Log(Helpers.boolArrayToPrettyString(encoded));

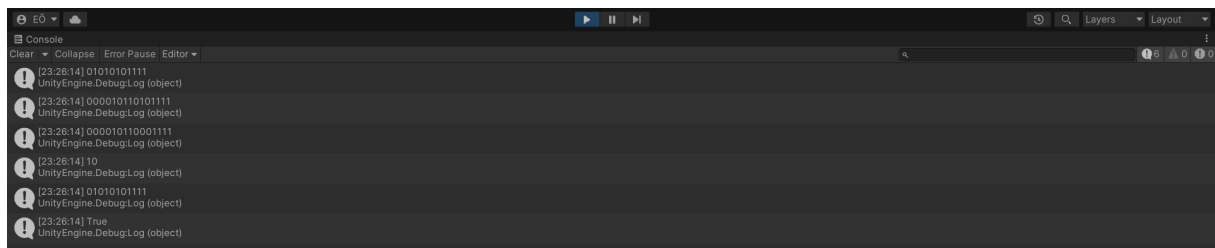
Debug.Log(HammingCode.ErrorSyndrome(encoded));
encoded[errorPosition-1] = !encoded[errorPosition-1];

var decoded = HammingCode.Decode(encoded);
Debug.Log(Helpers.boolArrayToPrettyString(decoded));

Debug.Log(Enumerable.SequenceEqual(code, decoded));

#endregion
```

Voici les résultats :



INTEGRATION DE DEUX ALGORITHMES

À ce point, j'ai pu coder et décoder à la fois dans LZW et Huffman (15,11), mais il était temps de connecter ces deux algorithmes. Le problème était que les structures de données acceptées par ces fonctions étaient différentes, ce qui était un peu difficile à résoudre pour moi. LZW travaillait avec une liste d'entiers, alors que le Huffman travaillait avec des tableaux booléens.

J'ai donc décidé de coder un script chargé de gérer la connexion entre ces deux algorithmes appelé « CodeManager.cs ». Fondamentalement, comme je l'ai expliqué ci-dessus, j'avais besoin de convertir la liste int en tableau bool et vice versa.

Je pensais que ce serait l'endroit idéal pour utiliser ChatGPT, car cela prendrait beaucoup moins de temps pour demander exactement ce dont j'ai besoin au lieu d'essayer de le coder moi-même. ChatGPT m'a donné raison, en fournissant deux méthodes que j'ai exactement demandées :

```

// Function that makes the connection between LZW and Hamming by
transforming the data type.
// Written by ChatGPT, did a little revision myself
public static bool[] ConvertIntToBoolArrayOfLengthEleven(int value)
{
    bool[] boolArray = new bool[11];

    for (int i = 0; i < 11; i++)
    {
        boolArray[i] = ((value >> i) & 1) == 1;
    }

    return boolArray;
}

// Function that makes the connection between LZW and Hamming by
transforming the data type.
// Written by ChatGPT, did a little revision myself
public static int ConvertBoolArrayOfLengthElevenToInt(bool[] boolArray)
{
    int value = 0;

    for (int i = 0; i < 11; i++)
    {
        value |= (boolArray[i] ? 1 : 0) << i;
    }

    return value;
}

```

Et voici comment j'ai tout lié :

```

public String EncodeAndDecode(string text)
{
    // Initial coding algorithm with LZW
    List<int> compressed = LZW.Encode(text);

    // Applying code correction algorithm with Hamming (15,11), first
    converting the codes from LZW to 11 byte binary code.
    foreach (int LZWCode in compressed)
    {
        hammingCodesCoded11.Add(Helpers.ConvertIntToBoolArrayOfLengthEleven(LZWCode
    ));
    }

    // Applying code correction algorithm with Hamming (15,11), then
    transforming the 11 byte code to 15 byte according to Hamming algorithm.
    foreach (bool[] hammingCode11 in hammingCodesCoded11)
    {
        hammingCodes15.Add(HammingCode.Encode(hammingCode11));
    }

    // Decoding the Hamming code back to 11 bytes.
    foreach (bool[] hammingCode15 in hammingCodes15)
    {
        hammingCodesDecoded11.Add(HammingCode.Decode(hammingCode15));
    }

    // Converting 11 byte bool[] to int to decode with LZW.
    foreach (bool[] hammingCodeDecoded11 in hammingCodesDecoded11)
    {

```

```

decompressedIntArray.Add(Helpers.ConvertBoolArrayOfLengthElevenToInt(hammingCodeDecoded11));
    }

    // Decompressing the code by using LZW.
    String decompressedString = LZW.Decode(decompressedIntArray);

    // Return for UIManager.cs
    return decompressedString;
}

```

COMPRESSION AVEC PERTE

À ce point, j'ai pu utiliser LZW et Huffman en tandem, mais il manquait un point dans le projet, à savoir la compression avec perte. Étant donné que les fichiers texte ne sont pas destinés à être compressés avec perte, il n'est pas très logique de le faire, mais je dois le faire pour répondre aux exigences du projet.

Le mieux que je pouvais faire était d'effectuer ces actions pour effectuer une compression avec perte :

- Convertir toutes les lettres en minuscules
- Supprimer les espaces et les ponctuations
- Supprimer les voyelles
- Transformer les lettres similaires.

Une fois de plus, j'ai trouvé cela comme une excellente occasion de demander à ChatGPT au lieu d'essayer de trouver des méthodes et une syntaxe spécifiques sur Internet. Voici les fonctions qu'il m'a donné :

