Graphes et couplages en Coq

Catherine Dubois, Sourour Elloumi, Benoit Robillard, Clément Vincent

Graphes et couplages en Coq

Catherine Dubois & Sourour Elloumi & Benoit Robillard & Clément Vincent

CEDRIC/ENSIIE,
ENSIIE, 1 square de la résistance 91025 Évry
nom.prenom@ensiie.fr

Résumé

Nous proposons une formalisation en Coq des graphes orientés et non orientés sans arête multiple. La bibliothèque développée offre non seulement l'expressivité requise pour exprimer et démontrer des propriétés sur les graphes mais aussi une implantation purement fonctionnelle permettant de mettre en œuvre efficacement les algorithmes de graphes. Nous spécifions ensuite, à l'aide de cette bibliothèque, les notions de couplage et de couverture d'arêtes et développons un vérificateur formellement vérifié dont l'objectif est de certifier le résultat d'une fonction qui calcule un couplage de cardinalité maximale. Le code exécutable de ce vérificateur est obtenu grâce au mécanisme d'extraction de Coq. Ce travail est une première contribution d'un projet plus ambitieux qui concerne le développement d'un algorithme de filtrage formellement vérifié pour la contrainte de différence (ALLDIFF) pour des domaines finis. Ce dernier algorithme utilise de nombreuses manipulations de graphe dont le calcul d'un couplage de cardinalité maximale.

1. Introduction

Disposer d'une bibliothèque couvrant de nombreuses notions dans un assistant à la preuve est un atout indéniable. Par exemple, la mise à disposition en Coq de modules de spécification et d'implantation de structures de données telles que les ensembles ou les tables (maps) finis [12] a permis aux utilisateurs de Coq de réutiliser ces développements dans d'autres formalisations. La conséquence en est également une forme d'uniformisation. Néanmoins, même si de nombreux travaux portent ou ont porté sur les graphes, on ne peut que constater l'absence en Coq d'une bibliothèque de graphes permettant de spécifier, prouver et calculer (pour les preuves et les programmes). Il en était de même en Isabelle jusque récemment [20].

Dans cet article, nous présentons le noyau d'une bibliothèque de graphes pour Coq qui combine expressivité et efficacité. Une grande expressivité est nécessaire pour écrire et prouver des propriétés sur les graphes de la manière la plus naturelle possible. L'efficacité passe par la définition de structures de données qui permettront d'obtenir un code extrait efficace pour les algorithmes définis et vérifiés formellement en Coq.

Nous utilisons ensuite cette bibliothèque pour spécifier la notion de couplage [3]. Nous développons un vérificateur formellement vérifié dont l'objectif est de certifier le résultat d'une fonction qui calcule un couplage de cardinal maximal. Pour les besoins de vérification, nous formalisons également la notion de couverture d'arêtes qui nous sert de certificat d'optimalité. Le code exécutable de ce vérificateur est obtenu grâce au mécanisme d'extraction de Coq [16]. Ce travail est une première contribution d'un projet plus ambitieux qui concerne le développement d'un algorithme de filtrage formellement vérifié pour la contrainte de différence (ALLDIFF) pour des domaines finis [21]. Ce dernier algorithme utilise de nombreuses manipulations de graphe dont le calcul d'un couplage de cardinal maximal que nous certifions en utilisant le vérificateur formellement vérifié précédent.

Dans la suite de l'article, nous commençons par présenter, dans la section 2, la bibliothèque de graphes Coq en expliquant son architecture et les différents modules fournis. Le code Coq correspondant est disponible à l'adresse suivante http://www.ensiie.fr/~robillard/Graph_Library/. La section 3 présente notre utilisation de la bibliothèque précédemment décrite pour formaliser les notions de couplage et de couverture d'arêtes. La section 4 est consacrée à la définition d'un vérificateur pour les couplages de cardinal maximal et à sa preuve de correction. La section 5 présente le contexte d'utilisation du vérificateur précédemment décrit. Avant de conclure, nous présentons, dans la section 6, quelques travaux connexes.

2. La bibliothèque de graphes

Cette bibliothèque a été développée par B. Robillard, elle résulte en partie de ses travaux sur la formalisation en Coq d'un algorithme d'allocation de registres [5, 4]. Néanmoins, la bibliothèque améliore de plusieurs manières la formalisation des graphes d'interférence utilisés dans cette allocation de registres. En particulier, elle permet trois vues différentes des graphes, la présence d'étiquettes sur les sommets et l'utilisation d'itérateurs.

La bibliothèque définit le type des graphes orientés et non orientés sans arête multiple, paramétré par le type des sommets et des étiquettes éventuellement associées aux arêtes et sommets. Dans la suite, par abus de langage, nous utiliserons parfois, le mot arête pour désigner une arête dans un graphe non orienté mais aussi pour désigner un arc dans le cas d'un graphe orienté.

2.1. Architecture globale

La bibliothèque de graphes comprend différents types de modules et modules dont l'organisation est présentée figure 1.

L'interface des graphes orientés (DG), un type de module en Coq, décrit la spécification générale des graphes. Cette interface est spécialisée pour décrire l'interface des graphes non orientés (UG), considérés ici comme des graphes orientés symétriques. La spécialisation signifie ici que le type de module des graphes non orientés est en relation de sous-typage avec le type des modules des graphes orientés. Chacune des deux interfaces est étendue en une interface dite *constructive*[1] (CDG et UDG) qui propose des fonctions de construction de graphes. Les spécifications décrites dans DG (resp. UG) sont générales et attendues pour n'importe quel graphe orienté (resp. non orienté) alors que les fonctions de construction, spécifiées dans CDG (resp. CUG), se verront remises en question en cas de graphes particuliers aux contraintes fortes. Par exemple, une spécification des arbres sera compatible avec UG mais non avec CUG car les fonctions de construction sont bien plus restrictives pour les arbres que pour les graphes non orientés.

La bibliothèque propose une implantation des graphes orientés et non orientés, répondant aux spécifications présentées dans CDG et CUG.

Enfin, la bibliothèque est extensible. Ainsi, on peut ajouter par exemple la notion de chemin ou de couplage. Cet ajout dans la bibliothèque se fait au niveau le plus abstrait possible. Ainsi, le développement présenté dans les sections 3 et 4 est défini comme un foncteur qui prend en paramètre un module de type graphe non orienté (Couplage dans l'architecture représentée figure 1).

1. Constructif, ici, ne fait aucunement référence à la logique constructive. Néanmoins la bibliothèque des graphes, présentée dans cet article, est écrite en logique intuitionniste.

