Flapstaff
is a Light and Powerful Social Text Analyser For Facebook

Benjamin Crespo Yann Pravossoudovitch Mathieu Triay

25 mai 2010

"Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence."

E.W Dijkstra

Table des matières

Chapitre 1

Remerciements

Merci **Google** pour être notre ami, et nous fournir tous les jours des outils pratiques et efficace. Merci à **BitBucket** d'offrir un service d'un telle qualité, et gratuit en plus. Merci à **Mercurial** pour avoir fonctionner beaucoup mieux que **Git** avant lui.

Merci à **Quentin Dejean** et **Florian Boeuf-Terru** pour nous avoir supporté tout au long du projet. Merci à **Michael NGuyen** pour nous avoir prodigué des astuces de qualité. Merci à **Dijkstra** pour son génie qui n'a d'égale que son arrogance. Merci à **Fred** pour sa voix particulière. Merci à **Jamy** pour écouter continuellement ce que Fred a à dire.

Et finalement, Merci aux **Mayas**, sans qui **Flapstaff** ne serait qu'un nom abscon inventé après de longues heures d'Analyse des Coûts et pas une superbe danse pratiquée lors de rituels bizarre. Ce qui serait beaucoup moins classe.

Chapitre 2

Introduction

Flapstaff (Flapstaff is a Light And Powerful Social Text Analyser For Facebook) est un programme permettant de générer des fichiers à partir de données récupérées de Facebook (du Social Text) et puis de les analyser pour connaître les **composantes fortement connexes**, mais aussi **le chemin le plus court** entre des personnes données suivant leur fréquence de connexion.
Pour réaliser ce programme, nous avons mis à profit nos connaissances sur les graphes pour obtenir un **programme fonctionnel et répondant au cahier des charges**. Ce rapport est une synthèse de la génèse de Flapstaff, se ses balbutiements à ses perspectives d'avenir.

Dans le domaine des graphes, l'algorithmique est **déjà grandement éprouvée**, c'est pourquoi nous avons choisi de donner à ce document une dimension moins technique pour mieux retracer nos choix, les outils utilisés et nos **interactions**.
En effet, le travail en groupe a été un plus grand challenge que les algorithmes sur les graphes - déjà fortement documentés sur l'Internet Mondial.
Les décisions que nous avons dû prendre tout au long du projet devaient être mûrement réfléchies car elles touchaient directement le **futur du projet**, c'est pourquoi il est tout aussi important de les connaître que de savoir comment et pourquoi elles ont été prise.

C'est pourquoi, nous verront premièrement la **mise en place du projet** dans le groupe, les outils que nous avons mis en place et les documents qui ont servis de référentiels, puis nous nous concentreront sur les **choix de structure et les algorithmes** qui en découlent, pour aboutir sur nos **procédures de tests** qui nous ont permis d'aboutir à une cohérence globale dans le projet, et nous conclurons sur **les apports de ce travail en groupe et les possibilités d'améliorations**.

Chapitre 3

Mise en place

3.1 La communication

Lorsque l'on travail en groupe, ce que l'on cherche en premier à être capable de faire, c'est de **communiquer**. La communication fût un élément crucial lors de la conception et de l'aboutissement de ce projet. Les disponibilités de chacun n'étant - presque - jamais coordonnées, nous avons dû alors mettre en place plusieurs systèmes pour pouvoir discuter des différents problèmes qu'apportait le projet.

Les premiers outils, en dehors des emails, furent **Google Wave** et **Google Docs** qui nous ont permis de construire des documents de conception **accessibles et modifiables par tous**, tout en gardant une trace des modifications de chacun. Ainsi, la présence directe des autres membre n'était pas une obligation, et nous avons pu **avancer rapidement**.

3.2 Les outils

Un autre problème de communication est celui concernant le codage du projet. Car une fois que l'on a décidé ce que l'on allait faire, il fallait qu'on puisse le faire ensemble. Nous avons alors mis en place un dépôt Mercurial sur le site BitBucket.

En plus de bénéficier des **avantages** évidents d'un système de contrôle de versions décentralisé taillé pour le travail en équipe, BitBucket propose aussi **un bug tracker et un wiki**. De cette façon il était possible de mettre du code sur le dépôt tout en attachant le commit à une ticket sur le bug tracker et ainsi **référencer l'avancement** de la résolution d'un problème.

