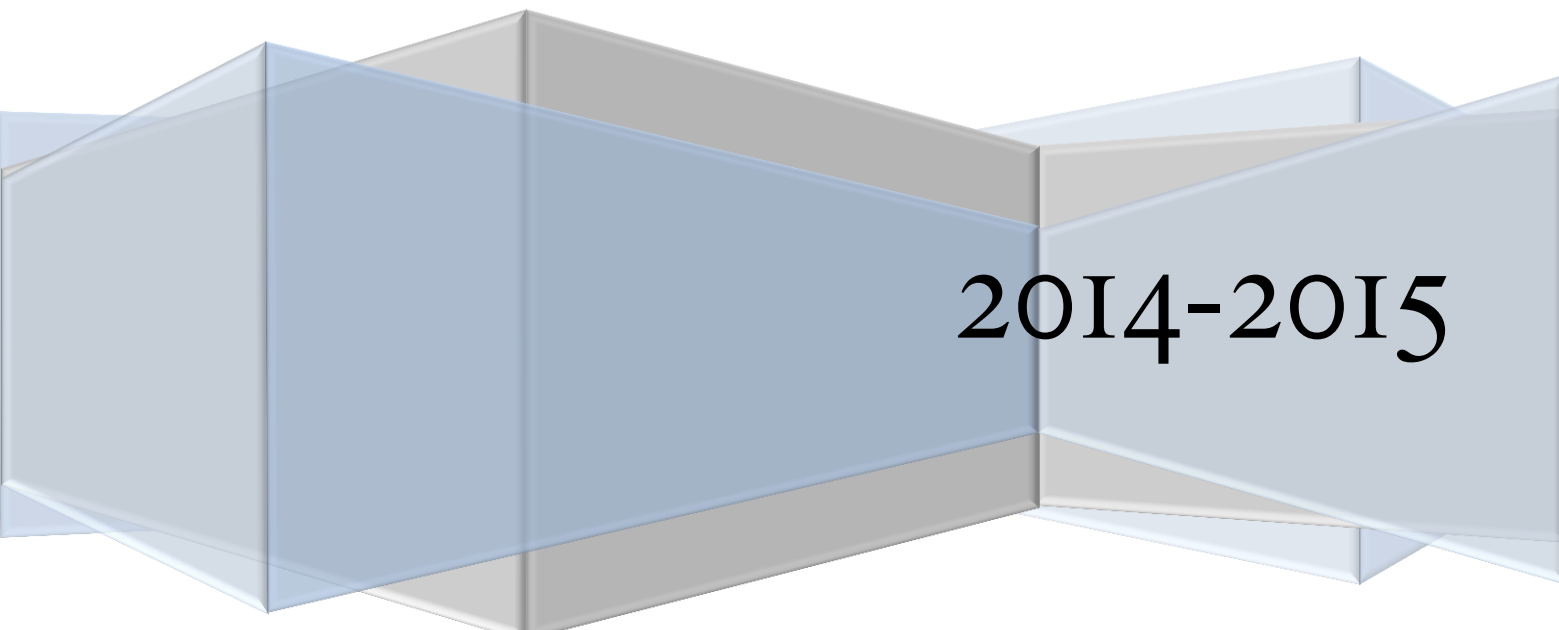


3ème année – département informatique – 2014-2015

Compte rendu

Projet d'algorithmique de 3ème année

Cyril CARRON – Pauline HOULGATTE – Maxime PINEAU



2014-2015

Table des matières

Introduction	1
Modélisation des données	1
Communication avec le serveur	1
Les différentes stratégies développées	2
La classe Stratégie principale	2
Stratégie aléatoire (annexe 1)	2
Stratégie analyse du terrain (annexe 2).....	2
Stratégie prévision (annexe 3)	2
Logiciels et librairies utilisés.....	3
Le partage et versionnage du projet avec Git et c9.io.....	3
Les librairies	3
La documentation avec Sphinx.....	3
Interface graphique.....	4
Les problèmes rencontrés	4
Les finitions et améliorations possibles.....	4
Répartition des tâches.....	5
Conclusion.....	5
Webographie	5
Annexes.....	6
Annexe 1 : Algorithme Stratégie Aléatoire	6
Annexe 2 : Algorithme Stratégie Analyse du terrain.....	6
Annexe 3 : Algorithme Stratégie Prévision	6
Annexe 4 : Installation et utilisation de Sphinx	6
Annexe 5 : Interface Graphique (exemples sur les cartes 6 et 8).....	8
Annexe 6 : Diagramme de Gantt	8

