ISTY
Rattrapage projet interfilière IATIC-4

Algorithmes de Matchgin

Encadré par:
Madame Laurence Pilard

Realisé par:
Karim KHARRAZ

Introduction

En théorie des graphes, le problème du couplage (matching en anglais) est un problème qui consiste étant donné un graphe non orienté à trouver un sous-ensemble d'arêtes du graphe qui n'ont pas de sommets en commun.
Un couplage maximum est un couplage contenant le plus grand nombre possible d'arêtes.
Un couplage maximal est un couplage M qui est maximal au sens de l'inclusion, tels que pour toute arête a qui n'est pas dans M, M ∪{a} n'est plus un couplage.
L'objectif de ce projet est d'implémenter une application web permettant à l'utilisateur de :

- Construire un graphe à partir d'un fichier envoyer au serveur web.
- Construire un graphe d'une manière aléatoire.
- Stockager et visualiser des graphes.
- Appliquer l'algorithme d'Edmonds (matching maximum) sur un graph fourni et visualiser le résultat.
- Appliquer un algorithme de glouton permettant l'obtention d'un matching maximal et visualiser le résultat.

Environnement et outils

Environnement de développement

SpringToolSuite4 est un eclipse préconfiguré qui englobe:
Spring : C'est un framework open source pour construire et définir l'infrastructure d'une application web en java et qui facilite le développement et les tests grâce à sa notion de tag. (@Controller, @GetMapping, @PostMapping….)
Le serveur Spring Boot : est un serveur web Tomcat préconfiguré et intégré à l'environnement.
La configuration du serveur est faite dans le fichie"application.properties". Le port (8888), la taille maximale d'un fichier transfert (128KB) et d'autres paramètres de configurations sont indiqués dans ce fichier.

Languages

L'implémentation du BackEnd de l'application a été faite en Java (8).

L'implémentation du FrontEnd de l'application a été faite en HTML(5) javascript et Css.

FrameWork et outils

Thymeleaf est un moteur de template, sous licence Apache 2.0, écrit en Java pouvant générer du XML/XHTML/HTML5.. Son but principal est d'être utilisé dans un environnement web pour la génération de vue pour les applications web basées sur le modèle MVC.
Thymeleaf s'intègre parfaitement avec Spring et avec l'architecture de notre application web.

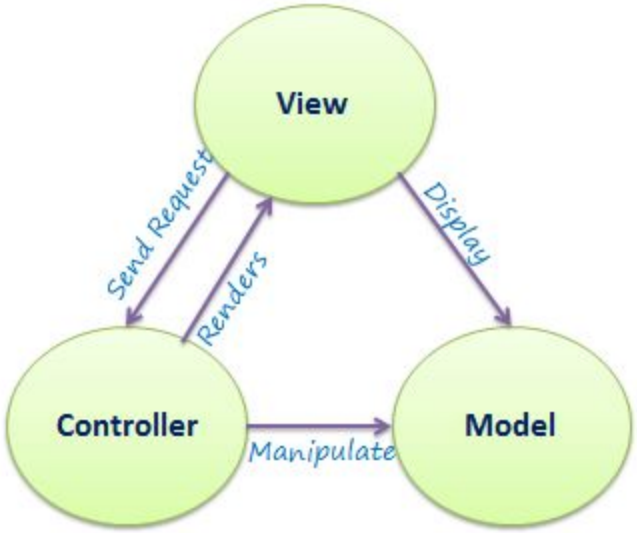
D3.js est une bibliothèque graphique JavaScript qui permet l'affichage de données numériques sous une forme graphique et dynamique. Elle m'a servi pour l'affichage des graphes.

AJAX (Asynchronous JavaScript + XML) n'est pas une technologie en soi, mais un terme désignant une « nouvelle » approche permettant l'échange des données entre le serveur et les pages html d'une manière rapide sans recharger toute la page web. Vous remarquerez donc que lors des différents manipulations faites (stockage, construction de graphe, applications des deux algorithmes de matching) la page ne se recharge pas (pas de refresh) seuls les données sont échangées.

