La Recherche d'Images Similaires
Projet intégrateur

Partie #2: Concurrence (Go)

uOttawa

CSI2520[A] Paradigmes de programmation

Présenté pour Robert Laganière

Yassine Moumine 300140139

Date: le 17 Mars 2024

Table de Matiere

Table de Figures

Introduction

Dans le monde moderne, la prolifération rapide des images nécessite des outils informatiques avancés pour analyser, rechercher, classer et découvrir des images d'intérêt. Ce projet vise à implémenter une méthode de recherche d'images similaires en utilisant la programmation orientée objet en Java.

Données d'Entrée

La base de données d'images est fournie au format JPG, avec les histogrammes précalculés enregistrés dans des fichiers texte. En outre, 16 images de requête sont fournies au format JPG et PPM. Pour cette partie nous n'utilisons pas le format PPM, mais le format JPEG à l'aide de la méthode `image.Decode(file)`, qui retourne un type image qui sert a avoir les mesure de l'image (width & height).

Objectifs:

- Résoudre le problème de similarité en décodant les histogrammes des images.
- Utiliser la programmation concurrents comme suit:
    - Créer le channel d'histogrammes.
    - Obtenir la liste des images se trouvant dans le répertoire d'images;
    - Subdiviser cette liste en K slices et transmettre chaque slice à une instance de la fonction go computeHistograms.
    - Dans un fil concurrent séparé, ouvrir l'image de requête et calculer son histogramme.
    - Lire le channel d'histogrammes:
        - Lorsqu'un histogramme est reçu, le comparer à l'histogramme de requête.
        - En se basant sur les résultats d'intersection d'histogrammes, maintenir la liste des 5 images les plus similaires.
    - Une fois toutes les images considérées, afficher la liste des 5 images les plus similaires.
    - Fermer tous les channels et assurez-vous que tous les fils soient arrêtés avant que le programme se termine.

Problème

1. Pour cette partie, nous pouvons déjà calculer les valeurs RGB de chaque pixel grâce a la méthode `img.At(x, y).RGBA()`. Cette méthode renvoie des valeurs RGB comprises entre [0, 65535], donc un décalage de 8 est nécessaire afin de réduire ce taux à la l'intervalle [0, 255].

```
red = red >> 8
blue = blue >> 8
green = green >> 8
```

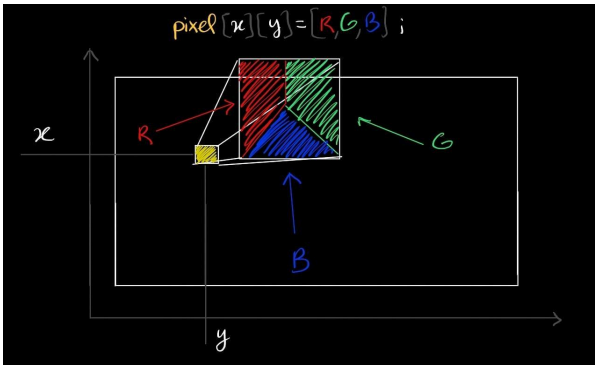

Figure 1: Représentation tabulaire des pixels

2. En addition, nous introduisons de nouvelles expériences dans le but d'optimiser la configuration de notre programme concurrent. Ces expériences comprennent la subdivision de la liste des images, renvoyées par la méthode readDataset(), par différentes valeurs de **K**.

```
func readDataset(folder string) (filenames []string) {
  // ...
}
```

K=1; K=2; K=4; K=16; K=64; K=256; K=1048

Solution Algorithmique

1. L'algorithme propose une méthode de recherche d'images similaires en calculant et comparant les histogrammes couleur. Pour simplifier le décompte des couleurs d'une image, l'histogramme est réduit en limitant l'espace de couleurs. En passant de 8 bits par canal à 3 bits, ceci ramène l'espace à 512 couleurs, facilitant le calcul. Nous aurons donc:

```
red = red >> 8 >> (8 - depth)
blue = blue >> 8 >> (8 - depth)
green = green >> 8 >> (8 - depth)
```

