Vérification formelle d'un algorithme d'allocation de registres par coloration de graphe

Sandrine Blazy, Benoît Robillard, Eric Soutif

Vérification formelle d'un algorithme d'allocation de registres par coloration de graphe

Sandrine Blazy & Benoît Robillard & Éric Soutif

Laboratoire CEDRIC, 292 rue Saint-Martin, 75141 Paris CEDEX 03
{blazy,robillard}@ensiie.fr,soutif@cnam.fr

Résumé

Le travail présenté dans cet article est à l'interface entre la recherche opérationnelle et les méthodes formelles. Il s'inscrit dans le cadre du projet CompCert ayant pour but le développement et la vérification formelle, utilisant l'assistant de preuve Coq, d'un compilateur du langage C potentiellement utilisable pour la production de logiciels embarqués critiques. Nous nous intéressons dans cet article à l'allocation de registres, qui consiste à optimiser l'utilisation des registres du processeur. Nous proposons d'aborder cette optimisation en la modélisant par un problème dit de coloration avec préférences dont nous vérifions formellement la résolution. Cette vérification prend deux formes : preuve de correction de la spécification Coq pour la première partie de l'algorithme et validation *a posteriori* pour la seconde.

Introduction

Les méthodes de développement formelles permettent de produire du code source certifié. Cependant, une faille du processus de certification demeure au niveau de la compilation de ce code. En effet, un bug dans le compilateur peut introduire des erreurs dans le code assembleur généré, et donc invalider le code source. De là sont nés le besoin de compilateurs certifiés et le projet CompCert. Ce projet a pour but de développer avec l'assistant à la preuve Coq un compilateur réaliste d'un vaste sous-ensemble du langage C vers le langage assembleur du processeur PowerPC, principalement dévolu au domaine des logiciels embarqués [Ler06, BDL06].

Le compilateur CompCert est un compilateur modérément optimisant, qui accomplit de nombreuses passes de transformations de programmes. Celui-ci est certifié, c'est-à-dire qu'il est accompagné d'une preuve Coq de préservation du comportement des programmes (tout au long du processus de compilation). Cette preuve consiste à établir la préservation sémantique de chaque passe du compilateur. Il s'agit d'écrire en Coq une passe de compilation et de prouver ensuite que celle-ci transforme un programme en un programme observationellement équivalent. L'intérêt d'une telle approche est que le compilateur est certifié une fois pour toutes, indépendamment des programmes à compiler.

Le développement du compilateur CompCert est une expérience de conception assistée par preuve. Il ne s'agit pas de prouver un compilateur existant, mais plutôt de définir conjointement les langages intermédiaires, les transformations de programmes et les preuves associées. C'est souvent à l'issue d'une preuve jugée trop difficile qu'il est décidé de modifier un langage intermédiaire, voire de définir un nouveau langage intermédiaire, ce qui nécessite de plus de prouver à nouveau certaines propriétés

ayant préalablement été prouvées. Ainsi, le compilateur CompCert actuel dispose de sept langages intermédiaires.

L'allocation de registres est la seule phase du compilateur CompCert qui n'est pas entièrement écrite en Coq. La raison principale est qu'il s'agit d'une transformation de programmes peu adaptée à une écriture fonctionnelle, et que l'effort nécessaire pour spécifier en Coq et prouver ensuite la préservation sémantique est trop important. En effet, classiquement, l'allocation de registres se ramène à un problème de coloration avec préférences de graphe. Ce problème étant $\mathcal{NP}$-difficile, CompCert implante une des heuristiques les plus performantes pour le résoudre.

Dans CompCert, l'allocation de registres est écrite en Caml puis validée *a posteriori* en Coq : pour chaque programme à compiler, il est vérifié formellement que la solution calculée par l'allocation de registres est correcte. L'intérêt de cette approche est que la preuve à effectuer est beaucoup plus facile, puisqu'il s'agit de vérifier que l'allocation de registres a bien renvoyé une coloration du graphe. En outre, cette technique permet de valider une transformation de programme qui n'a pas été écrite en Coq.

Cet article décrit une nouvelle méthode d'optimisation de l'allocation de registres pour le compilateur CompCert. Nous détaillons d'abord une formalisation en Coq d'un algorithme de coloration (sans préférences) adapté à une famille de graphes regroupant la majeure partie des graphes utilisés pour modéliser l'allocation de registres. Il s'agit d'un algorithme exact, dont nous prouvons également l'optimalité en Coq dans la famille de graphes précitée. Nous présentons ensuite la seconde partie de la méthode qui utilise la programmation linéaire en nombres entiers. Cette étape est réalisée par un solveur qui fournit une allocation de registres optimale. Comme il s'agit d'un solveur externe, cette solution est validée *a posteriori* en Coq. Le programme mathématique à résoudre dépend du résultat de la première partie de la méthode, d'où l'importance de l'optimalité du premier algorithme.

L'objectif de cet article est double. D'une part, nous proposons une amélioration de l'allocation de registres actuellement utilisée dans le compilateur certifié CompCert. D'autre part, nous présentons le début d'un travail de formalisation en Coq de structures de données et algorithmes de théorie des graphes et de programmation mathématique. Cet article est organisé comme suit. La première partie introduit l'allocation de registres et présente un état de l'art. Puis, la deuxième partie décrit les notions de théorie des graphes et de programmation mathématique utiles pour l'allocation de registres. Ensuite, la troisième partie explique notre algorithme d'allocation de registres. Enfin, la quatrième partie détaille la spécification Coq ainsi que les principales propriétés que nous avons prouvées. Le code source complet de ce développement est disponible sur la page `http://www.ensiie.fr/~blazy/register-allocation`.

1. Allocation de registres

1.1. Généralités

Au cours de l'exécution d'un programme, le processeur effectue un grand nombre d'accès à la mémoire afin de lire et écrire les valeurs des variables du programme. Ces accès sont naturellement gourmands en temps. Pour accélérer l'exécution des programmes, le processeur est muni d'un petit nombre de zones de stockage à accès beaucoup plus rapide, les registres. En général, le nombre de registres d'un processeur est nettement inférieur au nombre de variables utilisées dans un programme. Le but de l'allocation de registres est de déterminer où sont stockées les variables d'un programme à tout moment de son exécution : soit en registres si ces derniers sont disponibles, soit en mémoire le cas échéant. La difficulté est de proposer une affectation optimale des registres. Il est par exemple souvent nécessaire de choisir entre conserver une variable v dans un même registre R pendant l'exécution complète d'un programme (ce qui rend R inutilisable pour stocker d'autres variables), et réutiliser R lorsque la valeur de v n'a plus besoin d'être conservée en vue d'utilisations futures (ce qui nécessite

de transférer en mémoire la valeur de v).

