Cours de Data Mining – Exploration du web (web crawling)

A. Dragut

Univ. Aix-Marseille, IUT d'Aix-en-Provence

Plan du cours

- Les limitations des systèmes de bases de données $\Longrightarrow$ Motivation pour les systèmes avec de la recherche d'information (en anglais, "Information Retrieval")
- Recherche d'information
  - Indexation
  - Mesures de Similarité
- Recherche sur le Web
  - Les algorithmes PageRank et HITS

Les bases de données (BD) comparées à la recherche d'information (RI)

- Format des données :
  - BD : données structurées. Sémantique claire, basée sur un modèle formel
  - RI : non-structurées – texte libre
- Requêtes :
  - BD : formelles (par exemple, SQL)
  - IR : souvent – langues naturelles (recherche de mots-clé)
- Résultats :
  - BD : résultat exact
  - IR : résultats pertinents, parfois sans rapport

La Recherche d'Information RI

- But : trouver les documents les plus pertinents pour une requête donnée
- Notions :
  - collections de documents
  - requête (ce dont l'utilisateur a besoin comme information)
  - notion de pertinence

Quels types d'information

- text (documents)
- XML et documents structurés
- images
- audio (effets, chansons)
- video
- code source
- applications/services web

Types d'informations souhaitées

- rétrospective
  - « rechercher le passé »
  - requêtes différentes dans une collection statique de documents
- prospectives (filtrage)
  - « rechercher les tendances »
  - requête statique dans une collection dynamique de documents
  - dépendence temporelle

Le fonctionnement d'un moteur de recherche. Étapes :

L'exploration ou crawl faite par un robot d'indexation qui parcours récursivement tous les hyperliens qu'il trouve en récupérant les ressources intéressantes. L'exploration est lancée depuis une ressource pivot, comme une page d'annuaire web.

L'indexation des ressources récupérées extraire les mots considérés comme significatifs du corpus de documents à explorer. Les mots extraits sont enregistrés dans une BD. Un dictionnaire inverse ou l'index TF-IDF permet de retrouver rapidement dans quel document se situe un terme significatif donné.

La recherche un algorithme indentifie dans le corpus documentaire (en utilisant l'index), les documents qui correspondent le mieux aux mots contenus dans la requête.L es résultats des recherches sont donnés par ordre de pertinence supposée.

Robot d'indexation : *crawler, automatic indexer, bot, Web spider, Web robots*

- Comment obtenir tous ce qu'il y a à indexer ?
  - Partir de sites connus ou d'un annuaire
  - Suivre les liens depuis chaque site
  - Répéter le processus
- **GoogleBot – soumission de nouvelles URL :**

  avec des listings RSS (Really Simple Syndication) un fichier texte au format XML utilisé pour extraire d'un site web du contenu régulièrement mis à jour.

  directe dans un formulaire de plan de site fichiers sitemaps

Robot d'indexation – fonctionnement

- initialisation de la queue des URLs avec les URLs de départ
- tant qu'on peut encore visiter
  - prendre l'URL disponible suivante depuis la queue
  - télécharger le contenu et marquer l'URL comme visitée
  - extraire les hyperliens depuis le document nouvellement téléchargé et les rajouter dans la queue, s'ils satisfont les critères nécessaires
  - réévaluer les conditions pour continuer à visiter des sites (profondeur maximale, nombre maximal de documents récupérés, temps maximal d'exécution, queue d'URLs vide, etc.
  - attendre « un peu »avant de continuer (pour ne pas « assommer »le serveur)

Recherche basé sur le contenu

contenu pris en compte

- **meta-données** balises de description des pages qui renseignent le robot => ne sont plus trop utilisées
- **liens**
- **contenu txt** donné par un analyseur de pdf, html, .doc,...

contenu ignoré

- **robots.txt** un fichier demandant aux robots de ne pas indexer les pages mentionées.
- **directive NOFOLLOW** utilisée dans les les blogs pour les commentaires-spam
  <A HREF='...' REL='NOFOLLOW'>anchor text</A>
- **pages situés en profondeur** de plus de trois niveaux
- **pages dynamiques** souvent pas plus d'un niveau. Elles sont indexés plus lentement que ceux statiques.

