Preuve formelle d'un analyseur flot de données paramétré par un ordre d'itération

Roméo La Spina, Sandrine Blazy, Delphine Demange

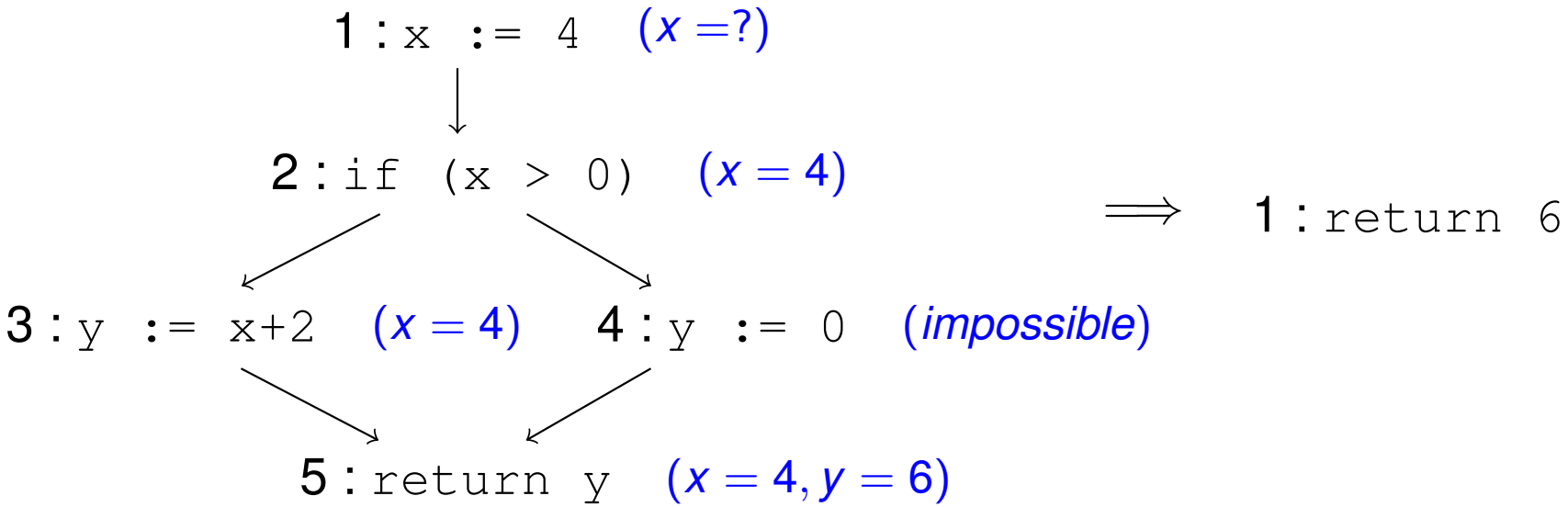
Univ Rennes, Inria, CNRS, IRISA

Journées CLAP/HiFi/LVP 2023

Analyse flots de données

But : Calcul d'information à chaque point d'un *graphe de flot de contrôle* (GFC)

$\Rightarrow$ **Exemple :** Optimisations de compilation



Problème

- **Historique :** Algorithme de Kildall [Kil73]
  - Référence de l'état de l'art.
  - Implémenté et prouvé en Coq dans CompCert.
- **Alternative :** Algorithme de Bourdoncle [Bou93]
  - Très utilisé en interprétation abstraite.
  - Possibilité d'utiliser des opérateurs d'élargissement
  - Mais peu de formalisation détaillée.