```

// Made by ChatGPT
static string RemoveSpacesAndPunctuations(string text)
{
    return Regex.Replace(text, @"[\s\p{P}]+", "");
}

// Made by ChatGPT
static string RemoveVowels(string text)
{
    return Regex.Replace(text, "[aeioöuüAEÏIOÖUÜ]", "");
}

// Made by ChatGPT
static string ConvertToLowercase(string text)
{
    return text.ToLower();
}

// Made by ChatGPT
static string TransformSimilarLetters(string text)
{
    text = text.Replace("ç", "c");
    text = text.Replace("ş", "s");
    text = text.Replace("ğ", "g");
    // Add more transformation rules if needed
}

```

```
    return text;
}
```

En effectuant toutes ces opérations avant d'appliquer l'algorithme LZW, j'ai fait fonctionner une méthode de compression de texte avec perte.

BILAN

J'avais besoin de calculer le gain, le taux de compression et l'utilisation de la capacité du canal des algorithmes sans perte et avec perte pour le projet. Tout d'abord, j'ai commencé par jeter un coup d'œil rapide sur les définitions formelles de chacun d'eux. Ensuite, j'ai essayé la méthode ChatGPT mais cette fois ça n'a pas très bien fonctionné, j'ai donc dû faire plus de petites touches. Enfin, je l'ai fait fonctionner.

Pour calculer le gain selon la théorie de l'information d'une compression, nous devons comparer la quantité d'informations dans les données d'origine avec la quantité d'informations dans les données compressées. La théorie de l'information traite de la quantification et de la communication de l'information, et elle fournit un cadre pour mesurer la quantité d'information dans un ensemble de données donné. Le gain en théorie de l'information peut être calculé à l'aide de la formule :

Gain = (Informations d'origine - Informations compressées) / Informations d'origine

```
static double CalculateEntropy(List<int> data)
{
    int totalCount = data.Count;
    var counts = data.GroupBy(x => x).ToDictionary(g => g.Key, g =>
g.Count());

    double entropy = 0.0;

    foreach (var count in counts.Values)
    {
        double probability = (double)count / totalCount;
        entropy -= probability * Math.Log(probability, 2);
    }

    return entropy;
}

public static double CalculateGainPercentage(List<int> LZWCode, List<int>
defaultCode)
{
    if (defaultCode == null || LZWCode == null)
        return 0;

    double gain = (CalculateEntropy(LZWCode) -
CalculateEntropy(defaultCode)) / CalculateEntropy(defaultCode) * 100;
    return gain;
}
```

Pour calculer le taux de compression, nous devons comparer la taille des données d'origine avec la taille des données compressées. Le taux de compression représente la réduction de taille obtenue grâce à la compression.

```
public static double CalculateCompressionRatePercentage(List<int> LZWCode,
List<int> defaultCode)
{

```

```
return (1 - ((float)LZWCode.Count / defaultCode.Count)) * 100;
}
```

Pour calculer l'utilisation de la capacité, nous devons comparer l'utilisation réelle de la capacité avec la capacité maximale disponible. L'utilisation est généralement exprimée en pourcentage et indique l'efficacité avec laquelle la capacité est utilisée.

```
public double CalculateChannelCapacityUtilisationPercentage()
{
    // 2048 = 2^11
    return (float)decompressedIntArray.Count / 2048 * 100;
}
```

Je veux aussi expliquer brièvement comment j'ai obtenu le « defaultCode ». J'ai modifié le script LZW de manière à ne vérifier que les valeurs initialement définies et à ne pas chercher d'autres clés possibles à ajouter au dictionnaire. De cette façon, je peux voir combien cela aurait été pris s'il n'y avait pas eu d'algorithme LZW. Voici le code de celui-ci :

```
// Written by me by simplifying the LZW Algorithm.
public static List<int> Encode(String text) {
    int dictSize = 512;
    Dictionary<String, int> dictionary = new Dictionary<string, int>();
    for (int i = 0; i < dictSize; i++) {
        dictionary.Add(((char) i).ToString(), i);
    }
    List<int> result = new List<int>();
    foreach (char character in text.ToCharArray()) {
        if (dictionary.ContainsKey(character.ToString())) {
            result.Add(dictionary[character.ToString()]);
        }
    }
    return result;
}
```

GUI

La principale raison pour laquelle j'ai choisi Unity était la commodité de concevoir une interface graphique. J'ai d'abord commencé par essayer de trouver un asset d'interface utilisateur approprié. J'essayais d'en trouver un avec le thème hacker, où il y aurait des couleurs vertes et sombres mais je n'en ai pas trouvé, alors j'ai opté pour un regard noir et blanc plus basique.

J'ai conçu 4 panneaux principaux et des transitions entre eux :

- Panneau principal
- Panneau sans perte
- Panneau avec perte
- Panneau de résultat

Ensuite, j'ai configuré un script pour gérer l'interface utilisateur nommé « UIManager.cs ». J'ai codé les fonctions nécessaires pour gérer les actions de l'interface utilisateur lorsqu'un bouton est enfoncé. Tous les codes UI sont écrits par moi. Voici le panneau principal :



LZW Compression With Hamming (15,11) Code Correction

Lossless LZW

Lossy LZW

SOURCE

- <https://www.youtube.com/watch?v=1KzUiklae6k>
- <https://gist.github.com/fend25/99347aba1903c881ae48>