S3:E1 – Prise en main de Git et du serveur GitLab

LICENCE INFORMATIQUE : PROGRAMMATION NÉCESSAIRE

Stéfane

1 Introduction

Ce document peut être téléchargé depuis Arche... ce que vous venez de faire !

1.1 Git et la gestion de versions

Git est un gestionnaire de versions, c'est à dire un logiciel qui permet de conserver l'historique des fichiers sources d'un projet, et d'utiliser cet historique pour fusionner automatiquement plusieurs révisions (ou *versions*) ; le terme anglophone pour la gestion des versions est *versioning*.

Chaque membre de l'équipe travaille sur sa version du projet et peut envoyer les versions suffisamment stables à ses coéquipiers via un dépôt partagé (commande `git push`) qui pourront les récupérer et les intégrer aux leurs quand ils le souhaitent (commande `git pull`).

Il existe d'autres gestionnaires de versions. La page https://fr.wikipedia.org/wiki/Gestion_de_versions vous en donne un aperçu de l'existant. Si, après ce premier travail pratique qui vous permettra de prendre en main les outils nécessaires à la réalisation du projet de synthèse, vous souhaitez approfondir vos connaissances sur Git, le livre https://git-scm.com/book/fr/v2 vous le permettra.

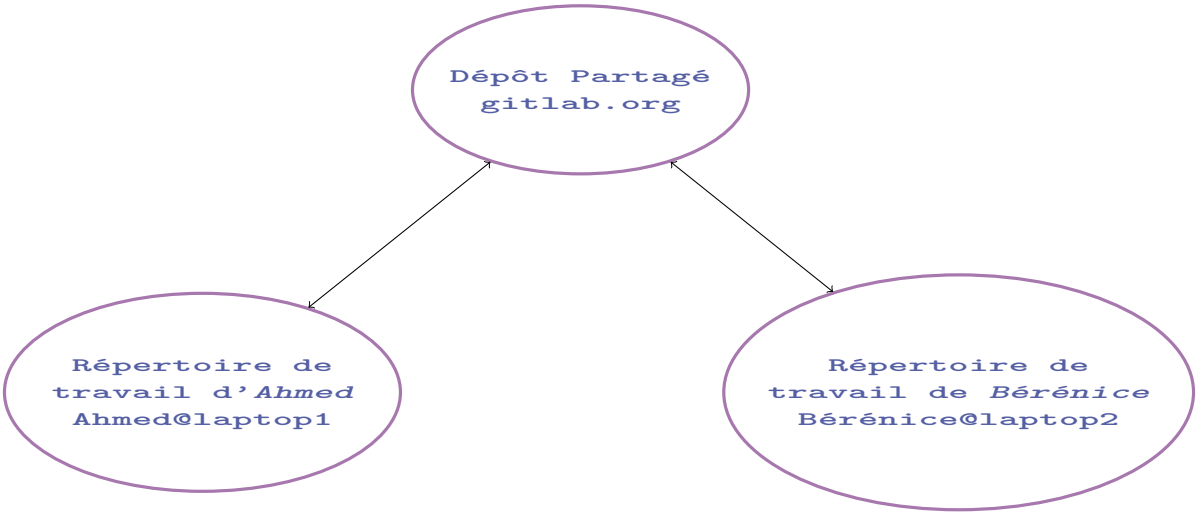
Cette séance de travaux pratiques est prévue avec le squelette du code `canevas.tgz` à télécharger depuis Arche. Les équipes sont de deux ou trois personnes que l'on nommera *Ahmed*, *Bérénice* et, pour les équipes de trois, *Chen*. *Ahmed* sera considérée l'initiatrice du projet partagé ; vous devez donc vous répartir ces rôles pour ce travail pratique.

1.2 Gestion des versions par Ahmed

Pour cette séance, choisissez deux (ou trois) ordinateurs adjacents par équipe ; vous pouvez également utiliser vos ordinateurs portables. Chaque étudiant.e travaille sur son compte.

On choisit le compte de l'étudiant.e qui créera le projet partagé sur `https://gitlab.org` : ce dépôt doit être accessible en permanence donc hébergé sur un serveur). Ce dépôt sera simplement un répertoire qui contiendra l'ensemble de l'historique du projet. On ne travaillera jamais dans ce répertoire directement, mais on utilisera Git pour envoyer et récupérer des révisions. Tous les membres de l'équipe auront accès à ce dépôt en lecture et en écriture, donc le choix du compte hébergeant le dépôt n'a pas beaucoup d'importance.

L'équipe est constituée d'*Ahmed* (qui travaille plutôt sur son portable, `laptop1`), de *Bérénice* (qui travaille également sur son portable `laptop2`) et, pour les équipes de trois, de *Chen* (qui travaille également sur son portable `laptop3`). Les explications sont écrites pour 2 utilisateurs pour simplifier, mais il peut y avoir un nombre quelconque de coéquipiers. Dans la suite des explications, on suppose que l'utilisateur *Ahmed* héberge le dépôt `https://gitlab.org`.



