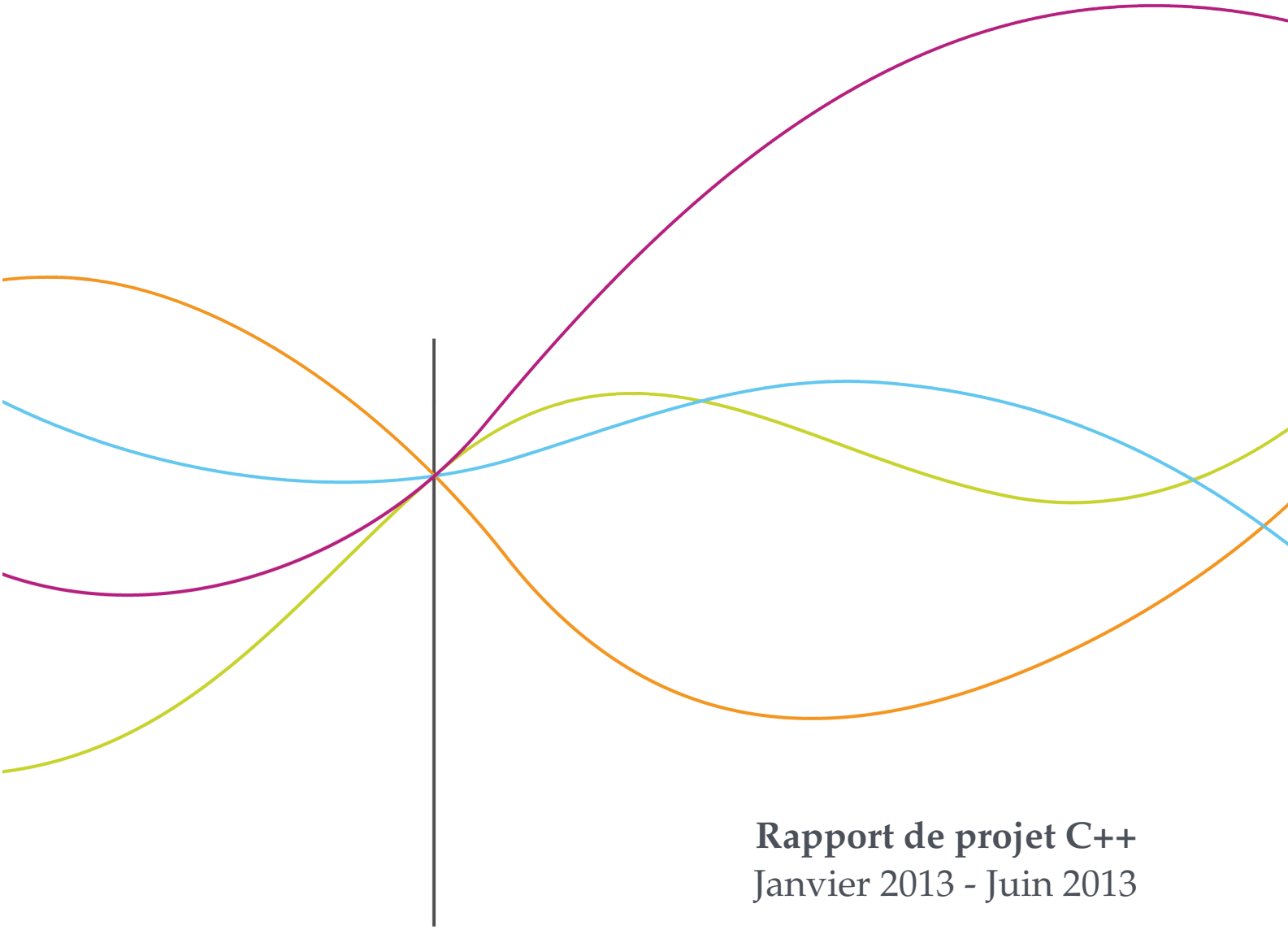
INSA ROUEN
À taille humaine,
à l'échelle du monde

