Preuves de Programmes – projet

Nathanaël Courant

18 février 2018

Note : les versions des prouveurs utilisées sont AltErgo 1.30, Eprover 1.9.1-001, et Z3 4.5.0.

1 Définitions

Les définitions de `sort` et `topological_sort` sont des traductions directes de leur spécification dans le sujet.

```
predicate sort (g: graph) (tsort:tsort) =
  (* Result in interval [0;S.cardinal (vertices g)[ *)
  (∀ x: vertex. S.mem x (vertices g) → 0 ⩽ M.get tsort x < S.cardinal (vertices g))
  ∧
  (* Edges are well ordered *)
  (∀ x y: vertex. S.mem (x, y) (edges g) → M.get tsort x < M.get tsort y)
  ∧
  (* Injectivity *)
  (∀ x y: vertex. S.mem x (vertices g) ∧ S.mem y (vertices g) ∧ M.get tsort x = M.get
     tsort y → x = y)

predicate topological_sort (g: graph) (order: order) (tsort: tsort) =
  (* order and tsort are in bijection *)
  (* ... order (tsort v) = v *)
  (∀ x: vertex. S.mem x (vertices g) → M.get order (M.get tsort x) = x) ∧
  (* ... tsort (order v) = v *)
  (∀ x: int. 0 ⩽ x < S.cardinal (vertices g) → M.get tsort (M.get order x) = x) ∧
  (* tsort is a good sorting *)
  sort g tsort ∧
  (* corollary hard to prove: ordered elements are vertices of the graph  *)
  (∀ x: int. 0 ⩽ x < S.cardinal (vertices g) → S.mem (M.get order x) (vertices g))
```

2 Définitions de test

Toutes ces lemmes, `path_ordered` excepté, sont également des traductions directes de ceux donnés dans l'énoncé. Le lemme `path_ordered` est prouvé par récurrence sur le chemin, et par conséquent nécessite une définition en `let rec lemma`. Par ailleurs, `path_ordered` et `no_cycle` étant utiles plus tard, ceux-ci ont été déplacés du module `TestDefinition` au module `Topo`.

```
lemma no_empty_path:
  ∀ g:graph. ∀ x:vertex. not (path_to g Nil x)

lemma path_for_two_edges:
  ∀ g:graph. ∀ x y z:vertex. S.mem (x,y) (edges g) ∧ S.mem (y,z) (edges g) → path_to
    g (Cons x (Cons y Nil)) z

lemma pred_smaller:
  ∀ g:graph. ∀ tsort:tsort. sort g tsort → ∀ x y:vertex. S.mem y (preds g x) → M.get
    tsort y < M.get tsort x

lemma no_direct_cycle1:
  ∀ g:graph. ∀ tsort:tsort. ∀ x y:vertex. sort g tsort → not (S.mem (x,y) (edges g) ∧
     S.mem (y,x) (edges g))

lemma no_direct_cycle2:
  ∀ g:graph. ∀ tsort:tsort. ∀ x:vertex. sort g tsort → not (S.mem (x,x) (edges g))
```

```
let rec lemma path_ordered (g:graph) (tsort:tsort) (p:path) (v1 v2:vertex)
  requires { path_to g (Cons v1 p) v2 }
  requires { sort g tsort }
  ensures  { M.get tsort v1 < M.get tsort v2 }
  variant { p }
  =
  match p with
  | Nil → ()
  | Cons v3 p1 → path_ordered g tsort p1 v3 v2
  end

lemma no_cycle:
  ∀ g:graph. ∀ tsort:tsort. sort g tsort → not (cycle_in g)
```

