Master Informatique

“Preuve mécanisée de l’algorithme de Tarjan”

Rapport
en vue de la validation de
l’UE Initiation à la recherche

2019-2020

Étudiants : Angela Ipseiz
Maxime Nicolas
Matthieu Olejniczak

Encadrant : Monsieur Merz

Décharge de responsabilités

L’Université de Lorraine n’entend donner ni approbation ni improbation aux opinions émises dans ce rapport, ces opinions devant être considérées comme propres à leur auteur.

Remerciements

Nous tenons à remercier toutes les personnes qui ont contribué au bon déroulement de notre travail et qui nous ont aidé lors de la rédaction de ce rapport.

Tout d'abord, nous adressons nos remerciements à notre encadrant, Monsieur Stephan Merz, enseignant chercheur au LORIA, pour son accueil, le temps passé ensemble, son écoute, ses conseils et ses interventions qui nous ont permis de progresser dans notre démarche et notre compréhension sur les preuves mécaniques. Il fut d'une aide précieuse dans les moments les plus délicats.

Enfin, nous tenons à remercier toutes les personnes qui nous ont conseillé et relu lors de la rédaction de ce rapport : nos familles, nos amis et surtout notre professeur de communication Madame Marie-Laure Alves.

Sommaire

Introduction

« Nous dépendons de plus en plus des logiciels, et il est important qu'ils fonctionnent de mieux en mieux. »[1] déclare Leslie Lamport, pionnier de l'algorithme distribué et du concept de preuve algorithmique. En effet, les logiciels sont de plus en plus présents dans notre quotidien et pour certains, notre sécurité en dépend, comme les logiciels utilisés dans l'aviation ou le ferroviaire. De nos jours, un programme incertain, c'est-à-dire avec au moins une condition d'utilisation qui rend le résultat incorrect, peut engendrer d'énormes conséquences financières, humaines.... C'est pourquoi, il est impératif de réaliser des **preuves d'algorithmes**, notamment pour ceux qui sont liés directement à la sécurité, comme celles qui ont été faites pour l'automatisation de la ligne de métro 14 à Paris, ou le respect de la vie privée que doivent garantir les entreprises du Cloud.

Différents types d'algorithmes peuvent nécessiter une preuve. En effet, il existe des algorithmes impliquant des **graphes** qui sont au coeur de nombreux problèmes actuels. Par exemple, la résolution d'un sudoku peut être transformée en problème 2-SAT qui détermine s'il existe une solution possible grâce à des graphes. Il est des graphes, des structures de données représentées par un ensemble de sommets et d'arêtes. Un graphe est **orienté** si ses arêtes ne sont parcourables que dans un seul sens dont la direction est représentée par une flèche. Le problème 2-SAT utilise la décomposition en **composantes fortement connexes**. On appelle composantes fortement connexes un sous-graphe maximal G' d'un graphe orienté G tel que, pour tout couple (u,v) de noeuds de G', il existe un chemin de u à v. Ainsi, un graphe fortement connexe est un graphe formé d'une seule composante fortement connexe.

Pour les identifier, on peut utiliser différents algorithmes tels que l'algorithme de Kosaraju et l'**algorithme de Tarjan**, tous deux fondés sur un algorithme de parcours en profondeur en temps linéaire. La principale différence entre ces deux algorithmes est que celui de Tarjan identifie les composantes fortement connexes en parcourant une seule fois le graphe au lieu de deux pour celui de Kosaraju. Nous voulons ici nous intéresser à l'algorithme de Tarjan : Est-ce que l'algorithme de Tarjan nous donne l'ensemble des composantes fortement connexes quelque soit le graphe orienté sur lequel on l'applique ?

Pour répondre à cette problématique, nous regarderons dans un premier temps les outils que nous avons utilisés afin de réaliser cette preuve, ainsi que le bon fonctionnement de l'algorithme par le biais de l'outil TLA+. Ensuite, nous nous intéresserons à sa preuve. Enfin, nous étudierons les problèmes que nous avons rencontrés et comment nous les avons réglés.

I. Les outils utilisés

I.i. TLA+

TLA+[(14)] est un langage de spécification, basé sur les mathématiques, permettant l'analyse et la description d'algorithmes au moyen de formules logiques. Créé par Leslie Lamport, il est utilisé dans l'environnement de développement toolbox TLA+. Cet environnement a pour but, à partir d'outils décrits dans les parties ci-dessous, de pouvoir connaître la correction des algorithmes et ainsi d'en faire leurs preuve.
Ce langage est la combinaison entre la théorie des ensembles - base des mathématiques classiques, servant à décrire les structures de données utilisées par un algorithme - et la logique temporelle, pour décrire les exécutions de l'algorithme. À ces bases classiques s'ajoute un langage simple de modules pour structurer une spécification. TLA+ ne possède pas de typage des variables mais fonctionne sur le principe de valeur de la variable aux différents états lors de l'exécution d'un algorithme . Si nous avons une variable i qui vaut 0 par exemple, dire que cette variable vaut 1 dans le prochain état revient à dire que i' vaut 1. C'est donc le caractère " ' " qui désigne la valeur de la variable qui la précède dans le prochain état.