Introduction

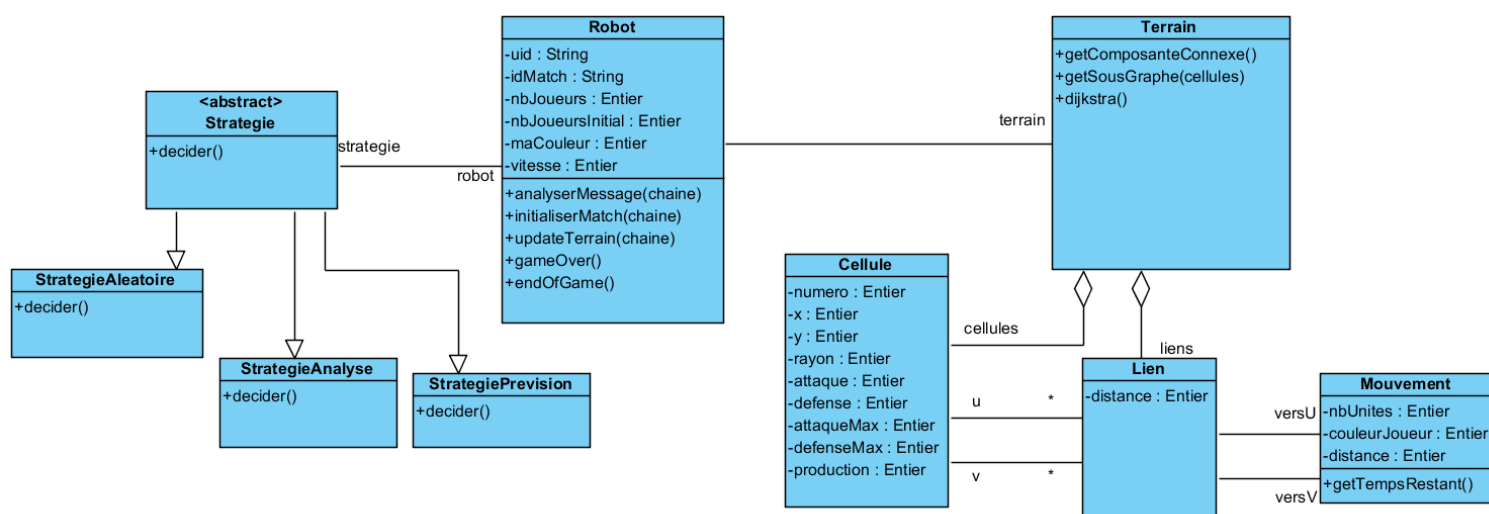
L'objectif de ce projet a été de réaliser une intelligence artificielle basée sur le jeu de Little Star for Little Wars 2. Celle-ci devra être capable d'échanger avec le serveur fourni en fonction du protocole de communication de ce dernier, de prendre une décision, et enfin d'envoyer ses décisions au serveur.

Dans ce rapport, nous vous présenterons la structuration de nos données, la réalisation de la communication avec le serveur. Nous exposerons ensuite les différentes stratégies implémentées pour prendre une décision, ainsi que les logiciels et bibliothèques qui nous ont été utiles. Nous terminerons par l'exposition des difficultés que nous avons rencontrées et les améliorations qu'il est possible de réaliser.

Modélisation des données

Dans le but de faciliter la structuration de nos données, nous avons réalisé en début de projet un diagramme de classe UML.

Nous avons choisi de diviser le terrain en cellule, reliée par des liens contenant des mouvements. Le robot communique avec les terrains en fonction de la stratégie dans laquelle il se trouve.



Communication avec le serveur

Dans l'objectif d'avoir des données les moins périmées possibles, nous avons utilisé trois threads, dont leur rôle est l'un des suivants :

- mettre à jour le temps du jeu, permettant ainsi d'avoir avec précision le temps restant avant l'arrivée à destination des unités sur les liens.
- mettre à jour l'état du terrain (les cellules et les mouvements)

- envoyer les ordres au serveur, tout en s'assurant que ceux-ci ne sont pas incohérents

Ces threads sont lancés dans la fonction `play_pooo()` et s'arrêtent lorsque le match est terminé (excepté pour le thread envoyant les décisions au serveur, qui lui s'arrête lorsque l'IA n'a plus le droit de jouer).

