Manuel d'utilisation de $\mathcal{XB}orne$ V1.3

J.M. Fourneau
DAVID lab, Université de Versailles St Quentin
45, avenue des États-Unis, 78000 Versailles, France
Email: Jean-Michel.Fourneau@uvsq.fr

April 13, 2017

Acknowledgement: Ce travail a été subventionné par plusieurs projets ANR: en particulier ANR SETIN 2006 CheckBound, ANR Blanc 2005 SMS, ACI Securité SurePaths, ANR MN MARMOTE. Merci à M. Ben Mamoun, A. Busic, N. Pekergin, F. Quessette, Y. Ait El Mahjoub, D. Vekris pour leurs contributions.

Contents

Chapter 1

Introduction

$\mathcal{XB}orne$ est un système ouvert permettant de construire des chaînes de Markov en temps discret et de les analyser quand la chaîne est de petite taille ou de les borner au sens d'un ordre stochastique quand elle est trop grande. L'analyse consiste essentiellement à calculer la distribution stationnaire et des récompenses sur cette distribution mais dans un proche avenir on devrait pouvoir calculer des distributions transitoires, des probabilités d'absorption et des temps moyens d'absorption.

Nous allons surtout décrire les algorithmes de bornes, de génération ou de manipulation. Puis, dans une approche par l'exemple, nous détaillerons plusieurs modèles élémentaires.

Puisque les chaînes sont en temps discret, on peut facilement modéliser des systèmes en temps discret. Mais, on peut aussi construire la version uniformisée d'un modèle en temps continu.

Le travail de comparaison stochastique s'effectue souvent à partir des équations d'évolution. On peut aussi se servir de ces équations pour spécifier une chaîne, ce que nous faisons ici. Dans tous les cas, obtenir la chaîne à partir des équations d'évolution reste une tâche délicate et fastidieuse conduisant éventellement à des erreurs de codage et donc d'analyse. Ils nous a donc semblé qu'il était indispensable de développer un outil générique qui utilise les équations d'évolution pour générer automatiquement la matrice de transition de la chaîne. La version actuelle suppose que l'on fournisse le code de plusieurs fonctions qui décrivent les équations d'évolution, l'espace des états ainsi que les probabilités des événements qui font évoluer le système.

On représente des systèmes multidimensionnels. C'est-à-dire qu'un état a plusieurs composantes. Les états sont numérotés lors de leur génération. Il faut donc pouvoir indiquer comment passer de cette numérotation à une représentation sur l'espace des composantes. Cette foncton de correspondance est construite automatiquement et stockée sur disque. Mais il est nécéssaire de fournir des indications de dimensionnement pour constuire la table de correspondance. Cette table sert également lorsqu'on calcule une récompense puisque la fonction de récompense est exprimée à l'aide des valeurs sur les composantes.

Une autre idée importante à toujours prendre en compte est que les modèles sont très gros et que les matrices ne tiennent pas en mémoire mais restent sur disque. Nous devons donc construire des outils qui manipulent les matrices sur disque pour les changer de format. En effet, les matrices générées le sont souvent par ligne (c'est à dire en donnant les successeurs d'un état) alors que certains algorithmes nécéssitent un accès par colonne, voire quelque chose de plus complexe. Il est donc nécéssaire de définir les différents éléments pour stocker une matrice sur disque et les formats de fichiers.

Le dernier point important concerne l'architecture des programmes. Pour constuire et étudier un modèle, il faut ajouter des fonctions C au programme général. Ces fonctions décrivent les parties spécifiques de chaque modèle mais elles doivent respecter une signature spécifique.

A terme, on devrait être capable de générer et d'étudier :

- des chaînes de Markov en temps discret
- des chaînes censorées
- des chaînes imprécises représentées par des ensembles de chaînes

- des automates
- des chaînes de Markov en temps continu
- des "stochastic reaction networks"

Enfin, ce document ne consitue ni une introduction ni un tutoriel sur les bornes stochastiques et leurs versions algorithmiques. On conseille plutôt de se référer à [12] pour les algorithmes et à [17, 16, 18] pour les aspects fondamentaux.

Chapter 2

Générer une matrice stochastique

Le principe est de fournir le code C de plusieurs fonctions qui décrivent l'espace des états et les transitions. Une transition est générée par un événement et est caractérisée par un état initial, un état final et une probabilité. Le générateur de matrices doit être recompilé avec ces nouvelles fonctions.

Il y a trois fichiers à modifier pour construire un nouveau modèle: "const.h", "var.h" et "fun.c". "var.h" est en général vide et sert à ajouter de nouvelles variables globales, au besoin. La description des variables et fonctions à modifier dans les deux autres fichiers est donnée ci-après.

2.1 Les états

Le modle fondamental est celui d'un espace d'état à plusieurs composantes. Un état est donc un vecteur d'entier de taille constante. On doit donc dans le fichier "const.h" définir la valeur de NEt, le nombre de composantes. Ces composantes seront numérotées de 0 à NEt-1. Cette constante permet de définir un Etat comme un vecteur d'entier:

```
typedef int Etat[NEt];
```

On construit les états atteignables depuis un état initial précisé dans le code en appliquant un algorithme de visite. Les numéros des états dans la matrice sont fixés lors de la premìere visite.

Il est donc nécéssaire de décrire le nombre de composantes et l'état initial, ainsi que la taille de chaque composante. Cette dernière information sert à gérer la numérotation des états. En effet, la matrice étant une structure plate et non pas un tenseur, et les états étant numérotés dans un ordre liés à la visite du graphe, il faut garder une correspondance entre l'ordre de génération et la représentation dans l'espace des composantes. Il est alors utile de connaitre les valeurs minimale et maximale de chaque composante. Ceci est réalisé grâce à la function InitEtendue située dans le fichiier "fun.c" qui permet d'initialiser deux tableaux Min() et Max(). Les indices varient de 0 à NEt-1.

Attention, ceci permet de calculer un espace produit englobant les états atteignables. Cet espace produit a pour taille $\pi_{j=1}^{NEt}(Max(j)-Min(j)+1)$. Et deux vecteurs de correspondance entre numérotation des états sont dimensionnés à cette taille. Ceci peut rendre impossible la génération (voir liste des messages d'erreur) même si l'espace atteignable est beaucoup plus petit. Il faut donc dimensionner avec précaution.

De plus, dans le fichier "fun.c", on doit définir la function void EtatInitial(E), qui retourne l'état initial E, racine de l'arbre d'atteignabilité généré. Cet état est la racine d'une visite en largeur du graphe. On ne fait donc que vérifier l'atteignabilité depuis cette racine et la forte connexité n'est jamais testée.

2.2 Les transitions

Le système est représenté par des états et des événements. Ces événements provoquent des transitions et sont probabilisés. Plusieurs événements peuvent provoquer la même transition et dans ce cas, les probabilités s'ajoutent. Pour définir un modèle, il faut donner le nombre d'événements, associer à chacun d'eux une probabilité et décrire la transition provoquée à partir de l'état courant.

La première étape est donc de spécifier le nombre d'événements dans le fichier "const.h" grâce à la constante NbEvtsPossibles. Les événements sont numérotés entre 1 et cette valeur.

On a ensuite à définir les probabilités et les transitions pour chaque événement grâce à deux fonctions contenues dans le fichier "fun.c":

- void Equation(E, indexevt, F, R)

  Cette fonction décrit la transition provoquée par l'événement de numéro *indexevt*. L'état initial est E et l'état terminal F. La recompense est un réel associé à l'arrivée de cet événement.

- double Probabilite(indexevt, E)

  Cette fonction retourne la probabilité d'apparition de l'événement *indexevt*. Cette probabilité peut dépendre de l'état E.

Les transitions de trop faible probabilité peuvent être omises. La constante $Epsilon1$ (dans le fichier const.h) est le seuil utilisé. Seules les transitions supérieures à $Epsilon1$ sont écrites dans la matrice de transtion.

2.3 Autres

- La fonction InitParticuliere() dans le fichier "fun.c" sert à initialiser des données communes. Elle est en général vide.

- La constante $Epsilon2$ sert à vérifier la différence avec 1.0 comme somme des probabilités d'une ligne quelconque. Si la différence est supérieure à $Epsilon2$ un message d'erreur est envoyé et la génération stoppe.

- Il y a dans le fichier "const.h" une dernière variable utile : Polynom. Selon sa valeur, on va générer la matrice de la chaîne ou des polynomes de cette matrice $Pol(M)$ selon le codage suivant:

0	M
1	$M/2 + Id/2$
2	$M^2/2 + Id/2$
3	M^2

  Si la somme des coefficients de ces polynomes vaut 1 et qu'ils sont tous positifs, alors $Pol(M)$ a la même mesure invariante que M mais surtout il est prouvé que l'algorithme de borne de Vincent ou IMSUB (avec $\epsilon = 0$) donne une borne plus précise [9, 10] quand M est "row diagonal dominant" (ce qui est obtenu en prenant 1). Attention les valeurs différentes de 0 ou 1 n'ont pas encore été testées. Il est conseillé de laisser cette variable Polynom à 0 si on ne veut pas faire de bornes stochastiques.

2.4 Codage des Evenements globaux

La plus grande difficulté est de décrire les événements: leur nombre, leur probabilité et leur effet. Puisque nous modélisons des systèmes à temps discret, on doit considérer des événements globaux alors que la description la plus intuitive des modèles consiste en des événements locaux. Dans les cas simples, il y a indépendence des événements locaux et les événements globaux sont obtenus par produit. Plus précisement:

- Le nombre d'événements (la constante NbEvtsPossibles) est le produit des nombres d'événements locaux. Ainsi, si il y a 3 événements d'arrivée dans une file et deux événements de service, il y 6 événements globaux.
- la probabilité d'un événement global est en cas d'indépendence le produit des probabilités des événements locaux.

Pour construire une bijection entre les numéros d'événements locaux et le numéro global on commence par utiliser une suite d'opérations div et mod, classiques lorsqu'on manipule des produits Cartésiens. On se reportera utilement aux chapitres d'exemples.

2.5 Créer le programme de génération

Il reste à créer un nouvel exécutable en compilant les programmes de génération :

- "GenerMarkov.c" pour la génération exhaustive.

Pour résumer, voici les constantes et fonctions :

Méthode	Constantes	Fonctions
GenerMarkov.c	NEt, NbEvtsPossibles, Epsilon1, Epsilon2 Polynom	Probabilite, Equation, EtatInitial, InitEtendue, InitParticuliere

Le programme récupère en entrée le nom du modèle (model dans la suite) avec l'option -f et crèe les fichiers suivants (pour les formats, voir le chapitre) :

- "model.sz" : la taille de la matrice
- "model.cd" : la description des états générés, c'est à dire la description dans l'espace des composantes des numéros des lignes et colonnes de la matrice.
- "model.Rii" : les transitions dans le format ligne.

Si les fichiers existent déjà, ils ne sont pas détruits et un message d'erreur est envoyé avant que la génération stoppe.

Chapter 3

Tester des propriétés

On regroupe ici les algorithmes dont le but est de vérifier que certaines propriétés sont vérifiées de manière à privilégier l'emploi d'une méthode ou d'un solveur. Pour les propriétés sur les NCD et sur la lumpabilité, voir le chapitre sur les manipulations de matrice.

3.1 St-Monotonicité

La syntaxe est "IsMonotone -f model". La commande retourne un message disant si la chaine est monotone et indique quand elle ne l'est pas, où on a détecté un contrainte non vérifiée. Le programme utilise les fichiers "model.sz" et "model.Rii".

3.2 Atteignabilité

La syntaxe est "AllReachable -f model". La commande retourne un message disant si tous les états de l'espace produit sont atteignables. Le programme utilise les fichiers "model.sz", "const.h" et "fun.c" employés pour créer la matrice.

