



UNIVERSITÉ PARIS CITÉ

REPRÉSENTATION DES CONNAISSANCES ET
RAISONNEMENT
INTRODUCTION À L'ARGUMENTATION - PROJET
RAPPORT

Implémentation d'Algorithmes pour les Systèmes d'Argumentation Abstraits

Élèves :

Hugo CONVERT
Kiswendsida OUEDRAOGO

Enseignant :

Elise BONZON

26 décembre 2024

Table des matières

1	Introduction	3
2	Objectifs et fonctionnalités attendues du projet	3
3	Algorithmes et structures de données	3
3.1	Algorithmes	3
3.2	structures de données	8
4	Implémentation et tests	8
4.1	Implémentation	8
4.1.1	Architecture	8
4.2	Tests et validation	9
4.2.1	Remarque sur les fichiers de test.	11
5	Conclusion	11

Résumé

Ce rapport présente le travail réalisé dans le cadre du développement d'un outil dédié à la résolution de problèmes dans le domaine des systèmes d'argumentation abstraits. Nous avons commencé par une exploration des algorithmes de calcul des extensions complètes et stables, en nous appuyant sur des travaux de recherche sur l'argumentation. Ces algorithmes ont été décrits, accompagnés d'explications sur leurs mécanismes et les structures de données utilisées pour leur implémentation.

Sur la base de ces algorithmes, nous avons pu répondre aux problématiques posées dans le sujet du projet, telles que la détermination des extensions d'un système d'argumentation donné et l'évaluation de l'appartenance d'un argument à une extension.

En complément, une interface graphique permettant de visualiser les systèmes d'argumentation, leurs arguments, et les relations entre eux, a été développée pour faciliter l'analyse des résultats.

1 Introduction

Ce projet a nécessité une revue de littérature, au cours de laquelle nous avons identifié des ressources clés pour notre travail.

Tout d'abord, le cours d'Élise Bonzon sur l'argumentation computationnelle [1] nous a offert une base pour appréhender les concepts fondamentaux. Ensuite l'article *An Introduction to Argumentation Semantics* [2], qui propose des définitions assez approfondies des différentes sémantiques de l'argumentation. Enfin, l'étude des algorithmes pour les sémantiques d'argumentation, présentée dans *Algorithms for Argumentation Semantics : Labeling Attacks as a Generalization of Labeling Arguments* [3], nous a permis d'approfondir nos connaissances sur les algorithmes utilisés dans ce domaine et ainsi implémenter la solution du projet présent.

Ces travaux de référence ont guidé notre démarche et orienté la définition des objectifs ainsi que des fonctionnalités attendues, conformément aux directives de l'énoncé.

2 Objectifs et fonctionnalités attendues du projet

Ce projet a pour objectif la mise en place d'une solution destinée à résoudre des problèmes spécifiques liés aux systèmes d'argumentation abstraits (AF). Un système d'argumentation est défini par un ensemble d'arguments (A) et d'un ensemble de relations d'attaque (R) entre ces arguments. L'enjeu principal est donc de concevoir l'outil capable pour identifier les extensions complètes et stables, tout en évaluant l'appartenance d'un argument données à ces extensions, selon des critères crédibles ou sceptiques.

Le programme est conçu pour lire un système d'argumentation à partir d'un fichier texte formaté, analyser les données et produire des résultats conformes à la sémantique utilisée. Initialement prévu pour fonctionner en mode console, il a été enrichi par l'ajout d'une interface graphique

3 Algorithmes et structures de données

3.1 Algorithmes

Les algorithmes utilisés pour l'implémentation de ce projet sont spécifiquement ceux dédiés à l'énumération des extensions complètes et stables. Ces algorithmes sont issus des travaux présentés dans l'article *"Algorithms for Argumentation Semantics : Labeling Attacks as a Generalization of Labeling Arguments"* [3]. Les deux algorithmes présentés ci-dessous s'appuient sur l'utilisation d'étiquettes qui sont IN, OUT, MUST_OUT, BLANK et UNDEC. MUST_OUT et BLANK, qui n'ont pas été abordés en cours, peuvent être considérés ici comme des détails propres à l'implémentation.