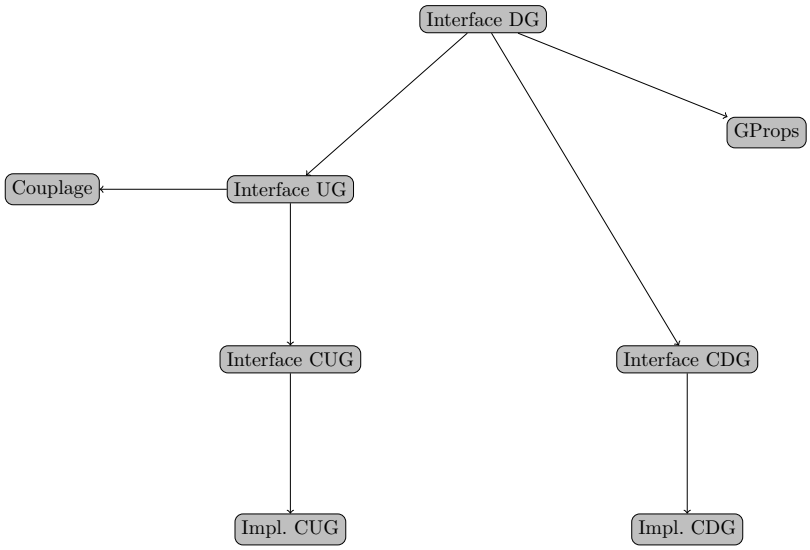

FIGURE 1 – Architecture de la bibliothèque. "DG" désigne les graphes orientés, "UG" les graphes non orientés et le préfixe C désigne les versions constructives.

2.2. Spécification des graphes orientés - DG

Le type de module DG est paramétré par un module *Vertex* de type *OrderedType* qui représente le type des sommets et par les types des étiquettes associées d'une part aux sommets (*VLabel*) et d'autre part aux arêtes (*ELabel*) [2]. Le type des sommets est accompagné d'un ordre total et d'une égalité décidable. Le type des arêtes est construit à partir du type des sommets, comme le produit $Vertex.t * Vertex.t$, dans le module *Edge*. On note *fst* (resp. *snd*) la fonction qui renvoie l'extrémité initiale (resp. finale) d'un arc. Le prédicat noté EXT, défini ci-dessous, est satisfait si et seulement si un sommet v est une extrémité d'une arête e. Enfin, le module *Edge* définit la fonction *permute* qui permute les extrémités d'une arête. Cette fonction est particulièrement utile pour les graphes non orientés.

`Definition` e EXT v `:=` $(fst\ e) = v \vee (snd\ e) = v$.

Différentes représentations ou caractérisations sont généralement proposées pour les graphes. La définition la plus simple - et c'est celle que l'on trouve le plus fréquemment dans un ouvrage de théorie des graphes - considère un graphe comme un ensemble de sommets et un ensemble d'arêtes. Un graphe est aussi souvent vu comme une relation entre sommets, représentée par une matrice. Enfin, un graphe peut être décrit par une fonction d'adjacence qui, à chaque sommet, associe l'ensemble de ses voisins. Afin de laisser à l'utilisateur le plus de souplesse possible pour exprimer des calculs ou des propriétés sur les graphes, la bibliothèque offre les trois vues et permet de passer de l'une à l'autre.
Nous insistons sur le fait que ceci est décorrélé de l'implantation concrète du graphe (qui utilise des arbres AVL dans l'implantation que nous proposons dans la bibliothèque).

Ainsi, l'ensemble des sommets munis de leur éventuelle étiquette est obtenu à l'aide la fonction V. Plus précisément, $(V\ g)$ est une table qui, à tout sommet du graphe, associe son éventuelle étiquette. De la même façon, l'ensemble des arêtes d'un graphe est obtenu avec la fonction E. La représentation matricielle est introduite via la fonction [3] *link* de type requis $t \to Vertex.t \to Vertex.t \to option\ (option$

2. Dans la suite, lorsqu'il n'y a pas d'ambiguité possible, on écrira f au lieu de $M.f$.
3. L'utilisation d'une fonction (*link*) pour modéliser la matrice d'adjacence est possible car nous ne formalisons pas les graphes multi-arêtes, c'est-à-dire les graphes où, par deux sommets, peuvent passer plusieurs arêtes.

ELabel) où t désigne ici le type abstrait des graphes. Ainsi, $link\ g\ v_1\ v_2\ = None$ signifie qu'il n'y a pas d'arc du sommet v_1 vers le sommet v_2 dans le graphe g, $link\ g\ v_1\ v_2 = Some\ l$ signifie qu'il y a un arc de v_1 à v_2. Dans ce dernier cas, la présence ou non d'une étiquette dépend de la valeur de l : si elle vaut $None$, l'arc n'est pas étiqueté, si l vaut $Some\ ll$, il y a une étiquette de valeur ll. Enfin, la dernière vue se décompose en deux fonctions, *successors* et *predecessors*, qui associent à tout sommet d'un graphe l'ensemble de ses successeurs ou prédécesseurs munis de l'éventuelle étiquette de l'arête correspondante. L'ensemble des successeurs ou prédécesseurs est ici encore représenté par une table dont les éléments sont de type *option ELabel*.

Le module DG introduit deux prédicats d'appartenance d'un sommet à un graphe, $\in_\bullet$ et $\in^L_\bullet$. Le premier prédicat spécifie qu'un sommet est dans le graphe, le second qu'un sommet étiqueté par une valeur donnée est dans le graphe. De même, pour les arêtes, on trouve les prédicats $\in_\rightarrow$ et $\in^L_\rightarrow$.

La spécification - complète - de ces différents éléments est donnée ci-dessous. Chaque formule est accompagnée de sa transcription en langue naturelle. Nous avons volontairement omis les quantificateurs et les types pour plus de concision : g désigne un graphe quelconque, v un sommet, e une arête et l une « étiquette »(de sommet ou d'arête selon le contexte). Plus précisément l désigne une valeur de type *option VLabel* ou *option ELabel*. Enfin le symbole $\in_m$ désigne l'appartenance d'un élément au domaine d'une table ou l'appartenance d'une association (notée à l'aide de $\mapsto$) à une table, selon le contexte.

(INEXT) $e \in_\rightarrow g \Rightarrow (fst\ e) \in_\bullet g \wedge (snd\ e) \in_\bullet g$

Si une arête e est dans le graphe g alors ses sommets extrémités sont aussi dans le graphe g.

(INV) $v \in_\bullet g \ \Leftrightarrow\ v \in_m (V\ g)$

Le sommet v est dans le graphe g ssi il est dans le domaine de la table des sommets $(V\ g)$.

(INVL) $(v, l) \in^L_\bullet g \ \Leftrightarrow (v \mapsto l) \in_m (V\ g)$

Le sommet v étiqueté par l est dans g ssi il figure dans la table des sommets (V g) avec l'étiquette l.

(INE) $e \in_\rightarrow g \ \Leftrightarrow\ e \in_m (E\ g)$

L'arête e est dans le graphe g ssi elle est dans le domaine de la table des arêtes (E g).

(INEL) $(e, l) \in^L_\rightarrow g \ \Leftrightarrow\ (e \mapsto l) \in_m (E\ g)$

L'arête e étiquetée par l est dans g ssi elle figure dans la table des arêtes avec l'étiquette l.