2 Configuration de Git

Si vous travaillez sur votre machine personnelle, vérifiez que Git est installé (la commande `git`, sans argument, doit vous donner un message d'aide). Si ce n'est pas le cas, installez le.[1]

On commence par configurer l'outil Git. Sur la machine sur laquelle on souhaite travailler (donc sur vos portables dans notre exemple) :

```
vscode ~/.gitconfig
```

Le contenu du fichier `.gitconfig` (à créer s'il n'existe pas) doit ressembler à ceci :

1. sous Ubuntu, `apt-get install git gitk` ou `apt-get install git-core gitk` devrait faire l'affaire, ou bien visitez le site : `http://git-scm.com/`.

```
[user]
  name = Prenom_Nom
  email = email@etu.univ-lorraine.fr
[push]
  default = simple
```

La section `[user]` est obligatoire, elle donne les informations qui seront enregistrées lors de l'enregistrement d'une version locale de votre projet avec la commande `git commit`. Nous vous demandons de respecter la syntaxe, à savoir `Prenom_Nom` pour le champ `name`. Il est conseillé d'utiliser votre vrai nom (pas juste votre login), votre adresse courriel officielle à l'UL, et d'utiliser la même configuration sur toutes les machines sur lesquelles vous travaillez. Il est conseillé de conserver une copie du fichier `.gitconfig` en lieu sûr; par exemple votre partition `z :`, un cumulonimbus (*cloud*) de votre choix ou encore une clef usb.

La section `[push]` permet d'avoir un comportement raisonnable avec les branches quand on utilise un dépôt partagé (c'est le cas par défaut avec Git 2.0). Avec les versions anciennes de Git, il faut utiliser `current` à la place de `simple`.

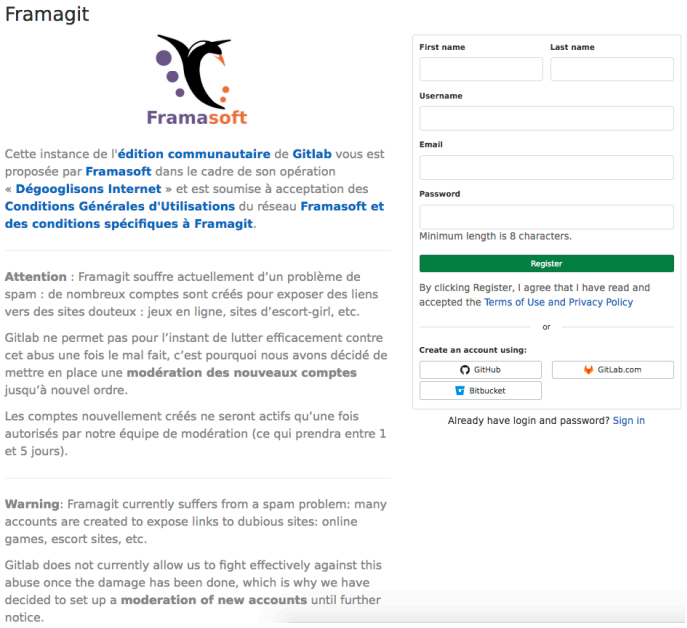
3 Mise en place

Le contenu de cette section est réalisé une bonne fois pour toute, au début du projet, principalement par *Ahmed*. Si certains membres de l'équipe ne comprennent pas les détails, ce n'est pas très grave, nous verrons ce que tout le monde doit savoir dans la section 4.

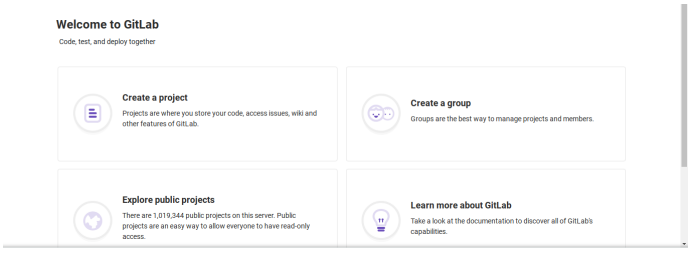
3.1 Création du dépôt partagé

On va maintenant créer le dépôt partagé. *Ahmed* et *Bérénice* créent leurs comptes sur le site `https://gitlab.com` comme l'illustre la figure 1a.

Dès que tous les membres sont enregistrés, *Ahmed* crée le projet comme l'illustre la figure 1b. Celui-ci sera vide à l'exception du répertoire de configuration `.git` qui ne doit pas être modifié car il sert de communication entre la version local et les versions distantes sur `GitLab`. *Ahmed* donne un nom au projet (par ex. *projetSynthese*) **sans rien changer aux autres attributs** (cf. Fig. 2a). Ensuite *Ahmed* donne accès au projet à *Bérénice* en tant que *Maintainer* en passant par le sous-menu *Members* du menu *Settings* (cf. Fig. 2b).