L'étiquette IN identifie les arguments qui peuvent appartenir à l'extension recherchée. L'étiquette OUT identifie un argument qui est attaqué par un argument IN. L'étiquette BLANK est attribuée à tout argument non traité dont l'étiquette finale n'est pas encore déterminée. L'étiquette MUST_OUT identifie les arguments qui attaquent des arguments IN. Enfin, l'étiquette UNDEC désigne les arguments qui pourraient ne pas être inclus dans l'extension, car ils pourraient ne pas être défendus par un argument IN.

— Énumération des extensions stables.

L'algorithme ci-dessous identifie toutes les extensions stables d'un AF (A, R) [3] (p. 642). On remarque l'absence d'utilisation du label UNDEC. La sémantique du label MUST_OUT peut être interprétée comme celle d'un UNDEC mais attaquant un argument étiqueté IN. Cependant, dans une labellisation visant à identifier une extension stable, aucun argument ne devrait rester étiqueté UNDEC à la fin.

Le label MUST_OUT est utilisé pour désigner les arguments attaquant des label IN et susceptibles d'évoluer vers OUT à la fin de la labellisation. Ainsi, un ensemble d'arguments S , étiqueté IN, est considéré comme stable uniquement s'il n'existe aucun argument étiqueté MUST_OUT à l'issue d'une labellisation de tous les Arguments.

$(y, z) \in R$ signifie que y attaque z (ou que z est attaqué par y). On note $\{y\}^-$ l'ensemble des arguments qui attaquent y , et $\{y\}^+$ l'ensemble des arguments attaqués par y .

Le processus commence par l'initialisation des arguments avec un label BLANK, signifiant qu'aucun argument n'a encore été catégorisé. Ensuite, l'algorithme cherche à trouver toutes les extensions stables possibles en étiquetant les arguments de manière appropriée. Tant qu'il existe des arguments non étiquetés, l'algorithme sélectionne un argument y avec un label BLANK. Le critère de sélection dépend de la situation : si l'argument y est BLANK avec ses attaquants OUT ou MUST_OUT, il est sélectionné pour être labellisé IN. Dans le cas contraire, l'algorithme choisit un argument y de manière à maximiser le nombre d'arguments attaqués par y (l'ensemble y^+) qui ne sont pas encore étiquetés OUT.

Une copie de l'état des labels est effectuée pour une première propagation des étiquettes ($Lab' \leftarrow Lab$).

Les éléments de l'ensemble $\{y\}^+$ reçoivent le label OUT, indiquant qu'ils sont exclus de l'extension, car attaqué par y . Les arguments voisins attaquant y correspondant aux éléments de $\{y\}^-$ mais qui ne sont pas encore traités (BLANK) reçoivent le label MUST_OUT, ils doivent être exclus et ne peuvent pas faire partie de l'extension.

Si l'argument z , qui était jusqu'à présent non traité, a tous ses attaquants déjà étiquetés autre que BLANK, cela entraîne la mise à jour du label de y en MUST_OUT. En effet, cela signifie que y devient incompatible avec l'extension en cours. Dans ce cas, on effectue un retour anticipé au début de la boucle.

Une fois que les labels ont été modifiés, l'algorithme procède à une exploration récursive en appelant à nouveau la procédure pour tester de nouvelles configurations.

On s'assure qu'il n'existe aucun argument z dans l'ensemble $\{y\}^-$ qui soit étiqueté BLANK dans l'état Lab . Si tel est le cas, l'argument y est étiqueté MUST_OUT. Sinon, l'état Lab est mis à jour pour être égal à l'état tampon Lab' .

Si tous les arguments attaqués par y ont des labels autres que MUST_OUT, l'extension est considérée comme stable et l'ensemble des arguments étiquetés IN est ajouté à l'ensemble des extensions stables.

L'exécution du bloc suivant l'appel récursif est mise dans la pile d'exécution, et elle sera traitée après tous les appels récursifs.

