Rapport projet de cours Théotime Le Hellard : mg (Gestionnaire de version / Git)

23 mai 2022

La page "*Write yourself a Git !*" : Pour débuter j'ai suivi le tutoriel de Thibault Polge. Vers la moitié je me suis mis à lire en diagonal pour voir où il menait et j'ai décidé de m'en détacher. C'est sûrement un excellent support pour faire un projet "*Vanille*", mais j'ai préféré faire le Git de façon un peu plus "*Rock 'n' roll*". Mon objectif était d'aboutir à un gestionnaire satisfaisant avec suffisamment de fonctionnalités. Là où le tutoriel mime le comportement de Git pour donner une ébauche de sa façon de faire. Enfin, je signale que je ne savais pas comment fonctionne Git. Ainsi même si l'idée générale reste proche, je m'en suis éloigné.

Les grosses différences : J'ai voulu minimiser les répétitions dans la mémoire. Exemple : une utilisatrice Alice a déjà sauvegardé un fichier *patate.ml* (version 1) ; elle le modifie légèrement et le ré-enregistre (version 2). Git dans sa version simple, enregistre la nouvelle version tout en gardant l'ancienne. Mais ces deux versions peuvent partager beaucoup en commun. mg lui ne veut garder en mémoire que la dernière version. On supprime le zip de l'ancienne version de *patate.ml* après avoir enregistré la nouvelle. Évidemment on ne veut perdre aucune informations, pour être capable de revenir sur nos pas les différences entre la nouvelle et l'ancienne version (le "delta" entre 1 et 2) sont inscrites dans le fichier de commit. Ainsi on ne sauvegarde que la version actuelle du projet.

Dans les faits la règle est plus développée. Par exemple si le fichier a été modifié sur plus de la moitié de ses lignes, il serait contre productif de noter toutes les différences dans le fichier de commit. Auquel cas on ne s'embête pas, on garde l'ancien et le nouveau zip.

La règle se complique surtout avec l'arrivée des branches. ATTENTION les "branches" en mg ne sont pas des "branches" de Git. Une branche pour mg est une instance de mémoire. Contrairement à ce que j'ai dit précédemment, on ne sauvegarde pas forcément la version la plus actuelle du projet. On peut appliquer (`forward`) ou défaire (`backward`) un commit. Ici Alice peut décider de `backward` son dernier commit pour revenir sur la version 1 de *patate.ml*. Pour cela on ré-ouvre le fichier de commit, on décompresse la version 2 et on reconstruit la version 1 en appliquant le delta. Ensuite on compresse et on enregistre la version 1, puis on supprime la version 2 (y compris le fichier compressé). À noter que ces opérations n'ont aucun impact sur les fichiers réels, elles impactent seulement la sauvegarde de Alice.

Imaginons maintenant que Alice souhaite travailler avec Bob, on souhaite une nouvelle branche Bob, ie une nouvelle instance de la mémoire. Pour débuter la branche Bob est une copie de celle d'Alice, ainsi actuellement il a la version 1 de *patate.ml*. Une branche est en fait un arbre de l'état du répertoire de travail. Une arborescence des dossiers connus et pour chaque fichiers une clé SHA qui donne la version. Ainsi pour créer la branche Bob on se contente de copier l'arbre et surtout dans aucun cas on ne garde deux copies d'une même version de *patate.ml*.

Maintenant Bob décide de `forward` le commit, il doit obtenir la version 2 de *patate.ml*. Puisqu'elle n'existe plus, à nouveau nous devons la créer en utilisant le delta. Mais cette fois-ci nous ne pouvons pas écraser la version 1 de la mémoire, puisqu'elle appartient à la branche d'Alice. Ainsi nous sommes contraints de garder les deux versions V1 pour Alice et V2 pour Bob. Ensuite, si Alice décide de `forward` le commit elle n'a pas besoin de reconstruire la V2 puisqu'elle existe déjà. En revanche la V1 ne sert plus à personne, on peut donc s'en débarrasser.

Ceci constitue le comportement de base de mg. Je vais étayer en présentant les différentes fonctionnalités.