Département Génie Mathématique

Affectation spatiale sous contraintes



Rapport de projet C++

Janvier 2013 - Juin 2013

Adeline Bailly, Gabrielle Diaferia,
Pauline Hubert & Alexandre Quemy

Table des matières

Introduction

L'affectation est un problème d'optmisation combinatoire qui consiste, dans sa version simple, à affecter un certain nombre de ressources disponibles à un nombres de tâches dans l'objectif d'optimiser une fonction objectif. Il peut s'agir de minimiser des coûts ou maximiser les bénéfices. Ce problème peut être résolu en temps polynomial par la méthode hongroise également appelée algorithme de Kunh.
On peut étendre ce problème par des objectifs multiples, des contraintes changeantes en temps réel ou encore des contraintes probabilites traduisant une observation ou une connaissance partielle de l'environnement. On peut alors parler plus largement de plannification, qui est un domaine ouvert de l'intelligence artificielle où de nombreuses formulations font apparaître des problèmes de la classe NP-difficile ou NP-complet sur lesquels de nombreuses équipes de recherches travaillent.

Dans le cadre de notre projet C++ nous avons choisi de réaliser un framework permettant la modélisation et l'implémentation de problème de plannification spaciale sous contrainte. Il s'agira de proposer, comme nous le verrons en détail par la suite, des structures de données de partitionnement spacial permettant la réduction de la complexité d'heuristiques par le principe de Divide and Conquer, divers algorithmes d'affectation selon plusieurs approches (ressources vers objectifs, objectifs vers ressources), et également divers moteurs de modélisation de contraintes et objectifs pour la plannification. Comme notre plannification concerne de l'affectation spatiale, des algorithmes de plus court chemins seront également proposés.
Enfin, pour illustrer la résolution temps réel des problèmes, en plus du framework, une application de simulation d'une Intelligence Artificielle sera développée. Il s'agira d'une application graphique montrant des unités d'un jeu vidéo affectées à divers postes pour maximiser divers objectifs pouvant changer au court du temps (équilibrage de ressources, objectif de construction, par exemple).

Le projet étant ambitieux à la vue de la multitude des algorithmes existants tant pour le plus court chemin que pour l'indexation spatiale voire pour la description des contraintes et objectifs, le travail réalisé sera axé sur la modélisation et l'architecture du framework. L'implémentation sera partielle mais les bonnes pratiques de développement occuperont une part importante de celle-ci dans le but de valoriser le livrable et permettre une implémentation complète en dehors du cadre de ce projet.

Ce rapport se divise en plusieurs chapitre. Le premier présentera dans un premier temps la méthodologie adoptée et les pratiques de développment. Le second chapitre traitera de

la modélisation. Dans un premier temps, des concepts généraux seront exposés, puis nous exposerons l'analyse des besoins, l'identification des briques logicielles et l'architecture globale du framework. Enfin, une présentation détaillée des divers composants sera effectuée : partitionnement et indexation spatial, algorithmes de plus court chemin, algorithmes d'affectation, moteur de contraintes et plannification.
Un troisième chapitre sera dédié à l'implémentation et un quatrième présentera l'application de démonstration : technologies, modélisation et implémentation.

Chapitre 1

Méthodologie

1.1 Management du code

1.1.1 Versionnage

Git

1.1.2 Construction logicielle

CMake

Résolution de la dépendance

Module CMake

1.1.3 Tests unitaires

CMake via Ctest

1.1.4 Tests d'intégration

CMake via Ctest Scénario -¿ Test unitaire en BN

1.1.5 Tests de couverture

Lcov

1.1.6 Documentation

Doxygen + Wiki

1.1.7 Repporting

1.1.8 Intégration continue

Jenkins

1.2 Développement de framework

Cette section présente quelques techniques de programmation, idiomes et pattern qui seront utilisés de manière globale pour l'ensemble de la modélisation et de la réalisation du projet.