Les différentes stratégies développées

La classe Stratégie principale

Afin de nous laisser le plus de liberté possible au niveau de la création d'une nouvelle stratégie, nous avons décidé de créer une classe abstraite, nommée `Stratégie`. Celle-ci ne comporte qu'une méthode abstraite `decider()`, permettant de prendre une décision en fonction de l'état du terrain.

À chaque fois que l'on souhaitera implémenter une nouvelle stratégie, nous n'aurons qu'à créer une nouvelle classe, et lui faire hériter la classe `Stratégie`.

Comme la méthode `decider()` est une méthode abstraite, elle n'a normalement pas de corps, or les classes abstraites n'existent pas en Python. Pour remédier à ce problème, cette méthode lance une exception, nous obligeant ainsi à la redéfinir à chaque création de nouvelle stratégie.

Stratégie aléatoire (annexe 1)

Cette stratégie a été la première réalisée. C'est une stratégie simple. Chaque cellule envoie un nombre aléatoire d'unités (dans la limite du possible) sur une cellule voisine qui sera sélectionnée au hasard.

Stratégie analyse du terrain (annexe 2)

La principale difficulté a été de choisir où envoyer les unités de nos cellules. Pour cela, nous avons divisé nos cellules en deux groupes, les productrices et les attaquantes.

Les productrices sont les cellules qui n'ont que des cellules alliées comme voisines, leur rôle est de supporter la cellule attaquante la plus proche en envoyant toutes leurs unités vers celle-ci. La cellule attaquante la plus proche est calculée au moyen de l'utilisation de plusieurs algorithmes Dijkstra.

Les attaquantes ont pour rôle de conquérir de nouvelles cellules. Pour choisir la cellule à attaquer, nous avons défini un indice `P`. Celui-ci prend en compte pour chaque cellule ennemie : sa production, son coût (unités offensives et défensives), son nombre de voisins, ainsi que la distance entre la cellule ennemie et la cellule attaquante. On a fait le choix de privilégier l'attaque sur la cellule maximisant cet indice, puis on enverra soit le coût de la cellule auquel on ajoutera un si l'on peut prendre cette cellule, soit l'excédent de la cellule attaquante (le nombre d'unités qu'elle va recevoir ou produire en trop).

Stratégie prévision (annexe 3)

Cette stratégie a pour objectif d'étudier les mouvements de l'adversaire tout en économisant ses troupes. Chaque cellule attaquante possédée se trouvera donc dans deux états différents :

- Soit la cellule attaquante observe que l'une des cellules adjacentes ennemies (neutre compris) va bientôt être prise par un autre adversaire. Elle va donc attendre le bon moment avant d'envoyer ses unités vers la cellule sur le point d'être prise, dans le but de ne subir aucune lourde perte. Elle attend donc que le premier affrontement s'effectue sans y prendre part, puis récupère la planète dès que les conflits sont réglés (par exemple, si une cellule neutre est sur le point d'être capturée par l'ennemie, on le laisse dépenser ses unités sur la cellule neutre)
- Soit ce n'est pas le cas et la cellule attaquante aura le même comportement que dans la stratégie d'analyse.

Les cellules productrices ont quant à elles toujours le même comportement que dans la stratégie d'analyse, elles enverront toutes leurs unités vers la cellule attaquante la plus proche.

Logiciels et librairies utilisés

Le partage et versionnage du projet avec Git et c9.io

Comme conseillé, nous avons utilisé Git afin de partager et gérer les différentes versions de notre code facilement, ainsi que c9.io, pour pouvoir y accéder n'importe où.

c9.io est un service web permettant d'accéder à son code en ligne, ainsi que le développement coopératif en temps réel (plusieurs personnes modifiant le même fichier en même temps).

L'utilisation de ces deux outils a grandement facilité le travail en équipe.

Les librairies

Nous avons utilisé les librairies suivantes :

- `re` : permet d'utiliser les expressions régulières. Nous l'avons utilisé pour récupérer les différentes informations que nous envoyait le serveur dans les chaînes au format du protocole.
- `threading` : permet de créer des processus légers, et est utilisé pour lancer les threads permettant la communication avec le serveur comme expliqué précédemment.
- `unittest` : permet de faire des tests unitaires automatisés. Nous n'avons malheureusement pas eu le temps de l'utiliser totalement par manque de temps.