Les fonctionnalités :

Sommaire :

POUR L'UTILISATION VOIR LE README.md

FIGURE 1 – Liste des commandes / options (mg -help)

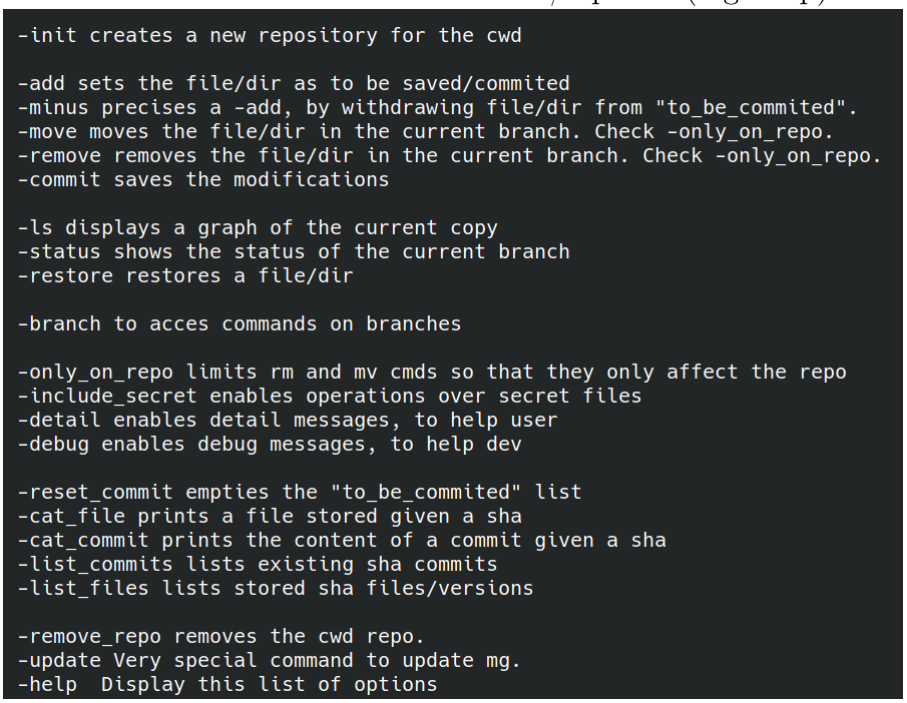
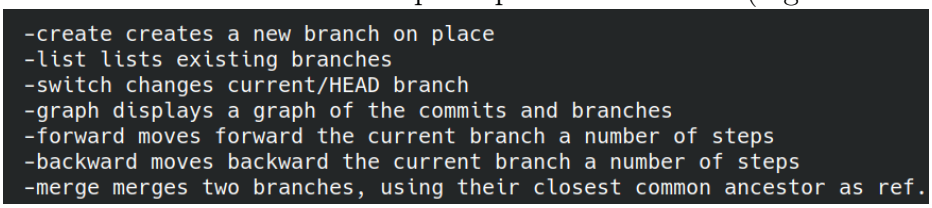


FIGURE 2 – Liste des commandes spécifiques aux branches (mg -branch -help)

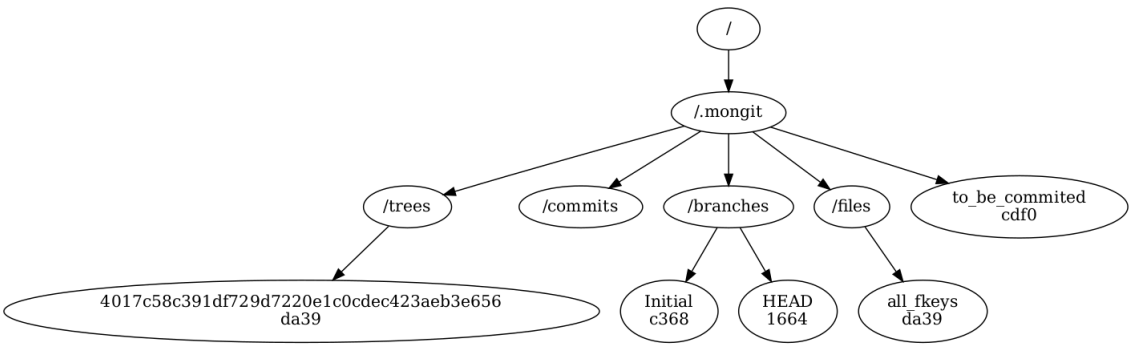


-init :

Crée un dossier `.mongit`, qui servira à stocker les fichiers compressés, l'état des branches etc. Voir figure ci-dessous. Contrairement à Git, je ne mélange pas tous les fichiers compressés sous une même notion d'object. J'utilise 4 sous dossiers :