"bot spoofer" Pour aider les dvéloppeurs web : un logiciel qui imite un robot d'indexation et voit les pages d'un site comme le robot.

Google calcule pour chaque page un score de crawl $=>$
niveaux de crawl de GoogleBot :

- **couche de base** : la plupart des pages du web. Elles sont crawlées avec une fréquence liée à la fréquence de mise à jour du contenu ainsi qu'à leur PageRank.
- **couche quotidienne** : un petit nombre de pages crawlées de façon quotidienne.
- **couche temps réel** : un nombre très petit de pages crawlées avec une fréquence de l'ordre de la minute, l'heure,... (ex. l'actualité)

Le robot récupère une page, son contenu et les liens => analysés

- l'analyse du contenu d'une page :

  calcul des signatures une à partir de l'URL de la page et une à partir du contenu de la page

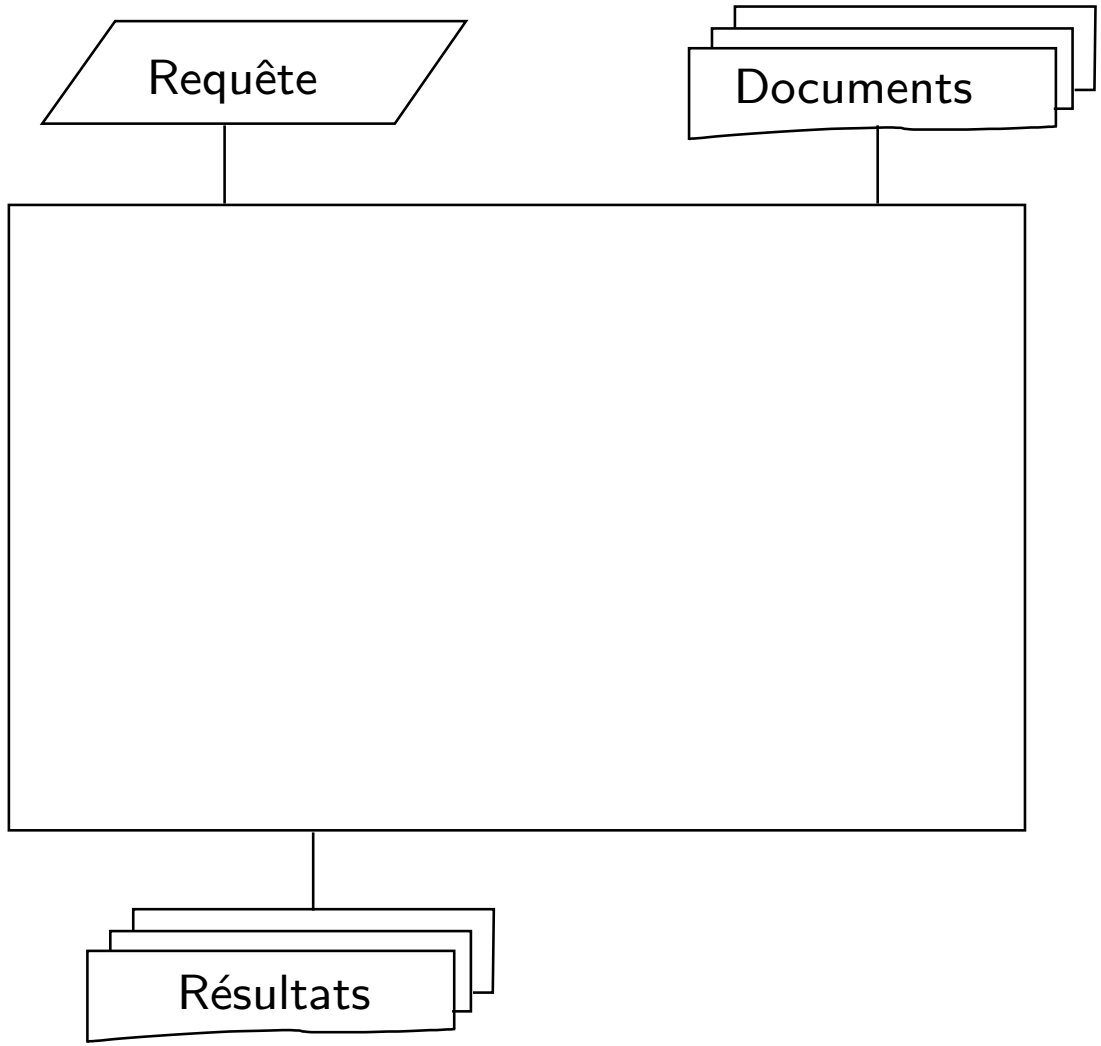
  détection de contenu dupliqué au niveau d'une page et d'un site