La documentation avec Sphinx

Afin d'automatiser la documentation, nous avons utilisé le logiciel Sphinx. Sphinx est un générateur de documentation qui s'appuie sur des fichiers au format rest (REStructuredText) qu'il peut convertir en html, PDF, man ou autres.

Il produit la documentation en fonction de commentaires écrits dans un formalisme qui lui est propre et qu'il reconnaît. Vous trouverez en annexe (annexe Z) la méthode pour installer et utiliser le logiciel Sphinx.

Nous avons choisi d'utiliser sphinx il s'agit du générateur de documentation le plus complet et le plus utilisé, contrairement à d'autres générateurs de documentation comme pydoc et epydoc. La documentation de python est générée à l'aide de Sphinx par exemple.

Interface graphique

Une interface graphique est en cours de réalisation. Celle-ci utilise le module tkinter pour dessiner les cellules, les liens et les unités en déplacement (mouvements), ainsi que pour les mettre à jour. Il faut noter que cette interface a des erreurs de comportement (elle ne veut plus utiliser une fonction du module tkinter, et bloque alors le programme) lors du troisième match. Un exemple de cette interface est disponible en annexe (annexe 5).

Les problèmes rencontrés

Nous allons maintenant aborder les différents problèmes que nous avons rencontrés, et leurs solutions lorsque cela a été possible.

Le principal problème a été l'organisation du travail en équipe. Pour résoudre celui-ci, nous avons décidé de nous réunir au moins une fois par semaine (généralement le jeudi). Lors de ces réunions nous évoquions l'avancement de notre travail ainsi que les difficultés rencontrées. Nous discutons ensuite de ces derniers s'ils n'avaient pas encore été réglés. Nous procédions ensuite à une mise à jour générale via l'utilisation d'un dépôt Git, ceci pour faciliter le partage de chacun.

Nous avons ensuite rencontré un second problème. Celui-ci fut l'abandon d'un des membres de notre groupe durant le développement du projet. Celui-ci a eu pour conséquences un réaménagement des tâches et un retard conséquent sur nos prévisions qui a conduit à l'abandon de la phase des tests unitaires.

Le dernier problème a concerné l'utilisation de Git, et principalement la résolution des conflits entre les différentes versions du code existant. Nous avons dû passer un certain temps dans la lecture de la documentation de Git pour résoudre ce problème.

Les finitions et améliorations possibles

La première tâche à terminer est celle des tests unitaires, c'est à dire vérifier si toutes nos fonctions font bien ce qu'elles devraient faire et ce au moyen de tests automatisés.

On pourrait ensuite finaliser l'interface graphique, tout d'abord en résolvant les problèmes liés à la fermeture des fenêtres tkinter, mais aussi en utilisant l'option -g de la commande poooserver.py (permettant ainsi de distinguer le fonctionnement du bot et de l'interface).

Et enfin, on pourrait optimiser la prise de décision de notre intelligence artificielle. Par exemple, en modifiant le comportement de celle-ci et changer dynamiquement de manière à prendre des décisions en utilisant un patron de conception Stratégie. Ou encore en prenant en compte plus de situations particulières (cas des différents types de cellules attaquantes).

Répartition des tâches

Au début du projet, nous nous sommes réunis pour discuter, partager nos idées pour le projet, réaliser le diagramme UML et ainsi définir comment structurer nos données. Nous nous sommes ensuite répartis les tâches en essayant d'équilibrer le travail de chacun et de satisfaire les envies de tous. Cette répartition peut être vue sur le diagramme de Gantt fourni en annexe (annexe 6).

Conclusion

Pour conclure, ce projet nous a permis de mettre en œuvre les compétences que l'on a pu acquérir au cours de ce premier semestre, notamment avec l'utilisation des graphes, ou encore le développement en Python, et enfin avec les différents problèmes de concurrences possibles entre threads. Il a été très intéressant à développer, et particulièrement pour la partie traitant sur l'intelligence artificielle.

De plus, il nous a permis de travailler en toute liberté, avec très peu de contraintes, et également d'apprendre à utiliser de nouveaux logiciels comme Sphinx et Git.

Et enfin, ce projet nous a donné une première approche du développement d'un logiciel en équipe, des problèmes qui y sont liés (la répartition des tâches et l'organisation) et leurs résolutions (comme par exemple le partage du code source avec Git).