L'allocation de registres est la passe de compilation la plus étudiée et la plus difficile à mettre en œuvre dans un compilateur. La qualité du code compilé dépend en effet de la qualité de l'allocation de registres. Les deux tâches principales de l'allocation de registres sont le *vidage* de registres en mémoire, et la *fusion* de registres. Le vidage décide quelles variables seront stockées ultérieurement en registres. La fusion tient compte le plus possible des préférences entre variables, afin de minimiser les transferts entre registres. Par exemple, une affectation $x = y$ entraîne une préférence entre les variables x et y correspondant à la condition optimisante, qu'au vu de cette affectation, il serait préférable de stocker x et y dans le même registre.

1.2. Approches heuristiques

L'allocation de registres consiste à minimiser le nombre d'accès à la mémoire, étant donné un nombre fixe de registres. Classiquement, ce problème se ramène à la recherche d'une coloration de graphe. Dans le cas de l'allocation de registres, il s'agit d'un graphe d'interférences (défini dans la section 2.1), obtenu suite à une analyse de vivacité du programme à compiler. Le problème de coloration étant dans le cas général $\mathcal{NP}$-difficile, la quasi-totalité des approches imaginées ont été heuristiques ([Cha82], [BCT94], [GA96], [PP05], ...). Ces heuristiques proposent différentes combinaisons des deux phases de vidage et de fusion. Les plus simples effectuent les deux phases de manière séquentielle tandis que d'autres, plus sophistiquées, les réalisent simultanément. Les heuristiques d'allocation de registres évoluent aujourd'hui encore, par exemple afin de tenir compte de la rapidité croissante des processeurs ainsi que du coût croissant des accès à la mémoire. Pour un compilateur d'un langage tel que C, l'heuristique la plus efficace à l'heure actuelle est celle d'Appel et George [GA96]. C'est d'ailleurs celle qui a été initialement choisie dans le compilateur CompCert.

Une autre heuristique récente est celle imaginée par Palsberg et Pereira [PP05]. Si son efficacité n'est pas suffisante pour remplacer l'heuristique d'Appel et George, ses fondements sont par contre forts intéressants. En effet, suivant la piste ouverte par Andersson [And03], ceux-ci ont mesuré qu'une large majorité des graphes d'interférences ont la propriété d'être triangulés [1]. Ils affirment en effet que plus de 95% des graphes d'interférences des méthodes de la bibliothèque Java 1.5 et des 27921 graphes de référence publiés par Appel et George ([AG05]) ont cette propriété.

L'allocation de registres est également une transformation de programmes qui :
- renomme les variables du programme,
- insère des instructions de lecture et écriture en mémoire, pour chaque variable à vider en mémoire,
- supprime des affectations lorsqu'elles concernent des variables stockées dans des registres ayant été fusionnés.

Il est donc important de s'assurer que l'allocation de registres préserve le comportement des programmes. De nombreuses approches de validation reposent sur l'utilisation de techniques d'analyse statique. Peu de travaux portent sur la vérification formelle (c'est-à-dire à l'aide d'un assistant à la preuve). [Oho04] propose un système de types dédié à l'allocation de registres, ainsi qu'un algorithme d'allocation de registres correct par construction, mais les instructions considérées sont celles d'un petit langage, et cette démarche semble difficilement applicable pour un compilateur tel que CompCert. [NPP07] propose un langage dédié à l'allocation de registres, ainsi qu'un système de types. La sûreté du typage de ce système de types a récemment été prouvée en Twelf. Le but de ce travail est de fournir un cadre général permettant de comparer différentes stratégies d'allocation de registres.

[1] La définition d'un graphe triangulé est donnée dans la section 2.2.

1.3. Programmation linéaire en nombres entiers appliquée à l'allocation de registres

Les premiers travaux utilisant la programmation linéaire en nombres entiers pour l'allocation de registres sur des problèmes de petite taille furent ceux de Goodwin et Wilken en 1996 [GW96] sur architecture CISC. Les résultats furent encourageants mais pas suffisamment pour supplanter l'approche traditionnelle. Il s'agissait alors d'une formulation incluant phases de vidage et de fusion. En 2001, Appel et George [AG01] ont introduit une formulation de la phase de vidage pour les processeurs à architecture CISC[2] puis ont appliqué une méthode heuristique pour la phase de fusion (leurs tentatives d'utilisation de la programmation mathématique pour la phase de fusion se sont avérées infructueuses, particulièrement en raison du temps de résolution du problème par le solveur). Les résultats furent meilleurs que ceux de Goodwin et Wilken.

Cette année, Grund et Hack [GH07] ont élaboré un algorithme de coupes (*i.e.* une extension de la résolution par programmation mathématique) pour obtenir un résultat optimal pour la phase de fusion. Leur démarche a permis d'obtenir un résultat optimal pour 471 des 474 graphes de l'*Optimal Coalescing Challenge*[3] dans des temps très raisonnables pour la plupart des cas (430 cas sont traités en moins de 6 secondes).

2. Fondements mathématiques

2.1. Modélisation graphique de l'allocation de registres

Une *coloration* d'un graphe G est une fonction qui à chaque sommet de G associe une couleur de sorte que pour toute arête (i, j) de G les couleurs associées à i et j sont différentes. Si p est un entier strictement positif, une coloration utilisant moins de p couleurs est appelée *p-coloration*. Une coloration est dite *partielle* si certains sommets ne sont pas colorés. Une coloration est dite *optimale* si elle utilise un nombre minimal de couleurs.

Dans le cas d'une allocation de registres, le graphe à colorier est un *graphe d'interférences*, dont les sommets représentent les variables du programme à compiler. Les arêtes sont de deux types. Les arêtes d'*interférence* relient tous les sommets qui représentent des variables qui ne doivent pas occuper les mêmes registres à un instant donné de l'exécution du programme. Les arêtes de *préférence* relient tous les sommets qui représentent des variables telles qu'il existe une instruction d'affectation entre celles-ci (et il n'existe pas d'arête d'interférence entre ces sommets). Un poids est associé à chaque arête afin de tenir compte de la fréquence d'exécution des instructions, ainsi que de la fréquence d'utilisation des variables.

