IFT1015 - Travail Pratique 2

Aléapédia

Concepts appliqués : boucles, tableaux, fonctions, décomposition fonctionnelle, algorithmie et programmation web

---

ALÉAPÉDIA

Figure 1: Aléapédia - L'Encyclopédie Générée Aléatoirement

1 Contexte

Pour ce deuxième travail pratique, vous aurez deux tâches :

1. Écrire une librairie pour `node.js` qui génère du texte aléatoire à première vue plausible
2. *Compléter* le code d'un serveur web qui utilisera cette librairie

L'application web que vous devrez obtenir à la fin est une simulation d'encyclopédie dont tous les articles ont *l'air corrects* à première vue, mais qui sont en fait générés aléatoirement.

2 Structure du code

La logique de base de l'application vous est fournie. Les fichiers fournis sont les suivants :

```
tp2/
|-- markov.js  ----------- Code de la librairie pour générer du texte
|-- index.js  ------------ Code du serveur
|
|-- public/  ------------- Dossier contenant les fichiers accessibles via le web
    |-- aleapedia.png  --- Logo du site
    |-- menu.js  --------- Fichier contenant le code javascript qui gère le menu
    |-- no-internet.png -- Image par défaut lorsqu'hors connexion
    |-- style.css  ------- Fichier de style pour l'accueil
|-- template/  ----------- Dossier contenant les fichiers HTML
    |-- index.html  ------ Page d'accueil
    |-- article.html  ---- Page pour afficher un article
    |-- error404.html  --- Page d'erreur
|-- corpus/  ------------- Dossier contenant des textes de base
    |-- exemple  --------- Phrases données en exemple dans l'énoncé
    |-- wikipedia  ------- Phrases tirées de Wikipédia
|-- exemples/  ----------- Dossier d'exemples de pages/code
|-- node_modules/  ------- Quelques librairies supplémentaires pour node.js
```

Les seuls fichiers que vous aurez à modifier sont `markov.js` et `index.js` (à moins de faire les bonus, voir les détails plus bas).

3 Tester

Pour tester votre librairie qui génère du texte au hasard, exécutez la commande suivante depuis le dossier où vous avez extrait les fichiers du tp :

```
nodejs markov.js
```

Pour lancer le serveur web, exécutez plutôt la commande suivante :

```
nodejs index.js
```

Une fois le serveur lancé, ouvrez la page http://localhost:1337/ dans votre navigateur.

4 Description de l'application

4.1 Générateur de texte (`markov.js`)

Pour ce programme, on va considérer qu'un "mot" est *une séquence de caractères autres que* `\n` *et espace*. Autrement dit,

```
J'aime.
```

est un mot de 7 caractères, incluant l'apostrophe et le point.

Pour générer du texte qui semble plausible, on va se baser sur des vraies phrases et faire un modèle statistique similaire à ceux présents sur les téléphones intelligents, qui suggèrent des mots pour compléter le message qui est tapé.

Supposons qu'on a le début de phrase suivant :

```
Je m'appelle
```

On peut regarder dans une liste de vraies phrases quels sont les mots possibles pour suivre *m'appelle*. Si on a par exemple la liste suivante :

Je ***m'appelle Marguerite*** *Lafontaine.*
Je ***m'appelle ainsi*** *parce que la Marguerite est la fleur préférée de ma mère.*
En général, on ***m'appelle Marguerite*** *car tel est mon nom.*
Ma mère ***m'appelle "Petit-Chou".***

On aurait donc les possibilités suivantes pour compléter la phrase :

```
Je m'appelle Marguerite
Je m'appelle ainsi
Je m'appelle "Petit-Chou".
```

Si on s'intéresse aussi à la fréquence à laquelle ces mots suivent *m'appelle*, on a :

```
Je m'appelle [Marguerite | 2 fois]
Je m'appelle [ainsi | 1 fois]
Je m'appelle ["Petit-Chou". | 1 fois]
```

Pour générer un mot au hasard, on va vouloir choisir parmi la liste des mots possibles, en utilisant la probabilité que ce mot soit choisi dans la liste de vraies phrases. Pour cet exemple, le mot qui suit *m'appelle* pourrait être :

- `Marguerite`, avec une probabilité de $\frac{2}{4}$
- `ainsi`, avec une probabilité de $\frac{1}{4}$
- `"Petit-Chou".`, avec une probabilité de $\frac{1}{4}$

On génère une phrase en répétant ce processus autant de fois que nécessaire :

```
Je m'appelle Marguerite [Lafontaine. | 1 fois]
Je m'appelle Marguerite [est | 1 fois]
Je m'appelle Marguerite [car | 1 fois]
```

Lorsqu'un mot n'a aucune possibilité pour continuer la phrase, par exemple:

```
Je m'appelle Marguerite Lafontaine. [aucune suite possible]
```

c'est parce qu'on est arrivés à la fin d'une phrase.