3.3 Irréductibilité

La syntaxe est "TarjanSCC -f model suffix". La commande retourne un message disant si la matrice est irréductible ou non. La matrice en entrée est stockée en format ligne (mais pas nécessairement Rii). Le programme utilise les fichiers "model.sz", et "model.suffix".

To Be DONE: icx-monotone, positive nabla

Chapter 4

Manipuler une matrice stochastique

Le générateur construit la matrice par ligne (pour chaque état, on donne la liste des successeurs). Et l'ordre des états n'est pas contrôlable puisque les numéros sont attribués lors de la visite en largeur. Or, les ordres stochastiques "st" ou "icx" demandent que la récompense soit une fonction croissante du numéro des états. Il faut donc renuméroter ou adapter la récompense. Dans ce chapitre, on examine la première solution.

De plus, certaines méthodes de bornes ou d'analyses nécéssitent que la matrice soit partitionée en blocs dont les états sont consécutifs. Certaines méthodes numériques utilisent des matrice en format Cii. Ces contraintes sont prises en compte par les outils de renumérotation et de conversion.

Enfin, on peut préparer grâce à un test de quasi-lumpabilité l'agrégation de la matrice. On trouve donc aussi les outils pour préparer des partitions et les appliquer à une matrice pour faire une agrégation. Le travail est décomposé grâce à divers outils. On peut créer des partitions en utilisant un algorithme de test de quasi-lumpabilité. On peut également définir a-priori des partitions.

4.1 Tester la Quasi-Lumpabilité et la lumpabilité ordinaire

L'application "Tarjan" construit à partir d'une matrice et d'un seuil, une liste de macro-états pour que la chaine soit quasilumpable. Lorsque le seuil est très petit (de l'ordre de l'incertitude sur le type flottant) on retrouve la partition optimale au sens de Tarjan modifié par Franceschini. On ne modifie pas la matrice mais on donne simplement la composition des blocs. La méthode emploie en entrée une première liste de macro-états qui sont ensuite subdivisés. La syntaxe est la suivante :

```
Tarjan -f model nom epsilon option
```

avec les arguments suivants:

- model est un nom de matrice, il permet de lire "model.Rii" et "model.sz"
- nom est un nom de variante, il permet de distinguer entre plusieurs variantes pour l'agrégation.
- epsilon est une valeur numérique pour tester la quasi-lumpabilité. Si cette valeur est nulle, on teste la lumpabilité ordinaire.
- option est un champs supplémentaire qui n'est utilisé que lorsque epsilon est supérieur à 0 et qui sert à choisir l'heuristique de création des blocs.
  - arbre (comme Tarjan)
  - ascendant
  - descendant

Les fichiers en entrée sont : "model.Rii", "model.sz" pour la matrice, et "model.part" pour la partition initiale. Les sorties sont les fichiers "model.Tarjan.nom.part" qui décrit les macro-états et "model.Tarjan.nom.epsi" qui contient la valeur de epsilon pour les applications qui suivront.

4.2 Tester la propriété NCD

L'application NCD construit la partition associée à la propriété de "Near Complete Decomposibility" pour le seuil epsilon. Apres renumérotation selon cette partition, les blocs hors diagonale principale ne contiennent que des éléments inférieurs à epsilon. L'algorithme retire les transitions inférieures à epsilon et cherche les composantes fortement connexes du graphe ainsi modifié grâce à l'algorithme de Tarjan. La syntaxe est

```
NCD -f model suffix nom epsilon
```

où suffix est un suffixe pour une matrice en format ligne (pas nécessairement Rii) et nom est un nouveau suffixe pour les fichiers de sortie. Les noms des fichiers de sortie sont "model.NCD.nom.part" qui décrit la partition et "model.NCD.nom.epsi" qui contient la valeur de epsilon. Les fichiers "model.sz" et "model.suffix" sont lus pour construire le graphe.

4.3 Créer des blocs liés aux récompenses

Le principe est le même que pour la génération : l'utilisateur fournit une fonction en C dans le ficier "funnum.c" qui donne l'ordre des états en donnant une numéro de bloc. La fonction se nomme "NumerodeBlock(et, j)" et retourne un long. Elle a en argument un état ("et") représenté par ses composantes et par son numéro initial ("j") et retourne un numéro de bloc supérieur ou égal à 0. Tous les états ayant le même numéro de bloc sont rangés consécutivement. A l'intérieur d'un bloc, l'ordre est donc l'ordre de visite lors de la création du modèle.

Comme il est assez difficile de trouver des numéros de bloc consécutifs directement sur le modèle, le programme tolère que certains blocs ne contiennent pas d'états. Ces blocs vides ne sont alors pas pris en compte et les blocs sont renumérotés pour éviter les vides. En conséquence, il est nécéssaire de tolérer que les numéros de blocs fournis par la fonction "NumerodeBlock" soient plus grands que le nombre de sommets atteignables. Ce ratio est fixé à deux dans le code (voir la constante MaxBlockNumber) La matrice n'est pas en mémoire et n'est pas modifiée par ces fonctions. On ne fait que construire le fichier de description des blocs. Ce fichier est est un fichier ".part" comme pour la méthode de Tarjan.

La syntaxe est "GiveABlockNumber -f model nom" et la description de la partition est dans le fichier "model.own.nom.part".

4.4 Partitions liées au graphe

La commande "Stable" construit une partition de l'espace des états telle que le blow NW (après avoir modifié l'ordre des états avec la commande "Reorder" est diagonal. La syntaxe est

```
Stable -f model suffix
```

où "suffix" indique le suffixe pour une matrice en format ligne (Rii ou plus général). La commande ne fait que créer le fichier "model.stable.part" qui décrit la partition. Il faut ensuite faire un "Reorder".

4.5 Agréger une matrice

L'application "Lump" fait une agrégation exacte ou approchée en prenant en compte les blocs proposés par l'utilisateur. Les sorties dépendent de la valeur du seuil epsilon (dans le fichier "model.method.nom.eps"). La syntaxe est

```
Lump -f model method nom
```

Les fichiers d'entrée sont "model.sz", et "model.Rii" qui décrivent la matrice, "model.method.nom.part" et "model.method.nom.eps" qui décrivent la partition.

Si epsilon est nul, on fait une agrégation exacte, les fichier en sortie sont "model.method.nom.lump.Rii" et "model.method.nom.lump.sz" pour décrire la nouvelle matrice agrégée.

Si epsilon est plus grand que 0, on construit deux matrices agrégées qui sont respectivement des bornes supérieures et inférieures par élément. Leurs noms sont "model.method.nom.lump.EU.Rii" "model.method.nom.lump.EU.s pour la borne sup et "model.method.nom.lump.EL.Rii" "model.method.nom.lump.EL.sz" pour la borne inf.

Dans les deux cas, il n'y a pas de création automatique de fichier ".cd" pour décrire les états. C'est à l'utilisateur de le faire en examinant les blocs.

4.6 Renuméroter

On peut ainsi définir une permutation pour réordonner les états.

L'application "GiveANewNumber" donne un nouveau numéro d'état. Les blocs ne doivent pas être vides et doivent être de taille 1. Un message d'erreur est envoyé dans le cas contraire avec le premier état posant probeème. Elle emploie la fonction "NumerodeBlock" contenu (et modifié par l'utilisateur) dans "funnum.c" comme le programme "GiveABlockNumber". Le fichier résultat est dans un fichier ".perm". La syntaxe est

"GiveANewNumber -f model nom" et la description de la permutation est dans le fichier "model.nom.perm".

L'application "GiveLexicographicNumber" donne à chaque état son numéro dans l'ordre lexicographique. Tous les états doivent être atteignables, sinon un message d'erreur est envoyé. Le programme utilise le fichier "fun.c" employé pour générer la matrice. Il doit donc être dans le même dossier.

La syntaxe est "LexicographicOrder -f model" et la description de la permutation est dans le fichier "model.lexi.perm".

4.7 Reordonner et Permutter

L'application "Reorder" crèe un fichier des transitions en format Rii en faisant en sorte que les états d'un même bloc soient consécutifs. C'est une des conditions d'applications de la méthode LIMSUB qui crèe une borne lumpable selon les blocs choisis (voir chapitre sur les bornes) mais c'est aussi utile pour les méthodes pour matrices NCD, ou pour des méthodes de résolution numérique par bloc. Les blocs sont rangés dans l'ordre de leur numéro. La syntaxe de la commandes permet de donner le nom du modele et un suffixe employé pour désigner le modèle renuméroté. La syntaxe est :

```
Reorder -f model method nom
```

Les fichiers d'entrée sont "model.sz", "model.Rii" et "model.cd" qui décrivent la matrice et "model.method.nom.part" qui décrit la partition. Le programme construit les fichiers "model.reorder.nom.Rii", "model.reorder.nom.cd", "model.reorder.nom.sz" pour décrire la nouvelle matrice, "model.reorder.nom.bk" pour donner les tailles des blocs, et "model.reorder.nom.perm" pour décrire la permutation réalisée.

L'application "Permute" applique une permutation à une matrice stockée en .Rii. La syntaxe est

```
Permute -f model nom
```

Les fichiers d'entrée sont "model.sz", "model.Rii" et "model.cd" qui décrivent la matrice et "model.nom.perm" qui décrit la permutation. Le programme construit les fichiers "model.nom.Rii", "model.nom.cd", et "model.nom.sz" pour décrire la nouvelle matrice.

La commande "Split" a pour but de décomposer une matrice en 4 blocs. La syntaxe est

```
Split -f model taille
```

où taille est le nombre d'états dans le premier bloc. On prend les premiers états de la chaine. Il faut donc avoir dans une étape précédente ré-ordonné la chaine. La commande crèe les matrices "model.NW", "model.NE", "model.SW", et "model.SE", chacune étant décrite par un fichier ".Rii" et un fichier ".sz".

4.8 Convertir

Il s'agit d'une conversion ligne-colonne et non pas d'un changement pour un autre format (par exemple HBM). La manipulation se fait sur disque grâce à deux fichiers intermédiaires et à l'utilisation de la commande unix "sort". La commande est donc assez longue à cause des accès aux disques mais elle peut agir sur une matrice de très grande taille qui ne tient pas en mémoire. La commande Convertir a deux arguments : le nom du fichier d'entrée et le suffixe cible qui donne les propriétés du fichier de sortie.

4.9 Visualiser

L'outil "Lam2TGF" permet de transformer la matrice en un graphe orienté dont les noeuds sont étiquetés mais sans information sur les arcs. Les boucles éventuelles sont conservées. Le fichier de sortie est codé en "trivial graphe format" (tgf). Il se somme model.tgf pour être lu par "yed" ou un autre outil de dessin de graphe permettant la visualisation à partir de graphes décrits en tgf.

L'outil "Lam2TGFWithoutLoops" fait un travail équivalent mais enlève les boucles pour obtenir un dessin plus lisible. Il y a aussi un outil de génération d'automates (voir GenerAutomata) directement à partir des spécifications du modèlle où on peut donner des étiquettes de transition aux arcs.

Il existe sur le web des outils de visualisation de matrice creuse. Il faut utiliser les programmes de conversion (voir MatView, Matrix Market, Summon et en particulier summatrix).

4.10 Exporter

Les programmes utilisent "model.sz" et "model.Suffixe".

- La commande Lam2Latex permet d'écrire dans un fichier textes de nom "model.tex" les commandes latex pour définir la matrice de transition qui peut alors être copiée dans un document Latex.
- "Lam2Sci" et "Sci2Lam" convertissent vers (et de) Scilab
- "Lam2MMArray" convertit au former Matrix-Market plein.
- "Lam2MMSparse" convertit au former Matrix-Market creux.
- Il faudrait aussi des programmes de transformation au format HBM.

4.11 Convertir les matrices de rang faible en matrice creuse

Dans la même catégorie d'outil, on trouve deux utilitaires pour convertir une matrice ayant une décomposition de rang faible donnée par son descripteur en une matrice explicite.