Dans ce qui suit, nous définissons un graphe G comme étant un triplet (S, I, P), où G est le graphe dont l'ensemble des sommets est S, l'ensemble des arêtes d'interférence est I et l'ensemble des arêtes de préférences est P. De plus, les graphes formés par S et I d'une part et S et P d'autre part sont respectivement appelés *interf-graphe* et *pref-graphe* de G.

Soient un entier strictement positif p et un graphe G. Le problème de *p-coloration avec préférences* consiste à trouver une p-coloration partielle de l'interf-graphe de G qui minimise la fonction $f = \sum_{(i,j)\in D} w_{ij} + c|NC|$, où D est l'ensemble des arêtes de préférences dont les extrémités sont de couleurs différentes, NC est l'ensemble des sommets non colorés, w_{ij} est le poids associé à l'arête (i, j), et c est une constante représentant le coût de vidage en mémoire d'un sommet non coloré. Une p-coloration avec préférences est optimale si elle minimise la fonction f.

[2] La particularité de l'architecture CISC est que lors d'une opération du processeur, certains opérandes peuvent être en mémoire, d'autres en registres, contrairement à l'architecture RISC dans laquelle tous les opérandes doivent être en registre.

[3] Il s'agit d'une bibliothèque de graphes de référence publiée par Appel pour l'optimisation de la phase de fusion.

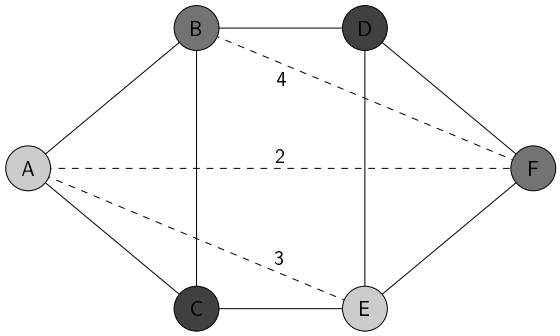

FIG. 1 – Instance du problème de 3-coloration avec préférences.

L'allocation de registres est finalement modélisée par le problème de k-coloration avec préférences appliqué au graphe d'interférences, où k désigne le nombre de registres utilisables. Chaque couleur représente un registre. La figure 1 est un exemple d'instance résolue du problème de 3-coloration avec préférences. En trait plein sont représentées les arêtes d'interférence et en pointillé les arêtes de préférences. Dans cet exemple la coloration réalisée est optimale puisqu'elle colore tous les sommets et que deux des trois arêtes de préférence ont des couleurs identiques aux deux extrémités. En effet, il est impossible de satisfaire les trois préférences car sinon les sommets A, B, E et F seraient de la même couleur et donc plusieurs contraintes de coloration seraient violées.

Étant donnée cette modélisation, la phase de vidage des registres en mémoire consiste à rechercher un ensemble de sommets k-colorable, c'est-à-dire pouvant être coloré avec k couleurs. La phase de fusion consiste à colorer cet ensemble de sommets.

2.2. Théorie des graphes et programmation mathématique

Les travaux décrits dans cet article nécessitent la définition de notions de théorie des graphes et de programmation mathématique. Nous commençons par présenter la classe des graphes triangulés qui joue un rôle central au sein de notre étude ainsi que les ordres d'élimination simpliciaux qui leur sont intimement liés pour finir par une description de la programmation linéaire en nombres entiers. Dans les définitions suivantes, $G = (S, I, P)$ désigne un graphe, et E désigne un ensemble de sommets de S.

1. Un cycle C est dit *sans corde* si aucune arête ne relie deux sommets non consécutifs de C.
2. Le *graphe induit* par E est la restriction de G aux sommets appartenant à E et aux arêtes reliant deux sommets de E.
3. Un graphe qui ne possède aucun cycle induit sans corde de longueur supérieure ou égale à quatre est dit *triangulé* (*chordal* en anglais).
4. Une *clique* est un graphe dont tous les sommets sont reliés deux à deux.
5. Un sommet s est *simplicial* dans un graphe G si le graphe induit par les voisins de s est une clique.
6. Un *ordre d'élimination simplicial* (noté *oes* ou *peo* pour *perfect elimination order* en anglais) est une permutation $(x_1, \ldots, x_n)$ de S telle que pour tout $i \in \{1, \ldots n\}, x_i$ est simplicial dans le graphe induit par $\{x_i, \ldots x_n\}$.

7. $n(G)$ désigne *l'ordre* de G, c'est-à-dire le nombre de sommets de G.
8. $m(G)$ est la *taille* de G, c'est-à-dire le nombre d'arêtes de G.
9. $\chi(G)$ est le *nombre chromatique* de G, c'est-à-dire le nombre minimal de couleurs nécessaire pour réaliser une coloration de G.
10. $\omega(G)$ est l'ordre de la plus grande clique induite de G.

Une autre méthode utilisée pour la résolution exacte de problèmes d'optimisation est la programmation mathématique. Il s'agit de décrire le problème comme la minimisation (ou la maximisation) d'une fonction (appelée *fonction économique*) sous contraintes de ses variables (égalités et inégalités). Tout programme mathématique (P) peut donc s'écrire sous la forme suivante, où X désigne l'ensemble des solutions admissibles et n désigne le nombre de variables :

$$(P) \begin{cases} Min & f(x) \\ & x \in X \subseteq \mathbb{R}^n \end{cases}$$

Nous parlerons particulièrement de la programmation linéaire en variables {0,1} (PLNE) c'est-à-dire où les variables appartiennent toutes à l'ensemble {0,1}, et où la fonction économique et toutes les contraintes du problème sont linéaires. Notons qu'il s'agit généralement de la résolution exacte de problèmes $\mathcal{NP}$-difficiles par des méthodes énumératives ce qui demande un temps de résolution exponentiel. La résolution est confiée à un solveur commercial (par exemple CPLEX [Ilo02]). L'intérêt de tels solveurs est le recours à des techniques sophistiquées qui permettent de diminuer le temps de résolution du programme mathématique.