Objectifs :

- Décrire formellement la méthode de Bourdoncle
- Etablir sa correction

Analyse flots de données (1)

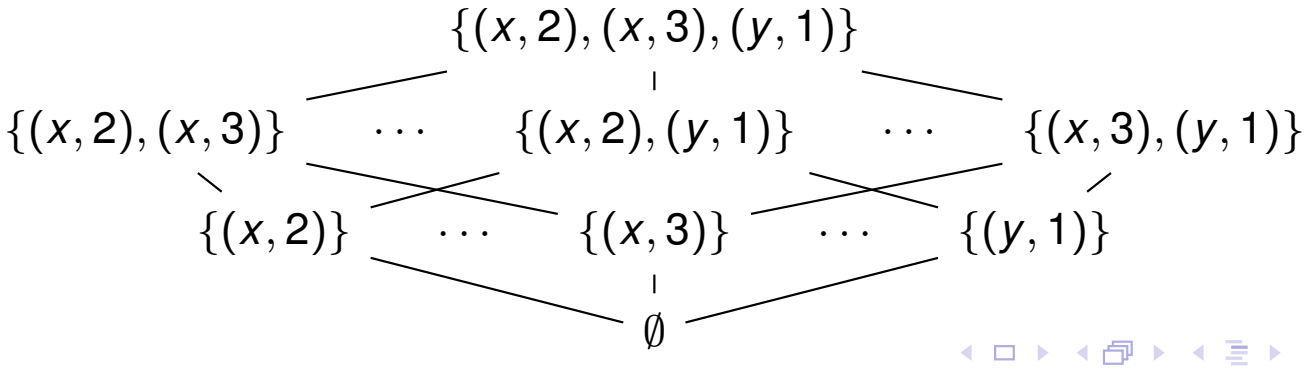
Définition (Demi-treillis)

Un *demi-treillis* est un ensemble *partiellement ordonné* $(L, \sqsubseteq)$ avec :

- un opérateur de borne supérieure $\sqcup$
- un plus petit élément $\bot$

Exemple (Demi-treillis des définitions)

Soit $\mathcal{V}$ est l'ensemble des variables, N l'ensemble des nœuds. On peut considérer le demi-treillis des définitions $(\mathcal{P}(\mathcal{V} \times N), \subseteq, \cup, \emptyset)$.