- `/files` : Les fichiers(/blobs) sont zippés, et le nom du fichier est le SHA1 de son contenu. Contrairement à Git le fichier est compressé tel quel, sans meta-données au début. Comme Git, j'utilise les deux premières lettres du SHA pour faire des sous-dossiers.
- `/commits` : les commits eux aussi sont compressés. Puisque j'y note les deltas de tous les fichiers modifiés lors du commit, ils peuvent être lourd.
- `/trees` : Les trees que j'utilise sont plus simples que ceux de Git. Ils servent à retenir quels fichiers sont suivis par une branche donnée. Ainsi pour chaque dossier `dir` existant pour une branche `br`. Le fichier nommé `SHA("br : dir")` contient la liste des sous dossiers suivis ainsi que la liste des fichiers, avec leur SHA (qui donne ainsi la version, et comment le retrouver dans `/files`).
- `/branches` : Le fichier `HEAD` contient le nom de la branche avec laquelle on travaille actuellement. Pour chaque branche `br`, il existe un fichier au nom de la branche où est écrit le nom du dernier commit (ou "None" si la branche est vierge). On n'a pas besoin de retenir le nom du tree associé puisqu'il est nommé par le nom de la branche (`SHA("br : ")` pour la racine). Connaître le nom du dernier commit est indispensable pour faire du lien entre les commits (pour pouvoir `forward` ou `backward` sans avoir à préciser le SHA du commit).
- Le fichier `to_be_commited` liste les actions en attente d'un commit.
- Le sous fichier `/files/all_fkeys` sert à savoir quelles branches utilisent quels Blobs. Dans l'exemple d'introduction :
  ⇒ Avant le `forward` de Bob, il est écrit `«SHA(V1 patate.ml) : Alice Bob»`.
  ⇒ Quand Bob `forward`, on est contraint de reconstruire la V2 puisqu'elle n'existe pas. `/files/all_fkeys` devient `«SHA(V1 patate.ml) : Alice; SHA(V2 patate.ml) : Bob;»`. La V1 est encore utile, donc on ne la supprime pas.
  ⇒ Puis quand Alice `forward`, `/files/all_fkeys` devient `«SHA(V2 patate.ml) : Alice Bob»`. La version 1 est désormais inutile, on peut la supprimer.

FIGURE 3 – .mongit à la création, figure générée en utilisant mg -add ".mongit" puis mg -ls



Pré-commit / Commit :

— **-add** On écrit la requête dans le fichier `to_be_commited`. L'utilisateur donne les fichiers en chemin relatif à où il se trouve, mais on enregistre les chemins relativement à la racine du dossier de travail. On peut évidemment ajouter tout un dossier à la fois. À noter que «`mg -add "."`» utilisé dans la dossier racine a un comportement particulier, on la retranscrit par un `all add` contrairement à `add truc/patate.ml` par exemple. (On fait attention à ne pas l'interpréter comme "ajouter un dossier appelé add".)

— **-minus** : sert à préciser le `-add` par exemple si on veut ajouter tout un dossier excepté l'un de ses sous-dossiers on peut enchaîner `mg -add machin` ; `mg -minus machin/truc.ml`.

— **-move** : sert à déplacer un fichier. Idem on écrit d'abord dans `to_be_commited` et c'est seulement le `mg -commit` qui déclenchera l'action. Attention, pour éviter un décalage avec le répertoire de travail, par défaut le move déplace réellement les dossiers / fichiers (comme un `mv oldpath newpath`). Il faut utiliser l'option `-only_on_repo` si on ne veut pas impacter le vrai répertoire de travail. On écrit dans `to_be_commited` si l'option est utilisé ou non (`move_true` vs `move_false`).
Difficulté : il peut y avoir un décalage entre ce qu'on déplace véritablement et virtuellement. Si on a un dossier *machin* contenant deux fichiers : *patate.ml* et *truc.ml* ; et que seul le fichier *patate.ml* est suivi dans la branche actuelle. Virtuellement (dans l'arbre de la branche), on n'a que le dossier *machin* et le fichier *patate.ml* à déplacer. Mais puisque physiquement le dossier entier est déplacé (sauf option `only_on_repo`), de fait *truc.ml* doit suivre .
-remove idem

— **-commit** On compile le fichier `to_be_commited` et on applique tout. Comme pour Git, on peut préciser `-m <msg>`.