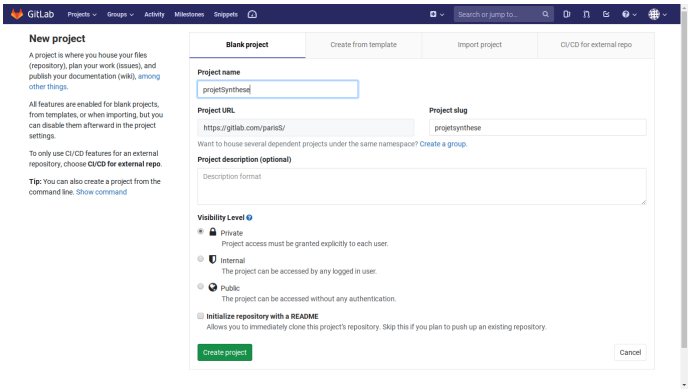


(a) *Créez un compte sur GitLab.*

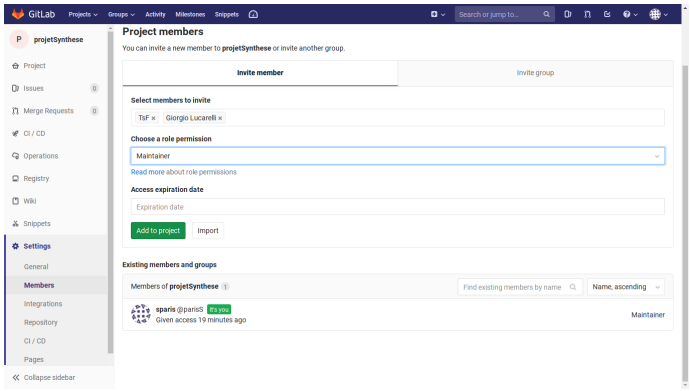


(b) *Créez le projet sur GitLab.*

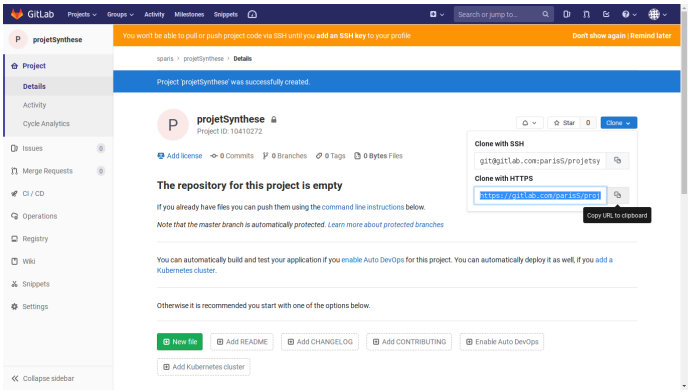
FIGURE *1* – *Création d'un compte* `GitLab` *et du dépôt sur GitLab par Ahmed. Dans la copie d'écran, le propriétaire est sparis qu'il faut remplacer par Ahmed dans ce sujet.*



(a) *Renseignez les attributs du dépôt en y apportant un nom. Par exemple* `TPGit`*. Ne rien changer d'autre.*



(b) *Enregistrer les autres membres de l'équipe en tant que Maintainer.*



(c) *Récupérer l'adresse de clonage du dépôt.*

FIGURE 2 – *Configurer le dépôt sur GitLab par Ahmed et copie de l'adressage de clonage du dépôt.*

Une fois toute l'équipe enregistrée, chaque membre peut récupérer le lien de clonage en version *https* (cf. Fig. 2c). Pour des raisons de sécurité interne à l'université, la commande `ssh` n'est pas autorisée. *Ahmed* récupère une copie du projet (vide pour l'instant) avec la commande :

```
>> git clone https://gitlab.org/Ahmed/projetsynthese.git
   projetsynthese
Clonage dans 'projetsynthese'...
Username for 'https://gitlab.org': Ahmed
Password for 'https://Ahmed@gitlab.org':
warning: Vous semblez avoir cloné un dépot vide.
```

où l'adresse de clonage `https://gitlab.org/Ahmed/projetsynthese.git` fait référence au projet *projetsynthese* du compte *Ahmed*. On choisit de cloner ce projet dans un répertoire *local* de même nom, à savoir *projetsynthese*. *Ahmed* devra se connecter avec ses identifiant et mot de passe.

Sur ARCHE, *Ahmed* récupère le canevas du projet, le développe dans le répertoire *local* du projet *projetsynthese* puis supprime l'archive avec la commande :

```
>> cd projetsynthese
>> tar xvf canevas.tgz
>> rm -f canevas.tgz
```

Ensuite *Ahmed* télédépose le projet sur le site gitlab.org avec la commande :

```
>> git add *
>> git commit -m ''dépot du canevas''
[master (commit racine) 804ccb0] dépot du canevas
 11 files changed, 292 insertions(+)
 create mode 100644 Makefile
 create mode 100644 data/input
 create mode 100644 include/geometry.h
 create mode 100644 include/list.h
 create mode 100644 include/rational.h
 create mode 100644 include/tree.h
 create mode 100644 src/algo.c
 create mode 100644 src/geometry.c
 create mode 100644 src/list.c
 create mode 100644 src/rational.c
 create mode 100644 src/tree.c
>> git push
Username for 'https://gitlab.org': Ahmed
Password for 'https://Ahmed@gitlab.org':
Décompte des objets: 16, fait.
Delta compression using up to 4 threads.
Compression des objets: 100% (15/15), fait.
Écriture des objets: 100% (16/16), 5.33 KiB | 0 bytes/s, fait.
Total 16 (delta 2), reused 0 (delta 0)
To https://gitlab.org/Ahmed/projetsynthese.git
 * [new branch]      master -> master
```

La commande `git` est générique :

— son argument `add <fichier1> ... <fichierN>` ajoute les fichiers indiqués à l'index du projet ; les règles des expressions régulières sont utilisables (i.e. wildcards) ; `*` désigne tous les fichiers ;

— son argument `commit -m message` enregistre les changements dans l'index du projet *local* et y adjoint un message ;

— son argument `push` met à jour le dépôt distant (sur GitLab) ; ceci se fait en conservant un historique de toutes les versions précédemment enregistrées.