Une fois ces outils prêt, il fallait encore se mettre d'accord sur une chose avant d'attaquer le codage du projet : **le coding style** (ou charte de programmation). Nous avons tous les trois des manières très différentes de coder, et il était donc très important d'avoir un document permettant d'uniformiser cela pour obtenir **un code propre et cohérent**. Nous nous sommes alors renseignés sur les coding style existants en C, et avons choisis de prendre exemple sur le coding style de **Linux** dicté par **Linus Torvalds**. Nous l'avons un petit peu modernisé, adapté à nos manières respectives de coder, et agrémenté de quelques conseils d'utilisation du bug tracker et des Unit Tests.

Ensuite, nous avons mis en place la **répartition du travail** grâce aux tickets du bug tracker, et nous avons pu attaquer la conception et le codage à proprement parler.

Chapitre 4

Structures et Algorithmes

4.1 Structures générales

4.1.1 Structure Personne

La structure Personne est la **base** du programme, elle est utilisée dans tous les autres modules. On retrouve aussi les listes de personnes qui servent notamment pour la liste d'amis.
Elle représente une personne à l'aide des attributs noms, id et fréquence de connexion au réseau social.
Attention, l'**ID n'est pas un int, mais une chaîne de caractères** car Facebook stocke ses IDs sur plus de 15 chiffres, ce qui dépasse largement l'Integer classique.

4.1.2 Structure Relation

La structure relation sert à la fois pour matérialiser les **relations** entre 2 personnes, mais aussi les **questions** (qui sont une forme de relation ponctuelle). De ce fait, on retrouve des listes de relations un peu partout dès que l'on veut manipuler des questions ou bien des relations. Elle contient juste 2 ids (id1 et id2), qui sont des chaînes de caractères.

4.1.3 Structure Graph

La structure de Graphe est largement composée des 2 structures précédentes mais aussi d'une matrice d'adjacence qui permet de localiser rapidement si 2 personnes sont amis ou non. La matrice à été privilégiée vis à vis de la liste de voisins pour **sa simplicité de mise en oeuvre et sa rapidité d'accès** (mais pas de parcours...). On utilise la fonction `"position"` pour **lier la matrice et la liste de personne** et ainsi retrouver à partir de la position dans la matrice à quelles personnes elle correspond, ainsi nous n'avons pu stocker les personnes qu'une seule fois et **éviter une redondance d'information**.

Pour des informations plus techniques voir la doc Doxygen dans le dossier "doc" du projet.

4.2 Recherche de Composantes Fortement Connexe

La recherche de composantes fortement connexe dans un graphe se fait assez simplement par des algorithmes connus, que l'on peut facilement retrouver sur Internet.
Nous avons choisi d'utiliser une méthode qui consiste à effectuer un parcours en profondeur sur le graphe normal, ce qui nous donne tous les points atteignables à partir d'un point donné, et d'effectuer ce même parcours dans le graphe dual.

Le parcours dans le graphe dual comporte une spécificité, en effet, il revient à faire un parcours sur le graphe normal, mais en **inversant les coordonnées de la matrice** pour obtenir le retour !

Une fois les 2 parcours effectués, il faut trouver l'**intersection** entre ces 2 ensembles pour obtenir tous les points de la composante. Notre parcours en profondeur renvoi un tableau de taille égale au nombre de personne et place des 1 sur les personnes que l'on peut atteindre. De cette manière, il nous permet de savoir instantanément si on peut accéder à une personne en utilisant la condition *parcoursEnProfondeur[positionDeLaPersonneDansLaListeD'amis] == 1*.

Nous avons privilégié l**a vitesse de calcul plutôt que l'occupation mémoire** dans ce cas là. Le calcul est effectué en *0(1)*, mais la complexité mémoire est très grande pour un parcours de graphe entier.

4.3 Envoi de messages

Le programme devait aussi permettre de connaître le temps le plus court pour faire passer un message sur le réseau suivant leur fréquence de connexion (généré aléatoirement et arbitrairement). Nous avons décidé que la fréquence de connexion au réseau variée entre 1 et 720h (soit 1 mois). Au delà de 720h on considère que **la personne n'utilise plus le réseau**.

L'algorithme revenait à trouver le chemin le plus court entre 2 points dans un **graphe pondéré par la fréquence**. Il existe un algorithme très connu et déjà très éprouvé dans le domaine : l'algorithme de *Dijkstra*. Vu en cours, dans ce là, nous avons pu le mettre en oeuvre de manière pratique et voir comment il se comportait sur des graphes de taille "réelle".