3. Description de l'algorithme

3.1. Séparation des phases et non optimalité

L'approche qu'il a été décidé de suivre pour cette étude est analogue à celle suivie par Appel et George [AG01], c'est-à-dire la voie de la programmation linéaire en nombres entiers et du traitement séquentiel des phases de vidage et de fusion. Il s'agit cependant d'adapter la modélisation, car l'architecture du PowerPC (le langage cible de CompCert) est RISC, et non pas CISC comme dans [AG01]. Concrètement, les contraintes portant sur les variables des programmes linéaires ne sont pas les mêmes. Par exemple, l'architecture RISC oblige toute variable à être en registre au moment de son utilisation tandis que ce n'est pas obligatoire pour une architecture CISC. En contrepartie, un processeur à architecture RISC possède plus de registres dédiés au stockage des variables. Enfin, les instructions à considérer sont plus simples dans le cas d'un processeur RISC, et le nombre de contraintes est donc plus petit que dans le cas d'une architecture CISC.

Par ailleurs, contrairement à [AG01], nous traitons également la phase de fusion par programmation linéaire en nombres entiers. Ce traitement séquentiel pose un problème non négligeable. En effet, la programmation mathématique permet d'obtenir un résultat optimal pour la phase de vidage puis pour la phase de fusion mais ces deux optimisations sont dépendantes l'une de l'autre et peuvent donc ne pas déboucher sur une allocation des registres qui soit globalement optimale.

3.2. Deux processus de résolution

Nous pouvons néanmoins, dans la majorité des cas, éviter ce problème en limitant l'optimisation à la phase de fusion. En effet, la phase de vidage n'est utile que si le graphe d'interférences n'est pas k-colorable. Or, le grand nombre de registres allouables sur architecture RISC rend cette éventualité peu probable. Il suffit donc de tester si le graphe d'interférences est k-colorable pour savoir s'il est nécessaire d'effectuer la phase de vidage. Ce test est possible [4] notamment dans les graphes dont l'interf-graphe est

[4] En un temps raisonnable, où le test de k-coloration est dit polynomial.

triangulé, puisqu'il existe des algorithmes efficaces réalisant des colorations optimales dans ces graphes. Nous appliquons donc au graphe d'interférences un algorithme de coloration, l'algorithme de coloration gourmande, qui renvoie une coloration optimale si le graphe est triangulé et quelconque sinon. Si le nombre de couleurs utilisé par cette coloration est inférieur ou égal à k, il n'est pas nécessaire de réaliser la phase de vidage et donc le résultat de la phase de fusion correspond à une solution optimale du problème global d'allocation puisque cette dernière phase est traitée par programmation linéaire en nombres entiers.

La figure 2 résume cette approche. La vérification formelle en Coq de l'algorithme est composée de deux parties. La coloration est spécifiée et prouvée en Coq, tandis que les phases de vidage et de fusion sont validées *a posteriori*. Plus précisément, il est vérifié en Coq que les résultats calculés par le solveur externe représentent bien une coloration du graphe d'interférences.

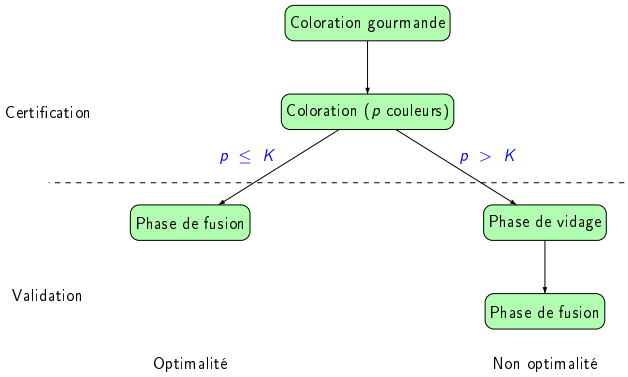

FIG. 2 – Principales étapes de l'algorithme.

3.3. Algorithme de coloration gourmande

Le test de k-colorabilité est effectué par l'algorithme de coloration gourmande. Cet algorithme fournit une coloration optimale (*i.e.* utilisant un nombre minimal de couleurs) si l'ordre dans lequel sont coloriés les sommets est l'ordre inverse d'un ordre d'élimination simplicial [Gav72]. Ces résultats ont été prouvés en Coq (*cf.* section 4.5). Il en découle que les graphes triangulés peuvent être colorés de façon optimale grâce à cet algorithme puisque la recherche d'ordre d'élimination simplicial est un problème pour lequel il existe divers algorithmes polynomiaux. Nous avons choisi d'écrire en Coq un algorithme de recherche d'un ordre d'élimination simplicial relativement naïf mais dont la correction est plus simple à montrer que pour les algorithmes les plus efficaces et dont la complexité (en temps) est comparable. L'algorithme est donné ci-dessous. Il consiste informellement à chercher un sommet simplicial s, le retirer du graphe et itérer ce procédé. Cette technique fonctionne car tout graphe induit d'un graphe triangulé est lui même triangulé et possède donc un ordre d'élimination simplicial.

L'appel à la fonction $is\_sv(s, S-T)$ teste si s est un sommet simplicial dans le graphe induit par $S-T$. Plus précisément, il est testé si s et ses voisins de $S-T$ forment une clique. De plus, un graphe est triangulé si et seulement s'il admet un ordre d'élimination simplicial [FG65]. Aussi, l'ordre renvoyé par l'algorithme 1 est un ordre d'élimination simplicial si et seulement si le graphe G est triangulé.

Algorithme 1 peo_search (G)

Entrée: Un graphe G=(S, I, P) dont l'interf-graphe est triangulé
Sortie: Un ordre $x = \{x_1, x_2, \ldots x_{n(G)}\}$ d'élimination simplicial de l'interf-graphe

```
1: i := 0, U := ∅
2: tant que i < n(G) faire
3:    trouvé := faux, T := U
4:    tant que trouvé = faux faire
5:       choisir s dans S − T
6:       si is_sv(s, S − T) alors
7:          x_{i+1} := s
8:          trouvé := vrai
9:       sinon
10:         T := T ∪ s
11:      fin si
12:   fin tant que
13:   i := i + 1; U := U ∪ {s}
14: fin tant que
```

Étant donné un ordre des sommets, l'algorithme 2 de coloration gourmande consiste à colorer les sommets selon cet ordre en affectant à chaque fois la plus petite couleur qui n'est utilisée par aucun des voisins du sommet courant x_i, en commençant la numérotation (coloration) à 1.