3 Tri topologique

Tout d'abord, l'invariant maintenu globalement est le suivant :

```
predicate inv (g:graph) (m:H.t int) =
  (∀ v: vertex. S.mem v (H.defined m) → S.subset (preds g v) (H.defined m)) ∧
  (S.subset (H.defined m) (vertices g)) ∧
  (∀ x: vertex. S.mem x (H.defined m) → 0 ⩽ m[x] < S.cardinal (H.defined m)) ∧
  (∀ x y: vertex. S.mem x (H.defined m) ∧ S.mem y (preds g x) → (m[y] < m[x])) ∧
  (∀ x y: vertex. S.mem x m.defined ∧ S.mem y m.defined ∧ m[x] = m[y] → x = y)
```

Il capture les propriétés suivantes :

- Tous les sommets ayant déjà été traités sont effectivement des sommets de `g` (nécessaire pour s'assurer qu'on ne fait pas d'accès en-dehors du domaine de définition du graphe),
- Tous les prédécesseurs d'un sommet qui a déjà été traité ont également déjà été traités (ainsi tout nouveau sommet ajouté à la fin respectera la condition d'ordre ci-dessous),
- La table `m` associe de manière injective un entier entre `0` et `S.cardinal (H.defined m) - 1` aux sommets déjà traités (nécessaire pour assurer la propriété de bijection sur l'ordre une fois complet),
- Cette association respecte l'ordre défini par le graphe (nécessaire pour assurer qu'on est bien en train de créer un tri topologique).

On s'occupe tout d'abord de la fonction `dfs`. Cette dernière effectue une recherche en profondeur à partir d'un nœud, en ajoutant ce nœud ainsi que tous ses antécédents au tri topologique en train d'être construit. L'argument `seen` représente l'ensemble des sommets vus sur le chemin jusqu'au nœud actuel depuis le début de la recherche en profondeur, et l'argument fantôme `seen_path` est une preuve qu'il existe un chemin du nœud actuel jusqu'à chacun des nœuds de `seen`.

```
let rec dfs (g:graph) (v:vertex)
            (seen:marked) (tsort:H.t int)
            (ghost seen_path: M.map vertex path)
            : unit
  requires { inv g tsort }
  requires { S.subset seen (vertices g) }
  requires { S.mem v (vertices g) }
  requires { ∀ x. S.mem x seen → path_to g (Cons v (M.get seen_path x)) x }
  variant { S.cardinal (vertices g) - S.cardinal seen }
  ensures { S.subset (H.defined (old tsort)) (H.defined tsort) }
  ensures { S.mem v (H.defined tsort) }
  ensures { inv g tsort }
  ensures { ∀ x: vertex. S.mem x (old tsort).defined → (old tsort)[x] = tsort[x] }
  ensures { ∀ x: vertex. S.mem x seen ∧ S.mem x tsort.defined → S.mem x (old tsort).
    defined }
  raises  { Cycle_found → cycle_in g }
 =
 'Init:
  if S.mem v seen then raise Cycle_found;
  if not (H.mem tsort v) then begin
    let p = ref (preds g v) in
    let seen' = S.add v seen in
    while not (S.is_empty !p) do
      invariant { inv g tsort }
      invariant { S.subset !p (preds g v) }
```

```
      invariant { S.subset (H.defined (at tsort 'Init)) (H.defined tsort) }
      invariant { S.subset (preds g v) (S.union !p (H.defined tsort)) }
      invariant { ∀ x: vertex. S.mem x (at tsort 'Init).defined → (at tsort 'Init)[x]
 = tsort[x] }
      invariant { not (S.mem v tsort.defined) }
      invariant { ∀ x: vertex. S.mem x seen ∧ S.mem x tsort.defined → S.mem x (at
 tsort 'Init).defined }
      variant { S.cardinal !p }
      let u = S.choose !p in
      let ghost npath = ref seen_path in
      let ghost z = ref seen in
      while not (S.is_empty !z) do (* ghost loop *)
        variant { S.cardinal !z }
        invariant { S.subset !z seen }
        invariant { ∀ x. S.mem x seen ∧ not S.mem x !z → path_to g (Cons u (M.get !
 npath x)) x }
        let ghost t = S.choose !z in
        npath := M.set !npath t (Cons v (M.get seen_path t));
        z := S.remove t !z
      done;
      npath := M.set !npath v Nil;
      dfs g u seen' tsort !npath;
      p := S.remove u !p
    done;
    H.add tsort v (H.size tsort)
  end
```