Outils sur la branche actuelle :

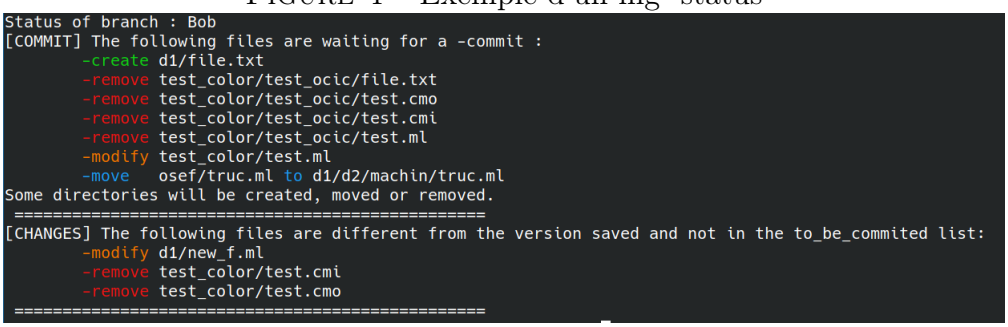
— **-ls** Affiche le graphe des fichiers suivis par la branche actuelle (exemple : la figure 3). Pour les fichiers on donne un abrégé de leur clé SHA pour les retrouver. Dans le détail on génère un fichier *repo_ tree.dot*, on le compile avec dot de Graphviz, puis on l'ouvre avec Evince. Quand l'utilisateur ferme Evince, on supprime *repo_ tree.dot* et *repo_ tree.pdf*. Toutefois vous pouvez utiliser l'option **-debug** pour les conserver (`mg -debug -ls`), si Graphviz ou Evince plante l'option est mise d'office.

— **-status** Git like. On utilise la même fonction que `-commit` pour compiler `to_be_commited` et on calcule le SHA des fichiers suivis pour savoir si ils ont changé.

— **-restore** Idem même comportement que Git.

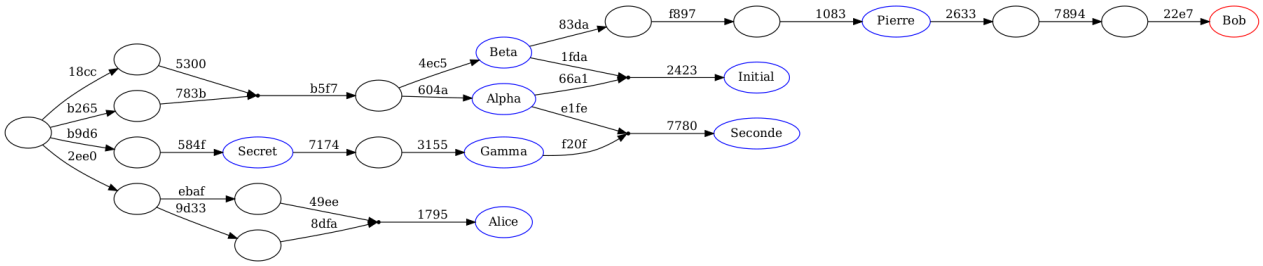
— **Outils annexes :** pour compléter un -ls ou pour choisir un -branch -forward :
`mg -list_commits` et `mg -list_files` pour retrouver les SHA entiers
`mg -cat_commit <SHA>` et `mg -cat_file <SHA>` pour les afficher

FIGURE 4 – Exemple d'un mg -status



Les branches :

FIGURE 5 – Exemple d'un mg -branch -graph



Comme expliqué en introduction, les branches sont des états de mémoire.

— **-branch -create
** crée une nouvelle branche dans le même état que la branche actuelle. On copie l'arbre associé et partout où est mentionné la branche actuelle dans `all_fkeys` on rajoute la nouvelle.

— **-branch -switch
** change de branche actuelle, ce qui demande extrêmement peu de travail. Il suffit de changer le nom de branche actuelle dans le fichier `branches/HEAD`.

— **-branch -list** donne la liste des branches existantes, la branche actuelle est écrite en bleu.

— **-branch -graph** affiche le graphe des commits et des branches. Les commits sont des arêtes orientées entre deux états de mémoire. Ensuite je place les branches dans les nœuds du graphe. Le nœud contenant la branche actuelle est en rouge.