Un algorithme, soit une suite d'instructions formelles permettant de répondre à un problème donné, en TLA+ est constitué d'un prédicat qui caractérise les états initiaux, habituellement appelé "Init", et un prédicat sur des couples d'états qui caractérise les transitions possibles d'un état à un autre, appelé "Next". Bien évidemment ce dernier peut faire appel à diverses autres opérateurs (tels que des fonctions, des ensembles, des relations...) définis précédemment dans le module TLA+. En général, une spécification en TLA+ est de la forme "Init /\ [][Next]_vars /\ L" où vars désigne la liste de toutes les variables utilisées dans la spécification et L est une formule temporelle qui exprime des contraintes supplémentaires sur les exécutions, typiquement des conditions d'équité pour demander que certaines actions possibles ne soient pas négligées. Quand il s'agit de démontrer la correction partielle d'un algorithme on n'a pas besoin de cette contrainte supplémentaire et on ne s'intéresse donc qu'à Init et Next.

I.ii. PlusCal

Exprimer un algorithme en TLA+ peut s'avérer compliqué, c'est pourquoi Lamport a créé le langage PlusCal. C'est un langage algorithmique similaire au pseudo-code. Les expressions de PlusCal sont celles de TLA+ et la sémantique de PlusCal est donnée par le traducteur d'algorithmes PlusCal en TLA+. Ce langage permet de transcrire des algorithmes pseudo-code en TLA+. PlusCal est souvent plus compréhensible notamment pour les débutants car celui-ci se rapproche plus d'un langage de programmation classique.
PlusCal et TLA+ sont donc souvent utilisés ensemble pour faire de la vérification de spécifications.

I.iii. Model checker

Alors que la spécification d'un programme contient généralement des paramètres, on appelle *modèle* une instanciation de ces paramètres par des ensembles concrets.

Le *model checker* est un outil intégré à la toolbox TLA+ permettant de montrer la correction de l'algorithme sur des exemples en utilisant des modèles. La vérification de modèle repose sur le calcul du graphe des états accessibles à partir d'un modèle donné, c'est-à-dire qu'à partir des états initiaux de la spécifications, le model checker va créer un graphe contenant tous les états successeurs possibles à partir de l'état courant. Au final, le model checker a énuméré l'ensemble des composantes fortement connexes. C'est à l'aide de cela (et d'un modèle) que l'on peut savoir si un algorithme fonctionne pour un exemple concret.

Là où prouver formellement un programme peut être long et exiger beaucoup de ressources, la vérification de modèles est très souvent une alternative employée, car moins coûteuse. Cependant, elle ne suffit pas à montrer que l'algorithme est correct pour tous les cas car il existe une infinité d'exemples possibles et certains exemples demandent beaucoup de ressources et de temps de calcul pour que cela puisse être possible. Par contre, on peut gagner en confiance en utilisant du model checking avant d'entamer la preuve formelle d'un algorithme.

I.iv. TLA+ proof system

TLA+ proof system[(15)] est une extension de TLA+ permettant de faire des preuves formelles mécanisées d'algorithme. Il a été développé par un laboratoire de Microsoft Research et Inria Paris avec la collaboration d'une équipe de recherche du Loria. Il fait appel à quatre outils de preuve semi-automatiques, appelés aussi *provers* : smt, Zenon, Isabelle et PTL.

L'assistant à la preuve offre la possibilité de vérifier formellement, à l'aide de ces prouveurs, les clauses que l'on veut montrer en utilisant des définitions ou d'autres morceaux de preuves que l'on a ou que l'on veut démontrer. Une preuve peut être décomposée grâce à un langage hiérarchique de preuve qui sert à développer une preuve interactive. En effet , si un énoncé que nous cherchons à prouver ne fonctionne pas tout seul, nous pouvons décomposer celui-ci en un raisonnement plus détaillé qu'il faudra montrer et qui permettront d'arriver à montrer que le cas initial est juste.

Ces outils forment la pierre angulaire de notre sujet de recherche. Nous allons les utiliser pour démontrer le bon fonctionnement de l'algorithme de Tarjan.