Algorithm 1 Enumerating all stable extensions of an AF (A, R)

```

1:  $Lab : A \rightarrow \{IN, OUT, MUST\_OUT, BLANK\}$ ;  $Lab \leftarrow \emptyset$ 
2: for all  $x \in A$  do
3:    $Lab \leftarrow Lab \cup \{(x, BLANK)\}$ 
4: end for
5:  $Estable \subseteq 2^A$ ;  $Estable \leftarrow \emptyset$ 
6: call find-stable-extensions( $Lab$ )
7: report  $Estable$  is the set of all stable extensions
8: procedure find-stable-extensions( $Lab$ )
9: while there exists  $y \in A$  such that  $Lab(y) = BLANK$  do
10:  if there exists  $y$  such that  $Lab(y) = BLANK$  and  $\forall z \in \{y\}^- : Lab(z) \in$ 
     $\{OUT, MUST\_OUT\}$  then
11:    select  $y$  such that  $Lab(y) = BLANK$ 
12:  else
13:    select  $y$  such that  $Lab(y) = BLANK$  and  $\forall z : Lab(z) = BLANK$ ,
14:     $|\{x : x \in y^+ \wedge Lab(x) \neq OUT\}| \geq |\{x : x \in z^+ \wedge Lab(x) \neq OUT\}|$ 
15:  end if
16:   $Lab' \leftarrow Lab$ 
17:   $Lab'(y) \leftarrow IN$ 
18:  for all  $z \in y^+$  do
19:     $Lab'(z) \leftarrow OUT$ 
20:  end for
21:  for all  $z \in y^-$  do
22:    if  $Lab'(z) = BLANK$  then
23:       $Lab'(z) \leftarrow MUST\_OUT$ 
24:    end if
25:    if for all  $w \in z^-$ ,  $Lab'(w) \neq BLANK$  then
26:       $Lab(y) \leftarrow MUST\_OUT$ 
27:    end if
28:  end for
29:  goto line 7
30:  call find-stable-extensions( $Lab'$ )
31:  if there exists  $z \in y^-$  such that  $Lab(z) = BLANK$  then
32:     $Lab(y) \leftarrow MUST\_OUT$ 
33:  else
34:     $Lab \leftarrow Lab'$ 
35:  end if
36:  if for all  $x$ ,  $Lab(x) \neq MUST\_OUT$  then
37:     $S \leftarrow \{x : Lab(x) = IN\}$ 
38:     $Estable \leftarrow Estable \cup \{S\}$ 
39:  end if
40: end while

```

— Énumération des extensions complètes.

L'algorithme ci dessous de identifie toutes les extensions complètes d'un AF (A,R) [3] (p.644).

L'algorithme 2 repose sur le même principe que le précédent avec quelques petits changements pour satisfaire les conditions d'extension stable. On part toujours sur la base d'arguments tous étiquetés à blanc ; le critère de selection d'argument pour la propagation des labels reste le même. On remarque l'utilisation du label UNDEC ici pour indiquer une indécision, alors que cette situation avait immédiatement été rejetée dans l'algorithme 1 en tant qu'incompatibilité, en la plaçant sous le label MUST_OUT.

on détermine si un ensemble S (les éléments dont le label est IN) est une extension complète en vérifiant deux conditions.

1. Tout d'abord, garantir qu' aucun élément x de A n'est marqué comme MUST_OUT.
2. Ensuite, on vérifie qu'il n'existe pas un élément z qui reste indéterminé (UNDEC ou BLANK) et qui aurait tous ses attaquants (les éléments de $\{z\}^-$) déjà marqués OUT.