— **-branch -forward <nb de pas>** avance la branche actuelle en suivant le graphe des commits. Toutefois si il y a une bifurcation, on demande à l'utilisateur de choisir.

— **-branch -backward <nb de pas>** Idem.

— **-branch -merge <branche 1> <branche 2> [branche produite]** on prend deux états de mémoire et on les amène vers un même état. Pour limiter le nombre de branches, un merge supprime une branche (elles étaient arrivées au même état !). Par défaut il garde la branche 1, mais on peut préciser un nouveau nom pour la branche produite (qui peut être distincte des deux branches instigatrices).

FIGURE 6 – Exemple d'un mg -branch -forward 8

```
The branch Alice cannot be moved forward after "none..." because several commits are possible :
"1" : 18cc8a88d8d46a5cc471f123390175aa533edf81
"2" : 2ee0ead0a7b3956f62d1537133666aed0d670d68
"3" : b265e50f2d0452be7b8df1884709d474c6eedc16
"4" : b9d60c184385abbf257f442aefae282d4b454224
2
Forward : branch Alice -> 2ee0...
The branch Alice cannot be moved forward after "2ee0..." because several commits are possible :
"1" : 9d33c85804c8a512c846d1843e9bca4a76e582a2
"2" : ebaf949bc02a034bb4108e26fd8f9ed0a360c195
1
Forward : branch Alice -> 9d33...
Forward : branch Alice -> 8dfa...
Forward : branch Alice -> 1795...
```

FIGURE 7 – Exemple d'un mg -branch -backward 10

```
"2423..." is a merge commit, to move backward branch Alpha you need to chose which commit you want to follow :
"1" : 1fda737b6083a4863b168b219cfd18090b3cb6a3
OR
"2" : 66a1ba65939f6153848b85077013f3f66e0ffb47
1
Backward : branch Alpha <- 1fda
Backward : branch Alpha <- 4ec5
"b5f7..." is a merge commit, to move backward branch Alpha you need to chose which commit you want to follow :
"1" : 5300c18fcccd0a8d1cf01d1943320bd69f8ca742
OR
"2" : 783bdfbb91086eaf532e5fe494434a29f6535e88
1
Backward : branch Alpha <- 5300
Backward : branch Alpha <- 18cc
```

Difficultés avec les branches :

L'exemple de la Figure 5 rassemble toutes les difficultés rencontrées. Tout d'abord il faut comprendre comment est construit ce graphe. Pour reconnaître les "états de mémoire" je n'ai pas d'autre solution que de me reposer sur les last commits. Pour savoir si deux branches sont dans le même état je pourrais tester l'égalité de leur arbre mais ce n'est pas généralisable en une relation d'ordre (pour forward et backward). Dans mon implémentation un état de mémoire est strictement caractérisé par le commit qui y mène. Ainsi l'état racine est simplement l'état associé au commit "none". Pour faire le branch graph j'associe à chaque commit un nœud initialement vide, puis j'écris une arête au nom du commit entre son nœud et le nœud de son commit parent. Pour placer une branche il me suffit de la rajouter au nœud de son last commit.

Les merges compliquent les choses. Lors d'un merge, en partant de deux états différents on veut arriver à un état produit unique. Mais puisqu'on part de deux états différents, on a deux deltas différents pour atteindre l'état produit. Exemple : si pour *patate.ml* on garde la version de Bob et que pour *truc.ml* on garde la version d'Alice, alors dans le delta depuis la position d'Alice il y aura écrit comment modifier *patate.ml* tandis que l'autre delta contiendra les modifications à apporter à *truc.ml* depuis la position de Bob. Ainsi il faut écrire deux commits. Mais ces commits mènent vers le même état, ce qui pose problème à l'idée de caractériser un état par le commit qui y mène. Il est indispensable de transcrire le fait que ces commits amène vers le même état. Le graphe ne sert pas qu'à faire jolie, il est nécessaire pour merge. En effet, pour merge deux branches, *i.e.* deux états, on a besoin de retrouver leur état ancêtre commun le plus proche. Pour ce faire on fait deux parcours en largeur dans le graphe depuis les états actuels, on trouve leur plus proche ancêtre et un chemin de commits à suivre pour y aller. (On crée une branche temporaire, qu'on backward jusqu'à cet état ancêtre pour reformer les fichiers nécessaires, à la fin on supprime cette branche.)