II. Algorithme de Tarjan

II.i Graphe et composante connexe

Avant de travailler sur l'algorithme de Tarjan en lui-même, nous avons testé le langage TLA+ et certaines de ses limites en définissant une recherche de composantes fortement connexes, aussi appelées SCC (Strongly Connected Component).

Pour cela, nous avons dû définir certaines notions essentielles :

```
Edges ==
  LET E(n) == { <<n,m>> : m \in Succs[n] }
  IN  UNION {E(n) : n \in Nodes}
```

- *Edges* représente l'ensemble des arêtes.

```
Path(m,n) == \E p \in Seq(Nodes) \ {<< >>} :
  /\ p[1] = m
  /\ p[Len(p)] = n
  /\ \A i \in 1 .. Len(p)-1 : p[i+1] \in Succs[p[i]]
```

- *Path* indique si un chemin existe d'un sommet A à un sommet B.

```
Connected == { mn \in Nodes \X Nodes : Path(mn[1], mn[2]) }
```

- *Connected* utilise Path pour identifier toutes les connexions dans le graphe.

```
FullyConnected(S) ==
  \A x, y \in S : <<x,y>> \in Connected /\ <<y,x>> \in Connected
```

- *FullyConnected* reconnaît si, dans un ensemble de nœuds, ceux-ci sont tous connectés deux à deux. Ce qui signifie que pour tous nœuds x et y de l'ensemble, il existe un chemin de x à y et de y à x.

```
SCC(S) ==
  /\ FullyConnected(S)
  /\ \A T \in SUBSET Nodes : S \subseteq T /\ FullyConnected(T) => S = T
```

- *SCC* prévient si un ensemble de nœuds forme un graphe fortement connexe maximal.

```
SCCs == { S \in SUBSET Nodes : SCC(S) }
```

- *SCCs* affiche l'ensemble des composantes fortement connexes du graphe.

II.ii Description et fonctionnement de l'algorithme

Avant de décrire l'algorithme, il convient de s'intéresser aux variables que nous allons utiliser.

```
variables
  index = 0,
  t_stack = << >>,
  num = [n \in Nodes |-> -1],
  lowlink = [n \in Nodes |-> -1],
  onStack = [n \in Nodes |-> FALSE],
  sccs = {},
  toVisit = Nodes;
```

- **Index** est un compteur permettant d'attribuer un nouveau numéro à un sommet.
- **t_stack** est la pile sur laquelle on place les sommets en cours de visite.
- La variable **num** nous donne à la fois le numéro du noeud et son état. En effet, s'il n'a pas encore été exploré, elle vaut -1.
- **lowlink** représente le plus petit index du sommet atteignable par le noeud courant entre ses descendants et lui-même.
- **onStack**, comme son nom l'indique, permet de savoir si un sommet est déjà sur t_stack.
- **sccs** est l'ensemble des composantes fortement connexes retourné par l'algorithme.
- **toVisit** est l'ensemble des sommets qui n'ont pas encore eu de numéro, donc qui n'ont pas encore été visités.

La fonction **main** de l'algorithme sera assez simple :

```
main:
    while (toVisit # {}) {
      with (n \in toVisit) {
        toVisit := toVisit \ {n};
        if (num[n] = -1) { call visit(n) }
      }
    }
  }
```

Au début de l'algorithme, toVisit est composé de tous les sommets du graphe donné en entrée. La fonction main fait que, tant que toVisit n'est pas vide, on sélectionne un sommet n de toVisit qu'on retire de cet ensemble, puis on effectue la procédure visit() sur ce sommet.

Comme son nom l'indique, la procédure **visit(n)** effectue une visite sur le noeud donné en paramètre.
Chaque visite d'un sommet v commence par une initialisation des variables dans une suite d'instructions regroupée sous l'étiquette **start_visit.**

```
start_visit:
    num[v] := index;
    lowlink[v] := index;
    index := index+1;
    t_stack := <<v>> \o t_stack;
    onStack[v] := TRUE;
    succs := Succs[v];
```

La ligne t_stack := <<v>> \o t_stack veut dire que l'on place le nœud courant v au sommet de la pile t_stack.
Après avoir initialisé les variables, on cherche les successeurs.

```
explore_succ:
    while (succs # {}) {
      with (s \in succs) { w := s; succs := succs \ {s} };
      if (num[w] = -1) {
visit_recurse:
        call visit(w);
continue_visit:
        if (lowlink[w] < lowlink[v]) { lowlink[v] := lowlink[w] }
      } else if (onStack[w]) {
        if (num[w] < lowlink[v]) { lowlink[v] := num[w] }
      }
    };
```

```
check_root:
    if (lowlink[v] = num[v] /\ (\E k \in 1 .. Len(t_stack) : t_stack[k] = v)) {
      \* new SCC found: pop all nodes up to v from the (Tarjan) stack
      with (k = CHOOSE k \in 1 .. Len(t_stack) : t_stack[k] = v) {
        sccs := sccs \cup {{t_stack[i] : i \in 1 .. k}};
        onStack := [n \in Nodes |-> IF \E i \in 1 .. k : n = t_stack[i] THEN FALSE
                                    ELSE onStack[n]];
        t_stack := SubSeq(t_stack, k+1, Len(t_stack))
      }
    };
    return;
  }
```

