Exploration randomisée de larges espaces d'états pour la vérification

Nazha Abed

THÈSE

présentée par

ABED Nazha

pour obtenir

le titre de Docteur de l'Université Joseph FOURIER- Grenoble 1

École Doctorale Mathématiques, Sciences et Technologies de lInformation
Spécialité : Mathématiques et Informatique
Équipe d'accueil : MESCAL - LIG

EXPLORATION RANDOMISÉE DE LARGES ESPACES D'ÉTATS POUR LA VÉRIFICATION

Thèse dirigée par M. Bruno GAUJAL et encadrée par M. Jean-Marc VINCENT

Date de soutenance : 16 juin 2009

Composition du jury

M. Jean-Claud	FERNANDEZ	Prsident
M. Enrico	VICARIO	Rapporteur
Mme. Nihal	PEKERGIN	Rapporteur
M. Stavros	TRIPAKIS	Examinateur
M. Bruno	GAUJAL	Examinateur
M. Jean-Marc	VINCENT	Examinateur

Remerciements

Je remercie M. Jean-Claud FERNANDEZ, qui me fait l'honneur de présider ce jury.

Je remercie M. Enrico VICARIO, Rapporteur, et Mme. Nihal PEKERGIN, Rapporteur, d'avoir bien voulu accepter la charge de rapporteur.

Je remercie M. Stavros TRIPAKIS, Examinateur, M. Bruno GAUJAL, Examinateur, et M. Jean-Marc VINCENT, Examinateur, d'avoir bien voulu juger ce travail.

Je remercie enfin S. Dublairet S.Dublairqui m'ont dirigé durant ma thèse pour leur disponibilité, leurs directives et conseils, et particulièrement leur patience et compréhension.

Table des matières

Table des figures

Liste des tableaux

Notations

Notations relatives au graphe d'état

$G(M, V_0, Succ)$	Graphe d'états
M	Ensemble des états
V_0	Ensemble des états initiaux
v_0	État initial
$Succ()$	Fonction qui retourne l'ensemble de successeurs d'un état
ϕ	Propriété à vérifier
$d(v)$	Degré de l'état v

Notations relatives aux algorithmes

A	Algorithme d'exploration
V	Ensemble de nœuds gardés en mémoire
N	Taille de la mémoire en états
n	Nombre d'étapes
k	Nombre de nœuds couvert
R	Nombre de répétitions de l'algorithme d'exploration
P	Paramètres statiques de l'algorithme
I	Informations globles exploitées par l'algorithme

Notations relatives aux critères

$\mathcal{C}_A$	Critère à maximiser pour un algorithme A
$\mathcal{C}_{A,max}$	Valeur maximale du critère $\mathcal{C}_A$
T_e	Temps d'exécution
τ_c	Taux de couverture
τ_r	Taux de redondance
$T_{A,G}(k)$	Temps moyen de couverture de k nœuds
$\underline{v}_k = (v_1, ..., v_k)$	Séquence de k nœuds distincts de G visités par A dans l'ordre
$\mathbb{P}_{A,G}(\underline{v}_k, n)$	Probabilité de couvrir $\underline{v}_k$ en n étapes
$\mathbb{P}_{A,G}(v, n)$	Probabilité d'atteindre le nœud v en n étapes
$\pi_{min}(A, G, n)$	La probabilité minimale d'atteignabilité sur G
G^N	Graphe de dimension N

Notations relatives aux formules théoriques

$\mathbb{P}(\underline{v}_k, n, a)$	Probabilité de couvrir la séquence $\underline{v}_k$ en n étapes
$\mathbb{P}^{\mathcal{R}}(\underline{v}_k, n, a)$	Terme de redondance de $\mathbb{P}(\underline{v}_k, n, a)$
$\mathbb{P}^{\mathcal{I}}(\underline{v}_k, n, a)$	Terme d'innovation de $\mathbb{P}(\underline{v}_k, n, a)$
m	Degré de l'arbre
h	Profondeur de l'arbre
K_n^j	v.a. : nombre de nœuds couverts au niveau j de l'arbre à l'étape n
$\underline{K}_n$	Vecteur de variables aléatoires $(K_n^1, ..., K_n^h)$
$\underline{k} = (k_1, ..., k_h)$	Vecteur d'entiers
$\underline{k} - 1_j$	$(k_1, ..., k_j - 1, ..., k_h)$, $1 \leq j \leq h$
$\mathbb{P}_A(\underline{K}_n = \underline{k}, a)$	Probabilité de couvrir k_j nœuds à chaque niveau j de l'arbre
$\mathbb{P}_A^{\mathcal{R}}(\underline{K}_n = \underline{k}, a)$	Terme de redondance de $\mathbb{P}_A(\underline{K}_n = \underline{k}, a)$
$\mathbb{P}_A^{\mathcal{I}}(\underline{K}_n = \underline{k}, a)$	Terme d'innovation de $\mathbb{P}_A(\underline{K}_n = \underline{k}, a)$
K_n	v.a. : le nombre de nœuds couverts à l'étape n
$\mathbb{P}_A(K_n = k, a)$	Probabilité de couvrir un nombre k nœuds à l'étape n
$T_A(\underline{k})$	Temps moyen de couverture du vecteur $\underline{k}$
$T_A(k)$	Temps moyen de couverture de k nœuds distincts
$\mathbb{P}_A(v; \underline{K}_n = \underline{k})$	Probabilité d'atteindre v en n étapes et avoir couvert le vecteur $\underline{k}$
$\mathbb{P}_A(v; K_n = k)$	Probabilité d'atteindre v en n étapes et avoir couvert k nœuds
$\mathbb{P}_A(i; \underline{K}_n = \underline{k})$	Probabilité d'atteindre un nœud se situant au niveau i de l'arbre en n étapes et avoir couvert le vecteur $\underline{k}$
$\mathbb{P}_A(i; K_n = k)$	Probabilité d'atteindre un nœud se situant au niveau i de l'arbre en n étapes et avoir couvert k nœuds
$\mathbb{P}_A(v, n)$	Probabilité d'atteindre le nœud v en n étapes
$\mathbb{P}_A(i, n)$	Probabilité d'atteindre un nœud du niveau i de l'arbre en n étapes
$\mathbb{P}_A^*(i, n)$	Probabilité d'atteindre un nœud se situant au niveau i en n étapes et avoir un nombre de nœuds couverts $k = N$
$NC_{RA}(n)$	Nombre moyen de nœuds couverts par l'algorithme RA en fonction du nombre d'étapes n
d	Dimension de la grille
L	Longueur de la grille
$C(v)$	Ensemble de successeurs (children) du nœud v
$F(v)$	Ensemble de prédécesseurs (fathers) du nœud v
$v(x_1, ..., x_d)$	Coordonnées du nœud v dans la grille
mx	Paramètre de contrôl du processus de mixage
$DF = m/h$	Facteur de densité caractéristique du graphe
$L(\underline{v}_k)$	Ensemble de feuilles dans la séquence $\underline{v}_k$
$L(\underline{v}_k)$	Ensemble de nœuds internes dans la séquence $\underline{v}_k$
$\mathbb{P}(\underline{L}_n = \underline{l}, \underline{I}_n = \underline{t})$	Probabilité de couvrir un nombre l_j de feuilles et un nombre t_j de nœuds internes à chaque niveau j de l'arbre

Chapitre 1

Introduction

1.1 Les enjeux de la vérification

La vérification est une procédure qui vise à assurer le bon fonctionnement d'un système, en général automatisé et critique, en garantissant qu'il vérifie certaines propriétés. De nos jours, les systèmes automatisés sont omniprésent : processus industriels, automobile, avionique, ferroviaire, énergie atomique... La présence de tels systèmes dans des applications critiques, couplée à leur complexité croissante, rend indispensable leur vérification de façon automatique afin de garantir la sûreté de leur fonctionnement. En plus, les contraintes économiques imposent souvent un temps de développement court, ce qui rend accru le besoin de méthodes de vérification efficaces et à coût réduit.

1.1.1 Model checking

Les méthodes formelles, incluant la méthode du "Model-Checking" [1] [58, 12, 13], ont été appliquées avec succès dans plusieurs domaines, entre autres, réseaux, protocoles de sécurité, protocoles de télécommunication et contrôle automatique. Ils permettent de produire une description, dite *spécification* du système en question, puis, à partir de la spécification, vérifier automatiquement les propriétés désirées avant d'implémenter le système. Les algorithmes de Model-Checking sont, en général, conçus pour la vérification totale du système considéré. Ils effectuent des parcours exhaustifs de tous les états du système, modélisé par un graphe, et vérifient que chaque chemin d'exécution satisfait les propriétés voulues. Les états explorés sont gardés en mémoire pour éviter des réexplorations éventuelles.

Lorsque le graphe modélisant le système est de taille raisonnable, c'est à dire qu'il peut être géré par la mémoire principale, les méthodes de model-checking sont très efficaces dans l'exploration de ce graphe et la détection des erreurs éventuelles. Cependant, la plupart des systèmes logiciels réels, possèdent des graphes d'états de très

1. Le model checking désigne une famille de techniques de vérification automatique des systèmes. Le terme anglais sera utilisé le long de la thèse car il s'agit d'une appelation universelle de ces techniques

grande taille. En effet, l'espace d'états, représentant tous les comportements possibles d'un système ou d'un protocole complexe est habituellement exponentiel en le nombre de processus le comportant. Cette évolution importante dans la taille du graphe connue sous le nom de *explosion combinatoire de la taille de l'espace d'états*, constitue encore, même après plusieurs années de travail [12], l'obstacle principal de la vérification automatique par model checking.

1.1.2 Problème de l'explosion de la taille de l'espace d'états

Plusieurs méthodes ont été développées pour pallier ce problème. On distingue deux classes : les méthodes déterministes et les méthodes randomisées. Dans la classe des méthodes déterministes, il y a principalement deux approches. La première consiste à réduire la taille de la mémoire nécessaire pour le stockage des états, citons en particulier dans cette approche, le bi-state hashing [39] et le hash compaction [63]. Ces approches gardent en mémoire des valeurs compressées des états au lieu de toute la description de l'état. Le gain en espace de stockage se fait au détriment d'un risque d'omission de certains états dans l'exploration. La deuxième approche vise à réduire l'espace d'états lui même. Ceci peut être obtenu par des techniques comme, par exemple, la réduction d'ordre partiel [53, 26, 27], qui est basée sur l'observation que l'exécution de deux événements indépendants dans un ordre ou un autre donne comme résultat le même état global, et la technique de réduction symétrique [14]. Une autre approche consiste en l'utilisation des techniques de remplacement en mémoire, appelée communément "caching" [25, 28]. Elle permet une exploration totale de l'espace d'état en ne gardant en mémoire qu'un sous ensemble des états explorés : chaque nouvel état exploré prend la place d'un état ancien dans la mémoire.

1.1.3 La vérification randomisée

Dans la classe des méthodes randomisées, la vérification est assurée par des algorithmes d'exploration aléatoires et semi aléatoires permettant de réaliser une bonne exploration de l'espace d'états du système. Une bonne exploration signifie l'atteinte et la vérification d'un grand nombre d'états du graphe sans dépasser les ressources mémoire disponibles. Les algorithmes randomisés ont vite pris place au monde informatique vu leur simplicité et rapidité. Ils ont pu dépasser les algorithmes déterministes dans diverses problèmes de recherche [45]. En effet, si le graphe d'états du système à vérifier est très grand, l'algorithme déterministe l'explore d'une façon ordonnée et garde les états explorés en mémoire. Dès épuisement de celle-ci, soit il abandonne l'exploration, soit il reste bloqueé dans un problème de swapping. Il échoue à explorer tous les états du système et en particulier à atteindre les états situés, selon l'ordre qu'il respecte, au delà de la taille mémoire disponible. Un algorithme randomisé donne à ces états une chance plus importante d'être atteints et vérifiés via une exploration mieux diffusée, c'est à dire qui ne respecte pas un ordre particulier et donne aux états une probabilité non nulle d'être atteint. Un tel algorithme utilise la mémoire disponible, et vu l'insuffisance de celle-ci, il n'y garde qu'un sous ensemble des états explorés. D'où,

la couverture totale de l'espace d'états n'est pas garantie. Néanmoins, au lieu d'abandonner l'exploration par manque de ressources et ne retourner aucune réponse quant à la satisfiabilité de la propriété en question par le système, le résultat de la vérification est donné approximativement, avec une probabilité que l'on peut contrôler.

1.2 Randomisation de la vérification

La majorité des méthodes randomisées de vérification utilisent la marche aléatoire (random walk : *RW*), comme schéma d'exploration. L'avantage principal de *RW* est qu'il ne nécessite, quasiment, aucune ressource mémoire puisqu'il ne garde que l'état courant au cours de l'exploration. Son inconvénient majeur est l'exploration redondante des mêmes états vu l'absence de toute information sur la partie déjà explorée. Pour l'optimiser, différentes approches proposent de le paralléliser, de le combiner avec des méthodes exhaustives d'exploration locale [62], ou de le doter de ressources mémoire qui lui permettent de garder une partie des états visités. En outre, lorsque des informations supplémentaires sur le système exploré sont disponibles, certaines améliorations visent à guider le *RW* pour arriver plus rapidement à détecter les erreurs du système.

L'objectif de la thèse est de montrer que l'amélioration du schéma même de l'exploration randomisée peut apporter des performances importantes, à la fois par rapport au schéma classique basé sur *RW* et par rapport aux algorithmes déterministes d'exploration : il permet d'augmenter de manière significative le nombre des états explorés et d'améliorer leur atteignabilité. En plus, l'efficacité des algorithmes proposés est contrôlée par une caractérisation du graphe exploré. Les méthodes d'exploration proposées dans cette thèse, couplées à des techniques de remplacement en mémoire (ou "caching") des états visités améliorent encore les performances de ces algorithmes.

La thèse propose donc un schéma d'exploration général et flexible car paramétré, qui englobe toute une panoplie de stratégies d'exploration randomisées, entre autres *RW*. La paramétrisation de ce schéma général permet d'en tirer plusieurs algorithmes qui seront étudiés théoriquement et par simulation. Ce schéma introduit également les méthodes de remplacement en mémoire qui seront explorées et étudiées à part.

1.2.1 Modélisation

Un algorithme d'exploration qui entre dans le schéma général est défini principalement par trois paramètres : une *condition d'arrêt* de l'exploration, une fonction de *sélection* de l'état suivant à explorer, à partir d'un état ou d'un ensemble d'états déjà explorés et stockés en mémoire, et une fonction d'*actualisation* qui définie la stratégie de remplacement des états gardés en mémoire. On cherche à maximiser la portion du graphe explorée et à minimiser le temps consommé dans cette exploration, sous la contrainte de l'espace mémoire limité. Nous avons, donc, considéré dans notre modélisation plusieurs critères pour évaluer une exploration, notamment le nombre

moyen d'états couverts et le temps de couverture moyen, ainsi que la probabilité d'atteignabilité. En ce qui concerne la taille, limitée, de la mémoire utilisée, elle est incluse explicitement comme paramètre principal dans notre modèle général. Ce paramètre permet de prendre en considération la contrainte posée par les ressources mémoire limitées et de contrôler à chaque instant l'état de la mémoire, ce qui garantit qu'elle ne soit pas saturée et évite le problème de swapping.

1.2.2 Les Algorithmes proposés

Les algorithmes proposés se situent dans un schéma générique qui inclue les formes des algorithmes d'exploration randomisés déjà existants et accepte d'autres formes plus riches. Un jeu assez complet d'algorithmes a été sélectionné à partir de ce schéma, citons en particulier, DORS (*Depth Oriented Random Search*) qui est un algorithme randomisé orienté en profondeur inspiré de DFS et BORS (*Breadth Oriented Random Search*) qui est son dual orienté en largeur et inspiré de BFS. En outre, nous avons proposé et étudié l'algorithme URS (*Uniform Random Search*) qui est un algorithme d'exploration uniforme, ne favorisant ni largeur ni profondeur, et qui vise à réaliser une exploration équilibrée du graphe détats. Par ailleurs, RMA (*Random Mixing Algorithm*) est un algorithme d'exploration aléatoire paramétrée. Il mélange l'exploration en profondeur/en largeur selon le paramètre de mixage mx. Le choix de ce paramètre est guidé par un facteur de densité DF caractéristique du graphe considéré. Une étude théorique et expérimentale a été effectuée pour chacun de ces algorithmes.

1.2.3 Les techniques de remplacement

Les techniques de remplacement, dites *techniques de "caching"* permettent de gérer les états explorés et gardés en mémoire ; si un nouvel état vient d'être visité, alors que la mémoire, dite *"cache"*, est pleine, ce nouvel état sera gardé à la place d'un ancien état en mémoire. Le choix des états à sacrifier constitue la fonction d'*actualisation* dans notre schéma générique. Le travail sur les stratégies de remplacement vise à réduire la réexploration redondante des états visités et supprimés du cache.

En effet, l'utilisation des techniques de remplacement avec l'exploration randomisée est indispensable pour garantir ses performances : d'un coté, sans garder aucun état en mémoire comme le *RW*, le risque des explorations redondantes devient très élevé. De l'autre coté, si on s'engage à garder tous les états en mémoire, on tombe dans le problème d'insuffisance de celle-ci et par la suite abandon de l'exploration ou swapping. La meilleure solution étant de garder le maximum d'états possibles en mémoire et de les manipuler par des remplacements afin d'optimiser le gain de cette mémoire.

L'intégration de ces techniques dans notre schéma général d'exploration a été effectuée selon deux formes : La première forme est la répétition avec réinitialisation. Lorsqu'un nouvel état est exploré, il est ajouté en mémoire. À chaque fois que la mémoire est pleine, elle est vidée de tous les états et l'algorithme d'exploration est réinitialisé puis

relancé. La deuxième forme est le remplacement classique état par état. Si la mémoire est pleine, chaque état nouvellement exploré prend la place d'un état déjà présent en mémoire. L'ancien état à sacrifier est choisi selon une stratégie de remplacement. Les deux formes nous ont permis de mettre en valeur les performances importantes des algorithmes randomisés d'exploration en couverture et en atteignabilité.

1.2.4 Monte Carlo Model Checking

La vérification peut être vue comme la recherche dans le graphe de transition d'un état défectueux. L'atteignabilité des états sera une question très importante dans ce cas. L'algorithme MC^2 (Monte Carlo Model Checking) [34] consiste à réitérer le *RW* simple un certain nombre R de fois. L'enjeux porte ici sur le nombre d'itérations à effectuer, autrement dit, le calcul d'une borne supérieure sur le nombre de répétitions R nécessaires pour garantir une certaine confiance quant à l'existence ou non d'une erreur dans le système. Cette borne dépend de la probabilité de détection d'erreur dans une itération du *RW*.

En suivant le même processus, nous réitérons nos algorithmes et nous cherchons une borne sur le nombre de répétitions R. Nous montrons que la borne calculée pour nos algorithmes est meilleure que celle calculée pour *RW* dans MC^2 car la probabilité d'atteinte des états est meilleure pour nos algorithmes que pour *RW*. Par conséquent, des gains significatifs seront apportées par notre approche en temps de vérification.

1.3 Organisation du document

Dans le chapitre 2, nous présentons le model checking et ses limites, puis les méthodes et techniques proposées pour palier au problème de l'explosion de la taille de l'espace d'états, en particulier, la technique de remplacement des états en mémoire "caching", qui sera combinée à notre approche plus tard. Nous présentons les principaux apports dans ce domaine, liés particulièrement à l'exploration totale.

Dans le chapitre 3, nous analysons l'introduction des approches randomisées dans le domaine de la vérification. Les fondement théoriques de la marche aléatoire ainsi que de l'approche Monte Carlo sont détailés. Ensuite, nous examinons les différentes approches utilisant la marche aléatoire et la marche aléatoire améliorée dans le domaine de la vérification et nous discutons leurs performances et leurs limitations.

Le chapitre 4 propose une modélisation générale de l'exploration. La problématique générale est discutée comme un compromis entre l'espace mémoire et le temps d'exécution et les critères de l'exploration sont détaillés. Nous introduisons ensuite, un schéma général de l'exploration, permettant de présenter une très grande classe d'algorithmes et nous examinons chacune de ses principales fonctions. À la fin du chapitre, une table récapitulative compare des différentes méthodes d'exploration.

Nous spécifions, ensuite dans le chapitre 5, les différents algorithmes aléatoires qui peuvent être inclus dans notre schéma générique. Nous traçons une méthodologie commune qui servira à l'étude de nos algorithmes dans les chapitres suivants et nous produisons les formules théoriques générales. Cette méthodologie et ses résultats généraux peuvent être appliqués à tout algorithme randomisé que l'on souhaite étudier.

Nous dédions les chapitres 6, 7 et 8, respectivement, à l'étude des algorithmes DORS (Depth Oriented Random Search) vs. BORS (Breadth Oriented Random Search), URS (Uniform Random Search) et RMA (Random Mixing Algorithm). Cette étude sera faite théoriquement et expérimentalement en suivant la méthodologie précisée au chapitre 5.

L'étude théorique porte sur le calcul de plusieurs mesures afin d'analyser les performances de nos algorithmes. En particulier, nous avons établi les formules récursives du *temps moyen de couverture* et du *nombre moyen d'états couverts* en fonction du temps. Le calcul de ces critères a été effectué pour différents cas de graphes (arbres et grilles) et leurs courbes d'évolution ont été tracées.

Ces algorithmes sont aussi comparés expérimentalement, afin de confirmer les résultats théoriques, sur des graphes issus de protocoles réels. La plate forme utilisée pour ces expérimentations est "IF toolset". Nous avons expérimenté les mêmes critères discutés théoriquement et nous avons montré la capacité des algorithmes aléatoires d'aller au delà de l'exploration déterministe présentée par BFS.

Nous nous sommes intéressés dans le chapitre 9 aux stratégies de remplacement des états en mémoire combinées à l'exploration randomisée. En outre, différents paramètres et techniques de stockage et de remplacement sont proposés et comparés. La comparaison des probabilités d'atteignabilité de nos algorithmes avec RW, nous mène à proposer une approche Monte Carlo généralisée nous permettant de réduire la borne de répétition R ainsi que le temps d'exécution. Cette étude sera finalisée par la présentation de nos résultats expérimentaux confirmant le gain apporté par nos algorithmes dans le cadre Monte Carlo.

Enfin, la conclusion générale de la thèse ainsi que quelques perspectives sont données dans le chapitre 10.

Chapitre 2

Vérification déterministe

2.1 Introduction

La vérification est le problème de décider si un système respecte ou pas certaines propriétés prédéfinies. Par exemple, un protocole d'exclusion mutuelle doit respecter la propriété qu'à tout moment, pas plus d'un seul utilisateur n'accède à des variables, ressources ou sections critiques.

Les systèmes actuels sont devenus de plus en plus complexes. Leur vérification automatique est, alors, devenue une obligation. L'automatisation de la vérification est soumise à plusieurs contraintes en ressources humaines (modélisation, interaction, correction) et matérielles (mémoire, temps de calcul, coût de développement...). Aujourd'hui, grâce à diverses techniques de vérification, regroupées sous le terme de "méthodes formelles", il est possible, dans la majorité des cas, de vérifier automatiquement, et en respectant ces contraintes humaines et matérielles, si un système respecte ou non les propriétés voulues. Les *méthodes formelles* regroupent des techniques qui utilisent un formalisme et des outils mathématiques pour mener à bien la tâche de vérification. On commence par produire une description intermédiaire, dite *spécification*, du système que l'on souhaite développer, puis, préciser par des assertions les propriétés qu'il doit satisfaire. Cette spécification est basée sur un langage formel. Elle est utilisée comme référence pendant le développement du système et sert à vérifier que les propriétés seront satisfaites pour toute exécution de celui-ci.

Dans le but d'être exhaustif dans la description du contexte de la thèse, nous présentons dans ce chapitre les différentes approaches pour la vérification déterministe. Notons que certaines de ces méthodes sont compatibles et utilisables conjointement avec notre approache randomisée. Cette utilisation sera détaillée particulièrement pour les techniques de remplacement en mémoire.

2.2 Les méthodes formelles de vérification

Au sein des méthodes formelles, on trouve deux grandes familles : la déduction automatique et le model checking :

2.2.1 La déduction automatique

La déduction automatique de théorème [36], consiste à laisser l'ordinateur prouver les propriétés automatiquement, étant donnés une description du système, un ensemble d'axiomes et un ensemble de règles d'inférences. Afin de vérifier par déduction automatique qu'un système vérifie une propriété donnée, il est nécessaire de pouvoir transformer le système et la propriété que l'on souhaite vérifier en objets mathématiques [40]. Cependant, la recherche de preuve est connue pour être un problème non décidable en général [31]. Dans ce cas, les assistants de preuves interactifs comme PVS [54], Coq [41] et HOL [30] permettent à l'utilisateur de guider la preuve. Ils fonctionnent en finalisant une preuve suggérée par l'utilisateur et interagissent avec l'utilisateur en cas de blocage. Cette interaction dans l'utilisation d'un outil de preuve nécessite, en général, un haut degré d'expertise.

2.2.2 Model Checking

Model checking [42, 13] désigne une famille de méthodes de vérification automatique des systèmes dynamiques (souvent d'origine informatique ou électronique). Il s'agit de vérifier algorithmiquement si un modèle (déterministe ou probabiliste) donné sous forme d'automate [65, 47] ou de OBDDs (Ordered Binary Decision Diagram) [7, 8, 9], satisfait une spécification, souvent formulée en termes de logique temporelle ; LTL (Linear Temporal Logic), CTL (Computation Tree Logic), CSL (Continuous Stochastic Logic)... [57, 2, 15, 61]. Model checking analyse exhaustivement l'évolution du modèle du système lors de toutes ses exécutions possibles. Il est mené à l'aide d'algorithmes permettant l'énumération de tous les états du système. On reçoit en sortie soit une confirmation ou une probabilité que la propriété est vérifiée ou bien un contre-exemple, c'est-à-dire une trace d'exécution qui conduit à l'état qui a produit l'erreur. L'efficacité de cette méthode dépend en général de la taille de *l'espace d'états*, c'est-à-dire l'ensemble de tous les états possibles du système et trouve donc ses limites dans les ressources mémoire de l'ordinateur.

2.2.2.1 Model checking explicite

Il consiste à exprimer le modèle considéré au moyen d'un graphe orienté, formé de nœuds et de transitions. Chaque nœud représente un état du système, chaque transition représente une évolution possible du système d'un état donné vers un autre état. Chaque nœud du graphe orienté est étiqueté par l'ensemble des propositions logiques vraies en cet état. Un tel graphe est appelé *structure de Kripke* [13]. Ensuite, la négation de la formule que nous souhaitons tester est exprimée à l'aide de logique temporelle. La négation de cette formule est donc elle-même transcrite sous forme d'une structure

de Kripke. La troisième étape consiste à réaliser le produit cartésien synchrone des deux structures de Kripke obtenues précédemment. Le langage reconnu par ce produit étant formé de l'ensemble des séquences respectant la spécification et ne vérifiant pas la propriété en question [13], si ce langage est vide, alors le système satisfait la formule logique. Sinon, toute séquence appartenant au langage du produit constitue un contre-exemple à la spécification.

2.2.2.2 Model checking symbolique

Énumérer explicitement tous les états de l'automate peut être coûteux, c'est pourquoi on procède par des méthodes symboliques, introduites par Ken McMillan et Ed Clarke. Cette approche est fondée sur la représentation des états et des transitions du système par des ensembles, i.e. chaque état (resp. transition) symbolique représente un ensemble d'états (resp. transitions) explicites. De nombreuses méthodes de représentation d'ensembles d'états ont vu le jour. Les plus connues utilisent des diagrammes de décision binaire (BDDs) [10, 68] et la résolution de satisfiabilité (SAT Solving) [4, 5]. Le fonctionnement de model-checking symbolique consiste en l'obtention de points fixes pour déterminer les états accessibles ainsi que ceux qui satisfont une ou plusieurs propriétés. Les points fixes sont calculés à l'aide de fonctions dites transformateurs de prédicats, qui opèrent sur les états symboliques.

2.3 L'explosion combinatoire de la taille de l'espace d'états

On nomme explosion combinatoire de la taille de l'espace d'états le fait que le nombre des états du système augmente de façon exponentielle en fonction du nombre de ses composants (nombre de processus, taille de données...). Les méthodes de vérification par model checking, bien qu'efficaces et pratiques [58, 12, 13], souffrent du problème connu de l'explosion de l'espace d'états, ce qui rend les ressources mémoire disponibles insuffisantes pour un stockage total et une vérification exhaustive.

Plusieurs techniques ont été développées pour faire face à ce problème :

2.3.1 Réduction de la taille de l'espace d'états

2.3.1.1 Réduction d'ordre partiel

La réduction d'ordre partiel est une technique qui vise à réduire la taille de l'espace d'états à explorer par un algorithme de model checking [26, 27, 53, 23, 50, 1]. Elle exploite la commutativité des transitions exécutées en concurrence et qui donnent le même état lorsqu'elles sont exécutées dans des ordres différents. Les techniques de réduction d'ordre partiel explorent seulement une partie réduite de l'espace d'états qui est suffisante pour vérifier la propriété. La taille des graphes d'états traités par model checking peut être considérablement réduite par cette technique.

2.3.1.2 Réduction symétrique

En pratique, les systèmes d'états finis concurrents présentent un haut degré de symétrie causée généralement par la présence de composants identiques dans le système. Intuitivement, deux états sont symétriques si leurs variables d'états et valeurs sont sémantiquement les mêmes. Par exemple, dans le cas de 4 processus identiques, chacun ayant deux états locaux possibles A et B, l'état (B, A, A, A) est équivalent aux états qui lui en sont obtenus par symétrie c'est à dire (A, B, A, A), (A, A, B, A) et (A, A, A, B). Ces quatres états sont donc deux à deux équivalents et forment ainsi une classe d'équivalence (par symétrie). Cette classe d'équivalence peut être représentée comme $(A = 3, B = 1)$.

Les symétries peuvent être exploitées pour réduire la taille de l'espace d'état, représentant le système, à explorer [14, 19]. Le fonctionnement de la réduction symétrique consiste à décomposer, en premier lieu, l'espace d'états en classes d'équivalence d'états symétriques. L'algorithme de model checking doit, ensuite, explorer au moins un état représentatif de chaque classe d'équivalence, les autres états de la classe ne devant pas, forcément, être explorés. Un problème associé à cette méthode est d'assurer l'identification et le calcul des classes d'équivalence qui peut être dans certains cas une tâche très difficile, notamment pour les grands systèmes [14, 19].

2.3.2 Réduction de la mémoire nécessaire pour le stockage des états

2.3.2.1 Hash compaction

Dans la vérification par énumération explicite, pour chaque état atteint, tout le descripteur de l'état est gardé en mémoire. Un descripteur d'état contient les valeurs des variables en cet état [67]. La méthode de "hash compaction", introduite par Wolper et Leroy [67] et améliorée par Stern et Dill [63], cherche à réduire les besoins mémoire nécessaires pour le stockage des descripteurs des états. Un descripteur d'états compressé, calculé à l'aide d'une fonction de hashage, est stocké au lieu du descripteur entier. Cette méthode permet un gain typique en mémoire de facteur 10. Ce gain vient au détriment d'une petite probabilité que la recherche soit incomplète, i.e. que l'exploration omet certains états et produit, ainsi, des "faux positifs".

2.3.2.2 Bit-state hashing

La méthode de "bit-state hashing" ou super-trace, introduite par Holzmann [39], vise à minimiser la mémoire nécessaire pour le stockage des états. Au lieu de maintenir en mémoire un ensemble de descripteurs d'états, "bit-state hashing" maintient un ensemble de bits initialement mis à zéro. Lorsqu'un état est visité, la fonction de hachage est appliquée au descripteur de cet état, donnant deux valeurs de hashage. Les bits ayant ces deux valeurs comme indice sont mis à un. Lorsque l'algorithme examine un nouvel état et trouve que les deux bits correspondants sont à un, il considère que cet état est déjà visité. Notons que la fonction de hachage est, en général, non injec-

tive : deux doublets de valeurs de hashage associés à deux descripteurs (états) différents peuvent avoir un ou deux bits en commun. De ce fait les deux bits associés à un état non visité peuvent être mis à 1 suite à la visite d'autres états. Dans ce cas létat non visité va être omis. L'algorithme risque, ainsi, d'omettre de vérifier certains états du système.

Dans la table 2.1, nous présentons un récapitulatif de ces méthodes :

Méthode	But	Principe
Réduction d'ordre partiel	Réduire l'espace d'états à explorer	Réduire les transitions qui reviennent à des exécutions concurrentes équivalentes
Réduction symétrique	Réduire l'espace d'états à explorer	Réduire les états ayant la même sémantique
Hash compaction	Réduire la taille des états stockés	Stocker en mémoire une valeur compactée du descripteur d'état
Bit-state hashing	Réduire la taille des états stockés	Stocker en mémoire deux bits au lieu de tout le descripteur d'état

TABLE 2.1 – Méthodes de model checking permettant l'optimisation des ressources

2.3.3 Le remplacement des états en mémoire

Un algorithme de model checking effectue des parcours de l'espace d'états du système. Les propriétés voulues sont testées pour les états visités, ces états sont alors dits explorés et sont gardés en mémoire. À la rencontre d'un état du système, l'algorithme vérifie s'il est déjà existant en mémoire, dans ce cas, il n'y aura pas besoin de le réexplorer. Cependant, il se trouve que la mémoire se remplit par les états explorés et il n'y aura pas de place pour stocker un nouvel état. Trois possibilités se présentent dans ce cas : abandonner l'exploration, continuer l'exploration sans stocker le dernier état visité ou remplacer un état déjà vu dans la mémoire (ou cache) par le dernier état visité. La troisième possibilité constitue la technique de "caching". C'est une technique qui se concentre sur le problème de stockage en mémoire des états explorés par un algorithme de model checking. Cette technique rend possible l'exploration exhaustive du système en ne gardant en mémoire à chaque instant, qu'un sous ensemble de l'espace d'états total [25, 28]. En remplaçant les anciens états, le model checker risque de les réexplorer de nouveau et refaire le même travail (redondance), mais ceci ne donne pas, en partique, de résultat incorrect car l'expérience a montré (voir la suite de la sous-section) que l'exploration totale de l'espace d'état est atteinte dans plusieurs cas, malgré ce problème de redondance.

2.3.3.1 Critères de comparaison

Comme nous l'avons signalé ci-dessus, la technique du caching vise à pouvoir explorer exhaustivement le système par remplacement des anciens nœuds dans le cache par les nouveaux. Le critère commun dans la littérature sur le caching est le taux de redondance à la couverture totale. Le taux de redondance étant exprimé en nombre de transitions par rapport à la taille du graphe. Notons que le nombre de transitions signifie ici le nombre de visites des états du système ce qui est proportionnel au temps d'exécution. Le critère utilisé revient donc au temps de couverture totale en fonction de la taille du graphe considéré. Les résultats des algorithmes de caching dépendent de la taille de la mémoire utilisée. Ils sont exprimés en fonction du ratio de la taille de la mémoire à la taille du graphe.

2.3.3.2 Stratégies de remplacement

Lors du remplacement des états en mémoire, il faut décider d'une stratégie de remplacement. Le nouvel état exploré prend la place d'un ancien état choisi selon cette stratégie. Différentes politiques ont été proposées. On trouve par exemple [38] :

- H1 : Les états les plus fréquemment visités
- H2 : Les états les moins fréquemment visités
- H3 : Les états dans la classe d'états la plus large actuellement, les états étant classés par le nombre de fois qu'ils ont été visités
- H4 : Les états les plus anciens dans le cache
- H5 : Les états dans la moitié inférieure de l'arbre de recherche courant.

À ces stratégies s'ajoute le remplacement aléatoire qui consiste, comme son nom l'indique, à choisir aléatoirement dans le cache le nœud à remplacer. La stratégie aléatoire est la plus utilisée dans la littérature.

2.3.3.3 Efficacité de la technique

Les premièrs études du "caching" pour model checking ont été faites par Holzmann [37]. Depuis, plusieurs résultats significatifs ont été obtenus. Dans [37], qui n'est pas un article consacré particulièrement au caching, Holzmann explore cette technique pour un seul modèle de 150 000 états. Les états sont remplacés en utilisant la sélection "round robin" [37]. La conclusion rapporte qu'un cache de taille d'environ la moitié de l'espace d'états apporte des performances acceptables.

Les cinq stratégies énumérées dans le paragraphe précédent ont été testées par Holzmann [38] pour des protocoles de tailles moyennes. Deux exemples sont présentés : un avec 4523 états et l'autre avec 8139 états. La conclusion du papier est que la stratégie H4 est la plus rapide car son temps d'exécution augmente raisonnablement, bien que dans l'un des exemples, elle produit le taux de redondance le plus élevé 59% comparé à un maximum de 50% pour les autres stratégies.