1.2.1 Idiome NVI

Présentation

L'idiome NVI, pour Non-Virtual Interface, est la réalisation en C++ du design pattern Template Method. Il possède plusieurs avantages notamment dans le développement d'un framework.
Il consiste à maintenir la consistance de certains comportements en un point de contrôle du code défini par le développeur. Cela permet d'une inversion de contrôle bénéfique dans de nombreux cas, dont le cas d'un framework puisqu'il déresponsabilise l'utilisateur final de certaines contraintes qui ne pourraient être exprimées que par de la documentation et dont le non respect entrainerait des comportements inattendus difficiles à localiser.
NVI est guidé par 4 règles décrites par Herb Sutter dans son article sur la virtualité :

- Prefer to make interfaces nonvirtual, using Template Method design pattern.
- Prefer to make virtual functions private.
- Only if derived classes need to invoke the base implementation of a virtual function, make the virtual function protected.
- A base class destructor should be either public and virtual, or protected and nonvirtual.

Exemple

Imaginons une bibliothèque de manipulation de matrices. La bibliothèque propose une classe abstraite Matrice avec l'API globale de toute matrice. On considérera ici la calcul du déterminant. Découle de cette classe une hiérarchie de matrices pour les cas particuliers : matrice triangulaire, matrice diagonale, etc.

```
1  class Matrice
   {
   public :
4      double det() const
       {
           // Pre-traitement : verification d'invariants, lock multithread,
              etc.
7          m.lock();
           assert(data_.check_invariants() == true);
           // Appel de l'implementation
10         return _det();
           // Post-traitement
           assert(data_.check_invariants() == true);
13         m.unlock();
       }

16 private :
```

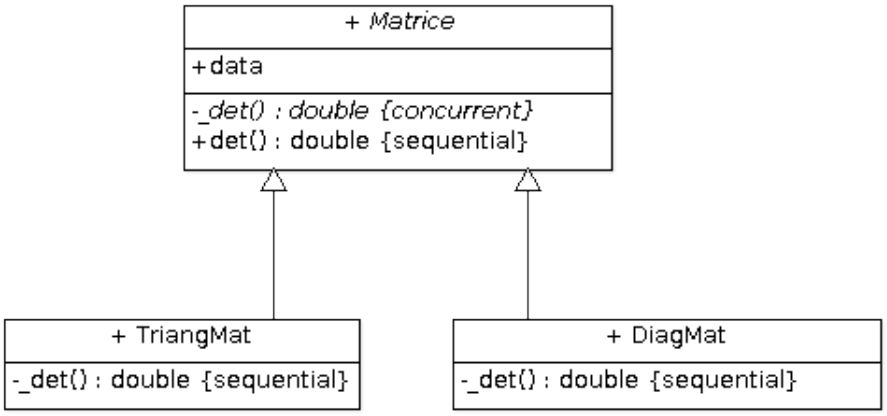


FIGURE 1.1 – Illustration du NVI.

```
    virtual double _det() const = 0;

protected :
    Data data;
    mutex m;
};

class TriangMat : public Matrice
{
private :
    virtual double _det() const
    {
        // Retourne le produit de la diagonale
    }
}

class DiagMat : public Matrice
{
private :
    virtual double _det() const
    {
        // Retourne le produit de la diagonale
    }
}
```

Listing 1.1– NVI : principe.

Ainsi, lorsque l'utilisateur voudra ajouter une nouvelle matrice, disons de forme Hessenberg, il aura simplement à implémenter le calcul du déterminant, les invariants étant toujours vérifiés dans le code de la classe de base qui va appeler l'implémentation.
Il y a donc une réelle inversion de contrôle et c'est le développeur du framework qui dirige le client à l'utiliser correctement.

Nous utiliserons par la suite du projet l'idiome NVI de manière intensive pour permettre de guider le flux d'exécution.