Les préconditions de cette fonction correspondent à la validité initiale de notre invariant global, le fait que `v` et les éléments de `seen` sont effectivement des nœuds de `g`, et à la validité de `seen_path`. De plus, comme toute itération ajoute un nœud à la l'ensemble de ceux déjà vus dans le chemin actuel, le nombre de nœuds non encore vus est bien un variant. Finalement, la fonction certifie que le nouveau tri topologique partiel défini est une extension du précédent (pas de valeur enlevée, les anciennes valeurs sont préservées), que l'invariant global est préservé, que `v` est à présent défini dedans, ainsi qu'une condition technique, qui dit qu'aucun nouvel élément de `seen` n'est défini (utile dans l'invariant plus loin). On assure également qu'on ne lèvera l'exception `Cycle_found` qu'au cas où `g` contient effectivement un cycle.

Pour cela, on a besoin d'un invariant de boucle puisqu'on doit traiter les prédécesseurs de `v` un à un. La variable `p` représentant les prédécesseurs de `v` restant à traiter, on a comme invariants, le fait que notre invariant global est toujours vérifié ; que `p` est bien un sous-ensemble des prédécesseurs de `v`, que le tri topologique est bien une extension du précédent, mais qui ne définit pas `v`, que tout prédécesseur de `v` est soit dans `p` (encore à traiter), soit déjà défini dans le tri actuel ; et enfin, qu'on a toujours notre condition technique qui est que l'on n'a pas défini dans le tri de nouvel élément de `seen`. Un variant est naturellement `S.cardinal !p` puisque `p` se contente de diminuer.

Cet invariant est effectivement préservé assez naturellement, la seule condition non-triviale à prouver étant que `v` n'a pas été défini entre-temps par un appel récursif. Cela est prouvé par notre condition technique, puisque `v` se trouve dans tout appel récursif. Lors des appels récursifs, on a également besoin de modifier `seen_path` puisque le sommet a changé ; c'est le rôle de la boucle interne fantôme. Son invariant est naturel (tout sommet qui n'est pas encore à traiter dispose à présent d'un chemin correct), on ne le détaillera pas ici.

De plus, cet invariant est initialement vrai grâce au préconditions de la fonction `dfs`, de manière assez immédiate ; tandis que son effet final est en particulier d'assurer que tous les prédécesseurs de `v` ont été ajoutés dans le tri topologique, ce qui permet d'autoriser l'ajouter de `v` lui-même (qu'on assure ne pas avoir déjà ajouté).

En ce qui concerne l'exception `Cycle_found`, celle-ci ne peut être levée que si `v` se trouve être dans `seen`, auquel cas `seen_path` nous fournit précisément un cycle de `g`.

Étudions à présent `topo_tsort`. Cette dernière fonction se contente d'appeler `dfs` avec tous les sommets du graphe successivement. On a donc la fonction suivante :

```
let topo_tsort (g:graph): H.t int
  raises  { Cycle_found → cycle_in g  }
  ensures { sort g result.contents }
  ensures { S.(==) result.defined (vertices g) }
 =
  let tsort = H.create () in
  let p = ref (vertices g) in
  while not (S.is_empty !p) do
    invariant { inv g tsort }
    invariant { S.subset !p (vertices g) }
```

```
    invariant { S.subset (vertices g) (S.union !p (H.defined tsort)) }
    variant { S.cardinal !p }
    let u = S.choose !p in
    dfs g u (S.empty) tsort (Const.const Nil);
    p := S.remove u !p
  done;
  tsort
```