Jard et Jérom étudiaient le caching dans leurs travaux [43, 44, 22]. Les auteurs génèrent des graphes aléatoires et les explorent en utilisant depth first search et en se basant sur un remplacement aléatoire. Ils rapportent que, dans un cas typique, un cache permet de visiter 70% plus d'états, par rapport au cas ou l'on n'utilise pas le caching, au détriment d'une augmentation dans le temps d'exécution de 50%.

Toutes ces méthodes utilisent le caching associé à depth-first search pour une exploration totale de l'espace d'états. Dans [64] le caching des états à été utilisé avec une exploration partielle basée sur breadth-first search et une stratégie de remplacement aléatoire. Le résultat de vérification est probabiliste. Lorsque la mémoire est pleine, l'algorithme devient plus lent mais n'abandonne pas l'exploration. Cet algorithme peut épargner jusqu'au 30% de la mémoire avec une pénalité en temps de 100%.

D'après ce qui précède, on peut dire, sommairement, que le caching des états est efficace pour une exploration totale lorsque la taille du cache est entre 33% et 50% de la taille de l'espace d'états total [38, 37, 26, 43, 44] et qu'un remplacement aléatoire des états du cache est la meilleure stratégie. En effet, lorsque la taille du cache et relativement grande (voir proportions ci-dessus), peu d'états sont remplacés. La revisite d'un état remplacé est peu probable et même si cela se produit, les successeurs de cet état seront probablement trouvés dans le cache. En revanche, lorsque la taille du cache est très petite par rapport à la taille de l'espace d'état, la probabilité de revisiter un état remplacé et avoir à revisiter ses successeurs et leurs successeurs est importante. En plus, chaque état revisité est réinséré dans le cache remplaçant un autre état, ce qui rend la situation pire encore et le temps d'exécution monte excessivement.

Dans le chapitre 9.2, nous proposons de coupler les stratégie de remplacement à nos schémas d'exploration aléatoires, proposés dans les chapitres 6, 7 et 8. Nous proposons également une stratégie de remplacement qui donne des résultats meilleurs que ceux obtenus avec des stratégies préexistentes, en particulier le remplacement aléatoire.

2.4 Conclusion

Nous avons présenté dans ce chapitre les méthodes formelles de vérification, à savoir la déduction automatique et le model checking. Nous avons vu que l'explosion de l'espace d'états constitue le problème majeur rencontré en model checking. Nous avons présenté, ensuite, les différentes solutions qui ont été proposées dans la littérature pour remédier à ce problème en réduisant soit l'espace d'états à explorer soit les besoins en mémoire ou en utilisant des techniques de remplacement des états explorés en mémoire. Ces techniques de remplacement seront étudiées plus en détails, dans le cadre de notre approche, dans le chapitre 9 où elles seront comparées à une stratégie de remplacement que nous proposons conjointement avec de nouveaux schémas d'exploration présentés dans les chapitres 5, 6 et 7.

Chapitre 3

Vérification randomisée

3.1 Introduction

Pour affronter le problème d'explosion combinatoire de la taille de l'espace d'états, certaines méthodes se sont orientées à faire des explorations partielles via des algorithmes randomisés. Il est important de noter que les algorithmes randomisés d'exploration sont *compatibles* avec les méthodes de réduction d'états (réduction d'ordre partielle, réduction symétrique) et les techniques de compression d'états (hash compaction, bistate hashing) connues. Cette compatibilité est importante car ces algorithmes ne sont pas de purs concurrents aux méthodes déterministes existantes mais viennent en complément et doivent être utilisés en même temps que celles-ci.

Les algorithmes randomisés d'exploration sont généralement basés sur la marche aléatoire [66, 35, 34, 51], ou sur une forme améliorée de celle-ci [62, 46]. La marche aléatoire a été appliquée au model-checking, en premier, par West en 1986 [66]. Les fondement théoriques de l'utilisation de la marche aléatoire en model checking sont donnés par Grosu et Smolka qui proposent l'approche "Monte Carlo Model Checking" (MC^2, QMC) [34], [33]. Dans ce chapitre, nous présentons l'utilisation de la marche aléatoire et certaines de ses variantes dans le contexte de model-checking. Nous détaillons, en particulier, l'approche Monte Carlo. Dans le chapitre 9 sera présentée notre approche pour Monte Carlo Model Checking et nos résultats seront comparés à MC^2 et QMC.

3.2 Exploration randomisée et vérification

Dans ce qui suit, on considère que le système est modélisé par un graphe d'état $G(M, V_0, Succ)$, où M désigne l'ensemble des nœuds représentants les états du système et V_0 désigne l'ensemble des états initiaux. Dans la suite on se limitera au cas d'un seul état initial v_0. La fonction Succ appliquée à un nœud v donne l'ensemble de ses successeurs. Du fait que l'on ne dispose pas du graphe complet, on ne peut que visiter les nœuds de proche en proche en utilisant cette fonction. La machine utilisée disposera d'un générateur de nombres aléatoires $rand()$ qui donne une variable (pseudo) aléatoire

uniformément distribuée sur $[0,1]$. Le degré $d(v)$ d'un état v est défini par $d(v) = |Succ(v)|$. En outre, les définitions suivantes seront considérées le long de la thèse.

Définition 3.1 *Soit v, v' deux nœuds du graphe G. On appelle chemin de v vers v', $C_{(v,v')}$ toute suite de nœuds distincts $v_1, v_2, ..., v_n$ tel que $v_{i+1} \in Succ(v_i)$ pour $i = 1, ..., n-1$, et $v_1 = v, v_n = v'$. La longueur de $C_{(v,v')}$ sera $|C_{(v,v')}| = n$.*

Définition 3.2 *Soit v_0 le nœud initial. À tout nœud v, on associe la distance*

$$d(v_0, v) = Min\{|C_{(v_0,v)}|, C_{(v_0,v)} \text{ est un chemin de } v_0 \text{ vers } v\}$$

La profondeur du graphe G est définie par $h = Max_{v \in G}\ d(v_0, v)$.

3.2.1 Les algorithmes randomisés

Un algorithme randomisé est un algorithme qui contient une affectation d'une variable basée sur un générateur de nombres aléatoires. Les algorithmes randomisés sont largement utilisés, principalement pour deux raisons : simplicité et rapidité [52]. Ils proposent souvent des solutions efficaces pour des problèmes difficiles connus [59, 48, 3]. Une conséquence de leur utilisation est que la correction du résultat ou la terminaison de l'exécution est garantie seulement avec une certaine probabilité. Les algorithmes aléatoires sont classés en deux types [52] :

- Un algorithme *Monte Carlo* est un algorithme randomisé qui donne un résultat approché dont on peut contrôler l'erreur. Son temps d'exécution est fonction déterministe de la taille des données.
- Un algorithme *Las Vegas* est un algorithme randomisé qui donne toujours un résultat correct. Son inconvénient est que le temps d'exécution change d'une exécution à une autre.

3.2.2 La marche aléatoire (Random Walk)

Une marche aléatoire sur un graphe G est un cas particulier du processus stochastique appelé "chaîne de Markov à temps discret", avec un espace d'états M des transitions de probabilité *homogènes* et *uniformes*. L'algorithme démarre de l'état initial du graphe, et à chaque étape, il choisit aléatoirement selon une distribution (de transition) *uniforme* un successeur de l'état courant et le visite. Ce choix est *indépendant* du chemin précédemment parcouru, ce qui est caractéristique d'une chaîne de Markov. L'algorithme se termine lorsqu'il n'y a plus de possibilité de choisir un successeur (deadlock) ou, dans le cas d'exploration en boucles, lorsqu'on atteint un nombre maximal d'étapes fixé par l'utilisateur.

Définition 3.3 *Une marche de longueur n sur un graphe G est une suite de nœuds $v_0, v_1, ..., v_n$ tel que $v_{i+1} \in Succ(v_i)$ pour $i = 0, ..., n-1$. La marche est dite aléatoire ssi chaque v_{i+1} est tiré aléatoirement et uniformément parmi les successeurs de v_i*

Du point de vue théorique, une caractéristique très importante de la marche aléatoire, et de tout algorithme d'exploration A appliqué au graphe G, est le *temps de couverture*.

Définition 3.4 *Le temps de couverture d'un algorithme A sur un graphe G est défini comme étant le nombre moyen d'étapes nécessaires à l'algorithme A pour visiter tous les nœuds de G*

Pour les graphes non orientés, le temps de couverture d'un graphe quelconque G par la marche aléatoire est polynomial en sa taille $n = |M|$. Plus précisément, il est compris entre $(1+o(1))n \log n$ [20] et $(4/27+o(1))n^3$ [21]. Notons que ces complexités (bornes), notamment la borne inférieure en $n \log n$, restent raisonnables pour des graphes qui ne sont pas très larges. Pour les graphes orientés, ce temps est en général exponentiel en n, sauf pour quelques classes restreintes de graphes [35, 51].

3.2.2.1 Utilisation de la marche aléatoire en model checking

Le problème de la vérification consiste aussi en un problème d'atteignabilité [49, 17, 24] dans l'espace d'états. La première utilisation de la marche aléatoire pour l'atteignabilité apparaît dans [66]. L'auteur a démontré sur un cas d'étude que la marche aléatoire constitue une bonne technique pour la détection d'erreurs.

Dans [35], les graphes considérés sont fortement connexes. Une borne supérieure a été établie sur le nombre d'étapes nécessaires à la marche aléatoire, qui démarre d'un état v du graphe, pour atteindre, avec une probabilité d'erreur inférieure à ϵ, un autre état v' du graphe. Cette borne est donnée par :

$$1/\epsilon.|M||E|,$$

où E désigne l'ensemble des arcs de G et ϵ la probabilité d'erreur. Si la marche aléatoire effectuée n'atteint pas v' alors on peut affirmer qu'avec une probabilité supérieure à $1-\epsilon$, il y a pas de chemin entre v et v'.

Notons que la borne établie est très grande en pratique et n'est valable que pour une classe restreinte des graphes. Cette classe est si restreinte qu'elle n'est pas très intéressante pour le model checking.

3.2.2.2 L'approche Monte Carlo Model Checking

Dans l'approche Monte Carlo Model Checking (MC^2) [34] [33], Grosu et Smolka se placent dans le contexte de graphes quelconques. La marche aléatoire est répétée un certain nombre R de fois. L'intérêt porte entièrement sur la répétition de la marche aléatoire jusqu'à l'atteinte de la performance souhaitée. Plus précisément on cherche une borne supérieure sur le nombre R de répétitions nécessaires à la marche aléatoire pour trouver un bug (nœud défectueux), avec une certaine probabilité et sous certaines hypothèses.

Le problème de model-checking consiste à vérifier si une formule ϕ est satisfaite par un système S. Alors, à l'instance $S \models \phi$ on peut associer la variable de Bernoulli Z définie comme suit :

Définition 3.5 *Considérons une exécution de l'algorithme A dédié à vérifier la formule ϕ pour le système S. On définit la variable aléatoire de Bernoulli Z par : $Z = 0$, si un contre exemple de ϕ est trouvé durant cette exécution, et $Z = 1$ sinon. Notons par q_Z la probabilité de trouver un contre exemple de ϕ dans une exécution aléatoire de A. La moyenne p_Z de Z est donc égale à $1 - q_Z$:*

$$q_Z = Pr(Z = 0),\ p_Z = 1 - q_Z = Pr(Z = 1)$$

Notons que nous avons allégé les notations dans la définition ci-dessus en omettant la dépendance de Z et de q_Z par rapport à l'algorithme d'exploration A considéré.

On considère la variable de Bernoulli Z de la définition 3.5 associée a l'algorithme de la marche aléatoire $(RW) : Z = 0$, se produit avec une probabilité q_Z, et correspond à un contre exemple (i.e. violation de ϕ). On définit la variable aléatoire géométrique X de paramètre q_Z dont la valeur correspond au nombre d'itérations indépendantes de RW nécessaires à l'obtention d'un contre exemple : $Z = 0$. La distribution de X est définie par :

$$p(R) = Pr[X = R] = p_Z^{R-1} q_Z.$$

Ce qui donne une fonction de distribution cumulative (CDF) de X :

$$F(R) = Pr[X \leq R] = \sum_{r=1}^{R} p(r) = 1 - p_Z^R.$$

Pour un paramètre de confiance donné δ, le nombre minimal d'itérations nécessaires pour détecter un contre exemple, avec une probabilité $1-\delta$ est donné par : $F(R) = 1-\delta$, soit :

$$R = ln(\delta)/ln(1 - q_Z) = ln(\delta)/ln(p_Z).$$

Du point de vue théorique, le résultat principal de [34] peut être exprimé, dans le cadre de test d'hypothèse, comme suit :

Théorème 3.1 *Soit $T = \log(\delta)/\log(1 - \epsilon)$, alors, $Pr(X > T|H_0) < \delta$ où H_0 est l'hypothèse que : $q_Z \geq \epsilon$.*

Autrement dit, sous l'hypothèse : $q_Z \geq \epsilon$, il n'est pas nécessaire d'aller au delà de T itérations pour s'assurer que la propriété ϕ est vérifiée avec une probabilité au moins égale à $1 - \delta$. Si, après R itérations, aucun contre exemple n'est trouvé, l'algorithme affirme que la probabilité de trouver des erreurs dans les prochaines exécutions de la marche aléatoire, sous l'hypothèse que $q_z \geq \epsilon$, est inférieur à δ.

En pratique le paramètre q_Z (ou p_Z) est en général inconnu et difficile à estimer. Conformément au théorème 3.1, [34] utilise une borne inférieure ϵ de q_Z, ce qui aboutit à un nombre d'itérations $T \geq R$ donné par :

$$T = \log(\delta)/\log(1-\epsilon).$$

L'algorithme MC^2 de [34], se présente donc ainsi :

//Structures de données :
ϵ, δ : **réels, avec** $0 < \epsilon \leq q_Z$ **et** $0 < \delta \leq 1$;
T : **réel** ;
i : **entier** ;

//Initialisation :
$T = \log(\delta)/\log(1-\epsilon).$;

//Corps de l'algorithme :
Pour i **de** 1 **à** M **faire**
  Exécuter RW , le chemin parcouru étant C.
  Si (C constitue un contre exemple) **Alors**
    retourner(Faux, C)
  Sinon
    retourner(Vrai, $\mathbb{P}(X > T) < \delta$)
  Fin Si
Fin Pour

FIGURE 3.1 – L'algorithme MC^2

L'application de MC^2 exige donc de fixer ou savoir a priori une borne inférieure ϵ de q_Z. Les résultats de MC^2 dépendent nettement de cette borne. Un choix non approprié de ϵ, comme borne inférieure de q_Z, peut conduire à des valeurs de T inutilement grandes. Ce problème est d'autant plus accru que la valeur de q_Z à minorer est plus petite.

Dans [33], l'auteur adopte une deuxième approche qui consiste toujours à appliquer l'algorithme MC^2, mais en estimant le paramètre p_Z moyennant l'algorithme OAA (Optimal Approximation Algorithm) [16] qui fournit une estimée $\tilde{p}_Z$ de p_Z en un nombre d'itérations optimal à une constante près. L'algorithme ainsi obtenu, appelé QMC (Quantitative Model Checking), consiste donc à générer des échantillons Z_i de Z (i.e. réitérer RW) autant que nécessaire pour l'algorithme OAA appliqué avec les

paramètres (δ, ϵ). Les ré-itérations sont arrêtées dès qu'un contre exemple est détecté. Sinon, l'estimée fournie par OAA est $\tilde{p}_Z = 1$, car dans ce cas tous les Z_i sont égaux à 1. Par conséquent, on aura dans ce cas de non détection d'erreur :

$$\mathbb{P}(q_z < \frac{\epsilon}{(1+\epsilon)}) = \mathbb{P}(p_z > \frac{1}{(1+\epsilon)}) > 1 - \delta.$$

En analysant la signification du résultat de QMC dans le cas de non détection d'erreur, on remarque que q_z dépend, de l'existence du bug, et de la probabilité d'atteignabilité P_a par l'algorithme considéré du nœud défectueux représentant le bug. La petitesse de q_z ne signifie pas que la probabilité d'erreur $P_e = q_z/P_a$ est petite, car, selon l'algorithme utilisé, P_a peut être relativement petite, voir très petite, ou grande. L'utilisation d'algorithmes possédant de meilleures probabilités d'atteignabilité que RW, va permettre d'obtenir de meilleurs résultats : le nombre d'itérations (et donc le temps) sera plus petit et meilleure sera la signification de la petitesse de q_z estimée par OAA.

3.2.2.3 Test d'hypothèses et l'approche Monte Carlo Model Checking

Dans les graphes probabilistes, où les transitions suivent une distribution de probabilité donnée, le problème de vérification consiste à vérifier une propriété probabiliste ϕ de la forme $P_{>\theta}(\rho)$ [69], ce qui signifie que la propriété ρ, qui peut être déterministe ou elle même probabiliste, est vraie avec une probabilité $p = P(\rho) > \theta$ à partir d'un état donné s. Ce problème s'écrit dans le cadre du test d'hypothèses comme suit :

Problème de décision (1)
Décider entre les hypothèses H_0 et H_1 suivantes :
$H_0 : p > \theta$
$H_1 : p < \theta$

La façon classique (Wald's SPRT : sequentiel probability ratio test) de résoudre ce problème est de le transformer en le problème plus simple suivant :

Problème de décision (2)
Décider entre les hypothèses :
$H_0 : p = p_1$
$H_1 : p = p_0$

Classiquement, on fixe une zone d'indifférence (indécision) de largeur 2δ et on pose :

$$p_1 = \theta + \delta, \quad p_0 = \theta - \delta.$$

On se fixe deux bornes α et β. La première sur l'erreur de type I (false negative) induite lorsqu'on décide que ϕ est fausse (i.e. on décide H_1) alors qu'elle est vraie (i.e. H_0 est vraie). La deuxième concerne sur l'erreur de type II (false positive) qui consiste à décider que ϕ est vraie (i.e. on décide H_0) alors qu'elle est fausse (i.e. H_1 est vraie).

Notons que α et β peuvent être choisies simultanément petites. En général elle sont de l'ordre de 0.1 à 0.5 selon l'application.

La décision du problème (2) est faite comme suit :

On génère à chaque itération m, un échantillon supplémentaire (ou chemins à partir de l'état s). On note y_i l'observation de chaque itération i ($y_i = 1$ si ρ est vérifiée à travers le chemin i et $y_i = 0$ sinon). On calcule le rapport de vraisemblance :

$$RL(m) = \frac{L(y_1, ..., y_m, p_1)}{L(y_1, ..., y_m, p_0)},$$

où L est la vraisemblance (likelihood). Dans le cas où les y_i sont binaires de paramètre p_1 (ou p_0 selon l'hypothèse) et indépendants, on obtient :

$$L(y_1, ..., y_m, p_1) = p1^{d_m}(1 - p_1)^{m-d_m}, \quad L(y_1, ..., y_m, p_0) = p0^{d_m}(1 - p_0)^{m-d_m}.$$

où d_m est le nombre de 1 obtenus pour les y_i. Donc :

$$RL(m) = (p_1/p_0)^{d_m}((1 - p_1)/(1 - p_0))^{m-d_m}.$$

Notons que le calcul de $RL(m)$ est récursif sur m. En plus, on prend d'habitude le logarithme de la vraisemblance pour obtenir une récurrence additive.

Si $RL(m) > k_0 = (1 - \alpha)/\beta$, on décide H_0. Si $RL(m) < k_1 = \alpha/(1 - \beta)$, on décide H_1. Si $k_1 < RL(m) < k_0$ on continue : on prend un échantillon supplémentaire $m + 1$.

Dans le cadre Monte Carlo Model Checking, décrit dans la section précédente, on cherche à décider si la propriété déterministe ϕ est vérifiée ou non. On note les observations y_i : $y_i = 1$ si ϕ est vérifiée sur l'échantillon i (exécution de RW) et $y_i = 0$ sinon. On arrête le processus dès qu'on trouve un contre exemple $y_i = 0$. Soit m tel que $RL(m) > k_0$, donc la décision est faite comme suit :

- Dès qu'on trouve un contre exemple ($y_i = 0$), on s'arrête et on décide H_1
- sinon, si on atteint un nombre d'échantillons m (sans trouver un contre exemple), on décide H_0.

Notons que dans notre cas, $RL(m) = (p_1/p_0)^m$ car $d_m = m$, tous les y_i sont de valeur 1. Notons aussi que k_0 est approximativement égal à $1/\beta$, car α (et β d'ailleurs) est petit. D'où $(p_1/p_0)^m > k_0$ est équivalent à $m > log(k_0)/log(p_1/p_0)$.

Finalement, on pose $M = E[log(k_0)/log(p_1/p_0)]$ (E : partie entière).
- si on trouve un contre exemple, on décide que ϕ est fausse (H_1)
- sinon, si $m \leq M$ on continue de tirer des échantillons
- sinon (m atteint $M + 1$ sans trouver un contre exemple), on décide que ϕ est vraie (H_0)

3.2.2.4 Inconvénient de la marche aléatoire

Rappelons que la marche aléatoire (RW) n'utilise aucune mémoire. À chaque étape, elle ne garde qu'un seul état et ne préserve aucune information sur les états visités précédemment. On remarque dans ce cas que la probabilité d'exploration des différents états du graphe est loin d'être équilibrée, du fait que certains états sont beaucoup plus souvent visités que d'autres, pour diverses raisons :

- Si le graphe contient beaucoup de points morts ou beaucoup de retours arrières alors les états de petite profondeur sont plus souvent visités.
- Si la marche aléatoire atteint une composante fermée, elle continue à visiter seulement les éléments de cette composante.
- Il y a des structures particulières de graphes qui favorisent le déséquilibre dans la distribution des probabilités d'atteignabilité. C'est le cas par exemple de la structure en diamant fréquemment présente dans les graphes de model checking [56].

3.2.3 La marche aléatoire améliorée

Puisqu'elle n'utilise aucune mémoire, la marche aléatoire pure ne peut pas distinguer entre les états visités et non visités. Par conséquent, elle peut dépenser beaucoup de temps à revisiter les mêmes états (la redondance). Pour cela, la couverture du graphe entier ou d'une large portion de celui-ci demande un temps très grand (voir paragraphe 3.2.2.1). Plusieurs améliorations ont été proposées pour éviter ce problème en exploitant plus de mémoire et en utilisant différentes heuristiques qui servent à décider de la prochaine direction de l'exploration. L'efficacité de ces améliorations dépend de la bonne gestion de la mémoire, du temps consommé dans le calcul des heuristiques et leur indépendance de structures de données particulières. Les améliorations principales apportées dans la littérature peuvent être classifiées dans les directions suivantes :

3.2.3.1 Tehniques de remplacement en mémoire

Cette méthode est utilisée lorsque la mémoire disponible n'est pas suffisante pour stocker tout le graphe. Une partie seulement des états visités est gardée en mémoire, dite cache, pour éviter les revisites redondantes et améliorer la couverture. Elle consiste a supprimer certains états du cache et les remplacer par d'autres[25] [28]. Les algorithmes utilisant le caching diffèrent, essentiellement, en la stratégie d'ajout et de suppression des états dans le cache. A titre d'exemple, on peut décider que les états les plus fréquemment visités seront stockés dans le cache avec une grande probabilité. Les techniques de remplacement ont été utilisées principalement avec l'exploration déterministe en vu d'effectuer une couverture totale. Elles ont été utilisées également pourl'exploration probabiliste [64] pour une couverture partielle. Elles ont été montrées efficaces en pratique car elles tirent profit de toute la mémoire disponible tout en évitant de dépasser sa capacité.

3.2.3.2 Réinitialisation

Elle permet d'éviter la situation où la marche aléatoire est bloquée dans une petite composante fortement connexe. La marche sera arrêtée périodiquement et réinitialisée. La réinitialisation peut être faite à partir d'un état choisi aléatoirement dans la marche précédente et non pas nécessairement à partir de l'état initial. Ceci a l'avantage de minimiser la redondance et explorer les états profonds [32]. L'efficacité de cette méthode dépend du seuil de réinitialisation, i.e. le nombre d'étapes après lesquelles il faut réinitialiser.

3.2.3.3 Marche pseudo-parallèle

C'est l'idée la plus simple pour augmenter les performances de la marche aléatoire. Elle permet d'explorer plus d'états [62] et de minimiser la probabilité d'erreur [35]. Dans ce cas, il faut gérer un ensemble d'états courants et choisir à chaque étape leurs successeurs. Afin de minimiser les collisions entre les chemins parallèles, on a intérêt à ce que les marches soient assez éloignées selon une certaine métrique. La parallélisation de la marche aléatoire a été combinée au caching [64], à l'exploration en largeur [62] et à d'autres heuristiques.

3.2.3.4 Recherche exhaustive locale

Vu que dans la marche aléatoire, les fréquences des visites des nœuds sont non uniformes, on la combine avec la recherche exhaustive locale [62] pour mieux explorer certaines régions d'intérêt (régions denses ou régions proches d'un état cible) qui ne sont pas explorées facilement avec une simple marche aléatoire. On décide d'effectuer une recherche exhaustive si, selon certaines heuristiques, on arrive à déterminer qu'on est proche d'un état cible.

3.2.3.5 Guiding

Il s'agit d'utiliser une heuristique qui permet de décider de la prochaine direction de l'exploration. Par exemple, utiliser une métrique [46], une fonction de performance ou d'évaluation [29, 49, 11] ou encore des informations de structure [18], pour estimer la probabilité d'atteindre un état cible et décider du nœud successeur à visiter, quand utiliser une recherche exhaustive locale, ou que stocker dans le cache ?... Autres méthodes utilisent des échantillonnages (tests) répétés [11, 60] et à chaque itération, les vecteurs en entrée sont modifiés en se basant sur la valeur de la fonction de performance dans l'itération précédente. Les méthodes de guiding s'interessent particulièrement au erreurs du type débordement de buffers ou situation de blockage (deadlock). Ce type d'erreur peut être évalué par une fonction de performance qui prend ses valeures maximales (ou minimales) au voisinage des nœuds cibles et qui prend des valeurs d'autant plus grandes (ou plus petites) que le nœud cible est proche. Par exemple, dans le cas d'erreurs de type débordement de capacité d'un buffer, la fonction de performance est le nombre de données dans le buffer. Pour arriver à un état ou le nombre de données

dépasse la capacité du buffer, il faut suivre, à chaque sélection du successeur, les nœuds ayant la plus grande valeurs de la fonction de performance (nombre de données dans le buffers).

Hormis quelques résultats théoriques concernant la marche aléatoire, cités ci-haut, les travaux réalisés dans le cadre de la vérification randomisée sont généralement motivés par des intuitions et validés par des expérimentations sur des graphes réels [32, 49, 11]. Ce sont ces expérimentations qui montrent l'évidence empirique et l'efficacité de telles méthodes.

3.3 Conclusion

Nous avons focalisé notre attention, dans ce chapitre, sur les méthodes randomisées de vérification partielle, qui font l'objet de la thèse, et souligné leur importance et leur rôle prometteur dans la vérification de grands systèmes, vu leurs besoins réduits en mémoire. Nous avons étudié la marche aléatoire et son utilisation en model checking, en particulier l'approche Monte Carlo. Ensuite, nous avons examiné les différentes améliorations apportées à la marche aléatoire simple pour optimiser ses performances. Ces formes améliorées utilisent généralement une mémoire supplémentaire pour le stockage des états, ou des heuristiques permettant de guider l'exploration pour arriver plus rapidement à détecter les erreurs.

Chapitre 4

Modélisation de l'exploration randomisée

4.1 Introduction

Les méthodes randomisées d'exploration, mentionnées dans le chapitre précédent sont basées principalement sur la marche aléatoire. Les améliorations apportées à celle-ci portent sur deux plans : l'ajout de mémoire supplémentaire pour réduire la redondance et l'utilisation d'heuristiques pour guider l'exploration. Dans ce chapitre, nous proposons un schéma générique des algorithmes d'exploration et nous précisons ses différentes fonctions et paramètres. Nous utilisons ensuite ce schéma pour classifier les algorithmes existants et proposer des algorithmes d'exploration probabilistes plus adaptés qui réduisent considérablement la redondance et améliorent la couverture et l'atteignabilité des états.

4.2 Description du problème

D'une façon générale, un algorithme d'exploration vise à maximiser sous certaines contraintes un critère quantitatif $\mathcal{C}(par)$ dépendant de certains paramètres par. Le critère $\mathcal{C}$ pourrait être, par exemple, le taux de couverture τ_c (= nb. nœuds couverts /nb. nœuds du graphe).

4.2.1 Le problème de l'espace mémoire et du temps

On souhaite, naturellement, que l'algorithme d'exploration A arrive à accomplir sa tâche (i.e. explorer efficacement l'espace d'état avec une bonne valeur de $\mathcal{C}(par)$) rapidement et en utilisant des ressources mémoire raisonnables.

Soit N la taille, en états, de la mémoire disponible. Si cette mémoire est très grande ($N \rightarrow +\infty$) et peut, en particulier, contenir tout l'espace d'états ($|M| < N$), alors une exploration exhaustive déterministe par un algorithme classique de model checking

entraînera un temps d'exécution T_e optimal. Cependant, ceci n'est pas le cas en général et les systèmes ont des tailles dépassant les ressources mémoires.

D'autre part, l'algorithme de la marche aléatoire simple (RW), qui ne stocke en mémoire que le nœud courant, suffit à explorer tout le graphe avec une probabilité 1 (lorsque $T_e \rightarrow +\infty$). Cependant, afin de pouvoir décider le plus tôt possible du bon fonctionnement du système modélisé par le graphe, le temps d'exploration T_e ne doit pas dépasser, en pratique, une certaine durée. Ainsi, l'algorithme RW, pour son inefficacité en temps, est loin d'être suffisant.

La performance d'un algorithme d'exploration A se mesure, alors, principalement, par la valeur obtenue pour $\mathcal{C}_A$, le temps d'exécution T_e et la mémoire utilisée N. On peut espérer couvrir avec un algorithme randomisé un pourcentage du graphe ou le graphe entier en moins de temps qu'avec l'algorithme exhaustif qui se bloquera rapidement à cause de l'insuffisance de la mémoire.

D'une façon générale, on peut dire qu'à une mémoire N et un temps de calcul T_e correspond une valeur maximale $\mathcal{C}_{A,max}(N, T_e)$ du critère en question qu'un algorithme d'exploration ne peut dépasser. L'enjeux principale en exploration consiste à concevoir des algorithmes pouvant atteindre $\mathcal{C}_{A,max}(N, T_e)$ ou s'y approcher le plus possible.

4.2.2 Le problème de dépendance aux graphes

En réalité, le critère $\mathcal{C}_A$ dépend non seulement des paramètres mémoire et temps, mais aussi du graphe G sur lequel on teste cet algorithme [55] $\mathcal{C}_{A,G}(N, T_e)$. Un algorithme donné A ne peut atteindre la même valeur de $\mathcal{C}$ sur tous les graphes qu'il explore.

Pour assurer une crédibilité aux résultats des comparaisons, on est amené à étudier les algorithmes proposés théoriquement et expérimentalement sur des graphes ayant des caractéristiques différentes. En effet, les graphes étudiés seront caractérisés par un facteur de densité, et des graphes issues de systèmes réels, avec des caractéristiques différentes, seront expérimentés. Du coté théorique, deux types extrêmes de graphes seront considérés en plus de la caractérisation précédente : les arbres et les grilles.

4.3 Critères recherchés

En général, les performances des algorithmes d'exploration destinés à la vérification se mesurent par deux critères principaux : la couverture et l'atteignabilité.

4.3.1 La couverture

Ce critère exprime la capacité de l'algorithme à explorer l'espace d'états. Une bonne couverture reflète moins d'explorations redondantes. Si l'algorithme A arrive à couvrir

k nœuds distincts dans le graphe considéré G, en n étapes de son exécution, alors le taux de redondance est donnée par :

$$\tau_r = \frac{n-k}{n}.$$

En effet, un algorithme d'exploration, à chaque étape de son exécution, peut soit visiter un, et un seul, nouveau nœud, soit répéter la visite d'un nœud déjà visité. Dans le premier cas, et le temps n et le nombre de nœuds couverts k sont incrémentés de un. Dans le deuxième cas, le temps est incrémenté mais non pas le nombre de nœuds couverts, ce qui augmente la redondance.
L'étude de la couverture peut se faire de plusieurs façons :

- *Le temps moyen de couverture* $T_{A,G}(k)$: c'est le nombre moyen d'étapes n nécessaires à un algorithme donné A qui commence de l'état initial pour couvrir un nombre k de nœuds dans le graphe G. Cette définition sera utilisé pour le calcul théorique. En expérimentations, c'est le temps moyen (en secondes) nécessaire pour couvrir un certain pourcentage du graphe G par l'algorithme A.

- *Le taux de couverture* $\tau_c(A,G) = \frac{k}{|M|}$: c'est le rapport du nombre moyen des états k visités par un algorithme donné A, au nombre total des états du graphe G (i.e. $|M|$).

- *le nombre moyen de nœuds couverts* k : dans le cas de graphes de très grande taille, le nombre des états atteignables peut être inconnu. On considère alors *le nombre moyen de nœuds couverts* k, qui sera mesuré en fonction du temps, au lieu de considérer le taux de couverture $\tau_c(A,G)$.

4.3.2 L'atteignabilité

Ce critère reflète la possibilité d'atteindre les états du graphe G par l'algorithme A considéré, en particulier, l'atteinte des états présentant une erreur. On obtient des informations sur l'atteignabilité à travers les probabilités suivantes :

- *La probabilité d'atteignabilité* : Étant donné un graphe G et un algorithme d'exploration randomisé A, la suite ordonnée $\underline{v}_k = (v_0, v_1, ..., v_k)$ des états distincts de G, visités par A après n étapes est une variable aléatoire dont la probabilité sera notée $\mathbb{P}_{A,G}(\underline{v}_k, n)$. L'apparition d'un nœud donné v dans cette suite, autrement dit l'atteignabilité de v en n étapes, est une variable aléatoire dont la probabilité $\mathbb{P}_{A,G}(v,n) = \sum_{\underline{v}_k | v \in \underline{v}_k} \mathbb{P}_{A,G}(\underline{v}_k, n)$ diffère d'un nœud à un autre.

- *La probabilité minimale d'atteignabilité* : elle consiste en le minimum sur les nœuds du graphe G des probabilités décrites ci haut :

$$\pi_{min}(A,G,n) = \min_v \mathbb{P}_{A,G}(v,n)$$

– *Le temps de détection* : Un processus utilisé en pratique par l'approche MC^2 est d'arrêter l'algorithme d'exploration A puis le relancer à plusieurs reprises [34]. Étant donné un nœud cible v, ayant une probabilité d'atteignabilité π_v, le critère utilisé dans ce cas est le calcul du nombre de répétitions R nécessaires à l'algorithme A pour détecter ce nœud.

4.4 Schéma général des algorithmes d'exploration

Un algorithme d'exploration quelconque peut être présenté par le schéma générique de la figure 4.1. Dans ce schéma, P représente les paramètres d'entrée de l'algorithme, par exemple, la propriété à vérifier ϕ, le nombre des exécutions parallèles initiales dans le cas d'une marche aléatoire parallèle [62, 32], etc. Ce dernier paramètre, entre autre, peut être modifié durant l'exécution de l'algorithme selon les ressources disponibles et les besoins de l'exploration. I contient des informations globales sur la structure du graphe G, par exemple, le nombre moyen de successeurs des états, le nombre moyen de boucles, de deadlock,... etc. Notons que ce type d'informations est généralement collecté à la volée et utilisé pour guider et optimiser l'exploration.

```
//Structures de données :
P : Paramètres de l'algoritme (statique) ;
I : Informations sur le graphe (dynamique) ;
V : Ensemble des nœuds gardés en mémoire ;
v : nœud courant ;

//Initialisation :
V ← {v0} ;
v ← {v0} ;

//Corps de l'algorithme :
Tant que (¬ condition d'arrêt) faire
    | v ← sélectionner (V, P, I) ;
    | visiter(v) ;
    | (V, I) ← actualiser (V, v, P, I) ;
Fait
```

FIGURE 4.1 – Schéma général des algorithmes d'exploration

Un algorithme respectant cette forme générale, peut être vu comme une marche aléatoire simple dans un graphe de dimension N, où N est la taille de la mémoire. Une exécution de l'algorithme randomisé A sur le graphe original G correspond à une marche aléatoire simple sur le graphe, de dimension N, G^N défini comme suit : Un état de G^N est un N-uplet $V = (v_0, v_1, ..., v_N)$, où chaque v_i est un état de G stocké en

mémoire ou un emplacement vide noté 0. Il y a une transition $V = (v_0, v_1, ..., v_N) \rightarrow V' = (v'_0, v'_1, ..., v'_N)$ dans le graphe G^N ssi $\exists j$, $1 \leq j \leq N$, tel que $v'_j \in Succ(v_j)$ et $v_i = v'_i$, $\forall i \neq j$, $1 \leq i \leq N$. Pour $N = 1$, on se retrouve dans le cas d'une marche aléatoire simple dans le graphe G. Sinon, l'algorithme consiste en une marche aléatoire multidimensionnelle.