Algorithm 2 Enumerating all complete extensions of an AF (A, R)

```

1:  $Lab : A \rightarrow \{IN, OUT, MUST\_OUT, UNDEC, BLANK\}; Lab \leftarrow \emptyset$ 
2: for all  $x \in A$  do
3:    $Lab \leftarrow Lab \cup \{(x, BLANK)\}$ 
4: end for
5:  $E_{complete} \subseteq 2^A; E_{complete} \leftarrow \emptyset$ 
6: call find-complete-extensions( $Lab$ )
7: report  $E_{complete}$  is the set of all complete extensions
8: procedure find-complete-extensions( $Lab$ )
9: if there exists  $y \in A$  such that  $Lab(y) = MUST\_OUT$  then
10:  return
11: end if
12: if for all  $x \in A, Lab(x) \in \{UNDEC, BLANK\} \implies \forall z \in \{x\}^-, Lab(z) = OUT$  then
13:   $S \leftarrow \{w \in A \mid Lab(w) = IN\}$ 
14:   $E_{complete} \leftarrow E_{complete} \cup \{S\}$ 
15: end if
16: while there exists  $y \in A$  such that  $Lab(y) = BLANK$  do
17:  if  $\forall z \in \{y\}^-, Lab(z) \in \{OUT, MUST\_OUT\}$  then
18:    select  $y$  such that  $Lab(y) = BLANK$ 
19:  else
20:    select  $y$  such that  $Lab(y) = BLANK$  and  $\forall z : Lab(z) = BLANK,$ 
21:     $|\{x \mid x \in y^+ \wedge Lab(x) \neq OUT\}| \geq |\{x \mid x \in z^+ \wedge Lab(x) \neq OUT\}|$ 
22:  end if
23:   $Lab' \leftarrow Lab$ 
24:   $Lab'(y) \leftarrow IN$ 
25:  for all  $z \in y^+$  do
26:     $Lab'(z) \leftarrow OUT$ 
27:  end for
28:  for all  $z \in y^-$  do
29:    if  $Lab'(z) \in \{UNDEC, BLANK\}$  then
30:       $Lab'(z) \leftarrow MUST\_OUT$ 
31:    end if
32:    if for all  $w \in z^-, Lab'(w) \neq BLANK$  then
33:       $Lab(y) \leftarrow UNDEC$ 
34:    end if
35:  end for
36:  goto line 7
37:  call find-complete-extensions( $Lab'$ )
38:  if there exists  $z \in y^-$  such that  $Lab(z) \in \{BLANK, UNDEC\}$  then
39:     $Lab(y) \leftarrow UNDEC$ 
40:  else
41:     $Lab \leftarrow Lab'$ 
42:  end if
43: end while

```

3.2 structures de données

La manipulation des données dans le projet repose essentiellement sur l'utilisation de listes et de dictionnaires pour la représentation des arguments et des relations d'attaque.

- Les arguments sont stockés dans une liste ou un ensemble. Par exemple :

$$arguments = \{A', B', C'\}$$

où A , B et C représentent les arguments.

- Les attaques sont représentées par un dictionnaire, chaque clé correspondant à un argument et la valeur étant l'ensemble des arguments qu'il attaque. Par exemple :

$$attacks = \{A' : \{B', C'\}, B' : \{C'\}\}$$

Ici, A attaque B et C , tandis que B attaque C .

4 Implémentation et tests

4.1 Implémentation

La proposition de solution a été faite **Python**. Nous avons choisi Python pour sa simplicité d'implémentation et sa flexibilité pour la manipulations des structures de données telles que les listes et les dictionnaires que nous utilisons.

4.1.1 Architecture

L'architecture du projet repose sur l'utilisation de classes et de méthodes afin de faciliter le partage des données et des fonctions entre différents composants répartis sur plusieurs fichiers. Les principales parties de l'architecture sont décrites ci-dessous :

- **main.py** : Ce fichier sert de point d'entrée principal pour l'exécution de la solution. L'utilisateur peut lancer le programme en mode console avec les paramètres suivants :
 - **-P** : pour spécifier la sémantique de calcul.
 - **-f** : pour indiquer le fichier contenant l'Argumentation Framework (AF).
 - **-a** : (optionnel) pour spécifier les arguments, en fonction de la sémantique choisie.

Ce fichier appelle les autres fichiers pour le traitement des données et l'exécution des algorithmes, parmi lesquels on a :

- **utilities.py** : Ce fichier contient des classes et des méthodes utilitaires, comme la classe `file_reader`, qui permet de lire les fichiers contenant l'AF. Ces méthodes facilitent la gestion des fichiers et la manipulation des données avant leur traitement.
- **controller.py** : Ce fichier contient les méthodes responsables de la gestion des flux de données. Il fait appel aux méthodes de calcul spécifiques en fonction des paramètres fournis par l'utilisateur, en organisant et en contrôlant l'exécution des algorithmes.
- **computing_methods.py** : Ce fichier contient l'implémentation des algorithmes d'énumération des extensions complètes et stables. Afin de garantir une bonne lisibilité du code, les parties répétitives dans les algorithmes ont été extraites et

regroupées dans des méthodes uniques. Par exemple, la méthode `select_argument` est utilisée pour sélectionner un argument dans les différents algorithmes afin de réduire la duplication du code.