(INLL) $((v_1, v_2), l) \in^L_\rightarrow g \ \Leftrightarrow\ link\ g\ v_1\ v_2 = Some\ l$

L'arête allant de v_1 *à* v_2 *étiquetée par l est dans g ssi link* $g\ v_1\ v_2$ *vaut Some l*

(INSL) $((v_1, v_2), l) \in^L_\rightarrow g \ \Leftrightarrow\ (v_2 \mapsto l) \in_m (successors\ g\ v_1)$

L'arête (v_1, v_2) *étiquetée par l est dans g ssi* v_2 *est un successeur de* v_1 *dans g et l'arête est étiquetée par l.*

(INPL) $((v_1, v_2), l) \in^L_\rightarrow g \ \Leftrightarrow\ (v_1 \mapsto l) \in_m (predecessors\ g\ v_2)$

L'arête (v_1, v_2) *étiquetée par l est dans g ssi* v_1 *est un prédécesseur de* v_2 *dans g et l'arête est étiquetée par l.*

L'interface DG spécifie également différents itérateurs sur les graphes qui permettent d'itérer une fonction sur les sommets d'un graphe, les arêtes d'un graphe ou encore les voisins (successeurs ou prédécesseurs) d'un sommet dans un graphe. Ces itérateurs auraient pu être définis hors de cette interface, ils y sont volontairement placés de manière à pouvoir les implanter efficacement, au plus proche de la structure de données attachée à la représentation des graphes. La spécification de ces itérateurs est donnée ci-dessous et utilise les itérateurs sur les tables (notés $fold_m$) présents dans la

bibliothèque standard de Coq. Comme f peut prendre des types différents, la variable f est introduite avec son type. Les autres variables sont comme précédemment quantifiées universellement et typées selon la convention donnée plus haut (avec a de type A).

(FV) $\forall A, \forall f : t \to Vertex.t \to option\ VLabel \to A \to A, fold_\bullet\ f\ g\ a = fold_m\ (f\ g)\ (\mathtt{V}\ g)\ a$

Itérer une fonction f sur les sommets de g avec la valeur initiale a donne le même résultat que l'itération de la fonction $(f\ g)$ sur la table des sommets $(V\ g)$ avec la valeur initiale a.

(FE) $\forall A, \forall f : t \to Edge.t \to option\ ELabel \to A \to A, fold_\to\ f\ g\ a = fold_m\ (f\ g)\ (E\ g)\ a$

Itérer une fonction f sur les arêtes de g avec la valeur initiale a donne le même résultat que l'itération de la fonction $(f\ g)$ sur la table des arêtes $(E\ g)$ avec la valeur initiale a.

(FS) $\forall A, \forall f : t \to Vertex.t \to Vertex.t \to option\ ELabel \to A \to A,$
$fold_{succ}\ f\ v\ g\ a = fold_m\ (f\ g\ v)\ (successors\ g\ v)\ a$

Itérer une fonction f sur les successeurs de v dans g avec la valeur initiale a donne le même résultat que l'itération de la fonction $(f\ g\ v)$ sur la table des successeurs $(successors\ g\ v)$ avec la valeur initiale a.

(FP) $\forall A, \forall f : t \to Vertex.t \to Vertex.t \to option\ ELabel \to A \to A,$
$fold_{pred}\ f\ v\ g\ a = fold_m\ (f\ g\ v)\ (predecessors\ g\ v)\ a$

Itérer une fonction f sur les prédécesseurs de v dans g avec la valeur initiale a donne le même résultat que l'itération de la fonction $(f\ g\ v)$ sur la table des prédécesseurs $(predecessors\ g\ v)$ avec la valeur initiale a.

De la spécification décrite précédemment découlent des propriétés qui sont démontrées dans le module GProps. Par exemple on démontre le théorème suivant (où les quantifications sont omises) : $(v, l) \in^L_\bullet g \Rightarrow v \in_\bullet g$.

2.3. Spécification des graphes orientés constructifs - CDG

La spécification présentée dans cette sous-section étend la précédente avec des fonctions de construction, pour ajouter et enlever des sommets et des arêtes. Elle comprend également la spécification du graphe vide, $G_\emptyset$, sans sommet ni arête.

(EG1) $v \notin_\bullet G_\emptyset$

Le sommet v n'appartient pas au graphe vide $G_\emptyset$.

(EG2) $e \notin_\to G_\emptyset$

L'arête e n'appartient pas au graphe vide $G_\emptyset$.

La fonction *add_vertex* ajoute un sommet éventuellement étiqueté dans un graphe alors que la fonction *remove_vertex* permet de supprimer un sommet ainsi que toutes les arêtes incidentes au sommet supprimé. Les spécifications des deux fonctions, données ci-dessous, utilisent le prédicat d'appartenance avec étiquette. Cependant, les propriétés se déclinent également avec le prédicat sans étiquette et leur preuve découle des propriétés ci-dessous. Elles sont prouvées dans un module annexe.

Dans les spécifications qui suivent, v' et l' désignent respectivement un sommet et une étiquette (de même que l dans le code Coq précédent, l' a le type *option VLabel* ou *option ELabel*).

(ADV1) $(v', l') \in^L_\bullet (add\_vertex\ v\ g\ l) \Leftrightarrow (v' \notin_\bullet g \wedge v' = v \wedge l = l') \vee (v', l') \in^L_\bullet g$

Le sommet v' étiqueté par l' est dans le graphe obtenu à partir de g en ajoutant le sommet v d'étiquette l ssi ou bien v' et l'sont égaux resp. à v et l et v n'est pas dans g ou bien le sommet v' d'étiquette l' est dans le graphe g.

(ADV2) $(e, l') \in^L_{\rightarrow} (add\_vertex\ v\ g\ l) \Leftrightarrow (e, l') \in^L_{\rightarrow} g$

L'arête e étiquetée par l' est dans le graphe obtenu à partir de g en ajoutant le sommet v d'étiquette l ssi elle est dans le graphe g.

(RMV1) $(v', l') \in^L_{\bullet} (remove\_vertex\ v\ g) \Leftrightarrow (v', l') \in^L_{\bullet} g \wedge v' \neq v$

Le sommet v' étiqueté par l' est dans le graphe obtenu à partir de g en supprimant le sommet v ssi il est dans le graphe g et si v et v' sont différents.

(RMV2) $(e, l) \in^L_{\rightarrow} (remove\_vertex\ v\ g) \Leftrightarrow (e, l) \in^L_{\rightarrow} g \wedge \neg\ (e\ \text{EXT}\ v)$

L'arête e étiquetée par l est dans le graphe obtenu à partir de g en supprimant le sommet v ssi cette arête est dans g et si v n'est pas une des extrémités de e.

La fonction *add_edge* permet d'ajouter une arête, éventuellement étiquetée, dans un graphe. Ses propriétés sont écrites ci-dessous. Elles mettent en évidence que l'ajout d'une arête à un graphe requiert que ses extrémités soient déjà dans le graphe. Si ce n'est pas le cas, le graphe reste inchangé (voir les propriétés **ADE5** et **ADE6**). Nous utilisons le prédicat noté $\in_{\text{EXT}}$ pour exprimer que les deux extrémités d'une arête sont dans le graphe :

`Definition` $e\ \in_{\text{EXT}}\ g$ `:=` $(fst\ e) \in_{\bullet} g \wedge (snd\ e) \in_{\bullet} g$.