- Le premier se nomme "Desc2Lam" et il est limité à des matrices de taille inférieure à 1000. il traduit les matrice de classe C généralisées qui sont des matrices ayant une décomposition de rang 2.
- L'autre utilitaire se nomme "LR2Rii" et il utilise les matrices LR.

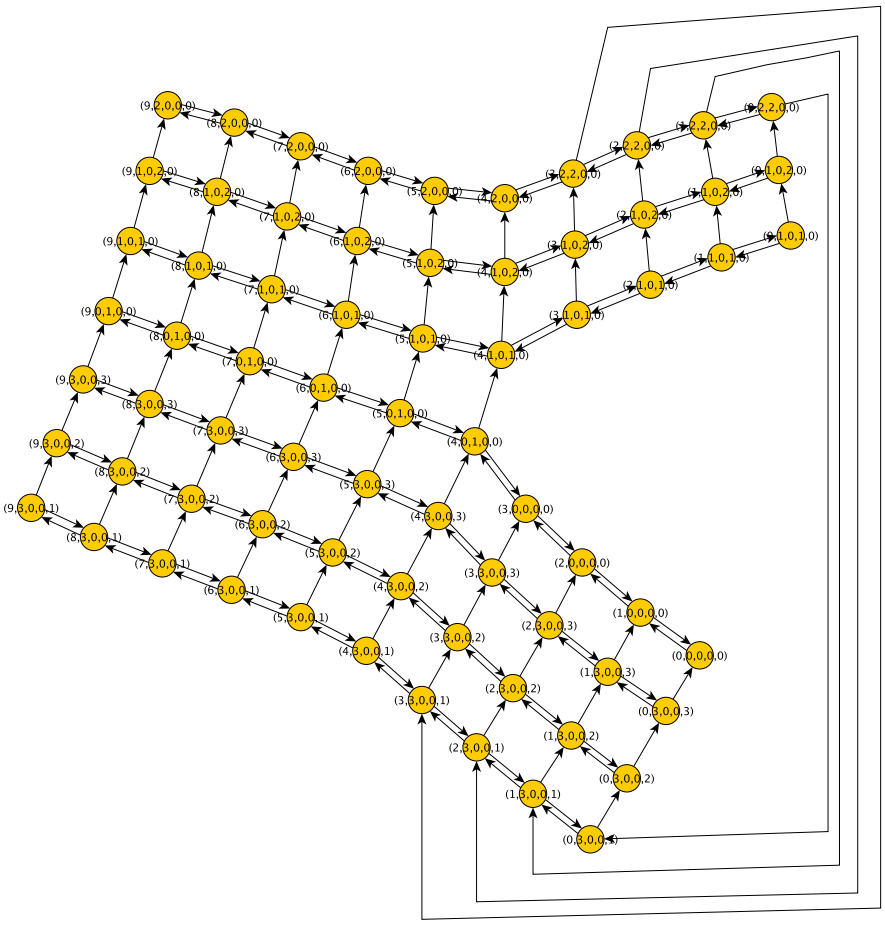

Figure 4.1: Modèle étendu de Mitrani pour un data center, construit par yed à partir de Lam2TGFWithoutLoops.

Chapter 5

Résoudre

5.1 Algorithmique usuelle pour le stationnaire

Il y a juste quelques algorithmes usuels. Pour plus de précisions sur les algorithmes de résolution numérique de la distribution stationnaire d'une chaine de Markov, on peut consulter [20].

5.1.1 GTH-Plein

Il s'agit d'une version en mémoire pleine de l'algorithme de Grassman, Takacs et Heyman [14]. L'algorithme est de complexité cubique en temps et quadratique en espace. Mais il est très stable numériquement [15] et il convient donc parfaitement à l'analyse de petite matrices (disons de taille inférieure à 1000).

Il lit en entrée un fichier de matrice dans un format quelconque de type Rxx et produit un fichier contenant la distribution stationnaire. Il n'y a deux arguments : le nom du modèle et le suffixe. On vérifie que le suffixe commence par R.

Le nom de la commande est gthLD. On passe en argument le nom d'un modèle, et le suffixe et on lit alors les fichiers model.sz et model.suffixe. La distribution stationnaire est écrite dans model.pi.

5.1.2 Gauss Seidel

Il s'agit d'un algorithme classique itératif, ici implémenté en version matrice creuse et sa convergence dépend du spectre. Le test d'arrêt porte simplement sur la différence entre deux itérations successives. Le seuil est fixé dans le fichier "GaussSeidel.h". Il lit en entrée un fichier de matrice dans un format Cuu (mais pas les matrices de classe C) et produit un fichier contenant la distribution stationnaire. Il n'y a pas d'autres arguments que le nom du modèle. Par contre, il est possible de modifier le fichier GaussSeidel.h pour y modifier les constantes et changer ainsi le comportement de l'algorithme.

```
// Maximal Iteration number
#define maxiter         100000
// Accuracy
#define maxdiff   1.0e-15
// Test for end : 1= Stewart, 0=sum des differences absolues
#define TestFin   1
// Comparison of pi_t and pi_{t-m}
#define mmm  10
// GS Forward  (1) ou Backward (0)
#define Frwrd 1
```

Les constantes sont:

- maxiter : le nombre maximum d'itérations avant l'arrèt du programme
- maxdiff : la précision requise pour la convergence
- TestFin : Description du calcul du test de convergence. Il vaut 0 ou 1. Lorsque la valeur est à 0, on calcule la somme des valeurs absolues des différences entre deux vecteurs de probabilités.

$$diff = sum_i |\pi_1(i) - \pi_2(i)|$$

Lorsqu'il est à 1, ce calcul est normalisé comme suggéré dans [20] page 158.

$$diff = max_i \frac{|\pi_1(i) - \pi_2(i)|}{\pi_1(i)}$$

5.1.3 SOR

Il s'agit d'un algorithme classique itératif, ici implémenté en version matrice creuse et sa convergence dépend également du spectre. La valeur de mélange de l'algorithme (Omega) et la valeur du test d'arret relatif (maxdiff) sont fixés dans le fichier Sor.h. Le test d'arret est semblable à celui de GaussSeidel et utilise les constantes "mm" pour le décalage et "TestFin" pour le type de test.

```
#define maxiter         100000
#define maxdiff   1.0e-15
#define TestFin   1
#define mmm  10
#define Omega 0.8
```

Le programme lit en entrée un fichier de matrice dans un format Cuu et produit un fichier contenant la distribution stationnaire. Il n'y a pas d'autres arguments que le nom du modèle.

5.1.4 Méthode des puissances

Logiquement la méthode des puissances est moins efficace que les deux méthodes précédentes. Elle est cependant disponible avec la même interface que les deux autres méthodes itératives et se nomme Power. Les deux paramètres sont dans le fichier "Power.h". Il s'agit de "maxiter", le nombre maximum d'itération et de "maxdiff", la norme de la différence entre deux valeurs successives de la distribution. La matrice est lue en format creux en colonne (format Cuu).

5.2 La classe C, la classe G et les décompositions de rang faible

Pour la classe C ou la généralisation (classe G), les algorithmes opèrrent à partir du fichier descripteur ("model.dscC"). Il faut donc commencer par le construire. Puis on calcule le stationnaire par le programme "SteadyClassC" et le transitoire par "TransientClassC". Pour le stationnaire, la syntaxe est

```
SteadyClassC -f model
```

Pour le transitoire, la syntaxe est

```
TransientClassC -f model time
```

On obtient dans le premier cas, un fichier nommé "model.pi", dans le second un fichier nommé "model.pi_time". Dans les deux cas, on liste la distribution de probabilités (une valeur par ligne).

Dans le cas des matrices de rang faible quelconque, il s'agit d'une généralisation de la classe C mais le descripteur est maintenant dans un fichier .LR. Les commandes sont :

- pour le stationnaire, "SteadyLR -f model". Le fichier descripteur doit être "model.LR". Les résultats sont écrits dans "model.pi"
- pour le transitoire, "TransientLR -f model time". Le fichiers d'entrée doivent être "model.LR" (pour le descripteur) et "model.pi_0" pour la distribution initiale. Les résultats sont écrits dans "model.pi_time"
- pour une méthode potentiellement plus rapide pour le transitoire, "FastTransientLR -f model time". Le fichiers d'entrée doivent être "model.LR" (pour le descripteur) et "model.pi_0" pour la distribution initiale. Les résultats sont écrits dans "model.pi_time"

La deux algorithmes pour le transitoire n'emploient pas la même méthode et il n'y a pas actuellement de comparaisons de résultats.

5.3 Algorithmes Nabla

Cinq algorithmes sont présents. Les quatre premiers s'appliquent quand le vecteur Nabla est non nul. Le dernier algolrithme quand Nabla est nul. Dans tous les cas, le fichier 'Nabla.h" contient le nombre maximum d'itération autorisées et la précision permettant de stopper l'algorithme.

- INABLAL: construit une séquence de bornes inférieures par éléments, croissante et convergeant vers la distribution stationnaire. Le test de convergence est prouvé. Utilise en entrée une matrice creuse en format "Cuu".
- INABLAU : construit une séquence de bornes supérieures par éléments, décroissante et convergeant vers la distribution stationnaire. Le test de convergence est prouvé. Utilise en entrée une matrice creuse en format "Cuu".
- DynamicL, comme INABLAL mais avec une matrice pleine et en format "Rii".
- DynamicU, comme INABLAU mais avec une matrice pleine et en format "Rii".
- SLUB : construit deux séquences de bornes inférieures et supérieures de la distribution stationnaire. Le programme doit lire une séquence d'initialisation est qui est inférieure par élément à la distribution stationnaire. Cette séquence doit se trouver dans le fichier "model.init.pi' ou "model" est le nom du modèle qui a été fixé à la génération. Les deux séquences convergent vers des multiples distincts de la distribution stationnaire. Le test de convergence n'est pas prouvé.

5.4 Algorithmique usuelle pour les matrices absorbantes

Pour déterminer les points absorbants il faut utiliser le programme "Absorbing" qui crèe une partition où les sommets absorbants sont en tête. La syntaxe est

```
Absorbing -f model suffix
```

Le fichier d'entrée est en format ligne mais pas obligatoirement en .Rii. La commande retourne un message d'erreur si il n'y pas de points absorbants. Sinon elle en donne le nombre et construit la partition dans le fichier "model.part". On doit ensuite réordonner la matrice pour mettre effectivement les sommets absorbants en tête (voir le programme Reorder). et décomposer la matrice en 4 blocs à l'aide de la commande Split (voir le programme Split). On suppose maintenant que la matrice ne contient pas de classes récurrentes (voir utilitaire de recherche de composantes fortement connexes).

La matrice fondamentale d'une chaine absorbante sans classe récurrente est calculée grâce à la commande "Fundamental" qui implémente l'algorithme de Sheskin [19]. La syntaxe est

```
Fundamental -f model suffix
```

La matrice d'entrée doit être en format ligne. Et il s'agit du bloc transitoire de la matrice absorbante obtenue grâce à la commande "Split".

Le résultat est une matrice dont le nom est "model.FUND" qui est stockée dans un fichier ".Rii" et un fichier ".sz". La matrice fondamentale est vraisemblablement très remplie. En mémoire, elle est stockée sous forme pleine. Il faut donc éviter de calculer une matrice fondamentale si le bloc des transitoires a plus de 1000 états.

5.4.1 Probabilités d'être absorbé

Pour calculer les probabilités d'être absorbé par les différents sommets, on doit faire un produit de la matrice fondamentale par le bloc projetant les états transitoires sur les absorbants : La syntaxe est

```
ProdFundSW -f filename
```

Le résultat est écrit en texte dans le fichier de nom "filename.rwdP" et sous forme d'une matrice bloc SW stockée avec les deux fichiers ".Cii" et ".sz". C'est un fichier ".SW" car l'indice de ligne correspondent aux états transitoires et l'indice de colonne aux états absorbants et que la commande de recherche des états absorbants les a mis en tête.