- l'analyse du texte des liens et du texte autour des liens

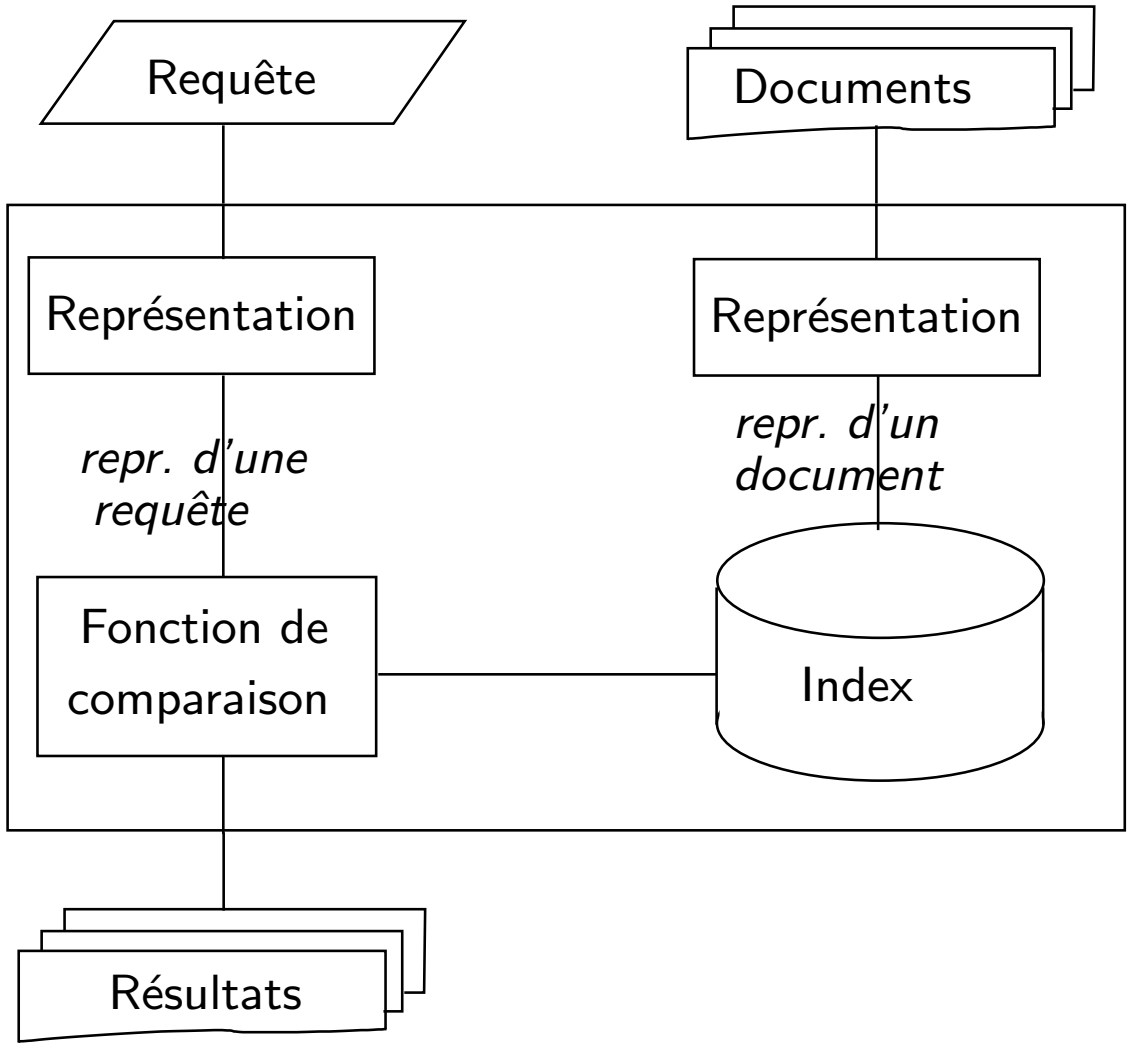
Recherche basée sur le contenu, modèle de l'espace vectoriel