Selon le schéma précédent, un algorithme d'exploration est complètement défini en spécifiant la *condition d'arrêt* et les deux fonctions *sélectionner* et *actualiser*. Avec ces trois fonctions, on peut définir plusieurs variantes, incluant les algorithmes déterministes et probabilistes décrits dans la littérature, ainsi que ceux que nous allons proposer.

4.4.1 La condition d'arrêt

En pratique, les algorithmes d'exploration ne vont pas forcément jusqu'à la couverture totale du graphe, mais ils sont arrêtés lorsque la condition d'arrêt est vérifiée. La condition d'arrêt peut être, par exemple, l'épuisement de la mémoire, l'atteinte du pourcentage de couverture désiré ou d'un nœud cible.

4.4.2 La fonction "sélectionner"

Il s'agit de choisir, à chaque étape, le prochain nœud à visiter dans l'ensemble des successeurs des nœuds déjà visités et stockés dans V. La façon de choisir les nœuds peut être déterministe ou probabiliste selon une certaine distribution. Elle peut être aussi guidée par les paramètres P et les informations disponibles I. Par exemple, si l'état cible correspond à un débordement de capacité d'un buffer, on a intérêt à choisir le successeur de l'état courant qui a le plus grand nombre d'éléments dans ce buffer. Le choix du nœud suivant peut être influencé aussi par des informations locales comme le nombre de successeurs ou de boucles....

Comme mentionné dans l'introduction, notre intérêt dans les chapitres 5-8 porte principalement sur la stratégie d'exploration même, ce qui correspond à la fonction de sélection. Dans le chapitre 9, nous traiterons la fonction d'actualisation y compris la répétition avec ré-initialisation des algorithmes.

4.4.3 La fonction "actualiser"

C'est la mise à jour de la mémoire (l'ensemble V des nœuds visités) ainsi que des informations I collectées durant l'exploration, afin d'optimiser les ressources et permettre l'évolution de l'exploration dans le sens désiré. Par ailleurs, l'ordre et le format du stockage dans V jouent un rôle important dans l'efficacité de cette mise à jour [28]. La fonction d'actualisation s'intéressent aux stratégies de stockage et de remplacement des états en mémoire [28] [25]. L'introduction de ces stratégies dans notre schéma générique va être vue en détail dans le chapitre 9.2.

Pour les graphes de très grandes tailles, un algorithme randomisé d'exploration peut explorer, au mieux, le même nombre d'états qu'un algorithme déterministe, il sera ensuite arrêté à cause de l'insuffisance de la mémoire principale. Deux solutions se présentent alors pour permettre l'évolution d'un tel algorithme et l'exploration d'un espace d'états plus important que celui exploré classiquement par les algorithmes déterministes. La première est citée ci-dessus, elle consiste en les politiques de remplacement (caching), qui remplace des nœuds existants en mémoire par des nœuds nouvellement explorés. La deuxième, peut être considérée comme un cas particulier de la première. Elle consiste en la réinitialisation de l'algorithme à chaque fois que la mémoire est pleine. La réinitialisation est la méthode de caching la plus simple et la plus utilisée pour les algorithmes randomisés d'exploration [34, 11, 60]. Le nombre de répétition R nécessaires pour garantir une confidence donnée sera discuté en détail dans le chapitre 9.3.

4.5 Classification

Dans le schéma général ci-haut, l'algorithme de la marche aléatoire par exemple, a comme condition d'arrêt l'atteinte d'une feuille ou d'un nœud cible. La fonction *sélectionner* consiste en un choix aléatoire uniforme entre les successeurs du nœud courant, le seul stocké dans V. La fonction *actualiser* consiste simplement en le remplacement du nœud courant par celui choisi par la fonction de sélection. La marche aléatoire est généralement répétée un certain nombre de fois jusqu'à la détection d'une erreur.

Le tableau 4.1 donne une description sommaires des principaux algorithmes existant dans la littérature selon le schéma que nous avons présenté. La dernière colonne du tableau exprime le critère principal que chaque algorithme cherche à optimiser. En effet, model checking a deux buts : la vérification de software correct et la recherche d'erreurs dans les softwares erronés. Certaines techniques qui visent à réduire le problème de l'explosion de l'espace d'états se concentrent sur le premier but tandis que d'autres techniques s'intéressent au deuxième.

Algorithme	Critères	Condition d'arrêt	Sélection	Actualisation	Répétition
Exploration déterministe	Couverture	Couverture totale	FIFO ou LIFO	Ajout du nouvel état exploré	Pas de répétition
PRW -[35, 51] -[66, 34]	Couverture Atteignabilité	-Une certaine couverture (confidence) -Deadlock, bug, lasso	-Uniforme parmi les successeurs de l'état courant - "	-le nouvel état choisi remplace l'état courant - "	-Graphes particuliers, pas de répétition -Répété R fois
DRS [32]	Atteignabilité	Deadlock, une profondeur prédéfinie ou un état mort (dont tous les successeurs sont visités)	Basée sur RW	Ajout des états exploré ayant au moins deux successeurs non visités en mémoire. Suppression des états morts	Réinitialisation à partir d'un état choisi aléatoirement dans les marche précédentes
Heuristiques combinant RW-BFS [62]	Couverture	Atteinte d'un nombre de pas précisée par l'utilisateur	Basée alternativement sur RW et BBFS (Bounded BFS)	BBFS : ajout des états visité. RW : remplacement de l'état courant	Plusieurs RWs sont lancés en parallèle
Random BFS [64]	Couverture	Probabilité d'omission $\leq \epsilon$	FIFO	choisir aléatoirement un nombre fixe k de successeurs possibles et les placer dans la file	Pas de répétition
Recherche [46] guidée par la probabilité d'atteignabilité	Atteignabilité	aucune probabilité initiale n'est > 0, rencontre d'un état cible, deadlock	File prioritaire. Priorité calculé selon les probabilité d'atteignabilité	FIFO	Pas de répétition
Algorithmes de caching en général [28] [25]	Couverture	Couverture totale ou partielle	FIFO	Selon une certaine stratégie de remplacement dans la mémoire	Pas de répétition
Algorithmes de guiding en général [46, 49, 11, 29, 60]	Atteignabilité	détection d'un état cible	Basée sur une fonction de performance	Aucune	Répétition possible

TABLE 4.1 – Caractéristiques des principaux algorithmes d'exploration randomisée pour la vérification

Les algorithmes d'exploration que nous proposons constituent un cadre général des algorithmes existants. Nous nous concentrons particulièrement sur la fonction de sélection que nous enrichissons et étudions en détails théoriquement et expérimentalement. Nous étudions également la fonction d'actualisation où nous combinons différentes stratégies existantes pour l'exploration déterministe à notre approache probabiliste. Nous visons principalement la couverture, mais nous nous intéressons également à l'atteignabilité (approache MC^2).

4.6 Conclusion

Nous avons présenté dans ce chapitre les différentes contraintes rencontrant la vérification, comme la mémoire limitée, le temps de calcul, la grande dépendance des résultats aux graphes considérés ainsi que le problème de redondance ou de réexploration des états. Puis, nous avons défini les critères principaux qui vont nous servir dans les chapitres suivants à évaluer les méthodes de vérification que nous proposerons. Nous avons établi, ensuite, un schéma générique qui englobe plusieurs variantes d'algorithmes d'exploration existant dans la littérature du domaine. Grâce à ce schéma, une classification de ces algorithmes a pu être faite, ce qui nous a permis de bien cadrer notre thèse et d'identifier, les caractéristiques principales des algorithmes que nous y proposons.

Chapitre 5

Randomisation, Méthodologie d'étude, Résultats généraux

5.1 Introduction

Dans ce chapitre nous présentons des résultats généraux qui peuvent être exploités dans l'étude de tout algorithme d'exploration randomisé. En outre, ce chapitre trace une méthodologie d'étude générale de ces algorithmes. Les chapitres qui vont suivre, vont concrétiser cette méthodologie et ces résultats généraux en les appliquant sur des algorithmes particuliers formants un jeu de choix logique et complet parmi la liste des algorithmes randomisés listés dans la section 2 ci-dessous.

5.2 Randomisation de l'exploration

5.2.1 Une sélection randomisée

En se basant sur le schéma général présenté dans le chapitre précédent, plusieurs algorithmes d'exploration randomisés peuvent être proposés en combinant les méthodes de sélection randomisées aux stratégies de remplacement. Les algorithmes que nous proposons dans les chapitres 6 7 8 sont conçus en jouant sur la sélection seulement et non pas sur l'actualisation dans la mémoire des nœuds stockés (qui sera traitée au chapitre 9.2). Cette fonction de sélection est très riche et donne lieu à diverses possibilités : Il s'agit de choisir un ou plusieurs états dans le sous ensemble V des états visités, puis de choisir un ou plusieurs de leurs successeurs à vérifier. Notre but, ici, est de montrer que l'amélioration du schéma même de l'exploration randomisée (i.e. la fonction sélectionner) peut apporter des performances importantes. L'étude détaillée des stratégies de stockage et de remplacement dans la mémoire, une fois que celle-ci est pleine, est différée au chapitre 9.2. Ci-dessous, nous énumérons sommairement quelques méthodes possibles pour une sélection randomisée :

1. Choisir uniformément un fils du dernier nœud visité et s'arrêter si le nœud courant n'a plus de successeurs (deadlock), ce qui correspond à l'algorithme RW.

2. Choisir uniformément un nœud dans le sous ensemble V des états visités, puis choisir uniformément l'un de ses successeurs. Ce choix permet de diffuser l'exploration dans toutes les directions sans favoriser l'exploration en profondeur comme c'est le cas des algorithmes basés sur RW. Il donne à tous les nœuds visités une même probabilité d'avoir un successeur exploré à l'étape suivante. Cette sélection "uniforme" des prédécesseur constitue un algorithme intéressant qui va être étudié en détails par la suite (voir chapitre 7).

3. Une *feuille* f est un nœud n'ayant aucun successeurs explorés $\text{Succ}(f) \cap V = \emptyset$. Un *nœud interne* i est un nœud ayant des successeurs déjà explorés $\text{Succ}(i) \cap V \neq \emptyset$. Choisir aléatoirement un fils d'une feuille ou d'un nœud interne. La décision entre feuille et nœud interne s'effectue selon une probabilité prédéfinie donnée en paramètre. Cette méthode de sélection permet d'orienter l'exploration en profondeur ou en largeur selon le taux désiré. Son étude sera détaillée au chapitre 8.

4. Choisir aléatoirement un fils du nœud n de V qui a le plus de fils non visités ($\text{Succ}(n) \setminus V = Max_{v \in V} \text{Succ}(v) \setminus V$) ou qui a le moins de fils visités ($\text{Succ}(n) \cap V = Min_{v \in V} \text{Succ}(v) \cap V$). Ces méthodes de sélection utilisent les informations disponibles pour mieux distribuer l'exploration sur les nœuds du graphe.

5. Choisir aléatoirement un petit fils d'un nœud visité, sélectionné selon l'une des façons décrite ci-dessus. Il s'agit d'appliquer la fonction *succ*() plusieurs fois consécu- tives et retenir dans V le nœud résultant sans retenir ses prédécesseurs, ce qui donne lieu aux "Algorithmes à sauts" qui. Ces algorithmes visent à atteindre de grandes profondeurs dans les graphes explorés.

6. Garder à chaque instant deux nœuds courants ou plus, et choisir les successeurs selon l'une des méthodes précédentes, ce qui amène à des "Algorithmes pseudo-parallèles".

7. L'exploitation des informations locales des états dans le choix des successeurs conduit aux "Algorithmes guidés". L'algorithme guidé effectue sa sélection en tenant compte des informations disponibles comme la taille et le contenu des cannaux ou buffers (voir section 3.2.3.5, 33).

5.2.2 Condition d'arrêt ressource-dépendante

Comme nous l'avons noté précédemment, nous nous concentrons sur la randomisation de la sélection des successeurs. Pour simplifier l'étude des algorithmes proposés, la condition d'arrêt considérée sera simple et efficace. Il s'agit de l'épuisement de la mémoire principale. Une propriété principale de notre condition d'arrêt est, donc, l'utilisation explicite du paramètre N qui présente le nombre maximum d'états pouvant être stockés en mémoire à chaque instant. De ce fait, nos algorithmes sont mémoire dépendants.

5.2.3 Actualisation par réinitialisation

La fonction d'actualisation se contentera d'ajouter chaque état, nouvellement exploré, à l'ensemble V jusqu'à épuisement de l'espace de stockage. Dans ce cas, la mémoire est vidée et l'algorithme est relancé à partir du nœud initial. Ceci est répété un certain nombre R de fois. Il s'agit d'une stratégie de remplacement aussi simple et efficace [66, 34] comme nous allons le montrer dans notre étude. Cette procédure de répétition rentre aussi dans le cadre d'une approche Monte Carlo qui permet de déterminer le nombre de répétitions nécessaires à la détection d'une erreur avec une confiance donnée. Cet aspect sera étudié au chapitre 9.3.

Notons que la condition d'arrêt et l'actualisation considérées dans cette partie (épuisement de la mémoire et réinitialisation) constituent un cas particulier des techniques de remplacement. Cette restriction de la fonction "actualiser" en un premier temps est considérée essentiellement pour permettre une étude théorique détaillée des différents algorithmes que nous proposons. Dans ce même cadre, nous avons fourni également une étude expérimentale qui considère la même condition d'arrêt et nous avons observé la cohérence des résultats obtenus dans les deux cas (voir les chapitres 6 7 8). Par la suite (chapitre 9.2), nous considérons des stratégies de remplacement plus générales que nous analysons expérimentalement.

Dans la suite, la version non répétée de l'algorithme randomisé sera noté A et sa version répétée sera notée RA. Les algorithmes que nous proposons vont être étudiés en détails théoriquement et expérimentalement. Nous allons considérer deux situations dans notre étude. La première, lorsque la mémoire est suffisante pour contenir tout l'espace d'état. Nous traitons alors la version sans répétitions A de l'algorithme où nous montrons l'importance de concevoir une bonne fonction de sélection. Dans le deuxième cas, qui est plus réaliste, notamment pour les exemples industriels de grande taille, nous considérons que la mémoire principale est insuffisante pour contenir tout l'espace d'états et que A est exécuté plusieurs fois avec réinitialisation. Dans ce cas, c'est l'algorithme RA qui sera étudié.

5.3 Étude théorique

Nous effectuons une analyse théorique des algorithmes A et RA en terme de différentes statistiques sur deux classes de graphes. Les mêmes statistiques seront mesurées expérimentalement sur des graphes de model checking. Les résultats théoriques et expérimentaux seront montrés cohérents. A l'issu de cette étude, l'efficacité de nos algorithmes sera confirmée, ainsi que leur souplesse, notamment la version paramétrée (voir chapitre 8).

5.3.1 Choix des graphes

En pratique, il y a plusieurs types de graphes, et un algorithme donné se comporte différemment selon la forme du graphe à explorer. Pour pouvoir calculer des formules

analytiques précises décrivant le comportement de ces algorithmes, on a eu recourt à analyser des classes particulières de graphes : les arbres et les grilles. Ces graphes particuliers sont convenables [56], pour l'étude analytique du comportement des algorithmes d'exploration et ce pour différentes raisons :

- La présence de composantes de structures particulières [56], en particuliers les grilles et les chaînes est fréquente dans les graphes de model checking.
- En manipulant ces graphes particuliers, on peut calculer des mesures probabilistes d'atteignabilité, ce qui est pratiquement impossible pour des graphes quelconques. En effet, dans un arbre ou une grille, on connait chaque nœud par sa position et on peut calculer, en fonction de cette position, tous les chemins qui mènent à un nœud donné et les probabilités de ces chemins.
- En contrôlant les deux paramètres de ces graphes (degré et profondeur), on peut obtenir des graphes larges (grand degré, petite profondeur) ou maigres (petit degré, grande profondeur) et caractériser les graphes étudiés selon ces paramètres. Ceci permettra d'orienter nos explorations, comme dans le chapitre 8.
- Les arbres et les grilles constituent deux cas extrêmes de graphes. Dans les arbres, il n'y a aucune intersection entre les successeurs (chaque nœud est successeur d'un seul nœud). Dans le cas de la grille, il y a des intersections entre tous les successeurs (chaque nœud a d parents, où d est le degré de la grille).

Les graphes considérés dans l'étude expérimentale parviennent d'exemples de systèmes de Model Checking. La différence entre les graphes considérés dans les deux cas, en particulier l'utilisation d'arbres et de grilles dans l'étude théorique n'affecte pas la validité des résultats obtenues. En effet, la cohérence des résultats théorique sur les mesures du temps moyen de couverture et du nombre moyen de nœuds couverts donnent plus de poids à ces mêmes mesures calculées expérimentalement.

Nous avons considéré dans notre étude théorique des graphes de tailles moyennes (plusieurs milliers). Ceci est fréquent dans la littérature [35, 38] car il permet de calculer des moyennes ou des mesures expérimentales. Dans notre cas, l'utilisation de graphes de grandes tailles engendre un temps de calcul très lourd vu que nos formules analytiques sont récursives.

La taille de la mémoire considérée pour le stockage est néanmoins une proportion de la taille de l'espace d'états. Les résultats obtenues peuvent être scalées sur des échelles plus grandes avec les mêmes proportions taille mémoire/ taille du graphe. En effet, le problème principale n'est pas la taille de l'espace d'états même, mais revient plutôt à l'insuffisance de la mémoire et peut être considéré pour des graphes moyens. Notons que dans notre étude expérimentale, nous utilisons des graphes de plusieurs millions d'états.

Pour l'étude des algorithmes RA, l'espace de stockage utilisé vaudra 10%, 15% et 20% de la taille du graphe dans chaque cas. Ces valeurs sont assez petites et permettent de se mettre dans un cas réel d'insuffisance de mémoire. Elles permettent aussi de

donner des ordres de grandeur sur le seuil de mémoire qui permet une bonne exploration. Les résultats obtenus dans ces prototypes peuvent être généralisés à des graphes plus grands avec les mêmes proportions de la taille mémoire à la taille du graphe.

5.3.2 Résultats généraux

Dans notre analyse théorique, nous explorons, dans l'ordre, le calcul des probabilités de couverture et d'atteignabilité afin de pouvoir mener le calcul du temps moyen de couverture et du nombre moyen de nœuds couverts. Et ce, pour les deux classes de graphes citées ci-dessus. Les formules théoriques seront énoncées sous leur forme générale dans ce chapitre. Les précisions et les détails dépendent de l'algorithme (sélection) utilisé et seront donnés dans les chapitres suivants.

Nous commençons par présenter la forme de la récurrence élémentaire exprimant la probabilité $\mathbb{P}_A(\underline{v}_k, n)$ de couvrir une suite ordonnée de k nœuds $\underline{v}_k = (v_1, ..., v_k)$ en n étapes par l'algorithme A en fonction d'autres probabilités élémentaires associées à l'étape $(n-1)$, en ne faisant intervenir que les vecteurs $\underline{v}_k$ et $\underline{v}_{k-1} = (v_1, ..., v_{k-1})$.

La récurrence élémentaire s'écrit en fonction d'un certain nombre d'attributs (qui seront détaillés dans les chapitres suivants). Soit $\mathcal{A}$ l'ensemble de ces attributs. La forme de la récurrence élémentaire sera la suivante :

$$\mathbb{P}_A(\underline{v}_k, n, a) \;=\; \sum_{b\in\mathcal{A}} \left[\alpha(\underline{v}_k, b)\mathbb{P}_A(\underline{v}_k, n-1, b) + \beta(\underline{v}_k, b)\mathbb{P}_A(\underline{v}_{k-1}, n-1, b)\right] \;\; \forall a \in \mathcal{A}$$

où les $\alpha(\underline{v}_k, b)$ et $\beta(\underline{v}_k, b)$ sont des coefficients dépendants de $\underline{v}_k$ et des attributs b.

Dans cette récurrence, on observe deux termes : Le premier sera dit *terme de redondance* et noté $\mathbb{P}^{\mathcal{R}}_A(\underline{v}_k, n, a) = \sum_{b\in\mathcal{A}} \alpha(\underline{v}_k, b)\mathbb{P}_A(\underline{v}_k, n-1, b)$. Il correspond au cas où le nœud choisi à l'étape n a déjà été exploré, et le nombre total de nœuds visité n'augmente pas. Le coefficient $\alpha(\underline{v}_k, b)$ correspond à la probabilité de cette situation pour un attribut b donné. Le deuxième est *un terme d'innovation* $\mathbb{P}^{\mathcal{I}}_A(\underline{v}_k, n, a) = \sum_{b\in\mathcal{A}} \beta(\underline{v}_k, b)\mathbb{P}_A(\underline{v}_{k-1}, n-1, b)$. Il correspond au cas où à l'étape n, le $k^{ième}$ nœud vient d'être exploré et de n'avoir couvert à l'étape $n-1$ que les $(k-1)^{ième}$ premiers éléments $\underline{v}_{k-1}$. Ceci provient avec probabilité $\beta(\underline{v}_k, b)$ pour chaque valeur de l'attribur b.

Ces récurrences seront, ensuite, précisées et simplifiées. Soit $\mathcal{V}_k$ l'ensemble de tous les vecteurs $\underline{v}_k$ et soit $\mathcal{S}_k = \{S^1_k, ..., S^i_k, ...\}$ une famille de sous ensembles de $\mathcal{V}_k$ vérifiant $\cup_i S^i_k = \mathcal{V}_k$ et $\cap_i S^i_k = \emptyset$. L'obtention de formules récursives simples pour un graphe donné, n'est possible que lorsque la sommation de la récurrence élémentaire peut se faire par sous ensembles S^i_k de vecteurs $\underline{v}_k$ convenablement choisis. Les coefficients de la récurrence élémentaire $\alpha(v_k, b)$ et $\beta(v_k, b)$ doivent être constants sur chaque ensemble S^i_k, et l'ensemble des $\underline{v}_{k-1}$, où $\underline{v}_k \in S^i_k$, doit être facile à identifier.

Par exemple, dans le cas des arbres, les récurrences simplifiés sont obtenus par sommation de l'équation élémentaire sur les ensembles S_k^i de suites $\underline{v}_k$ ayant à chaque niveau $j = 1, ..., h$, de l'arbre, k_j nœuds. Dans ce cas, les probabilités de couverture dépendent uniquement du nombre de nœuds de $\underline{v}_k$ dans chaque niveau de l'arbre et non pas de $\underline{v}_k$ lui même. Les coefficients $\alpha(v_k, b)$ et $\beta(v_k, b)$ sont donc constant sur ces ensembles. L'ensemble $\underline{v}_{k-1}$, correspond à $\underline{v}_k$ avec un nœud de moins dans un certain niveau j de l'arbre.

5.3.3 Cas des arbres

On considérera le cas d'un arbre m-aire complet de profondeur h. Les feuilles sont les nœuds du dernier niveau h de l'arbre. Chaque nœud du niveau $j < h$ a m successeurs et chaque chemin du nœud initial vers une feuille est de longueur h. Rappelons que n présente le nombre des étapes successives dans une exécution de l'algorithme A.

5.3.3.1 Probabilité de couverture

Pour les récurrences simplifiée, on considère le vecteur $\underline{K}_n = (K_n^1, ..., K_n^h)$ de variables aléatoires présentant le nombre de nœuds explorés à chaque niveau $j = 1, ..., h$, de l'arbre à l'étape n. Soit $\underline{k}_n = (k_1, ..., k_h)$ un vecteur d'entiers. La probabilité de couvrir un nombre k_j de nœuds à chaque niveau j de l'arbre est exprimée comme suit :

$$\mathbb{P}_A(\underline{K}_n = \underline{k}, a) = \sum_{b \in \mathcal{A}} \left[\alpha(\underline{k}, b) \mathbb{P}_A(\underline{K}_{n-1} = \underline{k}, b) + \sum_{j=1}^{h} \beta_j(\underline{k}, b) \mathbb{P}_A(\underline{K}_{n-1} = \underline{k} - 1_j, b) \right] \ \forall a \in \mathcal{A}$$

où $\underline{k} - 1_j = (k_1, ..., k_j - 1, ..., k_h)$, $1 \leq j \leq h$.

Soit K_n la variable aléatoire représentant le nombre de nœuds couverts à l'étape n. La probabilité de couvrir un nombre k de nœuds à l'étape n est donnée par :

$$\mathbb{P}_A(K_n = k) = \sum_{|\underline{k}|=k} \sum_{a \in \mathcal{A}} \mathbb{P}_A(\underline{K}_n = \underline{k}, a)$$

Comme décrit ci-dessus, on dfinit la probabilité de redondance et la probabilité d'innovation comme suit :

$$\mathbb{P}_A^{\mathcal{R}}(\underline{K}_n = \underline{k}, a) = \sum_{b \in \mathcal{A}} \alpha(\underline{k}, b) \mathbb{P}_A(\underline{K}_n = \underline{k}, b), \ \ \mathbb{P}_A^{\mathcal{I}}(\underline{K}_n = \underline{k}, a) = \sum_{b \in \mathcal{A}} \beta(\underline{k}, b) \mathbb{P}_A(\underline{K}_n = \underline{k}, b)$$

5.3.3.2 Temps moyen de couverture

Ces récurrences servent aussi à calculer le temps moyen de couverture $T_A(k)$ de k nœuds distincts. Le temps moyen $T_A(\underline{k})$ de couvrir le vecteur $\underline{k}$ par un algorithme A peut être exprimé en fonction des probabilités d'innovation comme suit :

$$T_A(k) = \sum_{|\underline{k}|=k} T_A(\underline{k}), \ \ T_A(\underline{k}) = \sum_{n=k}^{\infty} n P_A^{\mathcal{I}}(\underline{K}_n = \underline{k})$$

5.3.3.3 Probabilité d'atteignabilité

Les formules récursives de la probabilité d'atteignabilité par l'algorithme A seront également calculées. Soit $\mathbb{P}_A(i; K_n = k)$, la probabilité d'atteindre un nœud se situant au niveau i de l'arbre en n étapes et avoir couvert k nœuds à cette étape. Soit N le nombre maximal de nœuds pouvant être couvert par l'algorithme A. Notons que N peut être la taille de la mémoire ou un autre paramètre fixé a priori. La probabilité d'atteignabilité $\mathbb{P}_A(i, n)$ d'un nœud du niveau i en n étapes sera alors :

$$\mathbb{P}_A(i, n) = \sum_{k \leq N} \mathbb{P}_A(i; K_n = k)$$

On pose $\mathbb{P}^*_A(i, n)$ La probabilité d'atteindre un nœud se situant au niveau i en n étapes et avoir un nombre de nœuds couverts $k = N$. Elle s'écrit :

$$\mathbb{P}^*_A(i, n) = \sum_{k=N} \mathbb{P}_A(i; K_n = k)$$

5.3.3.4 Le nombre moyen de nœuds couverts par RA

On revient par la suite à l'étude des algorithmes RA. Soit N son seuil de réinitialisation. Le critère considéré ici est le nombre moyen de nœuds couverts en fonction du temps. Il peut être calculé en se basant sur les probabilités d'atteignabilité mentionnées précédem- ment. On note la relation du lemme 5.1, ci-dessous, entre la probabilité $\mathbb{P}_A(K_n = k)$ de couvrir k nœuds en n étapes et la probabilité d'atteignabilité $\mathbb{P}_A(v; K_n = k)$ d'avoir, en n étapes, atteint le nœud v et couvert k nœuds. Cette relation est valable pour tout algorithme A sur tout graphe G.

Lemme 5.1

$$\mathbb{P}_A(K_n = k) = \frac{1}{k} \sum_{v \in G} \mathbb{P}_A(v; K_n = k)$$

Preuve On dénote par Ω^k_A l'ensemble des suites $\underline{w}$ de longueur k qui peuvent être couvertes par l'algorithme A. Soit $1_{\underline{w}}$ la fonction caractéristique de $\underline{w}$: $1_{\underline{w}}(v) = 1$ si $v \in \underline{w}$ et $1_{\underline{w}}(v) = 0$ sinon. Notons que $\sum_{v \in G} 1_{\underline{w}}(v) = k$ pour tout $\underline{w} \in \Omega^k_{A,n}$. Donc,

$$\begin{aligned} \mathbb{P}_A(K_n = k) &= \sum_{\underline{w} \in \Omega^k_A} \mathbb{P}_A(\underline{w}, n) = \sum_{\underline{w} \in \Omega^k_A} \frac{\sum_{v \in G} 1_{\underline{w}}(v)}{k} \mathbb{P}_A(\underline{w}, n) \\ &= \frac{1}{k} \sum_{v \in G} [\sum_{\underline{w} \in \Omega^k_A} 1_{\underline{w}}(v) \mathbb{P}_A(\underline{w}, n)] = \frac{1}{k} \sum_{v \in G} \mathbb{P}_A(v; K_n = k) \end{aligned}$$

□

Donc, on a le théorème suivant qui donne la formule de $NC_{RA}(n)$, le nombre moyen de nœuds couverts par RA en n etapes :

Théorème 5.1 *Soit N le seuil de réinitialisation de l'algorithme A. Le nombre moyen $NC_{RA}(n)$ de nœuds couverts par RA en fonction du nombre d'étapes n est donné par :*

$$\begin{aligned} NC_{RA}(n) &= \sum_{i=0}^{h} m^i \ \mathbb{P}_{RA}(i,n), \ \ avec : \\ \mathbb{P}_{RA}(i,n) &= \mathbb{P}_A(i,n) + \sum_{n_1=N}^{n} [\mathbb{P}^*_A(i,n_1) + (1-\mathbb{P}^*_A(i,n_1)) \ \mathbb{P}_{RA}(i,n-n_1)] \end{aligned}$$

Preuve On a :

$$\begin{aligned} NC_{RA}(n) &= \sum_k k\mathbb{P}_{RA}(K_n=k) = \sum_k k \sum_{v\in G} \mathbb{P}_{RA}(v;K_n=k) \ \text{ d'aprs le lemme 5.1} \\ &= \sum_{v\in G}\sum_k k\mathbb{P}_{RA}(v;K_n=k) = \sum_{v\in G} \mathbb{P}_{RA}(v,n) \end{aligned}$$

Dans le cas d'un arbre m-aire complet, les nœuds v de G sont organisés en niveaux. Il existe m^i nœud dans chaque niveau i de l'arbre. On obtient alors :

$$NC_{RA}(n) = \sum_{i=0}^{h} m^i \ \mathbb{P}_{RA}(i,n)$$

Il reste de montrer la seconde égalité qui est une expression récursive de $\mathbb{P}_{RA}(i,n)$ en $\mathbb{P}^*_A$ et $\mathbb{P}_A$. Dans cette expression, le terme $\mathbb{P}_A(i,n)$ correspond au cas où aucune répétition n'a eu lieu durant le temps n. Quand à la somme en n_1, elle correspond aux cas de réinitialisations, dont la première se produit à l'étape n_1. Donc, il y a deux possibilités : 1. le nœud i est atteint avant l'étape n_1, qui a la probabilité $\mathbb{P}^*_A(i,n_1)$ d'apparaître. 2. Le nœud i n'est pas atteint en n_1 étapes, et doit être atteint après, dans les $n-n_1$ temps restant, ce qui donne une probabilité $(1-\mathbb{P}^*_A(i,n_1))\mathbb{P}_{RA}(i,n-n_1)$. □

5.3.4 Cas des grilles

On se placera dans une seconde étape dans le contexte d'une grille multidimentionnelle. Comme dans la section précédente, on est intéressé par le calcul efficace de certaines mesures comme le temps de couverture moyen et le nombre moyen de nœuds couverts pour l'algorithme A. On va analyser ces mesures en se basant sur les récurrences fondamentales.

Dans le cas des grilles, il est difficile de sommer (et donc simplifier) la récurrence élémentaire satisfaite par un algorithme A. Cette difficulté est due essentiellement au grand taux d'intersections. En effet pour une grille de dimension d, un nœud peut être successeur de d nœuds différents, ce qui constitue l'*intersection* (confluence), et non pas

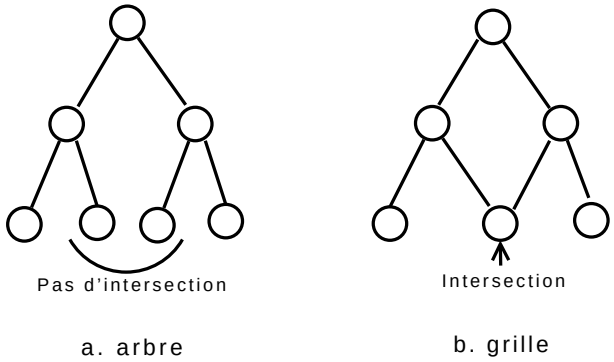


FIGURE 5.1 – Exemple d'intersection dans la grille

d'un seul nœud comme dans le cas des arbres (voir figure5.1).

Et pourtant, ces intersections sont la même raison pour laquelle cette récurrence est utilisable en pratique pour calculer des mesures exactes de l'exploration dans ce cas, spécialement en entreprenant certaines organisations expliquées ci-après. En effet, à cause des intersections, le nombre des suites ordonnées $\underline{v_k}$, de k nœuds distincts, générées par un algorithme A est raisonnable dans plusieurs cas d'étude. Notons aussi que la taille des grilles qui vont être considérées dans notre étude théorique est en général petite ou moyenne.

Dans le but d'utiliser facilement les récurrences élémentaires et calculer des mesures exactes pour un algorithme A dans le cas d'une grille multidimentionnelle, certains arrangements sont nécessaires. Premièrement, les suites $\underline{v_k}$ de longueur k sont obtenues récursivement, chacune en ajoutant un nœud à une suite $\underline{v_{k-1}}$ de longueur $k-1$ qui reste enregistrée en mémoire jusqu'à la génération de des nœuds v_k. Deuxièmement, les coefficients des équations élémentaires, sont enregistrés et incrémentés quand un nouveau nœud v_k est considéré et donc les probabilités sont mises à jour. Troisièmement, soit $C(v) = \{w \in G | w \in \text{Succ}(v)\}$ l'ensemble des successeurs (*Children*) du nœud v et soit $F(v) = \{w \in G | v \in \text{Succ}(w)\}$ l'ensemble des prédécesseurs (*Fathers*) du nœud v. Pour obtenir rapidement les ensembles $C(w)$ et $F(w)$, on doit indexer efficacement les nœuds de la grille (voir figure5.2). Supposons que la grille soit de dimension d et admet L nœuds sur chaque coté, le nombre total de nœuds est alors $N = L^d$. Ces nœuds seront indexés de 0 à $N-1$ comme suit : si v est un nœud de coordonnées $(x_1, ..., x_d)$, $0 \leq x_i \leq L-1$, alors il aura l'indice $ind(v) = \sum_{i=1}^{d} x_i L^{i-1}$. Par conséquent, on obtient le lemme suivant :

Lemme 5.2 *Soit v un nœud de la grille G et soient x_i, $i = 1, ..., d$ ses coordonnées. Donc, ayant seulement l'indice de v, $q = ind(v)$, on peut calculer les ensembles $\mathcal{I}_{in}(q) = \{i, x_i \geq 1\}$ et $\mathcal{I}_{out}(q) = \{i, x_i \leq L-2\}$ en $2d$ opérations.*

Preuve Puisqu'ils appartiennent à $\{0, ..., L-1\}$, les x_i sont les coefficients de la décomposition L-aire de l'entier q. Ceci peut être calculé comme suit : pour $i = d, ..., 1$, $x_i = E(q/L^i)$ et $q := q - x_i L^i$. □

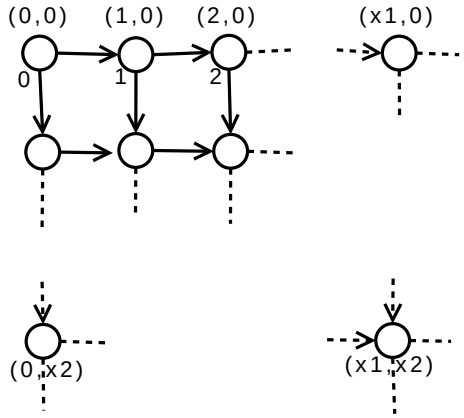


FIGURE 5.2 – Indexation des nœuds de la grille

Notons que si $i \in \mathcal{I}_{in}(q)$, alors on peut translater le nœud v selon la $i^{ième}$ dimension dans le sens qui fait décroitre la coordonnée x_i. Ceci donnera lieu à un prédécesseur de v selon cette dimension (voir théorème5.2). Autrement dit, l'ensemble $\mathcal{I}_{in}$ corresponds au dimensions selon lesquelles le nœud v a un prédécesseur. Symétriquement, les indices de $\mathcal{I}_{out}(q)$ aboutiront (voir théorème5.2) aux successeurs de q.