J'appelle commit résultant 1 (resp 2) le commit de delta entre l'ancienne position de la branche 1 (resp 2) et l'état produit. Pour résoudre le problème d'unifier les états d'arrivés des deux commits résultants, j'utilise un troisième commit. Ce commit de "merge" contient seulement le SHA des deux commits résultants. Ainsi je différencie les commits "simples" et ceux de "merges". Les commits simples vont d'un état vers un autre, ce qui est d'ailleurs le cas des commits résultants. Tandis que les commits de merges sont juste des tampons pour lier deux résultants.

Pour construire le graphe je commence par regarder tous les commits de merges. Je construis une table qui à un commit résultant associe son commit de merge. De plus, j'associe deux états aux commits de merges, un de "jointure" et l'autre de "résultat". Ensuite je peux attribuer à chaque commit simple un état, si il s'agit d'un résultant alors il pointe vers l'état de "jointure" du commit de merge associé. Sinon je crée juste un état associé, comme je le faisais avant.

Pour se déplacer dans ce graphe, si une branche se trouvant dans l'état "résultat" d'un commit de merge veut `backward` alors je demande quel commit résultant emprunter. Dans aucun cas je m'arrête à l'état de jointure, le commit de merge est juste un tampon. Idem pour `forward`, si il s'agit d'un commit résultant, je l'applique (le delta) puis je pousse la branche directement vers l'état "résultat" du commit de merge associé.

De fait aucune branche ne peut se retrouver dans un état de "jointure". Graphiquement je réduis ces états à de simple point. Dans l'exemple de la Figure 5, pour aller de "Alpha" à "Seconde" je suis d'abord le commit "résultant" e1fe... puis je poursuis nécessairement par le commit "merge" 7780...

Autres difficultés pour les branches :

L'unicité des commits Pour des commits classiques, deux commits partant du même état et apportant les mêmes modifications sont strictement identiques (même SHA), ce qui tombe sous le sens. Mais en cas de merge, on ne veut pas qu'un même commit résultant 1 puisse avoir deux commits résultants 2 associés.
Exemple de la Figure 5 : supposons qu'en partant de "Alpha" 66a1 consiste à supprimer *patate.ml* et e1fe idem. Techniquement ils partent et amènent vers le même état. D'ailleurs puisqu'ils ont le même contenu, il devrait porter le même SHA. Mais ils sont respectivement associés à 1fda... et à f20f... il fallait que je les différencie ! (sinon ça casse complètement ma solution développée page précédente) Pour cela j'ai renommé les deux commits résultants. Normalement tous les commits ont comme nom le SHA de leur contenu. Mais dans ce cas, j'ai nommé le résultant 1 SHA(SHA(res 1)SHA(res 2)) et le résultant 2 SHA(SHA(res 2)SHA(res 1)). D'où le fait que 66a1 et e1fe n'aient pas le même nom en ayant le même contenu.

All_fkeys La gestion d'`all_fkeys` est l'un des points les plus difficiles. C'est même le nerf de la guerre puisque c'est ce qui détermine quand peut-on supprimer un fichier zip (*ie* plus aucune branche en a besoin).
Reprenons l'exemple de la Figure 5, supposons que lors du commit 2633 (en haut à droite) Bob a créé un fichier "bidule.ml", puis qu'il l'a modifié légèrement en 7894 et à nouveau en 22e7. D'après les règles énoncées précédemment, au moment du commit 7894 personne d'autre que Bob a besoin de "bidule.ml", il le modifie et ne garde que le zip de la dernière version, idem avec 22e7. Ainsi Bob fait avancer l'unique copie de "bidule.ml". Mais lorsque Pierre voudra `forward` 2633 il ne trouvera pas le fichier "bidule.ml". Et il ne peut pas le construire en lisant le commit 2633. La solution pour laquelle j'ai opté c'est de dire que lorsqu'un fichier est créer, toutes les branches en ont besoin (dans all_fkeys). Si un jour Alice veut backward 4 puis forward 10 pour rejoindre Bob, elle aura aussi besoin de la version originale de "patate.ml" pour ensuite appliquer tous les deltas. (On pourrait considérer d'autres solutions, par exemple aller chercher la version de Bob et lui appliquer toutes modifications pour retrouver une version originale pour Pierre).

All_fkeys bis Si deux fichiers ont strictement le même contenu, alors ils auront le même SHA. Donc une branche peut avoir besoin 2 fois du même SHA. (All_fkeys étant une table qui associe à un SHA quelles branches en ont besoin). Il faut faire attention à accepter les redondances.

