Projet informatique: Honshu (Lot A)

Romain PEREIRA
Douha OURIMI
Afizullah RAHMANY
Guangyue CHEN

02/03/2018

Sommaire

1 Introduction **2**

2 Rapport lot A **3**
2.1 Organisation du projet, les dates clefs 3
2.1.1 03/03/18 -> 10/03/18 : préparation du dépot 3
2.1.2 10/03/18 -> 16/03/18 : 1ère implémentation du jeu 3
2.1.3 19/03/18 -> 25/03/18 : 2ème implémentation du jeu en équipe 4
2.2 Détails techniques de l'implémentation 5
2.2.1 Structure de données 5
2.2.2 Algorithme / complexité temporelle 6

3 Références **9**

Préambule

Ce projet est réalisé dans le cadre de nos études à l'ENSIIE. L'objectif est de prendre en main des outils de 'programmation agile', en developpant un jeu de carte: le Honshu.

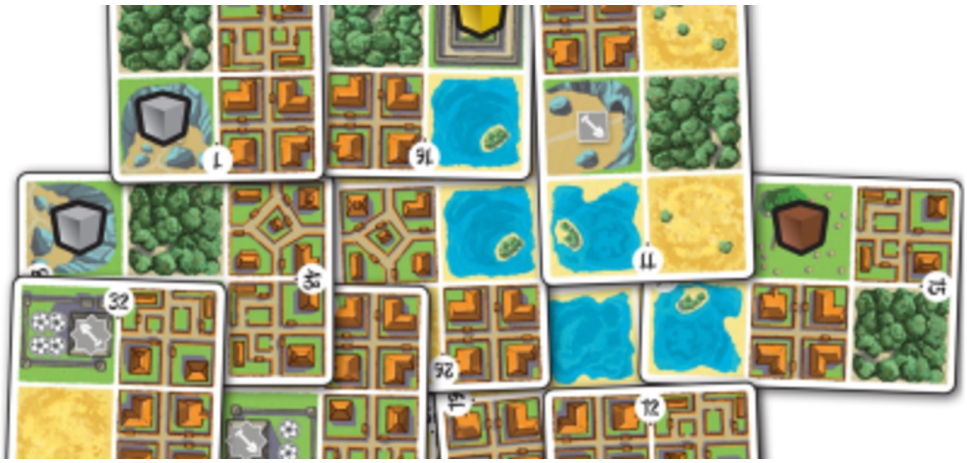

Figure 1: *Plateau de jeu*

1 Introduction

Ce document rapporte le travail effectué entre le 2 Mars 2018, et le 25 Mars 2018. Cette 1ère partie semble être la plus importante du projet: nous devons prendre en main les différents outils, structurer le projet, puis finalement, concevoir les structures de données et la logique global du programme. Une mauvaise conception à ce niveau risque de nous pénaliser sur le reste du projet, c'est pourquoi tous les choix techniques seront amplement justifiés au regard des futurs lots à rendre.

2 Rapport lot A

2.1 Organisation du projet, les dates clefs

2.1.1 03/03/18 -> 10/03/18 : préparation du dépot

03/03/18 : préparation de l'espace de travail. - Creation des comptes et de l'équipe Gitlab/Trello.
- Création du dépot Git, et organisation du projet:

- "src" : dossier contenant les sources '.c' du projet
- "includes" : dossier contenant les sources '.h' du projet
- "tests" : dossier contenant les fichiers tests ('.c' et '.h')
- "bin" : dossier contenant les binaires (executables et bibliothèques)
- "obj" : dossier temporaire contenant les fichiers objets (compilés)
- "doc" : dossier contenant la documentation generé via Doxygen
- "Makefile" : pour compiler le projet
- ".gitignore" : .gitignore généré via "https://www.gitignore.io/" (afin de ne pas envoyer de fichier compiler et de fichier Latex)
- "README.md" : un README indiquant les procédures d'installation du projet/règles du Makefile
- "rapport" : dossier contenant des prises de notes pour les rapports.

Toutes les procédures (compilation, génération de la documentation, tests) ont été automatisées dans le Makefile afin de gagner du temps lors du développement.

07/03/18 : Brainstorming. Nous nous sommes réunis ce Mercredi matin entre 9h et 11h.

- Installation des outils sur les ordinateurs de Douha et Guangyue
- Lecture et demystification du sujet ensemble
- Apprentissage des règles du jeu. (nottement à l'aide de la vidéo: [1]
- Réflexion autour de l'implémentation.