Son contrat est donc que cette fonction ne lève l'erreur `Cycle_found` que si `g` contient un cycle, et que si ce n'est pas le cas, elle renvoie un tri topologique de `g` bien défini sur tous les sommets de `g`. Pour cela, on a notre invariant global, ainsi que les invariants classiques d'une boucle qui itère sur un ensemble de sommets : l'ensemble des sommets non traités est un sous-ensemble de l'ensemble sur lequel itérer, l'union des sommets déjà traités et de ceux sur lesquels il faut encore itérer contient tous les sommets sur lesquels itérer, et l'ensemble des sommets sur lesquels itérer diminue en taille. L'invariant global est satisfait au début (par quantification sur ensemble vide), et le fait qu'il soit satisfait à la fin est juste une reformulation du fait que le résultat doit être un tri topologique.

Enfin, considérons `topo` définie ci-dessous.

```
let topo (g:graph): topo
  raises  { Cycle_found → cycle_in g }
  requires { not (S.is_empty (vertices g)) }
  ensures { topological_sort g result.order.elts result.tsort.contents }
  ensures { S.(==) result.tsort.defined (vertices g) }
  ensures { result.order.length = S.cardinal (vertices g) }
 =
  let tsort = topo_tsort g in
  let order = Array.make (S.cardinal (vertices g)) (S.choose (vertices g)) in
  let p = ref (vertices g) in
  let ghost todo = ref (range_set (S.cardinal (vertices g))) in
  while not (S.is_empty !p) do
    invariant { S.subset !p (vertices g) }
    invariant { ∀ x. S.mem x (S.diff (vertices g) !p) → order[H.([]) tsort x] = x }
    invariant { ∀ i. 0 ⩽ i < S.cardinal (vertices g) → S.mem order[i] (vertices g) }
    invariant { S.cardinal !todo = S.cardinal !p }
    invariant { ∀ x. S.mem x !p → S.mem (H.([]) tsort x) !todo }
    invariant { ∀ i. 0 ⩽ i < S.cardinal (vertices g) ∧ not (S.mem i !todo) → H.([])
   tsort order[i] = i }
    variant { S.cardinal !p }
    let u = S.choose !p in
    order[H.find tsort u] ← u;
    p := S.remove u !p;
    todo := S.remove (H.find tsort u) !todo;
  done;
  {order = order; tsort = tsort}
```

À nouveau, cette fonction ne peut lever une erreur que si `g` contient un cycle. De plus, on requiert cette fois-ci que `g` contienne au moins un sommet, pour pouvoir initialiser le tableau (qui serait sinon certes vide) contenant le tri topologique (dans le sens qui associe un sommet à chaque entier). Le résultat est alors un tri topologique avec les bijections dans chaque sens, comme défini par `topological_sort`. Cette fonction fonctionne avec l'algorithme classique d'inversion de bijection, après avoir calculé une des bijections avec `topo_tsort`.

On itère donc encore une fois sur un ensemble de sommets, avec les invariants et variants correspondants, un sommet déjà traité vérifiant `order[H.([]) tsort x] = x`. On ajoute un autre invariant qui spécifie que tous les éléments de `order` sont des sommets de `g`, qui est requis dans la définition de `topological_sort`. Cependant, on a besoin de la propriété sur la bijection inverse également : c'est le rôle de la variable fantôme `todo`. Celle-ci contient les images des éléments de `p` par `tsort` ; cette propriété est garantie par les invariants qui disent que ces deux ensembles ont le même cardinal, et que toute image d'un élément de `p` par `tsort` est dans `todo`. On garde donc l'invariant symétrique de celui sur `p`, ce qui nous permet de prouver la bijection inverse, qui dit que pour tout index `i` qui n'a pas été traité, `H.([]) tsort order[i] = i`. Comme `p` et donc `todo` sont vides à la fin, tous les sommets vérifieront la propriété de la bijection souhaitée.

4 Calcul des dominateurs immédiats

Par rapport au code fourni dans l'énoncé, on a légèrement modifié `find_common` pour qu'elle ne lève plus l'exception `Root` dans le cas où elle atteindrait la racine.