Remarque sur le commit : Initialement un commit faisait des modifications et les notait en même temps dans le fichier de commit. Mais avec l'ajout des branches ce n'est plus du tout le cas. Maintenant faire `mg -commit` revient à écrire le fichier delta qui synthétise les modifications et écrit le fichier de commit. Ensuite on lance automatiquement un forward de ce commit, qui va réellement appliquer les modifs aux fichiers zip.

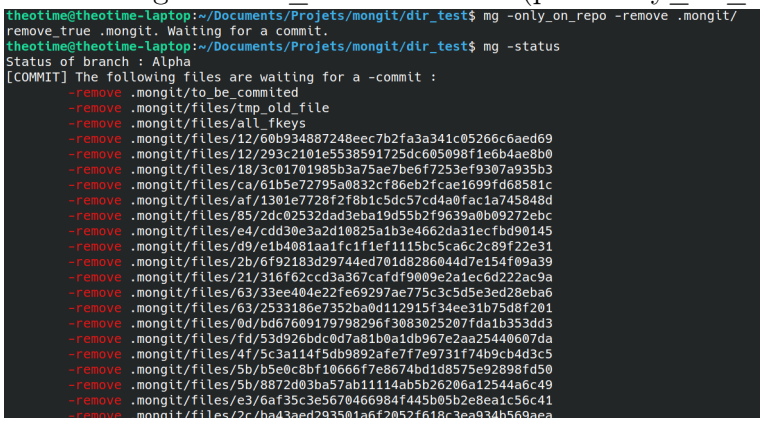
Options :

En plus de l'option `only_on_repo` utile pour move et remove. Il existe deux options `-detail` `-debug` pour étoffer les messages d'aides. À noter que `-detail` est à destination de l'utilisateur, alors que `-debug` est plus un outil pour le développeur.

Enfin il existe une option `-include_secret` pour inclure les sous-fichiers secrets (*ie* commençant par un '.'). Par défaut les commandes ne les impactent pas. Par exemple faire `mg -add "."`

pour tout ajouter, n'inclura pas le dossier `.mongit`. Pour inclure un fichier secret on peut l'ajouter l'explicitement, exemple : `mg -add ".mongit"` ; ou utiliser l'option `mg -include_secret -add "."`. C'est d'ailleurs ce que j'ai fait pour l'état "Secret" de la Figure 5, de fait le `mg -ls` est illisible avec tous les fichiers zip :

FIGURE 8 – Impact d'un mg -include_secret -add "." (puis -only_on_repo -remove ".")



Dernières commandes :

-remove_repo Supprime le dossier "./.mongit" donc ne s'applique qu'en étant dans le dossier racine du projet.

-update est une commande très spéciale pour mettre à jour mg, voir le README.md.

TODO

Les conflits de merge : Pour le moment ma résolution des conflits est très limité :
- lorsque par rapport à l'état de référence, la branche 1 a supprimé un fichier tandis que la branche 2 l'a modifié ; je demande à l'utilisateur ce qu'il veut faire.
- lorsque les deux branches ont modifié le même fichier, je demande à l'utilisateur quelle version il veut garder.
- lorsque les deux branches ont créé un fichier au même nom mais au contenu différent, je prends toujours la version de la branche 1.

C'est le gros manque de mg, TODO... Sachant que j'ai déjà refait une fonction scan_diff pour trouver les différences entre des fichiers (en essayant de regrouper au maximum par paquets de lignes modifiées / ajoutées ou supprimées).

Améliorer all_fkeys Je ne suis pas encore sûr de ce que je préfère comme fonctionnement pour all_fkeys (cf le paragraphe précédent). C'est d'ailleurs pour ça que je n'ai pas laissé de commande `mg -branch -delete`. Même si dans la version actuelle une telle commande ne poserait pas de problème.

Un mg_ignore : parce que ce serait simple à faire.

Des commits raccourcis entre états : l'idée est de rajouter une commande pour créer des commits raccourcis entre deux états. Exemple dans la Figure 5 : créer une arête entre l'état où se trouve Gamma et celui où se trouve Bob, il suffirait de calculer le delta approprié et ainsi nous pourrions passer d'un état à l'autre sans backward 4 et forward 10. On pourrait aussi envisager d'unifier les états identiques au sens "ceux qui ont la même arborescence de fichier".