- **af_visualizer.py** : Ce fichier gère l'interface graphique de l'application. Il est basé sur les mêmes méthodes des fichiers précédemment mentionnés, permettant une interaction visuelle avec l'utilisateur pour soumettre des fichiers et afficher les résultats sous une forme graphique.

4.2 Tests et validation

Pour vérifier l'exactitude de notre implémentation des algorithmes d'énumération des extensions stables et complètes, nous avons réalisé des tests sur plusieurs frameworks argumentatifs. Dans le tableau dessous nos recensons les résultats de quelques tests. Les fichiers de test, au format `.apx`, se trouvent dans un répertoire "tests" au sein du projet. Ces fichiers sont reçus en entrée, car ils contiennent la description des systèmes d'argumentation abstraits (AF).

Commande exécutée	Résultats (console et interface graphique)
<pre>python .\main.py -p SE-CO -f .\tests\test_af2.apx</pre>	<pre>PS C:\Users\H P\git\argumentation> python .\main.py -p SE-CO -f .\tests\test_af2.apx Extension complète: {'B'} =====Toutes les extensions complètes===== [set(), {'A'}, {'A', 'D'}, {'A', 'E'}, {'B'}, {'D', 'B'}, {'E', 'B'}, {'E'}]</pre> <pre>graph LR A((A)) --> B((B)) B --> C((C)) C --> D((D)) D --> E((E)) A --> D</pre>

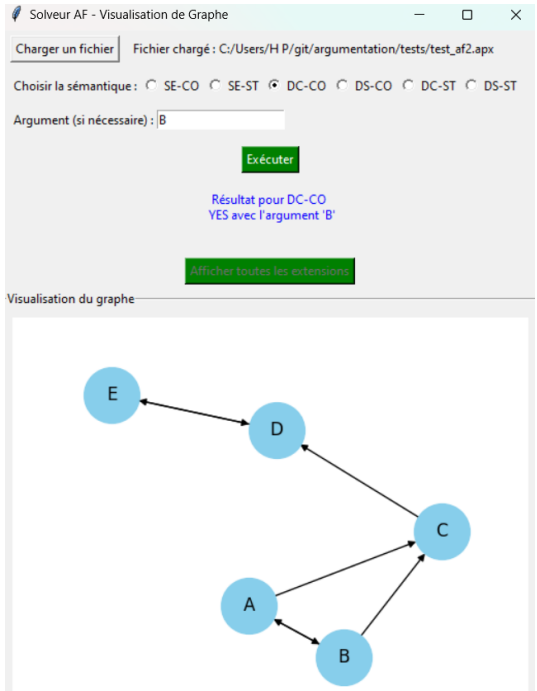
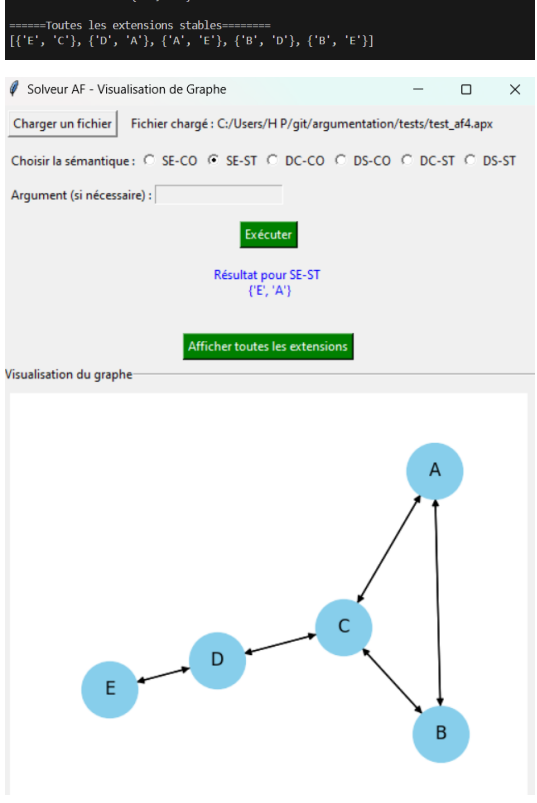
Commande exécutée	Résultats (console et interface graphique)
<pre>python .\main.py -p DC-CO -f .\tests\test_af2.apx -a 'B'</pre>	<p>PS C:\Users\H P\git\argumentation> python .\main.py -p DC-CO -f .\tests\test_af2.apx -a 'B'</p> <p>Yes</p> <p>Solveur AF - Visualisation de Graphe</p> <p>Charger un fichier Fichier chargé : C:/Users/H P/git/argumentation/tests/test_af2.apx</p> <p>Choisir la sémantique : <input type="radio"/> SE-CO <input type="radio"/> SE-ST <input checked="" type="radio"/> DC-CO <input type="radio"/> DS-CO <input type="radio"/> DC-ST <input type="radio"/> DS-ST</p> <p>Argument (si nécessaire) : B</p> <p>Exécuter</p> <p>Résultat pour DC-CO YES avec l'argument 'B'</p> <p>Afficher toutes les extensions</p> <p>Visualisation du graphe</p> 
<pre>python .\main.py -p SE-ST -f .\tests\test_af4.apx</pre>	<p>PS C:\Users\H P\git\argumentation> python .\main.py -p SE-ST -f .\tests\test_af4.apx</p> <p>Extension stable: {'B', 'E'}</p> <p>====Toutes les extensions stables=====</p> <p>[{'E', 'C'}, {'D', 'A'}, {'A', 'E'}, {'B', 'D'}, {'B', 'E'}]</p> <p>Solveur AF - Visualisation de Graphe</p> <p>Charger un fichier Fichier chargé : C:/Users/H P/git/argumentation/tests/test_af4.apx</p> <p>Choisir la sémantique : <input type="radio"/> SE-CO <input checked="" type="radio"/> SE-ST <input type="radio"/> DC-CO <input type="radio"/> DS-CO <input type="radio"/> DC-ST <input type="radio"/> DS-ST</p> <p>Argument (si nécessaire) :</p> <p>Exécuter</p> <p>Résultat pour SE-ST {'E', 'A'}</p> <p>Afficher toutes les extensions</p> <p>Visualisation du graphe</p> 