(ADE1) $(v', l') \in^L_{\bullet} (add\_edge\ e\ g\ l) \Leftrightarrow (v', l') \in^L_{\bullet} g$

Le sommet v' étiqueté par l' est dans le graphe obtenu à partir de g en ajoutant l'arête e étiquetée par l ssi le sommet v' étiqueté par l' est dans g.

(ADE2) $(e', l') \in^L_{\rightarrow} g\ \wedge e \neq e' \Rightarrow (e', l') \in^L_{\rightarrow} (add\_edge\ e\ g\ l)$

Si l'arête e' étiquetée par l' est dans le graphe g et si e et e' sont deux arêtes différentes, alors l'arête e' étiquetée par l' est dans le graphe obtenu à partir de g en ajoutant l'arête e étiquetée par l.

(ADE3) $e \in_{\text{EXT}} g \Rightarrow (e, l) \in^L_{\rightarrow} (add\_edge\ e\ g\ l)$

Si les deux extrémités de l'arête e sont dans g alors l'arête e étiquetée par l est dans le graphe obtenu à partir de g en ajoutant l'arête e étiquetée par l.

(ADE4) $e \in_{\text{EXT}} g \wedge (e', l') \in^L_{\rightarrow} (add\_edge\ e\ g\ l) \wedge e \neq e' \Rightarrow (e', l') \in^L_{\rightarrow} g$

Si les deux extrémités de l'arête e sont dans g et si l'arête e', différente de e et étiquetée par l', est dans le graphe obtenu à partir de g en ajoutant l'arête e étiquetée par l alors l'arête e' étiquetée par l' est dans g.

(ADE5) $e \notin_{\text{EXT}} g \Rightarrow ((v', l') \in^L_{\bullet} (add\_edge\ e\ g\ l) \Leftrightarrow (v', l') \in^L_{\bullet} g)$

Si l'arête e a une de ses extrémités qui n'est pas dans g, les sommets du graphe obtenu à partir de g en ajoutant l'arête e étiquetée par l sont ceux du graphe g.

(ADE6) $e \notin_{\text{EXT}} g \Rightarrow ((e', l') \in^L_{\bullet} (add\_edge\ e\ g\ l) \Leftrightarrow (e', l') \in^L_{\bullet} g)$

Si l'arête e a une de ses extrémités qui n'est pas dans g, les arêtes du graphe obtenu à partir de g en ajoutant l'arête e étiquetée par l sont celles du graphe g.

La fonction qui permet d'enlever une arête, *remove_edge*, est plus simple à spécifier car enlever une arête n'a pas d'impact sur le reste du graphe.

(RME1) $(v, l) \in^L_{\bullet} (remove\_edge\ v_1\ v_2\ g\ ) \Leftrightarrow (v, l) \in^L_{\bullet} g$

Les sommets du graphe obtenu à partir de g en retirant l'arête (v_1, v_2) sont ceux de g.

(RME2) $v_2 \notin (successors\ (remove\_edge\ v_1\ v_2\ g)\ v_1)$

Le sommet v_2 n'est plus un des successeurs de v_1 dans le graphe obtenu à partir de g en retirant l'arête (v_1, v_2).

(RME3) $v_1 \notin (predecessors\ (remove\_edge\ v_1\ v_2\ g)\ v_2)$

Le sommet v_1 n'est plus un des prédécesseurs de v_2 dans le graphe obtenu à partir de g en retirant l'arête (v_1, v_2).

2.4. Spécification des graphes non orientés - UG et CUG

La spécification des graphes non orientés est quasiment la même que celle des graphes orientés. La seule différence est que si une arête (v_1, v_2) appartient au graphe alors l'arête (v_2, v_1) y appartient également. Un graphe non orienté est donc considéré ici comme un graphe orienté symétrique.

(SYM) $(e, l) \in^L_{\rightarrow} g \ \Leftrightarrow\ (permute\ e, l) \in^L_{\rightarrow} g$

Cet invariant amène quelques petites modifications dans les spécifications des fonctions *add_edge* et *remove_edge*. Par exemple, la propriété analogue à **ADE2** requiert l'hypothèse supplémentaire $e' \neq (permute\ e)$.

2.5. Implantation des graphes orientés constructifs

Pour produire un code extrait efficace, il est nécessaire de fournir une structure de données purement fonctionnelle efficace et adaptée pour représenter le type des graphes. Cette implantation permet également d'établir la cohérence de la spécification décrite dans le (type de) module CUG, constituée de 52 fonctions et propriétés.

Nous avons choisi de représenter les graphes par des listes d'adjacence car cette représentation permet un accès rapide aux voisins d'un sommet et les fonctions de construction ne demandent que quelques mises à jour locales à chaque appel.

Un graphe orienté est défini comme une unique table (appelée dans la suite *mapg*) de type *maptype* : un sommet v est associé à son étiquette, si elle existe, ainsi qu'à deux autres tables qui recensent respectivement les successeurs et prédécesseurs de v.

```
Definition maptype :=
VertexMap.t (option VLabel * (VertexMap.t (option ELabel) * (VertexMap.t (option ELabel)))).
```

La bibliothèque standard de Coq fournit une implantation des tables à l'aide d'arbres AVL [12]. L'implantation des graphes proposée repose sur cette implantation et utilise ses preuves de correction des opérations d'insertion, d'appartenance, etc. La structure de données choisie est illustrée figure 2 avec un graphe orienté où sommets et arcs sont étiquetés par des entiers.

Afin qu'une telle structure de données représente effectivement un graphe orienté, il faut assurer une propriété de cohérence entre successeurs et prédécesseurs. En effet, un sommet y est un successeur du sommet x avec l'étiquette l si et seulement si x est un prédécesseur de y avec la même étiquette. La définition de cette propriété demande deux prédicats $is\_labeled\_succ\ y\ x\ m\ l$ et $is\_labeled\_pred\ y\ x\ m\ l$ qui spécifient que y est un successeur ou une prédécesseur de x avec l'étiquette l dans la *mapg* m.