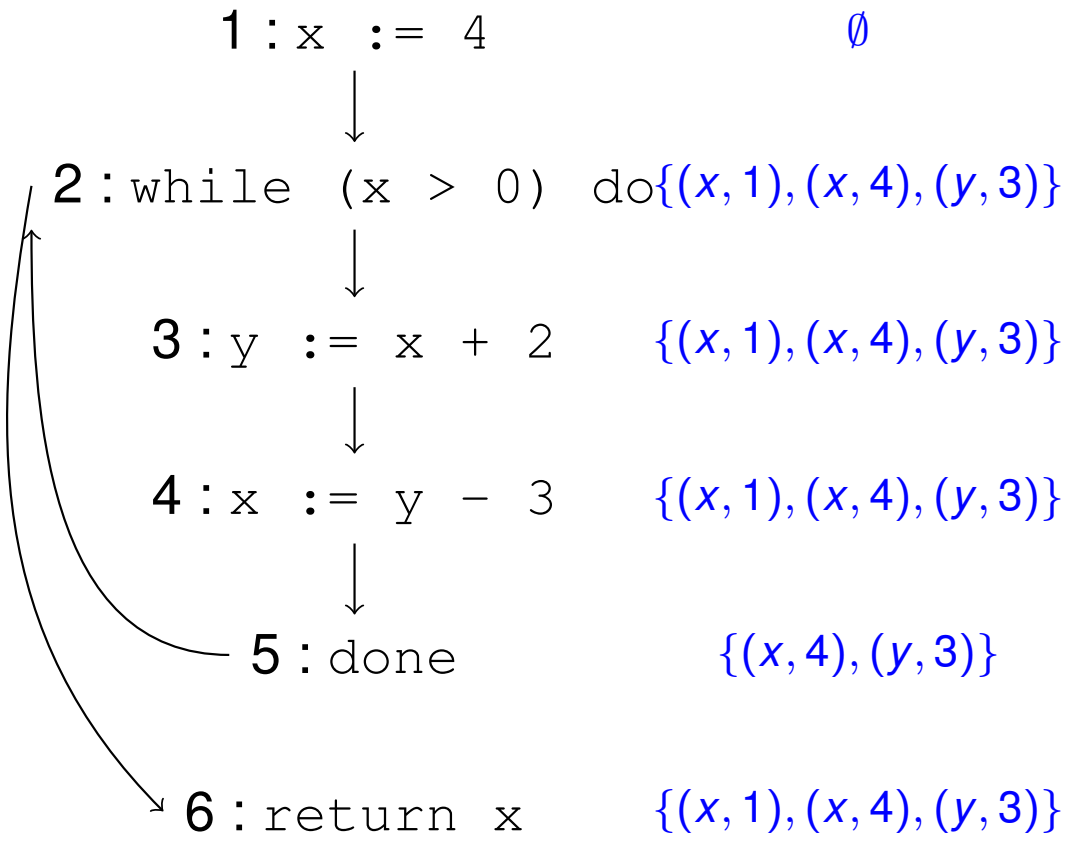


Analyse flots de données (2)

- GFC $G = (N, E)$
- Demi-treillis L
- Fonction de transfert $T : N \times L \to L$
- On calcule une valeur de L à chaque nœud du graphe
- **Système d'équations :** on cherche A tel que si $n \to s$,

$$T(n, A[n]) \sqsubseteq A[s]$$

Exemple : Définitions atteignantes

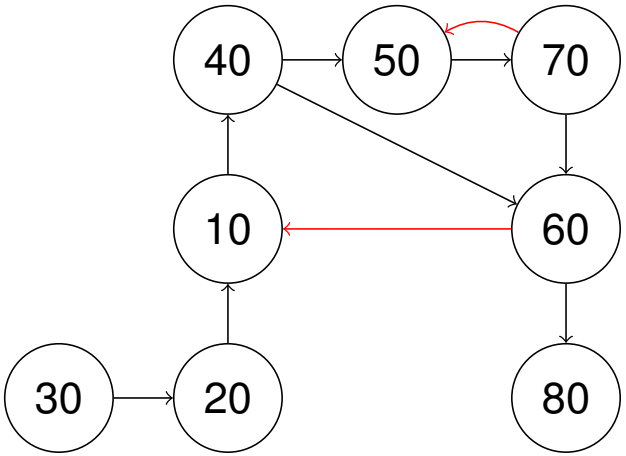


Principes de base

- **Kildall :** *W* est une liste contenant initialement un seul nœud d'entrée.
  - Prendre un nœud dans *W*
  - Propager l'information vers ses successeurs
  - Ajouter à *W* ceux auquels on a ajouté de l'information,
  - Recommencer jusqu'à ce que *W* soit vide.
- **Bourdoncle : paramétré par un ordre d'itération**
  $\Rightarrow$ Calcul d'un *tri topologique faible* sur le graphe.

Tri topologique faible

- Ordre + parenthésage construit à partir du GFC
- **Composante :** nœuds entre deux parenthèses
- **Tête :** premier nœud d'une composante
- **Propriété d'un t.t.f. :** $\forall$ arc arrière $u \to v$ de G, v est la tête d'une composante qui contient u
- *A priori*, têtes de composantes $\neq$ têtes de boucles !



$w_0 = 30\ 20\ (10\ 40\ (50\ 70)\ 60)\ 80$

Algorithme de Bourdoncle

- **Entrées :** un GFC muni d'un t.t.f. + $(L, \sqsubseteq, \sqcup, \bot)$, T
- **Sortie :** une solution A du système d'équations
- T.t.f. $\Rightarrow$ Ordre dans lequel on va traiter les nœuds

Soit $w_0 = 30\ 20\ (10\ 40\ (50\ 70)\ 60)\ 80$.

- **Stratégie itérative :**
  Itérations sur les composantes englobantes

$$30\ 20\ \underbrace{[10\ 40\ 50\ 70\ 60]^*}_{\textit{stabilisation}}\ 80$$

- **Stratégie récursive :** itérations récursives

$$30\ 20\ [10\ 40\ [50\ 70]^*\ 60]^*\ 80$$

Contribution - Tri topologique faible

- Représentation formelle d'un t.t.f.