1.2.2 Paramétrage par politique

Présentation

Le paramétrage par politique est une technique de programmation développée et démocratisée par Andrei Alexandrescu dans son livre Modern C++ Design : Generic Programming and Design Patterns Applied et dans la bibliothèque Loki dédiée à la méta-programmation en C++ dont beaucoup d'éléments ont été repris dans Boost puis dans le standard 2011.

Concrètement, il s'agit de profiter de l'héritage multiple et de la métaprogrammation pour permettre de séparer les différents comportements d'une classe ou de plusieurs classes et de créer sur mesure des comportements en combinant plusieurs politiques.

Fonctionnement

La clef d'un paramétrage par politique efficace réside dans l'analyse des différents comportements d'une classe ou d'un ensemble de classes. Imaginons que nous ayons à créer une bibliothèque de gestion de graphes. Un graphe peut être représenté sous différentes formes : matrice d'adjacence, matrice d'incidence , ou liste de successeurs. Chaque représentation à ses avantages et ses inconvénients en fonction des applications. On pourrait aisément créer 3 classes différentes mais cela ne serait pas très pertinent. On pourrait simplement templater la classe de graphe mais chaque type donnée n'a pas la même API. De plus, imaginons qu'un graphe puisse être partagé entre plusieurs thread ou non selon les applications.
Sans paramétrage par politique, il faudrait un nombre de classes égales au nombre de facteurs multiplié par le nombre de modes par facteur. Cela apporterait évidemment du code redondant et une maintenabilité moindre puisque dans le cas d'un paramétrage par politique on peut isoler complètement un comportement. Pour modifier l'intégralité du modèle multithread d'une application, la modification d'une seule classe de politique est nécessaire.

Chaque facteur est représenté par une classe abstraite permettant de définir l'API commune à tous les modes et qui pourra être utilisée par les classes paramétrées avec ce facteur. Les classes concrètes implémentent chacun des modes de la politique.
Enfin, la classe de services qui doit être paramétrée par politique va être templaté avec chacune des politique et en hériter de manière publique ou privée selon les besoins.

Exemple

On définit la première hiérarchie de classes correspondant à la première politique :

```
class ThreadingModel
{
protected :
    virtual void Lock() = 0;
};
```

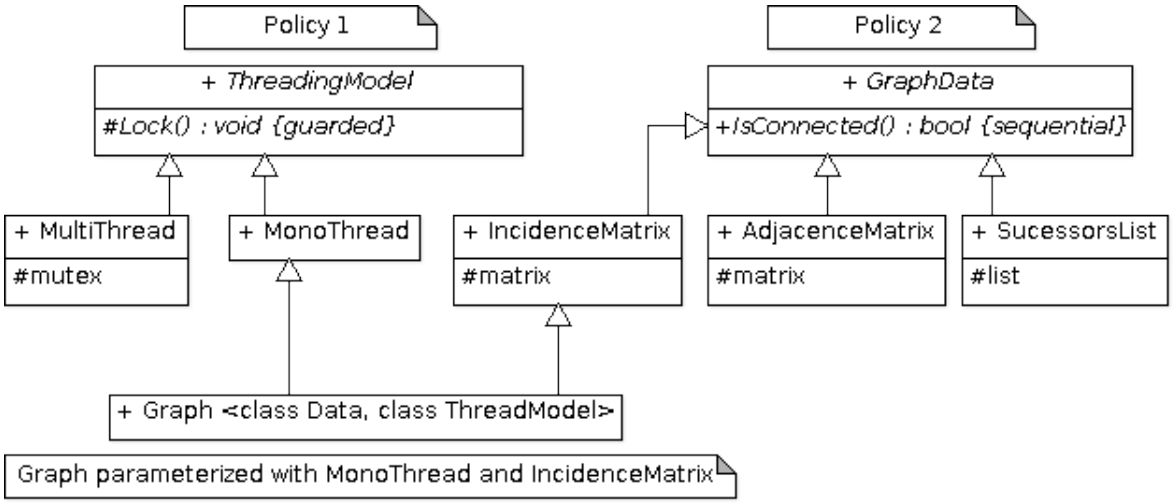


FIGURE 1.2 – Illustration du paramétrage par politique.

```
class MultiThread : public ThreadingModel
{
protected :
    virtual void Lock()
    {
        m.lock();
    }

    std::mutex m;
};

class MonoThread : public ThreadingModel
{
protected :
    virtual void Lock() = default
};

// ... Autres modeles ...
```