Afin de définir ces 2 prédicats, nous commençons par introduire la fonction *adj_map*, qui permet de voir une *mapg* comme une fonction totale :

(ADJM1) $(v \ \mapsto\ (l, s, p)) \in m \Rightarrow adj\_map\ v\ m = (Some\ l, s, p)$

(ADJM2) $v \notin m \Rightarrow adj\_map\ v\ m = (None, \emptyset, \emptyset)$

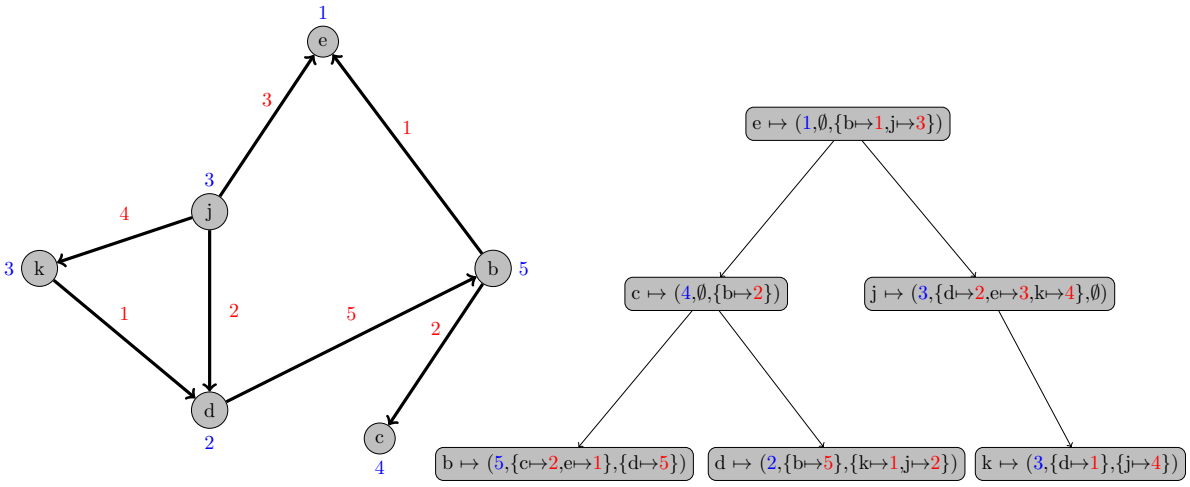

FIGURE 2 – Un graphe orienté et sa représentation. Pour une meilleure lisibilité, à chaque nœud, les tables des successeurs et prédécesseurs sont représentées comme des ensembles alors que ce sont, elles aussi, des arbres AVL

On peut alors définir les fonctions totales *succ* et *pred* qui calculent resp. l'ensemble (représenté par une table) des successeurs et prédécesseurs d'un sommet. Elles sont les deuxième et troisième projections de *adj_map*. Les définitions des prédicats *is_labeled_succ* et *is_labeled_pred* utilisent *succ* et *pred* :

`Definition` $is\_labeled\_succ\ y\ x\ m\ w$ `:=` $(y \mapsto w) \in (succ\ x\ m)$.

`Definition` $is\_labeled\_pred\ y\ x\ m\ w$ `:=` $(y \mapsto w) \in (pred\ x\ m)$.

Par conséquent, le type d'un graphe orienté dans cette implantation est défini par :

`Record` $graph : Type$ `:=` $Make\_Graph\{$
$mapg : maptype;$
$pred\_succ : \forall\ x\ y\ w, is\_labeled\_succ\ y\ x\ mapg\ w \Leftrightarrow is\_labeled\_pred\ x\ y\ mapg\ w$
$\}$.

Nous pouvons maintenant définir les fonctions de manipulation spécifiées dans l'interface et prouver qu'elles satisfont leurs spécifications. Nous renvoyons le lecteur au code Coq pour plus de détails sur l'implantation. Ce code compte environ 2000 lignes : 400 lignes de définitions et de spécifications, 1600 lignes de preuve.

Nous ne présentons pas ici l'implantation des graphes non orientés. La structure de données qui représente un tel graphe est très similaire à ce qui précède. Néanmoins seule la table des successeurs est présente puisque dans ce cas, successeurs et prédécesseurs sont identiques. L'implantation des graphes non orientés est d'une taille similaire, les preuves sont très similaires à celles pour les graphes orientés.

3. Couplage maximum

Dans cette section, nous formalisons en Coq les notions de couplage et de couverture d'arêtes conformément aux définitions classiques de la théorie des graphes [3].

Le code décrit dans cette section et la suivante (disponible à l'adresse `http://www.ensiie.fr/~dubois/Verif_couplage`) est général et concerne n'importe quel graphe non orienté étiqueté ou non.

Par conséquent nous introduisons ici un foncteur *Matching* dont les paramètres sont *Vertex* qui définit le type des sommets du graphe, *Label* qui définit le type des étiquettes des arêtes et G le module qui définit le type des graphes non orientés et construit à l'aide des modules *Vertex* et *Label*. Dans la suite, *Vertex.t* et *G.t* désignent respectivement le type des sommets et le type des graphes.

Par définition, un couplage d'un graphe g est un sous-ensemble d'arêtes de g disjointes deux à deux. La figure 3 donne deux exemples de couplage pour un même graphe.

Nous définissons ci-dessous en Coq les prédicats nécessaires à la formalisation d'un couplage. Ainsi, *disjoint_edges* e_1 e_2 spécifie que les arêtes e_1 et e_2 sont disjointes, c'est-à-dire qu'elles n'ont aucun sommet en commun. Le prédicat *disjoint_edges_list* permet de vérifier que toutes les arêtes d'une liste sont disjointes deux à deux.

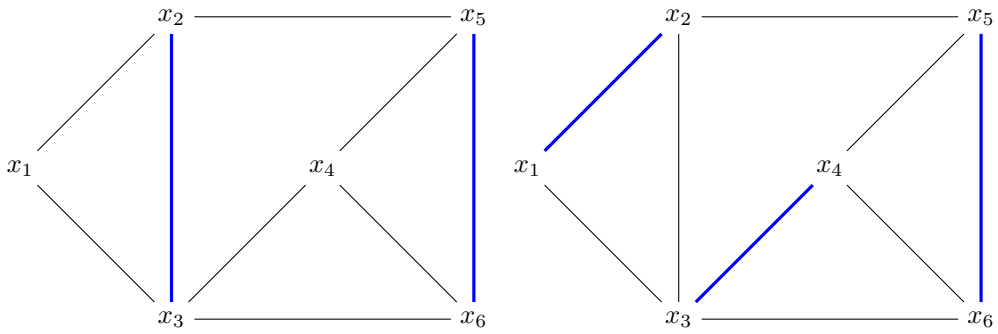

FIGURE 3 – En bleu, deux couplages du même graphe.

```
Definition disjoint_edges (e1 e2 : G.Edge.t) : Prop :=
(fst e1) ≠ (fst e2) ∧ (fst e1) ≠ (snd e2) ∧ (snd e1) ≠ (fst e2) ∧ (snd e1) ≠ (snd e2).
Definition disjoint_edges_list (l : list G.Edge.t) : Prop :=
∀ e1 e2, e1 ∈ l ⇒ e2 ∈ l ⇒ e1 ≠ e2 ⇒ disjoint_edges e1 e2.
```

Enfin *matching* m, si m est une liste d'arêtes, est satisfait si et seulement si m est une liste sans doublon d'arêtes de g, toutes disjointes deux à deux. La définition ci-dessous utilise le prédicat *noDup* qui vérifie qu'une liste est sans doublon.

```
Definition matching (m : list G.Edge.t)(g : G.t) : Prop :=
(noDup m) ∧ (m ⊆ (G.E g)) ∧ (disjoint_edges_list m).
```