  $\Rightarrow$ Définition mutuellement récursive d'un *élément* :

```
Inductive element : Type :=
| Vertex : positive → element
| Component : positive → list element → element.
```

  $\Rightarrow$ Formalisation des propriétés d'un t.t.f.

- Calcul préalable du t.t.f. :
  - Méthode DFN prouvée directement en Coq

$$w_0 = 30\ 20\ (10\ 40\ (50\ 70\ 60\ 80))$$

  - Méthode des sous-composantes fortement connexes, validée *a posteriori*

Contribution - Algorithme de Bourdoncle

- Pseudo-code des deux stratégies de Bourdoncle
- Implémentation dans CompCert + preuve de correction
- Implémentation **générique**
- Gestion des analyses avant/arrière
- Comparaison avec Kildall sur les analyses de CompCert

Spécification en Coq

Rappel de la spécification : Soit A le résultat de l'analyse. Alors si $n \rightarrow s$, on a

$$T(n, A[n]) \sqsubseteq A[s]$$

Spécification générique (non dépendante de l'algorithme).

```
Theorem fixpoint_solution:
∀ res n instr s,
  (* res is the result of the analysis *)
  fixpoint = Some res →
  (* s is a successor of n *)
  code!n = Some instr →
  In s (successors instr) →
  (* The result is indeed a solution *)
  L.ge res[s] (transf n res[n]).
```

Idée générale de la preuve

Théorème (Spécification)

Soit A le résultat de l'analyse. Alors si $n \to s$, on a

$$T(n, A[n]) \sqsubseteq A[s]$$

- On raisonne sur un ensemble S d'éléments stabilisés
- **Invariant :** $\forall$ élément $e \in S$, $\forall n \in e$, si $n \to s$ alors

$$T(n, A(n)) \sqsubseteq A(s)$$

- A la fin,
  - Tous les éléments sont stabilisés
  - Tous les nœuds du graphe sont dans un élément
- Induction mutuelle pour la stratégie récursive

Evaluation expérimentale

- Programmes de tests fournis avec CompCert
  $\Rightarrow$ Programmes de taille / de structure variables
- T.t.f. calculé avec la méthode des s.c.f.c.
- Pour **plusieurs analyses** de CompCert
  $\Rightarrow$ vivacité, code mort, analyse de valeurs, ...
- Kildall vs. Bourdoncle itér. vs. Bourdoncle réc. :
  - Temps d'analyse sans validation *a posteriori*
  - Nombre d'applications de $\sqcup$ (borne sup. du demi-treillis)
  - Nombre d'applications de la fonction de transfert
  - Précision de l'analyse

Résultats

- $\sqcup$ **et** T : $\text{Bourdoncle}_{\text{rec}} < \text{Bourdoncle}_{\text{iter}} \ll \text{Kildall}$
  $\Rightarrow$ Moins de propagation de l'information
- **Temps :** $\text{Bourdoncle}_{\text{iter}} \approx \text{Kildall} < \text{Bourdoncle}_{\text{rec}}$
  $\Rightarrow$ Plus de calculs pour ne pas propager inutilement
  $\Rightarrow$ Pourrait toutefois être avantageux pour des treillis hauts
- Différences de précision mineures
  $\Rightarrow$ Pas d'impact sur les optimisations de compilation

Conclusion et perspectives

- Formalisation de la notion de t.t.f.
- Vérification formelle des deux stratégies de Bourdoncle
- Performances encourageantes, comparables à Kildall
- Dans le futur :
  - Optimisations sur l'implémentation + preuve
    $\Rightarrow$ Test de stabilisation aux têtes de composantes
  - Preuve du calcul du t.t.f. avec l'algorithme des s.c.f.c.

Références I

François Bourdoncle, *Efficient chaotic iteration strategies with widenings*, Formal Methods in Programming and Their Applications (Berlin, Heidelberg) (Dines Bjørner, Manfred Broy, and Igor V. Pottosin, eds.), Springer Berlin Heidelberg, 1993, pp. 128–141.

Gary A. Kildall, *A unified approach to global program optimization*, Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (New York, NY, USA), POPL '73, Association for Computing Machinery, 1973, p. 194–206.