Algorithme 2 graph_coloring (G)

Entrée: Un graphe G
Sortie: Une coloration de G, optimale si G est triangulé

```
1: x = peo_search(G), U := ∅
2: pour tout i de n(G) à 1 faire
3:    T := get_nghbs(G, x_i), U := U ∪ x_i
4:    affecter à x_i la plus petite couleur qui n'est affectée à aucun sommet de T ∩ U
5: fin pour
```

3.4. Programmation linéaire en nombres entiers

Nous utilisons pour modéliser la phase de fusion sur un graphe $G = (S, I, P)$ le programme mathématique défini dans [GH07]. Il existe deux types de variables pour modéliser le problème : d'une part, les variables x_{ic} qui valent 1 si et seulement si le sommet i est de couleur c ; d'autre part les variables y_{ij} qui valent 1 si et seulement si (i, j) est une arête et i et j sont de couleurs différentes.

Le programme mathématique est défini dans la figure 3. Il comprend trois séries de contraintes :
- (C_1) à chaque sommet doit être affectée une et une seule couleur,
- (C_2) chaque arête d'interférence doit avoir des extrémités de couleurs différentes,
- (C_3) $y_{i,j}$ doit valoir 1 si i et j sont de couleurs différentes. En effet, si les couleurs sont différentes alors le membre droit de l'inégalité (C_3) vaut 1 lorsque c est la couleur de i.

Pour optimiser la coloration il suffit de minimiser le poids des arêtes de préférence dont les extrémités sont de couleurs différentes, c'est-à-dire à minimiser $f = \sum_{(i,j)\in P} w_{ij} \times y_{ij}$.

Nous avons également défini un modèle mathématique adapté au traitement simultané des deux phases. Ce modèle étant assez proche du précédent, il n'est pas présenté dans cet article. Il suffit en effet d'ajouter des variables x_{i0} qui valent 1 si et seulement si le sommet i n'est pas coloré et de

$$(P1)\begin{cases} Min & \displaystyle\sum_{(i,j)\in P} w_{ij} \times y_{ij} & & \\ \text{sous les contraintes} & & & \\ (C_1)\ \forall i\ \in \{1,\ldots,n(G)\}, & \displaystyle\sum_{c=1}^{k} x_{ic} & = & 1 \\ (C_2)\ \forall (i,j) \in I,\ \forall c \in \{1,\ldots,k\}, & x_{ic} + x_{jc} & \leq & 1 \\ (C_3)\ \forall (i,j) \in P,\ \forall c \in \{1,\ldots,k\}, & x_{ic} - x_{jc} & \leq & y_{ij} \\ (C_4)\ \forall i \in \{1,\ldots,n(G)\},\ \forall c \in \{1,\ldots,k\}, & x_{ic} \in \{0,1\} & & \end{cases}$$

FIG. 3 – Programme mathématique modélisant la fusion de registres.

modifier les contraintes et la fonction économique du problème en conséquence.

4. Spécification Coq

Cette partie détaille la spécification en Coq de l'algorithme de coloration gourmande, les structures de données utilisées et les théorèmes de correction et d'optimalité de la spécification. Afin de différencier les prédicats et fonctions définis lors du développement de ceux de bibliothèques existantes, les premiers sont en gras et les seconds en italique.

4.1. Définition des graphes

La structure de graphe a été définie en Coq avec la construction `Record`. Deux types de graphes ont été définis : les graphes quelconques et les graphes tels que la liste de leurs sommets est un ordre d'élimination simplicial inverse. La structure générique des graphes est la suivante.

```
Record Graph : Set := mk_Graph{
  vertices : list nat ;
  edges : list (nat×nat) ;
  (P1)  p_is_lex_sorted : is_lex_sorted edges ;
  (P2)  p_is_strict_ord : is_strict_ord edges ;
  (P3)  p_NoDup : NoDup edges ;
  (P4)  p_vertices_edges : vertices = edges_to_vertices edges }.
```

Il a été volontairement choisi de construire tout l'algorithme de coloration à partir uniquement des arêtes du graphe pour s'approcher autant que possible de la structure actuellement utilisée dans CompCert. C'est pourquoi le graphe est modélisé par une liste d'arêtes, une arête étant un couple de sommets. De même, le choix d'utilisation de liste est voulu même s'il conduit à une diminution de complexité d'implantation. En effet, la notion d'ordre d'élimination simplicial est essentielle et se représente idéalement par une liste, laquelle est par définition une permutation de la liste des sommets du graphe.

La liste *edges* des arêtes du graphe possède trois propriétés :
- (P_1) indique que la liste *edges* est triée par ordre lexicographique ;
- (P_2) stipule que la liste *edges* est strictement ordonnée, c'est-à-dire que dans chaque arête, l'identifiant du sommet source est inférieur à celui du sommet destination.
- (P_3) mentionne que la liste *edges* ne contient pas de doublons.

(P_1) et (P_2) servent à améliorer la vitesse des algorithmes. En effet, elles correspondent à des propriétés de tri qui peuvent être exécutées en temps relativement rapide et permettent de diminuer la complexité des algorithmes ou d'y incorporer des conditions d'arrêt. Ces propriétés permettent en outre d'effectuer des parcours parallèles des listes de sommets et d'arêtes du graphe. (P_3) est une propriété de bonne formation du graphe. Toutes les fonctions nécessaires pour transformer la liste initiale ont été implantées et un algorithme de tri rapide a été écrit en Coq à l'aide de la construction `Function` (*cf.* section 4.3).

La liste *vertices* représente le sous-ensemble des sommets du graphe qui possèdent au moins une arête incidente. Le graphe d'interférences étant connexe[5], cet ensemble de sommets est exactement l'ensemble de tous les sommets du graphe. De plus, cette liste est triée par ordre croissant et ne contient pas de doublons, ce qui induit qu'elle est triée par ordre strictement croissant. Ces propriétés découlent de la façon dont a été construite la liste *vertices* à partir de la liste *edges*, c'est-à-dire de la fonction **edges_to_vertices**.