Un couplage est dit maximum s'il est de cardinalité maximale. Le couplage représenté à droite dans la figure 3 est maximun.

```
Definition matchingMax (m : list G.Edge.t)(g : G.t) : Prop :=
matching m g ⇒ (∀ m1, matching m1 g ⇒ length m1 ≤ length m).
```

Le vérificateur proposé dans la section suivante s'appuie sur la notion de couverture des arêtes d'un graphe, notion duale à celle de couplage. Ainsi, k est une couverture des arêtes de g si et seulement si k est un sous-ensemble de sommets de g tel que chaque arête du graphe a une extrémité dans k. Par exemple, l'ensemble de sommets $\{x_2, x_3, x_4, x_5\}$ est une couverture d'arêtes du graphe de la figure 3. Le prédicat *cover* ci-dessous décrit formellement cette notion :

```
Definition cover (k : list Vertex.t)(g : G.t) : Prop :=
k ⊆ (G.V g) ∧ (∀ e, e ∈→ g ⇒ (fst e) ∈ k ∨ (snd e) ∈ k).
```

La propriété suivante, qui lie couplage et couverture, justifie que l'on peut choisir une couverture d'arêtes comme certificat d'un couplage maximum. En effet, si m est un couplage de g et k une

couverture d'arêtes de g, alors chaque arête de m a au moins une de ses extrémités dans k. Comme les arêtes de m n'ont pas d'extrémités communes, le cardinal de m est inférieur ou égal au cardinal de k (théorème *matching_cover* ci dessous). Et donc la cardinalité maximale d'un couplage est inférieure ou égale à la cardinalité minimale d'une couverture d'arêtes. Ainsi, si l'on a égalité, c'est que le couplage est maximum (théorème *matchingMax_cover*).

`Lemma` $matching\_cover\ :\ \forall\ g\ m\ k,\ matching\ m\ g \Rightarrow cover\ k\ g \Rightarrow length\ m \leq length\ k.$

`Lemma` $matchingMax\_cover\ :$
$\forall\ g\ m\ k,\ matching\ m\ g \Rightarrow cover\ k\ g \Rightarrow length\ m = length\ k \Rightarrow matchingMax\ m\ g.$

Ce dernier théorème est un corollaire du précédent, sa démonstration est quasi-immédiate en utilisant les définitions des prédicats et le lemme *matching_cover*.

4. Vérificateur formellement vérifié pour le calcul d'un couplage maximum

4.1. Définition du vérificateur

Le vérificateur est une fonction $bMatchingMax$ qui prend en argument une liste d'arêtes m, une liste de sommets k, un graphe g et renvoie *true* si (1) m est un couplage de g, (2) k est une couverture d'arêtes de g et si (3) m et k sont deux listes de même longueur.

Le vérificateur se décompose principalement en deux fonctions booléennes $bMatching$ et $bCover$ qui vérifient respectivement les points (1) et (2) précédents.

La fonction $bMatching$, dont le code Coq est donné ci-dessous, prend un couplage et un graphe en entrée, et teste si chaque arête du couplage est dans le graphe et si la liste de tous les sommets de toutes les arêtes du couplage (obtenue à l'aide de la fonction $flatten$) est sans doublon. La fonction $bNoDup$ est la fonction booléenne correspondante au prédicat *noDup*. La fonction *bInclEGraph* teste si les arêtes d'un couplage sont bien dans un graphe donné. Cette fonction se définit à l'aide d'un itérateur sur les listes et du lemme de décidabilité de l'appartenance d'une arête à un graphe.

`Definition` $bMatching\ m\ g : bool\ :=$
$(bInclEGraph\ m\ g)$ `&&` $bNoDup\ (flatten\ m).$

La fonction $bCover$ prend en entrée une couverture d'arêtes et un graphe, et teste si les sommets de la couverture sont des sommets du graphe et si toutes les arêtes du graphe ont au moins une extrémité dans la couverture. La première vérification utilise la fonction *bInclVGraph* dont la définition est très similaire à celle de *bInclEGraph*. La seconde vérification est réalisée à l'aide de l'itérateur sur les sommets d'un graphe fourni par la bibliothèque de graphes.

`Definition` $bCover\ k\ g : bool\ :=$
$(bInclVGraph\ k\ g)$ `&&` $G.fold_{\rightarrow}\ (fun\ g\ e\ l\ b \Rightarrow b$ `&&` $(((fst\ e) \in k)$ `||` $((snd\ e) \in k))\ g\ true.$

Il ne reste plus qu'à écrire le vérificateur complet (dans lequel *beq_nat* teste l'égalité de deux entiers naturels) :

`Definition` $bMatchingMax\ m\ k\ g : bool\ :=$
$(bMatching\ m\ g)$ `&&` $(bCover\ k\ g)$ `&&` $beq\_nat\ (length\ m)\ (length\ k).$

4.2. Preuve de correction du vérificateur

La preuve de correction du vérificateur consiste à montrer que si ce dernier retourne le résultat *true* alors le couplage fourni est bien un couplage maximum du graphe considéré. Si la réponse est

$false$, nous ne pouvons rien affirmer. Si le certificat, ici, la couverture d'arêtes, n'est pas minimun alors le vérificateur répondra $false$ même si le couplage est bien maximum. Il y a donc possibilité de fausse alarme.

La preuve de correction de $bMatchingMax$ s'appuie sur les preuves de correction de $bCover$ et $bMatching$ et sur le théorème $matchingMax\_cover$ mentionné précédemment.

Nous prouvons l'équivalence entre le prédicat qui définit la notion de couverture d'arêtes $cover$ et la fonction $bCover$:

`Lemma` $bCover\_correct\ :\ \forall\ k\ g, cover\ k\ g \Leftrightarrow bCover\ k\ g = true.$

La preuve utilise les définitions de $cover$ et $bCover$ ainsi que les propriétés de l'itérateur sur les sommets d'un graphe.

Le lemme de correction de $bMatching$ est énoncé ci-dessous. Sa preuve utilise de nombreux lemmes techniques concernant la fonction $flatten$ et les listes sans doublon..

`Lemma` $bMatching\_soundness\ :\ \forall\ m\ g,\ bMatching\ m\ g = true \Rightarrow matching\ m\ g.$

La réciproque requiert une hypothèse supplémentaire qui impose que le graphe soit simple, c'est-à-dire sans boucle (les deux extrémités de toute arête sont différentes). Mais cette propriété n'est pas nécessaire pour la preuve de correction du vérificateur :

`Definition` $simple\_graph\ (g : G.t) : Prop$ `:=` $\forall\ e,\ e \in_{\rightarrow} g \Rightarrow (fst\ e) \neq (snd\ e).$

`Lemma` $bMatching\_correctness$:
$\forall\ m\ g,\ simple\_graph\ g \Rightarrow (bMatching\ m\ g = true \Leftrightarrow matching\ m\ g).$

Enfin, le théorème de correction du vérificateur est le suivant :

`Lemma` $bMatchingMax\_soundness$:
$\forall\ m\ k\ g,\ bMatchingMax\ m\ k\ g = true \Rightarrow matchingMax\ m\ g.$

Notons que la réciproque n'est vraie que si le graphe g est biparti, c'est-à-dire qu'il existe une partition de l'ensemble des sommets en deux sous-ensembles U et V tels que toute arête du graphe a une extrémité dans U et l'autre dans V.