Pour chaque personne trouvée par l'algorithme il fallait accéder à la fréquence de connexion en utilisant la fonction "position" pour obtenir l'emplacement de la personne dans la liste d'amis. De ce fait, la fonction position est appelée maintes fois à travers tout le programme et s'est retrouvée être un **point clé lors de la recherche d'optimisation possible**.

4.4 Personnes importantes

Une fois le problème des CFC réglé, nous avons dû nous attaquer à celui des personnes importantes dans celle-ci (en bonus). Il n'existe pas d'algorithme connu - comme Dijkstra - qui soit efficace, nous avons alors fourni un algorithme de type **brute force** qui suit le schéma suivant :

```
Pour chaque composante fortement connexe
On choisit un élément à supprimer
On trouve fait la liste des points moins celui qu'on veut supprimer
On fait un parcours en profondeur pour le premier point
On vérifie que tous les autres points de la sous-composante sont encore accessible.
Si c'est le cas, on passe au point suivant
Sinon, le point qu'on veut supprimer est important
On choisit un autre élément à supprimer jusqu'à ce qu'on ait épuisé la composante
```

Cet algorithme vérifie bêtement et méchamment les propriétés qui doivent être respectées sans chercher de subtilité. Ce qui fait qu'il est plutôt **lent au delà de 1000 personnes**.
Une versions améliorée pourrait travailler sur des **sous-graphes** plutôt que le graphe entier pour éviter des parcours en profondeurs qui vont trop loin inutilement.

Chapitre 5

Tests et Résultats

5.1 Facebook

La première étape pour les tests a été de trouver des fichiers contenant des réseaux plus grands et plus proche de la réalité. Nous nous sommes alors tourné vers Facebook. Le sujet spécifiait que nous pourrions utiliser un code qui nous serait fourni plus tard, mais nous avions besoin de fichier dans l'immédiat, c'est pour cela que nous avons choisis de faire **notre propre librairie** pour accéder à Facebook et ainsi obtenir quelque chose qui s'intègre mieux à notre programme.

Nous nous sommes tout d'abord inscrit en tant que développeur et avons créé l'application **Flapstaff** pour pouvoir avoir accès à l'API. Celle-ci étant plutôt bien documentée, nous avons pu rapidement **cerner les fonctions** dont nous aurions besoin, mais nous avons pu aussi nous apercevoir qu'il manquait quelque chose de fondamental : la capacité de récupérer **les amis des amis**.
L'API Facebook ne permet pas, pour le moment, de récupérer les amis des amis sans leurs autorisations préalables, ce qui est très limitant pour un programme qui veut constituer des fichiers avec plus que les amis d'une seule personne.
En revanche, elle permet de savoir si **2 personnes**, même si on ne les connaît pas, **sont amis**, ce qui s'avérera très utile.

Dans tous les cas, ceci comportait la nécessité de faire des requêtes sur un serveur et donc d'utiliser une librairie externe de type libCUrl pour pouvoir **dialoguer avec l'API** qui est beaucoup plus facile d'utilisation avec des langages web comme PHP.
La schéma initial était celui-ci : se connecter avec un compte, récupérer les amis, puis les amis d'amis, et ensuite **calculer les relations entre toutes ces personnes** pour obtenir un graphe proche de la vrai vie.

Rapidement, nous avons pu mettre en place la connexion et la recherche d'amis, mais la recherche d'amis d'amis ainsi que le calcul des relations se sont **révélés beaucoup plus problématique**.

Pour pouvoir obtenir les amis d'amis, nous avons dû utiliser un faux compte et visiter les pages des personnes dont on veut les amis pour pouvoir **scanner la page à la recherche des noms et IDs** (avec des expressions régulières). Bien sûr, cela n'est possible que si la personne n'a pas "protégée" ses amis. Le **grand nombre de requêtes** que cela entraîne nous a assez vite

mis des bâtons dans les roues car Facebook finissait pas ne plus fournir de page valable...

La recherche d'amis supplémentaire devait néanmoins respecter une **certaine éthique**, même sur Facebook. Bien que le programme puisse être facilement détourné, il est conçu pour ne pouvoir récupérer que les amis de vos amis et pas plus loin. Juste suffisamment pour combler le trou dans l'API. Vous aurez besoin d'être loggé et d'avoir un ID Facebook valide, ainsi qu'avoir autoriser l'application Flapstaff à accéder à vos données. **De cette manière on protège un peu plus la vie privée des amis d'amis tout en se rapprochant du comportement de l'API**.