08/03/18 : automatisation de l'installation des dépendances. Ajout d'un script shell ("configure.sh") s'occupant d'installer les dépendances sous Linux/MacOSX.

10/03/18 : début de la rédaction du rapport. Début de la rédaction au propre de ce rapport. Les rapports seront rédigés sous Latex (un template est fourni dans le dépot: "rapport/template.tex".

2.1.2 10/03/18 -> 16/03/18 : 1ère implémentation du jeu

Le brainstorming (2.1.1) a posé les bases de l'implémentation techniques.
Pendant cette période, l'idée était donc d'avoir un prototype qui tourne, avec des fonctions 'à trous', que chacun remplirait en se séparant le travail.

Romain : Cependant, voulant avoir une 1ère version du jeu rapidement, j'ai fait du zèle et j'ai programmé presque l'intégralité du jeu seul.

Guangyue : Cependant, j'ai programmé les codes des fichiers dans le dossier : "tests" .

16/03/18 : entretien avec notre encadrant Pendant l'entretien de la séance encadré, il est ressorti que cette méthode de fonctionnement n'est pas bonne: la quantité de travail n'est pas bien repartie entre les membres du groupe, et tous les membres ne se sentent pas profondément intégrés au projet. Il est important que chaque membre soit intégré et investi dans le projet, afin d'être motivé pour le travail en équipe. C'est pourquoi, pendant ce week-end, nous reflechissons à une restructuration du projet qui se suivra par une réunion Lundi soir.(2.1.3)

2.1.3 19/03/18 -> 25/03/18 : 2ème implémentation du jeu en équipe

19/03/18 : restructuration du projet Après le premier entretien, nous avons alors décidé de nous réunir le lundi 18/03 afin de faire un point sur l'avancement du projet et l'organisation du travail. Nous avons remis en question la présentation du Trello mais au final elle nous a paru efficace . En effet cette présentation permet de voir exactement ce qui doit être fait, ce qui est en train d'être fait et ce qui est terminé et par qui cela a été fait. Nous sommes alors passé au projet et nous nous sommes de nouveau concertés sur les structures, vérifié que tout le monde les a bien comprises et surtout nous avons décidé de commenter en français et non en anglais comme au départ. D'une certaine manière tous les fichiers .c et .h on du être repris à 0 par chacun de nous.

Guangyue : Pendant le réunion, nous discutons l'architecture ensemble . Après la Révision de l'architecture du code , j'ai corrigé les codes des "tests" , et j'ai programmé les fonctions dans les fichiers qui sont dans le dossier : "src" .

25/03/18 : vérification des tests unitaires : dernier jour avant le rendu, on se dépêche d'intégrer le code de tout le monde, et que ca compile sans erreurs. On commence à debogger à l'aide de valgrind.

2.2 Détails techniques de l'implémentation

2.2.1 Structure de données

On représente une **case** par:

- type : ville, lac, plaine ...
- une **case** au dessus sur la grille (pouvant être nulle)
- une **case** en dessous sur la grille (pouvant être nulle)
- la **tuile** a laquelle cette **case** appartient

On représente une **tuile** par:

- 6 **cases**
- une rotation
- 1 position sur la carte (la **case** en haut à gauche de la **tuile** est l'origine)

On représente une **grille** par:

- un entier 'n'
- un tableau $n * n$ de **cases** (pouvant être nulle)

On représente une **partie** par:

- une **grille**
- un tableau de **tuiles**

Ces structures de données nous permettent de représenter une partie de Honshu efficacement en mémoire, et facilite l'implémentation des algorithmes. Pour une **partie**, avec une **grille** de taille n et m tuiles, la complexité spatiale est de l'ordre de:

$$8n^2 + 6 * 32 * m + o(n^2) + o(m)$$

- $8n^2$: espace mémoire de la grille (tableau de n^2 pointeurs)
- $6 * 32 * m$: espace mémoire des tuiles, une tuile contenant 6 cases, et 1 case faisant environ 32 octets.

2.2.2 Algorithme / complexité temporelle

Grâce à ces structures de données, nous pouvons, a n'importe quel instant, accéder à une case (et donc à sa tuile associé) en $O(1)$.

Tests d'insertion d'une tuile : permet de savoir si une tuile peut être insérée à une coordonnée donnée sur la grille (en respectant les règles d'insertions) Cet algorithme a une complexité en $O(1)$.