- Les requêtes et les documents sont représentés en tant que vecteurs de termes de l'index
- Indexer les documents dans la collection (offline)
- Traiter chaque requête
- La similarité est calculée entre deux vecteurs
  - les documents (réponses) : ordonnés selon la similarité (avec la requête)

Indexation et Recherche : La boîte noire "RI"



Indexation et Recherche : La boîte noire "RI"



Problème de la RI

Recherche d'information

Auteur

concepts

concepts

Termes de la recherche

$\Longleftrightarrow$

sont-ils pareils ?

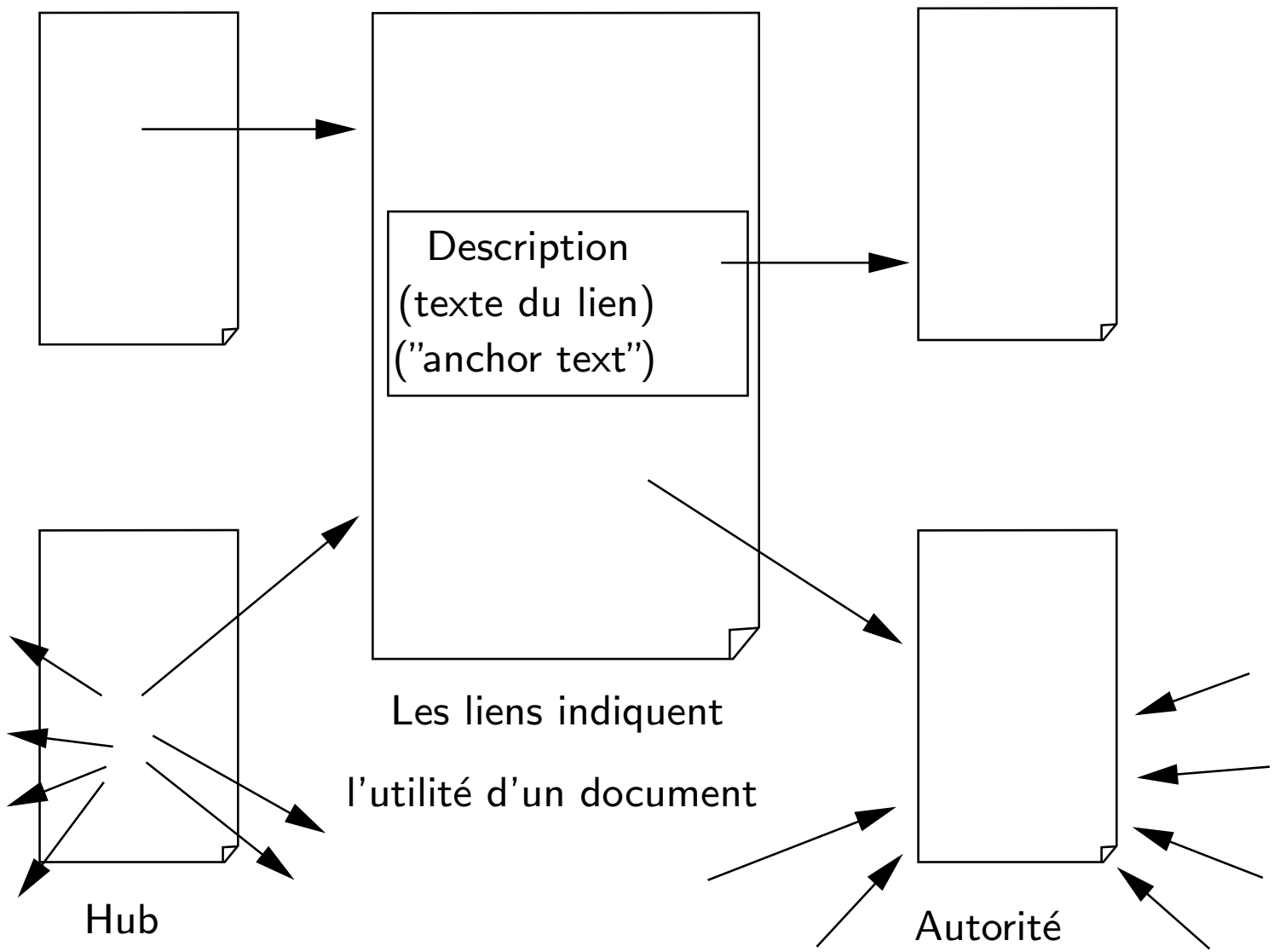
Termes du document

- *L'approche ensembliste* classique BD :
  - l'ensemble des documents s'obtient par une série d'opérations ($\cup$, $\cap$, passage au complémentaire, le langage de requêtes SQL)
- *L'approche algébrique* (ou vectorielle) :
  - les documents et les requêtes font partie d'un même espace vectoriel
- *L'approche probabiliste* :
  - essaie de modéliser la notion de pertinence

Amélioration de la recherche basé sur le contenu – Google

- Les caractéristiques clé de Google
  - Utilisation des indexes basés sur le contenu
    - Indexes peuvent être partitionnés sur plusieurs machines
      ou bien
    - les données sont dupliquées sur les machines et les requêtes sont distribuées parmi celles-ci
  - PageRank
    - ordonner les documents trouvés, en utilisant les liens entre pages comme indice de pertinence

Liens entre les documents



Que nous dit un lien ?

Liens internes entre deux pages différentes d'un même site

Liens externes un lien pointant depuis un site vers des pages quelconques d'un autre site.

- sortant Ils aident les moteurs de recherche à mieux comprendre le contenu du site vers le quel votre lien pointe.
- entrant les obtenir d'un annuaire, par négociation

Les liens sortant d'un site – règles :

- peu de liens externes sortant (environ 5), sinon Si le site est considér comme étant un annuaire.
- doivent être en rapport avec le thème du site ou de la page qui héberge le lien

La recherche basé sur le liens : PageRank

Notion d'« autorité »

Autorité : page référencée par beaucoup d'autres pages