Avant de faire le produit, il faut avoir construit la matrice d'entrée .SW en format Cii car on fait un produit de la matrice fondamentale par le bloc SW. On a donc besoin de SW en colonne. Utilisez la commande "convertir".

5.4.2 Temps moyen avant d'être absorbé

Pour calculer le temps moyen avant absorption, on applique l'utilitaire "RowSum" à la matrice fondamentale. La syntaxe est

```
RowSum -f filename
```

Le nom "filename" est typiquement "model.FUND". Le résultat est stocké en colonne dans le fichier "filename.rwdT".

Chapter 6

Calculer des récompenses

Il y a deux programmes pour calculer des récompenses, selon que le modèle possède ou pas un fichier .cd pour décrire les codages des états. En effet, la génération du modèle permet de générer une telle description mais elle n'est pas modifiée si on agrège la matrice en employant LIMSUB ou LMSUB. Elle est alors incohérente.

On calcule des récompense sur les états pour une distribution stationnaire ou transitoire. Quand le fichier .cd est absent, les récompenses possibles sont peu nombreuses. Par contre si le modèle comprfend un fichier de description des états, il est possible de définir de nouvelles récompenses. Une récompense d'état est définie par $R = \sum_i r(i)\pi(i)$ où π est une probabilité stationnaire ou transitoire.

6.1 Récompenses liées aux indices

Si on ne possède pas une description des états, on calcule simplement quelques récompenses usuelles sur la distribution de probabilités:

- la moyenne
- la variance
- l'état le plus probable
- la fonction de répartition
- les probabilités de fin de distribution : $\sum_{j>k} \pi(j)$.

La syntaxe est "rewardIndice -f model". les résultats sont écrits dans le fichier "model.rwdI".

6.2 Récompenses liées aux états

On utilise le même principe : définir des récompenses dans un fichier (nommé ici funReward.c") puis le recompiler. Il faut aussi que les fichiers "fun.c" et "const.h" qui ont servis à construire la matrice soient présents dans le dossier. Ils vont tre inclus. Les fichiers "model.pi", "model.sz" et "model.cd" vont être lus par le programme pour calculer les récompenses.

Le programme fonctionne comme suit. Il lit la distribution et le codage des états. Puis il calcule les marginales, les écrit dans le fichier résultat, puis il calcule $\sum_{i=Min}^{Max} i\pi_k(i)$ pour toutes les composantes k de l'espace des états. Il calcule la fin de la distribution et la fonction de répartition. Et il les imprime. Puis pour chaque composante, il passe la main à une fonction fournie par le modélisateur dans le fichier "funReward.c". Cette fonction ne retourne rien. Elle doit écrire les résultats de son calcul dans le fichier. Elle se nomme "MarginalMesCalculs". Aprs l'analyse de toutes les composantes, le programme appelle

une seconde fonction (nommée "GlobalMesCalculs") fournie par l'utilisateur qui effectue des calculs sur la distribution globale (et non pas sur une marginale).

- MarginalMesCalculs : ne retourne rien. Les arguments sont deux entiers : le numéro de la composante et le nombre d'états atteignables. La distribution de probabilité marginale est dans le vecteur de long double "marginale" qui est une variable globale. Les résultats sont écrits sur le fichier "model.rwdC" en utilisant le pointeur de fichier "pf1".
- GlobalMesCalculs : ne retourne rien. Le seul argument est le nombre d'états atteignables (un entier). La distribution de probabilité est dans le vecteur de long double "pi" qui est une variable globale. La description des états est dans le tableau "et" qui est une variable globale. "et(NEt*i+comp)" contient l'état de la composante "comp" de l'état global "i". Les résultats sont écrits sur le fichier "model.rwdC" en utilisant le pointeur de fichier "pf1".

Commee le fichier "func.c" est inclus dans le programme de calcul de récompense, on peut donc utiliser les fonctions définies dans "fun.c" pour définir les fonctions de "funreward.c". La syntaxe est "rewardComponent -f model". les résultats sont écrits dans le fichier "model.rwdC".

6.3 Autre

On peut aussi simplement trouver les marginales partie de la distribution (le fichier ".pi", du codage le fichier ".cd" et des informations sur le modèle (le fichier "fun.c"). Ceci est réalisé par la commande "Marginale" avec la syntaxe classique :

```
Marginale -f model
```

On crèe un fichier par composante. Les fichiers sont nommés "model.marginale.numero". Le premier numéro est 0 et chaque ligne d'un fichier est composé de l'état et de sa probabilité. L'idée est d'ensuite utiliser R pour analyser les distributions.

Chapter 7

Simuler

L'outil permet également de construire des simulateurs en temps discret des modèles spécifiés par les fichiers "fun.c" et "const.h". Il y a plusieurs programmes de construction de simulateur :

- SimulDTMCInverse
- RegenerationDTMC
- SimulDTMCAndCoverage
- SimulCTMC
- SimulDTMCAlias
- SimulDTMCTrace
- SimulSMPInverse
- SimulSMPTrace
- SimulSMPGeneral

7.1 SimulDTMCInverse

Le premier construit un programme de simulation d'une chaine en temps discret sans aucune hypothèse sur l'espace des ètats. Il faut juste les états "Min" et "Max" pour chaque composante mais aucun objet n'est construit avec l'espace produit. On donne dans le fichier "const.h" les renseignements classiques.

```
#define NEt               2   /*nombre de composantes du vecteur d'etat*/
#define NbEvtsPossibles   6  /*nombre de vecteurs d'arrivees possibles*/
```

On ajoute dans un second fichier nommé "constsim.h" le temps de simulation et le type de sortie.

```
#define Tmax 10  /* simulation time */
#define Compact 1 /* type of output */
```

"Compact" permet de définir le format de sortie (0 pour un format détaillé, 1 pour un format condensé). L'état initial dans le fichier "fun.c" est le point initial de la simulation.

Les résultats sont données par ligne: le premier élément est la date, le second l'état. Si "Compact" est à 1, on écrit la description de l'état visité. Si il vaut 0 on donne simplement le numéro de l'état. On fournit en entrée une graine pour le générateur. On obtient en sortie un fichier nommé "modelegraine.Dpath" qui contient la trajectoire. Les calculs doivent être faits ensuite avec un outil statistique tel que R.

La syntaxe d'appel est donc : (graine est un entier)

```
SimulDTMC -f model graine
```

Pour obtenir plusieurs fichiers il est inutile de recompiler le modèle, il suffit de changer la graine du générateur dans l'appel.

7.2 SimulDTMCAndCoverage

Les fichiers sont identiques et la syntaxe est la même :

```
SimulDTMCAndCoverage -f model graine
```

On commence par générer les états atteignables depuis l'état initial et on crèe le fichier "model.cd". Puis on construit le fichier "model.graine.path" qui contient le chemin avec les mêmes options que la précédente méthode. A la fin de la simulation on retourne aussi le nombre d'états visités et le pourcentage de couverture par cette simulation.

Pour obtenir plusieurs fichiers il est inutile de recompiler le modèle, il suffit de changer la graine du générateur dans l'appel.

7.3 SimulCTMC

Il s'agit maintenant de générer les trajectoires d'une chaine en temps continu uniformisable. Ceci entraine des modifications dans le fichier "fun.h" pour la fonction "Probabilite". Elle retourne maintenant un taux de transition et non plus une probabilité. Les autres fonctions dans "fun.c" ne sont pas modifiées. Le fichier "const.h" contient les mêmes constantes que pour les autres programmes. Il y a de plus un fichier nommé "constsim.h" qui contient des constantes supplémentaires :

- "Compact" pour définir le format de sortie (0 pour un format détaillé, 1 pour un format condensé)
- "Tmax" pour le temps de simulation maximal

7.4 RegenerativeSimul

Les fichiers "const.h", "constsim.h" et "fun.c" sont les mêmes que pour une simulation de DTMC. On doit fournir de plus un fichier "constregeneration.h" qui contient la constante "MinimalCycleLength" qui est le temps de cycle de régénération minimal accepté. La trajectoire de simulation est découpée en cycles de régénération. On suppose que le point initial est le point de régénération. Il est donc spécifié par la fonction "EtatInitial" qui se trouve dans "fun.c". Chaque cycle de régénération est conservée dans un fichier séparé dont le nom est "model.seed.nc.Dpath", où nc est le numéro du cycle. Le premier cycle porte le numéro 1. La syntaxe est :

```
RegenerativeSimul -f model graine
```

7.5 SimulDTMCAlias

Il s'agit d'une méthode de simulation d'une DTMC avec les mêmes descriptions que la méthode SimulDTMCInverse mais la méthode de génération utilise une méthode d'Alias dont la complexité est constante. Elle doit donc être plus rapide quand il y a beaucoup d'événements. La matrice de la chaine de Markov encodée sous forme de tables de la méthode d'Alias doit être fournie en entrée. Elle se somme "model.alias". Chaque transition utilise deux tirages du générateur aléatoire. La syntaxe est:

```
SimulDTMCAlias -f model graine
```

Les fichiers crées sont les mêmes que ceux crées par SimulDTMCInverse mais comme les deux méthodes ne consomment pas les tirages aléatoires à la même vitesse, les trajectoires se séparent.

7.6 SimulDTMCTrace

Il s'agit d'une méthode de simulation d'une DTMC avec les mêmes descriptions que la méthode SimulDTMCInverse mais les variables aléatoires uniformes sont lues dans un fichier trace. Le fichier trace se termine par l'extension ".trace" .

```
SimulDTMCTrace -f model fichiertrace
```

Les fichiers créés sont les mêmes que ceux créés par SimulDTMCInverse. La trace doit être plus longue que Tmax, sinon il y aura une erreur.

7.7 SimulSMPInverse

On simule un processus Semi-Markovien. Par rapport à une DTMC, on reste dans chaque tat avec une dure issue d'une distribution discrete passe en argument. Cette distribution est la même pour tous les tats. Le tirage sur la distribution discrete est effectu par une mthode de transformation inverse. La syntaxe est

```
SimulDTMCAlias -f model graine
```

7.8 SimulSMPTrace

On simule aussi un processus Semi-Markovien mais les variables aléatoires sont dans un fichier trace. Le fichier trace doit contenir exactement trois valeurs par ligne : un numéro d'événements, un flottant entre 0 et 1 pour trouver la transition suivante par une méthode de transformation inverse et un flottant positif qui est la durée de séjour dans cet état. La trace doit permettre de dépasser Tmax sinon il y aura une erreur. La syntaxe est

```
SimulSMPTrace -f model fichiertrace
```

Le fichier trace se termine par l'extension ".trace" .

7.9 SimulSMPGeneral

On simule encore un processus Semi-Markovien mais les variables aléatoires pour les durées sont générées par des fonctions ajoutées au code. Il faut donc modifier le fichier "funsim.c" pour modifier la fonction "Duree()". Cette fonction a en entrée le numéro de l'état et retourne un flottant positif (la durée de séjour dans cet état).

Chapter 8

Bornes stochastiques Strong sur espace d'états totalement ordonné

Les programmes de bornes stochastiques prennent en entrée une matrice en format ligne ou colonne et donnent en sortie une matrice stochastique ou, si la matrice bornante est de classe C généralisée, une distribution stationnaire ou transitoire.

8.1 Algorithme de Vincent

L'algorithme de Vincent [1] est implémenté ici en matrice pleine et en mémoire. Le but est surtout de disposer d'un programme pour faire des tests et des exemples de petite taille. L'algorithme calcule la plus petite matrice borne supérieure monotone de la matrice en entrée et la plus grande matrice borne inférieure.