En utilisant le lemme 8.1, on obtient les résultats suivants concernant le calcul de $C(q)$ et $F(q)$:

Théorème 5.2 *Les ensembles de successeurs (fils) et de prédécesseurs (parents) d'un nœud v d'indice q sont donnés respectivement par :*

$$C(q) = \{q + L^{i-1},\ i \in \mathcal{I}_{out}(q)\},\quad F(q) = \{q - L^{i-1},\ i \in \mathcal{I}_{in}(q)\}$$

Le degré sortant (resp. entrant) de q est alors $|C(q)| = |\mathcal{I}_{out}(q)|$ (resp. $|F(q)| = |\mathcal{I}_{in}(q)|$).

En plus, pour vérifier facilement si ces successeurs/ prédécesseurs sont couverts, les suites sont stockées d'une façon stratifiée en respectant leur niveau dans la grille : si v_i est stocké dans le niveau l et le nœud ajouté v_k est fils de v_i, alors v_k doit être stocké dans le niveau $l+1$ suivant.

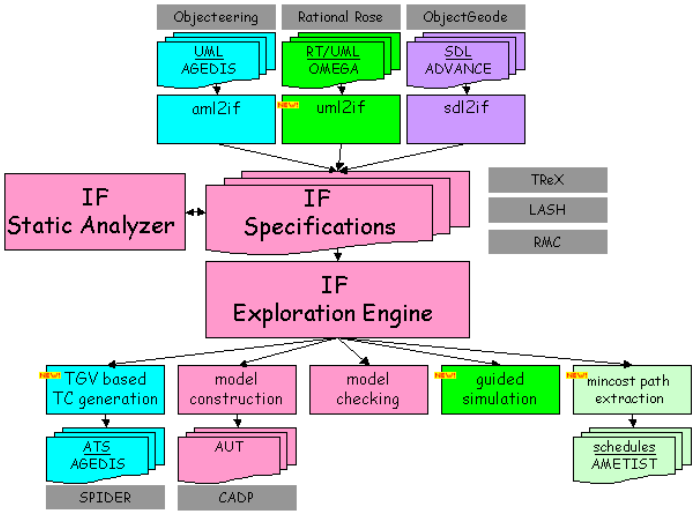


FIGURE 5.3 – Architecture de IF

5.4 Étude expérimentale

5.4.1 La plate forme IF

Nous avons implémenté les algorithmes sur le model-checker IF [6]. IF est une plate-forme à outils pour les systèmes distribués, développée par l'équipe DCS du laboratoire Verimag. Elle contient plusieurs composants de vérification : elle est dotée d'une série de plug-in de génération de test et contient des outils d'analyse statique permettant la simplification et l'optimisation des spécification, des simulateurs, un compilateur pour SDL, model-checkers et outils de comparaison de modèles, basé sur la forme intermédiaire IF. En plus, elle produit un debugger interactif et un générateur de modèle exhaustif permettant plusieurs stratégies d'exploration i. e. breadth or depth first, avec ou sans réduction d'ordre partiel, une présentation de temps discrète ou symbolique, etc. La plate-forme produit aussi différentes Interfaces de Programmation d'Applications (APIs) et en particulier, l'API d'exploration de l'espace d'état qui contient les fonctionnalités basiques nécessaires pour tout algorithme d'exploration avec des primitives pour le calcul direct des successeurs d'un état donnée.

5.4.1.1 Comment analyser une specification IF

Construction du simulateur :

```
$ make token.x
```

Exploration du modèle :

```
$ token.x inter
$ token.x -random
```

Construction du modèle :

```
$ token.x [-bfs|-dfs] -t token.aut -q token.states
```

Optimisation :

```
$ dfa -live token.if > token.live.if
```

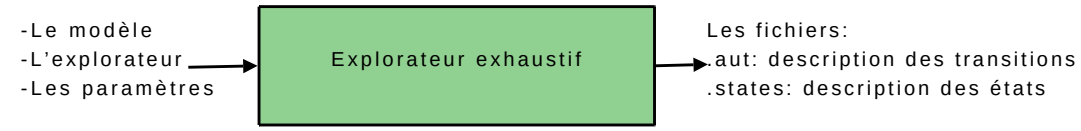


FIGURE 5.4 – Les entrées/ sorties

5.4.1.2 Exemple du Producteur/ Consommateur

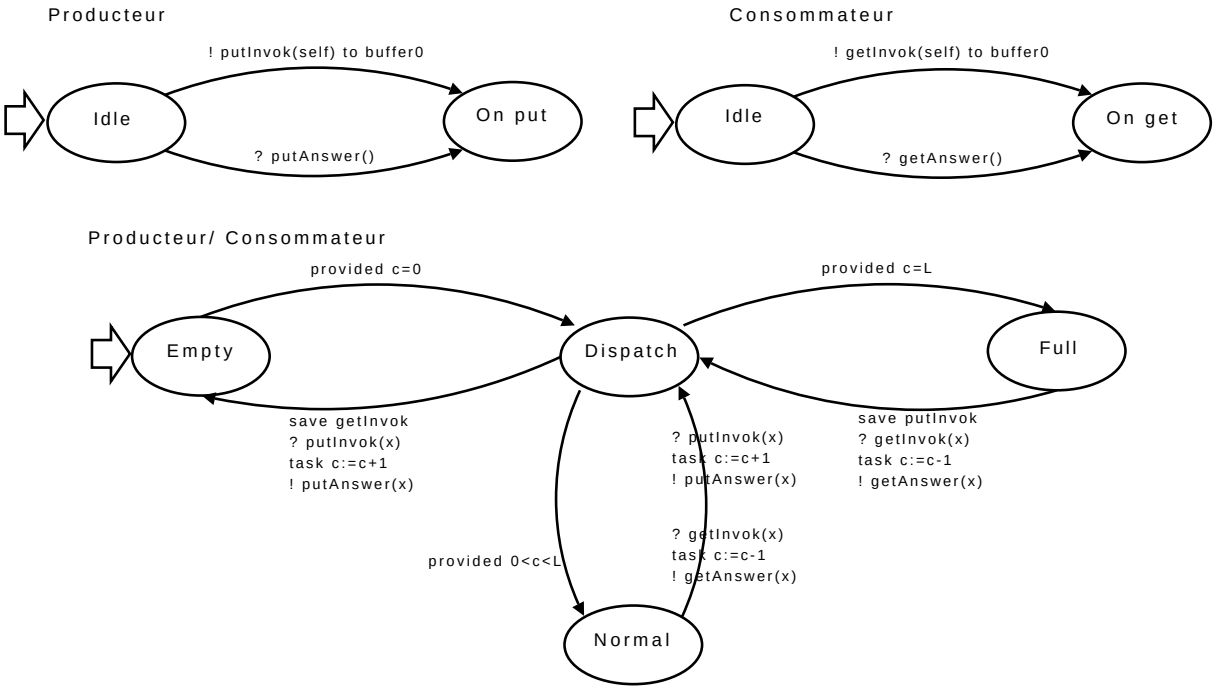


FIGURE 5.5 – Le modèle du producteur/ consommateur

Le modèle du producteur/ consommateur est défini dans la figure5.5. Dans le cas de 2 producteurs et 4 consommateur, un état est décrit comme suit :

```
41100 :
{producer}0 {putAnswer{}}
  @onPut {} {}
{producer}1 {}
  @onPut {} {}
{consummer}0 {}
  @onGet {} {}
{consummer}1 {}
  @onGet {} {}
{consummer}2 {getAnswer{}}
  @onGet {} {}
{consummer}3 {}
  @idle {} {}
{buffer}0            {getInvoke{{consummer}1},getInvoke{{consummer}0},
                      putInvoke{{producer}1}}
  @empty {0,{consummer}3} {}
```

La description des transitions est la suivante :

```
(0,"{producer}0 !putInvoke{{producer}0} ",1)
(0,"{producer}1 !putInvoke{{producer}1} ",2)
(0,"{consummer}3 !getInvoke{{consummer}3} ",3)
(0,"{consummer}2 !getInvoke{{consummer}2} ",4)
(0,"{consummer}1 !getInvoke{{consummer}1} ",5)
(0,"{consummer}0 !getInvoke{{consummer}0} ",6)
(1,"{producer}1 !putInvoke{{producer}1} ",7)
(1,"{buffer}0 ?putInvoke{{producer}0} {buffer}0 !putAnswer{} ",8)
(1,"{consummer}3 !getInvoke{{consummer}3} ",9)
(1,"{consummer}2 !getInvoke{{consummer}2} ",10)
...
```

5.4.1.3 Les modules de l'explorateur exhaustive

```
// explorator : defines abstract graph exploration
class IfExplorator : public IfDriver {

// bfs_explorator : breadth-first search exploration
class IfBfsExplorator : public IfExplorator {

// dfs_explorator : depth-first search exploration
class IfDfsExplorator : public IfExplorator {

// random explorator : random exploration
class IfRandomExplorator : public IfExplorator {
```

```
// interactive explorator
class IfInteractiveExplorator : public IfExplorator {

// debugger
class IfDebugger: public IfExplorator {
```

5.4.1.4 Les modules de l'explorateur randomisé

Les structures de données :

```
// abstract class: node
template <class C, class L>
class Node{

// abstract class: tree of nodes
template < class C, class L>
class Tree{

// random algorithm 1: URS
template <class C, class L>
class Tree1: public Tree<C, L>{

// random algorithm 2: DORS
template <class C, class L>
class Tree2: public Tree<C, L> {

// random algorithm 3: RMA
template <class C, class L>
class Tree3: public Tree2<C, L> {
```

L'explorateur randomisé :

```
// tree: defines heuristic tree exploration abstract class
class  IfTreeExplorator: public IfExplorator {

// tree1 : defines heuristic urs exploration class
class IfTree1Explorator: public IfTreeExplorator

// tree2 : defines heuristic dors exploration class
class IfTree2Explorator: public IfTreeExplorator

// tree3 : defines heuristic rma exploration class
class IfTree3Explorator: public IfTreeExplorator
```

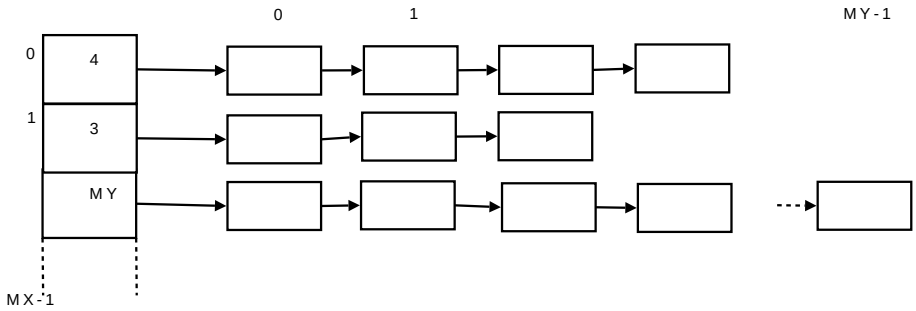


FIGURE 5.6 – La table de hashage des états visités

5.4.1.5 Implémentation de la sélection uniforme

L'ensemble des états visités est implémenté en table de hashage pour faciliter le stockage et la recherche des états. La sélection uniforme d'un état s dans la table de hashage se fait par la *méthode de rejet*. En effet, on génère deux variables aléatoires uniformes : X entre 0 et $MX - 1$, qui représente le rang de s dans la table de hashage (verticale), et Y entre 0 et $MY - 1$, qui représente le rang de s dans la liste chainée (horizontale). X correspond toujours à une case dans la table de hashage, mais Y peut être supérieure à la taille de la liste chainée de rang X, dans ce cas, il n'y a aucun état qui correspond au couple (X, Y). Si le couple (X, Y) ne correspond à aucun état, on le rejète et on regénère un autre couple jusqu'à l'obtention d'un couple valide.

5.4.1.6 Comment explorer aléatoirement un graphe

```
$ ./token.x -t1 -nstate 1000 -ch 100 -mx 50 -rseed 123 -rep 10 -mc 598
```

-t1, -t2, -t3 : les 3 algorithmes d'explorations
-nstates : nombre maximal d'états explorés
-ch : la taille du cache utilisé
-mx : taux de mixage pour l'algorithme t3 (RMA) (voir chapitre 8)
-rseed : random seed
-rep : nombre de répétitions (pour le calcul d'une moyenne par exemple)
-mc : numéro du nœud recherché (ou nœud cible)

5.4.2 Expérimentations

Le calcul théorique décrit précédemment donne de bonnes indications sur le comportement des algorithmes d'exploration proposés. Cependant, il est nécessaire d'effectuer une étude expérimentale sur des graphes issues du monde de model checking, sur lesquels, on compare l'efficacité des algorithmes proposés et de leurs versions répétées.

Nous avons implémenté les algorithmes proposés sur le model-checker IF [6]. Plusieurs exemples ont été considérés et plusieurs mesures ont été calculées :

5.4.2.1 Temps moyen de couverture

Dans un premier temps, nous nous intéressons au temps moyen de couverture. Pour cela, nous avons considéré les exemples de graphes suivants : *Token Ring Protocol*, *Client/Server Protocol*, *Alternating Bit Protocol*, *File System protocol*, *Fischer's Mutual Exclusion Protocol* et *Java Producer/Consumer Protocol*. Ces exemples ont été choisis selon les besoins de l'expérimentation. Nous avons réglé leurs paramètres afin d'obtenir des graphes de tailles moyennes. Ceci nous a permis de répéter chaque expérimentation 100 fois, réaliser la couverture totale de l'espace d'états à chaque fois et calculer la moyenne des temps d'exécution correspondant aux différents pourcentage de couverture (60%, 70%, 80%, 90%, 100%). Les exemples traîtés ont des caractéristiques différentes(tailles, profondeur), ce qui permet d'analyser leur comportement selon leurs formes.

5.4.2.2 Nombre moyen de nœuds couverts

Dans un deuxième temps, et dans le but de comparer les algorithmes aléatoires partiels et l'algorithme déterministe exhaustif BFS implémenté dans IF, nous avons utilisé trois exemples : *Fischer's Mutual Exclusion Protocol*, *Client/Server Protocol* et *Java Producer/Consumer Protocol*. Dans ce cas, on règle leurs paramètres de sorte à obtenir des graphes de tailles très grandes et inconnues (plusieurs millions). Pour chaque algorithme, le nombre de nœuds couverts est enregistré en fonction du temps et les courbes représentant les résultats sont tracés.

5.5 Conclusion

Nous avons présenté dans ce chapitre une méthodologie à suivre et des résultats généraux valables dans les chapitres suivants consacrés à l'étude détaillée des différents algorithmes. Cette méthodologie et ces résultats peuvent être appliqués à un algorithme d'exploration randomisé A quelconque. Nous nous sommes concentrés sur la randomisation de la sélection : un listing sommaire des façons de randomiser la sélection a été présenté, et pour des raisons de clarté, la condition d'arrêt et la fonction d'actualisation sont faites par réinitialisation.

La méthodologie générale de l'étude théorique commence par se fixer deux classes de graphes : les arbres et les grilles. Ensuite, le séquencement des mesures récursives qui seront effectuées a été précisé. Dans la méthodologie de l'étude expérimentale, on décrit la plate forme utilisée et les différents modules de l'exploration randomisées ajoutés. On explique également le choix des exemples de graphes utilisés et la méthode de calcul des différentes mesures, donnant ainsi le plan qui sera suivi dans l'étude de nos algorithmes dans les chapitres suivants.

Chapitre 6

Sélection orientée profondeur

6.1 Introduction

Dans ce chapitre, nous étudions un algorithme d'exploration aléatoire DORS (*Depth Oriented Random Search*) orienté en profondeur et nous le comparons à son dual BORS (*Breadth Oriented Random Search*) qui est orienté en largeur. En effet, les états de plus grande profondeurs ont de faibles probabilités d'être atteint dans une exploration de profondeur limitée. Notre algorithme est inspiré de DFS avec une sélection randomisée.

6.2 Depth Oriented Random Search

Conformément au schéma 4.1 présenté dans le chapitre 4, page 38, cet algorithme explore l'espace d'états et garde les nœuds visités en mémoire jusqu'à l'épuisement de celle-ci. Notons que le fait que DORS soit doté de mémoire, contrairement au RW, lui permet de distinguer entre les états visités et non visités et évite beaucoup d'explorations redondantes.

La fonction de sélection choisie, à chaque étape, uniformément, un successeur du dernier nœud visité v (voir figure 6.2 ci-dessous). Cependant, l'exploration ainsi conduite risque de se bloquer dans deux cas : en arrivant à une feuille ($succ(v) = \emptyset$) ou dans une boucle. Notons par D l'ensemble des nœuds feuilles visités et par W l'ensemble d'exploration courant qui consiste en l'ensemble des nœuds visités depuis le dernier blocage (dernière atteinte d'une feuille ou détection d'une boucle) jusqu'à l'instant courant. Dans le cas où $succ(v) \neq \emptyset$, mais que le successeur sélectionné de v se retrouve dans W, une boucle, dite engendrée par v, est alors détectée. Elle est en fait contenue dans W. Dans le cas de blocage (v est une feuille ou v engendre une boucle), l'exploration est dite être en un point fermé, autrement elle est dite en un point ouvert. En arrivant à un point fermé, l'exploration change de comportement : le nœud courant v est resélectionné uniformément dans $V \setminus D$ et l'exploration se poursuit en tirant aléatoirement, et uniformément, un nœud de $succ(v)$.

Il convient de noter le double avantage du choix de v parmi $V \setminus D$, dans le cas de blocage. D'une part, ceci permet d'éviter, l'inconvénient du RW qui, après chaque blocage, se trouve réinitialisé à partir de l'état initial, entraînant ainsi une exploration redondante des états de petite profondeur. D'autre part, il donne plus de chance à une exploration encore plus profonde.

```
V : ensemble des nœuds en mémoire ;
W,D : ensembles de nœuds ;
N : taille maximale de V ;
v : nœud courant ;
i : entier ;

V ← {v0} ;
W,D ← {} ;
v ← v0 ;
i ← 0 ;

Tant que ((i ≤ N)) faire
    Si (Succ(v) = ∅ ou v ∈ W) Alors
        Si (Succ(v) = ∅) Alors
            D ← D ∪ {v} ;
        Fin Si
        v ← choisir uniformément un nœud dans V ;
        W ← {v} ;
    Sinon
        v ← choisir uniformément un nœud dans Succ(v) ;
        Si (v ∉ V) Alors
            vérifier(v) ;
            V ← V ∪ {v} ;
            W ← W ∪ {v} ;
            i ← i + 1 ;
        Fin Si
    Fin Si
Fait
```

FIGURE 6.1 – Depth Oriented Random Search- DORS

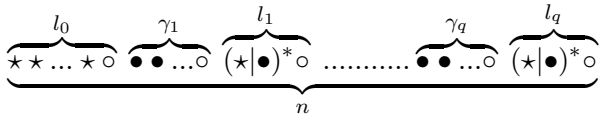
Bien que notre algorithme possède une fonction de sélection semblable à celle de DORS, proposé dans [32], il diffère de celui-ci dans plusieurs aspects. DORS est un algorithme randomisé du type Las Vegas. Il ne garde en mémoire que les nœuds "non-clos" (i.e. ceux qui ont au moins un successeur non visité). Le cas de saturation de la mémoire n'est pas étudié et la répétition de DORS n'est pas considérée (comme il sera détaillé pour DORS). Notre algorithme est ressource-dépendant, incluant explicitement la taille de la mémoire comme paramètre contrairement à DORS.

DORS va être étudié en détail théoriquement et expérimentalement selon la méthodologie et le schéma d'étude présenté dans le chapitre 5. L'étude de BORS, non détaillé ici, suit le même schéma. Les résultats de comparaison théorique et expérimentale des deux algorithmes seront présentés dans la suite.

6.3 Étude théorique

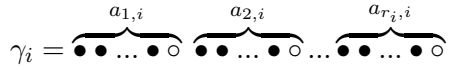
Cette section vise à effectuer une analyse théorique de l'algorithme DORS en terme des mesures mentionnées dans le chapitre 5. Il sera comparé a BORS, son dual orienté en largeur et dont les calculs sont duaux, à ceux de DORS détaillés ci-dessous.

Pour mieux expliquer, schématiquement, l'algorithme DORS, posons $V_n = (v_1, ..., v_n)$ la suite ordonnées des nœuds visités en n étapes par l'algorithme DORS. On observe que V_n peut être présentée comme suit :



Cette suite est constituée de plusieurs chemins : le premier chemin l_0 commence par l'état initial et se termine par un blocage ou point fermé de l'exploration (voir explication ci-dessus). Le blocage ou point fermé est représenté par un nœud vide, marqué $\circ$. Chacun des autres chemins l_i et γ_i commence par un nœud choisi aléatoirement de $V \setminus D$ et se termine par un point fermé, marqué $\circ$.

Les $\star$ représentent les k nœuds distincts et les $\bullet$ représentent les nœuds répétés. Le premier chemin l_0 est constitué exclusivement de nœuds distincts et nouvellement visités marqués par des $\star$. Les autres chemins l_i sont un mélange de nœuds répétés $\bullet$ et de nœuds nouvelement visités $\star$. Entre chaque deux chemins l_i et l_{i+1}, il existe une suite γ_i de plusieurs chemins répétés. Les chemins des γ_i ne contiennent que des nœuds répétés et ce depuis le dernier blocage ou point fermé de l'exploration jusqu'à l'atteinte d'un nouveau point fermé $\circ$. Avec plus de détails, les suites γ_i s'organisent comme suit :



6.3.1 Résultats généraux

Nous présentons d'abord quelques résultats généraux valables sur n'importe quel graphe en commençant par une récurrence élémentaire satisfaite par DORS. Cette récurrence élémentaire doit différencier entre les points fermés et les points ouverts de l'exploration soulignés ci-dessus. Elle est exprimée dans le lemme ci-dessous, où l'on a noté par $\underline{v_k} = (v_1, ..., v_{k-1}, v_k)$ la suite des nœuds visités distincts, ordonnés par ordre de visite.

Lemme 6.1 *Soit $\mathbb{P}(\underline{v}_k, n, C)$ (resp. $\mathbb{P}(\underline{v}_k, n, O, v)$) la probabilité de couvrir, en n étapes, la suite des nœuds $\underline{v}_k$ et d'être à l'étape n en un point fermé (resp. en un point ouvert au nœud v). Alors :*

$$\begin{aligned}
\mathbb{P}(\underline{v}_k, n, C) &= \frac{1}{k - |D(v_k)|} \sum_{v \in \underline{v}_k \setminus D(\underline{v}_k)} \frac{|C(v) \cap \underline{v}_k|}{|C(v)|}\ \mathbb{P}(\underline{v}_k, n-1, C) \\
&+ \sum_{v \in D(\underline{v}_k)} \frac{|C(v) \cap \underline{v}_k|}{|C(v)|}\ \mathbb{P}(\underline{v}_k, n-1, O, v) \\
\mathbb{P}(\underline{v}_k, n, O, v) &= \sum_{u \in F(v) \cap \underline{v}_k} \Big[\frac{1}{|C(u)|}\ \mathbb{P}(\underline{v}_k, n-1, O, u) + \frac{1}{k|C(u)|}\ \mathbb{P}(\underline{v}_k, n-1, C) \\
&+ \frac{1_{v_k}(v)}{|C(u)|} \Big(\mathbb{P}(\underline{v}_{k-1}, n-1, O, u) + \frac{1}{(k-1-|D(v_{k-1})|)} \mathbb{P}(\underline{v}_{k-1}, n-1, C) \Big) \Big]
\end{aligned}$$

où $D(\underline{v}_k)$ est l'ensemble de feuilles dans $\underline{v}_k$, $1_{v_k}(v) = 1$ si $v = v_k$ et $1_{v_k}(v) = 0$ sinon.

Preuve Voir l'annexe page 117. □

Notons que la récurrence élémentaire du lemme 6.1 ci-dessus est satisfaite par DORS pour tout graphe. Dans les sections suivantes, nous allons préciser ces résultats pour les cas particuliers des arbres et des grilles.

6.3.2 Cas des arbres

6.3.2.1 Probabilité de couverture de DORS

La récurrence élémentaire du lemme 6.1 revient à une récurrence plus simple, dépendante uniquement du nombre de nœuds de $\underline{v}_k$ dans chaque niveau de l'arbre et non pas de $\underline{v}_k$ lui même. Considérons $\underline{K}_n = (K_n^1, ..., K_n^h)$, le vecteur de variables aléatoires présentant le nombre de nœuds explorés à chaque niveau $j = 1, ..., h$, de l'arbre à l'étape n, et soit $\mathbb{P}_{DORS}(\underline{K}_n = \underline{k}, C)$ (resp. $\mathbb{P}_{DORS}(\underline{K}_n = \underline{k}, O)$) la probabilité de couvrir le vecteur $\underline{k} = (k_1, ..., k_h)$ en n étapes pour se trouver dans le cas fermé (resp. ouvert). On obtient :

$$\mathbb{P}_{DORS}(\underline{K}_n = \underline{k}) = \mathbb{P}_{DORS}(\underline{K}_n = \underline{k}, C) + \mathbb{P}_{DORS}(\underline{K}_n = \underline{k}, O)$$

avec pour le cas fermé :

$$\begin{aligned}
\mathbb{P}_{DORS}(\underline{K}_n = \underline{k}, C) &= \sum_{s=1}^{h+1} \frac{1}{m^{s-1}} \Big[\alpha(\underline{k}, C)\ \mathbb{P}_{DORS}(\underline{K}_{n-s} = \underline{k}, C) \\
&+ \sum_{j=h-s+2}^{h} \beta_j(\underline{k}, C)\ \mathbb{P}_{DORS}(\underline{K}_{n-s} = \underline{k} - 1_{j,h}, C) \Big] \\
\text{où} \quad \alpha(\underline{k}, C) = \frac{k_h}{k} \quad &, \quad \beta_j(\underline{k}, C) = \frac{m k_{j-1} - (k_j - 1)}{(k - h + j - 1) m^{j-h}} \\
\text{et } \underline{k} - 1_{j,j'} &= (k_1, ..., k_{j-1}, k_j - 1, ..., k_{j'} - 1, k_{j'+1}, ..., k_h).
\end{aligned}$$

et pour le cas ouvert :

$$\begin{aligned}\mathbb{P}_{DORS}(\underline{K}_n = \underline{k}, O) &= \sum_{s=1}^{h} \frac{1}{m^s}\Big[\alpha_s(\underline{k}, O)\, \mathbb{P}_{DORS}(\underline{K}_{n-s} = \underline{k}, C) \\ &+ \sum_{j=1}^{h} \sum_{l=\max(j,0)}^{\min(s+j-1,h)} \beta_{j,l}(\underline{k}, O)\, \mathbb{P}_{DORS}(\underline{K}_{n-s} = \underline{k} - 1_{j,l}, C)\Big]\end{aligned}$$

où $\alpha_s(\underline{k}, O) = \dfrac{k_s + ... + k_h}{k}$ et $\beta_{j,l}(\underline{k}, O) = \dfrac{mk_{j-1} - (k_j - 1)}{(k - l + j - 1)m^{j-l}}$

Dans chaque équation, deux termes apparaissent : $\alpha(\underline{k}, C)$ et $\alpha_s(\underline{k}, O)$ sont des termes de redondance, où aucun nouveau nœud n'est visité à l'étape n, et $\beta_j(\underline{k}, C)$ et $\beta_{j,l}(\underline{k}, O)$, qui sont des termes d'innovation, i.e. un nouveau nœud est visité à l'étape n. Ceci va nous permettre de calculer ci-après les probabilités d'innovations puis le temps moyen de couverture.

La probabilité d'innovation correspond à la probabilité de couvrir le k-ième élément du vecteur $\underline{k}$ en la n-ième étape et de n'avoir couvert à l'étape $n-1$ que $k-1$ éléments. Elle est exprimée par :

$$\mathbb{P}^{\mathcal{I}}_{DORS}(\underline{K}_n = \underline{k}) = \mathbb{P}^{\mathcal{I}}_{DORS}(\underline{K}_n = \underline{k}, C) + \mathbb{P}^{\mathcal{I}}_{DORS}(\underline{K}_n = \underline{k}, O).$$

où

$$\begin{aligned}\mathbb{P}^{\mathcal{I}}_{DORS}(\underline{K}_n = \underline{k}, C) &= \sum_{s=1}^{h+1} \frac{1}{m^{s-1}} \sum_{j=h-s+2}^{h} \beta_j(\underline{k}, C)\, \mathbb{P}_{DORS}(\underline{K}_{n-s} = \underline{k} - 1_{j,h}, C) \\ \mathbb{P}^{\mathcal{I}}_{DORS}(\underline{K}_n = \underline{k}, O) &= \sum_{s=1}^{h} \frac{1}{m^s} \sum_{j=1}^{h} \sum_{l=\max(j,0)}^{\min(s+j-1,h)} \beta_{j,l}(\underline{k}, O)\, \mathbb{P}_{DORS}(\underline{K}_{n-s} = \underline{k} - 1_{j,l}, C)\end{aligned}$$

6.3.2.2 Temps moyen de couverture

Comme précisé dans le chapitre précédent, le temps moyen $T_A(k)$ de couvrir k nœuds par un algorithme A est exprimé en fonction des probabilités d'innovation comme suit :

$$T_A(k) = \sum_{|\underline{k}|=k} T_A(\underline{k}), \quad T_A(\underline{k}) = \sum_{n=k}^{\infty} n P^{\mathcal{I}}_A(\underline{K}_n = \underline{k})$$

Avec plus d'investissement, et afin de pouvoir calculer efficacement le temps moyen de couverture, nous avons pu identifier les statistiques suivantes, calculées récursivement :

$$\begin{aligned}S^0_{DORS}(\underline{k}) &= \sum_{j=1}^{h} \gamma_j(\underline{k})\, S^0_{DORS}(\underline{k} - 1_{j,h}) \\ S^1_{DORS}(\underline{k}) &= \sum_{j=1}^{h} \Big(\gamma_j(\underline{k}) S^1_{DORS}(\underline{k} - 1_{j,h}) + \delta_j(\underline{k})\, S^0_{DORS}(\underline{k} - 1_{j,h})\Big) + \mu(\underline{k}) S^0_{DORS}(\underline{k})\end{aligned}$$

$$\text{où } \gamma_j(\underline{k}) = \frac{d_0(h-j+1,h)\beta_j(\underline{k},C)}{1-d_0(0,h)\alpha(\underline{k},C)} \quad , \quad \delta_j(\underline{k}) = \frac{d_1(h-j+1,h)}{d_0(h-j+1,h)}$$

$$\mu(\underline{k}) = \frac{d_0(0,h)\alpha(\underline{k},C)}{1-d_0(0,h)\alpha(\underline{k},C)} \quad , \quad d_0(j,j') = \frac{\frac{1}{m^{j-1}} - \frac{1}{m^{j'}}}{m-1}$$

$$\text{et } d_1(j,j') = \frac{md_0(j,j'+1) + \frac{j}{m^{j-1}} + \frac{j'+2}{m^{j'}}}{m-1}$$

On obtient le théorème 6.1 suivant :

Théorème 6.1 *Le temps moyen de couverture $T_{DORS}(\underline{k})$ est donné par :*

$$T_{DORS}(\underline{k}) = \sum_{j=1}^{h}\sum_{l=j}^{h}\left[c_{j,l}(\underline{k})S^1_{DORS}(\underline{k}-1_{j,l}) + d_{j,l}(\underline{k})S^0_{DORS}(\underline{k}-1_{j,l})\right] + a(\underline{k})\ S^1_{DORS}(\underline{k}) - b(\underline{k})\ S^0_{DORS}(\underline{k})$$

$$\textit{avec} \quad a(\underline{k}) = 1-d_0(0,h)\alpha(\underline{k},C) \quad , \quad b(\underline{k}) = d_1(0,h)\alpha(\underline{k},C),$$
$$c_{j,l}(\underline{k}) = \beta_{j,l}(\underline{k},O)d_0(l-j,l) \quad , \quad d_{j,l}(\underline{k}) = \beta_{j,l}(\underline{k},O)(d_1(l-j,l) - d_0(l-j,l)),$$
$$\textit{et} \quad \underline{k}-1_{j,l} = (k_1,...,k_j-1,...,k_l-1,...,k_h)$$

Preuve Voir l'annexe page 122. □

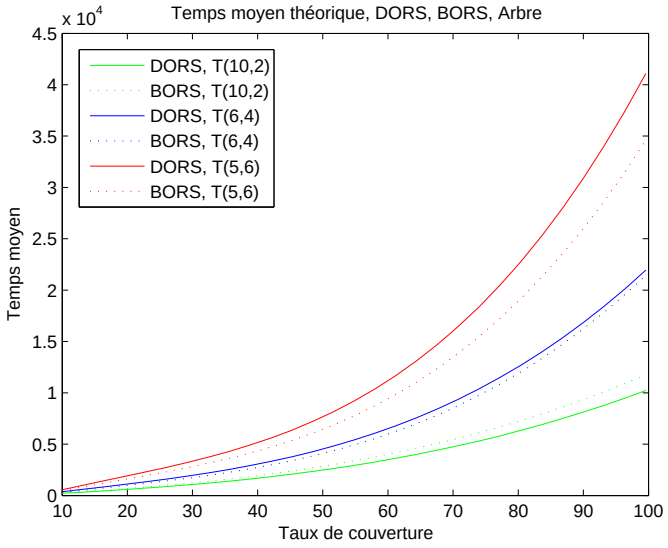


FIGURE 6.2 – Temps moyen de couverture de DORS et BORS (cas arbre)

Dans la figure 6.2, on présente l'évolution du temps moyen de couverture pour DORS et BORS en fonction du pourcentage couvert du graphe pour différents arbres

$T(h, m)$, où h est la profondeur de l'arbre et m est son degré.

On remarque que le temps de couverture de DORS est meilleur (plus petit) que celui de BORS pour les arbres profonds et maigres ($T(10, 2)$), ceci est dû au fait qu'il est orienté pour explorer en profondeur. En revanche, BORS couvre mieux (en moins de temps) les arbres larges ($T(5, 6)$). Les notions de "maigre" et "large" seront précisées plus loin via le coefficient de densité (DF) que nous définissons et qui n'est autre pour les arbres que le rapport m/h. La majorité des arbres réguliers sont larges (selon notre classification par DF), ce qui explique la supériorité de BORS dans ce cas, contrairement au cas de graphes de Model Checking comme nous le verrons plus loin.