Architecture et Logique d’implémentation

Architecture Globale

L’application développée est basé sur l’architecture MVC + AJAX (modifié).



Le modèle représente la forme des données et la logique métier. Il est généralement utilisé pour conserve les données de l'application dans une base de données.
Pour mon application j’ai fait un choix de persistance différent.
Vu que toute l’application tourne autour des graphes et des algorithmes exécutés sur ces derniers et que l'utilisateur va me communiquer la description d’un graphe sous forme de fichier.
J’ais le choix de tout stocker sous cette forme sous le répertoire (upload-dir).
Je précise que les fichiers ne sont pas stockés tels qu’ils sont reçus mais plutôt sous la forme d’un fichier texte formé suite à la construction du graphe correspondant.

Le modèle ne perd pas son utilité car j’ai exploité la sérialisation de cet objet en JSON à fin de le communiquer à la vue à travers le controlleur. Ce format me sera utile pour la représentation graphique.
Le modèle de mon application est l’objet Graphe qui est lui même composé d’un ensemble de noeuds. La composition de ces derniers sera expliqué et détaillé par la suite.

La vue : Les vues sont directement responsables de tout ce que l'on va envoyer aux navigateurs. L'application contient une seule vue "UploadForm". C'est une page HTML contenant des scripts.
La communication de cette page avec le contrôleur est faite via des Requêtes Web (POST et Get). Ces appels sont fait dans les fonctions JavaScript en mode AJAX.

Le contrôleur : Le contrôleur a pour rôle d'assurer la communication entre la vue et le modèle. "MatchingController" est le controller de mon application. Il écoute et répond aux requêtes du client.
Les traitements de données ne s'effectue pas dans le contrôleur.
Ce dernier fait appel à une couche service "MatchingServices" ,via son interface "MatchingServicesInterface", qui est respsonsable du traitement des données.
Le contrôleur retourne alors les résultats dans le "Body" de la réponse de la requête, plus un code dans le "header" indiquant le statut de la réponse(200 ->OK).

Logique d'implémentation

Cette partie sera dédiée à l'explication de la logique métier que j'ai adoptée afin de réaliser toutes les fonctionnalitées demandées.

Choix des structures

Un graphe (Graph.java) est représenté par les attributs suivant:

- nbnondes: un entier qui représente le nombre de noeuds.
- nbedges: un entier qui représente le nombre d'arcs.
- <nodes>: une ArrayList des noeuds que contient le graphe.
- <edges>: une ArrayList des arcs que contient le graphe.

=> Le choix d'avoir une liste de noeuds et non pas d'arcs a été motivé par deux principales raisons:

1. Dans ma première approche de l'algorithme d'Edmonds, j'ai relevé le fait que j'aurais plusieurs types de noeuds et que la recherche des chemins va dépendre de ses derniers.
2. La plupart des frameworks de visualisation de graphes (dont Ds3.js) adoptent une structure de données quasi-similaire (Objets nodes, objets edges) ce qui me faciliterait l'affichage des graphes par la suite.

Remarque:
La liste des edges n'a aucun rôle dans les algorithmes de matching, sa présence est pour faciliter l'interaction avec DS3.js afin de pouvoir dessiner les graphes. Elle est la juste pour l'affichage.
Un Noeud (Node.Java) est représenté par les attributs suivant:

Label : un entier qui identifie le Noeud;
Nbrs (@JsonIgnore): une LinkedHashSet<Node> qui représente les voisins d'un noeud.
matchedWith(@JsonIgnore): un Noeud qui a été couplé avec le noeud courant suite à l'exécution d'un algorithme de matching.
forIteration(@JsonIgnore): c'est entier compteur d'iterations.
type (@JsonIgnore): le type du noeud {EVEN, ODD, FREE}
Parent(@JsonIgnore) : Si notre noeud courant fait partie d'un chemin auguementé alors le noeud parent est celui avec qui il est lié
contracte(@JsonIgnore): un boolean qui indique si il s'agit d'un noeud contracté.