Le fichier "Vincent.h" contient une initialisation de la constante "epsNull" qui est le plus petit flottant accepté dans la matrice. Toutes les valeurs plus petites seront considérées égales à 0.

La syntaxe est "Vincent -f model Suffixe". Le suffixe doit commencer par R. Le programme utilise "model.sz" et "model.Suffixe". Il crèe quatre fichiers: deux fichiers "model.V.U.sz" qui contient le nombre de sommets, d'arcs et de composantes pour le modèle bornant sup, et "model.V.U.Rii" qui contient la matrice borne supérieure et enfin deux autres fichiers de nom "model.V.L.sz" et "model.V.L.Rdi" pour la matrice borne inférieure du modèle. Attention le fichier contenant la borne inférieure est codée en "Rdi" puisqu'on commence par la dernière ligne de la matrice.

```
Fichier Vincent.h
#define DEBUG 0
#define epsNull 1e-12
```

Il existe également une version optimisée de l'algorithme de Vincent pour minimiser le nombre d'opérations et la place mémoire grâce à une structure creuse ne représentant effectivement que les indices ou les sommes partielles changent. Sur cette structure de données, ceci permet de representer l'algorithme de Vincent comme la fusion de deux listes. Cette version ne calcule que la borne supérieure et n'admet comme entrée que des matrice en format "Rix". Qui plus est, il est préférable que la matrice soit en format "Rii" pour minimiser le nombre d'opérations en mémoire. La syntaxe est "RowVincent -f model". Le programme utilise "model.sz" et "model.Rix". Il utilise "Vincent.h" et crèe les deux fichiers de description de la borne supérieure : "model.V.U.sz" et "model.V.U.Rii".

8.2 IMSUB

L'algorithme IMSUB (Irreducible Monotonne Stochastic Upper Bound) est une variante de l'algorithme de Vincent [1] ayant pour but de préserver l'irréductibilité de la matrice initiale. En effet, l'algorithme de Vincent peut construire une borne réductible pour une matrice irréductible. L'algorithme IMSUB [11] empèche la destruction des transitions en autorisant des transferts de probabilités interdits par l'algorithme de Vincent.

L'amplitude de ces transferts est contrôlé par une constante "EPSILON" qui doit être entre 0 et 1. Si cette constante est nulle, on retrouve l'algorithme de Vincent. Si l'argument est grand, on perturbe beaucoup la matrice. Si il est trop petit, il y a un risque de construire des termes très petits et de provoquer des instabilités numériques. Comme pour l'algorithme de Vincent, on élimine les valeurs trop petites grâce à un seuil "epsNull".

La matrice en entrée est en format Rii. La matrice de sortie est une matrice en format ligne croissante et de même taille. On a donc, a-priori, rien gagné pour l'évaluation.

La syntaxe est "BandIMSUB -f model". Le programme utilise "model.sz" et "model.Rii". Il crèe deux fichiers "model.I.U.sz" qui contient le nombre de sommets, d'arcs et de composantes pour le modèle bornant, et "model.I.U.Rii" qui contient la matrice borne.

```
Fichier IMSUB.h
#define EPSILON 1e-6
#define epsNull 1e-12
```

8.3 LIMSUB, LMSUB et LL

LIMSUB permet de construire une borne fortement agrégeable [11], ce qui permet de réduire la taille de la matrice. La partition retenue est donnée par l'application de renumérotation et les blocs qui y sont définis. LIMSUB lit les fichiers de description d'un modèle renuméroté (les fichiers "model-suffix.bk", "model-suffix.Rii" et "model-suffix.sz") et produit en sortie un fichier de la matrice agrégée selon la partition proposée. La matrice en sortie a le type Rii, c'est-à-dire un rangement par ligne en débutant par la ligne de plus petit indice. Le nom des fichiers en sortie est passé en deuxime argument et sert à générer les fichiers ".sz" et ".Rii". On ne peut pas construire automatiquement les fichiers ".cd" . La syntaxe est (après avoir changé les constantes dans LimsubRow.h et recompilé le code) :

```
limsubrow model-in model-out
```

L'implémentation de LIMSUB proposée permet de faire tourner plusieurs algorithmes en changeant les constantes définies dans LimsubRow.h.

- LMSUB ou algorithme de Truffet pour la lumpabilité ordinaire ou l'agrégation forte. On n'impose pas que la matrice soit monotone. Ceci permet en particulier de traiter les problèmes de temps d'absorbtion [8]. Il faut initialiser MON à 1 et IRD à 0.

- LIMSUB : on impose un pattern qui force l'irréducitibilité [11]. Ce pattern implique de ne pas détruire les transitions du triangle supérieur et d'imposer une sous diagonale dont tous les élément sont non nuls. Il faut initialiser MON à 1 et IRD à 1.

- LL : on n'impose pas la monotonie. La constante MON est initialisée à 0.

Quand on impose l'irréductibilité, il est parfois nécéssaire de bouger une faible partie de la probabilité résiduelle. Cette fraction est fixée par le paramètre EPSILON. Si on ne peut pas bouger cette fraction, on peut stopper l'algorithme ou continuer.

Le fichier LIMSUB.h contient les initialisations des constantes citées précédemment.

```
#define EPSILON 1e-6
#define epsNull 1e-12
//monotonicity constraints (MON = 1) or not (MON = 0)
#define MON 1
//irreducible bound (IRD = 1) or not (IRD = 0)
#define IRD 0
//if the wanted bound cannot be computed the algorithm STOPs (if STOP = 1) or not (STOP = 0)
#define STOP 1
```

8.4 Méthode de Truffet pour les CMC

L'algorithme de Truffet est implémenté ici par ligne. Chaque ligne est écrite sur disque après son calcul. ll y a donc peu d'utilisation de la mémoire. La méthode de Truffet prend en entrèe une matrice carrée sous-stochastique et retourne deux matrices carrées stochastiques. La borne sup est formée en ajoutant a la dernière colonne les probabilités manquantes. Pour la borne inférieure, elles sont ajoutées à la première colonne. Le fichier "Truffet.h" contient une initialisation de la constante "epsNull" qui est le plus petit flottant accepté dans la matrice. Toutes les valeurs plus petites seront considérées égales à 0.

La syntaxe est "Truffet -f model Suffixe". Le suffixe est en général ".NW" quand on applique l'algorithme à une matrice issue d'une génération partielle. Le programme utilise "model.Suffixe.sz" et "model.Suffixe.Rii". Les fichiers de sortie décrivent les matrices bornant le complément stochastique du modèle. Le programme crèe quatre fichiers "model.SA.T.U.sz" qui contient le nombre de sommets, d'arcs et de composantes pour le modèle bornant sup du complément stochastique, "model.SA.T.U.Rii" qui contient la matrice borne, et les fichiers "model.SA.T.L.sz" et "model.SA.T.L.Rii" qui contiennent les informations pour la borne inférieure.

```
Fichier Vincent.h
#define epsNull 1e-12
#define DEBUG 0
```

8.5 Algorithme DPY pour les CMC

L'algorithme de Dayar, Pekergin et Younes est implémenté ici par ligne. Chaque ligne est écrite sur disque après son calcul. La méthode prend en entrèe deux matrices : une matrice carrée sous-stochastique (typiquement le bloc NW dans une CMC) et une matrice sous stochastique (typiquement le bloc SW dans une CMC). Elle retourne une matrice carrée stochastique qui est la borne supérieure du complément stochastique. Les deux matrices en entrée doivent avoir le même nombre de colonne. De plus la matrice associée au suffixe 1 doit être carrée.

Le fichier "DPY.h" contient une initialisation de la constante "epsNull" qui est le plus petit flottant accepté dans la matrice. Toutes les valeurs plus petites seront considérées égales à 0.

La syntaxe est "DPY -f model Suffixe1 Suffixe2". Le premier suffixe est en général ".NW" quand on applique l'algorithme à une matrice issue d'une génération partielle et le second ".SW". Le programme utilise "model.Suffixe1.sz", "model.Suffixe1.Rii", "model.Suffixe2.sz", "model.Suffixe2.Rii". Les fichiers de sortie décrivent la matrice borne supérieure du complément stochastique du modèle calculé par DPY. Le programme crèe deux fichiers "model.SA.DPY.U.sz" qui contient le nombre de sommets, d'arcs et de composantes pour le modèle bornant sup du complément stochastique, "model.SA.DPY.U.Rii" qui contient la matrice borne.

```
Fichier DPY.h
#define epsNull 1e-12
#define DEBUG 0
```

8.6 Algorithmes FPY07

Les algorithmes reposent sur des techniques de visite de graphe: BFS (visite en largeur) et SP (plus court chemin). Les deux programmes emploient les blocs NW et NE en format ligne (model.NW.Rii et model.NE.Rii) et des bornes inférieures par élément des blocs SE et SW (model.SE.EL.Rii et model.SW.EL.Rii). La syntaxe est "SP -f model ou BFS -f model". Les suffixes sont dans l'ordre ".NW", ".NE", "SW.EL", ".SE.EL". Le programme utilise "model.Suffixe1.sz", "model.Suffixe1.Rii", "model.Suffixe2.sz", etc...

Les fichiers de sortie décrivent la matrice borne supérieure du complément stochastique du modèle. Les deux programmes crèent deux fichiers "model.SA.BFS.U.sz" (ou model.SA.SP.U.sz) qui contient le nombre de sommets, d'arcs et de composantes pour le modèle de borne supérieure du complément stochastique, "model.SA.BFS.U.Rii" ou "model.SA.SP.U.Rii" qui contient la matrice borne.

```
Fichier BFS.h
#define epsNull 1e-12
#define DEBUG 0
#define Depth 5
```

8.7 Algorithmes BDF10

Busic, Djafri et Fourneau ont présentés dans [7] plusieurs algorithmes. Seuls quatre d'entre eux sont implémentés. Ils permettent de calculer une borne supérieure du complement stochastique du bloc NW lorsqu'on connait en plus du bloc NW :

- le bloc SW et le bloc NE pour l'algorithme 1.

  L'algorithme consiste à faire $DPY(NW + NE * SW)$.

- le bloc NE et une borne inférieure par élément des blocs SW et SE pour l'algorithme 2,

  L'algorithme consiste à faire $NW + NE * (I + SE.EL).SW.EL + Truffet(Residu)$.

- les blocs NE, SW et SE (sans passer par le calcul de l'inverse de (Id-SE)) pour les algorithmes 3 et 4.

  L'algorithme 3 consiste à faire $DPY(NW + NE * (I + SE) * SW)$.

  L'algorithme 4 consiste à faire $DPY(NW + NE * (I + Z) * SW)$ avec $Z = \sum_{i=1}^{k} SE^k$ et k est un paramètre de l'algorithme. On voit bien que prendre $k = 0$ consiste à faire l'algorithme 1 et que prendre $k = 1$ revient à l'algorithme 3. Ils sont cependant séparés pour des raisons de performance, l'algortihme 4 pouvant effectuer de nombreuses entrees-sorties.

8.8 Classe C Généralisé et décomposition de rang fabile

Ce programme est sensiblement différent. On calcule une matrice bornante de classes C généralisé et on stocke le descripteur ("fichier model.dscC") qui donne une description compacte, grâce à laquelle on peut faire des calculs plus facilement. On peut se référer à [3] pour les propriétés des matrices de classe C et leur distribution stationnaire, à [4] pour les transitoires et à [2] pour une première généralisation (la classe G) et à [6] pour une généralisation algébrique plus large : les matrices de rang faible (matrice de type LR).

Deux algorithmes de projection sur une matrice de classe LR peuvent être trouvés dans [6]. Leur syntaxe est la même :

```
LR4UPBOUND -f model k
```

et

```
LR5UPBOUND -f model k
```

où k donne la profondeur de décomposition de la matrice. Les deux méthodes utilisent les matrices d'entrée en format ".Rii". La méthode LR5UPBOUND suppose également que la borne sup de la matrice initiale a déjà été calculée par l'algorithme de Vincent et est présente dans le dossier (dans les fichiers "model.V.U.Rii" et "model.V.U.sz").

