MATHÉMATIQUES ET INFORMATIQUE
Sciences
Université Paris Cité

Compte Rendu - Projet

Représentation des connaissances et raisonnement

Thomas GOURMELEN - Rayane KHATIM

Décembre 2025

Contents

1 Introduction

1.1 Contexte

Un **système d'argumentation** (**AS**) est défini par $\mathbf{F} = \langle A, R \rangle$, où :

- $\mathbf{A}$ est un ensemble d'arguments abstraits,
- $\mathbf{R} \subseteq \mathbf{A} \times \mathbf{A}$ est la relation d'attaque entre ces arguments.

Par exemple, la Figure 1 représente graphiquement le système $\mathbf{F} = \langle \mathbf{A}, \mathbf{R} \rangle$ avec $\mathbf{A} = \{a, b, c, d\}$ et $\mathbf{R} = \{(a, b), (b, c), (b, d)\}$.

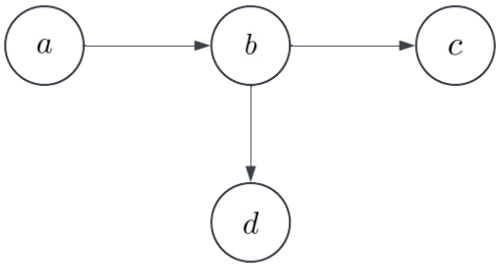

Figure 1: AF de la Figure 1

Étant donné un système d'argumentation $\mathbf{F} = \langle \mathbf{A}, \mathbf{R} \rangle$, on rappelle les notions suivantes :

- $S \subseteq \mathbf{A}$ est **sans conflit** ssi il n'existe pas $a, b \in S$ tels que $(a, b) \in \mathbf{R}$,
- S est un **ensemble admissible** ssi S est sans conflit et défend tous ses arguments,
- S est une **extension préférée** ssi S est maximal pour l'inclusion parmi les extensions admissibles,
- S est une **extension stable** ssi S est sans conflit et attaque tout argument de $\mathbf{A}$ n'appartenant pas à S.

Ces notions permettent de formaliser et d'analyser l'acceptabilité des arguments selon différentes sémantiques.

1.2 Objectif du projet

L'objectif principal de ce projet est de développer un programme permettant d'analyser des systèmes d'argumentation abstraits (AS) et de déterminer l'acceptabilité des arguments selon différentes sémantiques.

Le programme prend en entrée un fichier au format `.apx` décrivant un système d'argumentation à l'aide d'arguments et de relations d'attaque entre les arguments. À partir de cette description, il calcule les différentes extensions associées au système, notamment les extensions admissibles, préférées et stables.

Une fois ces extensions calculées, le programme est capable de répondre à des requêtes de type **VE** (vérification d'extension), **DC** (acceptation crédulée) et **DS** (acceptation sceptique), en fonction de la sémantique choisie par l'utilisateur.

Ce projet a pour but de mettre en pratique les notions vues en cours sur l'analyse de systèmes d'argumentation de taille modérée.

2 Représentation du système d'argumentation

2.1 Modèle théorique

Cette section rappelle les principales notions des systèmes d'argumentation utilisées dans le projet. Ces éléments ont été introduits dans le contexte, mais sont reformulés ici afin de servir de base aux choix de représentation et aux algorithmes implémentés.

Un système d'argumentation est défini comme un couple $\mathbf{F} = \langle A, R \rangle$ où :

- **A** est un ensemble fini d'arguments abstraits.
- **R** est un ensemble de relations d'attaque entre ces arguments.

La relation d'attaque **R** est composée de couples (a, b), où a et b sont des arguments appartenant à l'ensemble **A**. Un couple (a, b) signifie que l'argument a attaque l'argument b. Ces attaques représentent des conflits entre arguments.

Les systèmes d'argumentation peuvent être représentés sous la forme de graphes orientés, où les sommets représentent les arguments et les arcs représentent les attaques. Cette représentation graphique permet de mieux visualiser les relations entre arguments et de comprendre plus facilement la structure du système.

Avant de définir les différentes sémantiques, certaines notions fondamentales sur les ensembles d'arguments sont nécessaires. Trois notions principales sont utilisées dans ce projet :

- Un ensemble **sans conflit**, c'est-à-dire un ensemble d'arguments qui ne s'attaquent pas entre eux.
- La **défense d'un argument** par un ensemble, lorsqu'un ensemble d'arguments contre-attaque tous les attaquants de cet argument.
- La **défense d'un ensemble par lui-même**, lorsque chaque argument de l'ensemble est défendu par l'ensemble.

Ces notions permettent ensuite de définir différentes sémantiques d'argumentation, telles que les sémantiques **admissible**, **préférée** et **stable**, qui seront utilisées dans ce projet afin d'analyser l'acceptabilité des arguments.