On crée, comme suggéré dans l'énoncé, un prédicat `partial_domine g root x y a`, qui a la même signification que `domine` lorsqu'on le restreint aux chemins passant par au moins un élément de `a`. On définit alors `partial_idomine` à partir de celui-ci comme on avait défini `idomine` à partir de `domine`.

```
predicate partial_domine (g:graph) (root x y:vertex) (a: S.set vertex) =
  y ≠ root ∧
  ∀ p. path_to g (Cons root p) y → not_disjoint a (S.add root (elements p)) → S.mem x
     (S.add root (elements p))

predicate partial_idomine (g:graph) (root x y: vertex) (a: S.set vertex) =
  partial_domine g root x y a ∧ (∀ x'. x' ≠ x → partial_domine g root x' y a → domine
     g root x' x)
```

Ce prédicat est particulièrement utile dans la spécification de `find_common` qu'on verra plus loin. Les trois lemmes les plus importants à son propos sont :

```
lemma partial_idomine_self: ∀ g root x y. y ≠ root ∧ x ≠ root ∧ S.mem (x,y) (edges g)
     → partial_idomine g root x y (S.singleton x)

lemma partial_idomine_preds: ∀ g root x y. y ≠ root ∧ partial_idomine g root x y (
    preds g y) → idomine g root x y

lemma partial_idomine_root: ∀ g root y a. y ≠ root ∧ partial_idomine g root root y a
    → idomine g root root y
```

En particulier, ces lemmes permettent de prouver que certains nœuds sont des dominateurs immédiats partiels, et à partir d'un dominateurs immédiat partiel, d'en déduire qu'il est un dominateur immédiat. Le chaînon manquant sera la spécification de `find_common`.

On ne donnera pas précisément ici les autres lemmes utilisés, qui servent principalement à prouver les lemmes ci-dessus. Leur preuve est toujours assez naturelle, même si il faut un peu aider les prouveurs à se convaincre de leur véracité.