TABLE 1 – La commande pour lancer l’interface graphique est `python af_visualizer.py`.

Les résultats de nos tests confirment le bon fonctionnement de notre implémentation des algorithmes. Les extensions calculées correspondent aux attentes théoriques pour chaque cas testé, validant ainsi l’exactitude de notre implémentation.

4.2.1 Remarque sur les fichiers de test.

Dans le cadre des tests réalisés, nous avons constaté une petite disparité dans les formats des fichiers fournis. En effet, alors que les spécifications de l’énoncé imposaient que chaque ligne décrivant un argument ou une attaque se termine par un point (par exemple, `arg(a).` ou `att(a,b).`), certains fichiers de test présentaient des lignes sans point final, comme `arg(a)` ou `att(a,b)`. Afin de garantir une compatibilité avec ces fichiers, notre implémentation a été adaptée pour gérer les deux formats.

5 Conclusion

En conclusion, ce projet a été l’occasion d’élargir nos connaissances sur les systèmes d’argumentation abstraits, à travers une revue de littérature et l’implémentation pratique d’algorithmes d’énumération d’extensions.

L’intégration de ces algorithmes dans un outil fonctionnel a permis de concrétiser notre apprentissage. Ce travail constitue une contribution à l’application des théories de l’argumentation et ouvre la voie à des perspectives d’amélioration.

Références

- [1] Élise Bonzon. Argumentation computationnelle. https://moodle.u-paris.fr/pluginfile.php/47364/mod_resource/content/19/03_RCR_argumentation.pdf, 2024. Cours donné à l’université Paris Cité, [Novembre].
- [2] Pietro Baroni, Martin Caminada, and Massimiliano Giacomin. An introduction to argumentation semantics. *Knowledge Eng. Review*, 26 :365–410, 12 2011.
- [3] Samer Nofal, Katie Atkinson, and Paul E. Dunne. Algorithms for argumentation semantics : Labeling attacks as a generalization of labeling arguments. *J. Artif. Intell. Res.*, 49 :635–668, 2014.