TP 5

Max Halford

1 Carnet de bord

Le sujet de base ne comporte pas de difficultés particulières, si ce n'est un travail d'analyse de la page HTML concernée. Ce que l'on a appris dans le TP4 est amplement suffisant pour y arriver. Mon approche a été d'ouvrir la page en question avec un éditeur de texte adéquat (avec un code couleur) pour aisément étudier les blocs html recherchés. Tout au long de ce processus je me suis efforcé de généraliser au maximum les expressions régulières : elles peuvent parfaitement fonctionner sur une page sans pour autant le faire sur une autre. J'ai constaté que c'était la lacune principale des expressions régulières pour le "parsing" de code HTML. Une des difficultés à laquelle je ne connaissais rien était l'encodage des pages. De ce que j'ai compris, il se trouve que la plupart des pages sont encodées en UTF-8, et que Perl utilise par défaut les caractères ASCII (256 possibilités), il faut donc utiliser le package `Encode`, la commande `use open qw/:std :utf8/` se révèle très pratique pour dire à Perl de décoder les pages ouvertes en UTF-8. Cependant cette commande oblige à encoder en UTF-8 `print()`. Bref, en tâtonnant on y arrive assez bien, pas besoin d'être ceinture noire en encodage. J'aimerais faire remarquer que l'expression régulière pour récupérer les sources est assez douteuse et n'est pas du tout généraliste.

Pour ce qui est des **fonctionnalités optionnelles** :

• Lire une URL directement est même plus simple que de sauvegarder une page HTML manuellement et de la lire : le module `LWP::Simple` et sa fonction `get()` est imbattable.
• Pour traiter une liste d'URL il suffit de maîtriser la procedure `open` (que ce soit en écriture ou en lecture, elle est cruciale dans la retransmission de données ou d'écritures de script). On peut remarquer qu'ouvrir un fichier avec un itérateur peut apporter quelques soucis puisque on ne peut le parcourir qu'une fois (par définition).

• Créer un module n'est pas chose compliquée. Par choix je n'ai pas tout mis mon module : par exemple j'aurais pu y écrire une fonction pour afficher les données récupérées dans le terminal, mais le nombre d'arguments serait assez large et ça ne valait pas la peine de s'embêter (mettre plusieurs listes en arguments peut s'avérer assez crispant).
• Une fois qu'on a nos données, on peut simplement les stocker dans un hash en utilisant l'URL comme clé. J'ai cependant décidé de stocker le hash sous forme JSON car c'est assez commun et utilisable par une pléthore d'autres programmes, de plus le module `JSON` rend le paradigme assez facile.
• J'ai décidé d'utiliser quatre tableaux pour stocker mes données : une table "Pages" pour stocker les informations basiques (`id` qui est un nombre qui s'incrémente au fil des pages parcourues; `url`, `titre`, `dateEcriture` sont assez explicites), et trois tables contenant respectivement les sommaires, les catégories et les sources. Le schéma des trois dernières tables est le même, il est, somme toute, assez banale, mais fait l'affaire. En effet SQL n'est pas très souple pour stocker des listes puisque chaque élément doit être atomique. Avec plus de temps et de la documentation j'aurais voulu créer une petite base sous MongoDB qui aurait été bien plus propre. En tout le script SQL marche, pour insérer les données il suffit de parcourir les données stockées précedemment dans un hash.
• J'ai fait en sorte que l'utilisateur puisse décider de l'exécution du script avec un système de "flags" : l'utilisateur peut accéder à une liste d'options, avec différentes commandes il peut activer/désactiver des éléments du script via l'affectation d'1 ou 0 à des valeurs (les flags). Ensuite durant l'exécution du script les éléments sont enclenchés seulement si leur flag respectif vaut 1. Par défaut tout le script se lance. C'est particulièrement pratique pour réduire le temps d'exécution quand on a une longue liste d'URL à traiter.

Pour ce qui est des **fonctionnalités personnelles** :

• Bien que les résumés de chaque page s'affichent sur le terminal et soient stockés (grâce au script SQL), une page HTML contenant le résumé d'une page (pour chaque page) est agréable, ne serait-ce que pour voir que notre programme renvoie les bonnes données. La génération de ces pages est relativement simple si on a réussi à créer le script SQL, il faut seulement bien écrire les balises HTML et ne pas oublier `<meta charset=ÜTF-8/̈>` pour que le navigateur web qui va lire les pages sache décoder les caractères qu'on lui donne.
• On peut utiliser ce qu'on a fait dans le TP28 : stocker les liens que contient la page qu'on parcourt. A partir de là on peut faire beaucoup de choses, en

effet on peut rendre le programme récursif et faire ce qu'on fait pour une page sur toutes les pages référées par les liens contenus dans ladite page. Cependant pour que ceci fonctionne il faudrait changer quelques lignes du code, par exemple les données "écrites" dans des fichiers ne doivent pas écraser les données existantes. De plus notre script est conçu pour fonctionner sur des pages d'uncertain type (wikinews), rien ne nous assure de sa polyvalence! C'est d'ailleurs cette remarque qui me fait me dire que les expressions régulières ne sont pas faites pour ce qu'on fait.
• Il est très pratique d'analyser le contenu des pages et notamment les mots utilisés, en effet on pourrait alors catégoriser les pages selon leurs thèmes ou bien utiliser les mots-clés pour créer des moteurs de recherche. En tout cas, après avoir retiré le code HTML, on peut facilement calculer les occurences de chaque mot (cf. TP3).
• Enfin j'ai décidé de donner la possibilité au lecteur de chercher la présence (et si oui le nombre d'occurences) d'un mot dans toutes les pages analysées, c'est assez simple puisque on les a stockés avec la fonctionnalité précédente.