La fonction de calcul des dominateurs immédiats est alors :

```
let idoms g (root:vertex) (topo:Topo.topo) (ghost exi_path : M.map vertex path)
    : array int
  requires { S.mem root (vertices g) }
  requires { S.is_empty (preds g root) }
  requires { ∀ v. v ≠ root → S.mem v (vertices g) → path_to g (Cons root (M.get
    exi_path v)) v }
  requires { topological_sort g topo.Topo.order.elts topo.Topo.tsort.Topo.H.contents
    }
  requires { S.(==) topo.Topo.tsort.Topo.H.defined (vertices g) }
  requires { topo.Topo.order.length = S.cardinal (vertices g) }

  ensures { ∀ i. 0 < i < S.cardinal (vertices g) →
               idomine g root (topo.Topo.order[result[i]]) (topo.Topo.order[i]) }
  ensures { result[0] = 0 }
 =
  let a = Array.make (S.cardinal (vertices g)) (-1) in
  a[0] ← 0;
  for nv=1 to (S.cardinal (vertices g) - 1) do
    invariant { a[0] = 0 }
    invariant { ∀ i. 0 < i < nv → idomine g root (topo.Topo.order[a[i]]) (topo.Topo.
    order[i]) }
    invariant { ∀ i. 0 < i < nv → 0 ⩽ a[i] < i }
    let v = topo.Topo.order[nv] in
    assert { v ≠ root by (∀ v. v ≠ root → S.mem v (vertices g) → path_to g (Cons root
     (M.get exi_path v)) v) };
    let rec find_common n1 n2 (ghost a1 a2 : S.set vertex)
      requires { 0 ⩽ n1 < nv }
      requires { 0 ⩽ n2 < nv }
      requires { a[0] = 0 }
      requires { ∀ i. 0 < i < nv → idomine g root (topo.Topo.order[a[i]]) (topo.Topo.
    order[i]) }
      requires { ∀ i. 0 < i < nv → 0 ⩽ a[i] < i }
      requires { partial_domine g root (topo.Topo.order[n1]) v a1 }
      requires { partial_domine g root (topo.Topo.order[n2]) v a2 }
      requires { ∀ x'. x' ≠ (topo.Topo.order[n2]) ∧
```

```
31                 partial_domine g root x' v (S.union a1 a2) →
32                   domine g root x' (topo.Topo.order[n2]) }
33     requires { ∀ x'. x' ≠ (topo.Topo.order[n1]) ∧
34                 partial_domine g root x' v (S.union a1 a2) →
35                   domine g root x' (topo.Topo.order[n1]) }
36     ensures { 0 ⩽ result ⩽ n1 }
37     ensures { 0 ⩽ result ⩽ n2 }
38     ensures { partial_idomine g root (topo.Topo.order[result]) v (S.union a1 a2) }
39     variant { n1 + n2 }
40    =
41     if n1 = n2 then n1
42     else
43     let a1, a2 = if n2 < n1 then (a1, a2) else (a2, a1) in
44     let n1, n2 = if n2 < n1 then (n1,n2) else (n2,n1) in
45     assert { topo.Topo.order[n1] ≠ topo.Topo.order[n2] };
46     assert { not (partial_domine g root (topo.Topo.order[n1]) v (S.union a1 a2)) by
47              not (domine g root (topo.Topo.order[n1]) (topo.Topo.order[n2])) };
48     assert { idomine g root (topo.Topo.order[a[n1]]) (topo.Topo.order[n1]) };
49     find_common a[n1] n2 a1 a2
50   in
51   try
52     let p = ref (preds g v) in
53     let ghost _ = path_to_has_pred g v (Cons root (M.get exi_path v)) in
54     assert { not (S.is_empty !p) };
55     let u = S.choose !p in
56     (if u = root then raise Root);
57     let nu = Topo.H.find topo.Topo.tsort u in
58     let idom_v = ref nu in
59     let ghost a1 = ref (S.singleton u) in
60     p := S.remove u !p;
61     while not (S.is_empty !p) do
62       invariant { S.subset !p (preds g v) }
63       invariant { S.(==) (preds g v) (S.union !p !a1) }
64       invariant { partial_idomine g root (topo.Topo.order[!idom_v]) v !a1 }
65       invariant { 0 ⩽ !idom_v < nv }
66       invariant { a[0] = 0 }
67       invariant { ∀ i. 0 < i < nv → idomine g root (topo.Topo.order[a[i]]) (topo.
 Topo.order[i]) }
68       invariant { ∀ i. 0 < i < nv → 0 ⩽ a[i] < i }
69       variant { S.cardinal !p }
70       let u = S.choose !p in
71       (if u = root then raise Root);
72       let nu = Topo.H.find topo.Topo.tsort u in
73       idom_v := find_common !idom_v nu !a1 (S.singleton u);
74       p := S.remove u !p;
75       a1 := S.add u !a1;
76     done;
77     a[nv] ← !idom_v;
78   with Root →
79     a[nv] ← 0;
80   end
81 done;
82 a
```

Cette fonction est assez compliquée, expliquons-la donc partie par partie.

Tout d'abord, les lignes 3 à 12 expriment le contrat de la fonction. Celui-ci demande un tri topologique du graphe (en particulier, celui-ci doit donc être acyclique), ainsi que l'existence d'une racine du graphe depuis laquelle tout sommet est accessible (l'argument fantôme `exi_path` étant une preuve de cette propriété). Le résultat de l'exécution de la fonction sera un tableau donnant pour chaque sommet son dominateur immédiat, et pour la racine elle-même, les sommets étant identifiés par leur ordre dans le tri topologique.