2.2 Structures de données utilisées

Dans ce projet, nous avons utilisé des structures de données simples et adaptées pour représenter les systèmes d'argumentation. Ces choix facilitent la manipulation des données et sont bien adaptés au langage utilisé, c'est-à-dire Python.

Représentation des arguments

Les arguments du système d'argumentation sont stockés dans un ensemble Python (`set[str]`). Ce choix présente plusieurs avantages :

- Il évite la présence de doublons.
- Il permet de tester facilement si un argument appartient à un ensemble.

Représentation des attaques

La relation d'attaque est représentée par un ensemble de couples (`set[(str, str)]`). Chaque couple (a, b) signifie que l'argument a attaque l'argument b. Cette structure correspond directement à la définition de la relation d'attaque et permet également d'éviter les doublons.

Ensembles d'arguments et extensions

Les ensembles d'arguments, comme les ensembles sans conflit ou les extensions, sont aussi représentés par des ensembles Python (`set[str]`). Cela facilite les opérations sur les ensembles, notamment les tests d'inclusion et les comparaisons entre ensembles.

Les différentes extensions (admissibles, préférées et stables) sont quant à elles stockées dans des listes (`list[set[str]]`). L'utilisation de listes permet de parcourir facilement les extensions afin de répondre aux différentes requêtes du projet.

Classe AS

La structure globale du système d'argumentation est regroupée dans une classe `AS`. Cette classe contient :

- L'ensemble des arguments.
- La relation d'attaque associée.

Elle fournit également des méthodes simples permettant d'accéder aux informations du système, comme les arguments qui attaquent un argument donné ou les arguments attaqués par un autre. Cette organisation permet de centraliser la représentation du système d'argumentation et de faciliter son utilisation dans les différents algorithmes du projet.

Extrait de code Le code suivant présente un extrait de la classe `AS`, montrant la structure principale du système d'argumentation ainsi qu'une méthode permettant d'obtenir les attaquants d'un argument.

```python
class AS:
    def __init__(self, A: set[str], R: set[tuple[str, str]]):
        """
        Initialise un AS.
        Args:
            - A: ensemble des arguments.
            - R: ensemble des attaques (x, y) avec x attaque y.
        Returns:
            None
        """
        self.A = A
        self.R = R

    def attackers_of(self, a: str) -> set[str]:
        """
        Donne les attaquants de a.
        Args:
            - a: argument cible.
        Returns:
            L'ensemble contenant tous les attaquants de a.
        Raises:
            ValueError: si a n'est pas dans A.
        """
        if a not in self.A:
            raise ValueError(f"L'argument {a} n'est pas dans les arguments.")

        attackers = set()
        for r in self.R:
            if r[1] == a:
                attackers.add(r[0])

        return attackers
```

3 Parsing et validation des entrées

3.1 Parsing du fichier .apx

Le programme prend en entrée un fichier texte au format `.apx`. Ce fichier décrit un système d'argumentation à l'aide de deux types de lignes :

- `arg(x).` pour déclarer un argument.
- `att(x,y).` pour déclarer une attaque entre deux arguments.

Lecture du fichier Le fichier est lu ligne par ligne afin de traiter chaque instruction une à une. Lors de la lecture, les espaces inutiles ainsi que les retours à la ligne sont supprimés pour éviter les erreurs. Les lignes vides sont ignorées, ce qui permet de ne conserver que les lignes réellement utiles à la construction du système d'argumentation.

Traitement des arguments Lorsqu'une ligne correspondant à la déclaration d'un argument est rencontrée, c'est-à-dire de la forme `arg(x).`, le nom de l'argument est extrait. Cet argument est ensuite ajouté à l'ensemble des arguments du système d'argumentation.

Afin d'éviter les problèmes, tous les noms des arguments sont normalisés, avec conversion en minuscules et suppression des espaces et retours à la ligne inutiles, avant d'être stockés.

Traitement des attaques Lorsqu'une ligne correspondant à la déclaration d'une attaque est rencontrée, c'est-à-dire de la forme `att(x,y).`, les deux arguments x et y sont extraits. Une vérification est ensuite faite afin de s'assurer que ces deux arguments ont bien été déclarés avant dans le système d'argumentation.

Si l'un des arguments n'a pas été déclaré, une erreur est levée et le programme s'arrête. Cette vérification permet d'éviter la création d'attaques incorrectes.

Gestion des erreurs Des vérifications sont effectuées tout au long du parsing afin de détecter les erreurs dans le fichier d'entrée. Une erreur est levée lorsqu'une attaque utilise un argument qui n'a pas été déclaré. Les fichiers mal formés sont également détectés dès la lecture.