5. Extraction et expérimentations

Nous avons utilisé le vérificateur décrit précédemment pour certifier les résultats d'une fonction $MaxMatching$ qui calcule, pour un graphe donné, un couplage maximum et qui produit un certificat qui est une couverture d'arêtes de cardinal minimal, pour ce même graphe. La couverture d'arêtes se calcule aisément et ne produit pas de surcoût.

Cette fonction apparaît dans l'algorithme de filtrage [21] d'une contrainte de différence ALLDIFF$(x_1, x_2 \ldots x_n)$ pour un solveur de contraintes à domaines finis. Notre objectif à plus long terme est de vérifier formellement l'algorithme de filtrage d'une telle contrainte afin d'améliorer le solveur vérifié formellement existant développé par Carlier, Dubois et Gotlieb [6].

Une solution de la contrainte ALLDIFF$(x_1, x_2 \ldots x_n)$, quand chaque variable x_i a un ensemble fini de valeurs $D(x_i)$, est un couplage maximum d'un graphe biparti construit à partir des variables et des valeurs des domaines, appelé graphe de valeurs. Ce graphe relie chaque variable à chacune des valeurs de son domaine. La figure 4 contient un exemple d'un tel graphe. L'algorithme de filtrage, qui supprime des domaines les valeurs inconsistantes, i.e. celles qui ne feront partie d'aucune solution, calcule en particulier un couplage de cardinal maximal pour le graphe de valeurs. Il trouve d'abord un premier couplage, puis le rend maximum en recherchant des chaînes augmentantes [3]. De telles chaînes sont associées à un couplage donné et permettent, quand elles existent, d'en augmenter le cardinal. Cet algorithme est complexe et sa preuve de correction nécessiterait de nombreuses propriétés sur

variables : x_1, x_2, x_3
$D(x_1) = D(x_2) = \{1, 2\}$
$D(x_3) = \{2, 3\}$

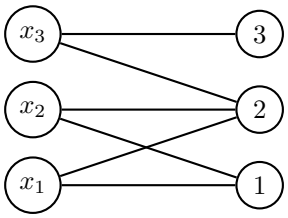

FIGURE 4 – Graphe de valeurs associé à la contrainte ALLDIFF(x_1, x_2, x_3) avec les domaines D.

les graphes bipartis. Nous proposons donc, dans un premier temps, une approche dite de vérification *a posteriori* qui consiste à vérifier le résultat fourni à l'aide d'un certificat. Nous proposons d'utiliser le vérificateur décrit dans la section 4 et d'en extraire un programme écrit en OCaml [16]. Pour cela, nous avons instancié le foncteur décrit dans les sections 3 et 4 en utilisant l'implantation des graphes non orientés fournie par la bibliothèque de graphes décrite dans cet article. Les sommets sont ici de deux natures : des variables et des valeurs entières. Il n'y a pas d'étiquette.

Nous avons mené deux expériences successives. La première a été d'écrire une version purement fonctionnelle en OCaml pour cet algorithme de calcul d'un couplage maximum en lui fournissant en argument le graphe de valeurs de la contrainte ALLDIFF. La fonction OCaml manipule une représentation des graphes *ad hoc*, il faut donc, pour appliquer le vérificateur, transformer le graphe utilisé par la fonction OCaml dans le format requis par le vérificateur. Il y a là risque d'erreurs. La deuxième expérience nous a amenés à écrire la fonction $MaxMatching$ directement en Coq en utilisant la représentation des graphes et les fonctions de manipulation issues de la bibliothèque de graphes. Par extraction, on obtient une fonction OCaml qui travaille sur la même structure de données que son vérificateur. La transcription de la fonction en Coq n'a pas posé de problème particulier, le code OCaml initial ayant été écrit le plus fonctionnellement possible. Une des fonctions intermédiaires présente une récursion non structurelle, nous l'avons réécrite de manière à ce qu'elle mette en œuvre une récursion structurelle, en ajoutant un argument supplémentaire qui compte les étapes de calcul et qu'il faudra choisir suffisamment grand pour obtenir un résultat. Cette astuce est usuelle mais certes, non satisfaisante. Une alternative consiste à écrire cette fonction en utilisant la construction `Function` mais ceci requiert de prouver explicitement la terminaison de cette fonction intermédiaire, travail à l'étude.

6. Travaux connexes

Dans cette section, nous présentons quelques travaux existants sur la formalisation des graphes à l'aide d'assistants à la preuve. En 1994, Chou a formalisé en HOL quelques notions classiques de la théorie des graphes [8], par exemple les graphes, les graphes acycliques, les arbres. La formalisation des graphes planaires [22] ainsi que la formalisation d'algorithmes de recherche [23] ont suivi les travaux de Chou. En 2001, Duprat a formalisé les mêmes notions en Coq, en utilisant des définitions inductives (`http://coq.inria.fr/V8.2pl1/contribs/GraphBasics.html`). Mizar compte de nombreux développements autour de la théorie des graphes, souvent indépendants les uns des autres. On peut citer par exemple la formalisation des graphes cordaux [15], graphes tels que chacun des cycles de quatre sommets ou plus possède une corde, c'est-à-dire une arête reliant deux sommets non-adjacents du cycle. Cependant tous ces travaux ne visent pas à produire du code. Les graphes y sont considérés comme une structure mathématique à étudier. Notre objectif dans le développement de la bibliothèque de graphes était de poser les bases d'une formalisation permettant non seulement de spécifier et prouver des propriétés sur les graphes mais aussi de pouvoir exprimer et vérifier des algorithmes de graphes et d'une extraire du code fonctionnel, opérationnel et raisonnablement efficace.

Bauer et Nipkow ont formalisé les graphes planaires non orientés et ont prouvé le théorème des 5 couleurs en Isabelle/HOL [2]. En 2011, Nipkow a proposé une énumération efficace et vérifiée formellement des graphes planaires [19]. Gonthier et Werner ont fourni la première preuve complètement mécanisée, en Coq, du théorème des 4 couleurs en utilisant une formalisation des hypergraphes qui généralisent les graphes [14]. Dans le domaine de la modélisation géométrique, des formalisations des cartes et des hypercartes, graphes particuliers, ont été proposées [10, 9]. Contrairement à notre proposition, les travaux mentionnés dans ce paragraphe s'intéressent à des graphes particuliers.

Quelques travaux concernent également la vérification d'algorithmes comme par exemple la recherche du plus court chemin dans un graphe, sans pour autant chercher à fournir une bibliothèque de graphes raisonnablement complète. En 1998, Paulin et Filliâtre ont prouvé l'algorithme de Floyd en utilisant Coq et Caduceus [13]. L'algorithme écrit dans un style impératif utilise une représentation matricielle des graphes. On peut aussi citer des formalisations de l'algorithme du plus court chemin de Dijkstra avec Mizar [7] et ACL2 [18].