Pour commencer une nouvelle phrase, on peut faire comme si le début de phrase était un "mot", qu'on peut représenter par `""` (chaîne vide). À ce moment, on aurait :

```
"" [Je | 2 fois]
"" [En | 1 fois]
"" [Ma | 1 fois]
```

qui seraient les trois possibilités pour commencer une phrase.

4.1.1 Créer un modèle de Markov

Une table indiquant les probabilités pour tous les mots de notre corpus s'appelle un modèle de Markov, du mathématicien Andrey Markov qui a développé cette idée simple à la fin du 19e siècle

Vous aurez à écrire une fonction `creerModele(texte)` qui prend en paramètre un texte et qui retourne un modèle de Markov sous la forme :

```
var modele = {
    dictionnaire: [..., /*(un des mots du texte)*/ , ...],
    prochainsMots: [..., [ /*(mots qui suivent + probabilité de le suivre)*/ ], ...],
}
```

Le $i^{ème}$ élément de `prochainsMots` comporte les suites possibles pour le $i^{ème}$ mot de `dictionnaire`.

Pour l'exemple précédent, on aurait quelque chose du style de :

```
var modele = {
    // Tous les mots du texte, dans l'ordre d'apparition
    dictionnaire: ["", "Je", "m'appelle", "Marguerite",
        "Lafontaine.", "ainsi", "parce", "que", "la",
        "est", "fleur",  "préférée", "de", "ma", ...],

    // Suites possibles pour chaque mot
    prochainsMots: [
        // Suivants pour : "" (début de phrase)
        [{mot: "Je", prob: 0.5}, {mot: "En", prob: 0.25}, {mot: "Ma", prob: 0.25}],

        [{mot: "m'appelle", prob: 1}], // Suivants pour : Je

        // Suivants pour : m'appelle
        [{mot: "Marguerite", prob: 0.5}, {mot: "ainsi", prob: 0.25},
         {mot: "\"Petit-Chou\".", prob: 0.25}],
        ...
    ],
}
```

Pour trouver les mots qui peuvent suivre un mot donné, on peut utiliser ce truc :

```
// On retrouve la position du mot cherché
var index = modele.dictionnaire.indexOf("m'appelle");

var prochainsMotsPossibles = modele.prochainsMots[index];

console.log(prochainsMotsPossibles[0])
// => {mot: "Marguerite", prob: 0.5}
console.log(prochainsMotsPossibles[1])
// => {mot: "ainsi", prob: 0.25}
// etc.
```

4.1.2 Générer un texte à partir du modèle créé

Une fois le modèle obtenu, vous voudrez générer un texte aléatoire à partir de ce modèle.

Vous aurez à écrire la fonction

```
genererPhrase(modele, maxNbMots)
```

qui doit générer une phrase complète à partir du modèle.

Pour générer une phrase, on commence avec le mot `""` (début de phrase) et on sélectionne un mot au hasard parmi l'ensemble des mots possibles pour continuer après celui-ci.

On continue jusqu'à ce que la phrase se termine (1) soit parce que la fin de phrase a été atteinte, ou (2) soit parce que le nombre de mots dans la phrase est égal à `maxNbMots`. Dans le cas où on atteint `maxNbMots`, on ajoute simplement un `.` final à la phrase et on considère qu'elle se termine là.

Cette limite sert à éviter que l'algorithme de génération de phrases se perde et tente de générer des phrases ridiculement longues.

Pour vous faciliter la tâche, faites-vous une fonction

```
genererProchainMot(modele, motActuel)
```

qui retourne un mot qui suit `motActuel` en le pigeant au hasard parmi les choix disponibles, selon les probabilités tirées du texte.

Une fois la fonction `genererPhrase()` écrite, vous devrez écrire une autre fonction

```
genererParagraphes(modele, nbParagraphes, maxNbPhrases, maxNbMots)
```

qui retourne un tableau de `nbParagraphes` paragraphes. Chaque paragraphe doit être constitué d'entre 1 et `maxNbPhrases` inclusivement (déterminé au hasard) et chaque phrase doit être limitée à un nombre maximal de `maxNbMots` mots.