Définitions (1)

Définition (**Tri hiérarchique**)

Un *tri hiérarchique* $\mathcal{H}$ d'un ensemble est une permutation des élements de cet ensemble. $u \preceq_{\mathcal{H}} v$ signifie "u est avant v dans le tri hiérarchique, ou $u = v$". Cette relation définit un ordre total sur les élements de l'ensemble.

Définition (**Parenthésage**)

Un *parenthésage* d'un tri hiérarchique est un parenthésage bien formé sur les éléments de la permutation donné par le tri hiérarchique, tel qu'on n'a jamais deux parenthèses ouvrantes consécutives.

Définitions (2)

Définition (**Composantes, têtes**)

Une *composante* d'un parenthésage d'un tri hiérarchique correspond aux éléments contenus entre deux parenthèses correspondantes dans le parenthésage. Le premier élement de la composante est appelé la *tête* de la composante. Pour un élément u, on note $\omega(u)$ l'ensemble des têtes des composantes qui contiennent u.

Définition (**Arc arrière**)

Etant donné un tri hiérarchique $\mathcal{H}$, un arc $(u, v) \in E$ du graphe de flot de contrôle est un *arc arrière* si $v \preceq u$.

Définitions (3)

Définition (**Tri topologique faible**)

Un *tri topologique faible* (t.t.f.) d'un graphe de flot de contrôle G est un parenthésage d'un tri hiérarchique de l'ensemble des nœuds de G, qui vérifie la propriété suivante : pour tout arc arrière $u \to v$ de G, on a $v \in \omega(u)$

Définition (**Composantes externes**)

On définit $\mathcal{O}(w)$ comme étant la liste des *composantes externes* d'un t.t.f. w, c'est-à-dire la liste obtenue en appliquant H_{elt} sur les élements de w.

Bourdoncle - Stratégie itérative

Algorithm 1 Bourdoncle with iterative strategy

```
1:  S ← ∅
2:  aval ← the mapping from every node to ⊥
3:  OK ← ∅
4:  m ← false
5:  procedure PROPAGATE(n)
6:      out ← T(n, aval[n])
7:      OK ← OK ∪ {n}
8:      for each s ∈ succ(n) do
9:          in ← aval[s] ⊔ out
10:         if in ≠ aval[s] then
11:             aval[s] ← in
12:             OK ← OK \ {s}
13:             m ← true
14:         end if
15:     end for
16: end procedure
17: procedure STABILIZE(l)
18:     repeat
19:         m ← false
20:         for each node n ∈ l do
21:             if n ∉ OK then PROPAGATE(n)
22:             end if
23:         end for
24:     until m = false
25: end procedure
26: for each outermost element o ∈ 𝒪(w) do
27:     if o = OV n then PROPAGATE(n)
28:     else if o = OC l then STABILIZE(l)
29:     end if
30:     S ← S ∪ {o}
31: end for
32: return aval
```

Bourdoncle - Stratégie récursive

Algorithm 2 Bourdoncle with recursive strategy

```
1:  S ← ∅
2:  aval ← the mapping from every node to ⊥
3:  OK ← ∅
4:  m ← false
5:  function STABILIZE(l)
6:      m_loc ← false
7:      repeat
8:          m ← false
9:          for each element e ∈ l do
10:             if e = V n then
11:                 if n ∉ OK then
12:                     PROPAGATE(n)
13:                 end if
14:             else if e = C h t then
15:                 m_old ← m
16:                 m ← STABILIZE(V h :: t) ∨m_old
17:             end if
18:         end for
19:         m_loc ← m_loc ∨ m
20:     until m = false
21:     return m_loc
22: end function
23: procedure PROPAGATE(n)
24:     out ← T(n, aval[n])
25:     OK ← OK ∪ {n}
26:     for each s ∈ succ(n) do
27:         in ← aval[s] ⊔ out
28:         if in ≠ aval[s] then
29:             aval[s] ← in
30:             OK ← OK \ {s}
31:             m ← true
32:         end if
33:     end for
34: end procedure
35: for each element e ∈ w do
36:     if e = V n then PROPAGATE(n)
37:     else if e = C h l then STABILIZE(V h :: l)
38:     end if
39:     S ← S ∪ {e}
40: end for
41: return aval
```