L'intuition : des sites de moins bonne qualité n'ont pas autant de sites qui pointent vers eux

La recherche basé sur le liens : HITS

Notion de « Hub » et « autorité »

Autorité : page référencée par beaucoup d'autres pages

Hub : Page contenant des listes de bonnes références

L'intuition : des sites de moins bonne qualité n'ont pas autant de sites de bonne qualité qui pointent vers eux

SimpleCrawler

Une instance **CrawlerURL**– URL parcourue par le crawler, pour chaque document retrouvé elle est crée dans **SimpleCrawler**

- **: :computeURL()** : Crée l'objet URL (en Java) à partir de la chaîne de caractères.
- **: :getURL()** : Rend l'objet URL (en Java) crée par : :computeURL()), utilisée dans `::setRawContent()`
- **: :getDepth()** : Rend le niveau de profondeur courant (limiter le parcours), utilisée dans `::addUrlsToUrlQueue()`
- **: :isAllowedToVisit()** : Rend le booléen qui dit si cet URL est permise pour aller chercher le contenu du document. Appelée dans `SimpleCrawler::doWeHavePermissionToVisit()`
- **: :isVisited()** : Rend le booléen qui dit si on est déjà passé par là. Appelée par `CrawlerURL::isUrlAlreadyVisited()`.
- **: :getUrlString()** : Utilisée dans `SimpleCrawler`, pour obtenir la chaîne de caractères qui représente l'URL (pour gérér leur liste, les afficher, etc.).
- **: :toString()** : Surcharge de la méthode Java générique.

Document – Parser JSOUP : Parse-document-from-string

public class CrawlerUrl {//donnees et méthodes pour le contenu telechargé depuis l'URL}

1. http ://jsoup.org/cookbook/input/parse-document-from-string
2. **private Document htmlJsoupDoc ;**

Document : Elements + TextNodes + nodes

1. **Jsoup.parse(String html, String baseUri)**
2. public static Document parse(String html,String baseUri)
3. **html** - HTML to parse ; **baseUri** - base URI du document pour résoudre les liens relatifs.
4. **Throws :** MalformedURLException -l'URL demanddé n'est pas HTTP ou HTTPS URL
   - **htmlJsoupDoc = Jsoup . parse(htmlText,baseURL) ;**

public CrawlerUrl(String urlString, int depth)
public void setRawContent(String htmlText)
Problème :