Afin de spécifier ce qu'est un graphe triangulé, il est nécessaire de donner la spécification des ordres d'élimination simpliciaux. Le prédicat (**sv** x v e) signifie que x est un sommet simplicial dans la liste des sommets v par rapport aux arêtes de e. Nous ne donnons pas sa spécification car elle fait elle même appel à d'autres prédicats. La spécification des ordres d'élimination simpliciaux est décomposée en deux définitions **_peo** et **peo** afin d'en alléger l'utilisation.

```
Inductive _peo : list nat → list nat → list (nat×nat) → Prop :=
  peo_cons : ∀ (l vert : list nat) (edg : list (nat×nat)),
                Permutation vert l ⇒
                (∀ (x : nat) (ll rl : list nat), l = ll ++ x : : rl ⇒ sv x (x : : rl) edg) ⇒
                    _peo l vert edg.

Definition peo (l : list nat) (g : graph) : Prop := _peo l (vertices g) (edges g).
```

Le second type de graphes représente les graphes triangulés. Le prédicat (**est_clique** g l t) signifie que les sommets de la liste l forment une clique de taille t dans le graphe g. Ce type de graphe est donc celui où l'ordre inverse des sommets est simplicial. Ainsi, pour tout x l'ensemble des sommets qui précèdent x dans la liste *vertices* (autrement dit ceux qui sont colorés avant lui par l'algorithme de coloration gourmande) forme une clique.

```
Record Chordal_graph : Set := mk_Chordal_graph{
  gph : Graph;
  self_peo : ∀ (x : nat), In x (vertices gph) ⇒
               is_clique gph (x : : get_nghbs gph x) (length (get_nghbs gph x) + 1)}.
```

4.2. Description générale de l'implantation

L'algorithme de coloration gourmande a été écrit en Coq et son optimalité a été prouvée en Coq sur les graphes triangulés. Pour ce faire, il est nécessaire de construire un graphe *my_chordal_gph* afin de créer un enregistrement de type *Chordal_graph* car c'est sur ce type de graphe que l'algorithme de coloration permet d'obtenir une coloration optimale. La figure 4 résume les différentes étapes de la coloration de graphe.

Il faut donc tout d'abord, à partir de la liste des arêtes du graphe, construire *my_graph* de type *Graph*. Ensuite, si *my_graph* est triangulé alors il admet un ordre d'élimination simplicial. Dans ce cas, il est nécessaire de renommer les sommets de *my_graph* de sorte que la liste inverse de l'ordre d'élimination simplicial trouvé corresponde à la liste $\{1,\ldots n\}$. Dès lors, il est possible de construire un graphe *my_chordal_gph* permettant de définir un enregistrement de type *Chordal_graph*, de réaliser

[5] Un graphe connexe est un graphe pour lequel il est possible de relier toute paire de sommets par une liste d'arêtes telle que deux arêtes consécutives sont adjacentes.

la coloration de *my_chordal_gph* (et la prouver optimale), puis de renommer dans le sens inverse les sommets pour prouver que la coloration obtenue (elle-même renommée de la même façon) est une coloration optimale de *my_graph*. Dans le cas où *my_graph* n'admet pas d'ordre d'élimination simplicial, *my_graph* est coloré, mais la coloration obtenue n'est pas optimale.

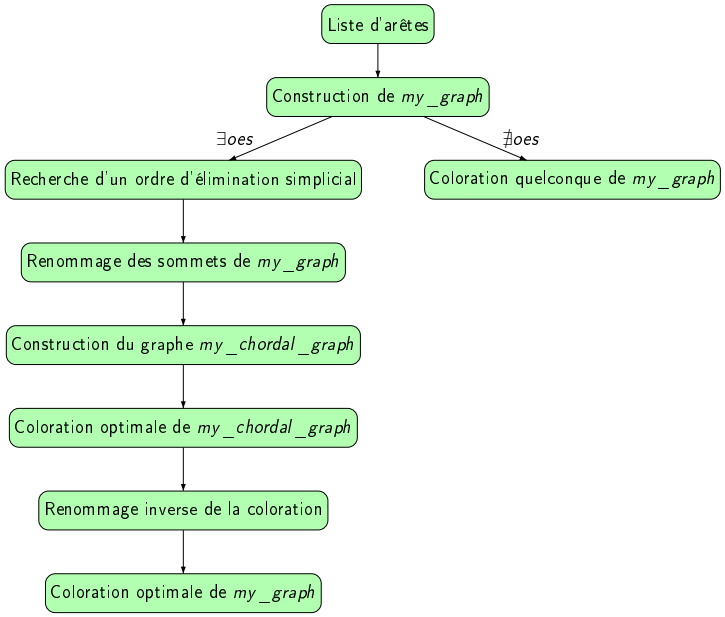

FIG. 4 – Démarche générale de la coloration.

Nous nous sommes attachés à conférer à notre implantation quelques optimisations afin d'obtenir bonne une complexité algorithmique. En particulier, l'utilisation de divers ordres sur les listes permet un gain de complexité appréciable mais allonge considérablement le développement. Les trois principales propriétés d'ordre que nous utilisons sont :

- l'ordre lexicographique sur des listes de couples,
- l'ordre sur les composantes d'un couple (la première composante doit être plus petite que la seconde),
- la croissance des listes d'entiers.

Il a également été nécessaire de définir d'autres ordres connexes à ceux-ci comme les ordres stricts et les ordres inverses. Il existe plusieurs cas de figure pour lesquels l'usage d'ordre procure un gain significatif. Nous présentons ici un cas très simple : l'élimination des doublons d'une liste.

Si l est une liste de longueur lg, alors un algorithme classique d'élimination de doublons se code en $O(lg^2)$. Par contre, si la liste est triée par ordre croissant, il est possible d'effectuer cette opération en $O(lg)$ puisqu'il suffit de vérifier récursivement que les deux éléments de tête de liste sont différents et d'en supprimer un si ce n'est pas le cas. Si la liste n'est pas triée il est donc préférable d'un point de vue de la complexité algorithmique de la trier puis de supprimer les doublons puisque le tri rapide a une complexité en $O(lg\ log(lg))$.

4.3. Recherche d'un ordre d'élimination simplicial

Lorsque le graphe est triangulé, connaître un ordre d'élimination du graphe permet d'obtenir une coloration optimale. Pour le trouver nous utilisons l'algorithme 1 (*cf.* section 3.3). Dans la définition

suivante, la fonction (**sv_search_aux** l l') permet de trouver un sommet simplicial étant donnés les sommets de la liste l et les arêtes de la liste l'. La fonction (**remove** x l) permet de supprimer le sommet x de la liste l, et la fonction (**rm** n l) supprime de la liste l tous les couples dont l'une des composantes est n.