6.3.2.3 Probabilité d'atteignabilité

Des formules récursives de la probabilité d'atteignabilité ont été obtenues. Soit $\mathbb{P}_A(i; \underline{K}_n = \underline{k})$ [1] la probabilité d'atteindre un nœud se trouvant au niveau i de l'arbre en n étapes, et couvrir le vecteur $\underline{k} = (k_1, ..., k_h)$, i.e. couvrir k_j nœuds dans chaque niveau j de l'arbre. Soit la probablité $\mathbb{P}_A(i; K_n = k)$ d'atteindre un nœud au niveau i et couvrir un nombre k de nœuds par un algorithme A. Cette probabilité est calculée pour un nombre de nœuds couverts k inférieur au seuil N. Pour DORS, on obtient :

$$\mathbb{P}_{DORS}(i; \underline{K}_n = \underline{k}) = \mathbb{P}_{DORS}(i; \underline{K}_n = \underline{k}, C) + \mathbb{P}_{DORS}(i; \underline{K}_n = \underline{k}, O)$$

où, en posant $\gamma_j = \frac{1}{(k-h+j-1)m^{-h}}$ et $\gamma_{j,l} = \frac{1}{(k-l+j-1)m^{-l}}$

$$\begin{aligned}
\mathbb{P}_{drs}(i; \underline{K}_n = \underline{k}, C) &= \sum_{s=1}^{h+1} \frac{1}{m^{s-1}} \Big[\alpha(\underline{k}, C) \mathbb{P}_{drs}(i; \underline{K}_{n-s} = \underline{k}, C) \\
&+ \sum_{j=h-s+2}^{h} \beta_j(\underline{k}, C)\ \mathbb{P}_{drs}(i; \underline{K}_{n-s} = \underline{k} - 1_{j,h}, C) \\
&+ \sum_{j=h-s+2}^{i} \frac{\gamma_j}{m^i} \Big(\mathbb{P}_{drs}(j-1; \underline{K}_{n-s} = \underline{k} - 1_{j,h}, C) \\
&- \mathbb{P}_{drs}(j; \underline{K}_{n-s} = \underline{k} - 1_{j,h}, C) \Big) \Big] \\
\mathbb{P}_{drs}(i; \underline{K}_n = \underline{k}, O) &= \sum_{s=1}^{h} \frac{1}{m^s} \Big[\alpha(\underline{k}, O)\ \mathbb{P}_{drs}(i; \underline{K}_{n-s} = \underline{k}, C) \\
&+ \sum_{j=1}^{h} \sum_{l=max(j,0)}^{min(s+j-1,h)} \beta_{j,l}(\underline{k}, O)\ \ \mathbb{P}_{drs}(i; \underline{K}_{n-s} = \underline{k} - 1_{j,l}, C) \\
&+ \sum_{j=1}^{i} \sum_{l=max(j,0)}^{min(s+j-1,h)} \frac{\gamma_{j,l}}{m^i} \Big(\mathbb{P}_{drs}(j-1; \underline{K}_{n-s} = \underline{k} - 1_{j,l}, C) \\
&- \mathbb{P}_{drs}(j; \underline{K}_{n-s} = \underline{k} - 1_{j,l}, C) \Big) \Big]
\end{aligned}$$

1. Notons que l'utilisation du " ;" dans $\mathbb{P}_A(i; \underline{K}_n = \underline{k}, C)$ signifie que ce terme désigne la probabilité d'atteindre le nœud i "et" couvrir en même temps le vecteur $\underline{k}$. L'utilisation de "," signifie que les termes suivants sont des attributs de cette probabilité

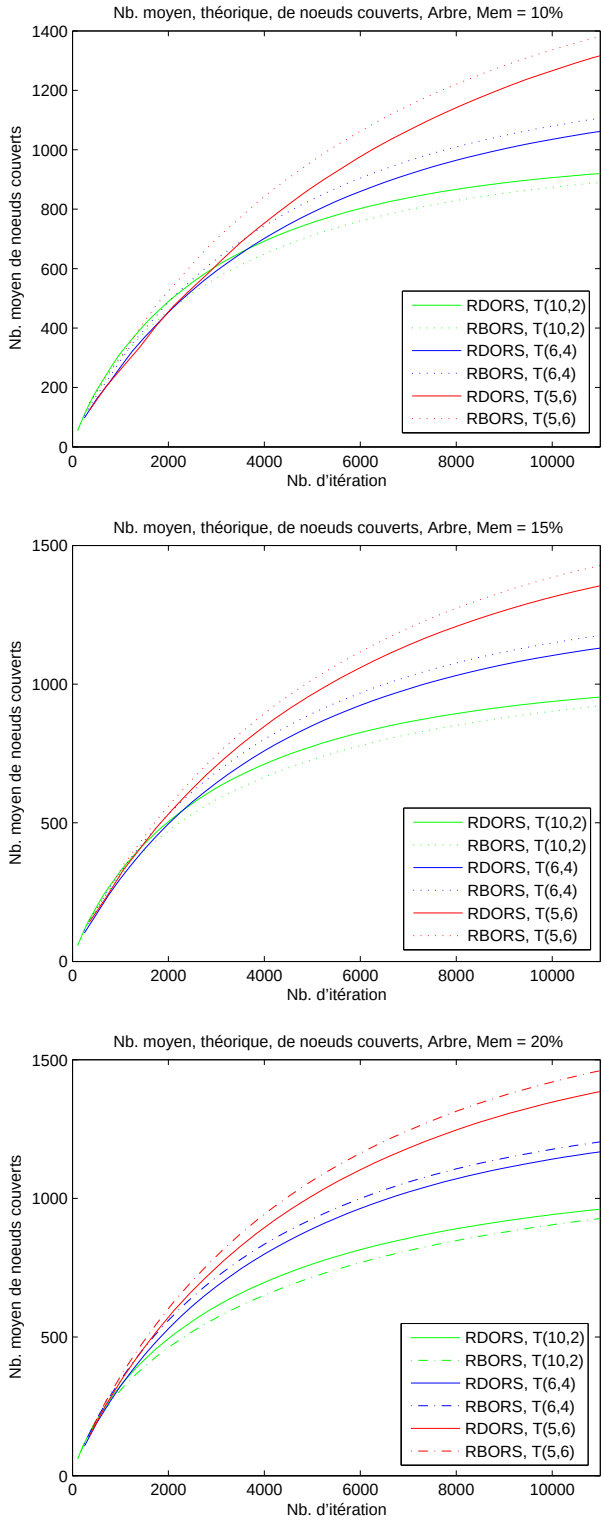


FIGURE 6.3 – Nombre moyen de nœuds couverts par DORS et BORS (cas arbre)

6.3.2.4 Le nombre moyen de nœuds couverts par RDORS

Le nombre moyen de nœuds couverts par RDORS peut être calculé en se basant sur les probabilités d'atteignabilité données avant, et en utilisant le théorème 5.1 du chapitre 5, page 50. Le nombre moyen de nœuds couverts par RDORS et RBORS est tracé en fonction du temps pour différents arbres $T(h, m)$, où h est la profondeur de l'arbre et m son degré, et différents pourcentages de la taille mémoire. Les résultats de comparaison sont illustrés dans les figures 6.3 :

Ces figures montrent les résultats attendus : DORS est meilleur pour les arbres maigres et profonds. En revanche, pour des arbres larges, il convient de favoriser l'algorithme BORS.

6.3.3 Cas des grilles

Conformément au chapitre 5, section 5.3.4, on se place dans le contexte d'une grille multidimentionnelle. On est intéressé par le calcul des mêmes mesures que dans le cas des arbres, à savoir le temps moyen de couverture et le nombre moyen des nœuds couverts par les algorithmes DORS et BORS. Nous analysons ces mesures en se basant sur les récurrences élémentaires du lemme 6.1 page 62.

Les courbes du temps moyen de couverture sont illustrées dans la figure 6.4, et ce pour plusieurs grilles $G(L, d)$, où $L + 1$ est la longueur de la grille et d est son degré :

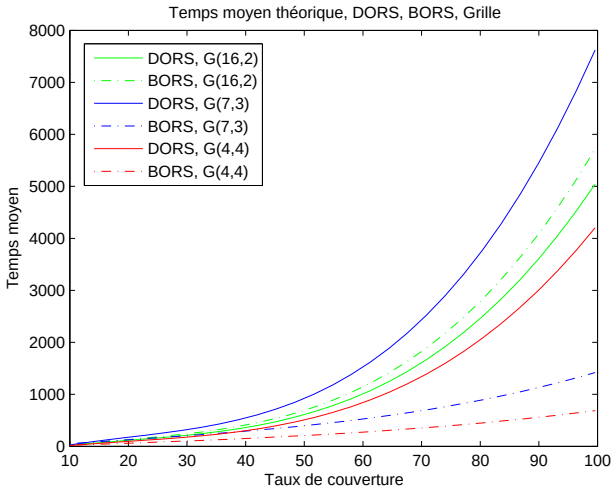


FIGURE 6.4 – Temps moyen de couverture de DORS et BORS (cas grille)

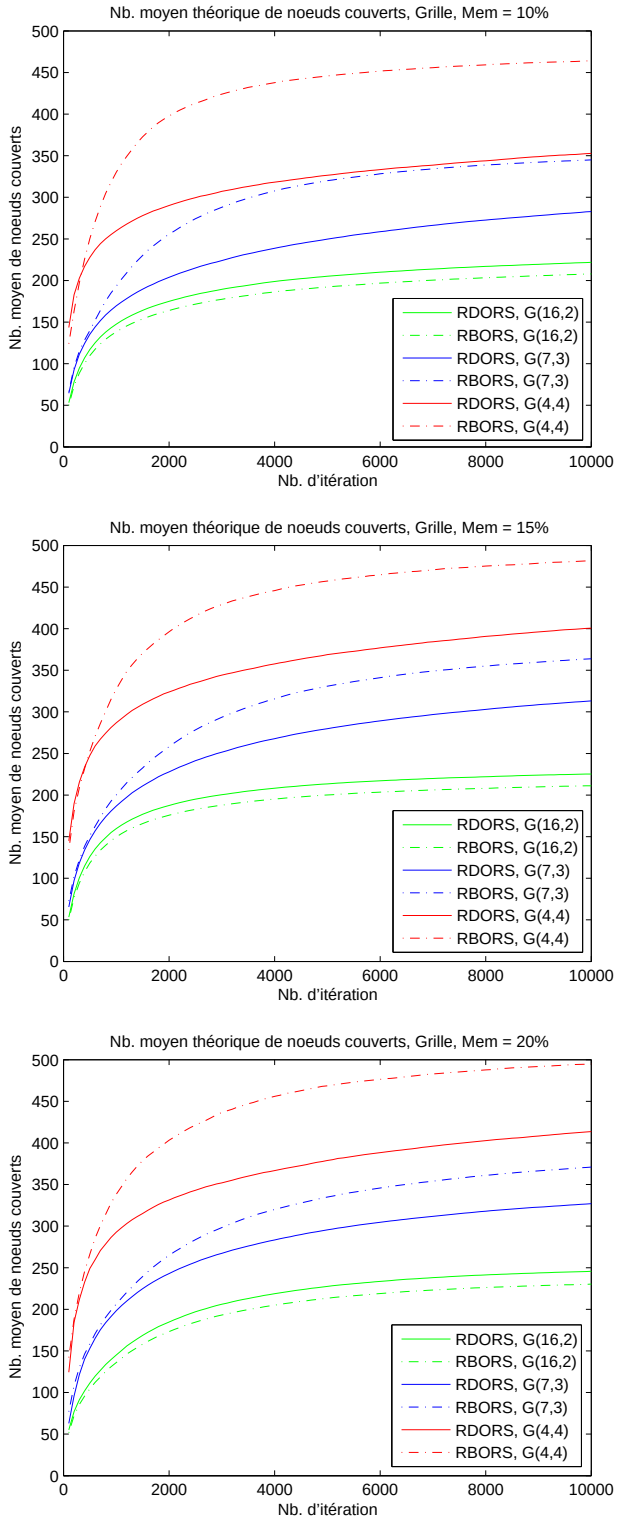


FIGURE 6.5 – Nombre moyen de nœuds couverts par DORS et BORS (cas grille)

Nous avons calculé le nombre moyen de nœuds couverts par les algorithmes répétés RDORS et RBORS pour plusieurs grilles G(L, d) et plusieurs pourcentages de la taille mémoire par rapport à celle du graphe. Les résultats sont tracés dans les figures 6.5 :

Que ce soit pour le temps moyen de couverture ou le nombre moyen de nœuds couverts, on note le même résultat que dans le cas des arbres : DORS est mielleur que BORS pour les graphes maigres et moins bon dans les cas de graphes larges.

En comparant les trois figures de couverture avec différents taux de mémoire, on remarque évidemment que le nombre de nœuds couverts est plus grand si la mémoire utilisée est plus importante. Cependant, si on s'approche de la couverture totale, cette différence est moins visible comme c'est le cas dans les courbes correspondantes à T(10,2).

6.4 Étude expérimentale

L'étude expérimentale de DORS et *BORS* sera différée au prochain chapitre où ils seront comparés à l'algorithme *URS* étudié dans ce chapitre.

6.5 Conclusion

Nous avons proposé dans ce chapitre un algorithme d'exploration aléatoire DORS (resp. son dual BORS) dont la fonction de sélection favorise l'exploration en profondeur (resp. en largeur). Nous avons étudié ces algorithmes puis leurs versions répétées : RDORS et RBORS conformément la méthodologie décrite au chapitre précédent 5. Les mesures calculées théoriquement ont permis de comparer les performances de DORS et BORS sur des classes particulières de graphes (arbres et grilles). Les résultats obtenus ont confirmé nos prévisions : DORS est préférable pour les graphes maigres, tandis que BORS est meilleur pour les graphe larges. Les résultats de simulation sur des graphes de model checking, qui seront détaillés dans le chapitre suivant, sont conforme à cette constatation.

Chapitre 7

Sélection uniforme

7.1 Introduction

Dans ce chapitre, nous proposons un algorithme d'exploration basé sur une loi de sélection uniforme. L'uniformité apparaît dans le choix du nœud, à partir duquel, on va choisir le successeur à explorer. Ce choix est motivé, d'une part, par le fait que les algorithmes d'exploration aléatoires proposés sont généralement basées sur RW et favorisent ainsi l'exploration en profondeur, et d'autre part, par le fait que les algorithmes orientés largeur, tels que BORS (voir chapitre 5), couvrent mal les nœuds profonds du graphe étudié. Une couverture équilibrée du graphe est toujours souhaitable car elle donne une idée sur le comportement du système dans la plupart des régions du graphe, notamment en absence d'informations sur la structure de celui-ci. Par ailleurs, du point de vue statistique, le tirage uniforme, outre qu'il est naturel et simple, est souvent efficace, à moins que des paramètres de guidage ne soient disponibles et permettent d'identifier certaines distributions mieux adaptées au problème d'exploration considéré. L'algorithme URS va être analysé théoriquement et expérimentalement pour être comparé avec les algorithmes DORS et BORS.

7.2 Uniform Random Search

Comme détaillé dans la figure 7.2, on dispose d'un ensemble V d'états déjà visités de taille N. De ce fait, l'algorithme assure qu'il n'y aura pas plus que N états visités stockés à la fois (dans V). Initialement, cet ensemble contient l'état initial v_0. À chaque étape i, l'algorithme URS tire aléatoirement et d'une façon uniforme un état visité u de V, puis choisit aléatoirement et uniformément un successeur v de u. Notons que ceci n'implique pas un choix uniforme parmi tous les successeurs des nœuds visités. Si v n'est pas déjà visité, alors il est vérifié par rapport à la propriété voulue puis ajouté à l'ensemble des nœuds visités.

De même que DORS et BORS, cet algorithme est arrêté à chaque fois que la mémoire est pleine $i = N$ pour être réinitialisé et relancé. Sa version répétée sera notée RURS.

```
V : ensemble des nœuds en mémoire ;
N : taille maximale de V ;
v, u : nœuds ;
i : entier ;

V ← {v_0} ;
i ← 0 ;

Tant que ((i ≤ N)) faire
    u ← choisir uniformément un nœud dans V ;
    Si (Succ(u) ≠ ∅) Alors
        v ← choisir uniformément un nœud dans Succ(u) ;
        Si (v ∉ V) Alors
            vérifier(v) ;
            V ← V ∪ {v} ;
            i ← i + 1 ;
        Fin Si
    Fin Si
Fait
```

FIGURE 7.1 – Uniform Random Search- URS

7.3 Étude théorique

Dans cette section, on suit les mêmes étapes que dans le chapitre précédent, conformément à notre méthodologie fixée dans le chapitre 4. On donne d'abord les formules récursives générales, puis les formules simplifiées valables sur les classes de graphes choisies : arbres et grilles. On calcule ensuite le temps moyen de couverture pour l'algorithme URS à comparer avec ceux de DORS et BORS, ainsi que le nombre moyen de nœuds couverts pour RURS afin de le comparer à ceux de RDORS et RBORS.

7.3.1 Résultats généraux

La suite ordonnée $\underline{v}_k = (v_1, ..., v_k)$ des k nœuds distincts visités par URS en n étapes peut être représentée comme suit :

$$\underbrace{v_1, \overbrace{\ldots}^{\alpha_1}, v_2, \overbrace{\ldots\ldots}^{\alpha_2}, v_3, \overbrace{\ldots\ldots}^{\alpha_3}, ..., v_{k-1}, \overbrace{\ldots\ldots}^{\alpha_{k-1}}, v_k}_{\underline{v}_k=(v_1,...,v_k)}, \overbrace{\ldots}^{\alpha_k}$$

Où chaque v_i correspond à un nouvel état visité, suivi par des suites α_i de visites redondantes. La somme des longueurs de toutes les suites α_i étant égale à $n-k$ ($\sum_{\alpha=1}^{k} = n-k$). Notons par $F(v_i)$ (resp. $C(v_i)$) l'ensemble des prédécesseurs (resp. successeurs) du nœud v_i, $i = 1, .., k$.

Lemme 7.1 *La probabilité* $\mathbb{P}(\underline{v}_k, n)$ *de couvrir le vecteur* $\underline{v}_k$ *en* n *étapes par l'algorithme URS est donnée par :*

$$\mathbb{P}(\underline{v}_k, n) \quad = \quad \alpha(\underline{v}_k)\mathbb{P}(\underline{v}_k, n-1) + \beta(\underline{v}_k)\mathbb{P}(\underline{v}_{k-1}, n-1)$$

$$o\grave{u} \;\; \alpha(\underline{v}_k) = \frac{1}{k}\sum_{i=1}^{k} \frac{|C(v_i) \cap \underline{v}_k|}{|C(v_i)|} \;\; et \;\; \beta(\underline{v}_k) = \frac{1}{k-1} \sum_{v \in F(v_k) \cap \underline{v}_{k-1}} \frac{1}{|C(v)|}$$

Preuve La probabilité $\alpha(\underline{v}_k)$ de revisiter un nœud parmi $\underline{v}_k$ est la somme sur les v_i, $i = 1, ..., k$, de $1/k$, qui est la probabilité de choisir v_i, entant que père, multiplié par le facteur $\frac{|C(v_i) \cap \underline{v}_k|}{|C(v_i)|}$ exprimant la probabilité de choisir un fils de v_i dans $\underline{v}_k$. Le facteur $\beta(\underline{v}_k)$ est la probabilité de choisir à l'étape n un nouvel état v_k. Le facteur $\frac{1}{k-1}$ dans $\beta(\underline{v}_k)$ correspond au choix du père v de v_k dans $\underline{v}_{k-1}$. Le choix de v_k s'effectue ensuite avec une probabilité $\frac{1}{|C(v)|}$ parmi les fils de v. □

Notons que $\alpha(\underline{v}_k)$ est un facteur de redondance. Il est égal à la probabilité de revisiter un nœud à l'étape n (i. e. aucun état n'est nouvellement visité à cette étape). $\beta(\underline{v}_k)$, quand à lui, est un facteur d'innovation qui exprime la probabilité de couvrir à l'étape n un nouvel état, qui doit être v_k, vu que l'ensemble $\underline{v}_k$ est stocké dans l'ordre des visites.

Notons dans ce cas aussi, que la récurrence élémentaire du lemme 7.1 est satisfaite par URS pour tout graphe. Les résultats pour les cas particuliers des arbres et des grilles seront précisés dans les sections suivantes.

Dans ce qui suit, nous allons calculer les mêmes statistiques que dans le chapitre précédent, à savoir le temps moyen de couverture et le nombre moyen de nœuds couverts, afin de pouvoir comparer URS avec DORS et BORS.

7.3.2 Cas des arbres

La récurrence élémentaire du lemme 7.1 revient dans ce cas à une récurrence plus simple ne faisant intervenir que le vecteur $\underline{K}_n = (K_n^1, ..., K_n^h)$ des variables aléatoires K_n^j, présentant les nombres de nœuds explorés, à l'étape n, à chaque niveau $j = 1, ..., h$ de l'arbre.

7.3.2.1 Probabilité de couverture de URS

Soit $\mathbb{P}_{urs}(\underline{K}_n = \underline{k})$ la probabilité d'avoir couvert (exactement) le vecteur $\underline{k} = (k_1, ..., k_h)$ en n étapes par l'algorithme URS.

$$\mathbb{P}_{urs}(\underline{K}_n = \underline{k}) \quad = \quad \alpha(\underline{k})\, \mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k}) + \sum_{j=1}^{h} \beta_j(\underline{k})\, \mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k} - 1_j)$$

où $\underline{k} - 1_j = (k_1, ..., k_j - 1, ..., k_h)$, $1 \leq j \leq h$.

Comme dans la récurrence élémentaire, on distingue le terme de redondance des termes d'innovation, les mêmes notations sont utilisées. Le facteur de répétition $\alpha(\underline{k})$ est égale à la probabilité de revisiter un nœud à l'étape n et donné par :

$$\alpha(\underline{k}) = \frac{mk_h + k - 1}{mk}.$$

Les facteurs d'innovation $\beta_j(\underline{k})$ correspondent aux cas où, à l'étape n, un nouveau nœud est couvert à un certain niveau j, $j = 1, ..., h$, donc :

$$\beta_j(\underline{k}) = \frac{mk_{j-1} - k_j + 1}{m(k-1)}.$$

7.3.2.2 Temps moyen de couverture de URS

Le temps moyen $T_{urs}(\underline{k})$ nécessaire pour couvrir le vecteur $\underline{k}$ par URS est donné dans le théorème 7.1 ci-dessous. Il est calculé par l'intermédiaire des statistiques suivantes :

$$\begin{aligned} S^0_{urs}(\underline{k}) &= \sum_{i=1}^{h} \frac{\beta_i(\underline{k})}{1 - \alpha(\underline{k})}\, S^0_{urs}(\underline{k} - 1_i) \\ S^1_{urs}(\underline{k}) &= \frac{1}{1 - \alpha(\underline{k})} S^0_{urs}(\underline{k}) + \sum_{i=1}^{h} \frac{\beta_i(\underline{k})}{1 - \alpha(\underline{k})}\, S^1_{urs}(\underline{k} - 1_i) \end{aligned}$$

Le calcul de ces mesures est précisé dans la preuve du théorème 7.1.

Théorème 7.1 *Avec les notations ci-dessus, Le temps moyen $T_{urs}(\underline{k})$ est donné par :*

$$T_{urs}(\underline{k}) \quad = \quad (1 - \alpha(\underline{k}))S^1_{urs}(\underline{k}) - \alpha(\underline{k})S^0_{urs}(\underline{k})$$

Preuve Voir l'annexe pag 119. □

7.3.2.3 Comparaison des temps de couverture de URS, DORS et BORS

Le calcul du temps moyen de couverture de URS a été effectué à l'aide du théorème 7.1. Pour le comparer avec DORS et BORS, on considère différents arbres notés $T(h, m)$, où h présente la profondeur et m le degré. On montre dans la figure 7.2 le coût, en nombre moyen d'étapes, pour couvrir différents pourcentages des graphes considérés.

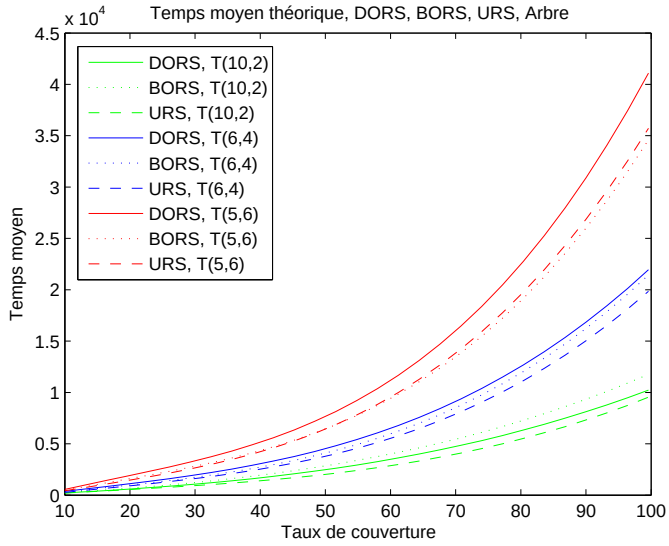


FIGURE 7.2 – Temps moyen de couverture- cas des arbres

On observe dans la figure 7.2 que l'algorithme URS, comparé à DORS, prend moins de temps pour couvrir une proportion donnée du graphe. Cette observation est d'autant plus claire que la proportion couverte du graphe est supérieures à 70% et dans les arbres larges. En effet, plus l'arbre est large (degré élevé), plus la différence dans les temps de couverture de URS et DORS est importante. Dans le cas d'arbres maigres, (i.e. de degré petit par rapport à la profondeur), DORS peut se conduire mieux que URS. Ceci est obtenu seulement pour des graphes très maigres. Inversement, on note que BORS est un peu plus performant que URS pour les graphes larges alors que URS se comporte mieux que BORS généalement, en particulier pour les graphes moins larges. Rappelant que le cas des arbres réguliers correspond en général à des graphes larges contrairement au cas des graphes issus de modèles réels comme il sera vu dans nos expérimenations, en particulier dans la section 8.2.1, page 86 où sera définit le facteur de densité .

7.3.2.4 Probabilité d'atteignabilité pour URS

La formule récursive de la probabilité $\mathbb{P}_{urs}(i; \underline{K}_n = \underline{k})$ d'atteindre un nœud se situant au niveau i de l'arbre et couvrir le vecteur $\underline{k}$ est obtenue ci-dessous. Les termes $\alpha(\underline{k})$ et $\beta(\underline{k})$étant conservés, on obtient avec $\gamma(\underline{k}) = \frac{1}{m(k-1)}$:

$$\begin{aligned}\mathbb{P}_{urs}(i; \underline{K}_n = \underline{k}) &= \alpha(\underline{k})\, \mathbb{P}_{urs}(i; \underline{K}_{n-1} = \underline{k}) + \sum_{j=1}^{h} \beta_j(\underline{k})\, \mathbb{P}_{urs}(i; \underline{K}_{n-1} = \underline{k} - 1_j) \\ &+ \gamma(\underline{k}) \left[\mathbb{P}_{urs}(i-1; \underline{K}_{n-1} = \underline{k} - 1_i) - \mathbb{P}_{urs}(i; \underline{K}_{n-1} = \underline{k} - 1_i)\right]\end{aligned}$$

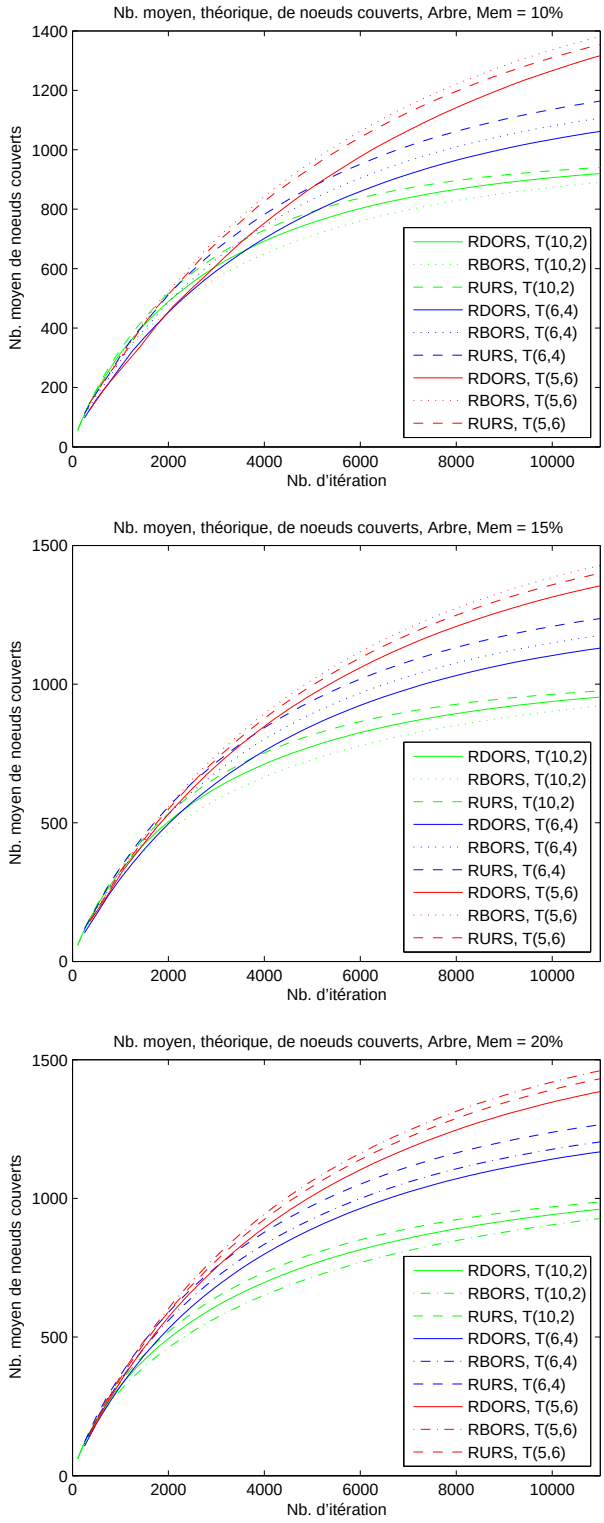


FIGURE 7.3 – Nombre moyen de nœuds couverts- cas des arbres

7.3.2.5 Comparaison des nombres moyens de nœuds couverts par RURS, RDORS et RBORS

Les formules générales du nombre moyen de nœuds couverts par la version répétée RA d'un algorithme randomisé A ont été données par le théorème 5.1 au chapitre5, page 50. Ces formules ont été calculées pour tracer différentes courbes nous permettant de comparer RURS, RDORS et RBORS. La figure 7.3 montre l'évolution du nombre de nœuds couverts en fonction du temps. Ces courbes, représentant le comportement des algorithmes répétés RURS RDORS et RBORS, sont tracées pour trois arbres de degrés et profondeurs différents. Le nombre de nœuds couverts est calculé pour une taille mémoire N de 10%, 15% et 20% des tailles des graphes.

L'algorithme RURS est clairement meilleur que RDORS, spécialement lorsque le nombre de nœuds couverts est grand. On observe aussi que la différence entre RURS et RDORS dans le nombre de nœuds couverts est plus importante lorsque l'arbre est plus dense (degré élevé). Quant à BORS, il donne le meilleur résultat de couverture dans le cas de l'arbre le plus large. Pour les arbres moins larges (moyen ou maigres), URS donne un nombre de nœuds couverts plus important que BORS.

7.3.3 Cas des grilles

On a montré dans le chapitre 5 la difficulté de produire des formules récursives simplifiées dans le cas des grilles, et on a expliqué les aménagements effectués pour calculer le temps moyen de couverture et le nombre moyen de nœuds couverts.

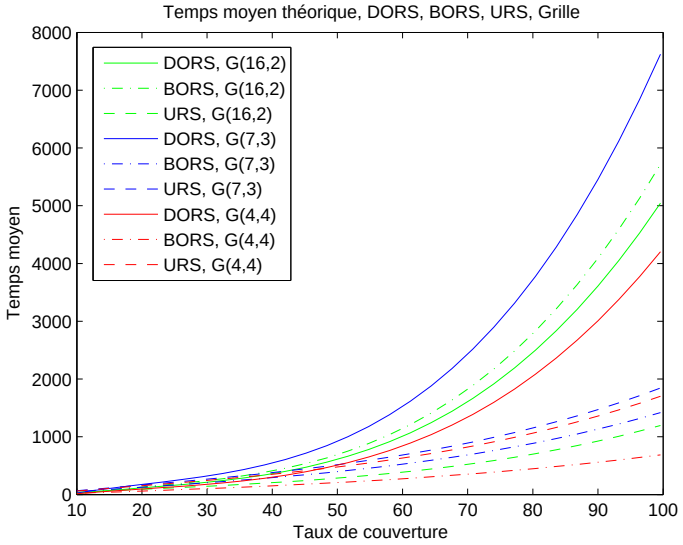


FIGURE 7.4 – Temps moyen de couverture- cas des grilles

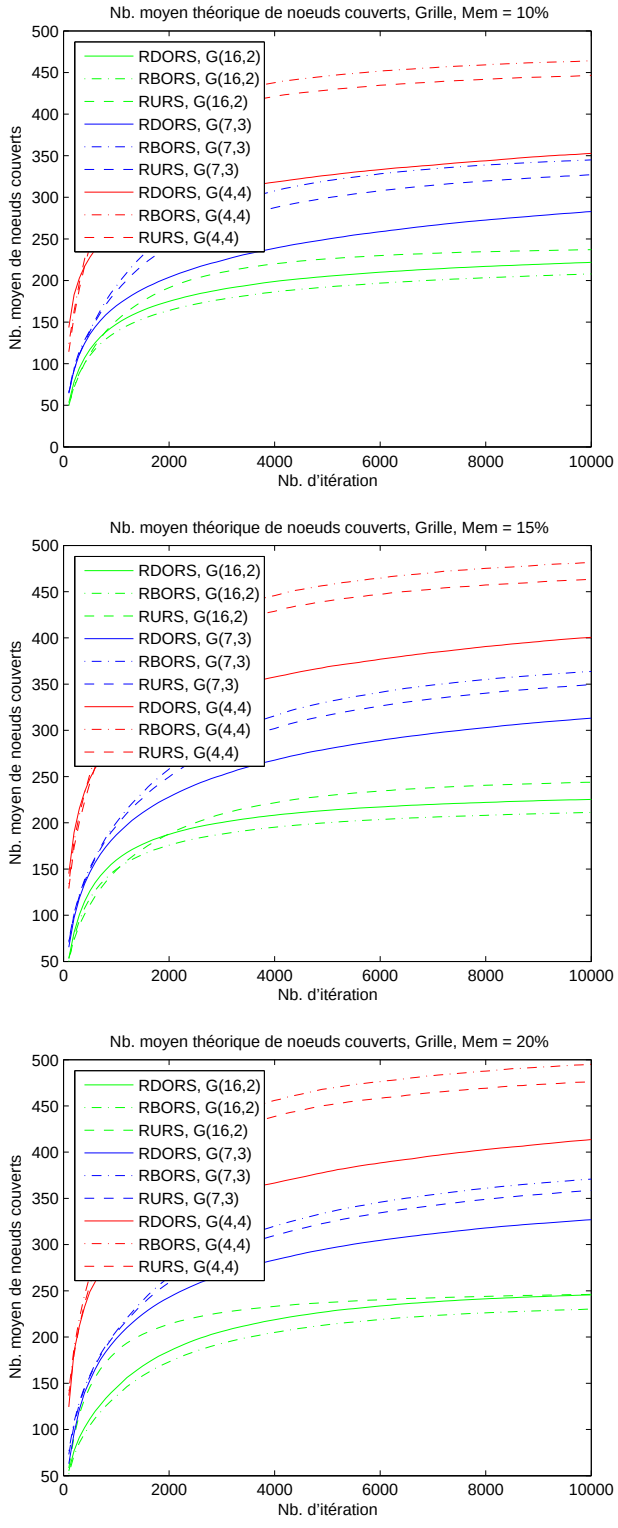


FIGURE 7.5 – Nombre moyen de nœuds couverts- cas des grilles

La figure 7.4 donne le résultat de comparaison du temps moyen de couverture pour trois grilles de dimensions différentes, où $G(L, d)$ dénote une grille hypercubique de degré d (i.e. dimension) et de longueur (d'un coté) $L + 1$. Il est clair dans cette figure que l'algorithme URS dépasse dans ses performances l'algorithme DORS. Sa supériorité est même plus claire que dans le cas des arbres. Il est également plus performant que RBORS de façon générale.

Le nombre moyen de nœuds couvert a été tracé en fonction du temps pour différentes grilles et différentes tailles de mémoire égales à 10%, 15% et 20% de la taille des graphes. Les résultats sont reportés en figure 7.5.

Ces résultats montrent l'importance des performances de RURS qui se montrent nettement meilleures que celles de RDORS et de RBORS. Cette différence est plus marquée pour les niveaux de couverture élevés. Elle est aussi plus claire dans les grilles que pour les arbres.

7.4 Comparaison expérimentale de URS, DORS et BORS

Plusieurs exemples ont été considérés et plusieurs mesures ont été calculées. Dans un premier temps, le temps de couverture moyen a pu être calculé grâce au fait que nous avons paramétré les exemples de sorte qu'ils aient des tailles moyennes (quelques milliers). Nous avons répété l'exécution des algorithmes un nombre suffisant de fois, en réalisant à chaque fois la couverture totale de l'espace d'états atteignable. Pour chaque algorithme et chaque pourcentage de couverture, la moyenne des temps d'exécutions mis par l'algorithme a été calculée, donnant ainsi son temps moyen de couvrir le pourcentage considéré du graphe. Les exemples utilisés ont des tailles et profondeurs différentes ce qui permet d'analyser leur comportement selon leurs formes. Dans un deuxième temps, et dans le but de comparer les algorithmes aléatoires partiaux et l'algorithme déterministe exhaustif BFS implémenté dans IF, nous avons utilisé les mêmes exemples, avec plus de processus et/ou données pour obtenir des graphes de tailles très grandes et inconnues. Nous avons utilisé, également, dans l'implémentation des algorithmes une table de hashage, facilitant, ainsi, la recherche et le stockage des nœuds visités.