Maintenant *Bérénice* peut cloner le dépôt avec la commande :

```
>> git clone https://gitlab.org/Ahmed/projetsynthese.git
   projetsynthese
Clonage dans 'projetsynthese'...
Username for 'https://gitlab.org' : Bérénice
Password for 'https://Bérénice@gitlab.org' :
remote : Enumerating objects : 16, done.
remote : Counting objects : 100% (16/16), done.
remote : Compressing objects : 100% (15/15), done.
remote : Total 16 (delta 2), reused 0 (delta 0)
Dépaquetage des objets : 100% (16/16), fait.
```

Le dépaquetage s'est effectué correctement : 16 objets sur 16 ont été récupérés. La commande `gitk` permet de visualiser l'historique des versions sur le serveur (cf. Fig. 3).

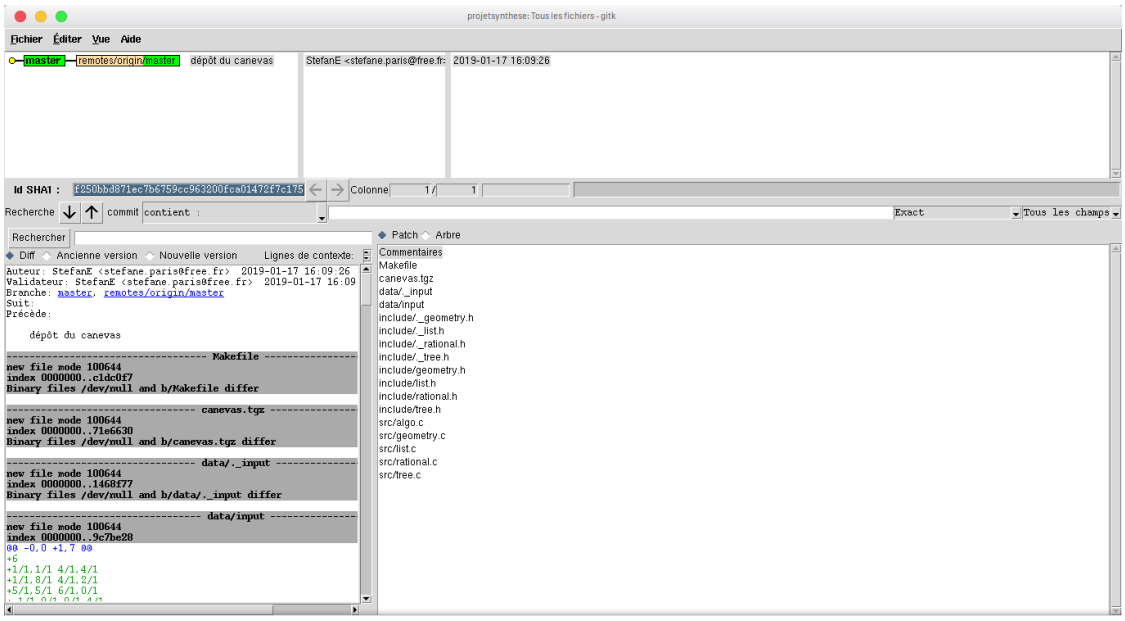


FIGURE 3 – *Avec le logiciel* `gitk` *vous pouvez visualiser le résultat du télédépôt.*

4 Utilisation de Git pour le développement

Pour commencer, *Ahmed* et *Bérénice* vont travailler dans leurs fichiers *locaux* respectifs `include/geometry.h`. Chaque membre a une tâche à remplir expliquée par les commentaires. *Ahmed* et *Bérénice* travaillent séparément. Une fois leur tâche accomplie, chacun.e télédépose son projet local sur `gitlab`.

4.1 Création de nouvelles révision

Visualisez les changements avec la commande en ligne `git status`. Vous devriez obtenir une réponse similaire à ce qui suit :

```
>> git status
Sur la branche master
Votre branche est à jour avec 'origin/master'.
Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera
     validé)
  (utilisez "git checkout -- <fichier>..." pour annuler les
     modifications dans la copie de travail)

    modifié :         include/geometry.h

aucune modification n'a été ajoutée à la validation
  (utilisez "git add" ou "git commit -m message")
```

Le message principal est que la modification du fichier `geometry.h` a bien été prise en compte. Vous pouvez également utiliser `gitk`. Notez que ces changements sont locaux ; *Ahmed* voit les siens mais pas ceux de *Bérénice* et réciproquement. Lancez la commande :

```
1   >> git diff HEAD
```

et expliquez ce qu'elle vous indique. Constatez vous (et alors comment) que *Ahmed* et *Bérénice* ont fait des modifications différentes ?

Notes.

— Les lignes commençant par '-' correspondent à ce qui a été enlevé

— les lignes commençant par '+' à ce qui a été ajouté par rapport au précédent commit.