Après avoir marqué la racine dans le tableau créé, le reste de la fonction s'occupe dans l'ordre du tri topologique de calculer tous les dominateurs immédiats. Cet invariant est spécifiés aux lignes 17 à 19. Il dit simplement que les dominateurs immédiats ont correctement été calculés jusqu'au sommet actuel, en prenant `root` pour le dominateur immédiat de `root`. De plus, il certifie la propriété que tout dominateur immédiat d'un sommet est avant ce dernier dans l'ordre du tri topologique, propriété qui peut être pénible à prouver séparément. L'assertion ligne 21 sert à aider les prouveurs, qui ont des difficultés à prouver cette propriété sans l'aide de cette ligne (et même avec, seul Eprover est capable de la prouver, malgré l'aide d'un lemme séparé prouvant exactement cette propriété).

Passons donc à la fonction `find_common`, qui est le cœur de l'algorithme. Son contrat est spécifié aux lignes 23 à 39. Cependant, une version plus forte des préconditions des lignes 28 à 35 est donnée ci-dessous :

```
requires { partial_idomine g root (topo.Topo.order[n1]) v a1 }
requires { partial_idomine g root (topo.Topo.order[n2]) v a2 }
```

Cela montre l'intérêt de la fonction `find_common` : à partir de deux dominateurs immédiats partiels de `v` pour les ensembles `a1` et `a2`, elle renvoie un dominateur partiel pour l'ensemble `S.union a1 a2`. En particulier, elle se combine avec les lemmes précédents qui nous donnaient un dominateur immédiat partiel pour tout ensemble qui était un singleton contenant un prédécesseur de `v`, ainsi que le fait qu'un dominateur immédiat partiel pour pour les prédécesseurs de `v` était en fait un dominateur immédiat de `v`.

La condition utilisée est cependant plus faible, ce qui est nécessaire pour qu'elle passe lors de l'appel récursif. Elle stipule que `n1` et `n2` correspondent à des dominateurs partiels de `v` pour `a1` et `a2`, ainsi que le fait que tout dominateur partiel de `v` pour `S.union a1 a2` domine les sommets correspondants à `n1` et `n2`. Cette dernière propriété peut se comparer à celle qui dit que lors du calcul d'un pgcd, tout diviseur commun des nombres de départ divise les deux nombres présents à chaque étape.

On a également quelques autres conditions, qui sont simplement notre invariant de boucle précédent, ainsi que des bornes sur `n1` et `n2` qui nous garantissent que les accès au tableau des dominateurs immédiats se font toujours à des valeurs déjà calculées. Enfin, cette fonction termine puisque `n1 + n2` décroît, étant donné qu'à chaque étape, au moins un des ces sommets est remplacé par son dominateur immédiat.

Muni de cette fonction, calculer le dominateur immédiat de `v` est naturel : on itère sur les prédécesseurs, en rajoutant un prédécesseur (correspondant au dominateur partiel pour un ensemble singleton le contenant) à chaque étape. Il nous faut pour ça un moins un prédécesseur du nœud, ce qui est assuré par la ligne 53 qui sert à aider les prouveurs à vérifier l'assertion de la ligne suivante. L'invariant de cette boucle est donné aux lignes 62 à 69. On a comme d'habitude un ensemble de sommets restant à traiter ; on garde comme invariant l'invariant global sur la correction des valeurs déjà calculées du tableau des dominateurs immédiats, ainsi que le fait que la valeur actuelle `idom_v` est un dominateur partiel immédiat pour l'ensemble des prédécesseurs déjà traités. On a finalement aussi les invariants classiques de ce genre de boucle, stipulant que l'ensemble des sommets restant à traiter est bien un sous-ensemble de ceux à traiter, et que son union avec ceux déjà traités est égale à ceux à traiter.

Après cette boucle, on peut alors mettre à jour le tableau des dominateurs immédiats avec la valeur obtenue, et passer à l'itération suivante. Dans le cas de l'exception `Root`, on utilise `root` comme dominateur immédiat, ce qui ne peut se produire que si `root` est un dominateur immédiat partiel, et donc a fortiori un dominateur immédiat.

Une fois la boucle extérieure terminée, le tableau des dominateurs immédiats est complet – ce qui est ce que l'on souhaitait calculer.