7.4.1 Le temps de couverture

Chaque algorithme a été testé sur différents exemples de graphes. Les graphes considé- rés sont : *Token Ring Protocol*, *Client/Server Protocol*, *Alternating Bit Protocol*, *File System protocol*, *Fischer's Mutual Exclusion Protocol* et *Java Producer/Consumer Protocol*. La table 8.1 donne la taille (i. e. le nombre d'états) et la profondeur (i. e. la longueur du plus long chemin acyclique) de chaque exemple. Pour chaque exemple, nous avons répété l'expérimentation 100 fois et calculé le temps de couverture moyen de 60%, 70%, 80%, 90% et 100% du graphe. Les résultats sont reportés dans la table 7.2, le temps étant en secondes.

Exemple	Token	Server	Bitalt	Filesys	Fischer	Prodcons
Taille (nb. états)	550153	429118	325404	243234	184524	97285
Profondeur	115	35	19	39	11	23

TABLE 7.1 – Description des graphes

τ	Algo.	Token	Fischer	Server	Bitalt	Filesys	Prodcons
60%	URS	5.85 ± 0.06	0.68 ± 0.01	2.79 ± 0.05	3.29 ± 0.05	1.65 ± 0.02	0.64 ± 0.01
60%	DORS	1.01 ± 0.01	0.61 ± 0.01	1.64 ± 0.02	0.81 ± 0.01	1.22 ± 0.01	0.52 ± 0.01
60%	BORS	5.89 ± 0.10	0.67 ± 0.01	2.25 ± 0.04	2.45 ± 0.02	1.68 ± 0.01	0.70 ± 0.01
70%	URS	7.67 ± 0.11	1.02 ± 0.01	3.42 ± 0.05	3.93 ± 0.08	2.77 ± 0.03	0.90 ± 0.01
70%	DORS	1.46 ± 0.01	0.86 ± 0.01	1.96 ± 0.01	1.08 ± 0.02	1.69 ± 0.02	0.68 ± 0.01
70%	BORS	7.81 ± 0.15	0.93 ± 0.01	2.55 ± 0.04	3.12 ± 0.04	2.34 ± 0.03	0.72 ± 0.01
80%	URS	10.24 ± 0.14	1.41 ± 0.01	4.25 ± 0.07	6.43 ± 0.09	3.86 ± 0.05	1.06 ± 0.01
80%	DORS	2.13 ± 0.02	1.23 ± 0.01	2.64 ± 0.02	1.54 ± 0.02	2.83 ± 0.03	0.85 ± 0.01
80%	BORS	10.40 ± 0.20	1.16 ± 0.01	3.68 ± 0.05	4.14 ± 0.05	2.66 ± 0.03	0.97 ± 0.01
90%	URS	14.35 ± 0.16	2.21 ± 0.02	6.21 ± 0.13	7.57 ± 0.12	5.90 ± 0.09	1.50 ± 0.02
90%	DORS	3.35 ± 0.04	2.01 ± 0.02	4.23 ± 0.04	2.24 ± 0.05	4.92 ± 0.05	1.16 ± 0.02
90%	BORS	15.16 ± 0.25	1.90 ± 0.02	5.88 ± 0.11	5.59 ± 0.09	4.39 ± 0.06	1.80 ± 0.03
100%	URS	37.91 ± 0.50	17.18 ± 0.20	25.83 ± 0.49	21.44 ± 0.39	40.97 ± 0.75	6.12 ± 0.10
100%	DORS	34.10 ± 0.42	18.94 ± 0.23	22.02 ± 0.24	23.81 ± 0.45	41.64 ± 0.41	6.74 ± 0.11
100%	BORS	42.89 ± 0.77	17.91 ± 0.21	29.96 ± 0.54	18.54 ± 0.25	44.61 ± 0.63	8.91 ± 0.16

TABLE 7.2 – Temps moyen de couverture (secondes)

On remarque dans la table 7.2 que le temps de couverture de BORS est meilleur (plus petit) que DORS pour les graphes relativement larges, à savoir *Fischer*. En revanche, quand le graphe est relativement maigre, ici *Token*, *Filesys* et *Serveur*, c'est DORS qui emporte sur BORS. Ceci est conforme aux résultats théoriques précédemment obtenus dans le chapitre 5.

D'autre part, le temps de couverture de URS est meilleur (plus petit) que BORS et ce conformément à ce que nous avions noté précédemment que les graphes réels sont en général maigres par rapport aux arbres réguliers pour lesquels nous avons vu que BORS est meilleur que URS dans plusieurs cas d'arbres larges et que URS est meilleur que BORS lorsque l'arbre était maigre.

S'agissant maintenant, de la comparaison de l'algorithme URS avec DORS, on observe que URS donne un temps de couverture meilleur sauf pour l'exemple *Token* et que même pour cet exemple, URS est meilleur pour le temps de couverture totale de tout le graphe. Notons que le but de ce chapitre était de montrer la supériorité de URS dans la majorité des cas sur les deux autres algorithmes DORS et BORS, ce qui est

clairement visible dans le tableaux ci-dessus.

7.4.2 Ressources-Dépendants vs. Vérification Exhaustive

Nous avons expérimenté nos algorithmes également sur des graphes de tailles très grandes et inconnues. Ils ont été obtenus en augmentant le nombre de processus des exemples *Fischer*, *Server* et *Prodcons*. Nous avons comparé nos algorithmes randomisés entre eux et contre l'algorithme exhaustif BFS. Dans ces expérimentations, les algorithmes aléatoires sont réinitialisés à chaque fois que la mémoire est pleine, vu que l'espace d'états ne tient pas dans la mémoire principale, d'où les notations RURS, RDORS dans les courbes suivantes. Le nombre des états explorés de toutes les exécutions est collecté et présenté dans la figure 7.6 en fonction du temps.

On remarque que l'algorithme BFS stagne dès qu'il atteint la limite du nombre d'états qui peut être contenu dans la mémoire principale. Une répétition éventuelle de BFS couvre exactement les mêmes états dans le même ordre. Les algorithmes aléatoires répétés vont au delà de cette limite et peuvent explorer jusqu'à 40% de nœuds en plus. Notons que la limite de BFS apparaît à un nombre de nœuds différent pour chacun des trois exemples bien qu'ils utilisent tous la même mémoire principale. Ceci est dû au fait que pour chaque cas, l'espace mémoire nécessaire pour stocker un état est différent : il est plus grand dans *Prodcons* que dans *Server* et légèrement plus grand dans *Server* que dans *Fischer*.

On observe dans l'exemple *Fischer* que l'algorithme aléatoire stagne aussi après un certain temps. Selon les observations effectuées dans les expérimentations précédentes sur les graphes de tailles moyennes, ceci se produit lorsqu'on atteint près de 90% de couverture. Dans ce cas, l'exploration des derniers états devient excessivement difficile à cause de la redondance.

7.5 Conclusion

Nous avons présenté dans ce chapitre l'algorithme URS qui poursuit son exploration, à chaque étape, à partir d'un état choisi uniformément parmi tous les états visités. De ce fait, cet algorithme ne suit pas une marche aléatoire, dans le sens où il choisit de faire ses branchements uniformément à partir de tous les nœuds le long des chemins explorés. Il produit ainsi une couverture équilibrée et efficace. Nous avons établi pour URS les formules récursives des probabilités de couverture et d'atteignabilité, puis nous avons calculé le temps moyen de couverture et le nombre moyen de nœuds couverts pour comparer URS avec DORS et BORS. L'algorithme URS est meilleur dans la plupart des cas étudiés, pourvu que le graphe ne soit pas trop large ou trop maigre. Nous avons effectué également une comparaison expérimentale de ces algorithmes. URS montre sa supériorité sur DORS et BORS dans l'exploration de différents graphes réels. Il couvre l'espace d'états plus rapidement. Cependant, certains graphes de Model Checking sont très maigres et c'est DORS qui est meilleur dans ce cas. Nous avons montré aussi, à

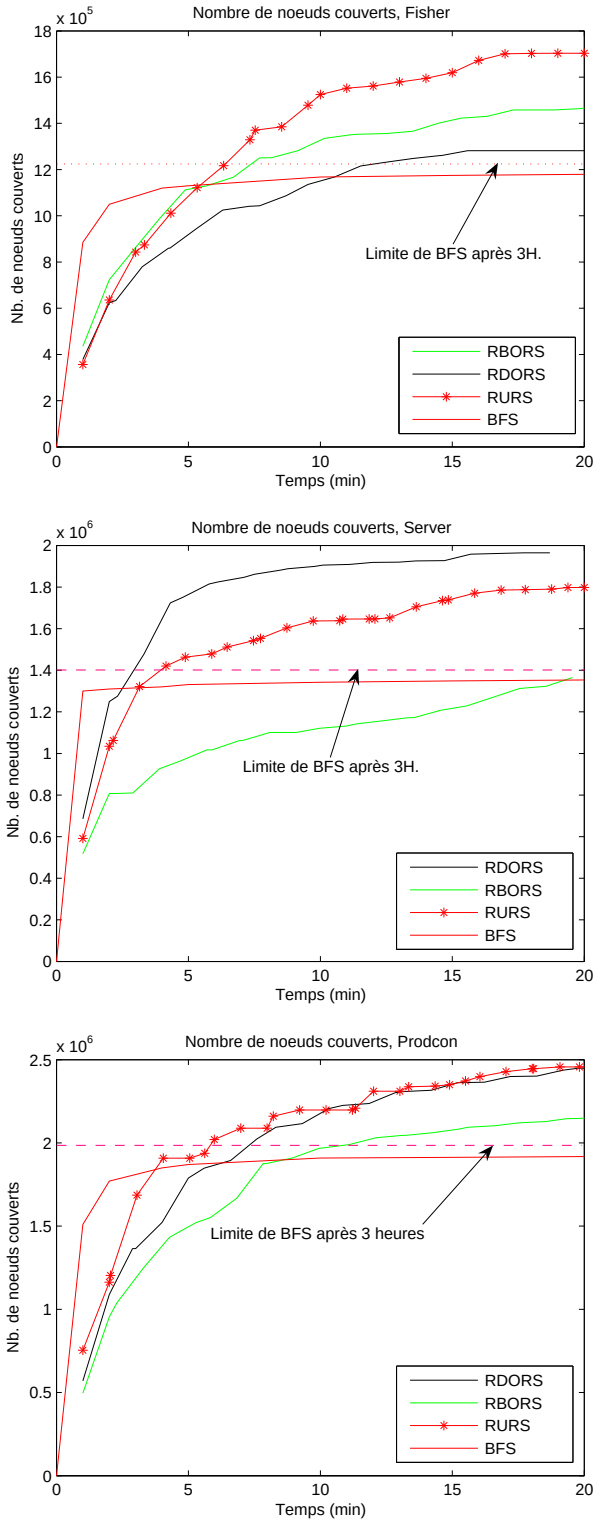


FIGURE 7.6 – L'évolution du nombre de nœuds couverts

travers les expérimentations que les algorithmes aléatoires lorsqu'ils sont réinitialisés et répétés plusieurs fois, notés alors RURS, RDORS et $RBORS$, explorent l'espace d'états plus efficacement qu'une exploration déterministe exhaustive basée sur l'algorithme BFS, et ce dans le cas de graphes de très grandes tailles dépassant la taille de la mémoire. Notre algorithme URS, en particulier, peut explorer jusqu'au 40% plus d'états que l'algorithme BFS.

Chapitre 8

Sélection paramétrée

8.1 Introduction

Comme nous l'avons vu dans les chapitres précédents, l'efficacité d'un algorithme d'exploration aléatoire dépend de la forme du graphe considéré. Bien que l'algorithme URS est assez performant de façon générale, on constate par exemple qu'un algorithme orienté profondeur comme DORS est plus convenable pour les graphes profonds. De même, dans le cas de graphes larges, il est préférable d'orienter l'exploration en largeur. Ceci nous pousse à penser à une version paramétrée de URS qui est l'algorithme nommé *Random Mixing Algorithme* RMA présenté dans ce chapitre. Cet algorithme permet de contrôler la distribution du choix aléatoire de l'exploration en largeur et en profondeur avec un certain *taux de mixage* mx. Ainsi, il peut aller en profondeur, en largeur ou d'une façon uniforme. Le taux profondeur/largeur est contrôlé à travers le paramètre mx qui peut être choisi selon la forme du graphe : maigre (grande profondeur, petit degré) ou large (grand degré, petite profondeur). L'algorithme RMA rentre dans le schéma général décrit dans le chapitre 3 en introduisant le paramètre *taux de mixage* dans le vecteur des paramètres. Sa version répétée (voir chapitre 4) sera également considérée, dans la même logique des algorithmes étudiés dans les chapitres précédents.

8.2 Classification des graphes

Plusieurs classifications ont été établies pour les graphes de model checking [55] pour déterminer le mode d'exploration et de vérification le plus approprié. Ces classifications se basent sur la structure du graphe : le nombre de composantes connexes, le degré moyen, le diamètre, les diamonds et les boucles. Certaines de ces classifications nécessitent d'importantes informations a priori sur le graphe à explorer. Nous montrons dans ce chapitre que la connaissance ou l'estimation d'un *facteur de densité* DF (Density Factor) permet d'orienter efficacement notre algorithme qui sera comparé avec DORS, BORS et URS. Plus précisément, nous montrons la cohérence entre le facteur de densité DF et le *taux de mixage optimal* mx_{opt}.

8.2.1 Le facteur de densité

Dans notre cas, on utilise une classification basée sur le facteur de densité DF. Ce paramètre donne une indication sur la forme générale du graphe et s'il est plutôt large ou maigre et à quel degré. Il est précisé dans la définition ci-dessous.

Définition 8.1 *Le facteur de densité DF est défini comme étant le rapport du degré moyen d'un graphe m à sa profondeur h (voir la définition 3.2, page 26) : $DF = \frac{m}{h}$. Si le graphe est de taille M et son degré moyen est inconnu, le DF est estimé, par référence à un arbre régulier, de taille $M = \frac{m^{h+1}-1}{m-1}$, comme étant $DF(T) = \frac{m^*}{h}$, où m^* est solution de l'équation $X^{h+1} - MX + M - 1 = 0$. Si par contre la profondeur h qui est inconnue, le DF est défini équivalemment comme : $DF = \frac{m \log(m)}{\log(M - \frac{M-1}{m})}$*

Le DF défini ci-haut est compris entre $\frac{1}{M-1}$, dans le cas le plus profond (correspondant au cas d'une chaîne), et $M - 1$, dans le cas le plus large (graphe en étoile). Cependant, on remarque dans la pratique que DF prend des valeurs assez petites. Un graphe est considéré large s'il possède un $DF > 0.25$ et maigre si $DF < 0.02$. Ces valeurs sont tirées de nos constatations dans notre étude théorique et expérimentale (voir section 8.6, page 94). Les graphes ayant un $0.02 < DF < 0.25$ sont considérés comme étant des graphes intermédiaires.

Notons que le facteur de densité DF est défini dans le but de décider d'orienter l'exploration en profondeur plus qu'en largeur ou inversement. Plus le DF est grand, plus l'algorithme doit favoriser l'exploration en largeur et le paramètre de mixage mx choisi doit être petit. Similairement, l'algorithme doit avoir tendance à explorer en profondeur lorsque le graphe est maigre ou profond (h est grand et donc DF est petit). Dans ce cas, le mx choisi doit être autour de 1.

8.3 Random Mixing Algorithm

RMA est basé sur l'algorithme URS, mais avec plus de flexibilité grâce au taux de mixage mx qui permet d'orienter l'exploration et de contrôler le taux largeur/profondeur en se basant sur la caractérisation du graphe considéré. Comme l'algorithme URS, RMA choisit le prochain nœud à explorer parmi les successeurs d'un nœud choisi aléatoirement dans l'ensemble des nœuds déjà visités V. En revanche, RMA fait la différence, contrairement à URS, entre deux sous ensembles des nœuds visités : V_I, pour les nœuds internes, et V_L, pour les nœuds feuilles. L'ensemble de tous les nœuds visités est donc $V = V_I \cup V_L$. Rappellons que l'ensemble V_L est l'ensemble des feuilles dans le sous graphe déjà exploré et non pas dans le graphe entier (voir page 44).

À chaque étape, une variable aléatoire binaire a est générée pour décider si le parent du prochain successeur sera choisi dans V_I ou dans V_L, avec $P[a = 1] = mx$ et $P[a = 0] = 1 - mx$. Lorsque $a = 1$, on choisit aléatoirement et uniformément le nœud père v dans l'ensemble V_L. Sinon, v est choisi dans V_I. Ensuite, un nœud u est choisi

uniformément parmi les successeurs de v. Le choix d'une feuille correspond à l'exploration en profondeur et le choix d'un nœud interne correspond à l'exploration en largeur. La forme générale de l'algorithme de mixage est donnée dans la figure 8.1 suivante :

```
V_I : ensemble des nœuds internes en mémoire ;
V_L : ensemble des nœuds feuilles en mémoire ;
N : taille maximale de V_I ∪ V_L ;
mx : paramètre de mixage ;
a : variable aléatoire ;
v, u : nœuds ;
i : entier ;

V_I ← ∅ ;
V_L ← v_0 ;
v ← v_0 ;
i ← 0 ;

Tant que (i < N) faire
    a ← générer une variable alétoire qui prend la valeur 1 avec probabilité mx et la
    valeur 0 avec probabilité 1 − mx ;
    Si ((a = 0 and W_I ≠ {}) ou (a = 1 and W_L = {})) Alors
        v ← choisir uniformément un nœud dans V_I ;
    Sinon
        v ← choisir uniformément un nœud dans V_L ;
        V_L ← V_L \ {v} ;
        V_I ← V_I ∪ {v} ;
    Fin Si
    u ← choisir uniformément un nœud dans succ(v) ;
    Si (u ∉ V) Alors
        vérifier(u) ;
        i ← i+1 ;
        V_L ← V_L ∪ {u} ;
    Fin Si
Fait
```

FIGURE 8.1 – Random Mixing Algorithm- RMA

Lorsque la mémoire principale est pleine, elle est vidée et l'algorithme RMA est réinitialisé et relancé. C'est la version répétée de l'algorithme qui sera notée RRMA.

8.4 Les deux cas extrêmes de l'algorithme RMA

Pour un paramètre $mx \in]0,1[$, l'algorithme RMA ne risque pas de se bloquer sur une feuille ou dans une boucle, puisque la probabilité que le nœud courant passe d'une feuille à un noeud interne, ou inversement, est non nulle et elle devient au bout de quelques étapes importante. En revanche, pour les paramètres particuliers $mx = 0$ et $mx = 1$, l'algorithme risque de se bloquer en cherchant à explorer une feuille (ou nœud interne), alors qu'il n'en existe pas, et on dit alors qu'on est dans un point fermé de l'exploration. Autrement, l'exploration est dite être dans un point ouvert. Pour $mx = 1$, on tente toujours d'explorer une feuille, i.e. effectuer un RW. En arrivant à un nœud sans successeur, l'algorithme se trouve bloqué, comme ça a été expliqué dans le chapitre 5 pour $DORS$. Dans ce cas, il continue l'exploration à partir d'un nœud interne quelconque et non pas du nœud courant bloquant (feuille). Dans le cas $mx = 0$, l'algorithme risque de se bloquer après avoir explorer complètement les nœuds internes, c'est à dire après avoir visiter tous leurs fils. Pour éviter ce genre de blocage, l'algorithme, une fois qu'il explore le nœud interne courant, passe à une feuille au lieu de retenter de trouver un nœud interne non exploré.

8.5 Étude théorique

Les mêmes statistiques que dans les chapitres précédents sont calculées pour l'algorithme RMA et ce conformément à notre méthodologie générale présentée dans la chapitre 4. Nous commençons par les résultats généraux suivants s'appliquant sur n'importe quel graphe. Ces résultats seront ensuite précisés dans les deux cas particuliers des arbres et des grilles.

8.5.1 Résultats généraux

Comme expliqué ci-dessus, l'algorithme RMA se ramené dans les deux cas extrêmes de $mx = 1$ et $mx = 0$ à DORS et BORS respectivement. Par conséquent, pour $mx = 1$ (resp. $mx = 0$), la récurrence élémentaire de RMA est celle de DORS (resp. BORS). Pour un paramètre $p \in]0,1[$, la récurrence élémentaire est exprimée dans le lemme ci-dessous où $\underline{w}_k = (w_1, ..., w_{k-1}, w_k)$ désigne, comme précédemment, la suite des nœuds visités distincts ordonnés par ordre de visite. L'ensemble $L(\underline{w}_k)$ (resp. $I(\underline{w}_k)$) désigne les nœuds feuilles (resp. les nœuds internes) dans $\underline{w}_k$.

Lemme 8.1 *Soit $\mathbb{P}(\underline{w}_k, n)$ la probabilité de couvrir, en n étapes, l'ensemble de nœuds $\underline{w}_k$.*

$$\mathbb{P}(\underline{w}_k, n) = \alpha_{RMA}(\underline{w}_k)\mathbb{P}(\underline{w}_k, n-1) + \beta_{RMA}\mathbb{P}(\underline{w}_{k-1}, n-1)$$

$$\textit{avec} \quad \alpha_{RMA}(\underline{w}_k) = mx \ . \frac{\sum_{v \in L(\underline{w}_k)} \frac{|C(v) \cap \underline{w}_k|}{|C(v)|}}{|L(\underline{w}_k)|} + (1 - mx) \ . \frac{\sum_{v \in I(\underline{w}_k)} \frac{|C(v) \cap \underline{w}_k|}{|C(v)|}}{|I(\underline{w}_k)|}$$

$$\textit{et} \quad \beta_{RMA}(\underline{w}_k) = mx \ . \frac{\sum_{v \in F(w_k) \cap L(\underline{w}_{k-1})} \frac{1}{|C(v)|}}{|L(\underline{w}_{k-1})|} + (1 - mx) \ . \frac{\sum_{v \in F(w_k) \cap I(\underline{w}_{k-1})} \frac{1}{|C(v)|}}{|I(\underline{w}_{k-1})|}$$

Preuve Voir l'annexe page 118. □

8.5.2 Cas des arbres

La récurrence élémentaire de l'algorithme RMA est simplifiée comme suit :

$$\begin{aligned}\mathbb{P}(\underline{L}_n = \underline{l}, \underline{I}_n = \underline{t}) &= \alpha_p \, \mathbb{P}(\underline{L}_{n-1} = \underline{l}, \underline{I}_{n-1} = \underline{t}) \\ &+ \sum_{j=1}^{h} \beta_{p,j} \, \mathbb{P}(\underline{L}_{n-1} = \underline{l} - 1_j, \underline{I}_{n-1} = \underline{t}) \\ &+ \sum_{j=1}^{h} \gamma_{p,j} \, \mathbb{P}(\underline{L}_{n-1} = \underline{l} - 1_j + 1_{j-1}, \underline{I}_{n-1} = \underline{t} - 1_{j-1})\end{aligned}$$

$$\begin{aligned}\text{où} \quad \alpha_p &= ((1-p)\frac{l+t-1}{t.m} + p\frac{l_h}{l}), \;\; \beta_{p,j} = (1-p)\frac{1}{t}(t_{j-1} - \frac{l_j + t_j}{m}), \\ \text{et} \quad \gamma_{p,j} &= p.\frac{l_{j-1}}{l}\end{aligned}$$

En exploitant ces récurrences, comme dans les chapitres 6 et 7 et selon notre schéma général d'étude, présenté dans le chapitre 5, nous obtenons les résultats théoriques présentés dans les figures 8.2 ci-dessous. Ces résultats donnent le temps moyen des différents niveau de couverture. On note que l'algorithme RMA utilisé avec un paramètre mx optimal arrive à réaliser un gain significatif par rapport au autres algorithmes. Ce gain en temps moyen de couverture peut atteindre 25%.

Nous avons calculé également le nombre moyen de nœuds couverts par les algorithmes répétés pour une mémoire de 15% de la taille du graphe. Les résultats sont illustrés dans les figures 8.3, et ce, comme précédemment, pour plusieurs arbres $T(h, m)$. Pour ce critère également, l'algorithme RMA pour un paramètre optimal mx_{opt} arrive à réaliser dans tous les cas des performances meilleures que les autres algorithmes.

8.5.3 Cas des grilles

Le cas de la grille est traité en appliquant les récurrences élémentaires conjointement avec les outils fournis au chapitre 5, entre autres, l'indexation particulière de la grille. Les courbes théoriques obtenues du temps moyen de couverture sont ceux de la figure 8.4. Pour le nombre moyen de nœuds couverts par les algorithmes répétés, les résultats sont illustrés dans les figures 8.5 pour plusieurs grilles $G(L, d)$, où $L + 1$ est la longueur de la grille et d son est degré, et pour un pourcentage de la taille mémoire par rapport à celle du graphe égal à 15% :

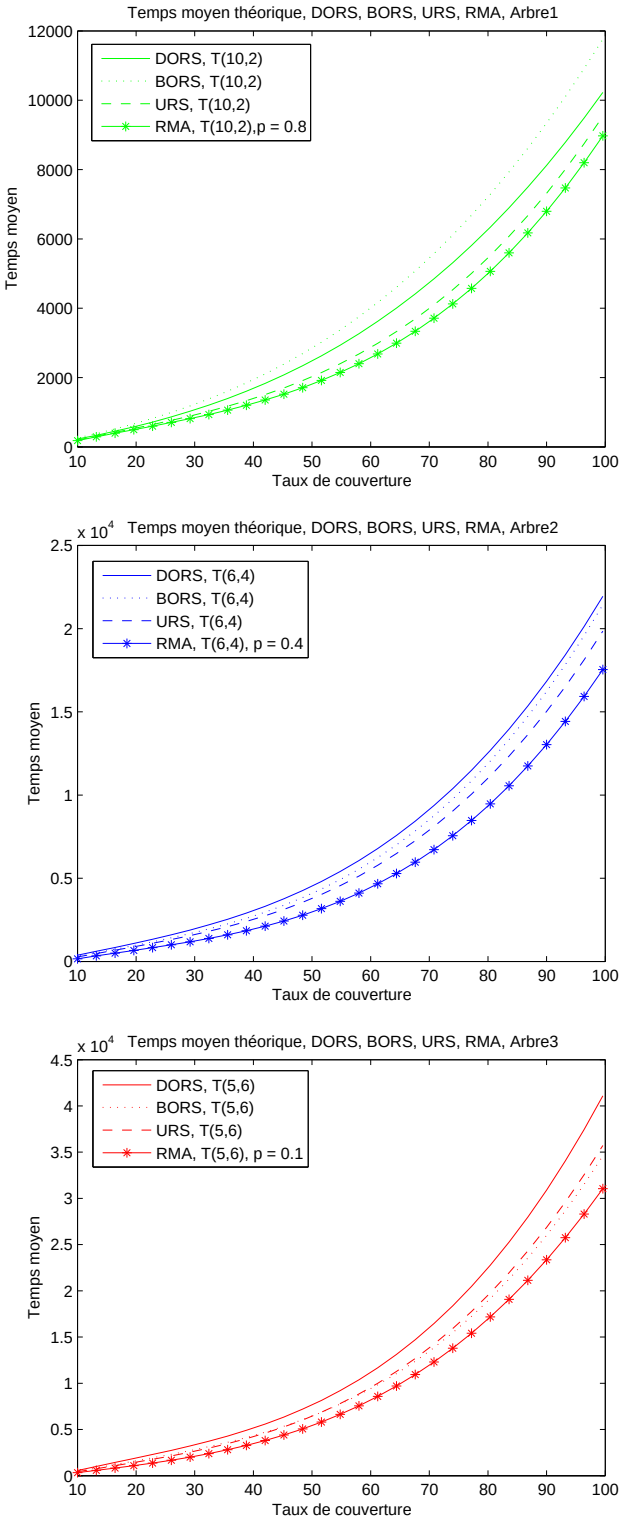


FIGURE 8.2 – Temps moyen de couverture par URS et RMA (cas arbre)

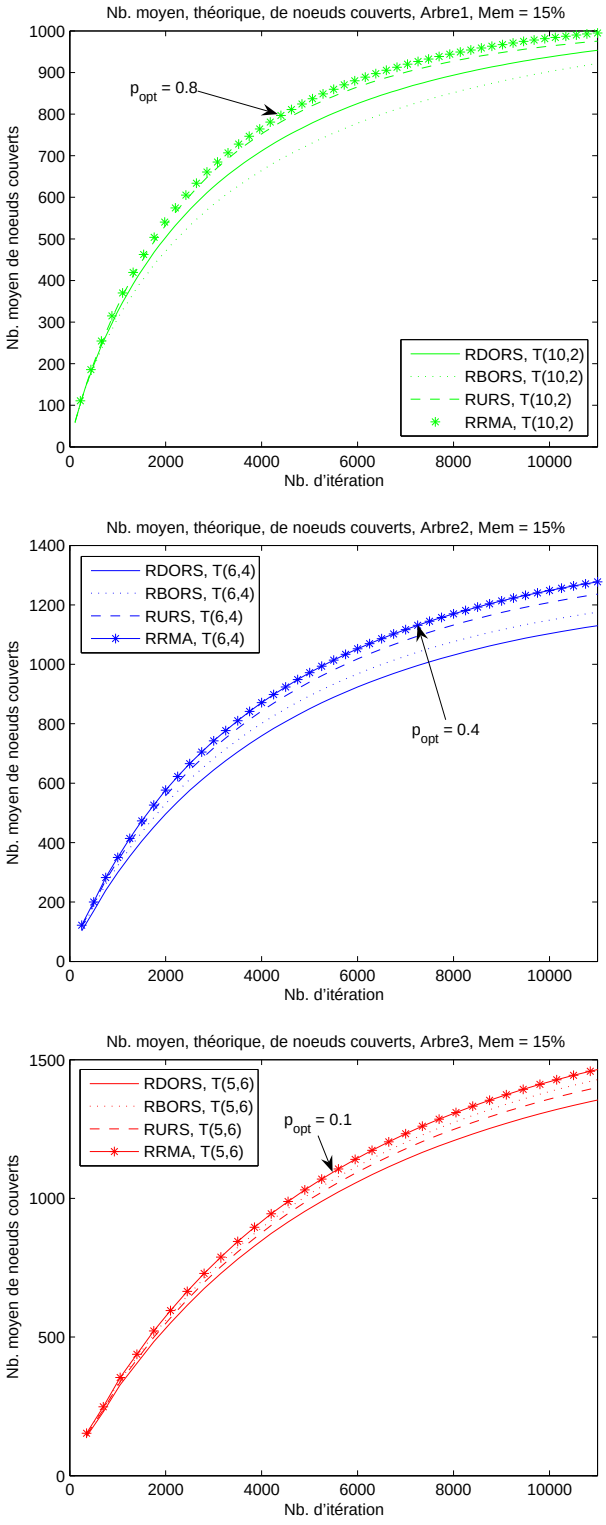


FIGURE 8.3 – Nombre moyen de nœuds couverts par URS et RMA (cas arbre)

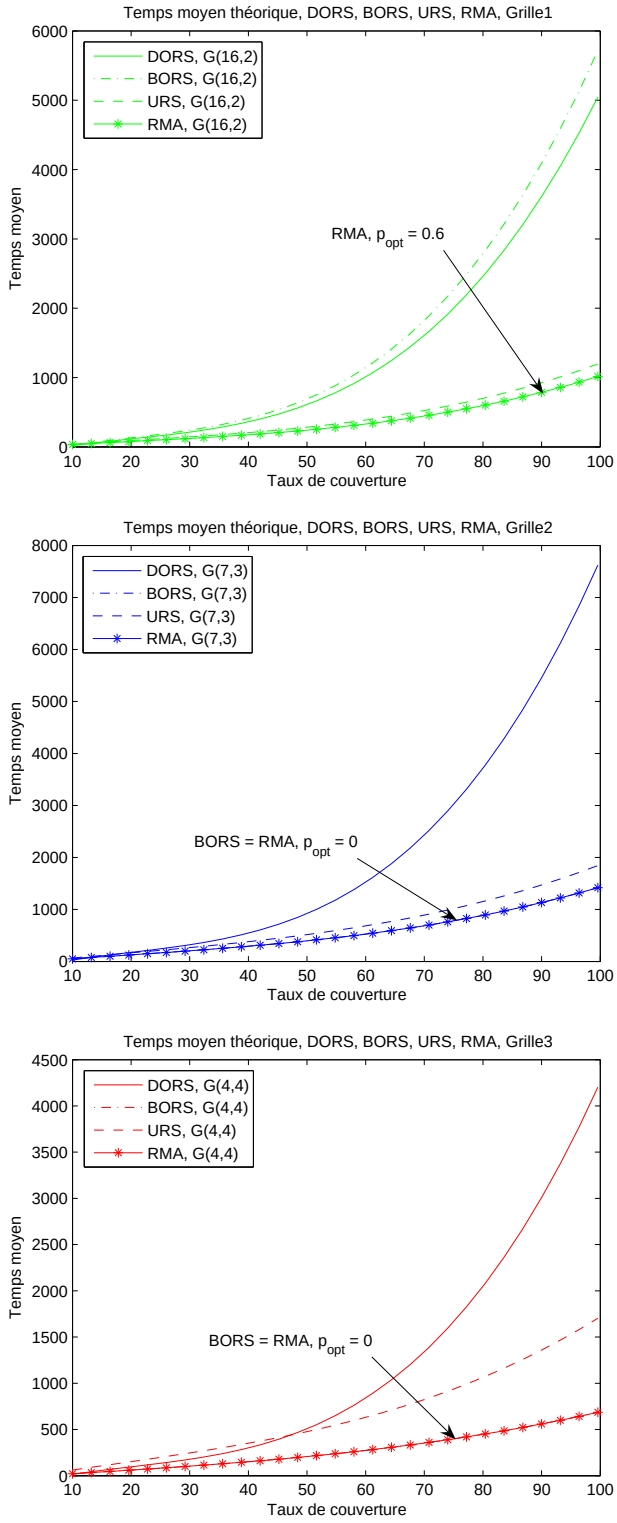


FIGURE 8.4 – Temps moyen de couverture par URS et RMA (cas grille)

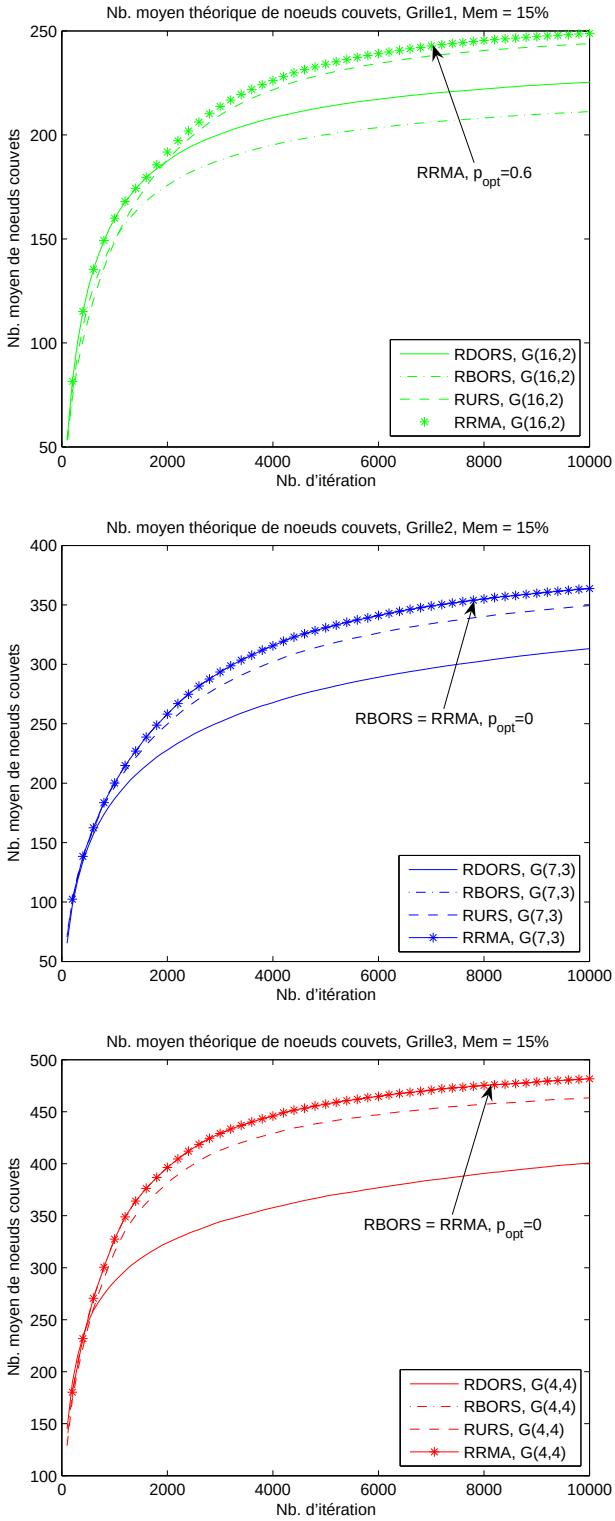


FIGURE 8.5 – Nombre moyen de nœuds couverts par URS et RMA (cas grille)

L'algorithme RMA est de nouveau meilleur que les autres pour tous les cas traités, pourvu que mx soit bien choisi. Sa supériorité est plus marquée dans le cas des grilles que dans le cas des arbres.