Listing 1.2– Définition de la première politique

On définit la seconde politique :

```
class IncidenceMatrix : public GraphRepresentation
{
public :
    virtual bool IsConnected() const
    {
        // Determine si un graphe est connexe
    }
protected :
    std::vector<std::vector<int>> data;
};
```

```
class AdjacenceMatrix : public GraphRepresentation
{
public :
    virtual bool IsConnected() const
    {
        // Determine si un graphe est connexe
    }
protected :
    std::vector<std::vector<bool>> data;
};

//... Autres representations ...
```

Listing 1.3– Définition de la seconde politique

On définit la classe principale de graphe et on la paramétrise à l'instanciation selon les besoins :

```
template <class Rep = AdjacenceMatrix, class ThreadModel = MonoThread>
class Graph : public Rep, public ThreadModel
{
public :
    bool IsConnected() const
    {
        ThreadModel::Lock();
        return Rep::IsConnected();
    }
};

using GraphInciMT = Graph<IncidenceMatrix, MultiThread>;
using GraphAdjMT = Graph<AdjacenceMatrix, MultiThread>;

// Exemples
Graph a; // Adjacence, MonoThread
GraphInciMT b; // Incidence, MultiThread
GraphAdjMT c; // Adjacence, MultiThread
```

Listing 1.4– Illustration de la paramétrisation

La classe Graph fait appelle à l'aveugle à sa politique de thread ainsi qu'à sa politique de représentation. La bonne écriture d'une politique est guidée par l'API de la classe abstraite en haut de la hiérarchie mais aucune vérification de type n'est effectuée par la classe Graph. Ainsi, un utilisateur pourrait écrire sa propre politique qui ne serait pas basée sur une des classes abstraites.
La vérification s'effectue à la compilation en regardant les parties de l'API utilisée (ici la méthode Lock et IsConnected) ce qui est moins restrictif que le typage. On voit clairement l'avantage en terme de maintenabilité puisque la politique ThreadingModel sera surement partagée par l'ensemble des classes de l'application et ainsi, si le modèle doit évoluer, l'ensemble de la gestion multithread se trouve dans une seule classe. On isole la technologie (`std::thread`, `pthread`, `boost::thread`, etc.) et on peut répercuter un changement dans le modèle sur l'ensemble de l'application très facilement.

La partie `using` n'est pas du simple sucre synthaxique puisqu'il contribue également à la maintenabilité de l'application. L'utilisateur final utilise des types en « dur », sans template, permet si le besoin s'en fait sentir de ne changer le template à un unique endroit.

1.2.3 Pattern Chaine de responsabilité

Logger

Chapitre 2

Modélisation

2.1 Analyse fonctionnelle

2.1.1 Introduction

2.1.2 Cas d'utilisation

2.1.3 Scénarios

2.2 Briques logicielles

2.3 Partitionnement et indexation spatial

2.3.1 QuadTree

2.3.2 BSP

2.3.3 Arbre kd

2.3.4 Implicit k-d tree

2.3.5 Vantage-point tree

2.3.6 R Tree

2.3.7 R+Tree

2.3.8 R+Tree

2.3.9 R*Tree

2.3.10 Hilbert R Tree

2.4 Algorithme de plus court chemin

2.4.1 Algorithme de Dijkstra

2.4.2 A*

2.4.3 IDA*

2.4.4 D*

Chapitre 3

Implémentation

Chapitre 4

Application

Chapitre 5

Livrable

Conclusion