En utilisant la commande "LR4UPBOUND -f model 1" on obtient une matrice de classe G mais dans un fichier de type ".LR". Il faudrait un utilitaire pour convertir de LR en DscC et réciproquement.

8.9 Algorithmes sur un ensemble de matrices

Ces algorithmes utilisent en entrées deux matrices qui sont respectivement une borne inférieure par élément (disons L) et une borne supérieure par élément (disons U). L'ensemble comprend toutes les matrices stochastiques comprises, au sens de la comparaison entre éléments, entre ces deux matrices. Les deux matrices doivent être cohérentes :

1. leurs tailles doivent être égales,
2. $0 \leq L(i,j) \leq U(i,j) \leq 1$ pour tout i et j,
3. $\sum_j L(i,j) \leq 1$ pour tout i,
4. $\sum_j U(i,j) \geq 1$ pour tout i,

Ces conditions ne sont pas vérifiées par les algorithmes de calcul. Seule la première est testée. On vérifie aussi que le nombre d'éléments non nuls de U est plus grand que celui de L. En ce qui concerne la génération de ces deux matrices, il faut faire en sorte que les états soient dans le même ordre, et donc qu'ils aient obtenu le même numéro lors de la création des matrices définissant l'ensemble. Sinon il faut renuméroter. Les données sont en format Rii. Les matrices définissant l'ensemble se nomment "model.EL.sz" et "model.EL.Rii" pour la borne inférieure par élément et "model.EU.sz" et "model.EU.Rii" pour la borne sup.

8.9.1 Algorithme de Haddad et Moreaux

Cet algorithme calcule une borne inférieure absorbante. Le point d'absorbtion de la chaîne est le dernier état.

8.9.2 Algorithme de Haddad, Moreaux et Busic

Cet algorithme calcule une borne supérieure stochastique et une borne inférieure stochastique (voir thèse de Busic [5]). Les bornes sont au sens "st" . On ne vérifie pas si la matrice obtenue est irréductible. Les bornes ne sont pas monotones. La syntaxe est "HMB_LOW -f model" pour le calcul de la borne inférieure. On vérifie l'existence de "model.EL.sz", "model.EL.Rii" , "model.EU.sz" et "model.EU.Rii" et on crèe "model.HM.L.sz" et "model.HM.L.Rii". Le programme de borne supérieure (HMB_UP) construit les matrices "model.HM.U.sz" et "model.HM.U.Rii" à partir des mêmes données. HMB_2UP est une deuxième implémentation avec les mêmes interfaces.

Attention tous ces algorithmes retourne une matrice qui a le nombre d'éléments non nuls de la matrice U. Il peuvent donc écrire des éléments nuls qui sont laissés dans la matrice pour bien montrer que l'élément de la matrice bornante au sens "st" est égal à 0. De même les éléments de module très petits sont laissés dans la matrice. Il faut au besoin rajouter un filtrage pour éliminer les éléments négligeables et les zéros afin d'avoir une matrice plus creuse.

8.10 Algorithmes combinant bornes stochastiques et algorithme Nabla

L'algorithme présenté dans [13] procède comme suit: il construit une borne supérieure montone st en faisant en sorte que toutes les entrées de la colonne i soient positives. On peut alors résoudre rapidement par les algorithmes Nabla (INABLAL et INABLAU).

Le programme "BandFQNabla.c" impose que toutes les entrées de la colonne i (passé en argument) soit plus grandes qu'un seuil également passé en argument. Dans le cas d'impossibilité de construction, un message d'erreur est envoyé et la méthode stoppe. La syntaxe est "BandFQNabla -f model 3 0.01" pour construire une borne sup monotone pour la matrice "model" dont toutes les entrées de la colonne 3 sont supérieures ou égales à 0.01. La représentation interne repose sur un vecteur bande plein pour la ligne courante.

Il existe un autre programme faisant la même opération mais utilisant une matrice creuse. On n'a pas comparé la vitesse des deux versions. La syntaxe est : "ColonneVincent -f model 3 0.01"

Chapter 9

Génération partielle et chaine de Markov censorée

Pour effectuer une génération partielle de la chaîne, il faut faire un filtrage des états. Il n'y a pas de filtrage sur les transitions.

9.1 Etats Tabous et Etats Initiaux

La génération peut avoir une taille maximale, ou des états tabous. Enfin on peut imposer la visite de certains états quite à ne pas être sûr de la connexité de la chaîne générée.

La taille maximale de l'espace des états est nommée "MaximalEtat". C'est une constance initialisée dans "const.h".

On rappelle qu'un état censurée est un état vu (et donc échantillonné) lors de l'algorithme de visite. Un état tabou bloque la génération de nouveaux états censorés. Ses successeurs ne sont pas insérés dans la liste de visite. En conséquence, en utilisant des états tabous dans l'algorithme de visite on peut terminer l'algorithme sans avoir atteint le nombre maximal d'état.

Les états obligatoirement visités sont mis en tête de l'algorithme de visite en largeur. Ils sont donc tous visités dès le début de l'algorithme. Les états obligatoirement visités et les états tabous sont spécifiés dans le fichier "const.h" et "fun????.c" (le nom dépend de la méthode de génération) comme suit. On ajoute dans le fichier "const.h" trois constantes supplémentaires: MaximalEtat, NOblig, NTabou.

- La première constante sert à donner la taille maximale de l'espace des états.
- NOblig est le nombre d'états obligatoires dans le modèle qui seront ensuite précisés dans le fichier "fun????.c".
- Enfin NTabou est la taille de l'espace des états tabou qui seront, eux aussi, listés dans le fichier "fun????.c".

Dans le fichier "fun???.c", les états obligatoires sont les états de démarrage de l'algorithme de visite. Ils sont donc logiquement listés dans la fonction InitOblig qui modifie la variable globale "TabOblig". Cette variable est un vecteur d'entier. TabOblig[i] est le numéro interne du i-ème état obligatoire obtenu par la conversion en numéro d'un état.

Les états interdits sont listés dans la fonction EtatTabou qui utilise la variable globale "TabTabou". Cette variable est un tableau à une dimension. TabTabou[i] est le numéro interne du i-ème état tabou obtenu par la conversion en numéro d'un état. Un exemple de construction du tableau des états tabou est donné ci-après. On y définit l'état $(5, 1)$ comme le troisième état tabou.

```
E[0]=5;
E[1]=1;
k=Etat2Int(E);
TabTabou[2]=k-1;
```

Il n'est pas impossible de déclarer un état comme tabou et censoré mais cela donne une chaine absorbante.

La fonction EtatInitial qui est utilisée pour la méthode de génération exhaustive n'existe plus pour la génération avec filtrage.

Les algorithmes procèdent comme suit:

1. Faire une visite BFS à partir des états initiaux
2. Si l'état actuellement visité est Tabou, le mettre dans une structure annexe et ne pas mettre ses successeurs dans la file des sommets à visiter en BFS
3. Sinon, mettre les successeurs non déjà visités dans la file de visite BFS
4. Arrèter la visite BFS lorsque la file est vide
5. La suite dépend précisément de la méthode et s'appuie sur la structure des états mis de coté.

9.2 Les méthodes de visites

Il y a plusieurs programmes différents pour générer la matrice.

- La première méthode (i.e. Gener2Block) ne génère que les blocs NE et NW et n'utilise que la fonction de transition forward (et donc les deux fonctions qui sont déjà décrites dans la génération exhaustive). Le fichier de description s'appelle "fun2Blocks.c".
- La méthode GenerBFS ne demande que la fonction forward utilisée déjà dans la méthode de génération exhaustive. Mais en conséquence, on ne génère pas toutes les transitions dans les blocs Sud-Ouest et Sud-Est car on n'explore que partiellement (ou pas du tout) les voisins des états non visibles. On obtient alors une borne inférieure par élément des blocs SW et SE.

  L'algorithme procède par niveau de visite en largeur. Le nombre de niveau est donné par une constante MaxLevelBFS initialisée dans "funBFS.c". Par construction, les derniers états visités sont placés dans les blocs SW et SE mais ils sont supposés de degré sortant nul puisqu'on ne génère pas leurs successeurs.

  Cet algorithme va construire pour le bloc :

  - Nord Ouest, le fichier "modele.NW.cd", "modele.NW.sz", pour la description de la matrice et des codages des états visibles. Les transitions sont dans "modele.NW.Rii"
  - Nord Est, les fichiers "modele.NE.sz", "modele.NE.cd" et "modele.NE.Rii" pour la description de la matrice, du codage des états et des transitions.
  - Sud Ouest, les fichiers "modele.SW.EL.sz", "modele.SW.EL.cd" et "modele.SW.EL.Rii" pour la description de la matrice, du codage des états et des transitions. Le codage des noms de fichier rapelle qu'il s'agit d'une borne inférieure par élément car on ne génère pas toutes les transitions.
  - Sud Est, les fichiers "modele.SE.EL.sz", "modele.SE.EL.cd" et "modele.SW.EL.Rii" pour la description de la matrice, du codage des états et des transitions. Le codage des noms de fichier rapelle qu'il s'agit d'une borne inférieure par élément car on ne génère pas toutes les transitions.

- La méthode (Gener3Block) suppose que l'on connaisse la fonction de transition dans chaque sens (forward et backward). Il faut donc dans le fichier "fun3Blocks.c" fournir 4 fonctions :
  - Probabilite(indexevt, E) retourne la probabilité d'occurence de l'événement numéro "indexevt" quand on quitte l'état E.
  - Equation(E, indexevt, F, R) met dans F l'état obtenu quand on applique l'événement "indexevt" à l'état E.
  - InverseEquation(E, indexevt, F) met dans F l'état nécéssaire pour arriver dans E en appliquant l'événement "indexevt" à l'état E.
  - InverseProbabilite(indexevt, E) retourne la probabilité d'occurence de l'événement numéro "indexevt" quand on arrive dans l'état E.

  Les deux premières fonctions sont identiques à celles qui sont décrites dans le cas d'une génération exhaustive. La troisième est l'inverse de la fonction de transition. La dernière donne les probabilités connaissant l'état d'arrivée.

  Cet algorithme va construire pour le bloc :

  - Nord Ouest, le fichier "modele.NW.cd", "modele.NW.sz", pour la description de la matrice et des codages des états visibles. Les transitions sont dans "modele.NW.Rii"
  - Nord Est, les fichiers "modele.NE.sz", "modele.NE.cd" et "modele.NE.Rii" pour la description de la matrice, du codage des états et des transitions.
  - Sud Ouest les fichiers "modele.SW.sz", "modele.SW.cd" et "modele.SW.Cuu" pour la description de la matrice, du codage des états et des transitions. Le codage du nom de fichier des transitions rapelle qu'il a été généré par colonne et dans un ordre quelconque.

Pour résumer, voici les constantes et fonctions pour chaque méthode :

Méthode	Constantes	Fonctions
Gener2Block.c	NTabou, NOblig, MaximalEtat, NEt, NbEvtsPossibles, Epsilon1, Epsilon2	EtatTabou, EtatOblig, Probabilite, Equation, InitEtendue, InitParticuliere
Gener3Block.c	NTabou, NOblig, MaximalEtat, NEt, NbEvtsPossibles, Epsilon1, Epsilon2	EtatTabou, EtatOblig, Probabilite, Equation, InverseProbabilite, InverseEquation, InitEtendue, InitParticuliere
GenerBFS.c	NOblig, MaximalEtat, MaxLevelBFS, NbEvtsPossibles, Epsilon1, Epsilon2, NEt	EtatOblig, Probabilite, Equation, InitEtendue, InitParticuliere

9.3 Créer le programme de génération

Il reste à créer un nouvel exécutable en compilant les programmes de génération :

- "Gener2Blocks.c" , "Gener2Blocks.c", "GenerBFS.c" pour la génération de la chaîne Censorée par les diverses méthodes.