Algorithme de Kildall

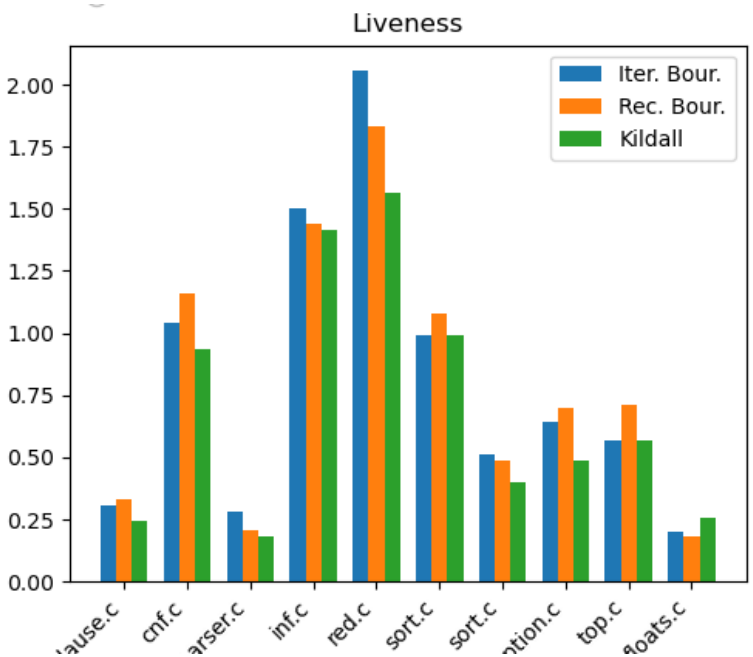
n_e est le nœud d'entrée, initialisé à v_e.

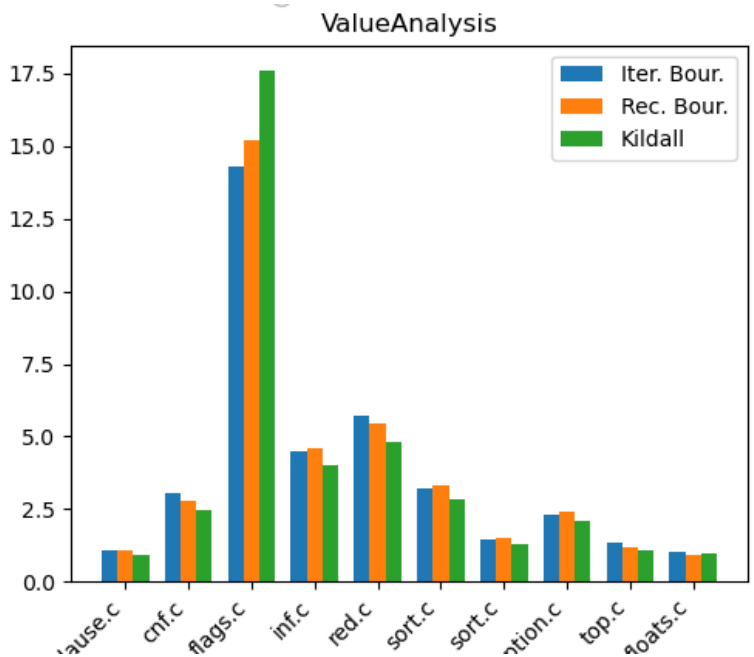
```
1  W ← {n_e}
2  A ← le mapping qui associe chaque nœud à ⊥
3  A[n_e] ← v_e
4  tant que W ≠ ∅ faire
5  |   extraire un nœud n de W
6  |   out ← T(n, A[n])
7  |   pour chaque successeur s de n faire
8  |   |   in ← A[s] ⊔ out
9  |   |   si in ≠ A[s] alors
10 |   |   |   A[s] ← in
11 |   |   |   W ← W ∪ {s}
12 |   |   fin si
13 |   fin pour chaque
14 fin tq
15 retourner A
```