Webographie

Guide d'utilisation de Git : <https://www.atlassian.com/git/tutorials/>

Documentation de Python : <https://docs.python.org/3/>

Documentation de Git : <http://git-scm.com/documentation>

Sphinx : <http://sphinx-doc.org/>

Annexes

Annexe 1 : Algorithme Stratégie Aléatoire

```
pour chaque planète:
    envoyer un nombre aléatoire d'unités
    sélectionner une planète voisine
    envoyer les unités sur la planète
```

Annexe 2 : Algorithme Stratégie Analyse du terrain

```
pour chacune de mes cellules attaquantes :
    pour chaque cellule adjacente ennemie :
        je calcule son indice P
    je sélectionne la cellule ayant l'indice P le plus grand ( = cellule ciblée )
    je calcule le cout de la cellule ciblée
        je calcule mon excédent ( = trop plein de cellule prochain )
    si mon excédent est supérieur au cout de la cellule
        j'envoie mon excédent vers la cellule ciblée
    sinon
        j'envoie le cout de la cellule ciblée + 1 vers la cellule ciblée

pour chacune de mes cellules productrices :
    je calcule le plus court chemin vers chaque cellule attaquante avec Dijkstra
    je sélectionne le chemin minimisant la distance que Dijkstra retourne
    j'envoie mes unités vers la deuxième cellule du chemin (la 1ere nous appartenant)
```

Annexe 3 : Algorithme Stratégie Prévision

```
pour chaque cellule attaquante:
    on vérifie si une planète voisine est sur le point de se faire capturer
    si c'est le cas:
        on récupère le cout et la distance de la planète la plus proche
        on vérifie que le temps de déplacement de nos unités est supérieur à celui de l'adversaire
        si c'est le cas:
            on envoie nos troupes
        sinon:
            on attend
    fin si
    sinon:
        on entre en stratégie d'analyse
    fin si
fin pour
```

Annexe 4 : Installation et utilisation de Sphinx

Commandes pour installer Sphinx :

- sous windows : Une fois que python est installé, lancer avec l'invité de commandes 'easy_install sphinx'
- sous linux : dans un terminal, 'apt-get install python-sphinx'

Générer automatiquement la documentation :

- créer un dossier dans lequel on se placera pour créer le projet sphinx.

- ouvrir un terminal et se placer dans ce dossier, taper la commande ‘sphinx-quickstart’. Celle-ci permet de paramétrer la création du projet avec de nombreuses options.
- se placer ensuite à la racine du projet, ce dossier contient normalement les dossiers source et build.
- Avant de lancer la génération de la documentation, il faut spécifier l’emplacement du code du projet. Il faut donc modifier le fichier conf.py présent dans le dossier source et modifier la ligne `sys.path.insert(0, os.path.abspath(‘emplacement du code’))`
- Afin de pouvoir spécifier quels fichiers intégrer dans la documentation, il faut éditer le fichier `index.rst` et spécifier manuellement les modules ou packages à prendre en compte. Par exemple dans notre cas, pour intégrer le module Graphique, il faut écrire ces lignes dans le fichier :

```
Graphique
-----
.. automodule:: Graphique
   :members:
```

Désormais, Sphinx sait qu’il doit intégrer le fichier `Graphique.py` pour générer la documentation.

Pour que Sphinx reconnaisse les éléments à documenter, il faut respecter une syntaxe particulière :

```
def fonction( ... ):
    """
    description générale de la fonction

    :param nom_du_paramètre : description du paramètre
    :type nom_du_paramètre : type_du_parametre
    :rtype : type du retour de la fonction
    """
```

Pour documenter les paramètres d’une classe, il faut placer les commentaires avant le constructeur.
Exemple :