Maintenant, *Ahmed* et *Bérénice* vont mettre à jour l'index de leur projet (local) avec la commande :

```
>> git commit -m "message"
```

L'éditeur est lancé et demande d'entrer un message de 'log'. Ajoutez alors des lignes et d'autres renseignements sur les modifications apportées à `geometry.h`. Un bon message de log commence par une ligne décrivant rapidement le changement, suivi d'une ligne vide, suivi d'un court texte expliquant pourquoi la modification est bonne.

On voit ensuite apparaître :

```
[master 98224e9] Première modification
 1 file changed, 2 insertions(+)
```

Ceci signifie qu'une nouvelle `commit` [2], parfois appelé *revision version*, du projet a été enregistrée dans le dépôt. Ce commit est identifié par une chaîne hexadécimale (`98224e9` dans notre cas).

À nouveau on peut visualiser ce qui s'est passé avec les commandes :

```
>> gitk              # Visualiser l'historique graphiquement
```

Puis séparément, *Ahmed* et *Bérénice* vont mettre à jour le dépôt. Assurez vous que *Bérénice* le fasse en premier.

```
>> git push
Username for 'https://gitlab.org': Bérénice
Password for 'https://Bérénice@gitlab.org':
Décompte des objets: 4, fait.
Delta compression using up to 4 threads.
Compression des objets: 100% (4/4), fait.
Écriture des objets: 100% (4/4), 563 bytes | 0 bytes/s, fait.
Total 4 (delta 1), reused 0 (delta 0)
To https://gitlab.org/Ahmed/projetsynthese.git
   c377263..96d9d1f  master -> master
```

2. Le terme français le plus approprié à ce verbe anglophone semble être réaliser.

Avant que *Ahmed* fasse son dépôt, *Bérénice* visualise le résultat du dépôt avec `gitk`. Quand *Ahmed* veut effectuer son dépôt, il ne peut le faire comme *Bérénice* car `git` constate que sa version n'est pas la dernière. Il doit donc opérer la commande `git pull` pour **fusionner** (*Merge*) ses modifications avec ceux de *Bérénice*. Cette commande indique le fichier concernée par la fusion :

```
Merge made by the 'recursive' strategy.
 include/geometry.h | 2 ++
 1 file changed, 2 insertions(+)
```

Pour que *Bérénice* puisse utiliser cette fusion des modifications, *Ahmed* effectue un `push` :

```
>> git push
Username for 'https://gitlab.org': Ahmed
Password for 'https://Ahmed@gitlab.com':
Décompte des objets: 8, fait.
Delta compression using up to 4 threads.
Compression des objets: 100% (8/8), fait.
Écriture des objets: 100% (8/8), 1.08 KiB | 0 bytes/s, fait.
Total 8 (delta 2), reused 0 (delta 0)
To https://gitlab.org/Ahmed/projetsynthese.git
   96d9d1f..d70fa95  master -> master
```

Pour terminer, *Bérénice* effectue un `pull` pour effectuer la fusion avec sa version ; si elle tente un `push`, elle obtient un message d'erreur comme celui-ci :

```
Username for 'https://gitlab.org': Bérénice
Password for 'https://Ahmed@gitlab.org':
To https://gitlab.org/Ahmed/projetsynthese.git
 ! [rejected]        master -> master (fetch first)
error: impossible de pousser des références vers 'https://gitlab.org
   /Ahmed/projetsynthese.git'
astuce: Les mises à jour ont été rejetées car la branche distante
   contient du travail que
astuce: vous n'avez pas en local. Ceci est généralement causé par un
    autre dépot poussé
astuce: vers la même référence. Vous pourriez intégrer d'abord les
   changements distants
```

```
astuce : (par exemple 'git pull ...') avant de pousser à nouveau.
astuce : Voir la 'Note à propos des avances rapides' dans 'git push
    --help' pour plus d'information.
```

Maintenant, *Ahmed* et *Bérénice* possède la dernière version où les types `Point` et `Segment` sont corrigés. Le résultat de la fusion peut être visualiser avec `gitk` ou bien avec `git log`.

4.2 Fusion ambiguë

Qu'en est-il lorsque *Ahmed* et *Bérénice* effectuent des modifications sur les mêmes lignes dans le même fichier ?

Supposons que sans se concerter *Ahmed* et *Bérénice* ajoute quelques lignes à la fonction `main` du fichier `src/test.c`. *Ahmed* crée deux segments en début de la fonction `main` :

```
// Segment d'origine (1,1) et d'extrémité (6,1)
struct Segment S1 = { { {1,1}, {1,1} }, { {6,1}, {1,1} } };
// Segment d'origine (1,1) et d'extrémité (6,6)
struct Segment S2 = { { {1,1}, {1,1} }, { {6,1}, {6,1} } };
```

Une fois les modifications apportées et validées localement, *Ahmed* effectue son dépôt :

```
>> git commit -m "les deux Segments de Ahmed"
>> git push
```

De son côté, *Bérénice* crée également deux segments de coordonnées respectives :

```
// Segment d'origine (6,1) et d'extrémité (6,6)
struct Segment S1 = { { {6,1}, {1,1} }, { {6,1}, {6,1} } };
// Segment d'origine (6,1) et d'extrémité (1,6)
struct Segment S2 = { { {6,1}, {1,1} }, { {1,1}, {6,1} } };
```