- Input : string JAVA contenant un HTML
- analyser le HTML pour retrouver son contenu, vérifier/modifier
- l'analyseur manipule http ://jsoup.org :
  - unclosed tags (e.g. Blah ⇒ Blah)
  - implicit tags (e.g. <td>Table data</td> ⇒ <table><tr><td>)
  - créant la structure de document HTML : head +title+ body.(text())

Document : Elements + TextNodes + nodes

- méthodes avec des selecteurs CSS ou similaires à jquery :
  Element.select(String selector), Elements.select(String selector)

tagname : ". title()"

- **private String title ; title = htmlJsoupDoc . title() ;**

attribute : "[href]"

- **Elements hrefJsoupLinks = htmlJsoupDoc . select("a[href]") ;**

- attributs qui commençent [attr^=value], finissent [attr$=value], contiennent [attr*=value], par exemple [href*=/path/]
  - **Elements ImagesPNGs = htmlJsoupDoc .select("img[src$=.png]") ;**

Retrouver les données (les attributs, le texte, et le HTML d'extrait) stockées dans **Elements** après l'analyse d'un document **Document htmlJsoupDoc**

- pour obtenir la valeur d'un attribut, emploient la mèthode de Node.attr (clef de corde)
- pour le texte sur un élément : Element.text ()
- **private String niceText ; niceText = htmlJsoupDoc . body() . text() ;**
- pour le HTML, l'utilisation Element.html (), ou Node.outerHtml ()

```
Elements hrefJsoupLinks = htmlJsoupDoc . select("a[href]");
        for (Element link : hrefJsoupLinks) {
    String thisLink = link.attr("abs:href");
    if(thisLink . startsWith("http://")) {
System.out.println("JSOUP Found: " + thisLink);
linkList . add(thisLink);
    }
}
```

`SimpleCrawler` – parcourir le web

Problème :

- aller sur le web retrouver un document HTTP (comme votre browser préféré)
- récupérer depuis ce document les liens pointant vers d'autres documents
- aller retrouver ces documents également
- continuer tout ceci tout en
  - gardant trace du "trajet" parcouru, pour ne pas boucler (par exemple si D1→D2→D3→D1)
  - respectant les éventuelles interdictions posées par les administrateurs des sites distants (fichier `robots.txt`)

1. `httpclient` – retrouve des documents depuis des serveurs HTTP – on s'en crée un objet et `execute()` des requêtes (classe `HttpGet`)
2. `Queue<CrawlerUrl> urlQueue` – une liste dans laquelle on rajoute au fur et à mesure les liens des doc. téléchargés
3. `visitedUrls` – une table de hachage pour noter le parcours au fur et à mesure : on y rajoute les URLs visités, un par un, et on la consulte pour savoir si on est déjà passé par là
4. `sitePermissions` – pour chaque site distant, la liste des pages à ne pas explorer, donnée dans le fichier distant `robots.txt` de ce site-là

`SimpleCrawler` – parcourir le web

Méthode pour le parcours :

- dans `SimpleCrawler::crawl()` on fait ceci :
  - on initialise la liste `urlQueue` avec l'url initiale
  - tant que la liste n'est pas vide et qu'on n'est pas au max des docs vus
    1. on *extrait* l'URL-élément `U` qui est en tête d'`urlQueue`,
    2. on le valide (`U` ne doit pas déjà avoir été visité, etc.)
    3. on récupère depuis le web le doc `D` pointé par `U`, utilisant `httpclient.execute()`
    4. on marque `U` comme ayant été visité
    5. on traite `D` (analyse avec `Jsoup::parse()`, obtention liens)
    6. on sauvegarde localement le texte de `D`
    7. on *insère* les liens obtenus depuis `D` dans `urlQueue`
- Si au pas 2 `U` est invalidée, on itère la boucle
- Les pas 3,4,5 sont dans `SimpleCrawler::getContent(CrawlerUrl url)`
- Le pas 4 : appelle la méthode de `SimpleCrawler` qui met `U` dans `SimpleCrawler::visitedUrls` et qui appelle ensuite pour `U` la méthode `CrawlerUrl::setIsVisited()`
- Le pas 5 : appelle (avec le texte `D` en paramètre) la méthode de `CrawlerUrl` qui appelle `Jsoup::parse()` dessus et récupère ensuite le titre, le corps, les liens, etc.