Dans CompCert

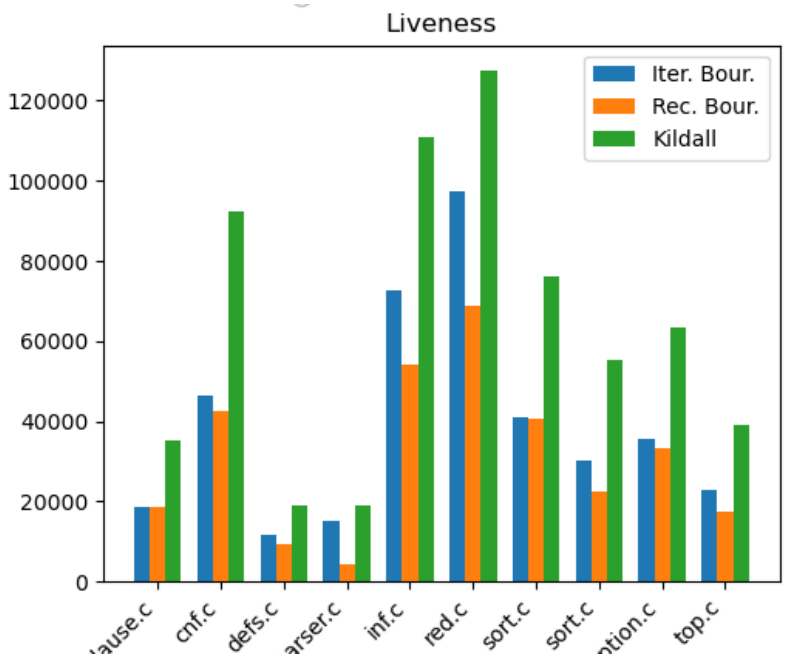
- Module `Kildall` : implémentation existante de l'algorithme de Kildall
- Modules `Bourdoncle` et `BourdoncleRec` : notre implémentation des 2 stratégies + preuve de correction
- Module `Wto` pour calculer le t.t.f.

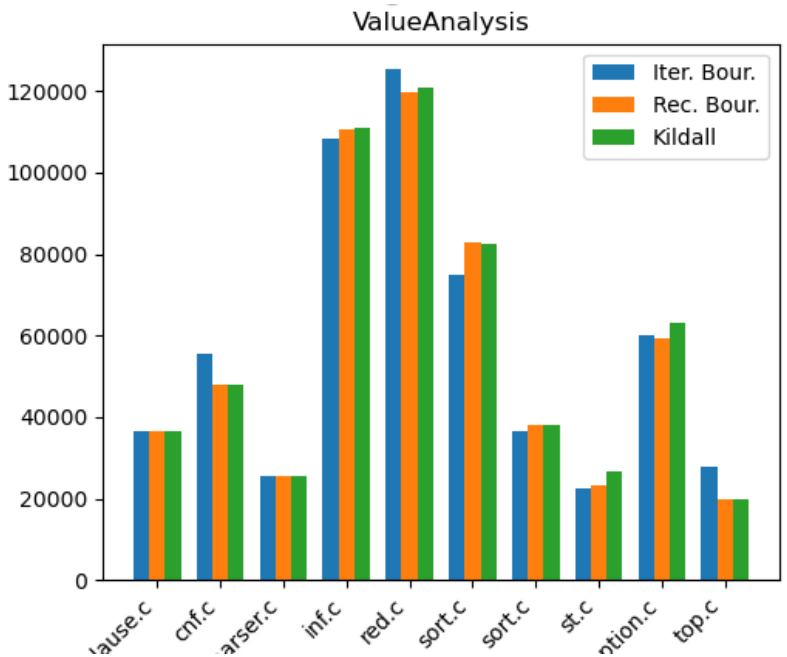
Temps d'analyse





Application de ⊔





Application de T