La définition de la fonction **peo_search_aux** n'est pas structurelle puisqu'elle repose sur la décroissance de la longueur de la liste. Aussi nous utilisons la construction `Function`. Il suffit ensuite de prouver que la longueur de la liste décroît bien à chaque itération. La correction de l'algorithme de recherche d'ordre d'élimination simplicial consiste à montrer que s'il existe un ordre d'élimination simplicial pour un graphe alors l'algorithme en trouve un.

```
Function peo_search_aux (l : list nat) (l' : list (nat×nat)) {measure length l} : list nat :=
  match l with nil ⇒ nil
  | _ ⇒ (sv_search_aux l l') : : (peo_search_aux (remove eq_nat_dec (sv_search_aux l l') l)
                                                 (rm (sv_search_aux l l') l'))
  end.

Definition peo_search (g : graph) : list nat :=peo_search_aux (vertices g) (edges g).

Lemma peo_peo_search : ∀ (g : graph), (∃ l : list nat, peo l g) ⇒ peo (peo_search g) g.
```

4.4. Algorithme de coloration gourmande

Une fois l'ordre d'élimination simplicial trouvé, il n'est pas encore possible de définir une structure de type *Chordal_graph* puisque l'ordre d'élimination trouvé n'est pas croissant. Il faut donc renommer les sommets de façon à ce que l'ordre d'élimination corresponde à une liste croissante d'entiers. La fonction inverse doit aussi être spécifiée afin de pouvoir renommer les sommets après que la coloration du graphe ait été réalisée. Cette phase est périlleuse mais peu intéressante, c'est pourquoi elle n'est pas détaillée ici. Pour la consulter le lecteur pourra se rapporter à la page web du développement complet.

Une coloration est représentée par une liste de couples (s, c) où c représente la couleur (représentée par un entier) affectée au sommet s. L'algorithme de coloration gourmande construit la coloration au fur et à mesure du parcours de la liste des sommets du graphe. Au moment de la coloration d'un sommet, il est nécessaire de connaître les couleurs affectées aux sommets déjà colorés pour pouvoir choisir la couleur du sommet courant. Il faut donc stocker la coloration dans un accumulateur *col* qui est initialisé par la liste vide. La fonction (**get_available_color** g x *col*) permet de rechercher dans le graphe g la plus petite couleur n'étant affectée à aucun voisin du sommet x par la coloration partielle *col*.

```
Fixpoint graph_coloring_aux (g : graph) (l : list nat) (col : list (nat×nat)) {struct l} :
            list (nat×nat) :=
  match l with nil ⇒ col
  | x : : l" ⇒ graph_coloring_aux g l" ((x, get_available_color g x col) : : col)
  end.

Definition graph_coloring (g : graph) := graph_coloring_aux g (vertices g) nil.
```

4.5. Propriétés prouvées

Nous avons prouvé en Coq deux familles de propriétés concernant d'une part la correction d'une coloration, et d'autre part l'optimalité de l'algorithme de coloration gourmande dans les graphes triangulés.

Le lemme **is_coloring_graph_coloring** est le lemme de correction de la coloration gourmande.

Il établit que pour tout graphe g, la coloration calculée par la fonction de coloration gourmande **graph_coloring** est bien une coloration valide de g. Une coloration valide est définie par le prédicat **is_coloring**. Soit une coloration *col* d'un graphe g. Soit k tel que les couleurs de *col* sont numérotées de 1 à k. Alors, *col* est valide si et seulement si tout sommet de g possède une et une seule couleur comprise entre 1 et k (lignes (1) à (3)), et si tout couple de sommets formant une arête d'interférence est coloré par deux couleurs distinctes (ligne (4)).

```
Inductive is_coloring : graph → list (nat×nat) → Prop :=
  coloring_cons : ∀ (g : graph) (col : list (nat×nat)),
  (1)      (∀ (x : nat), In x (vertices g) ⇔ ∃ c, In (x,c) col)⇒
  (2)      no_fst_dup col ⇒
  (3)      (∀ (x cx : nat), In (x,cx) col ⇒ 1 ≤ cx) ⇒
  (4)      (∀ (x y cx cy : nat), In (x,y) (edges g)⇒ In (x,cx) col ⇒ In (y,cy) col ⇒ cy ≠ cx) ⇒
    is_coloring g col.

Lemma is_coloring_graph_coloring : ∀ (g : graph), is_coloring g (graph_coloring g).
```

De plus, nous validons *a posteriori* les colorations calculées par le solveur externe que nous avons utilisé. Étant donné un graphe g, la validation *a posteriori* d'une coloration *col* calculée par un solveur externe consiste à vérifier en Coq le lemme **is_coloring** *g col*. Nous vérifions ainsi la même propriété que celle actuellement vérifiée dans CompCert.

Le lemme suivant est utile pour prouver l'optimalité de la coloration gourmande. Soit *my_peo* l'ordre d'élimination simplicial ayant été trouvé, et *my_chordal_gph* le graphe triangulé (de la structure de type Chordal_graph) obtenu à partir de *my_peo* après renommage des sommets. Le lemme **is_coloring_renaming** établit que si *col* est une coloration du graphe triangulé *my_chordal_graph*, alors la coloration obtenue par renommage de la coloration *col* est une coloration du graphe initial *my_graph*. Dans ce lemme, **coloring_renaming** est la fonction qui renomme une coloration afin de rétablir la numérotation des sommets du graphe d'origine.

```
Lemma is_coloring_renaming : ∀ (col : list (nat×nat)),
        is_coloring my_chordal_gph col ⇒
        is_coloring my_graph (coloring_renaming col (rev my_peo)).
```

Le lemme **coloration_optimality** établit l'optimalité de la coloration gourmande. L'optimalité consiste à vérifier que toute coloration valide du graphe utilise au moins autant de couleurs que celle renvoyée par l'algorithme, ou de manière équivalente que la plus grande couleur utilisée par toute coloration est supérieure à la plus grande couleur renvoyée par la coloration de l'algorithme. La fonction **max_color** renvoie le maximum des deuxièmes composantes d'une liste d'entiers (donc ici la couleur maximale de la coloration).

```
Lemma coloring_optimality : ∀ (col : list (nat×nat)),
        is_coloring my_graph col→
        max_color (coloring_renaming (graph_coloring my_chordal_gph) (rev my_peo)).
          ≤ max_color col.
```