Algorithm 1: Tests d'insertion d'une tuile

Require: une *grille* ; une *tuile* ; $0 \leq x, y < n$
Ensure: Renvoie VRAI ou FAUX, si *tuile* peut être ajoutée à *grille* en *(x, y)*

```
compteur ← 0
for Chaque case, (dx, dy) de tuile do
    dessous ← grille.cases(x + dx, y + dy)
    if dessous != NULL then
        compteur = compteur + 1
        if dessous.type == LAC then
            Renvoyer FAUX (tentative d'insertion sur un lac)
        end if
    end if
    if compteur == 0 then
        Renvoyer FAUX (aucunes cases en dessous de la tuile a inséré)
    end if
end for
for Chaque 'tuile' dessous sous tuile do
    x ← Compter le nombre de cases recouvertes de dessous
    y ← Compter le nombre de cases que tuile recouvrirait sur dessous
    if x + y == 6 then
        Renvoyer FAUX (la tuile dessous serait entierement recouverte
    end if
end for
Renvoyer VRAI (tous les tests sont passés)
```

Insertion d'une tuile : aucun test d'insertion n'est à proprement effectué ici. Un simple jeu de pointeurs relie les cases entre elles afin d'assurer l'intégrité de nos structures de données. Cet algorithme a une complexité en $O(1)$.

Algorithm 2: Ajout d'une tuile

Require: une *grille* ; une *tuile* ; $0 \leq x, y < n$
Ensure: Ajoutes *tuile* à *grille*
  for Chaque *case, (dx, dy)* de *tuile* **do**
    dessous ← *grille*.cases(x + *dx*, y + *dy*)
    case.au_dessus ← NULL
    case.au_dessous ← *dessous*
    if *dessous* != NULL **then**
      dessous.au_dessus ← *case*
    end if
    grille.cases(x + dx, y + dy) ← *case*
  end for

Recuperer le village associé à une ville : Cet algorithme se programme sur le modèle d'un parcours en largeur. [2] Il a une complexité en $O(T)$, où T est la taille du village associé à la case.

Algorithm 3: Taille d'un village associé à une case : T(x, y)

Require: une *grille* ; $0 \leq x, y < n$
Ensure: $T(x, y)$
  $T \leftarrow 0$
  if *grille*.cases(x, y).type != VILLE **then**
    Renvoyer $T = 0$
  end if
  Soit *file*, une file.
  file.ajout(x, y)
  Marquer *grille*.cases(x, y) comme visité
  while *file*.nonVide() **do**
    $T = T + 1$
    $(xi, yi) \leftarrow$ *file*.pop()
    case ← *grille*.cases(xi, yi)
    Marquer *case* comme 'visité'
    for Chaque *voisin* de *case* **do**
      if *voisin*.type == VILLE *et* *voisin*.nonVisite() **then**
        file.ajout(*voisin*.x, *voisin*.y)
        Marquer *voisin* comme visité
      end if
    end for
  end while
  Renvoyer T

Supprimer une tuile de la grille : fonction recursive qui s'applique sur les tuiles au dessus de celle qu'on supprime, afin d'assurer l'intégrité de la grille.

Algorithm 4: Supprime une tuile de la grille (et les tuiles qui en dépendent)

Require: une *grille* ; une *tuile* ; $0 \leq x, y < n$
Ensure: Supprimes la *tuile* de la *grille*
  Supprimer les cases de *tuile* de la grille.
  for Chaque 'tuile' *dessus* au dessus *tuile* **do**
    if *dessus* ne respecte plus les règles d'insertion **then**
      Supprimer *dessus* de la grille (récursivement)
    end if
  end for
  Renvoyer T

tests tous les autres fonctions qui sont déjà écrits : CUnit est une combinaison d'un framework indépendant de la plate-forme avec diverses interfaces utilisateur. Le framework de base fournit un support de base pour gérer un registre de test, des suites et des cas de test. Les interfaces utilisateur facilitent l'interaction avec le framework pour exécuter des tests et afficher les résultats. Tout d'abord, opérez sur chaque structure. Ensuite, testez chaque fonction dans chaque fichier "*.test.c" . Parce que c'est un test en boîte blanche, pour chaque fonction , on conçoit la situation spécifique qu'il accomplissait, et il conçoit tous les résultats possibles et les compare au résultat courant. Finalement, nous avons constaté que tous les tests ont réussi et que toutes les fonctions écrites ont bien marché.

3 Références

[1] Vidéorègle jeu de société "Honshu",
http://videoregles.net/, 6 Sept. 2017,
https://www.youtube.com/watch?v=WHD6B_NCd-4.

[2] Algorithme de parcours en largeur,
Wikipédia, 11 octobre 2017 à 16:40.,
https://fr.wikipedia.org/wiki/Algorithme_de_parcours_en_largeur.