Nous regardons donc le successeur et avons deux choix :

- S'il n'a pas encore été visité, alors on le visite. On fait donc un appel récursif à la procédure **visit_recurse**.
  Puis, une fois que l'appel récursif est terminé, on entre dans **continue_visit**.
  Si le plus petit sommet accessible par le noeud successeur est inférieur au plus petit sommet accessible par le noeud courant, alors le lowlink du noeud courant prend la valeur du lowlink du successeur.

- Si le noeud successeur a déjà été visité, donc s'il fait partie de la SCC actuelle, et que son numéro est inférieur au lowlink du noeud courant, alors le lowlink du noeud courant prend cette valeur.

Enfin, il ne nous reste plus qu'à traiter toutes les informations obtenues avec **check_root**.
Si le noeud courant est une racine (lowlink = num) et qu'il est déjà sur la pile, alors on a trouvé une composante fortement connexe. Il ne reste plus qu'à dépiler jusqu'à l'instance du noeud sur la pile, en ajoutant les sommets rencontrés dans l'ensemble des SCC.

On remarquera que dans check_root, pour l'expression de onStack on a :

On a décidé d'utiliser cette forme, soit \E i in 1….k au lieu de onStack[v] pour simplifier la preuve autant se faire que peu. En effet, l'expression CHOOSE, c'est avéré être une difficulté.

II.iii L'algorithme en TLA+

L'algorithme donné précédemment est la version Pluscal de l'algorithme de Tarjan. Nous avons utilisé la toolbox pour le transcrire en version TLA+. Le traducteur PlusCal va générer une action par étiquette dans le code et la disjonction de celles-ci correspond à la formule next du module TLA+.

Dans cette nouvelle version, on commence par la définition de l'opérateur Init, où l'on initialise nos variables. De plus, viennent principalement deux nouvelles variables : *stack* et *pc*.

Comme on peut le voir dans les images suivantes, pc nous indique quelle définition effectuer.

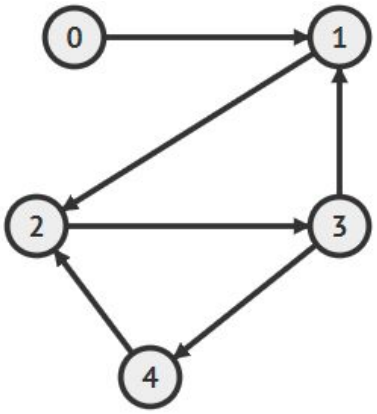
```
start_visit == /\ pc = "start_visit"
               /\ num' = [num EXCEPT ![v] = index]
```

```
explore_succ == /\ pc = "explore_succ"
                /\ IF succs # {}
```

Quant à la variable stack, il s'agit d'une pile utilisée pour effectuer des appels récursifs.

II.iv Vérification du modèle de l'algorithme

Nous avons testé l'algorithme sur un exemple en utilisant le Model checker pour vérifier ses limites. On a alors créé un modèle composé de quelques sommets et arêtes.



Graphe orienté construit avec VisuAlgo[16]

Il faut commencer par initialiser les variables Nodes et Succs.

- Nodes <- {0,1,2,3,4}
- Succs <- (0:>{1}) @@
  (1:>{2}) @@
  (2:>{3}) @@
  (3:>{1,4}) @@
  (4:>{2})

Le but de l'algorithme de Tarjan est de retourner les composantes fortement connexes d'un graphe. On utilise les définitions vues en II.iv pour tester le modèle.
En demandant les arêtes du graphe avec Edges, on obtient le résultat :

{<<0, 1>>, <<1, 2>>, <<2, 3>>, <<3, 1>>, <<3, 4>>, <<4, 2>>}

En recherchant toutes les connexions du graphe avec Connected, on obtient que tous les nœuds sont connectés les uns avec les autres, sauf avec 0 (qui est connecté avec lui-même).
Enfin, il est intéressant de voir si l'algorithme de Tarjan remplit bien son rôle. En utilisant la définition de SCCs, on obtient tous les SCC présents dans le graphe. Il en résulte l'ensemble :

{{0}, {1, 2, 3, 4}}

Le model checker nous donne aussi des informations sur le nombres d'états. Pour. cette exemple, plus de 600 états ont été générés, dont 500 différents. Pourtant, le graphe est relativement petit. Si on prend un graphe à 10 nœuds et 13 arêtes (Annexe 1), quasiment 190 000 états différents sont générés pour un total de plus de 400 000 états au total.