La preuve [Wes00] repose sur la propriété suivante : Si $(x_1, \ldots x_{n(G)})$ est un ordre d'élimination simplicial de G, et si l'algorithme de coloration gourmande est appliqué selon l'ordre des sommets $(x_{n(G)}, \ldots x_1)$ alors la coloration est optimale.
Soit G un graphe triangulé et x un ordre d'élimination simplicial de G. Soit i appartenant à $\{1, \ldots n(G)\}$. x étant un ordre d'élimination simplicial, x_i est un sommet simplicial du graphe induit par $\{x_i, \ldots x_n\}$ c'est-à-dire du graphe induit par les sommets déjà colorés et x_i. Ainsi, le graphe induit par x_i et ses voisins déjà colorés est une clique. Soit t_i la taille de cette clique. La couleur affectée à x_i est donc inférieure ou égale à t_i. Cette inégalité étant valide pour tout i, la plus grande couleur

utilisée est inférieure ou égale à la taille de la plus grande clique. Ainsi, nous obtenons l'inégalité $\chi(G) \leq \omega(G)$. De plus, l'inégalité inverse est évidente puisqu'il faut au moins autant de couleurs pour colorer G que pour colorer sa plus grande clique induite, ce qui permet de déduire l'égalité de $\chi(G)$ et $\omega(G)$ et donc l'optimalité de la coloration obtenue.

Notre développement Coq représente environ 10000 lignes de code. Les spécifications et les énoncés des preuves représentent respectivement 4% et 12% du développement. Environ 360 lemmes ont été prouvés. La principale difficulté a été de définir de nombreuses structures de données ainsi que des ordres sur ces structures. Le mécanisme d'extraction automatique de Coq a permis de générer un programme Caml effectuant la coloration gourmande d'un graphe triangulé. Ce programme représente 400 lignes de Caml.

Ce développement Coq a été l'occasion d'utiliser la construction `Function` afin de définir des fonctions récursives non structurelles. Enfin, nous avons été confronté à la lenteur du typeur de Coq. En effet, sur certains lemmes, Coq passait de 30 à 60 minutes à valider la fin de la preuve (la ligne `Qed.`).

Conclusion

Cet article a présenté la vérification formelle en Coq d'un algorithme exact de coloration avec préférences de graphe dédié à l'allocation de registres du compilateur certifié CompCert. L'algorithme est composé de deux phases : 1) la coloration gourmande dont nous avons prouvé la correction dans les graphes quelconques ainsi que l'optimalité dans les graphes triangulés, et 2) une étape de programmation linéaire en nombre entiers qui est résolue par un solveur externe et dont le résultat est validé *a posteriori*.

Afin d'améliorer cette validation *a posteriori*, nous cherchons actuellement à spécifier en Coq la notion de programme linéaire (via une bibliothèque dédiée que nous comptons développer), qui serait toujours résolu par un solveur externe, mais qui serait vérifié par le programme linéaire de Coq. Il s'agit donc de construire la coloration de façon interne à Coq, et de prouver en Coq que toute solution valide du programme linéaire correspond à une coloration valide du graphe d'interférences.

À plus long terme, nous souhaitons vérifier formellement d'autres méthodes de recherche opérationnelle, qui seraient utiles pour améliorer notre allocation de registres et faciliter l'utilisation des méthodes d'optimisation dans les développements de programmes certifiés.

Références

[AG01] Andrew W. Appel and Lal George. Optimal spilling for CISC machines with few registers. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 243–253, 2001.

[AG05] Andrew W. Appel and Lal George. 27,921 actual register-interference graphs generated by standard ML of New Jersey, version 1.09 – `http ://www.cs.princeton.edu/~appel/graphdata/`, 2005.

[And03] Christian Andersson. Register allocation by optimal graph coloring. In *Compiler Construction (CC)*, pages 33–45, 2003.

[BCT94] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *Transactions on Programming Languages and Systems (TOPLAS)*, 16(3) :428 – 455, 1994.

[BDL06] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *FM 2006 : Int. Symp. on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer, 2006.

[Cha82] G J Chaitin. Register allocation and spilling via graph coloring. *Symposium on Compiler Construction*, 17(6) :98 – 105, 1982.

[FG65] D.R. Fulkerson and O.A. Gross. Incidence matrices and interval graphs. *Pac. J. Math.*, 15 :835–855, 1965.

[GA96] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3) :300–324, 1996.

[Gav72] Fănică Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SIAM J. Comput.*, 1 :180–187, 1972.

[GH07] Daniel Grund and Sebastian Hack. A fast cutting-plane algorithm for optimal coalescing. volume 4420 of *Lecture Notes in Computer Science*, pages 111–125. Springer, 2007.

[GW96] David W. Goodwin and Kent D. Wilken. Optimal and near-optimal global register allocations using 0-1 integer programming. *Softw. Pract. Exper.*, 26(8) :929–965, 1996.

[Ilo02] Ilog. Ilog ampl cplex system, version 8.0, user's guide, 2002.

[Ler06] Xavier Leroy. Formal certification of a compiler back-end or : Programming a compiler with a proof assistant. *33rd symposium Principles of Programming Languages*, pages 42–54, 2006.

[NPP07] V. Krishna Nandivada, Fernando Magno Quintão Pereira, and Jens Palsberg. A framework for end-to-end verification and evaluation of register allocators. In *Static Analysis, 14th Int. Symp., SAS 2007, August, 2007, Proc.*, volume 4634 of *Lecture Notes in Computer Science*, pages 153–169. Springer, 2007.

[Oho04] Atsushi Ohori. Register allocation by proof transformation. *Science Computer Programming*, 50(1-3) :161–187, 2004.

[PP05] Fernando Magno Quintão Pereira and Jens Palsberg. Register allocation via coloring of chordal graphs. *Programming Languages and Systems, 3rd Asian Symp., APLAS 2005, Japan, November, 2005, Proc.*, 3780 :315–329, 2005.

[Wes00] Douglas B. West. *Introduction to Graph Theory (2nd Edition)*. Prentice Hall, August 2000.