8.6 Expérimentations de l'Algorithme RMA

RMA est comparé à l'exploration déterministe BFS, déjà présente en IF, ainsi qu'aux algorithmes DORS, BORS et URS présentés dans les chapitres précédents.

Les exemples de graphes que nous considérons sont : *Token Ring Protocol*, *Client/Server Protocol*, *Alternating Bit Protocol*, *File System protocol*, *Fischer's Mutual Exclusion Protocol* et *Java Producer/Consumer Protocol*. Ces exemples de graphes ont des valeurs différentes du facteur de densité DF, ce qui permet l'analyse de leur comportement selon ce paramètre. La description de ces exemples (tailles, profondeurs et DF) est résumée dans la table ci-dessous :

Exemple	Token	Server	Bitalt	Filesys	Fischer	Prodcons
Nombre	1	2	3	4	5	6
Taille (nb. états)	550153	429118	325404	243234	184524	97285
Profondeur	115	35	19	39	11	23
DF (density factor)	0.01	0.04	0.1	0.03	0.27	0.07

TABLE 8.1 – Description des exemples

Au début, pour remontrer, sur des graphes pratiques, l'importance et l'impact du paramètre de mixage mx et sa relation avec le facteur de densité DF, nous avons expérimenté RMA avec plusieurs valeurs de mx sur plusieurs exemples de tailles moyennes. Nous avons choisi des tailles moyennes de graphes, pour être capable de répéter chaque expérimentation 100 fois et calculer le temps de couverture moyen sur ses exécutions. La table 8.2, contient les résultats obtenus sur le temps moyen d'exécution. Notons que la dernière colonne donne les temps moyens de couverture de URS.

On observe de la table 8.2, que le DF est inversement proportionnelle à la valeur optimale de mx. Notons que $mx = 0$ correspond à BORS et $mx = 1$ correspond à DORS. On peut dire que RMA avec une valeur $mx = 0$ explore mieux les graphes larges, typiquement pour $DF > 0.25$, alors que le choix $mx = 1$ est mieux approprié pour les graphes maigres (typiquement pour $DF < 0.02$). Pour les valeurs intermédiaires du DF, il y'a toujours une valeur de mx pour laquelle l'exploration de RMA est meilleure. Ceci est conforme avec les résultats théoriques obtenus précédemment.

Par la suite, et dans le but de comparer l'algorithme RRMA avec l'exploration BFS, nous avons utilisé nos exemples avec un nombre plus grand de processus et données

	RRMA, plusieurs valeurs de mx					RURS
	0	0.25	0.50	0.75	1	
1	42.89 ± 0.77	41.51 ± 0.45	38.76 ± 0.54	38.02 ± 0.69	34.10 ± 0.42	37.91 ± 0.50
2	29.96 ± 0.54	28.23 ± 0.28	27.04 ± 0.46	25.97 ± 0.25	22.02 ± 0.24	25.83 ± 0.49
3	18.54 ± 0.25	16.12 ± 0.21	21.61 ± 0.36	22.70 ± 0.33	23.81 ± 0.45	21.44 ± 0.39
4	44.61 ± 0.63	42.94 ± 0.59	39.16 ± 0.66	36.12 ± 0.66	41.64 ± 0.41	40.97 ± 0.75
5	17.91 ± 0.21	14.36 ± 0.19	17.30 ± 0.19	20.66 ± 0.23	18.94 ± 0.23	17.18 ± 0.20
6	8.91 ± 0.16	10.07 ± 0.10	7.43 ± 0.14	5.02 ± 0.09	6.74 ± 0.11	6.12 ± 0.10

TABLE 8.2 – Temps moyen de couverture (secondes)

pour obtenir des graphes de très grandes tailles. Nous avons exécuté les explorations BFS, RURS et RRMA avec plusieurs valeurs de mx et le nombre des états explorés de toutes les répétitions ont été collectés pour être comparés. Les courbes montrant l'évolution de l'exploration ont été tracées en fonction du temps. Elles sont données dans les figures 8.6.

Comme on l'a déjà prévu et constaté au chapitre précédent, on observe dans la figure 7.6, que l'exploration exhaustive BFS stagne dès l'atteinte de la limite N du nombre maximal d'états pouvant être contenus en mémoire. Les algorithmes aléatoires répétés dépassent cette limite et RRMA, lorsqu'il est appliqué avec le paramètre de mixage approprié, est généralement meilleur que les autres pour les graphes larges et maigres. Il arrive à explorer jusqu'à 45% de plus que BFS.

Ce gain de 45% est observé après 20 mn d'exécution. La courbe de RRMA est de pente moins importante aux étapes avancées de l'exploration comme on l'a remarqué également dans notre étude théorique. Néanmoins, cette courbe progresse toujours, contrairement à BFS qui a une progression quasi nulle après l'épuisement de la mémoire. On montre dans cette courbe que dans un temps de 10 mn, on dépasse la limite atteinte par BFS dans 3 heures. Ces résultats sont obtenues en vu d'un temps d'exécution donné. En se donnant plus de temps, les gains réalisés sont plus importants.

Il est important de noter que nos algorithme, vu leur caractère aléatoire, commencent par explorer rapidement l'espace d'états. Le rendement de cette exploration (nombre d'états explorés par unité de temps) diminue avec le temps et l'exploration des derniers états sera excessivement lente. On conseille à ce stade d'abandonner l'exploration et se contenter du taux de couverture réalisé. Intuitivement et selon notre étude, ceci se produit lorsqu'on s'approche d'une couverture totale (e.g., Figure 7.6 : Server, $mx = 1$). Ce manque d'efficacité se produit aussi lorsque la valeur du mx n'est pas du tout adaptée au graphe exploré (e.g., Figure 7.6 : Fischer, $mx = 0.75$ et $mx = 1$).

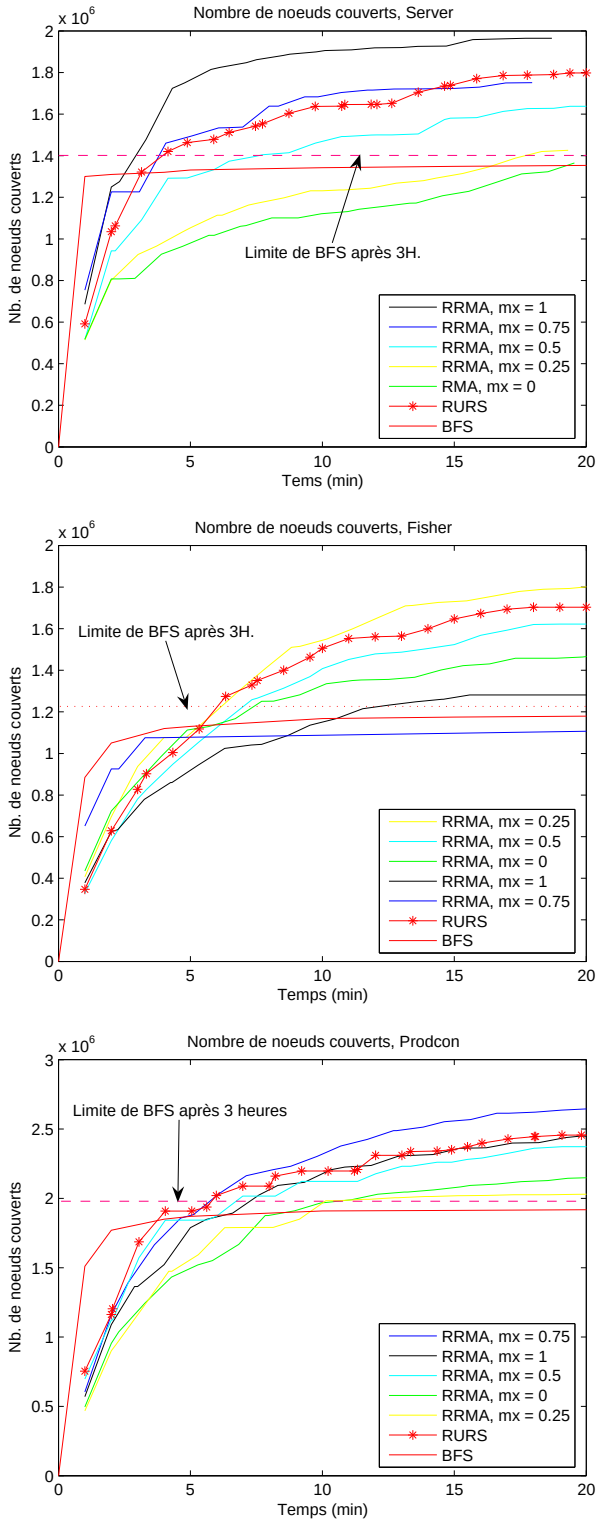


FIGURE 8.6 – The evolution of number of covered nodes

8.7 Conclusion

Nous avons présenté dans ce chapitre l'algorithme RMA qui constitue une version paramétrée de URS. Il permet de contrôler l'exploration en profondeur et en largeur selon le taux de mixage désiré. Nous avons présenté également la version répétée de cet algorithme qui est très efficace pour explorer de grandes portions d'espaces d'états très larges. Notre étude, à la fois théorique et expérimentale, a montré que le temps d'exploration de RMA, comparé à URS, est toujours meilleur pour les valeurs les plus appropriées du paramètre de mixage mx. Nous avons vu ensuite dans le cas de graphes de très grandes tailles, que la version réinitialisée RRMA, lorsqu'elle est appliquée avec le bon paramètre de mixage, arrive à explorer jusqu'au 45% plus de l'espace d'états que BFS. On a montré aussi une cohérence entre la caractérisation du graphe par le *facteur de densité* DF et la valeur optimale du *taux de mixage* mx_{opt}.

Chapitre 9

Actualisation

9.1 Introduction

En plus de la sélection, nous étudions dans ce chapitre d'autres améliorations possibles dont on peut tirer profit. D'une part, la fonction d'actualisation de l'ensemble des états visités qui peut être effectuée selon différentes stratégies et non seulement par réinitialisation comme nous avons pu voir jusque là. D'autre part la condition d'arrêt des répétitions avec réinitialisation, c'est à dire le calcul d'une borne sur le nombre de répétition R en fonction des performances souhaitées et de certaines informations sur le système via l'application de l'approche Monte Carlo généralisée.

9.2 Les techniques de remplacement des états en mémoire

Le schéma général des algorithmes d'exploration présenté dans le chapitre de modélisation 4 prévoit une fonction *actualiser* qui permet de mettre à jour l'ensemble des états stockés V à chaque fois qu'un nouvel état est visité. Cette opération d'actualisation n'est autre qu'une technique de remplacement des états. C'est une des premières techniques conçues pour remédier au problème d'explosion de la taille de l'espace d'états. Elle est utilisée principalement pour effectuer une exploration totale de l'espace d'états, et ce en ne gardant dans le cache qu'un sous ensemble des états explorés. Dans ce chapitre, nous présentons la technique du caching des états et nous montrons via les expérimentations, l'efficacité de son application à nos algorithmes dans le cadre de l'exploration partielle.

9.2.1 Le remplacement des états et l'exploration aléatoire

Les techniques de remplacement, appelées aussi "caching", peuvent être appliquées à l'exploration aléatoire de la même façon que celle décrite pour l'exploration exhaustive, et ce en suivant le schéma générique du chapitre 3 et en remplaçant la fonction d'actualisation par la stratégie de remplacement en mémoire choisie. Pour un algorithme randomisé donné A, la version associée aux techniques de remplacement (ou "caching") sera notée CA. Au début, à chaque fois qu'un nouvel état est visité, on le

stocke en mémoire. Si un nouvel état v vient d'être visité alors que le vecteur V est plein, on choisit un ancien état dans V pour le remplacer par v. Plusieurs stratégies peuvent être utilisées. Dans ce chapitre, nous expérimentons trois d'entre elles : FIFO, random et RLFU (Random Least Frequently Used) et les comparons avec l'algorithme BFS-caching décrit dans [64]. En effet, la stratégie RLFU que nous proposons est une combinaison entre les stratégies Random, généralement efficace en pratique, et LFU, qui consiste à remplacer l'état ayant la moindre fréquence d'utilisation. Les états les moins utilisés sont ceux n'ayant aucun successeurs car ils ne donnent lieu à aucun choix de nœud dans la suite explorée. RLFU choisit aléatoirement un nombre S d'états visités et remplace celui qui a la moindre fréquence d'utilisation parmi eux.

Notons que dans ce cas d'utilisation des stratégies de remplacement, il y aura pas besoin des répétitions (ou réinitialisation) dans algorithmes (i.e. la version *RA*) car l'espace mémoire se regénère tout seul, état par état, selon la stratégie de remplacement et non pas en supprimant tous les états à la fois pour réinitialiser l'algorithme. De ce fait, la réinitialisation peut être considérée comme une stratégie particulière du caching des états. Intuitivement, le fait de garder certains états en mémoire et ne remplacer que des états bien choisis est le plus souvent avantageux, car il permet de réduire la redondance de manière plus efficace que la réinitialisation qui supprime tous les états d'un seul coup et ne garde aucune trace des états déjà visités et qui peuvent être de bon points de départ ou de continuation de l'exploration. En revanche, la réinitialisation permet de changer radicalement la direction de l'exploration, ce qui évite les situations de blocage dans des composantes fermées par exemple.

9.2.2 Etude expérimentale des techniques de remplacement

L'implémentation a été réalisée sur la plate forme IF. Les algorithmes CURS, CRMA et RBFS sont testés sur trois exemples de graphes : *Client/Server Protocol*, *Fischer's Mutual Exclusion Protocol* et *Java Producer/Consumer Protocol*. Ces protocoles ont différentes caractéristiques : tailles, types et complexités. Plusieurs mesures ont été calculées pour chaque algorithme, sur chaque exemple de graphe et pour plusieurs tailles de la mémoire (ou "cache").

9.2.2.1 Comparaison des techniques de remplacement

Dans [64], il a été rapporté que l'algorithme RBFS (i.e. BFS avec un remplacement aléatoire) arrive à réaliser un gain en nombre d'états exploré de 30% par rapport à un algorithme déterministe utilisant la même taille de mémoire. Afin de comparer CURS et CRMA avec RBFS, nous avons utilisé une mémoire de 10% de la taille du graphe. En outre, différentes stratégies de remplacement ont été considérées : FIFO, qui a été montrée comme une des meilleures politiques dans la littérature, la stratégie aléatoire (random) et notre stratégie RLFU présentée précédemment. Nous avons exécuté chaque algorithme 100 fois sur l'exemple *Fischer*, puis nous avons tracé le taux de couverture moyen réalisé dans chaque cas en fonction du temps. Les courbes obtenues sont

présentés dans la figure 9.1.

Les valeurs maximales des intervalles de confiance du taux de couverture sont présentées dans la table 9.1.

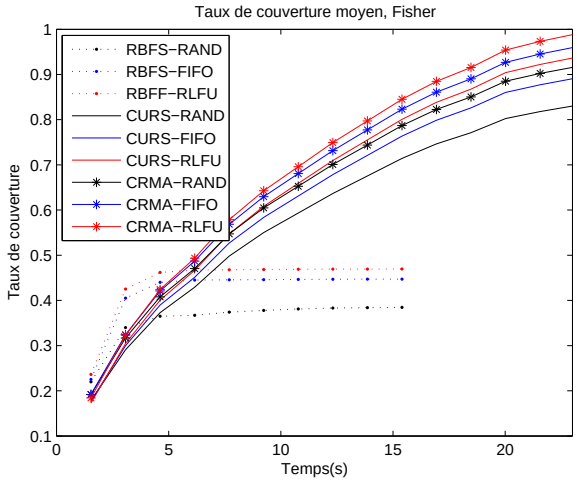


FIGURE 9.1 – Taux de couverture moyen pour CURS, RMA et RBFS- Fischer

Strat. Cach.	FIFO	Random	RLFU
RBFS	0.0054	0.0060	0.0064
CURS	0.0100	0.0089	0.0135
CRMA	0.0102	0.0111	0.0099

TABLE 9.1 – Les valeurs maximales des intervalles de confiance sur le taux de couverture pour différentes stratégies de remplacement

À partir des figures 9.1, on remarque que, pour la même taille de mémoire et la même stratégie de remplacement, CRMA et CURS, couvrent considérablement plus d'états que Random-BFS (en moyenne 100% et jusqu'à 140%). On peut expliquer cela par le fait que les algorithmes randomisés permettent une exploration bien distribuée sur le graphe et donc moins de redondance. En effet, un algorithme ordonné comme BFS explore toujours les états du même voisinage. Ces états explorés sont concentré dans la partie supérieure du graphe. Cependant, les états explorés par un algorithme randomisé comme URS, sont aléatoirement dispersés sur tout le graphe. Notons par ailleurs que le taux de redondance augmente particulièrement lorsque la couverture maximale est approchée. Le nombre des états nouvellement visités est quasiment nul. Dans une telle situation, il est généralement préferable d'abandonner l'exploration et se contenter du taux réalisé.

9.2.2.2 Influence de la taille de mémoire

Pour étudier l'effet de la taille de la mémoire disponible sur l'efficacité de l'exploration, on a expérimenté les algorithmes CURS et CRMA avec la stratégie de remplacement FIFO sur trois graphes et pour différentes valeurs de la taille de la mémoire. Le tableau 9.2 donne le pourcentage de la taille mémoire à la taille du graphe. Pour chaque graphe et chaque valeur de N (la taille mémoire), l'algorithme a été exécuté 100 fois et le taux de couverture moyen a été calculé. Les résultats sont présentés dans le tableau 9.3 donnant les couvertures maximales dans chaque cas. Les valeurs maximales des intervalles de confiance du taux de couverture sont présentées dans le tableau 9.4.

Taille mémoire	10000	20000	50000
Server (429 118)	2%	4%	10%
Fischer (184 524)	5%	10%	25%
Prodcons (97 285)	10%	20%	50%

TABLE 9.2 – Les pourcentages des tailles mémoires aux tailles des graphes

Taille mémoire	10000	20000	50000
Server (429 118)	60%	76%	99%
Fischer (184 524)	78%	92%	100%
Prodcons (97 285)	95%	100%	100%

TABLE 9.3 – Les couvertures max. de CURS, CRMA et RBFS pour différentes mémoire

Tailles Mém.	10000	20000	50000
Server (429 118)	0.0325	0.0303	0.0233
Fischer (184 524)	0.0138	0.0108	0.0103
Prodcons (97 285)	0.0862	0.0935	0.0855

TABLE 9.4 – La valeurs maximale des intervalles de confiance sur le taux de couverture pour plusieurs tailles de mémoire

Avec une taille mémoire de seulement 2% de la taille de l'espace d'états, nous avons obtenu une couverture de plus de 60% de l'espace d'états, soit 30 fois plus d'états qu'un algorithme déterministe de type BFS, limité traditionnellement par le manque de mémoire. Dans la diagonale de la table 9.3, avec une mémoire de 10% et indépendamment de la taille de l'espace détats, nous avons pu s'approcher de la couverture totale (environ 95% de la taille de l'espace d'états est couverte). Ces résultats de couverture sont obtenus en un temps donné qui sert de référence de la comparaison, et qui est relatif à la taille de l'espace d'état de l'exemple étudié (environ 25 mn pour l'exemple Server, 17 mn pour l'exemple Fischer et 7 mn pour Prodcons)

9.2.2.3 Distribution des états explorés

L'un des avantages des algorithmes d'exploration aléatoire est la bonne distribution des états explorés même s'il y'en a peu. Dans certains cas, un ensemble réduit d'états bien distribués, peut être suffisamment représentatif du comportement du système. Par exemple, à partir d'un état où les canaux sont mi-pleins, il faut traverser plusieurs transitions pour arriver à des états de débordement. Dans le but d'observer une telle distribution, nous avons choisi une petite taille pour la mémoire $N = 10000$, puis nous avons lancé l'algorithme CURS avec la stratégie FIFO, pour atteindre un ensemble R^0 de N états. Ensuite, nous avons cherché l'ensemble D^1 des états non inclus dans R^0 et pouvant être atteints en une étape à partir des états de R^0, et nous avons posé $R^1 = R^0 \cap D^1$. On a construit de la même façon les ensembles $R^2, R^3, ...$ etc. On peut dire que les états de R^0 sont bien positionnés (: distribués ou dispersés), dans le graphe si D^1, et par suite R^1, est large. Ceci dépend évidemment de la profondeur du graphe, du degré moyen des états, du taux de connections, ... Pour chacun des trois graphes, on a calculé successivement $R^0, ..., R^{20}$ et observé leurs progressions. Les résultats sont présentés dans la figure 9.2 suivante.

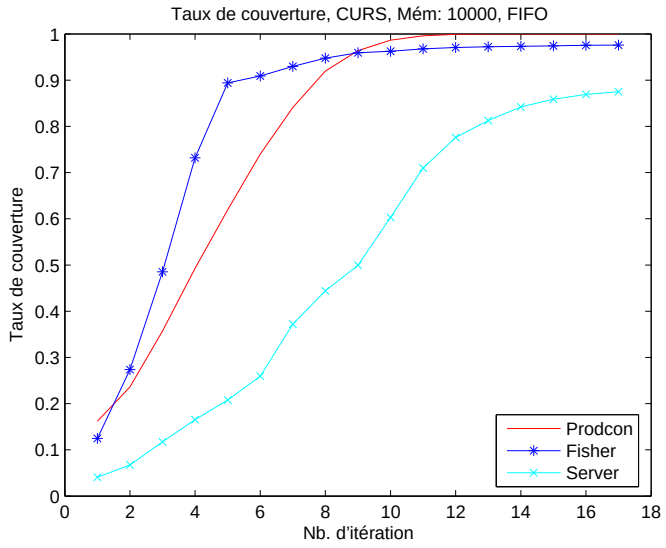


FIGURE 9.2 – Atteignabilité des états du graphe

On remarque qu'après environ une dizaine d'itérations, on peut couvrir la majorité des états du graphe, ce qui reflète la bonne distribution des états, contrairement au cas trop ordonné d'un algorithme déterministe tel que BFS.

9.3 L'approche Monte Carlo Model Checking

Dans cette section, nous nous penchons sur l'étude de nos algorithmes sous l'approche Monte Carlo. L'enjeux principal consiste à préciser le seuil de répétition $\mathcal{R}$ pour un degré de confiance donné. Nous montrons qu'en utilisant l'approche Monte Carlo Model Checking Généralisée GMC^2 que nous proposons, on peut obtenir plus de performances sur le nombre de répétitions $\mathcal{R}$, ainsi que sur le temps d'exécution, que l'approche MC^2 se basant sur un simple RW. En outre, grâce à l'approche Monte Carlo Model Checking, nous complétons nos algorithmes répétés RA, par la précision du nombre de répétitions $\mathcal{R}$. Ce nombre qui est resté jusqu'au là imprécisé ou fixé a priori. Notre approche Monte Carlo, qui aboutit à l'algorithme GMC^2, est ensuite proposée et une comparaison de performances théoriques avec MC^2 est menée. Nous finissons par une comparaison de performances expérimentales validant notre approche.

9.4 L'approche GMC^2

L'application de l'approche Monte Carlo au Model Checking s'avère générale et ne se limite pas à RW. Un algorithme d'exploration aléatoire quelconque A peut être appliqué avec une approche Monte Carlo : L'algorithme A est relancé à partir du nœud initial à chaque fois qu'une condition d'arrêt σ est rencontrée (l'atteinte d'une feuille par exemple ou d'un cycle, ou l'épuisement d'un nombre d'étapes fixé). Dans le cas de nos algorithmes d'exploration : DORS, BORS, URS et RMA, les réitérations Monte Carlo sont arrêtées lorsqu'un nœud cible est rencontré, puisque le but de l'exploration est, essentiellement, l'atteinte d'un état représentant un bug, par exemple. Autrement, on s'arrête lorsqu'un certain nombre d'échantillons (itérations) $\mathcal{R}$ est atteint.

Comme nous l'avons signalé précédemment l'enjeu principal dans l'approche Monte Carlo Model Checking est le choix du paramètre $\mathcal{R}$ qui assure, avec une certaine confiance, que la propriété ϕ en question est satisfaite par le système.

L'algorithme MC^2 utilise une borne inférieure ϵ de q_Z, la probabilité qu'une itération de RW permette de détecter un bug i.e. un nœud défectueux dans le graphe d'états. Cependant, la probabilité q_Z dépend, outre l'existence même d'un bug, de la localisation de ce bug, du nœud défectueux, et de la probabilité d'atteignabilité d'un tel nœud. Notons aussi que le système peut contenir plusieurs nœuds défectueux.

La première idée de notre approche consiste à noter l'applicabilité de l'approche Monte Carlo à tout algorithme d'exploration A et au fait que la majorité de ces algorithmes permettent d'obtenir des probabilités d'atteignabilité des nœuds $\mathbb{P}_a(A)$ meilleures que celles obtenues par RW. C'est le cas, en particulier, pour nos algorithmes comme nous l'avons vu dans les chapitres précédents. Notons que ces probabilités d'atteignabilité sont en général petites, ce qui rend le problème de fixer ou choisir une borne inférieure de $q_Z(A)$ appropriée (proche de $q_Z(A)$) plus difficile, et ce autant que $q_Z(A)$ (ou $\mathcal{P}_a(A)$) est proche du zéro. Ce problème est critique étant donné que le nombre

d'échantillons R est inversement proportionnel à $log(1-\epsilon)$ ou encore à ϵ pour $q_Z(A)$ et ϵ petits :

$$R = \log(\delta)/\log(1-\epsilon) \sim \log(1/\delta)/\epsilon$$

Un mauvais choix de ϵ peut, donc, conduire à une valeur très grande de R. Le gain dans $q_Z(A)$ par rapport à q_Z, associé à RW, permet donc d'obtenir des valeurs de R nettement plus petites. Ayant cette idée, notre algorithme GMC^2 constitue une généralisation de MC^2. Il est présenté dans la figure 9.3.

//Structures de données :
ϵ, δ : **réels, avec** $0 < \epsilon \leq q_Z$ **et** $0 < \delta \leq 1$;
T : **réel** ;
i : **entier** ;

//Initialisation :
$T = \log(\delta)/\log(1-\epsilon).$;

//Corps de l'algorithme :
Pour i **de** 1 **à** T **faire**
  Exécuter A, jusqu'à la détection d'un état s présentant une violation de ϕ ou atteinte du nombre d'états maximal N.
  Si (s présente une violation de ϕ) **Alors**
    retourner(Faux, s)
  Sinon
    retourner(Vrai, $\mathbb{P}(X > T) < \delta$)
  Fin Si
Fin Pour

FIGURE 9.3 – L'algorithme GMC^2

La deuxième idée revient au fait que nos algorithmes A gardent au cours d'une itération tous les nœuds intermédiaires parcourus. Ceci implique, intuitivement, qu'une itération de l'algorithme A, de longueur (nombre d'étapes) $L' = kL$ permet une meilleure atteignabilité (en nombre d'états) que k itérations de longueur moyenne L de RW. Le temps global, tenant compte du nombre d'itérations Monte Carlo et de la longueur moyenne d'une itération, permettant l'atteinte d'un nœud défectueux sera donc plus petit pour nos algorithmes GMC^2 que pour MC^2 comme nous allons le voir clairement dans nos expérimentations. Par conséquents, malgré qu'une itération d'un de nos algorithmes prend en moyenne plus de temps qu'une itération de RW, le gain global en temps sera au profit de nos algorithmes.

Analysons sur un exemple ce gain en temps global. Soit L le nombre moyen d'étapes dans une itération de RW. Le nombre d'itération de A (un de nos algorithmes d'exploration) pour assurer une confiance de $1-\delta$ est donné par :

$$T(A) = log(\delta)/log(p_z(A)).$$

Donc le nombre total d'étapes associées à RW, $N_T(RW)$, et à A, $N_T(A)$, sont :

$$\begin{aligned} N_T(RW) &= L.log(\delta)/log(p_Z(RW)) \\ \text{et } N_T(A) &= L'.log(\delta)/log(p_Z(A)) \end{aligned}$$

Avec $L' = kL$, on a :

$$N_T(A) < N_T(RW) \rightarrow p_Z^k(RW) \geq p_Z(A)$$

Remarquons que $p_Z^k(RW) = p_Z(RW_k)$ où l'on a désigné par RW_k la concaténation de k itérations de RW, avec L étapes chacune, en réinitialisant du nœud initial à chaque fois. Or au cours d'une itération à L' étapes, l'algorithme A enregistre les nœuds explorés qui peuvent servir tous comme nœuds initiaux pour atteindre le nœud cible (défectueux). Ainsi en le même nombre d'étapes L', la probabilité de ne pas atteindre ce nœud cible par RW_k est supérieure à celle associée à A. Par conséquent, $p_Z^k(RW) \geq p_Z(A)$ et $N_T(A) < N_T(RW)$ pour la même performance $(1-\delta)$.

9.5 Comparaison expérimentale

Nous présentons dans le tableau 9.5 ci-dessous les résultats de comparaisons expérimentales qui ont été effectuées pour les algorithmes répétés URS, RMA et RW. Pour chaque exemple et chaque algorithme, nous avons effectué 100 expérimentations. On calcule dans chacune le nombre de répétitions et le temps nécessaire pour la détection d'un nœud cible fixé a priori.

Ex	RW		URS		$mx=0$		$mx=0.2$		$mx=0.4$		$mx=0.6$		$mx=0.8$		$mx=1$	
	R	T	R	T	R	T	R	T	R	T	R	T	R	T	R	T
1	19567	32.41	62	0.65	62	0.62	57	0.54	58	0.57	56	0.59	445	5.76	897	6.27
2	24543	40.12	378	5.89	323	4.49	320	4.35	291	4.04	495	6.47	784	6.46	813	6.35
3	24017	39.88	383	5.95	148	1.96	142	1.99	178	2.42	421	7.49	977	6.99	978	6.36
4	32144	45.31	183	1.93	98	1.20	105	1.32	107	1.26	122	1.54	438	6.18	945	6.53
5	47856	56.53	177	2.62	214	2.42	153	1.96	176	2.08	194	2.46	654	8.06	994	8.20
6	92765	114.28	893	22.70	510	9.21	594	9.93	617	11.04	893	18.86	900	24.76	987	24.55

TABLE 9.5 – Nombre de répétitions et temps de la détection du nœud cible

Les moyennes des temps d'atteignabilité des nœuds désignés sont reportés pour les différents algorithmes dans le tableau 9.5. Les intervalles de confiance correspondants sont reportés dans le tableau 9.6. $\mathcal{R}$ désigne le nombre de répétitions de l'algorithme avant l'atteinte du nœud cible et T le temps d'exécution.

Ces résultats montrent que dans le cadre Monte Carlo Model Checking, nos algorithmes, RURS et RRMA, sont avantageux par rapport à MC^2. Ils permettent de détecter des nœuds cibles, c'est à dire atteindre les états défectueux en nettement moins de temps. Rappelons que RDORS et $RBORS$ correspondent à $RRMA$ avec $p = 1$ et $p = 0$, respectivement.

9.6 Conclusion

Nous avons examiné dans ce chapitre la technique de remplacement des états en mémoire. Des résultats intéressants sont obtenus en combinant cette technique à nos algorithmes d'exploration aléatoire, ce qui a permis d'obtenir des couvertures encore plus larges. Nous avons expérimenté plusieurs stratégies de remplacement dans la mémoire et nous avons proposé une nouvelle : RLFU. Cette stratégie est moins coûteuse dans son application et donne de meilleures performances. Avec une mémoire de seulement 2% de la taille de l'espace d'états et sans aucune technique supplémentaire de réduction, on arrive à réaliser un taux de couverture de plus de 60%. Dès que la taille du cache est de 10% de l'espace d'états, le graphe est, en général, entièrement couvert.

Nous avons ensuite analysé les fondements théoriques de l'approche Monte Carlo MC^2. Nous avons proposé une approche Monte Carlo généralisée qui nous a permis de réduire la borne supérieure $\mathcal{R}$ sur le nombre de répétitions nécessaires pour garantir une certaine confiance sur la probabilité d'erreur dans le système. Par conséquent, des performances significatives sont apportées par notre approche généralisée sur le temps de détection d'érreurs dans un système. Ces performances sont confirmées par nos résultats expérimentaux.

Ex	RW	RURS	$mx = 0$	$mx = 0.2$	$mx = 0.4$	$mx = 0.6$	$mx = 0.8$	$mx = 1$
Token	0.3904	0.0083	0.0063	0.0072	0.0085	0.0073	0.0747	0.0715
Fischer	0.4251	0.0799	0.0680	0.0629	0.0498	0.0872	0.0919	0.0894
Server	0.5895	0.0788	0.0299	0.0270	0.0349	0.0829	0.0695	0.0918
Bitalt	0.4590	0.0095	0.0177	0.0184	0.0193	0.0154	0.0689	0.0971
Filesys	0.6819	0.0405	0.0295	0.0263	0.0296	0.0256	0.1129	0.0846
Prodcons	1.3029	0.2284	0.1077	0.1375	0.1554	0.1916	0.2972	0.2688

TABLE 9.6 – Les intervalles de confiance sur le temps d'exécution

Chapitre 10

Conclusion générale et perspectives

Nous nous sommes penchés, principalement, sur trois aspects importants liés à la vérification randomisée. Nous avons procédé, d'abord, à la modélisation des algorithmes de vérification en général et plus particulièrement des algorithmes randomisés. Ensuite, nous avons conduit une étude théorique et expérimentale détaillée des algorithmes aléatoires que nous avons proposé et qui s'inscrivent dans notre modèle générale. Enfin, nous avons traité les algorithmes randomisés avec les techniques de Caching et Monte Carlo et ce dans le but d'améliorer leur performances.

Dans la première partie de cette thèse, nous avons présenté différents aspects et techniques de la vérification et nous avons conçu un schéma générique qui englobe plusieurs variantes d'algorithmes d'exploration préexistants. Ce schéma a été conçu pour les algorithmes randomisés, en particulier ceux que nous avons proposés, mais il reste valable pour les algorithmes déterministes également. Ces algorithmes, aléatoires et déterministes, ont été, ensuite, classifiés selon ce schéma et moyennant certains critères. La taille limitée de la mémoire utilisée a été introduite explicitement comme paramètre principale dans le modèle afin d'obtenir algorithmes ressources dépendants. On peut ainsi contrôler à tout moment de l'exploration l'état de la mémoire pour qu'elle ne soit pas dépassée et afin d'éviter le stagnation de l'exploration a cause du swapping.

Dans la deuxième partie, nous avons, d'abord, établi des résultats généraux sur le temps moyen de couverture et le nombre moyen de nœud couvert pour un algorithme d'exploration randomisé quelconque sous sa forme simple et répétée. Puis nous avons proposé quatre algorithmes d'exploration aléatoires, ainsi que leurs versions répétées, conformes au schéma général : DORS (*Depth Oriented Random Search*) qui est un algorithme randomisé orienté en profondeur, son dual BORS (*Breadth Oriented Random Search*) orienté en largeur, l'algorithme URS (*Uniform Random Search*) qui est un algorithme d'exploration uniforme, réalisant une exploration équilibrée, et RMA (*Random Mixing Algorithm*) un algorithme d'exploration aléatoire flexible, paramétrée par

un taux de mixage de l'exploration en profondeur/en largeur et guidé par un facteur de densité DF que nous avons défini et qui est caractéristique du graphe. Nous nous sommes attardés, ensuite, sur l'étude, théoriquement et expérimentalement, détaillée pour chacun de ces algorithmes sous sa forme simple ainsi que sous sa forme réitérée. Les résultats théoriques et pratiques obtenus sont cohérents. S'agissant de leur comparaison avec des algorithmes déterministes, nos algorithmes sont bien meilleurs et pouvant atteindre jusqu'au 45% plus de performances en nombre de nœuds couverts. En ce qui concerne, la comparaison de ces algorithmes entre eux, on peut affirmer clairement que l'algorithme RMA est toujours meilleur que les autres, pourvu que le paramètre de mixage soit approprié, ensuite c'est URS qui est meilleur que DORS et BORS dans la plus part des cas et notamment sur les graphes réels de Model Checking. Enfin, comparé à BORS, l'algorithme DORS est meilleur dans la plus part des cas pratiques et dans le cas théoriques correspondant à des graphes maigres (i.e. ayant un DF petit). En revanche, l'algorithme $BORS$ est à préférer si le graphe, réels ou non, est assez large (i.e. son DF est grand).