Ensuite, pour normaliser l'histogramme et permettre la comparaison entre images de résolutions différentes, chaque entrée est divisée par le nombre total de pixels, assurant des valeurs comprise entre 0.0 et 1.0. Puis une simple opération d'intersection sera utilisée pour comparer deux histogrammes.

```
hist.NH[i] = float64(hist.H[i]) / float64(totalPixels)
                    // ...
        d += math.Min(query.NH[i], data.NH[i])
```

2. Comme solution pour ces exigences, j'ai créé plusieurs fils, une pour chaque expérience afin de bien démontrer la différence en temps. Bien sûr, ceci exige de la bienveillance en termes d'utilisation et gestion des goroutines et des channels. C'est pourquoi nous utilisons `sync.WaitGroup` afin de bien gérer les goroutines.

En premier, j'ai créé une interface afin de pouvoir créer des sous-slices pour chaque expérience et les tout enregistrer dans une map, ou l'index sera le nom de l'expérience et les valeurs seront les sous-slices appropriés.

```
// interface for ceating sub slices for each experiment (K)
type KSlice interface {
```

```
    CreateKSlice() [][]string
}
```

```
// method for creating the K sub slices appropriate for each experiment
func (e Expo) CreateKSlice() [][]string {

    length := len(e.filenames)
    sub := int(length / e.K)
    var res [][]string

    for i := 0; i < length; i += sub {
        end := i + sub
        if end > length {
            // fix rounding up from length division
            // int (2485 / 2) = 1243 * 2 = 2486 > 2485
            end = length
        }
        res = append(res, e.filenames[i:end])
    }
    return res
}
```

```
// method for creating each experiment, it's sub slices and adding to the
overall map of experiments
func createExperiments(filenames []string) (m map[string][][]string) {

    m = make(map[string][][]string)
    var cs KSlice

    // split filenames into K slices for each experiment and add it to map
m
    // Experiment 1
    // k := 1;
    exp1 := Expo{filenames, 1}
    cs = exp1
    m["exp1"] = cs.CreateKSlice()
```

```
    // Experiement 2
    // k := 2;
    exp2 := Expo{filenames, 2}
    cs = exp2
    m["exp2"] = cs.CreateKSlice()

      // ...
```

Dans mon programme, nous avons des goroutines pour chaque expérience, puis des goroutines pour chaque slice, puis des goroutines pour chaque image. Ceci garantit l'optimisation de notre programme concurrent pour telle tâche.J'ai aussi créé un channel pour chaque expérience `expCh`.

```
//  thread for each experiment - compute dataset histograms and runtime
for each experiment
    for expName, slices := range m {

      // Create a channel for each experiment
        expCh := make(chan Histo)


        // goroutine for each experiment/thread
        go func(slices [][]string, expCh chan<- Histo) {
            // ...
            for _, slice := range slices {

                // goroutine for each slice
                go func(slice []string) {
                        // ...
                        // for each image in a slice
                        for _, e := range imagePath {

                            // ...

                            // goroutine for each image
                            go func(e string) {
                            // ...
```

Résultat Attendu

1. L'objectif est de trouver les 5 images les plus similaires à chaque image de requête, en utilisant l'intersection de leurs histogrammes comme vu précédemment dans la partie object-oriented.