Exemple de parcours :

- Supposons que notre `urlQueue` contient ceci : $U_1, U_2, \ldots, U_n$
- Faisons tourner la boucle de `SimpleCrawler::crawl()`
  - on extrait U_1 – donc `urlQueue` contient maintenant $U_2, U_3, \ldots, U_n$
  - on valide U_1 – supposons qu'elle est bien valide
  - on récupère depuis le web le doc D_1 pointé par U_1
  - U_1 est mise dans la table de hachage `SimpleCrawler::visitedUrls` et marquée comme visitée avec `CrawlerUrl::setIsVisited()`
  - on traite D_1 obtenant ses liens sortants $U_1^D, U_2^D, \ldots, U_k^D$
  - on sauvegarde D_1 localement
  - on insère ces liens dans la liste, donc `urlQueue` contient maintenant $U_2, U_3, \ldots, U_n, U_1^D, U_2^D, \ldots, U_k^D$
  - on itère
  - on extrait ainsi U_2 – donc `urlQueue` contient maintenant $U_3, \ldots, U_n, U_1^D, U_2^D, \ldots, U_k^D$
  - on valide U_2,
  - si elle est valide, on récupère D_2
  - etc.

- donnée-membre `URL url`, utilisant la classe `Java URL`.
  - entité de base pour gérér les noms d'hôte, le port, l'adresse elle-même, les tests pour savoir si on a le droit d'y aller, etc.
- donnée-membre booléenne `isVisited`
  - mise à vrai par `CrawlerUrl::setIsVisited()`
  - sa valeur est rendue par `CrawlerUrl::isVisited()`
- donnée-membre `List<String> linkList`
  - contient les liens sortant du document pointé par cette URL et téléchargé (dans `SimpleCrawler`)
  - rendue par `CrawlerUrl::getLinks()` (par exemple appelée dans `SimpleCrawler::addUrlsToUrlQueue()`)
  - peuplée suite à l'analyse du texte d'un document avec `CrawlerUrl::setRawContent(String htmlText)`
    - reçoit le texte `HTML` tel qu'il vient d'être téléchargé
    - l'analyse avec `Jsoup::parse()`
    - en extrait le titre avec `Jsoup::title()` et le corps avec `Jsoup::body()`
    - en extrait les liens dans une boucle parcourant les éléments HTML `href`
    - met ces liens dans la donnée-membre `linkList`

JavaDoc – mot clé **this**

```
public class Point {
    public int x = 0;
    public int y = 0;

    //constructor
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

la liste de URL à parcourir

Queue<CrawlerUrl>= new LinkedList<CrawlerUrl>

```
Interface Collection<E> // l'interface "racine" ------------------//
    boolean isEmpty()   // renvoie VRAI si la collection
                        // ne contient aucun element
    int     size()      // renvoie le nombre d'elements de
                        // la collection

Interface Queue<E>      // herite de Collection<E> -----------------//
    boolean add(E e)    // rajoute l'element e a la queue, renvoiant
                        // VRAI si succes
    E remove()          // retrouve l'element qui est en tete de la
                        // queue, le rend en retour, et l'enleve
                        // egalement depuis la queue

class LinkedList        // implementation de Interface List --------//
                        // qui a son tour implemente Collection
```

- **private Map<String, CrawlerUrl> visitedUrls**
- **private Map<String, Collection<String> > sitePermissions**

```
Interface Map<K,V> // associe valeurs V a des cles K (uniques) ------//
                   // mieux eviter de modifier les cles une fois dedans
                   // permet d'enumerer les cles, les valeurs ou encore
                   // les paires (cle, valeur)
 int     size()    // renvoie le nombre total de paires (cle, valeur)
 boolean containsKey(Object cle) // VRAI si cette cle est dans la 'map'
                                 // i.e. la map a une paire (cle,valeur
 V put(K cle, V valeur) // associe valeur a cle dans la map
 V get(Object cle)  // renvoie la valeur associee a cle si elle existe
                    // dans la map, et sinon elle renvoie null
```

public SimpleCrawler(...)

- **this.visitedUrls=new HashMap<String,CrawlerUrl>();**
- **this.sitePermissions=new HashMap<String,Collection<String> >();**