=> Les attributs marqué par la balise @JsonIgnore sont des attributs métiers et ne seront donc pas envoyer à DS3.js.
Le fait d'avoir cette panoplie d'attributs n'est pas anodin, j'aurais pas à implémenter une structure de forêt pour l'algorithme Blossom car cette notion est retrouvée grâce à un noeud. On peut voir le noeud comme un graphe ce qui sera utile par la suite pour la recherche des chemins d'augmentations ceci est expliqué par la présence de la liste de voisins Nbrs.
La contraction sera donc faite sur les noeuds.

Un blossom (Blossom.java) est représenté par les attributs suivants:
Node root
List<Node> cycle
Node blossomAlias

Services

L'interface "MatchingServicesInterface" présente les différentes méthodes (services REST) qui peuvent être appelé par le contrôleur dans le but de répondre aux requêtes du client(navigateur web).
Tous clic de l'utilisateur (save,constuct,upload,...) est en fait une requéte (POST or GET) en mode **REST** pour le serveur.
Cette interface constitue une couche d'abstraction et de sécurité qui expose quelques méthodes et garde d'autres comme méthodes privées.

```
public interface MatchingServicesInterface {

    //upload files services
    void init();
    void store(MultipartFile file);
    Stream<Path> loadAll();
    Path load(String filename);
    Resource loadAsResource(String filename);
    void deleteAll();

    // graph services
    Graph constructGraph(MultipartFile file);
    Graph constructGraph(JSONObject graph);
    void printGraph(Graph g);
    Graph edmondExec(Graph graph);
    Graph greedyMatching(Graph g);
    Graph constractRandomGraph(int nbnodes, int nbedges);
    Graph visaulMatchedGraph(Graph g);
    void saveGraph(JSONObject graph, String name);
```

On peut partager les services en deux :

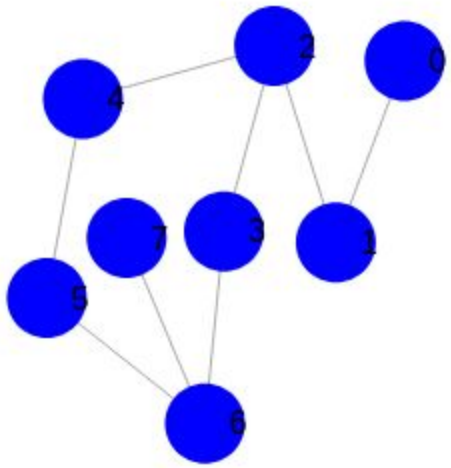
Les premières méthodes sont dédiées à la réception et l'enregistrement des fichiers que l'utilisateur peut transfert au serveur. Il s'agit de fichier descriptif d'un graphe.
Je souligne que l'enregistrement est fait après la construction du graphe (liste de noeuds, liste d'arcs) et qu'il faut que l'utilisateur demande l'enregistrement du graphe.
On peut enregistrer un graphe après l'exécution des algorithmes de matching ou avant.
Il s'agit de méthodes relativement simple et qui ne reflète pas du coeur du projet, il est donc inutile de les détailler.

La deuxième partie est dédiée aux fonctionnalitées liées à la construction de graph et l'application des algorithme de matching:

Graph constructGraph(MultipartFile file)

Cette fonction permet de prendre un fichier décrivant le graph ayant la structure suivante:

- Nombre de noeuds
- Nombre d'arcs
- Chaque ligne décrit décrit un arc (u,v) avec u et v séparer par un seul espace.
- Tous les noeuds commence par 0.