Désormais, nous avons pu nous faire une idée quant au fonctionnement de l'algorithme et à la validité des résultats qu'il renvoie. Cependant, cela ne montre pas que ceux-ci seront toujours bons. Voilà pourquoi, nous devons approfondir avec une preuve formelle de l'algorithme.

III. Preuve de l'algorithme de Tarjan

III.i Invariants

Un invariant de boucle est une propriété qui est vraie avant, pendant et après une boucle, soit à tout moment de l'algorithme, à condition que l'algorithme consiste en une seule boucle. Ici la structure est plus complexe, et on parle d'invariant d'algorithme – un prédicat qui est vrai tout au long de l'exécution. Les invariants peuvent être plutôt généraux, par exemple à tout état du déroulement de l'algorithme, un numéro de noeud est un entier, ou plus restreint, comme les numéros des noeuds qui valent -1 en début d'algorithme. Un invariant permet alors de montrer qu'une propriété est valide pour tout état. Plusieurs invariants ont alors été prouvés pour montrer le bon fonctionnement de l'algorithme.

Dans la preuve de l'algorithme de Tarjan, nous nous sommes inspirés du travail de l'équipe Veridis du Loria pour écrire les invariants.(7) En effet, cette équipe a déjà fait la preuve de cet algorithme mais à l'aide d'autres outils tels que Why3, Isabelle et Coq, qui sont également des assistants à la preuve. Nous nous servons donc des invariants qui ont été réalisés pour la preuve précédente afin de réaliser la preuve en TLA+. Bien évidemment, nous devons rechercher comment transformer ces invariants pour qu'ils soient adaptés au langage TLA+ et à l'algorithme correspondant, car il peut y avoir des différences d'implémentation. En effet Why3, Isabelle et Coq sont basés sur des représentations de l'algorithme sous la forme de fonctions récursives (OCaml par exemple) alors que la formulation en PlusCal est un programme impératif.

III.ii Preuve des invariants

Maintenant qu'on a montré l'utilité des invariants, il faut les prouver. On part d'invariants très généraux auxquels on ajoute des informations supplémentaires pour les affiner. L'objectif est de démontrer que la correction de l'algorithme découle de l'invariant. Il faut alors accumuler assez d'informations pour discuter l'état final. La séparation des invariants est intéressante, car dans un premier temps il est très complexe de trouver l'invariant intégral dès le début, mais aussi parce qu'on a rarement besoin de revenir sur les précédents une fois qu'ils ont été prouvés, ou du moins on essaie de modifier un invariant trop fréquemment. On peut alors se servir de ceux-ci une fois prouvés pour démontrer les suivants. Nous avons utilisé 4 invariants dans ce projet, bien qu'il en existe d'autres, qui sont :

- L'invariant de typage qui permet de montrer dans quels ensembles les variables vont prendre leurs valeurs tout au long de l'exécution du programme :

- L'invariant des sommets sur la pile
- L'invariant des couleurs
- L'invariant de liaison entre les piles

-> Expliquer ce que les invariants montrent

Pour faire la preuve d'un invariant, il faut prouver le théorème correspondant. Ce théorème dit que pour n'importe quel état que l'algorithme peut atteindre en partant des états initiaux et en suivant la relation de transition, toutes les clauses de l'invariant doivent être satisfaites, c'est-à-dire que chaque élément de l'invariant doit être correct.

Pour prouver ce théorème, il faut décomposer le théorème en deux sous-parties. Pour la première, il faut montrer que l'invariant est correct pour tout état vérifiant la condition initiale. La deuxième sous-partie consiste à prouver que l'invariant est correct dans l'état suivant en sachant qu'il est vrai dans l'état courant et en connaissant les états suivants possibles. Si nous sommes en train de prouver un n-ième invariant, nous avons donc n-1 invariants prouvés. Nous pouvons alors utiliser ces n-1 invariants pour nous aider à prouver le n-ième. Grâce à cela, nous pouvons commencer par décomposer un invariant qui peut être grand et difficile à montrer en plusieurs invariants plus simples pour ensuite les utilisés dans la preuve d'autres invariants.