Très récemment, Noschinski a développé une bibliothèque de graphes en Isabelle/HOL [20] qui comprend la formalisation de graphes orientés avec des arcs étiquetés et multiples. Elle propose plusieurs représentations des graphes. Cette bibliothèque a été utilisée pour formaliser et vérifier des algorithmes de vérification proposés par la bibliothèque LEDA [17], en particulier un algorithme de vérification pour un couplage de cardinal maximal [1]. Le certificat utilisé par cet algorithme de vérification (*odd-set cover* en anglais) peut être vu comme une généralisation de la notion de couverture des arêtes par des sommets. La preuve de correction du vérificateur s'appuie sur le théorème d'Edmonds [11].

7. Conclusion

Dans cet article, nous avons présenté un noyau de bibliothèque de graphes pour l'assistant à la preuve Coq. Cette dernière contient la spécification des graphes orientés et non orientés, dont les sommets et arêtes sont éventuellement étiquetés. Elle contient également une implantation de ces graphes qui utilise fortement les tables finies de la bibliothèque standard. Elle permet d'exprimer et prouver des propriétés sur les graphes mais aussi d'implanter et de vérifier des algorithmes de graphe dont on peut, grâce au mécanisme d'extraction de Coq, extraire du code fonctionnel OCaml par exemple.

Nous avons utilisé cette bibliothèque pour obtenir un vérificateur formellement vérifié qui permet de vérifier qu'un couplage est maximum. Il a été utilisé pour vérifier les résultats d'une étape intermédiaire d'un algorithme de filtrage pour la contrainte de différence ALLDIFF.

Les perspectives de ce travail sont nombreuses et concernent tout aussi bien l'extension de la bibliothèque de graphes que la vérification de l'algorithme de filtrage. L'extension de la bibliothèque peut se faire en l'enrichissant de nouvelles notions (comme par exemple les graphes bipartis) et d'algorithmes classiques. D'autres implantations peuvent également être définies. La vérification de l'algorithme de filtrage, quant à elle, requiert de définir les notions de chaînes augmentantes et d'exprimer et de prouver de nombreux résultats de la théorie des graphes, comme le théorème de Berge (1957) qui établit qu'un couplage est maximum si et seulement s'il n'existe pas de chaîne augmentante.

Remerciements : les auteurs remercient Cédric Bentz et Arnaud Gotlieb pour les discussions fructueuses. Ces travaux ont en partie été financés par le laboratoire Cedric du CNAM et le projet Aurora CertiSkatt.

Références

[1] Eyad Alkassar, Sascha Böhme, Kurt Mehlhorn, and Christine Rizkallah. A framework for the verification of certifying computations. *Journal of Automated Reasoning*, 52(3) :241–273, 2014.

[2] Gertrud Bauer and Tobias Nipkow. The 5 colour theorem in Isabelle/Isar. In *TPHOLs*, volume 2410 of *LNCS*, pages 67–82, 2002.

[3] Claude Berge. *Graphes et hypergraphes*. Dunod, 1973.

[4] Sandrine Blazy, Benoît Robillard, and Andrew W. Appel. Formal verification of coalescing graph-coloring register allocation. In Andrew D. Gordon, editor, *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6012 of *Lecture Notes in Computer Science*, pages 145–164. Springer, 2010.

[5] Sandrine Blazy, Benoît Robillard, and Eric Soutif. Vérification formelle d'un algorithme d'allocation de registres par coloration de graphe. In *JFLA'08 Journées Francophones des Langages Applicatifs*, pages 31–46, January 2008.

[6] Matthieu Carlier, Catherine Dubois, and Arnaud Gotlieb. A certified constraint solver over finite domains. In *Formal Methods (FM)*, volume 7436 of *LNCS*, pages 116–131, Paris, 2012.

[7] Jing-Chao Chen. Dijkstra's shortest path algorithm. *Journal of Formalized Mathematics*, 15, 2003.

[8] Ching-Tsun Chou. A formal theory of undirected graphs in higher-order logic. In *Workshop on Higher Order Logic Theorem Proving and Its Applications*, 1994.

[9] Christophe Dehlinger and Jean-François Dufourd. Formalizing generalized maps in coq. *Theor. Comput. Sci.*, 323(1-3) :351–397, 2004.

[10] Jean-François Dufourd. An intuitionistic proof of a discrete form of the jordan curve theorem formalized in coq with combinatorial hypermaps. *J. Autom. Reasoning*, 43(1) :19–51, 2009.

[11] Jack Edmonds. Maximum matching and a polyhedron with 0, 1-vertices. *Journal of Research of the National Bureau of Standards B*, 69 :125–130, 1965.

[12] Jean-Christophe Filliâtre and Pierre Letouzey. Functors for proofs and programs. In *ESOP*, pages 370–384, 2004.

[13] Jean-Christophe Filliâtre and Claude Marché. The why/krakatoa/caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, 2007.

[14] Georges Gonthier. Formal proof – the four-color theorem. *Notices of the American Mathematical Society*, 55(11) :1382–1393, December 2008.

[15] Krzysztof Hryniewiecki. Graphs. *Journal of Formalized Mathematics*, 2, 1990.

[16] Pierre Letouzey. Coq Extraction, an Overview. In C. Dimitracopoulos A. Beckmann and B. Löwe, editors, *Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE 2008*, volume 5028 of *Lecture Notes in Computer Science*. Springer-Verlag, 2008.

[17] Kurt Mehlhorn, Stefan Näher, and Christian Uhrig. The leda platform for combinatorial and geometric computing. In Pierpaolo Degano, Roberto Gorrieri, and Alberto Marchetti-Spaccamela, editors, *Automata, Languages and Programming*, volume 1256 of *Lecture Notes in Computer Science*, pages 7–16. Springer Berlin Heidelberg, 1997.

[18] J. Strother Moore and Qiang Zhang. Proof pearl : Dijkstra's shortest path algorithm verified with ACL2. In *TPHOLs*, volume 3603 of *LNCS*, pages 373–384, 2005.

[19] Tobias Nipkow. Verified efficient enumeration of plane graphs modulo isomorphism. In van Ekelen, Geuvers, Schmaltz, and Wiedijk, editors, *Interactive Theorem Proving (ITP 2011)*, volume 6898 of *LNCS*, pages 281–296, 2011.

[20] Lars Noschinski. A graph library for isabelle. *Mathematics in Computer Science*, 2014.

[21] Jean-Charles Régin. A filtering algorithm for constraints of difference in csps. In *12th National Conference on Artificial Intelligence (AAAI'94)*, pages 362–367, 1994.

[22] Mitsuharu Yamamoto, Shin-ya Nishizaki, Masami Hagiya, and Yozo Toda. Formalization of planar graphs. In *Workshop on Higher Order Logic Theorem Proving and Its Applications*, pages 369–384, 1995.

[23] Mitsuharu Yamamoto, Koichi Takahashi, Masami Hagiya, Shin-ya Nishizaki, and Tetsuo Tamai. Formalization of graph search algorithms and its applications. In *TPHOLs*, LNCS, pages 479–496, 1998.