Exemple:

8
8
0 1
1 2
2 3
3 6
2 4
4 5
6 7
6 5

Graph constractRandomGraph(int nbnodes, int nbedges)

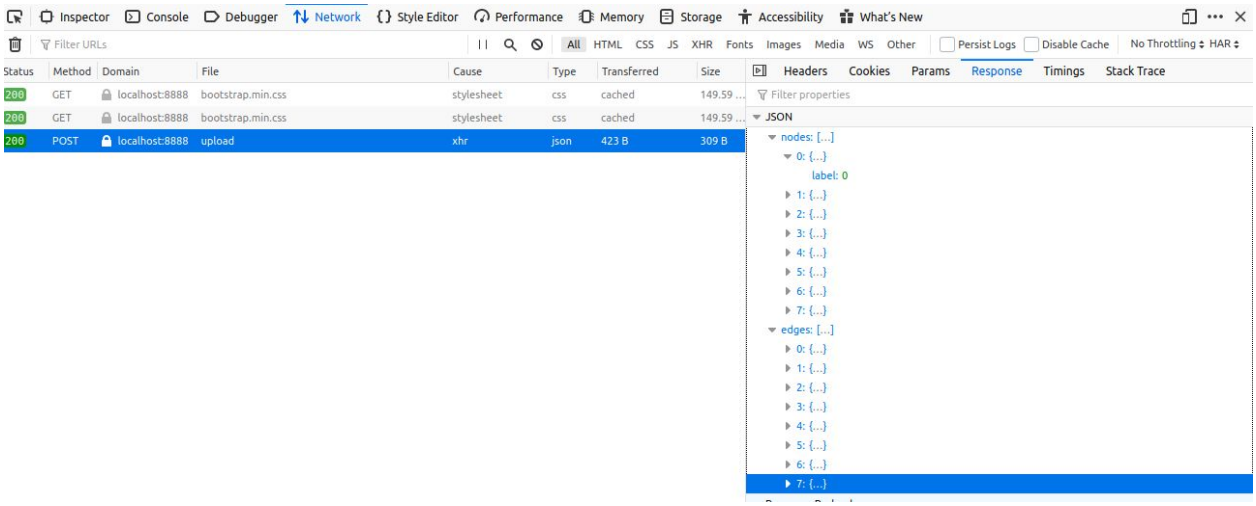
Cette fonction prend en entrée le nombre de noeuds et le nombre d'arcs d'un graphe est construit un graphe d'une manière aléatoire en respectant les conditions suivantes:

- Le nombre d'arcs doit être inférieur à **((nbnodes*(nbnodes-1))/2))**.=> il y a au plus un arc reliant deux noeuds.
- Un noeud ne peut pas avoir un arc qui pointe vers lui même.

Graph constructGraph(JSONObject graph)

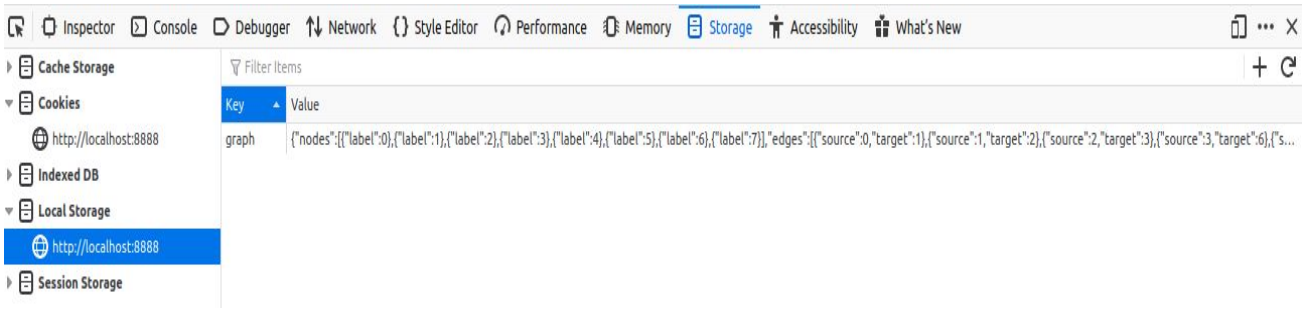
Suite à la construction d'un graphe à partir d'un fichier ou d'une manière aléatoire, le BackEnd de l'application l'envoie vers le FrontEnd sous la forme d'un graphe visuel.

Cette forme consiste en une liste de noeuds et une liste d'arcs sous format JSON.