Le programme récupère en entrée le nom du modèle avec l'option -f.

Chapter 10

Autres méthodes de génération

10.1 Générer deux matrices pour encadrer un ensemble de DTMC

La méthode GenerSet permet de construire deux matrices L et U pour définir un ensemble de matrices stochastiques M telles que $L \leq M \leq U$ où la comparaison se fait élément par élément. Le fichier "funSet.c" remplace le fichier "fun.c". Les functions "EtatInitial", "InitEtendue", "Equation" sont les mêmes. La fonction "Probabilite" est remplacée par les deux fonctions "InfProbabilite(j,E)" et "SupProbabilite(j,E)" qui retournent respectivement des bornes inférieures et supérieures de la probabilité de transition. Dans les deux as, les arguments sont le numéro de l'événement et l'état où s'applique l'événement. La méthode crée les fichiers .sz et .cd pour la matrice U mais surtout deux fichiers pour les transitions: "model.EU.Rii" pour la matrice U et "model.EL.Rii" pour la matrice L.

La syntaxe est après recompilation:

```
GenerSet -f model
```

10.2 Générer un automate

On utilise le même algorithme de génération pour construire une représentation sous forme de graphe d'un automate. Le format de sortie est codé en "trivial graph format". Il faut comme pour tous les autres programmes de génération, donner :

- un état initial,
- le nombre d'événements,
- le nombre de composantes de l'espace des états,
- le min et le max pour chaque composante,
- une fonction donnant le successeur pour chaque événement,

De plus, il faut indiquer le symbole écrit sur la transition pour chaque événement. Pour cela, on utilise la fonction InitLabel qui remplit le tableau global "Label" indexé par les événements et qui contient le caractère décrivant l'événement. Les états portent une étiquette qui est leur vecteur de numéros dans l'espace des composantes. Les arcs portent une étiquette qui résulte de la concaténation des étiquettes décrivant les événements.

Il reste à créer un nouvel exécutable en compilant le programme de génération :

- "GenerAutomata.c" pour la génération exhaustive des états et des transitions labelisés.

Pour résumer, voici les constantes et fonctions :

Méthode	Constantes	Fonctions
GenerAutomata.c	NEt, NbEvtsPossibles,	Equation, EtatInitial, InitLabel, InitEtendue, InitParticuliere

Et voici le fichier "tgf" généré par l'outil, les noms des états sont les tailles des files, les codages des transitions sont "a" ou "A" pour arrivée, "d" ou "D" pour départ, "r" pour routage et "l" pour boucle.

```
0  (0,0)
1  (1,0)
2  (0,1)
3  (2,0)
4  (1,1)
5  (0,2)
6  (3,0)
7  (2,1)
8  (1,2)
9  (0,3)
10  (3,1)
11  (2,2)
12  (1,3)
13  (3,2)
14  (2,3)
15  (3,3)
#
0          0 (l,r,d,D)
0          1 (a)
0          2 (A)
1          0 (d)
1          1 (l,D)
1          2 (r)
1          3 (a)
1          4 (A)
2          0 (D)
2          2 (l,r,d)
2          4 (a)
2          5 (A)
3          1 (d)
3          3 (l,D)
3          4 (r)
3          6 (a)
3          7 (A)
4          1 (D)
4          2 (d)
4          4 (l)
4          5 (r)
4          7 (a)
4          8 (A)
5          2 (D)
5          5 (l,r,d)
```

```
 5          8 (a)
 5          9 (A)
 6          3 (d)
 6          6 (l,a,D)
 6          7 (r)
 6         10 (A)
 7          3 (D)
 7          4 (d)
 7          7 (l)
 7          8 (r)
 7         10 (a)
 7         11 (A)
 8          4 (D)
 8          5 (d)
 8          8 (l)
 8          9 (r)
 8         11 (a)
 8         12 (A)
 9          5 (D)
 9          9 (l,A,r,d)
 9         12 (a)
10          6 (D)
10          7 (d)
10         10 (l,a)
10         11 (r)
10         13 (A)
11          7 (D)
11          8 (d)
11         11 (l)
11         12 (r)
11         13 (a)
11         14 (A)
12          8 (D)
12          9 (r,d)
12         12 (l,A)
12         14 (a)
13         10 (D)
13         11 (d)
13         13 (l,a)
13         14 (r)
13         15 (A)
14         11 (D)
14         12 (r,d)
14         14 (l,A)
14         15 (a)
15         13 (D)
15         14 (r,d)
15         15 (l,a,A)
```

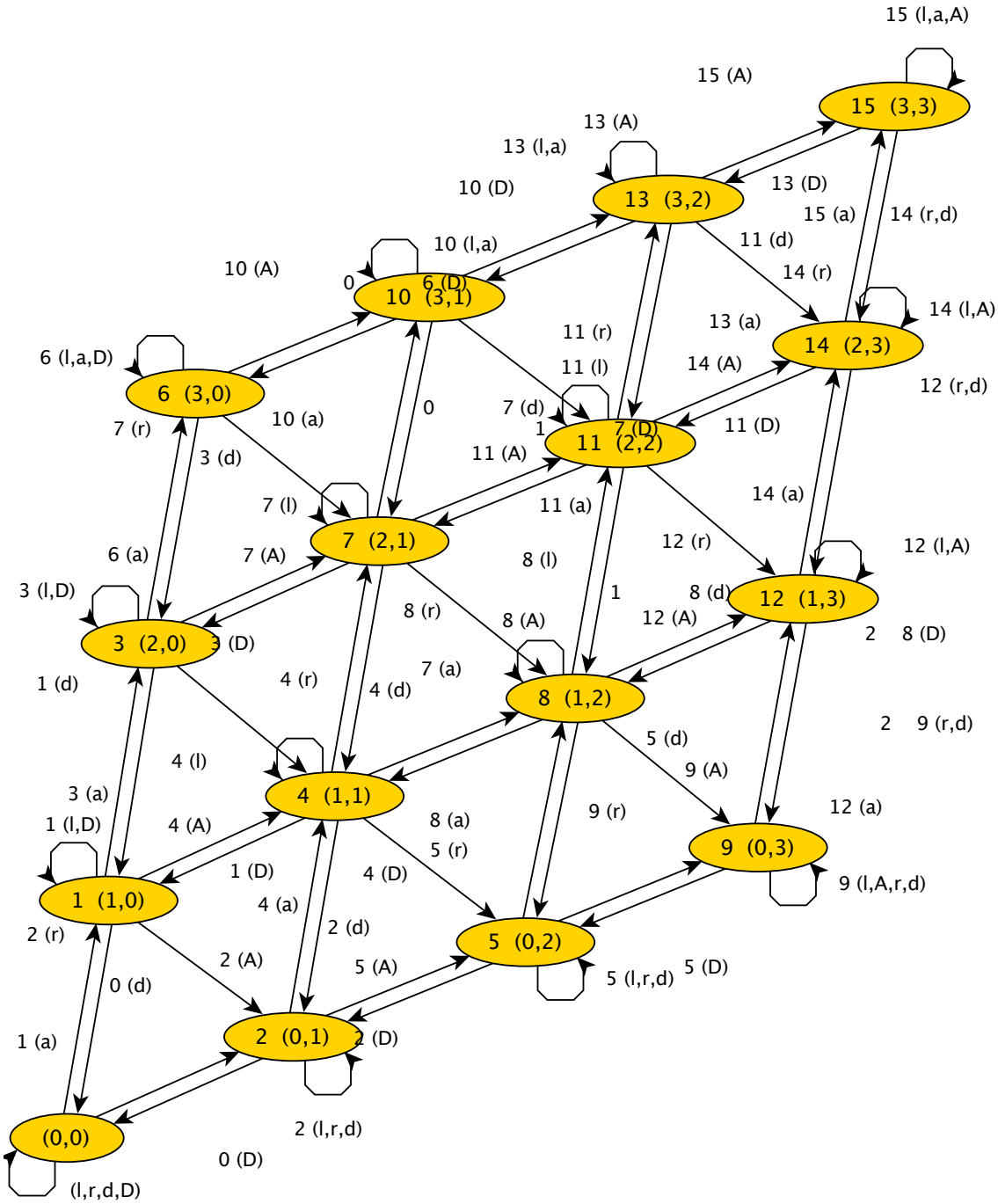

Figure 10.1: Automate pour un réseau de 2 files en tandem avec pertes sur file pleine.

Chapter 11

Divers Algorithmes

11.1 Générer des tables pour la méthode d'Alias

Chapter 12

Formats de matrice ou de fichiers

Les matrices sont stockées en format creux et il faut plusieurs fichiers pour décrire une chaîne. Les différents fichiers sont reconnus grâce à leur suffixe et ont des formats des fichiers différents.

suffixe	contenu
.sz	taille de la matrice
.cd	codage des états
.Rxy	données par ligne,
.Cxy	données par colonne
.dscC	descripteur d'une matrice de classe C

Et les "y" et "z" permettend de coder i pour increasing, d pour decreasing, u pour undefined. Par exemple, Rid veut dire "stockage par lignes, dans l'ordre croissant des lignes et l'ordre décroissant à l'interieur de chaque ligne".

Les algorithmes de génération partielle créent des matrices qui ne sont pas toutes carrées. Les extensions sont :

suffixe	contenu
.NW	bloc Nord Ouest
.SW	bloc Sud Ouest
.NE	bloc Nord Est
.SE	bloc Sud Est

Et on ajoute les extensions générales (.sz, .Rxx) pour décrire la matrice. Ainsi "model.NW.sz" est le fichier décrivant la taille du bloc Nord Ouest.

De même, on définit un ensemble de matrices stochastiques comprises par élément entre deux matrices bornantes par ces deux matrices borne inférieure et borne supérieure. Les deux matrices sont positives. Comme il s'agit de bornes par élément, elles ne sont pas en général stochastiques. La matrice borne inférieure par élément est sous-stochastique, la matrice borne supérieure est telle que la somme des éléments par ligne est supérieure ou égale à 1. Les extensions sont:

suffixe	contenu
.EU	borne supérieure par élément
.EL	borne inférieure par élément

Et on ajoute les extensions générales (.sz, .Rxx) pour décrire la matrice. Ainsi "model.EU.sz" est le fichier décrivant la taille de la matrice borne inférieure par élément.

Les algorithmes de calcul de bornes stochastiques ajoutent des extensions au nom du modèle et les extensions de définition des matrices.

suffixe	contenu
.U	borne supérieure stochastique
.EU	borne supérieure par élément
.L	borne inférieure stochastique
.EL	borne inférieure par élément
.V	résultat de l'algorithme de Vincent
.I	résultat de l'algorithme IMSUB
.LI	résultat de l'algorithme LIMSUB
.LM	résultat de l'algorithme LMSUB
.LL	résultat de l'algorithme LL
.T	résultat de l'algorithme de Truffet sur complément stochastique
.DPY	résultat de l'algorithme de Dayar, Pekergin et Younes sur complément stochastique
.HM	résultat de l'algorithme de Hadad et Moreaux sur un ensembre de matrices sous-stochastiques
.HMB	résultat de l'algorithme de Hadad et Moreaux, modifié par Busic, sur un ensembre de matrices stochastiques

Les suffixes s'ajoutent: ".DPY.U.sz" désigne le fichier taille de la borne sup calculée par DPY; "V.L.Rii" est le fichier des transitions de la matrice borne inférieure stochastique calculée par l'algorithme de Vincent.