III.iii Invariant et correction

Un algorithme est correcte s'il fait ce qu'on attend de lui. C'est à dire qu'il faut que son exécution termine, et que son état final nous donne le résultat que nous attendons.
Sachant que l'algorithme de Tarjan combine une procédure récursive et une boucle imbriquée sur les états successeurs, un invariant a donc le rôle idéal ici. En effet, un invariant montre qu'un ensemble de propriétés est vrai à n'importe quel état, et que si celui-ci est démontré, alors la propriété, de la correction, traitée est respecté.
Par définition de l'algorithme de Tarjan, l'algorithme finit quand la boucle principale finit. Ainsi, si on a un invariant témoignant de la validité des résultats obtenus en fin de boucle et que cet invariant est prouvé, alors l'algorithme de Tarjan est démontré.

III.iv Terminaison de l'algorithme

Par manque de temps, nous n'avons pu prouver la terminaison de l'algorithme.
Cependant, cela ne nous empêche pas d'avoir une idée sur comment la démontrer.
Sachant que l'algorithme s'arrête seulement lorsqu'il n'y a plus de sommet ayant pour numéro -1, et qu'il parcourt tous les sommets en leurs adressant un numéro dès qu'il les rencontre, on remarque que l'algorithme s'arrêtera donc une fois qu'il aura parcouru tous les sommets du graphe.

Ainsi la réponse à la question "L'algorithme finit-il ?" est assez intuitive. C'est pourquoi, on comprend tout de suite que l'essentiel et la difficulté de la preuve réside dans le

fait de montrer qu'à la fin de l'algorithme, nous obtenons bien le résultat attendu, c'est-à-dire l'ensemble des composantes fortement connexes du graphe.

IV. Problèmes et solutions apportées

IV. i Transcription de l'algorithme en TLA+

Notre premier projet après la compréhension des outils utilisés fut la transcription de l'algorithme de Tarjan en langage reconnu par TLA+. Cependant, les multiples boucles ont fait l'objet de différents problèmes, car le langage TLA+ ne permet pas de faire de la récursivité, il est donc impossible de traduire l'algorithme tel qu'il est écrit en langage naturel. Nous avons donc utilisé l'outil décrit précédemment, PlusCal, pour transcrire un algorithme initialement récursif en TLA+.

IV.ii Preuve de l'algorithme

Une première difficulté de la preuve provient des invariants. Nous nous sommes donc inspirés des invariants de l'équipe Veridis. Cependant, ceux-ci n'étaient pas complets. En effet, il arrive qu'une clause ne puisse pas être prouvée, et que même en la décomposant, rien n'y change. Cependant, rien ne nous fait savoir que le problème provient de l'invariant en lui-même. Et ce n'est pas toujours évident de le renforcer. De plus, une fois renforcée, il faut prouver à nouveau tout ce qui fait référence à l'invariant, car certaines clauses précédemment justes peuvent ne plus l'être avec ce changement.

- difficultés pour les parties reposants sur le fonctionnement de la pile

- énoncer un cas complexe de preuve ?

Conclusion

Il est facile de comprendre le fonctionnement superficiel d'un programme, mais il est difficile de comprendre son fonctionnement dans les moindre détails - comment il arrive à certain résultats - et c'est notamment cela qu'il faut savoir pour réussir à prouver qu'il fonctionnera dans tous les cas. Au mieux, on peut s'informer sur les résultats qu'il peut donner en utilisant des outils comme un model checker. Mais ceci ne suffit pas. Comme nous l'avons vu, ces outils ont leurs limites. Par exemple, pour l'algorithme de Tarjan, nous ne pouvons avoir un retour que sur des petits graphes. Ainsi, il est très compliqué de savoir s'il fonctionne et finit sur un graphe avec plus d'une centaine de nœuds et il est impossible de tester tous les graphes possibles car il en existe une infinité. Cependant, même si cela permet d'avoir une idée sur la correction du programme, encore reste-t-il à la prouver. Pour ce faire, nous avons vu que l'on peut utiliser des logiciels de preuve semi-automatique comme TLA+ proof system, un outil associé à TLA+. Bien qu'ils soient d'une grande aide pour avoir une idée de ce qu'il faut prouver, ils ne nous indiquent pas comment le faire.