=> Cette structure est faite pour l'affichage et on ne peut pas lui appliquer les algorithmes de matching (pas de liste de voisins nbrs).
Ce qui explique l'utilité de la fonction "constructGraph(JSONObject graph)" qui nous permet de reconstruire le graphe afin de lui appliquer les algorithmes qu'on souhaite.

Remarque:
Tout au long du cycle de vie d'un graphe, ce dernier est stocké sur le navigateur dans un emplacement dédié: **localStorage**



Graph edmondExec(Graph graph)
Graph greedyMatching(Graph g)

Les deux fonction suivantes sont respectivement responsable de l'exécution de l'algorithme de d'Edmonds et un autre algorithme glouton de matching que j'ai implémenté moi même.
L'explication en détail de ces deux algorithmes sera faite dans la partie suivante du rapport.

Blossom algorithme

EXPLICATION GENERALE

Définition

L'algorithme Blossom est un algorithme en théorie des graphes pour construire des matching (M) maximales sur des graphes. L'algorithme a été développé par Jack Edmonds. Étant donné un graphe général G = (V, E), l'algorithme trouve un M correspondant tels que chaque sommet de V est incident avec au plus un arc dans M et | M | est maximisé.
Le matching est construit en améliorant de manière itérative un matching initialement vide, à chaque recherche d'un chemin d'augmentation dans le graph.
L'idée clé de cet algorithme est qu'un cycle de longueur impaire(Blossom) est contracté en un seul sommet, la recherche se poursuit de manière itérative dans le graphe contracté.

Chemins d'augmentation

Un sommet v est dit "exposé" si aucun arc de M n'est incident avec v.
Un chemin dans G est un chemin alternatif, si ses arcs ne sont alternativement pas dans M et dans M (ou en M et non en M).
Un chemin d'augmentation P est un chemin alternatif qui commence et se termine à deux sommets exposés distincts.
Le nombre d'arêtes non matchées dans un chemin d'augmentation est supérieur de un au nombre d'arêtes appariées, et donc le nombre total d'arêtes dans un chemin d'augmentation est impair.
Une augmentation du matching le long d'un chemin d'augmentation P consiste à remplacer M par une nouvelle correspondance M 1 = M $\oplus$ P = (M $\setminus$ P) $\cup$ (P $\setminus$ M).
M est maximal si et seulement s'il n'y a pas de chemin d'augmentation de M dans G.
Par conséquent, soit un matching est maximale, soit il peut être augmentée.
Ainsi, à partir d'un matching initiale, nous pouvons calculer un matching maximale en augmentant le matching actuel avec des chemins d'augmentation tant que nous pouvons les trouver.

Blossoms et contraction

Blossom B est un cycle en G composé de 2k + 1 arcs dont exactement k appartiennent à M, et où l'un des sommets v du cycle est tel qu'il existe un chemin alterné de longueur paire de v à un sommet exposé w.
Le graphe contracté G' est un graphe obtenue à partir de G en contractant tous les arcs de B.
M' est le matching de G' correspondant à M.

Recherche de chemin augmenté

D'une manière générale, la recherche d'un chemin d'augmentation utilise une structure de données auxiliaire constituée d'une forêt F dont les arbres individuels correspondent à des parties spécifiques du graphique G.
Mais vu la structure du noeud que j'utilise dans mon implémentation, un arbre est un noeud (PseudoNode). On a pas besoin d'avoir une autre structure de données.
A chaque itération, l'algorithme trouve soit un chemin d'augmentation, ou bien un Blossom et revient sur le graphique contracté correspondant, sinon on conclut qu'il n'y a pas de chemin d'augmentation.