```
static interface Map.Entry<K,V>    // classe declaree dans la classe Ma
                                   // pour representer les paires
    Set<Map.Entry<K,V>> entrySet() // renvoie l'ensemble des paires

class HashMap<K,V>  // implementation de Interface Map ---------------/
                    // en tant que table de hachage

                    // on l'utilise pour instancier nos Map d'URLs
                    // visites, respectivement de permissions de sites
```

```
public interface Set<E> // collection d'elements sans duplicata  ---//
       extends   Collection<E>

Iterator<E> iterator()  // renvoie un iterateur -- enumerer les elements


public interface Iterator<E> // un iterateur pour une collection -- //

 boolean   hasNext()         // VRAI s'il existe un element suivant

  E        next()            // renvoie l'element suivant


// Donc par exemple, apres avoir rempli une (hash)map, on peut recuperer
// ses entrees avec entrySet(), se prendre un iterateur dessus et les
// enumerer une par une pour les examiner, les traiter, etc.
```

Documentation Java – Entrées(-sorties)

```
public abstract class InputStream  // la superclasse de toutes ------//
       extends        Object       // les classes representant des
       implements     Closeable    // flots d'entree d'octets

public class          FileInputStream // utilisee pour lire un fichier
       extends        InputStream

FileInputStream(String cheminFichier) // constructeur

public final class Scanner            // analyseur lexical simple avec
       extends     Object             // lequel nous lirons depuis des
       implements  Iterator<String>   // fichiers (e.g. ligne par ligne)

Scanner(InputStream source)           // constructeur -- nous allons
                                      // l'utiliser pour lire depuis un
                                      // FileInputStream

 boolean hasNextLine()                // VRAI s'il y a une ligne dispo

 String  nextLine()                   // avance le scanner avec toute
                                      // une ligne et la renvoie

 void   close()                       // ferme le scanner
```

Documentation Java – (Entrées-)sorties

```
public abstract class Writer       // classe abstraite pour ecrire --//
       extends        Object       // sur des flots de caracteres
       implements     Appendable, Closeable, Flushable
public void write(String str)      // ecrit la chaine de characteres str
public Writer append(CharSequence csq) //ecrit en rajout la chaine de chrs
    // (notez que la classe String implemente l'interface CharSequence)

public class   OutputStreamWriter  // "pont" pour encoder les caracteres
       extends Writer              // correctement (Unicode, etc.)

public class   FileWriter          // classe pour ecrire "simplement"
       extends OutputStreamWriter  // dans des fichiers texte

FileWriter(String cheminFichier)   // constructeur

public class   BufferedWriter      // pour ecrire du texte sur un flot
       extends Writer              // de sortie de caracteres

BufferedWriter(Writer out)         // constructeur -- on va l'utiliser
                                   // avec des FileWriter pour ecrire
void   flush()                     // vide les tampons
void   close()                     // ferme le BufferedWriter
```

Documentation Java – Chaînes de caractères

```
public final class StringBuilder       // chaine de caracteres -------//
       extends     Object              // modifiable
       implements  Serializable, CharSequence

StringBuilder   append(CharSequence s) // rajoute la chaine de caracteres

String          toString()             // renvoie le 'String' contenant
                                       // la chaine de caracteres "constru


public final class String              // chaine de caracteres constantes
       extends     Object
       implements  Serializable, Comparable<String>, CharSequence
  int  length()                      // renvoie la longueur de la chaine
  String[] split(String regexp) // decompose le String en lexemes selon
                                // l'expression rationnelle regexp, par
                                // exemple :
 String    a = "aaa    bbb  ccc";
 String[]  t = a.split("\\s+"); // fera t[0]="aaa" t[1]="bbb",
                                //      t[2]="ccc"
 // la concatenation de String se fait avec l'operateur +:
     String    c = "aa" + " " + "bb";
```

```
public final class URL          // represente une URL --------------//
       extends     Object
       implements  Serializable

URL(String chaineDeChrUrl)      // constructeur

 String       getHost()         // extrait la partie hote de l'url

 String       getPath()         // extrait la partie chemin de l'url

 int          getPort()         // extrait le numero de port de l'url

 int          hashCode()        // cree un entier pour la
                                // table de hachage
```