Dans la troisième partie, dédiée à des améliorations supplémentaires de nos algorithmes, nous nous sommes intéressés, dans un premier temps, au couplage de nos schémas d'exploration avec des techniques de caching dont nous avons proposé, d'ailleurs, une nouvelle qui est moins coûteuse dans son application et donne de meilleures performances que les autres. Des résultats intéressants sont obtenus grâce à ce couplage : avec un cache de seulement 2% de la taille de l'espace d'états et sans aucune technique supplémentaire de réduction, on arrive à réaliser un taux de couverture de plus de 60%. Dès que la taille du cache est de 10% de l'espace d'états, le graphe est, en général, entièrement couvert. Ces résultats sont nettement meilleurs que les résultats préexistants obtenus par couplage des techniques de caching avec d'autres schémas d'exploration déterministes (DFS) ou aléatoires ($RBFS$).
Enfin, nous avons traîté, analytiquement et expérimentalement, nos algorithmes dans le cadre de l'approche Monte Carlo et nous les avons comparé aux algorithmes bien connus MC^2 et QMC. La comparaison des probabilités d'atteignabilité de nos algorithmes avec RW, utilisé dans MC^2 et QMC, nous a mené à proposer une approche Monte Carlo généralisée nous permettant de réduire la borne de répétition R qui constitue l'enjeux principal pour les techniques Monte Carlo, ainsi que le temps d'exécution. Les résultats expérimentaux confirment nettement la supériorité de nos algorithmes par rapport à MC^2 et QMC.

Bibliographie

[1] R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Partial-order reduction in symbolic state-space exploration. In *CAV*, LNCS, pages 340–351, London, UK, June 1997. Springer-Verlag.

[2] M. Ben-Ari, A. Pnueli, and Z. Manna. The temporal logic of branching time. *Acta Informatica*, 20 :207–226, 1983.

[3] M. Ben-Or. Another advantage of free choice : Completely asynchronous agreement protocols. In *ACM SIGACT-SIGOPS symposium on PODC*, pages 27–30, New York, NY, USA, August 1983. ACM.

[4] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using sat procedures instead of bdds. In *DAC*, pages 317–320, New York, NY, USA, June 1999. ACM.

[5] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *TACAS*, pages 193–207, London, UK, March 1999. Springer-Verlag.

[6] M. Bozga, J. C. Fernandez, L. Ghirvu, S. Graf, J. P. Krimm, and L. Mounier. If : A validation environment for timed asynchronous systems. In *CAV*, LNCS, pages 543–547. Springer, July 2000.

[7] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8) :677–691, 1986.

[8] R. E. Bryant. On the complexity of vlsi implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, 40(2) :20–213, February 1991.

[9] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3) :293–318, 1992.

[10] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking : 10^{20} states and beyond. In *LICS*, pages 428–439, Washington, DC, USA, June 1990. IEEE Computer Society.

[11] H. Chockler, E. Farchi, B. Godlin, and S. Novikov. Cross-entropy based testing. In *FMCAD*, pages 101–108, Washington, DC, USA, November 2007. IEEE Computer Society.

[12] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2) :244–263, April 1986.

[13] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, December 1999.

[14] E. M. Clarke, S. Jha, R. Enders, and T. Filkorn. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1/2) :77–104, August 1996.

[15] E.M. Clarke and I.A. Draghicescu. Expressivity results for linear-time and branching-time logics. In *Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency*, volume 354 of *LNCS*, pages 428–437. Springer, June 1988.

[16] P. Dagum, R. Karp, M. Luby, and S. Ross. An optimal algorithm for monte carlo estimation. In *FOCS*, pages 142–149, Washington, DC, USA, 1995. IEEE Computer Society.

[17] R. Drechsler and D. Groβe. Reachability analysis for formal verification of system c. In *Euromicro Symposium on DSD*, page 337. IEEE, September 2002.

[18] S. Edelkamp, A. Lafuente, and S. Leue. Directed explicit model checking with hsf-spin. In *SPIN*, LNCS, pages 57–79, New York, NY, USA, May 2001. Springer-Verlag.

[19] E. A. Emerson, S. Jha, and D. Peled. Combining partial order and symmetry reductions. In *TACAS*, LNCS, pages 19–34, London, UK, April 1997. Springer-Verlag.

[20] U. Feige. A tight lower bound on the cover time for random walks on graphs. *Random Structures and Algorithms*, 6(4) :433–438, 1995.

[21] U. Feige. A tight upper bound on the cover time for random walks on graphs. *Random Structures and Algorithms*, 6(1) :51–54, 1995.

[22] J. C. Fernandez, L. Mounier, C. Jard, and T. Jéron. On-the-fly verification of finite transition systems. *Formal Methods in System Design*, 1(2-3) :251–273, 1992.

[23] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *ACM SIGPLAN-SIGACT Symposium on POPL*, pages 110–121, New York, NY, USA, January 2005. ACM.

[24] R. Fraer, G. Kamhi, B. Ziv, M. Y. Vardi, and L. Fix. Prioritized traversal : Efficient reachability analysis for verification and falsification. In *CAV*, LNCS, pages 389–402, London, UK, July 2000. Springer-Verlag.

[25] J. Geldenhuys. State caching reconsidered. In *SPIN*, LNCS, pages 23–38. Springer, April 2004.

[26] P. Godefroid. Using partial orders to improve automatic verification methods. In *CAV*, LNCS, pages 176–185. Springer, June 1990.

[27] P. Godefroid. On the costs and benefits of using partial-order methods for the verification of concurrent systems. In *Partial Order Methods in Verification*, volume 29 of *DIMACS*, pages 289–303, New York, USA, 1997. American Mathematical Society, AMS Press, Inc.

[28] P. Godefroid, G. J. Holzmann, and D. Pirottin. State-space caching revisited. In *CAV*, LNCS, pages 178–191. Springer, June 1992.

[29] P. Godefroid and S. Khurshid. Exploring very large state spaces using genetic algorithms. *International Journal on STTT*, 6(2) :117–127, August 2004.

[30] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL : a theorem proving environment for higher order logic*. Cambridge University Press, New York, NY, USA, 1993.

[31] J. Goubault. *Démonstration automatique en logique classique : complexité et méthodes*. PhD thesis, Ecole polytechnique, Palaiseau, FRANCE, 1993.

[32] R. Grosu, X. Huang, S. A. Smolka, W. Tan, and S. Tripakis. Deep random search for efficient model checking of timed automata. In *Monterey Workshop*, LNCS, pages 111–124, Berlin, Heidelberg, October 2006. Springer.

[33] R. Grosu and S. A. Smolka. Quantitative model checking. In *ISoLA*, Technical Report, pages 165–174, Paphos, Cyprus, October, November 2004. Department of Computer Science, University of Cyprus.

[34] R. Grosu and S. A. Smolka. Monte carlo model checking. In *TACAS*, LNCS, pages 271–286, Berlin, Heidelberg, Avril 2005. Springer.

[35] P. Haslum. Model checking by random walk. In *ECSEL Workshop*, 1999.

[36] C. A. R. Hoare. An axiomatic basis for computer programming. *Communication of the ACM*, 12(10) :576–580, October 1969.

[37] G. J. Holzmann. Tracing protocols. *AT&T Technical Journal*, 64(10) :2413–2433, 1985.

[38] G. J. Holzmann. Automated protocol validation in *argos :* assertion proving and scatter searching. *IEEE Transactions on Software Engineering*, 13(6) :683–696, 1987.

[39] G. J. Holzmann. An analysis of bistate hashing. In *PSTV*, volume 38 of *IFIP Conference Proceedings*, pages 301–314. Chapman & Hall, June 1995.

[40] W.A. Howard. *To H.B. Curry : The formulae-as-types notion of construction. Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press, 1969.

[41] G. Huet, G. Kahn, and C. Paulin-Mohring. The coq proof assistant, a tutorial. Technical report, INRIA, February 2007.

[42] M. Huth and M.D. Ryan. *Logic in computer science - Modelling and reasoning about systems*. Cambridge University Press, 1999.

[43] C. Jard and T. Jéron. On-line model checking for finite linear temporal logic specifications. In *Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 189–196. Springer, June 1989.

[44] C. Jard and T. Jéron. Bounded memory algorithm for verification on-the-fly. In *CAV*, volume 575 of *LNCS*, pages 192–202. Springer, July 1991.

[45] H. Kautz and B. Selman. Pushing the envelope : Planning, propositional logic and stochastic search. In *AAAI / IAAI*, pages 1194–1201, Menlo Park, CA, August 1996. AAAI Press / MIT Press.

[46] A. Kuehlmann, K. L. McMillan, and R. K. Brayton. Probabilistic state space search. In *ICCAD*, pages 574–579. IEEE, November 1999.

[47] O. Kupferman, M. Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2) :312–360, March 2000.

[48] D. Lehmann and M. O. Rabin. On the advantages of free choice : a symmetric and fully distributed solution to the dining philosophers problem. In *ACM SIGPLAN-SIGACT symposium on POPL*, pages 133–138, New York, NY, USA, January 1981. ACM.

[49] F. J. Lin, P. M. Chu, and M. T. Liu. Protocol verification using reachability analysis : the state space explosion problem and relief strategies. *SIGCOMM Computer Communication Review*, 17(5) :126–135, October 1987.

[50] A. Lluch-Lafuente, S. Edelkamp, and S. Leue. Partial order reduction in directed model checking. In *SPIN*, LNCS, pages 112–127, London, UK, April 2002. Springer-Verlag.

[51] M. Mihail and C. H. Papadimitriou. On the random walk method for protocol testing. In *CAV*, LNCS, pages 132–141, Berlin, Heidelberg, 1994. Springer.

[52] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, New York, NY, USA, August 1995.

[53] R. Nalumasu and G. Gopalakrishnan. An efficient partial order reduction algorithm with an alternative proviso implementation. *Formal Methods in System Design*, 20(3) :231–247, May 2002.

[54] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. Pvs : Combining specification, proof checking, and model checking. In *CAV*, LNCS, pages 411–414. Springer, August 1996.

[55] R. Pelánek. Properties of state spaces and their applications. *International Journal on STTT*, 10(5) :443–454, September 2008.

[56] R. Pelánek, T. Hanžl, I. Černá, and L. Brim. Enhancing random walk state space exploration. In *FMICS*, pages 98–105, NY, USA, September 2005. ACM.

[57] A. Pnueli. Linear and branching structures in the semantics and logics of reactive systems. In *Colloquium on Automata, Languages and Programming*, pages 15–32, London, UK, July 1985. Springer-Verlag.

[58] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *Symposium on Programming*, pages 337–351, London, UK, April 1982. Springer-Verlag.

[59] M. O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1) :128–138, 1980.

[60] S. Sankaranarayanan, R. Chang, G. Jiang, and F. Ivancic. State space exploration using feedback constraint generation and monte-carlo sampling. In *ESEC-FSE*, pages 321–330, New York, NY, USA, September 2007. ACM.

[61] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3) :733–749, 1985.

[62] H. Sivaraj and G. Gopalakrishnan. Random walk based heuristic algorithms for distributed memory model checking. In *PDMC*, ENTCS, pages 51–67. Elsevier, July 2003.

[63] U. Stern and D. L. Dill. Improved probabilistic verification by hash compaction. In *CHARME*, LNCS, pages 206–224, London, UK, October 1995. Springer-Verlag.

[64] E. Tronci, G. D. Penna, B. Intrigila, and M. V. Zilli. A probabilistic approach to automatic verification of concurrent systems. In *APSEC*, pages 317–324, Washington, DC, USA, December 2001. IEEE Computer Society.

[65] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Proceedings of the VIII Banff Higher order workshop conference on Logics for concurrency : structure versus automata*, LNCS, pages 238–266, Secaucus, NJ, USA, August 1996. Springer-Verlag.

[66] C. H. West. Protocol validation by random state exploration. In *PSTV*, pages 233–242, Amsterdam, Netherlands, 1986. North-Holland.

[67] P. Wolper and D. Leroy. Reliable hashing without collision detection. In *CAV*, pages 59–70, London, UK, June 1993. Springer-Verlag.

[68] B. Yang, R. E. Bryant, D. R. O'Hallaron, A. Biere, O. Coudert, G. Janssen, R. K. Ranjan, and F. Somenzi. A performance study of bdd-based model checking. In *FMCAD*, pages 255–289, London, UK, November 1998. Springer-Verlag.

[69] H. L. S. Younes and R. G. Simmons. Probabilistic verification of discrete event systems using acceptance sampling. In *CAV*, LNCS, pages 223–235, London, UK, July 2002. Springer-Verlag.

Annexe A

A.1 Preuve du lemme 6.1

Soit $\mathbb{P}(\underline{w}_k, n, C)$ (resp. $\mathbb{P}(\underline{w}_k, n, O, v)$ la probabilité de couvrir, en n étapes, l'ensemble de nœuds $\underline{w}_k$ et d'être, à la fin de l'étape n, dans un point clos (resp. dans un point ouvert au nœud v). On note par $D(\underline{w}_k)$ l'ensemble des nœuds de blockage (ou feuilles) dans $\underline{w}_k$ et on pose $1_{w_k}(v) = 1$ si $v = w_k$ et $1_{w_k}(v) = 0$ sinon.

Donc pour $\mathbb{P}(\underline{w}_k, n, C)$, l'exploration doit être dans un point clos, et aucun nœud n'est nouvellement atteint à l'étape n : à l'étape $n-1$ l'algorithme a atteint un nœud de blockage et à l'étape n il cherche, sans succès un successeur de ce nœud, donc il sera dans un point clos à l'étape n. Il y a deux cas : à l'étape $n-1$, l'exploration est dans un point clos ou dans un point ouvert en un nœud de blockage v. Dans le premier cas, elle doit redémarrer à l'étape n, avec probabilité $\frac{1}{k}$, à partir d'un point de blockage, ce qui donne le terme $\frac{|D(\underline{w}_k)|}{k}$. Dans le deuxième cas, l'exploration est ouverte dans le nœud v, donc elle doit continuer dans l'ensemble des successeurs de v. Cet ensemble est vide et l'exploration atteint un point clos avec probabilité 1. Ceci donne la récurrence :

$$\mathbb{P}(\underline{w}_k, n, C) = \frac{|D(\underline{w}_k)|}{k}\mathbb{P}(\underline{w}_k, n-1, C) + \sum_{v \in D(\underline{w}_k)} \mathbb{P}(\underline{w}_k, n-1, O, v)$$

Pour $\mathbb{P}(\underline{w}_k, n, O, v)$, il y a quatre cas pour l'algorithme $DORS$:

- Cas 1 : Aucun nouveau nœud n'est couvert à l'étape n et, à l'étape $n-1$, l'exploration était dans un point ouvert, en un nœud u. u doit être dans $F(v) \cap \underline{w}_k$ (i.e. un père de v dans $\underline{w}_k$), et à l'étape n, v est choisi uniformément parmi $C(u)$ (avec probabilité $\frac{1}{|C(u)|}$). Ceci donne le premier terme dans la récurrence qui suit.
- Cas 2 : Aucun nouveau nœud n'est couvert à l'étape n et, à l'étape $n-1$, l'exploration était dans un point fermé. Donc à l'étape n, on choisit, avec probabilité $\frac{1}{k}$, un nœud u dans $F(v) \cap \underline{w}_k$ et on tire v uniformément parmi $C(u)$. Ceci donne le second terme dans la récurrence.
- Cas 3 : un nouveau nœud v est couvert à l'étape n et, à l'étape $n-1$, l'exploration était dans un point ouvert en un nœud u. Le nouveau nœud couvert doit être w_k puisque la suite $\underline{w}_k$ est stockée dans l'ordre des visites, u doit être dans $F(v) \cap \underline{w}_{k-1}$

et à l'étape n, v est choisi uniformément parmi $C(u)$. Ceci donne le troisième terme dans la récurrence. Notons que le terme $1_{w_k}(v)$ exprime le fait qu'on doit avoir $v = w_k$ dans le troisième et le quatrième termes de la récurrence.

– Cas 4 : un nouveau nœud v est couvert à l'étape n et, à l'étape $n-1$, l'exploration était dans un point clos. donc, à l'étape n, on choisit, avec une probabilité $\frac{1}{k-1}$, un nœud u dans $F(v) \cap \underline{w}_{k-1}$ et on tire v uniformément parmi $C(u)$. Ceci donne le quatrième terme dans la récurrence.

On obtient la récurrence suivante :

$$\mathbb{P}(\underline{w}_k, n, O, v) = \sum_{u \in F(v) \cap \underline{w}_k} \Big[\frac{\mathbb{P}(\underline{w}_k, n-1, O, u)}{|C(u)|} + \frac{\mathbb{P}(\underline{w}_k, n-1, C)}{k|C(u)|} + 1_{w_k}(v) \Big(\frac{\mathbb{P}(\underline{w}_{k-1}, n-1, O, u)}{|C(u)|} + \frac{\mathbb{P}(\underline{w}_{k-1}, n-1, C)}{(k-1)|C(u)|} \Big) \Big]$$

A.2 Preuve du lemme 6.1

Soit $\mathbb{P}(\underline{w}_k, n)$ la probabilité de couvrir, en n étapes, par l'algorithme RMA, l'ensemble de nœuds $\underline{w}_k$. On a alors :

$$\mathbb{P}(\underline{w}_k, n) = \alpha_{\mathrm{RMA}}(\underline{w}_k)\mathbb{P}(\underline{w}_k, n-1) + \beta_{\mathrm{RMA}}\mathbb{P}(\underline{w}_{k-1}, n-1)$$

avec

$$\begin{aligned} \alpha_{\mathrm{RMA}}(\underline{w}_k) &= mx \,.\, \frac{\sum_{v \in L(\underline{w}_k)} \frac{|C(v) \cap \underline{w}_k|}{|C(v)|}}{|L(\underline{w}_k)|} + (1 - mx) \,.\, \frac{\sum_{v \in I(\underline{w}_k)} \frac{|C(v) \cap \underline{w}_k|}{|C(v)|}}{|I(\underline{w}_k)|} \\ \beta_{\mathrm{RMA}}(\underline{w}_k) &= mx \,.\, \frac{\sum_{v \in F(w_k) \cap L(\underline{w}_{k-1})} \frac{1}{|C(v)|}}{|L(\underline{w}_{k-1})|} + (1 - mx) \,.\, \frac{\sum_{v \in F(w_k) \cap I(\underline{w}_{k-1})} \frac{1}{|C(v)|}}{|I(\underline{w}_{k-1})|} \end{aligned}$$

En effet, le terme en α_{RMA} correpond au cas d'une redondance (aucun nouveau nœud n'est couvert), alors que le terme en β_{RMA} correspond au cas où un nouveau nœud est couvert. Ce nœud doit être forcément w_k, puisque la séquence $\underline{w}_k$ est rangée dans l'ordre de visite.

Expliquons d'abord le premier terme dans l'expression de α_{RMA}. Ce terme correspond au cas où aucun nouveau nœud n'est couvert à l'étape n et un nœud feuille v est choisit avec une probabilité $mx \,.\, \frac{1}{|L(\underline{w}_k)|}$, mx étant la probabilité pour que le nœud courant v soit une feuille et $\frac{1}{|L(\underline{w}_k)|}$ étant la probabilité de choisir v dans l'ensemble des feuilles $L(\underline{w}_k)$. La probabilité de choisir un fils de v dans $\underline{w}_k$ (puisqu'on est dans le cas de la redondance) est, alors, donnée par $\frac{|C(v) \cap \underline{w}_k|}{|C(v)|}$. Similairement, le second terme de α_{RMA} correspond au choix d'un nouveau nœud v parmis les nœuds internes (i.e. dans l'ensemble $I(\underline{w}_k)$), ce qui explique, en particulier, le coefficient $1 - mx$ au lieu de mx. Les deux termes dans l'expression de β_{RMA} sont analogues à ceux de α_{RMA}, sauf qu'ici un nouveau nœud est couvert (qui doit être w_k comme on l'a expliqué ci-dessus) et

donc le nœud courant v doit être choisi dans l'ensemble $F(w_k)$ des pères de w_k. Selon le cas du choix, d'une feuille (premier terme de β_{RMA}) ou d'un nœud interne (second terme de β_{RMA}), le nœud courant v doit être également choisi, respectivement, dans $L(\underline{w}_{k-1})$ ou dans $I(\underline{w}_{k-1})$. Une fois ce choix est fait, le nœud w_k est simplement choisi parmis les fils de v, donc avec une probabilité $\frac{1}{C(v)}$.

A.3 Preuve du théorème 7.1

On commence par montrer la formule du temps moyen de couverture relative à l'algorithme URS :

$$T_{urs}(\underline{k}) \;=\; (1-\alpha(\underline{k}))S^1_{urs}(\underline{k}) - \alpha(\underline{k})S^0_{urs}(\underline{k})$$

Premièrement, on a déjà obtenu la récurrence suivante qui exprime la probabilité de couvrir le vecteur $\underline{k} = (k_1, ..., k_h)$ en n étapes par l'algorithme URS :

$$\begin{aligned}\mathbb{P}_{urs}(\underline{K}_n = \underline{k}) &= \alpha(\underline{k})\,\mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k}) \\ &+ \sum_{j=1}^{h} \beta_j(\underline{k})\,\mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k} - 1_j) \qquad \text{(A.1)}\end{aligned}$$

Le temps moyen de couverture de URS est donc donnée par :

$$\begin{aligned}T_{urs}(\underline{k}) &= \sum_{n=k}^{\infty} n\mathbb{P}^{\mathcal{I}}_{urs}(\underline{K}_n = \underline{k}) \\ &= \sum_{n=k}^{\infty} n \sum_{i=1}^{h} \beta_i(\underline{k})\,\mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k} - 1_i) \\ &= \sum_{i=1}^{h} \beta_i(\underline{k}) \sum_{n=k}^{\infty} n\,\mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k} - 1_i) \\ &= \sum_{i=1}^{h} \beta_i(\underline{k}) \times S(\underline{k} - 1_i),\end{aligned}$$

avec

$$\begin{aligned}S(\underline{k} - 1_i) &= \sum_{n=k}^{\infty}((n-1)+1)\,\mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k} - 1_i) \\ &= \sum_{n=k}^{\infty}\mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k} - 1_i) + \sum_{n=k}^{\infty}(n-1)\,\mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k} - 1_i) \\ &= \sum_{n=k-1}^{\infty}\mathbb{P}_{urs}(\underline{K}_n = \underline{k} - 1_i) + \sum_{n=k-1}^{\infty} n\,\mathbb{P}_{urs}(\underline{K}_n = \underline{k} - 1_i) \\ &= S^0_{urs}(\underline{k} - 1_i) + S^1_{urs}(\underline{k} - 1_i),\end{aligned}$$

où

$$\begin{aligned} S^0_{urs}(\underline{k}) &= \sum_{n=k}^{\infty} \mathbb{P}_{urs}(\underline{K}_n = \underline{k}) \\ S^1_{urs}(\underline{k}) &= \sum_{n=k}^{\infty} n\, \mathbb{P}_{urs}(\underline{K}_n = \underline{k}) \end{aligned}$$

Donc :

$$T_{urs}(\underline{k}) = \sum_{i=1}^{h} \beta_i(\underline{k}) \Big(S^0_{urs}(\underline{k} - 1_i) + S^1_{urs}(\underline{k} - 1_i) \Big) \tag{A.2}$$

En utilisant la récurrence A.1, on obtient :

$$\begin{aligned} S^0_{urs}(\underline{k}) &= \sum_{n=k}^{\infty} \mathbb{P}_{urs}(\underline{K}_n = \underline{k}) \\ &= \sum_{n=k}^{\infty} [\alpha(\underline{k})\ \mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k}) + \sum_{i=1}^{h} \beta_i(\underline{k})\ \mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k} - 1_i)] \\ &= \alpha(\underline{k}) \sum_{n=k}^{\infty} \mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k}) + \sum_{i=1}^{h} \beta_i(\underline{k}) \sum_{n=k}^{\infty} \mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k} - 1_i) \end{aligned}$$

Notons que pour $n = k$, $\mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k}) = 0$, car en $k-1$ étapes, l'algorithme ne peut couvrir plus que $k-1$ nœuds, donc il ne peut pas couvrir le vecteur $\underline{k}$ qui contient k nœuds. On aura :

$$\begin{aligned} S^0_{urs}(\underline{k}) &= \alpha(\underline{k}) \sum_{n=k+1}^{\infty} \mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k}) \\ &+ \sum_{i=1}^{h} \beta_i(\underline{k}) \sum_{n=k}^{\infty} \mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k} - 1_i) \end{aligned}$$

Donc, par le changement de variables $n := n - 1$, on a :

$$\begin{aligned} S^0_{urs}(\underline{k}) &= \alpha(\underline{k}) \sum_{n=k}^{\infty} \mathbb{P}_{urs}(\underline{K}_n = \underline{k}) + \sum_{i=1}^{h} \beta_i(\underline{k}) \sum_{n=k-1}^{\infty} \mathbb{P}_{urs}(\underline{K}_n = \underline{k} - 1_i) \\ &= \alpha(\underline{k}) S^0_{urs}(\underline{k}) + \sum_{i=1}^{h} \beta_i(\underline{k}) S^0_{urs}(\underline{k} - 1_i) \end{aligned}$$

Donc,

$$S^0_{urs}(\underline{k}) = \frac{1}{1 - \alpha(\underline{k})} \sum_{i=1}^{h} \beta_i(\underline{k})\ S^0_{urs}(\underline{k} - 1_i) \tag{A.3}$$

Similairement,

$$\begin{aligned}
S^1_{urs}(\underline{k}) &= \sum_{n=k}^{\infty} n\mathbb{P}_{urs}(\underline{K}_n = \underline{k}) \\
&= \sum_{n=k}^{\infty} \mathbb{P}_{urs}(\underline{K}_n = \underline{k}) + \sum_{n=k}^{\infty}(n-1)\mathbb{P}_{urs}(\underline{K}_n = \underline{k}) \\
&= S^0_{urs}(\underline{k}) + \sum_{n=k}^{\infty}(n-1)[\alpha(\underline{k})\ \mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k}) \\
&+ \sum_{i=1}^{h} \beta_i(\underline{k})\ \mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k} - 1_i)] \\
&= S^0_{urs}(\underline{k}) + \alpha(\underline{k}) \sum_{n=k}^{\infty}(n-1)\mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k}) \\
&+ \sum_{i=1}^{h} \beta_i(\underline{k}) \sum_{n=k}^{\infty}(n-1)\mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k} - 1_i)
\end{aligned}$$

Comme précédemment, en posant le changement de variable $n := n - 1$, on obtient :

$$\begin{aligned}
S^1_{urs}(\underline{k}) &= S^0_{urs}(\underline{k}) + \alpha(\underline{k}) \sum_{n=k+1}^{\infty}(n-1)\mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k}) \\
&+ \sum_{i=1}^{h} \beta_i(\underline{k}) \sum_{n=k}^{\infty}(n-1)\ \mathbb{P}_{urs}(\underline{K}_{n-1} = \underline{k} - 1_i) \\
&= S^0_{urs}(\underline{k}) + \alpha(\underline{k}) \sum_{n=k}^{\infty} n\mathbb{P}_{urs}(\underline{K}_n = \underline{k}) \\
&+ \sum_{i=1}^{h} \beta_i(\underline{k}) \sum_{n=k-1}^{\infty} n\ \mathbb{P}_{urs}(\underline{K}_n = \underline{k} - 1_i) \\
&= S^0_{urs}(\underline{k}) + \alpha(\underline{k})S_1(\underline{k}) + \sum_{i=1}^{h} \beta_i(\underline{k})S_1(\underline{k} - 1_i)
\end{aligned}$$

et donc,

$$S^1_{urs}(\underline{k}) = \frac{1}{1-\alpha(\underline{k})}\Big(S^0_{urs}(\underline{k}) + \sum_{i=1}^{h} \beta_i(\underline{k})\ S^1_{urs}(\underline{k} - 1_i)\Big) \tag{A.4}$$

Par conséquent, de l'équation A.2, on a :

$$T_{urs}(\underline{k}) = \sum_{i=1}^{h} \beta_i(\underline{k})S^0_{urs}(\underline{k} - 1_i) + \sum_{i=1}^{h} \beta_i(\underline{k})S^1_{urs}(\underline{k} - 1_i)$$

et par l'application des équations A.3 et A.4, on obtient :

$$\begin{aligned} T_{urs}(\underline{k}) &= (1-\alpha(\underline{k}))S^0_{urs}(\underline{k}) + [(1-\alpha(\underline{k}))S^1_{urs}(\underline{k}) - S^0_{urs}(\underline{k})] \\ &= (1-\alpha(\underline{k}))S^1_{urs}(\underline{k}) - \alpha(\underline{k})S^0_{urs}(\underline{k}) \end{aligned}$$

A.4 Preuve du théorème 6.1

Maintenant, pour la formule du temps moyen de couverture, relative à DORS, on établit d'abord les récurrences similaires des probabilités $P_{DORS}(\underline{K}_n = \underline{k}, C)$ (resp. $P_{DORS}(\underline{K}_n = \underline{k}, O)$) de couvrir $\underline{k} = (k_1, ..., k_h)$ (i.e. k_i nœuds sont couverts à chaque niveau $i = 1, ..., h$ de l'arbre) en n étapes et étant dans un point fermé (resp. ouvert) de l'exploration :

$$\mathbb{P}_{DORS}(\underline{K}_n = \underline{k}) = \mathbb{P}_{DORS}(\underline{K}_n = \underline{k}, C) + \mathbb{P}_{DORS}(\underline{K}_n = \underline{k}, O)$$

avec

$$\begin{aligned} \mathbb{P}_{DORS}(\underline{K}_n = \underline{k}, C) = \sum_{s=1}^{h+1} \frac{1}{m^{s-1}} \Big[& \alpha(\underline{k}, C)\ \mathbb{P}_{DORS}(\underline{K}_{n-s} = \underline{k}, C) \\ & + \sum_{j=h-s+2}^{h} \beta_j(\underline{k}, C)\ \mathbb{P}_{DORS}(\underline{K}_{n-s} = \underline{k} - 1_{j,h}, C) \Big] \end{aligned} \tag{A.5}$$

où

$$\alpha(\underline{k}, C) = \frac{k_h}{k},\ \ \beta_j(\underline{k}, C) = \frac{mk_{j-1} - (k_j - 1)}{(k - h + j - 1)m^{j-h}},\ \ 1_{j,j'} = 1_j + 1_{j+1} .... + 1_{j'}.$$

et

$$\begin{aligned} \mathbb{P}_{DORS}(\underline{K}_n = \underline{k}, O) = \sum_{s=1}^{h} \frac{1}{m^s} \Big[& \alpha(\underline{k}, O)\ \mathbb{P}_{DORS}(\underline{K}_{n-s} = \underline{k}, C) \\ & + \sum_{j=1}^{h} \sum_{l=\max(j,0)}^{\min(s+j-1,h)} \beta_{j,l}(\underline{k}, O)\ \mathbb{P}_{DORS}(\underline{K}_{n-s} = \underline{k} - 1_{j,l}, C) \Big] \end{aligned} \tag{A.6}$$

où

$$\alpha(\underline{k}, O) = \frac{k_s + ... + k_h}{k},\ \ \beta_{j,l}(\underline{k}, O) = \frac{mk_{j-1} - (k_j - 1)}{(k - l + j - 1)m^{j-l}}$$

Le temps de couverture moyen de DORS est donnée par :

$$T_{DORS}(\underline{k}) = \sum_{n=k}^{\infty} n\mathbb{P}^{\mathcal{I}}_{DORS}(\underline{K}_n = \underline{k})$$

Notons que la probabilité d'innovation pour $DORS$ est composée de deux termes :

$$\mathbb{P}^{\mathcal{I}}_{DORS}(\underline{K}_n = \underline{k}) = \mathbb{P}^{\mathcal{I}}_{DORS}(\underline{K}_n = \underline{k}, C) + \mathbb{P}^{\mathcal{I}}_{DORS}(\underline{K}_n = \underline{k}, O)$$

où

$$\mathbb{P}^{\mathcal{I}}_{DORS}(\underline{K}_n = \underline{k}, C) = \sum_{s=1}^{h+1} \frac{1}{m^{s-1}} \Big[\sum_{j=h-s+2}^{h} \beta_j(\underline{k}, C) \times \mathbb{P}_{DORS}(\underline{K}_{n-s} = \underline{k} - 1_{j,h}, C) \Big]$$

$$\mathbb{P}^{\mathcal{I}}_{DORS}(\underline{K}_n = \underline{k}, O) = \sum_{s=1}^{h} \frac{1}{m^{s}} \Big[\sum_{j=1}^{h} \sum_{l=\max(j,0)}^{\min(s+j-1,h)} \beta_{j,l}(\underline{k}, O) \times \mathbb{P}_{DORS}(\underline{K}_{n-s} = \underline{k} - 1_{j,l}, C) \Big]$$

Donc, comme précédemment, le temps moyen de couverture T_{DORS} peut être exprimé en fonction de S^0_{DORS} et S^1_{DORS}, où :

$$\begin{aligned} S^0_{DORS}(\underline{k}) &= \sum_{n=k}^{\infty} \mathbb{P}_{DORS}(\underline{K}_n = \underline{k}, C) \\ S^1_{DORS}(\underline{k}) &= \sum_{n=k}^{\infty} n\, \mathbb{P}_{DORS}(\underline{K}_n = \underline{k}, C) \end{aligned}$$

des équations A.5 et A.6, on obtient la récurrence :

$$\begin{aligned} S^0_{DORS}(\underline{k}) &= \sum_{j=1}^{h} \gamma^0_j(\underline{k})\, S^0_{DORS}(\underline{k} - 1_{j,h}) \\ S^1_{DORS}(\underline{k}) &= \sum_{j=1}^{h} \Big(\gamma^1_j(\underline{k}) S^1_{DORS}(\underline{k} - 1_{j,h}) + \delta_j(\underline{k})\, S^0_{DORS}(\underline{k} - 1_{j,h}) \Big) \\ &+ \mu(\underline{k}) S^0_{DORS}(\underline{k}) \end{aligned}$$

où,

$$\begin{aligned} \gamma^0_j(\underline{k}) &= \gamma^1_j(\underline{k}) = \frac{d_0(h-j+1,h)\beta_j(\underline{k}, C)}{1 - d_0(0,h)\alpha(\underline{k}, C)} \\ \delta_j(\underline{k}) &= \frac{d_1(h-j+1,h)}{d_0(h-j+1,h)} \gamma^0_j(\underline{k}) \\ \mu(\underline{k}) &= \frac{d_0(0,h)\alpha(\underline{k}, C)}{1 - d_0(0,h)\alpha(\underline{k}, C)} \\ d_0(j,j') &= \frac{\frac{1}{m^{j-1}} - \frac{1}{m^{j'}}}{m-1} \\ d_1(j,j') &= \frac{m}{m-1} d_0(j, j'+1) + \frac{\frac{j}{m^{j-1}} + \frac{j'+2}{m^{j'}}}{m-1} \end{aligned}$$

Finalement, des calcules similaires à ceux effectués pour l'algorithme URS, donnent l'équation requise pour $DORS$:

$$\begin{aligned} T_{drs}(\underline{k}) = & \sum_{j=1}^{h}\sum_{l=j}^{h}\Big[c_{j,l}(\underline{k})\ S^1_{drs}(\underline{k}-1_{j,l}) + d_{j,l}(\underline{k})\ S^0_{drs}(\underline{k}-1_{j,l})\Big] \\ & + a(\underline{k})\ S^1_{drs}(\underline{k}) - b(\underline{k})\ S^0_{drs}(\underline{k}) \end{aligned}$$

où les coefficients dans l'équation sont donnés par :

$$\begin{aligned} c_{j,l}(\underline{k}) &= \beta_{j,l}(\underline{k},O)d_0(l-j,l) \\ d_{j,l}(\underline{k}) &= \beta_{j,l}(\underline{k},O)(d_1(l-j,l)-d_0(l-j,l)) \\ a(\underline{k}) &= 1-d_0(0,h)\alpha(\underline{k},C) \\ b(\underline{k}) &= d_1(0,h)\alpha(\underline{k},C) \end{aligned}$$