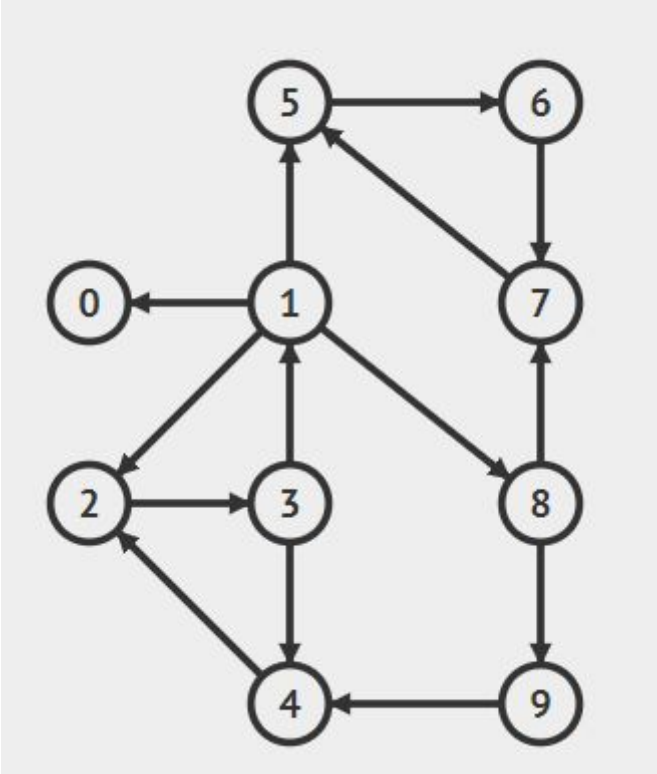
Malgré ces outils, la construction d'assertions intermédiaires, comme les invariants mais aussi la décomposition d'une étape de la preuve, continuent de se faire de manière empirique. Par exemple, lorsque l'on veut prouver un invariant, on peut se rendre compte assez vite que l'on est bloqué, qu'il nous manque quelque chose pour avancer et donc on se retrouve obligé de lui rajouter de nouvelles conditions, de le rendre plus strict. Évidemment, ce que l'on ajoute doit être à nouveau prouvé. C'est donc ici que l'on fait un pas en arrière pour pouvoir continuer à avancer. Et des pas en arrière, nous en avons fait quelques uns lors de la preuve de l'algorithme de Tarjan, mais qui nous ont permis de faire beaucoup de pas en avant dans notre preuve. Nous avons donc pu faire une partie de la preuve de l'algorithme de Tarjan.

L'algorithme de Tarjan n'est qu'un cas d'étude. Il existe des algorithmes et des programmes bien plus compliqués à prouver. Cependant, certaines méthodes, comme celle que nous avons utilisée, peuvent et doivent leurs être appliquées. En effet, si nous développons un algorithme qui va être utilisé pour gérer le trafic ferroviaire, il est nécessaire de s'assurer qu'il fonctionne dans n'importe quelle situation. Fort heureusement, le domaine de la preuve est un domaine émergent sur lequel des centres de recherches développent des outils de preuve plus puissant, c'est pourquoi nous pouvons espérer un jour savoir pertinemment ce que l'on fait.

Ce projet d'initiation à la recherche nous a donc permis de mieux comprendre le domaine de la preuve algorithmique, son importance et le besoin que nous en avons aujourd'hui. De plus, nous avons pu apprendre à faire une telle preuve sur un algorithme classique, et même si nous n'avons pas pû la faire entièrement, ceci fut une très bonne expérience.

Annexes

Annexe 1:



Graphe à 10 sommets et 13 arêtes

Bibliographie

(1) Le Monde, *Portrait de Leslie Lamport, un informaticien dans les nuages* : https://www.lemonde.fr/sciences/article/2014/06/16/leslie-lamport-un-informaticien-dans-les-nuages_4439171_1650684.html

(2) Définition détaillé de la *qualité logicielle* : https://fr.wikipedia.org/wiki/Qualit%C3%A9_logicielle

(3) Définition d'un *graphe orienté :* https://fr.wikipedia.org/wiki/Graphe_orient%C3%A9

(4) Définition précise de la *complexité* d'un programme : http://zanotti.univ-tln.fr/ALGO/I31/Complexite.html#def:successeur

(5)Documentation de *TLAPS* :

https://tla.msr-inria.inria.fr/tlaps/content/Documentation/Tutorial/The_example.html

(6)Leslie lamport, site web officiel de TLA+ :
https://lamport.azurewebsites.net/tla/tla.html

(7) Ran Chen, Cyril Cohen, Jean-Jacques Lévy, Stephan Merz, Laurent Théry. Formal Proofs of Tarjan's Strongly Connected Components Algorithm in Why3, Coq and Isabelle. 10th Intl. Conf. Interactive Theorem Proving (ITP). Leibniz Intl. Proc. in Informatics, 2019. http://drops.dagstuhl.de/opus/volltexte/2019/11068/

(8)François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond, Andrei Paskevich. The Why3 platform, version 0.86.1. Technical Report, LRI, CNRS, Univ. Paris Sud, Inria Saclay, May 2015. http://toccata.lri.fr/gallery/why3.en.html.

(9) The Coq Development Team. The Coq Proof Assistant v.8.3. Reference Manual. Inria, 2010. http://coq.inria.fr/

(10) TLA+ Proofs. Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, Hernán Vanzetto. 18th Intl. Symp. Formal Methods (FM). Springer LNCS 7436, pp. 147-154. Paris, France, 2012.