Le calcul des relations quant à lui consiste en un algorithme plutôt brutal. Il devait, pour tous les couples possibles dans une liste d'amis (*(n * n-1) / 2*), envoyer une requête à Facebook pour connaître le status de la relation entre les 2 personnes. Seulement, on arrive rapidement à un nombre **astronomique** de requêtes pour des groupes d'amis un peu conséquent. C'est pourquoi, nous avons choisi d'effectuer une seule requête avec tous les couples d'un coup. Mais l'API de Facebook ne permettait que ce format :

```
ids1=1, 2, 3 ids2= 4, 5, 6
```

pour comparer `1,2 et 2,5 et 3,6`. De ce fait, on a du former les couples de manière **primaire** (`ids1=1, 1, 1 ids2= 4, 5, 6`) pour obtenir un résultat efficace. Mais là encore, nous avons été confronté à un problème de temps. La formule donnée plus haut, sur plus de 2000 personnes donne des chiffres très grands et Facebook était amené à renvoyer des pages de plus de 100Mo et souvent ne pouvait pas compléter la requête.

Nous avons alors cherché la limite de réponse au delà de laquelle **le risque d'obtenir une liste de réponse vide est trop grand** pour partitionner la requête. Celle-ci varie entre 200 000 et 300 000 tests (environ 15Mo renvoyé). On calcul alors le nombre total de relations que l'on divise par 250 000 pour obtenir le nombre de requêtes nécessaires.
Et de cette manière, nous avons pu obtenir des **fichiers de tests complet et proche d'une situation réelle et probable**.

Vu le temps que cela peut prendre (jusqu'à 30 minutes sur les très grands groupe d'amis car Facebook met beaucoup de temps à répondre (entre 1 et 2 minutes)), nous avons aussi rajouté une option permettant de générer des relations aléatoire entre les personnes pour avoir des fichiers un peu plus variés. Car pour coller à la réalité, les relations d'amitiés sont **dans les 2 sens**, ce qui ne fournit dans le cas réel qu'**une seule grosse composante fortement connexe**.

C'est donc avec une grande variété de fichier de test que nous avons pu mettre à l'épreuve notre programme pour déceler des bugs dans des cas obscurs. Le travail sur Facebook nous a notamment fait découvrir qu'ils utilisaient des ID à 15 chiffres qui ne rentrent pas des dans Integers classiques, nous avons alors dû utiliser des chaînes de caractères.

5.2 Stabilité

Pour avoir une **cohérence** certaine et une **stabilité** assurée, nous avons choisi d'utiliser la librairie d'Unit Tests "*CU*" couplé à *GCov* (GNU Coverage) pour être certain que **tout le code est testé**.

Les Units Tests sont une partie importante du développement et de la phase de test, car il permette de **tester continuellement que le comportement attendu des fonctions est toujours celui réalisé**, et que celui-ci n'a pas été impacté par des **changements externes**. Chaque fonction dispose d'un Unit Test, ce qui nous as permis d'obtenir un Code Coverage très proche de **100%** (la seule fonction qui n'est pas testée fait partie de la librairie Facebook, car elle nécessite de se logger sur un compte et de compter le nombre d'amis, et comme celui-ci est amené à varier, on ne peut pas faire un Unit Test fiable dans le temps).

Il nous ont aussi permis de cerner quelles fonctions prenaient le plus de temps à l'exécution, couplé avec le Code Coverage qui nous indique combien de fois sont appelées les fonctions pour voir celles qui valent la peine d'être **optimisé si nécessaire**. De la même façon, il nous aussi permis de nettoyer le code en enlevant les parties qui n'étaient pas utilisée malgré le passage sur différents cas.
Le rapport de test est donc obtenu en lançant les Unit Tests et en vérifiant le Code Coverage (`codeCoverage.sh` dans `"test"`).

Nous avons aussi été amené à utiliser **Valgrind** pour vérifier les éventuelles fuites de mémoire. Il nous a entre autres permis de trouver que **libCUrl** laissait derrière lui des traces de mémoire non libérée malgré tous nos efforts pour supprimer les fuites mémoire.

Par la même occasion nous avons pu nous rendre compte de différence entre la version compilée sur Mac OS et celle compilée sur un GNU/Linux. En effet, nous avons dû passer plusieurs heures à **rendre le code compatible pour les 2 systèmes** sans vraiment comprendre la raison des différences. Par exemple, la fonction `"rand"` sous Mac OS acceptait volontiers de prendre une nouvelle graine chaque fois alors que sous Linux, elle ne voulait qu'une seule et unique graine pour le programme entier sous peine de ressortir les mêmes nombres inlassablement.

