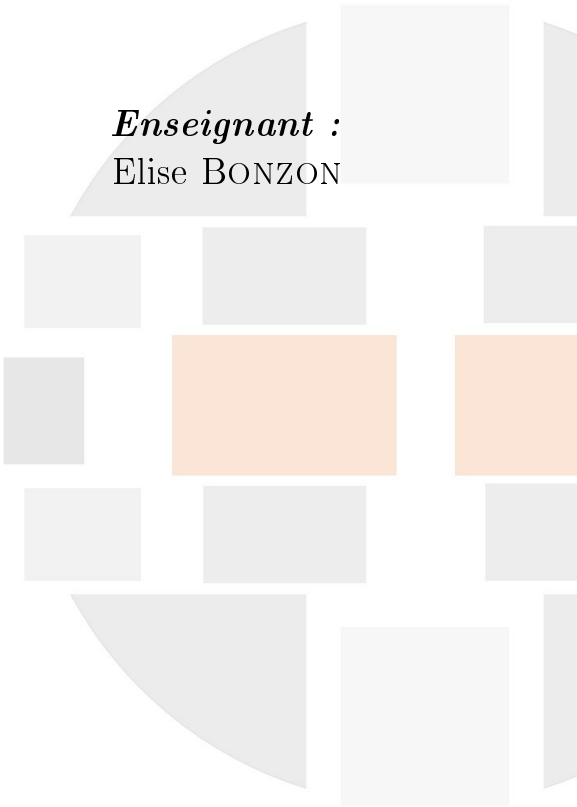
Université Paris Cité

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

Résumé

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

Grâce à 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, nous avons développé une interface graphique conviviale permettant de visualiser les systèmes d'argumentation, leurs arguments, et les relations entre eux, offrant ainsi une meilleure interaction utilisateur et facilitant l'analyse des résultats. Ce rapport détaille les méthodologies employées, les choix techniques effectués et les résultats obtenus, illustrant ainsi notre contribution à la mise en œuvre pratique des concepts théoriques de l'argumentation.

1 Introduction

Pour la réalisation de ce projet, une revue de littérature s'est naturellement imposée. Parmi les ressources pertinentes, nous avons trouvé plusieurs travaux particulièrement intéressants.

En premier lieu, nous avons étudié le document sur l'argumentation computationnelle, notamment le cours d'Élise Bonzon, qui nous a servi de base pour comprendre les concepts fondamentaux. Nous avons également exploré l'article *An Introduction to Argumentation Semantics*, 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*, nous a permis d'approfondir nos connaissances sur les algorithmes utilisés dans ce domaine et ainsi implémenter la solution du projet présent.

Cette exploration nous a permis de définir les objectifs du projet, présentés dans la section suivante.

2 Objectifs et fonctionnalités attendues du projet

Ce projet vise à développer un outil capable de résoudre des problèmes spécifiques liés aux systèmes d'argumentation abstraits (AF). Un système d'argumentation est défini par un ensemble d'arguments (A) et une relation d'attaque (R) entre ces arguments. L'objectif est de fournir une solution capable d'identifier les extensions complètes et stables, et évaluer l'appartenance d'un argument à ces extensions selon des critères crédibles ou sceptiques.

Le programme lit un système d'argumentation à partir d'un fichier texte formaté, analyse les données, et produit des résultats conformes selon la sémantique utilisée.

Le projet propose également une interface visuelle representant les arguments, leurs relations sous forme de graphes et ainsi que les résultats des analyses pour faciliter l'interaction utilisateur et l'interprétation des résultats.

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"* [1]. Pour les deux algorithmes qui seront présentés dessous, tournera autour autour de l'utilisation d'étiquettes qui sont IN, OUT, MUST_OUT, BLANK et UNDEC. MUST_OUT et BLANK n'ayant pas été abordé en cours, sont en effet ici des détails d'implémentation.

L'étiquette **IN** identifie les arguments qui pourraient 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), comme décrit dans [] (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** qui attaque un argument étiqueté **IN**. Cependant, dans une labellisation visant à identifier une extension stable, aucun argument ne devrait rester étiqueté **UNDEC** à la fin.

C'est pourquoi le label **MUST_OUT** est utilisé à la ligne 34 pour désigner les arguments 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 de la labellisation.

$(y, z) \in R$ 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 n'a pas d'attaquants non étiquetés (c'est-à-dire que tous ses attaquants ont déjà été étiquetés comme OUT ou), il est selectionné, ce qui signifie qu'il fait partie de l'extension. Dans le cas contraire, l'algorithme choisit un argument y de manière à maximiser le nombre d'arguments attaqués par y (y^+) qui ne sont pas encore étiquetés OUT.

Une copie complète de l'état des labels est effectuée pour la propagation des étiquettes (Lab' <- Lab)

Une fois un argument sélectionné et étiqueté IN, l'algorithme modifie les labels des arguments attaqués par y. Tous les arguments attaqués par y (les voisins de y dans l'ensemble $\{y\}^+$) reçoivent le label OUT, indiquant qu'ils sont exclus de l'extension. Les arguments voisins attaquant $y \in \{y\}^-$ mais qui ne sont pas encore étiquetés (BLANK) reçoivent le label MUST_OUT, signifiant qu'ils doivent être exclus de manière forcée et ne peuvent pas faire partie de l'extension.

Si tous les attaquants $\{z\}^-$ d'un argument z sont déjà étiquetés autre que (BLANK), cela entraîne la mise à jour du label de y en MUST_OUT l'état des labels original, car l'argument y devient ainsi incompatible avec l'extension en cours et on execute un retour anticipé au début de la boucle while.

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

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

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

elle sera traitée après tous les appels récursifs.

— Énumération des extensions complètes. L'algorithme ci dessous de identifie toutes les extensions complètes d'un AF (A,R) correspond[1] (p.644).

L'algorithme est repose sur le même principe que le préccédent avec quelques petits changements pour satisfaire les conditions d'extension stable. On part toujours sur la base d'arguments tous etiquettés à blanc ; le critère de selection d'argument pour la propagation des labels restant le même. On constate l'utilisation du label UNDEC ici lorsqu'une incompatibilité est levé en lieu et place de MUST_OUT dans l'algorithm 1.

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 comme OUT.

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

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