(11)Leslie Lamport. Specifying Systems. Addison Wesley (Boston, Mass.), 2002. http://lamport.azurewebsites.net/tla/tla.html .

(12)Tobias Nipkow, Lawrence Paulson, Markus Wenzel. Isabelle/HOL. A Proof Assistant for Higher-Order Logic. Springer LNCS 2283, 2002.

(13) Robert Tarjan. Depth first search and linear graph algorithms. SIAM Journal on Computing, 1972.

(14) A High-Level View of TLA+, Leslie Lamport, *2018*
https://lamport.azurewebsites.net/tla/high-level-view.html
(15) https://tla.msr-inria.inria.fr/tlaps/content/Home.html

(16) Visualiseur de graphes https://visualgo.net/en/dfsbfs

Déclaration sur l'honneur contre le plagiat

Je soussignée,

Ipseiz Angela

Régulièrement inscrit à l'Université de Lorraine

N° de carte d'étudiant : 31608993

Année universitaire : 2019-2020

Niveau d'études : M

Parcours : Informatique

N° UE : 811

Certifie qu'il s'agit d'un travail original et que toutes les sources utilisées ont été indiquées dans leur totalité. Je certifie, de surcroît, que je n'ai ni recopié ni utilisé des idées ou des formulations tirées d'un ouvrage, article ou mémoire, en version imprimée ou électronique, sans mentionner précisément leur origine et que les citations intégrales sont signalées entre guillemets.

Conformément à la loi, le non-respect de ces dispositions me rend passible de poursuites devant la commission disciplinaire et les tribunaux de la République Française.

Fait à Vandœuvre-lès-Nancy, le……..

Signature :

Déclaration sur l'honneur contre le plagiat

Je soussigné,

Nicolas Maxime

Régulièrement inscrit à l'Université de Lorraine

N° de carte d'étudiant : 31509056

Année universitaire : 2019-2020

Niveau d'études : M

Parcours : Informatique

N° UE : 811

Certifie qu'il s'agit d'un travail original et que toutes les sources utilisées ont été indiquées dans leur totalité. Je certifie, de surcroît, que je n'ai ni recopié ni utilisé des idées ou des formulations tirées d'un ouvrage, article ou mémoire, en version imprimée ou électronique, sans mentionner précisément leur origine et que les citations intégrales sont signalées entre guillemets.

Conformément à la loi, le non-respect de ces dispositions me rend passible de poursuites devant la commission disciplinaire et les tribunaux de la République Française.

Fait à Vandoeuvre-lès-Nancy, le XX avril 2020

Signature :

Déclaration sur l'honneur contre le plagiat

Je soussigné,

Olejniczak Matthieu

Régulièrement inscrit à l'Université de Lorraine

N° de carte d'étudiant : 31506421

Année universitaire : 2019-2020

Niveau d'études : M

Parcours : Informatique

N° UE : 811

Certifie qu'il s'agit d'un travail original et que toutes les sources utilisées ont été indiquées dans leur totalité. Je certifie, de surcroît, que je n'ai ni recopié ni utilisé des idées ou des formulations tirées d'un ouvrage, article ou mémoire, en version imprimée ou électronique, sans mentionner précisément leur origine et que les citations intégrales sont signalées entre guillemets.

Conformément à la loi, le non-respect de ces dispositions me rend passible de poursuites devant la commission disciplinaire et les tribunaux de la République Française.

Fait à Vandœuvre-lès-Nancy, le……..

Signature :

(Quatrième de couverture)

Depuis l'essor du numérique, les logiciels ont pris de plus en plus de place dans notre vie et notre quotidien. En plus de leur importance, le nombre de lignes de code qui les compose continue de grandir lui aussi, augmentant la possibilité d'avoir des erreurs. Plus un logiciel est employé à de grandes fins, plus une erreur aura de répercussions. Ainsi il devient important de vérifier qu'un logiciel termine et fonctionne correctement.
Pour effectuer cette vérification, plusieurs méthodes et outils ont été inventés tels que la vérification de modèle et la preuve formelle. Ce sont les deux procédés que nous présenterons et appliquerons, ici, à l'algorithme de Tarjan, notre cas d'étude.

Since the rise of digital technologies, softwares have become increasingly important in our everyday life. With a software's value increasing, the amount of lines of code, making it, is growing too, which further expands the risk of mistakes. The more far-reaching a software is, the greater the effect of errors will be. So it is important to ascertain that it ends and works correctly.
To do this, there are several methods and tools that have been invented like the property checking method and the Formal proof method. That's the two procedures we are introducing and using, here, at Tajan's algorithm, our case study.