5.3 Résultat Final

Dans son ensemble, le projet **respecte le cahier des charges** donné et comporte aussi l'ajout de quelques bonus. Le temps d'exécution varie assez sensiblement selon les paramètres du fichier source utilisé. En revanche l'utilisation de la **recherche de personnes importantes** ralentie considérablement le programme, de manière exponentielle.
Après des **tests avec des nombres de questions différents**, nous avons pu constater que celui-ci impacter directement le temps d'exécution pouvant même le faire dépasser une minute si celui-ci dépasse plus de 1000 sur plus de 3000 personnes.

La génération de fichier à partir de Facebook a été testée avec **des valeurs allant de 500 à 10 000 amis**. On se rend vite compte que c'est la **recherche de relations qui est très lente** car le nombre de requêtes grandit très vite. D'un autre côté, la récupération d'amis d'amis est plutôt rapide atteignant un maximum de 4 minutes pour 10 000 personnes. Il faut environ 10 minutes pour générer le fichier pour 2000 personnes avec les relations réelles.

Chapitre 6

Conclusion et perspectives

6.1 Synthèse

Tout au long du projet, nous avons cherché à obtenir **un code propre et cohérent pour avoir un programme qui soit stable, évolutif et maintenable sur le long terme**. Pour ce faire nous avons utilisé plusieurs outils de collaboration qui nous ont permis d'arriver à nos fins.

L'utilisation de BitBucket et de ses outils annexes s'est révélée extrêmement pratique et il aurait été **difficile d'envisager de coder en groupe sans un tel outil**. Le respect de la charte de programmation n'a pas toujours été une mince affaire, mais nous a permis d'y voir plus clair dans le code plus d'une fois (merci les indentation de **8 espaces** !).
Le travail avec une API et un serveur distant a été une très bonne expérience pour la création d'une librairie minimalise exploitant celle-ci. Car créer une interface pour une API s'est aussi essayer de **s'adapter à la manière de coder et de concevoir les choses des autres développeurs**.

Les procédés de tests utilisant les **Unit Tests et le Code Coverage** nous ont donné un petit aperçu de ce qui est pratiqué à grande échelle en entreprise, et ainsi de nous **familiariser avec des outils que nous seront très certainement amené à réutiliser dans un futur proche**. l'utilisation d'un débugger mémoire tel que Valgrind nous a aussi permis de prendre conscience de beaucoup de problèmes liés à la programmation approximative et "sale" de certains algorithmes et nous ont donc invités indirectement à nous rediriger **vers une approche papier** plutôt que directement dans l'éditeur de texte.

L'application directe des algorithmes sur les graphes vus en cours nous ont donné une autre vision de la chose, un côté plus **pragmatique** que la simple utilisation théorique. Cette découverte des réseaux sociaux sous forme de graphe nous a aussi fait découvrir beaucoup de chose quant à leur fonctionnement et la mise en place de certaines fonctionnalités comme la suggestion d'amis sur Facebook.

Le travail en groupe s'est donc avéré être **très formateur et les différents moyens mis en oeuvre pour coder et communiquer nous ont permis de trouver une bonne dynamique de groupe** qui a fait que le projet a été fini et testé dans les temps.

6.2 Améliorations

En revanche, il reste quelques points inachevés qui restent alors des améliorations possibles dans le futur.

On notera particulièrement l'utilisation tardive de de l'**intégration continue** avec Hudson CI (*http ://hudson-ci.org/*) qui sur les dernières étapes du projet s'est révélé être un **atout précieux**, et si il avait été mis en place plus tôt aurait permis un développement plus propre directement (respect de la charte de programmation, exécution des Unit Tests automatiques, etc...)

En ce qui concerne la librairie Facebook, il aurait été intéressant d'utiliser les **threads** pour permettre une exécution plus rapide de la recherche de relations en les parallélisant. Un autre point à prendre en considération est la recherche de personne importante qui reste pour le moment un algorithme brutal et sans subtilité (mais pas sans saveur !).

La dernière étape pour la librairie Facebook est la **libération du code** après une avoir rendu le code assez abstrait pour s'adapter à d'autres programmes. Elle **comblerait** ainsi, plutôt efficacement et en respectant la philosophie de l'API Facebook, une **lacune** dans ladite API en permettant la récupération des amis des amis de la personne loggée et qui a acceptée l'application.