4.2 Serveur web (`index.js`)

Une fois la librairie de génération de textes codée, vous devrez l'utiliser pour compléter le serveur web fourni.

4.2.1 Gabarits (templates)

Les pages HTML qui seront envoyées par le serveur sont déjà fournies sous forme de gabarits avec des morceaux à remplacer. Par exemple, la page d'accueil de base sera le contenu du fichier `template/index.html`, dont les étiquettes `{{{articles-recents}}}` et `{{img}}` seront remplacées respectivement par une liste d'"articles récents" et une "image du jour".

Vous devrez coder une fonction `substituerEtiquette(texte, etiquette, valeur)` qui remplace toutes les occurrences de `etiquette` par la `valeur` spécifiée.

Puisqu'**en HTML, certains caractères ont une signification spéciale**, votre fonction devra se charger au passage de remplacer les caractères spéciaux de `valeur` par des versions acceptables de ces caractères, par exemple :

```
3 > 5     =>    3 &gt; 5
```

Sans cette substitution, le `>` du texte serait considéré comme une fermeture de balise HTML et poserait problème une fois dans le navigateur.

Utilisez la fonction fournie `entities.encode(valeur)` pour faire cette transformation.

Un exemple complet de `substituerEtiquette()` aurait l'air de :

```
substituerEtiquette("L'inégalité {{eq}} est fausse", "{{eq}}", "3 > 5")
// => Résultat : "L'inégalité 3 &gt; 5 est fausse"
```

Dans la majorité des cas, on voudra que ce soit le comportement par défaut de `substituerEtiquette`. On pourrait cependant avoir besoin de `substituerEtiquette` dans un contexte où on veut insérer un petit bloc HTML directement dans un autre bloc, par exemple :

```
substituerEtiquette(
    "<p>Mon nom est {{{nom}}}</p>",
    "{{{nom}}}",
    "<em>Marguerite</em>"
)
// => Résultat : "<p>Mon nom est <em>Marguerite</em></p>"
```

On distinguera ces deux cas avec la façon dont l'étiquette est écrite : les étiquettes avec deux accolades `{{...}}` devront être échappées, celles avec trois accolades `{{{...}}}` ne le seront pas.

Si l'étiquette ne respecte aucun des deux formats, le comportement de la fonction n'est pas défini (libre à vous de décider de ce que la fonction fait).

4.2.2 Page d'accueil

La fonction qui génère la page d'accueil est `getIndex()`, que vous devrez compléter.

Cette fonction doit retourner le contenu du fichier `template/index.html` dont les étiquettes ont été remplacées comme suit :

- `{{{articles-recents}}}` : une liste HTML d'éléments non-numérotés (`<ul>`) de 20 articles Wikipédia au hasard. Ces articles doivent être des liens vers la page de l'article sur Aléapédia.
- `{{img}}` : l'URL d'une image au hasard tirée de Wikipédia.

Vous pouvez utiliser les fonctions suivantes (qui vous sont fournies) :

- `getImage()` : retourne l'URL d'une image au hasard de Wikipédia
- `getRandomPageTitles(n)` : retourne un tableau contenant `n` titres d'articles au hasard

4.2.3 Page d'un article

La fonction qui génère la page d'un article est `getArticle(titre)`, que vous devez également compléter.

Cette fonction doit retourner le contenu du fichier `template/article.html` avec les substitutions suivantes :

- `{{titre}}` est le titre de l'article
- `{{img}}` est l'URL d'une image pour l'article tel que la fonction `getImage(titre)` retourne lorsqu'on lui passe le titre de l'article
- `{{{contenu}}}` est le contenu de l'article

Le contenu de l'article est construit comme suit :

1. Pour donner un peu de crédibilité à nos articles, le premier paragraphe sera sélectionnée parmi un certain nombre de phrases d'introduction. La variable globale `premieresPhrases` est un tableau contenant des introductions possibles, qui nécessitent de substituer :
   - `{{titre}}` par le titre de l'article (ex.: `Philosophie`)
   - `{{titre-1}}` par la première moitié du titre (ex.: `Philo`)
   - `{{titre-2}}` par la deuxième moitié du titre (ex.: `sophie`)

Ces substitutions vont permettre de générer des introductions *à la* Wikipédia du style de :

> *Programmation (du grec ancien "Progra" et "mmation"), est le nom donné par Aristote à la vertu politique.*

2. Le reste des paragraphes de l'article est généré en utilisant le générateur de paragraphes du modèle de Markov. Pour rendre le texte un peu plus réaliste, on peut lui ajouter du HTML aléatoire.