Puis *Bérénice* tente d'effectuer de même.

```
>> git commit -m "les deux Segments de Bérénice"
>> git push
To https://gitlab.org/Ahmed/projetsynthese.git
 ! [rejected]        master -> master (fetch first)
```

```
error : impossible de pousser des références vers 'https ://gitlab.org
   /Ahmed/projetsynthese.git'
astuce : Les mises à jour ont été rejetées car la branche distante
   contient du travail que
astuce : vous n'avez pas en local. Ceci est généralement causé par un
    autre dépôt poussé
astuce : vers la même référence. Vous pourriez intégrer d'abord les
   changements distants
astuce : (par exemple 'git pull ...') avant de pousser à nouveau.
astuce : Voir la 'Note à propos des avances rapides' dans 'git push
   --help' pour plus d'information.
```

Bérénice constate une erreur due au fait que *Ahmed* a enregistré une nouvelle version sur le serveur. Elle doit donc opérer une *fusion* avec sa version locale

```
>> git pull
Username for 'https ://gitlab.org' : Bérénice
Password for 'https ://Bérénice@gitlab.org' :
remote : Enumerating objects : 13, done.
remote : Counting objects : 100% (13/13), done.
remote : Compressing objects : 100% (7/7), done.
remote : Total 7 (delta 3), reused 0 (delta 0)
Dépaquetage des objets : 100% (7/7), fait.
Depuis https ://gitlab.org/Ahmed/projetsynthese
   d70fa95..16abec7  master     -> origin/master
Fusion automatique de src/test.c
CONFLIT (contenu) : Conflit de fusion dans src/test.c
La fusion automatique a échoué ; réglez les conflits et validez le r
   ésultat.
```

Cependant, ce n'est pas aussi désespéré que ce message pourrait le laisser penser. En consultant la fonction `main` de son fichier `src/test.c`, *Bérénice* peut visualiser les différences :

```
int main(int argc, char ** argv) {
<<<<<<< HEAD
  // Segment d'origine (6,1) et d'extrémité (6,6)
```

```
    Segment s1 = { { {6,1}, {1,1} }, { {6,1}, {6,1} } };
    // Segment d'origine (6,1) et d'extrémité (1,6)
    Segment s2 = { { {6,1}, {1,1} }, { {1,1}, {6,1} } };
=======
    // Segment d'origine (1,1) et d'extrémité (6,1)
    Segment s1 = { { {1,1}, {1,1} }, { {6,1}, {1,1} } };
    // Segment d'origine (1,1) et d'extrémité (6,6)
    Segment s2 = { { {1,1}, {1,1} }, { {6,1}, {6,1} } };
>>>>>>> 16abec77effa1d0e4e2ad3ac4cd910b5d7ecd004

    display_segment(s1);
    display_segment(s2);
}
```

Ainsi, *Bérénice* peut choisir les suppressions et/ou les modifications à apporter. Elle a reçu comme tâche de ne conserver (et de n'afficher) que les segments horizontaux et verticaux en n'oubliant pas de supprimer les commentaires ajoutés par `git`, à savoir :

```
<<<<<<< HEAD
=======
>>>>>>> 16abec77effa1d0e4e2ad3ac4cd910b5d7ecd004
```

Une fois les modifications faites elle dépose sa version.

```
>> git commit -m "version finale du TP1 par Bérénice"
[master e786d34] version finale du TP1 par Bérénice
>> git push
Username for 'https://gitlab.org': Bérénice
Password for 'https://Bérénice@gitlab.org':
Décompte des objets: 8, fait.
Delta compression using up to 4 threads.
Compression des objets: 100% (8/8), fait.
Écriture des objets: 100% (8/8), 784 bytes | 0 bytes/s, fait.
Total 8 (delta 6), reused 0 (delta 0)
To https://gitlab.org/Ahmed/projetsynthese.git
   16abec7..e786d34  master -> master
```

Maintenant si *Ahmed* tente de déposer à nouveau sa version, il reçoit un message d'erreur. Il retrouve la nouvelle version du serveur grâce à la commande :

```
git pull
Username for 'https://gitlab.org': Ahmed
Password for 'https://Ahmed@gitlab.org':
remote: Enumerating objects: 14, done.
remote: Counting objects: 100% (14/14), done.
remote: Compressing objects: 100% (8/8), done.
remote: Total 8 (delta 6), reused 0 (delta 0)
Dépaquetage des objets: 100% (8/8), fait.
Depuis https://gitlab.org/Ahmed/projetsynthese
16abec7..e786d34  master     -> origin/master
Mise à jour 16abec7..e786d34
Fast-forward
src/algo.c | 6 +++---
1 file changed, 3 insertions(+), 3 deletions(-)
```

Avec la commande `gitk` les modifications et leur historique sont visualisables, comme l'illustre la figure 4.