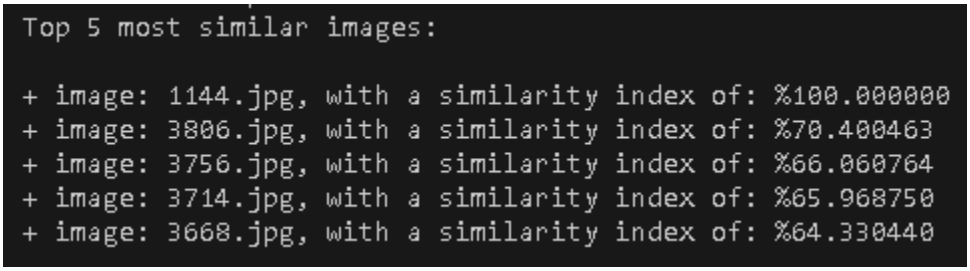

Figure 4: Résultats de sortie

2. Toutefois, l'objectif principal de cette partie est la programmation concurrente en utilisant plusieurs fils. Pour chaque expérience nous avons créé un fils et dedans nous avons créé plusieurs fils pour chaque nombre K de slices correspondant. Donc:

$$7\ Experience\ =\ 7\ Threads$$

Et dans chaque fil

$$Experience\ 1:\ K\ =\ 1:\ 1X\ Thread$$
$$Experience\ 2:\ K\ =\ 2:\ 2X\ Threads$$
$$Experience\ 3:\ K\ =\ 4:\ 4X\ Threads$$
$$\dots$$
$$Experience\ 7:\ K\ =\ 1048:\ 1048X\ Threads$$

Et chaque image dans une slice a ***une instance goroutine***

Afin de déterminer la configuration optimale pour notre programme, nous calculons le temps d'exécution pour chaque expérience.

Les spécifications du système utilisé, pour ces expériences, sont:

- Windows 11 v.23H2
- 11th Gen Intel(R) Core(TM) i5-11400H @ 2.70GHz 2.69 GHz

CPU Specifications	
Total Cores	6
Total Threads	12
Max Turbo Frequency	4.50 GHz
Cache	12 MB Intel® Smart Cache
Bus Speed	8 GT/s
Configurable TDP-up Base Frequency	2.70 GHz
Configurable TDP-up	45 W
Configurable TDP-down Base Frequency	2.20 GHz
Configurable TDP-down	35 W

Figure 2: Spécifications du processeur

Donc voici le graphe qui représente les temps d'exécution par rapport au nombre de fils créés:

Iteration #1

Experiment #	Runtime #1 (in s)
1	4.0129243
2	3.5853029
3	3.470758
4	3.2553889
5	3.2184379
6	3.8744651
7	3.9989282

Iteration #2

Experiment #	Runtime #2 (in s)
1	3.2282732
2	3.1409198
3	2.9848698
4	3.1668569
5	3.1940158
6	3.8712986
7	3.9911556

Iteration #3

Experiment #	Runtime#3 (in s)
1	3.2246424
2	3.5558342
3	3.8690644
4	3.9546957
5	3.3152347
6	3.2255748
7	3.076595

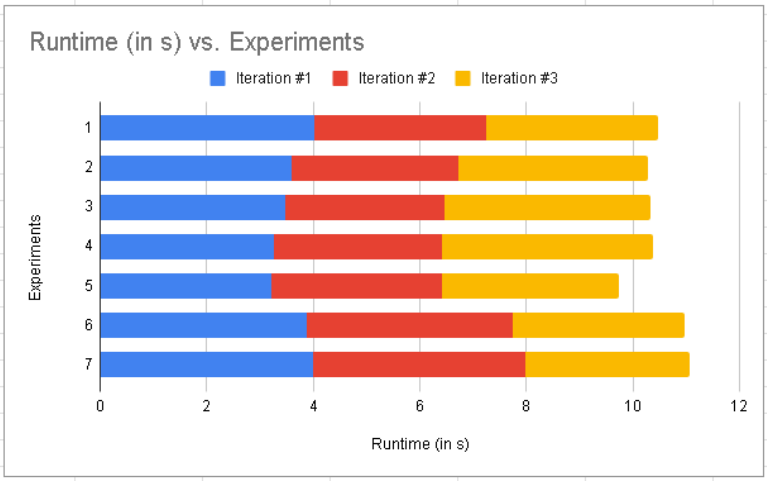

Figure 3: Graph du temps d'exécution par rapport au nombre de fils créés

Nous pouvons donc conclure que le programme performe le mieux avec K=5