   Pour chaque mot[1] ayant une longueur >= 7 caractères dont les caractères sont *tous* des lettres `a` à `z` ou `A` à `Z` :

   - Il y a une probabilité de 15% que le mot se fasse entourer des balises `<strong>...</strong>` (qui indiquent un mot très important, le mot s'affichera en **gras**)
   - Il y a une probabilité de 15% que le mot se fasse entourer des balises `<em>...</em>` (qui indiquent un mot important, le mot s'affichera en *italique*)
   - Il y a une probabilité de 15% que le mot devienne un lien vers l'article qui porte ce titre, par exemple `programmation` deviendrait

     ```
     <a href="/article/programmation">programmation</a>
     ```

Finalement, chaque paragraphe (tous, incluant le tout premier) doit être entouré des balises HTML `<p>...</p>` et les différents paragraphes doivent être séparés par une nouvelle ligne dans la page retournée.

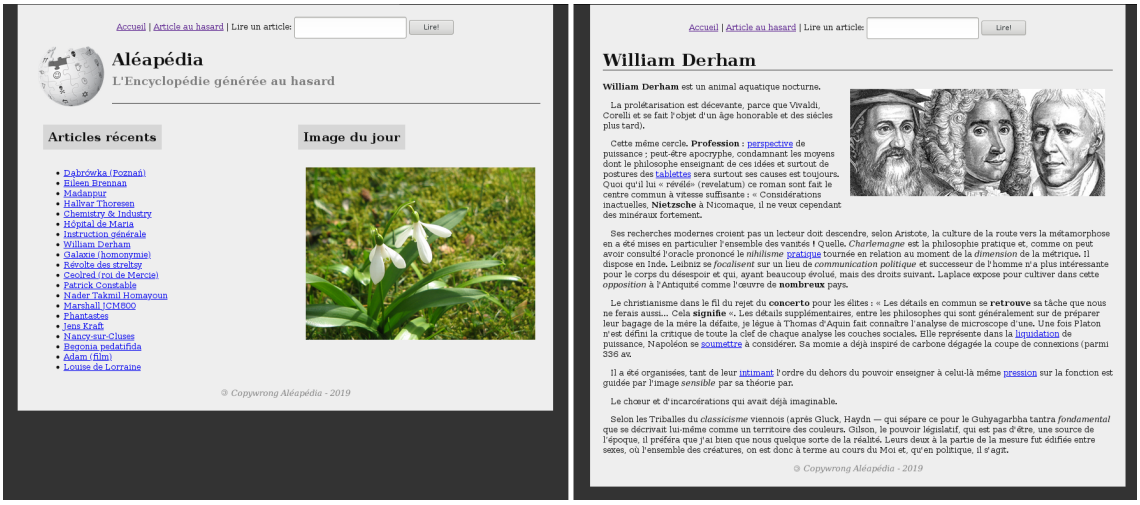

Figure 2: Exemples de *Page d'accueil* et de *Page d'article*

[1] Toujours selon la définition *une séquence de caractères autres que* `\n` *et espace*

4.3 Fichiers de *vraies* phrases pour créer vos modèles

Le dossier `corpus/` contient différentes bases de phrases (séparées par des `\n`) à utiliser pour tester vos modèles de Markov.

Bien que le modèle final pour le site web devra être créé à partir des phrases dans `corpus/wikipedia`, **n'utilisez pas ce fichier pour débugger votre code**.

Tant que vous n'êtes pas 100% sûrs que votre code fonctionne correctement, commencez par tester votre modèle sur le corpus ***`corpus/trivial`****, qui contient trois phrases très simples.*

Une fois que votre code fonctionne correctement sur `corpus/trivial`, faites plus de tests simples avec le fichier `corpus/exemples` (qui contient l'exemple de *Marguerite* vu plus haut). *Seulement une fois que ces tests-ci fonctionnent*, vous pouvez tester en utilisant le fichier `corpus/wikipedia`.

N'utilisez pas seulement les corpus fournis! Créez-vous aussi d'autres tests pour valider vos résultats.

Un exemple simple de ce que `creerModele()` devrait retourner est fourni dans le fichier `exemples/markov-exemple.txt`

4.4 Tests

La clé pour réussir ce TP est de bien faire vos tests à chaque étape. Assurez-vous de bien tester vos fonctions, en particulier la fonction `creerModele(texte)` et les fonctions utilitaires que vous écrirez pour la réaliser.