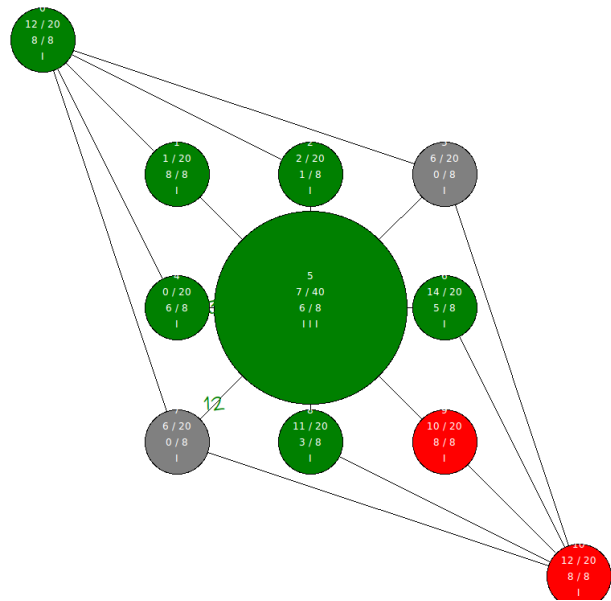
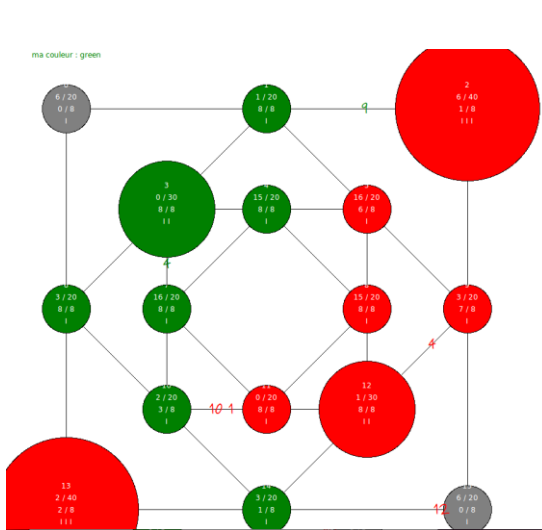
```
class Exemple
    """
    Description de la classe

    :param xxx:
    etc...
    """
```

Tous les éléments sont facultatifs, cependant il est préférable de les mettre pour chaque fonction car cela augmente la lisibilité une fois la documentation générée.

Lancer la commande ‘make html’ (pour générer la documentation au format html). La documentation est générée dans le dossier `build/html/index.html`

Annexe 5 : Interface Graphique (exemples sur les cartes 6 et 8)



Annexe 6 : Diagramme de Gantt

Planificateur de projet

ACTIVITÉ	Semaine Débutée (en semaine)		Semaine Débutée (en semaine)		POURCENTAGE	PARTAGE DES TÂCHES	PRÉCISÉ
	Prévu	Prévu	Réel	Réel			
Compréhension protocole	1	1	1	1	100%	Equipe	
Compréhension des objectifs	1	1	1	1	100%	Equipe	
Prise en main du jeu	1	1	1	1	100%	Equipe	
Mise en place de l'environnement de dével	1	1	1	1	100%	Equipe	
Modélisation UML (diagrammes de classes)	1	1	1	1	100%	Maxime & Pauline	
Réalisation d'un diagramme de Gantt	1	1	1	1	100%	Pauline	
Implémentation en python	1	2	1	2	100%	Equipe	
Ecriture de la fonction register	1	1	1	1	100%	Maxime & Cyril	
Ecriture de la fonction init	1	1	1	1	100%	Maxime & Cyril	
Ecriture de la fonction play	1	2	1	2	100%	Maxime & Cyril	
Implémentation de l'algorithme de Dijkstra	2	1	2	1	100%	Pauline	
Idées de stratégies	1	3	1	3	100%	Equipe	
Stratégie Random	2	1	2	1	100%	Maxime & Pauline	
Stratégie principale	2	1	2	1	100%	Pauline & Maxime	
Stratégie prévision	2	1	2	1	100%	Pauline & Maxime	
Rédaction de pseudo code	2	2	2	2	100%	Equipe	
Implémentation des différentes stratégies	3	3	3	4	100%	Equipe	
Stratégie Random	3	1	3	1	100%	Maxime & Cyril	
Stratégie principale	3	1	3	1	100%	Maxime	
Stratégie prévision	3	1	3	1	100%	Pauline	
Réalisation de l'interface graphique	6	1	6	1	80%	Maxime	
Réalisation des tests unitaires	6	1	6	1	0%	Equipe	
Réalisation des tests d'intégrations	6	1	6	1	0%	Equipe	
Génération de la documentation du prograi	6	1	6	1	100%	Cyril	
Rédaction du readme	7	1	7	1	100%	Pauline	
Rédaction rapport	7	1	7	1	100%	Equipe	
Préparation soutenance	7	1	7	1	100%	Equipe	
Rendu du projet	8	1	8	1	100%	Equipe	
Soutenance	9	1	9	1	100%	Equipe	