Deux autres fonctionnalités pourraient être le stockage des images de chaque page et la génération d'un *wordcloud* des mots de chaque page. En vain j'ai tenté d'implémanter la dernière fonctionnalité, malheureusement les modules concernés sont obsolètes.

2 Le dossier

Une fois que le script est lancé une première fois `Project_Max.pl` des fichiers sont créés :
• "Informations" contient les résumés de chaque page en HTML.
• "JSON" contient les hashs stockants les résumés de chaque page.
• "Liens" contient (pour chaque page) une liste de liens trouvés.
• "Occurences" contient les occurences de mots pour chaque page.
• "bdd.sql" est explicite.
• "liens.txt" contient les liens de départ.
• "maxhtmlparsing" est le module (à l'utilisateur de le placer dans le bon répertoire, par exemple j'ai utilisé `sudo cp home/max/Documents/Courses/L3/Recherche dÍnformations/TP 5/maxhtmlparsing.pm usr/lib/perl5/` sous Linux.)

3 Le script pas à pas

Le script a besoin de certains modules qui ne sont pas inclus dans le Perl de base, notamment `JSON`, `Switch`, `Term::Clui` et `Term::Clui::FileSelect`.

Lignes 16 à 102 : Premièrement on donne à l'utilisateur plusieurs choix quant à l'exécution du script, mon conseil est de ne rien changer au départ (donc d'appuyer directement sur `CTRL+D`) et de voir les résultats, si le script est lent lorsque le nombre de pages augmente l'utilisateur peut désactiver des fonctionnalités. Il faut bien faire attention à la cohérence de l'exécution, par exemple on ne peut pas chercher de mots si les occurences de mots n'ont pas été stockés.

Lignes 107 à 122 : Lire le fichier "liens.txt" est simple, tant que l'on sait où aller. Le module `Term::Clui::FileSelect` permet de demander à l'utilisateur d'indiquer où se situe le fichier texte contenant les liens à analyser. Avec l'expression régulière `/(.+).*/` appliquée au chemin dudit fichier on récupère le chemin vers le dossier contenant le projet! On pourra ensuite utiliser ce chemin dans la suite du script. Je n'ai pas inclus cette possibilité dans la liste des fonctionnalités optionnelles, de fait elle me paraît nécessaire. En effet ce serait vraiment embêtant de modifier les chemins à la main pour l'utilisateur.

Lignes 124 à 161 : On a maintenant une liste de liens. On la parcourt et on utilise la fonction `get()` pour récupérer le contenu en HTML. On le met sur une seule ligne. On en extrait le titre, la date, le sommaire (liste), les catégories (liste), les sources (liste), les liens (hash) et les occurences de mots (hash) grâce aux fonctions du module `maxhtmlparsing`. Pour les liens et les occurences on vérifie que l'utilisateur a désactivé ou non les flags correspondants. De plus les liens et les occurences sont stockés en utilisant `open()` en écriture. Pour les occurences on vérifiera le body et non pas l'integralité de la page, pour éviter du superflu.

Lignes 165 à 177 : Si l'utilisateur le décide les données extraites sont affichées.

Lignes 180 à 193 : Cette partie concerne le stockage du hash contenant les résumés des pages sous forme de fichier JSON, c'est assez explicite.

Lignes 197 à 276 : Ensuite il s'agit de créer des tables avec un schéma

relationnel et d'insérer nos données dans ces tables avec le language SQL. Les tables ne sont pas compliquées, il suffit de ne pas faire d'erreur de syntaxe. Pour stocker les données on utilise le hash contenant les résumés, il est à noter que ce hash comporte des hashs, donc il ne faut pas se perdre dans les appels. En effet `@$data$keySommaire` n'est pas très intuitif et demande quelques essais.

Lignes 281 à 318 : Il s'agit maintenant de stocker les résumés sous forme de pages HTML. Si on a réussi à créer un script SQL on n'aura aucun souci à faire cela, à condition que l'on possède des notions d'HTML. Il faut bien préciser au début du script que la page doit être lue en tenant compte de l'encodage en UTF-8.

Lignes 323 à 347 : Enfin on implémante une fonction de recherche. Il faut utiliser la procédure `opendir` pour ouvrir le dossier contenant les occurences de mots pour chaque page. On regarde chaque fichier, on regarde pour chaque ligne si le mot contient le texte qu'on cherche (c'est plus flexible que le mot exact), si oui on regarde de l'autre côté de la flèche (il ne faut pas oublier que c'est la représentation d'un hash) et on somme au total.

4 Question bonus

Après avoir pris un peu de recul, je me suis rendu compte que si je voulais généraliser mon programme, les expressions régulières deviendraient monstrueuses : il y aurait une infinité de cas particuliers. De plus, notre nomenclature (titre, date, sources etc.) est subjective, il vaudrait mieux étudier la structure du code HTML (étude des balises). Ma réponse est donc non, ce n'est pas la meilleure façon de parser un fichier HTML.
Il se trouve que quelques modules de qualité ont été écrits en Perl, notamment `HTML` et `Local::SiteRobot`. En effet, dans le premier cas, avec l'utilisation de "tags" il est possible d'extraire des données bien plus facilement qu'avec des expressions régulières.
On peut trouver beaucoup de "web crawlers" écrits en Perl sur le web, par exemple : `http://www.analyticbridge.com/group/codesnippets/forum/topics/simple-perl-code-to-visit-web` et `http://blogs.perl.org/users/stas/2013/01/web-scraping-with-modern-perl-part-1.html`.