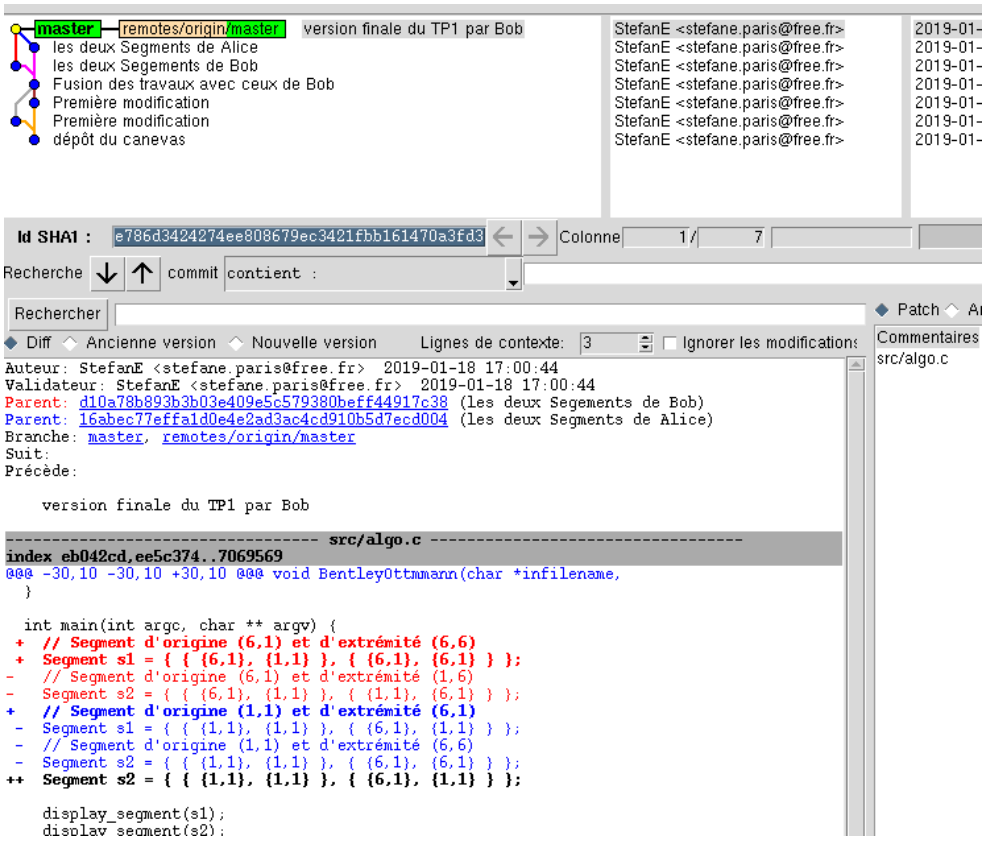


FIGURE 4 – *Visualisation des modifications enregistrées par Bérénice avec la commande* `gitk`.

4.3 Ajout de fichiers

À présent, *Ahmed* crée un nouveau fichier, `src/toto.c`, avec un contenu quelconque. Puis elle visualise le changement avec la commande :

```
>> git status
Sur la branche master
Votre branche est à jour avec 'origin/master'.
Fichiers non suivis :
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera
     validé)

    src/totot.c
```

```
aucune modification ajoutée à la validation mais des fichiers non
    suivis sont présents (utilisez "git add" pour les suivre)
```

Le fichier `src/toto.c` est considéré comme *Untracked* (non suivi par Git). Si *Ahmed* souhaite que `toto.c` soit ajouté au dépôt, il faut qu'il l'enregistrer (`git commit` ne suffit pas) : `git add src/toto.c`. Ainsi la prise en compte dans l'index du dépôt local du nouveau fichier :

```
>> git status
Sur la branche master
Votre branche est à jour avec 'origin/master'.
Modifications qui seront validées :
  (utilisez "git reset HEAD <fichier>..." pour désindexer)

    nouveau fichier  : src/totot.c
```

Ahmed met *effectivement* à jour le dépôt local avant de la télédéposer sur le serveur (`-m` permet de donner directement le message de log) :

```
>> git commit -m "ajout de src/toto.c"
[master a3a1ef1] ajout de src/toto.c
 1 file changed, 1 insertion(+)
 create mode 100644 src/totot.c
>> git push
Username for 'https://gitlab.org': Ahmed
Password for 'https://Ahmed@gitlab.org':
Décompte des objets: 4, fait.
Delta compression using up to 4 threads.
Compression des objets: 100% (3/3), fait.
Écriture des objets: 100% (4/4), 331 bytes | 0 bytes/s, fait.
Total 4 (delta 2), reused 0 (delta 0)
To https://gitlab.org/Ahmed/projetsynthese.git
   e786d34..a3a1ef1  master -> master
```

Maintenant, *Bérénice* télécharge le projet :

```
>> git pull
```

et vérifie que le fichier `src/toto.c` est maintenant présent.

4.4 Fichiers ignorés par Git

À son tour, *Bérénice* crée à présent un nouveau fichier `MONFICHIER.md`, puis fait :

```
>> git status
Sur la branche master
Votre branche est à jour avec 'origin/master'.
Fichiers non suivis :
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera
     validé)

    MONFICHIER.md

aucune modification ajoutée à la validation mais des fichiers non
   suivis sont présents (utilisez "git add" pour les suivre)
```

Si *Bérénice* souhaite que `MONFICHIER.md` soit *ignoré* par Git, autrement dit, *ne soit pas* enregistré dans le dépôt distant, elle doit inscrire le nom de ce fichier (`MONFICHIER.md`) dans un fichier appelé `.gitignore` *dans le répertoire contenant* `MONFICHIER.md`. Concrètement, *Bérénice* tape la commande :

```
nano .gitignore
```