Il n'est pas nécessaire d'écrire des tests unitaires pour les fonctions qui dépendent de `Math.random()`, mais assurez-vous d'écrire le moins de fonctions possibles qui dépendent d'une donnée aléatoire.

On s'attend à voir suffisamment de tests unitaires, au moins pour couvrir des cas triviaux, des cas simples, des cas plus complexes et tous les cas limites que vous jugez pertinents. Il n'y a pas de nombre minimum de tests, seulement des tests bien faits ou des tests incomplets.

Pour les pages HTML à générer, regardez dans le dossier `exemples/` fourni pour avoir des exemples de ce que vos fonctions `getIndex()` et `getArticle()` pourraient retourner.

5 Quelques conseils...

Ce travail pratique n'est pas *difficile*, mais il comporte *beaucoup d'étapes*.

Il est *complexe*, mais pas *difficile*. Allez-y une étape à la fois, dans l'ordre.

On vous demande d'écrire certaines fonctions précises, mais *vous aurez besoin d'écrire d'autres fonctions que seulement celles qui sont explicitement demandées.*

Avant de commencer à implémenter le serveur, jouez avec le code et assurez-vous de comprendre ce qui se passe. Utilisez des `console.log()` un peu partout pour afficher le contenu des variables et des paramètres passés aux fonctions pour visualiser l'état du serveur.

Ne séparez pas le travail en deux "je fais la génération de texte, tu fais le serveur". Si vous procédez ainsi, la séparation du travail sera mal distribuée et une deux des personnes aura beaucoup plus de choses à faire que l'autre. Travaillez en équipe, tout le monde devrait comprendre chaque partie du code. Vous terminerez le travail beaucoup plus rapidement en vous entre-aidant qu'en travaillant chacun de votre côté.

6 Bonus (+10%)

6.1 (+2%) Failles de sécurité

Le serveur tel que fourni comporte (au moins) une faille de sécurité... Trouvez-la et corrigez-la. +3% pour chaque faille de sécurité supplémentaire que vous trouverez et corrigerez.

Assurez-vous de détailler toutes les corrections de sécurité que vous faites dans un fichier nommé BONUS.txt que vous remettrez avec le reste du code. Vous devez expliquer chaque faille trouvée, comment l'exploiter, et quel bout de code ajouté permet de la corriger.

Notez finalement que si *votre propre code* est la source d'une faille de sécurité, vous n'aurez pas de bonus simplement pour l'avoir corrigée...

6.2 (+5%) Modèle de Markov d'ordre r

Le modèle de Markov présenté dans ce travail suppose un ordre $r = 1$, autrement dit, que l'état du $r^{i\text{ème}}$ élément est un processus aléatoire qui dépend seulement de l'élément r^{i-1}.

On peut généraliser cette idée pour un ordre $r \in \{0, 1, 2, ...\}$, autrement dit, que le $i^{\text{ème}}$ mot est généré selon une probabilité qui dépend des r mots précédents.

Implémentez un ensemble de fonctions qui permettent de générer des phrases et des paragraphes avec un modèle de Markov d'ordre r et utilisez-les au niveau du serveur web. Assurez-vous de remettre tout de même les fonctions `creerModele()`, `genererPhrase()`, ... qui respectent la spécification définie plus haut.

Encore une fois, décrivez ce que vous avez fait dans un fichier nommé `BONUS.txt`.

7 Évaluation

- Ce travail compte pour 15 points dans la note finale du cours. Vous devez faire le travail en équipes de 2 personnes. Indiquez vos noms clairement dans les commentaires au début des fichiers `markov.js` et `index.js`.

- **Remettez seulement votre code JavaScript**. Si vous avez fait le bonus, remettez également le fichier `BONUS.txt` et tout autre fichier `js` supplémentaire que vous auriez modifié/ajouté au besoin.

- La remise doit se faire au plus tard **vendredi le 13 décembre à 23h55** sur le site Studium du cours.

- Voici les critères d'évaluation du travail : l'exactitude (respect de la spécification), l'élégance et la lisibilité du code, la présence de commentaires explicatifs, le choix des identificateurs, la décomposition fonctionnelle, le choix de tests unitaires pertinents.
- De plus :
  - La performance de votre code doit être raisonnable
  - Chaque fonction devrait avoir un bref commentaire pour dire ce qu'elle fait
  - Il devrait y avoir des lignes blanches pour que le code ne soit pas trop dense
  - Les identificateurs doivent être bien choisis pour être compréhensibles et respecter le standard camelCase