Edmonds Pseudo Code

```
for vertex u in Graph G
        if u is unlabelled  // Free Node
        path = findAlternatingPath(u)  // Recherche de chemin alternatif
        if (path not Empty)
                        if (path is a pseudoNode) //Tree
                                while ((path not Empty) && (path not pseudoNode))
                                path = findAlternatingPath(u)
                        if (path not Empty)
                                //update the augmenting path
                        remove the edges in the path that are in matching
                replace them in the matching with those that are not


 findAlternatingPath(u)

        label(u) =  EVEN
        for each vertex v, in nbrs(u)
        if (v is unlabelled) //found alternating path
        parent(v) = u
        label(v) = ODD
        return v
        else
        if(label(v)== EVEN) //there is an odd cycle
        contract the odd cycle into a pseudonode
        add to nbrs of every node in the cycle the edge to pseudonode
        remove  the edges from nbrs to the nodes in the cycle.
        label(pseudonode) = EVEN
        return pseudonode
```

```
else
label(v) = ODD
w  = matched(v)
parent(w) = v
label(w) = EVEN
path = findAlternatingPath(w)
if (path not Empty)
        return path
else
        continue //take the next nbr
```

Vision globale de la complexité

Pour le calcul de la complexité, je me suis basé sur l'algorithme de Blossom en général et non sur l'implémentation.

On commence par le calcul du nombre d'itérations que va effectuer l'algorithme général.
On aura au plus n/2 appel à la fonction de recherche de chemin alternatif "findAlternatingPath"
Ceci est expliqué par le fait que le nombre maximal d'arcs dans un matching maximum d'un graphe de n noeuds est de n/2.
Chaque appel à la fonction le nombre d'arcs dans M est augmenté de 1.
Vu qu'on commence par M étant un ensemble vide on fera au plus n/2 appels.

À chaque itération du chemin d'augmentation, nous parcourons au plus tous les nœuds du graphe. Et pour chaque nœud, nous passons par les noeuds voisins (nbrs) non marqués(unmatched).
Il y a 3 cas possibles :

Premier cas : Blossom récursivité
On contract les noeuds dans le blossom.
On parcours tous les noeuds et arcs de notre graphe et on renomme les blossom avec un nouvel identifiant => O(E+V)=O(M)

Deuxième cas: L'ajout d'un arc à la forêt
(add to nbrs of every node in the cycle the edge to pseudonode
remove the edges from nbrs to the nodes in the cycle)

Pour un arc e= (v,w) si (w,x) $\in$ M , quand x et w sont marqués, alors on ajoute (v,w) et (w,x) à la forêt.=> O(1)
On a au plus m appels de ce cas ce qui ramène la complexité à O(m).

Troisième cas : Ajout d'un chemin d'augmentation
On retourne à la fonction principale, qui va déterminer si on a une autre itération.
=> O(n) (c'est un parcours).

Complexité total= O(n) * [O(m)+ (O(m)+O(m))*O(n)] = $O(n^2m)$.

Algorithme Glouton de matching

Explication générale

Un algorithme glouton suit le principe de faire, étape par étape, un choix optimum local. Dans certains cas cette approche permet d'arriver à un optimum global, mais dans le cas général c'est une heuristique.

Pour cet algorithme, j'ai fait une implémentation très simple qui consiste à:
Choisir un noeud de départ.
Matcher ce noeud à un de ses voisins(via nbrs).
Marquer les deux noeuds comme matché.
Récupérer un voisin du deuxième noeud qui n'est pas matché.
Marqué les deux noeuds
Et refaire jusqu'où au moment ou arrive sur un noeud qui n'a pas de noeuds non marqué comme voisin.

Déploiement

ngrok

Le déploiement de mon application web a été fait via Ngrok. Il s'agit d'un tunnel qui permet l'accès à mon web serveur local (spring boot server) via l'adresse suivante:
http:/karim-matching.eu.ngrok.io.ngrokIo/

Conclusion

Ce projet m’a permis de comprendre le problème de matching, d'implémenter l’algorithme d’Edmonds pour déterminer un matching maximal d’un graphe et un algorithme glouton pour un matching maximum.

J’ai aussi appris à utiliser pas mal d’outils et de framework qui me seront surement utiles dans le futur.

Deux évolution que je trouve à la fois utile et très intéressantes à faire, sont le parallélisme de l’algorithme d’Edmonds via le multithreading et l’application de ce dernier sur des graphes pondérés.