Enfin les algorithmes de calcul de probabilités et de récompenses ajoutent leur propres extensions au nom donné en entrée.

suffixe	contenu
.pi	un vecteur de distribution
.pi_d	idem mais à la date d
.rwdI	récompense sur Indice
.rwdC	récompense sur les Composantes
.bk	description des blocs pour une agrégation
.perm	permutation des états pour redonner la matrice
.eps	seuil pour décomposer une matrice

12.1 Format du fichier .sz

Il se compose pour les matrices carrées de trois lignes contenant chacune un entier : le nombre d'éléments non nuls de la matrice, puis le nombre d'états atteignables depuis l'état initial choisi et enfin le nombre de composantes du vecteur des états. La lecture de ce fichier permet de faire les dimensionnement mémoire dès le démarrage et de stopper immédiatement en cas de problème.

Dans le cas de génération de blocs d'une DTMC, les matrices Sud Ouest et Nord Est sont rectangulaires, le fichier .sz a une syntaxe légèrement différente. La seconde ligne contient le nombre de ligne et la quatrième ligne contient, dans cet ordre, le nombre de colonne, l'indice minimal de ligne et l'indice minimal de colonne.

12.2 Format du fichier model.cd

Un fichier model.cd sert à conserver les codages des états lors de leur génération. Chaque ligne décrit un état. Elle commence par le numero de l'état lors de la génération (le premier état est numéroté 0) et se poursuit par la valeur de chaque composante dans l'ordre des composantes. Les états ne sont pas toujours stockés dans l'ordre de numéro croissant: en particulier lorsqu'on génère des blocs non carrés. En effet on stocke la descriptions des indices de lignes et de colonne en ne respectant pas un ordre précis.

12.3 Format des fichiers .Rxx et .Cxx

Chaque ligne décrit les transitions sortantes (pour les Rxx) ou entrantes (pour les Rxx) pour un état. Plus précisement on trouve sur chaque ligne, et dans cet ordre:

1. le numéro du sommet,
2. le degré entrant (pour les fichiers .Cxx) ou sortant (pour les fichiers .Rxx) du sommet (un entier),
3. puis pour chaque transition, sa probabilité (un réel) suivi de son autre extrémité (un entier).

Dans un fichier de suffixe .Rii les numéros de ligne sont croissant. De plus dans chaque ligne, les transitions sont citées par ordre croissant de destination. A l'inverse, les fichiers de suffixe .Rdd sont par ordre décroissant, tant pour les sommets que pour les listes de transition. Enfin, dans un fichier "model.Ruu", on ne connait pas l'ordre.

12.4 Format du fichier .pi, .pi_0 et .pi_t

Il s'agit d'une distribution de probabilités. On trouve sur chaque ligne une probabilité. Les états sont dans l'ordre de la génération. Un fichier avec le suffixe .pi_t est la distribution à la date t et .pi_0 est une distribution initiale.

12.5 Format du fichier .dscC et .LR

Ces formats sont utilisés pour décrire les matrices ayant une décomposition de rang faible : en particulier les matrices de classe C généralisé qui sont de rang au plus 2. Il s'agit dans les deux cas d'un fichier texte, avec en premire ligne le rang de la décomposition et la taille des vecteurs.

Dans le cas d'un descripteur .dscC, il y a alors 2 lignes, chaque ligne contenant deux noms de fichiers. La seconde ligne contient les noms des fichiers stockant les vecteurs v et e, la ligne suivante stocke les noms des fichiers pour r et c. Les fichiers doivent se trouver dans le dossier courant. Voici un exemple de descripteur ".dscC"

```
2 n
model-e model-v
model-r model-c
```

Le vecteur dans model-e doit voir toutes les entrées à 1. Le vecteur dans model-v est un vecteur de probabilités. Le vecteur dans model-c est de somme égale à 0.

Dans le cas d'une matrice .LR, cette ligne est suivie de $2k+1$ lignes (si k est le rang trouvé dans la première ligne. La seconde ligne contient le vecteur v dans la décomposition. Les lignes suivantes contiennent alternativement les vecteurs r_i et c_i (un vecteur complet par ligne), ce qui permet de construire la matrice :

$$M = e^T.v + \sum_{i=1}^{k} r_i^T c_i$$

Voici un exemple de fichier descripteur .LR

```
2 8
0.1 0.2 0.1 0.1 0.1 0.0 0.2 0.2
0.0 0.01 0.01 0.1 0.1 0.2 0.5 0.5
-0.05 -0.05 -0.1 0.2 0 0 0 0
0 0.5 1 0 1 0 1 0
0 0 0 0 -0.1 0.1 -0.1 0.1
```

On rappelle que le vecteur v est un vecteur positif dont la somme des composantes vaut 1, les vecteurs r_i sont positifs et les vecteurs c_i sont des vecteurs dont la somme des composantes vaut 0.

Pour une matrice de classe C (non généralisé), on a $r(j) = j - 1$. Ces familles de matrice disposent d'algorithmes spécifiques et très simples pour la résolution en transitoire et en stationnaire [3, 2].

12.6 Format des fichiers .rwdI, rwdT, rwdP, et .rwdC

Ils contiennent en format texte les résultats liés aux rćompenses. Les résultats sont précédés d'un commentaire. Pour le premier, il s'agit de calcul ne prenant pas en compte les codages des états et qui ne demandent aucune définition à l'utilisateur. Le dernier intègre des calculs spécifiés par l'utilisateur (voir le chapitre sur le sujet et les exemples).

12.7 Format des fichiers .Cpath, .Spath et .Dpath

Il s'agit de la sortie des simulateur. Chaque ligne contient une date et un état. Selon les options de générations du simulateur, l'état peut être codé par un entier unique ou par la description de l'état en clair. Dans le premier cas, l'entier peut être le numéro de visite (SimulWithCoverage) ou le numéro dans l'ordre lexicographique (SimulMarkov). Les fichiers "Dpath" correspondent à des simulations en temps discret alors que les fichiers "Cpath" sont produits par des simulateurs en temps continu et les fichiers "Spath" par des simulateurs de processus Semi-Markoviens. Le format des dates diffère donc pour les fichiers.

12.8 Format des fichiers .bk

Ces fichiers sont utilisés en entrée pour agréger des matrices. Ils sont construits par exemple par l'algorithme de Tarjan qui cherche une partition des états qui rendent la matrice lumpable.

Ces fichiers contiennent en premire ligne, le nombre de blocs et ensuite l'indice du premier élément de chaque bloc. Les matrices sont, au préalable, permutées pour que les éléments des blocs soient consécutifs.

12.9 Format des fichiers .perm

Le fichier contient deux colonnes et est trié selon la première colonne. Chaque colonne correspond à un état. La première colonne contient le numéro initial de l'état et la seconde le numéro après permutation.

12.10 Format des fichiers .part

Le fichier décrit une partition de l'ensemble des états. La première ligne contient un entier, le nombre de partition. Puis chaque ligne décrit un sous ensemble. En début de ligne, on trouve la taille du sous ensemble, puis la liste des éléments de ce sous ensemble.

12.11 Format des fichiers .epsi

Il contient juste une ligne avec la valeur de seuil utilisé par l'algorithme de construction (utilisé pour les matrices NCD, ou quasi-lumpable, ou lumpable). C'est un réel.

12.12 Format des fichiers .trace

Il contient juste une trace pour les simulateurs reposant des mesures. La premiere ligne contient deux entiers : le nombre d'elements de la trace et ensuite le nombre de mesures pour chaque éléments. Chaque ligne suivante contient un numéro d'élément (débutant à 1) et le nombre d'éléments de mesure cités en première ligne (des flottants).

Bibliography

[1] O. Abu-Amsha and J. M. Vincent. An algorithm to bound functionals on Markov chains with large state space. In *4th INFORMS Conference on Telecommunications*, Boca Raton, Floride, E.U, 1998. INFORMS.

[2] M. Ben Mamoun, A. Busic, and N. Pekergin. Generalized class C Markov chains and computation of closed-form bounding distributions. *Probability in the Engineering and Informational Science*, 21:235–260, 2007.

[3] M. Ben Mamoun and N. Pekergin. Closed-form stochastic bounds on the stationary distribution of Markov chains. *Probability in the Engineering and Informational Sciences*, 16(4):403–426, 2002.

[4] M. Ben Mamoun, N. Pekergin, and S. Younès. Class C Markov chains and transient analysis. In *Positive systems Theory and Application conference*, volume 341 of *Lecture Notes in Control and Information Sciences*, pages 177–184. Springer, 2006.

[5] A. Busic. *Comparaison Stochastique de modèles Markoviens : une approche algorithmique et ses applications en fiabilité et en évaluation de performances.* PhD thesis, Université de Versailles Saint-Quentin, 2007.

[6] A. Busic and J.-M. Fourneau M. Ben Mamoun. Stochastic bounds with a low rank decomposition. *Stochastic Models, Special Issue with selected papers from the Eighth International Conference on Matrix-Analytic Methods in Stochastic Models*, 30(4):494–520, 2014.

[7] A. Busic, H. Djafri, and J.-M. Fourneau. Stochastic bound for censored Markov chain. In *6th International Workshop on the Numerical Solution of Markov Chain, USA*, 2010.

[8] A. Busic and J.-M. Fourneau. Bounds for point and steady-state availability: An algorithmic approach based on lumpability and stochastic ordering. In Mario Bravetti, Leïla Kloul, and Gianluigi Zavattaro, editors, *Formal Techniques for Computer Systems and Business Processes, European Performance Engineering Workshop, EPEW 2005 and International Workshop on Web Services and Formal Methods, WS-FM 2005, Versailles, France, September 1-3, 2005*, volume 3670 of *Lecture Notes in Computer Science*, pages 94–108. Springer, 2005.

[9] T. Dayar, J.-M. Fourneau, and N. Pekergin. Transforming stochastic matrices for stochastic comparison with the st-order. *RAIRO Operations Research*, 37:85–97, 2003.

[10] T. Dayar, J.-M. Fourneau, N. Pekergin, and J.-M. Vincent. Polynomials of a stochastic matrix and strong stochastic bounds. In *Markov Anniversary Meeting*, pages 211–228, Charleston, 2006. Boson Books, Raleigh, North Carolina.

[11] J.-M. Fourneau, Mathieu Le Coz, and F. Quessette. Algorithms for an irreducible and lumpable strong stochastic bound. *Linear Algebra and Applications*, 386:167–185, 2004.

[12] J.-M. Fourneau and N. Pekergin. An algorithmic approach to stochastic bounds. In *Performance Evaluation of Complex Systems: Techniques and Tools, Performance 2002, Tutorial Lectures*, volume 2459 of *LNCS*, pages 64–88. Springer, 2002.

[13] J.-M. Fourneau and F. Quessette. Some improvements for the computation of the steady-state distribution of a Markov chain by monotone sequences of vectors. In *ASMTA*, Lecture Notes in Computer Science. Springer, 2012.

[14] W.K. Grassman, M.I. Taksar, and D.P. Heyman. Regenerative analysis and steady state distributions for markov chains. *Operations Research*, 33(5):1107–1116, 1985.

[15] D.P. Heyman. Further comparisons of direct methods for computing stationary distributions of Markov chains. *SIAM Journal of Agebraic Discrete Methods*, 8(2):226–232, April 1987.

[16] M. Kijima. *Markov Processes for Stochastic Modeling.* Chapman & Hall, London, UK, 1997.

[17] A. Muller and D. Stoyan. *Comparison Methods for Stochastic Models and Risks.* Wiley, New York, NY, 2002.

[18] M. Shaked and J. G. Shantikumar. *Stochastic Orders and their Applications.* Academic Press, San Diego, CA, 1994.

[19] Theordore Sheskin. Computing the fundamental matrix for a reducible Markov chain. *Mathematics magazine*, 68(5):391–398, 1995.

[20] W.J. Stewart. *Introduction to the numerical Solution of Markov Chains.* Princeton University Press, New Jersey, 1995.