Bérénice y ajoute une ligne

```
1 MONFICHIER.md
```

sauvegarde le fichier et visualise le changement : small

```
Sur la branche master
Votre branche est à jour avec 'origin/master'.
Fichiers non suivis :
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera
     validé)

    .gitignore
```

```
aucune modification ajoutée à la validation mais des fichiers non
    suivis sont présents
  (utilisez "git add" pour les suivre)
```

Si *Bérénice* souhaite que le nouveau fichier `.gitignore` soit accessible à tous les utilisateurs, elle met l'index à jour :

```
>> git add .gitignore
>> git status
Sur la branche master
Sur la branche master
Votre branche est à jour avec 'origin/master'.
Modifications qui seront validées :
  (utilisez "git reset HEAD <fichier>..." pour désindexer)

    nouveau fichier  : .gitignore
```

Quelques remarques :

— Le fichier *MONFICHIER.md* n'apparaît plus. C'était le but de la manœuvre. Une bonne pratique est de faire en sorte que `git status` ne montre jamais de *Untracked files* :

  — soit un fichier doit être ajouté dans le dépôt,

  — soit il doit être explicitement ignoré.

  Cela évite d'oublier de faire un `git add`.

— En général, on met dans les `.gitignore` les noms des fichiers générés (*.o, fichiers exécutables, ...), ce qu'il faudra adapter pour faire le `.gitignore` de *votre projet*.

— Les *wildcards* usuels (`*.o`, `*.ad?`, ...) sont acceptés pour ignorer plusieurs fichiers.

— Le fichier `.gitignore` vient d'être ajouté (ou bien il a été modifié s'il était déjà présent). Il faut à nouveau faire un `commit` et un `push` pour que cette modification soit disponible pour tout le monde :

```
git commit -m ".gitignore créé"
[master 248fc54] .gitignore créé
 1 file changed, 2 insertions(+)
 create mode 100644 .gitignore
```

```
projetsynthese »git push
Username for 'https://gitlab.org': Bérénice
Password for 'https://Bérénice@gitlab.org':
Décompte des objets: 3, fait.
Delta compression using up to 4 threads.
Compression des objets: 100% (2/2), fait.
Écriture des objets: 100% (3/3), 290 bytes | 0 bytes/s, fait.
Total 3 (delta 1), reused 0 (delta 0)
To https://gitlab.org/Ahmed/projetsynthese.git
   a3a1ef1..248fc54  master -> master
```

5 Pour conclure...

A ce stade, vous devriez avoir les bases pour l'utilisation quotidienne de Git. Il serait pertinent que vous fassiez à chaque fin de séance de travaux pratiques une sauvegarde sur le serveur du projet... d'autant que cet historique intervient dans la note finale. Pour utiliser Git sur un autre projet, on peut reprendre les explications du début de ce document (section 3.1) pour créer un nouveau dépôt Git. Si vous souhaitiez pour une raison quelconque, reprendre le canevas initial, il faut avant tout le supprimer :

```
>> git rm -r projetsynthese
>> git status            # Pour vérifier qu'on n'a pas fait de bêtise
>> git commit -m "Suppression de projetsynthese"
```

Si vous avez besoin d'importer un squelette de code, vous pouvez le faire par exemple avec ces commandes :

```
>> tar xzvf ~/chemin/vers/le/squelette.tar.gz
>> git add .
>> git commit -m "import du squelette"
```

Bien sûr, Git est bien plus que ce que nous venons de voir, et nous encourageons les plus curieux à se plonger dans le manuel utilisateur et les pages de man de Git pour en apprendre plus. Au niveau débutant, voici ce qu'on peut retenir :

Les commandes

— `git commit -m message` enregistre l'état courant du répertoire de travail,

— `git push` publie les commits,

— `git pull`] récupère les commits publiés,

— `git add`, `git rm` et `git mv` permettent de dire à Git quels fichiers il doit surveiller (*traquer* ou *versionner* dans le jargon),

— `git status`, `git diff HEAD` pour voir où on en est.

Conseils pratiques

— Ne *jamais* s'échanger des fichiers sans passer par Git (email, scp, clé USB), sauf si vous savez *vraiment* ce que vous faites.

— Toujours utiliser `git commit` avec l'option `-a`.

— Faire un `git push` après chaque `git commit -m message`, sauf si on veut garder ses modifications privées. Il peut être nécessaire de faire un `git pull` avant un `git push` si des nouvelles révisions sont disponibles dans le dépôt partagé.

— Faire des `git pull` régulièrement pour rester synchronisés avec vos collègues. Il faut faire un `git commit -m "message"` avant de pouvoir faire un `git pull` (ce qui permet de ne pas mélanger modifications manuelles et fusions automatiques).

— Ne faites jamais un `git add` sur un fichier binaire généré : si vous les faites, attendez-vous à des conflits à chaque modification des sources ! Git est fait pour gérer des fichiers sources, pas des binaires.

(Quand vous ne serez plus débutants vous verrez que la vie n'est pas si simple, et que la puissance de Git vient de `git commit` sans `-a`, des `git commit` sans `git push`, ... mais chaque chose en son temps !)

Quand rien ne va plus...

— Consulter le livre https://git-scm.com/book/fr/v2.

— Demander de l'aide sur la mailing-list de Git,

— En cas de problème non-résolu et bloquant, poser la question sur le forum du cours sur Arche.