Extrait de code Le code suivant présente un extrait de la fonction utilisée pour parser les fichiers `.apx`. Il illustre la lecture du fichier, le traitement des arguments et des attaques, ainsi que la gestion des erreurs.

```python
def parse_apx(path: str) -> tuple[set[str], set[tuple[str, str]]]:
    A = set() # Ensemble des arguments.
    R = set() # Ensemble des relations d'attaque.

    with open(path, 'r', encoding='utf-8') as fichier:
        for ligne in fichier:
            ligne = ligne.strip()

            if ligne.startswith("arg("):
                A.add(ligne[4:-2].lower())
            elif ligne.startswith("att("):
                args = ligne[4:-2]
                args = args.split(',')
                x = args[0].lower()
                y = args[1].lower()
                if x not in A or y not in A:
                    raise ValueError(f"Error: Argument utilise dans une attaque avant d etre
                        declare.")
                else :
                    R.add((x, y))
    return A, R
```

3.2 Gestion de la ligne de commande

Le programme est exécuté en ligne de commande et utilise plusieurs options permettant de préciser la requête à effectuer, le fichier à analyser et le ou les arguments. La gestion de la ligne de commande permet de vérifier que les informations fournies par l'utilisateur sont correctes avant l'exécution des algorithmes.

Options disponibles Le programme attend trois options obligatoires :

- `-p` : Le type de problème à résoudre (VE, DC ou DS) ainsi que la sémantique choisie (PR ou ST).
- `-f` : Le chemin du fichier `.apx` décrivant le système d'argumentation.
- `-a` : Le(s) argument(s) de la requête, sous la forme d'un argument unique ou d'une liste d'arguments séparés par des virgules.

Validation des entrées Avant de lancer les calculs, les arguments fournis par l'utilisateur sont vérifiés. Le type de requête détermine la forme attendue pour l'option `-a`. Pour les requêtes de type VE, plusieurs arguments peuvent être fournis. Pour les requêtes de type DC ou DS, un seul argument est attendu.

Les noms des arguments sont normalisés tout comme pour la lecture de fichier `.apx`. Les arguments sont mis en minuscules et les espaces inutiles sont supprimés.

Gestion des erreurs Si une option est manquante ou si les arguments fournis ne respectent pas le format attendu, une erreur est levée. Le message d'erreur est affiché sur la sortie d'erreur standard, puis le programme s'arrête.

Extrait de code Le code suivant présente un extrait de la gestion de la ligne de commande. Il montre comment l'option `-a` est validée en fonction du type de requête (VE ou DC/DS).

```
def parse_and_validate_query(problem: str, raw_a: str):
    """
    Parse et valide l'argument -a en fonction du probleme.
    Args:
        - problem: type de probleme.
        - raw_a: valeur brute de l'option -a (chaine de caracteres).
    Returns:
        - Un ensemble d'arguments (set[str]) pour VE.
        - Un argument unique (str) pour DC / DS.
    Raises: ValueError: si le format de -a est invalide.
    """
    raw_a = raw_a.strip().lower()
    tokens = []
    # DC / DS: un seul argument:
    if not problem.startswith("VE-"):
        if "," in raw_a:
            raise ValueError("Error: -a doit contenir un seul argument.")
        if raw_a == "":
            raise ValueError("Error: -a ne peut pas etre vide.")
        return raw_a
    # VE: liste d'arguments:
    for t in raw_a.split(","):
        t = t.strip()
        if t != "":
            tokens.append(t)
    if not tokens:
        raise ValueError("Error: -a doit contenir au moins un argument.")
    return set(tokens) # On retourne le(s) argument(s) sous forme d'ensemble.
```

4 Algorithmes implémentés

Cette partie présente les différents algorithmes utilisés dans le projet afin d'analyser les systèmes d'argumentation et de répondre aux différentes requêtes. Les algorithmes implémentés reposent sur les définitions vues en cours.

Vue d'ensemble :

$$\texttt{.apx} \rightarrow (A, R) \rightarrow \text{AS} \rightarrow \{\text{extensions}\} \rightarrow \text{requêtes (VE/DC/DS)}$$

4.1 Fonctions de base

Nous avons implémenté plusieurs fonctions afin de manipuler les systèmes d'argumentation et les ensembles d'arguments. Ces fonctions sont utilisées dans les algorithmes qui permettent de réaliser les différents tests nécessaires.

Vérification de conflit

$$S \text{ est sans conflit} \iff \nexists a, b \in S \text{ tels que } (a, b) \in R$$

Une première fonction permet de vérifier si un ensemble d'arguments est sans conflit. Un ensemble est considéré comme sans conflit lorsque :

- Aucun argument de l'ensemble n'attaque un autre argument de l'ensemble.
- Il n'existe donc aucune attaque entre deux arguments du même ensemble.

Cette vérification est systématiquement réalisée avant le calcul des différentes extensions.

Défense d'un argument

$$S \text{ défend } a \iff \forall x \left((x, a) \in R \Rightarrow \exists y \in S,\ (y, x) \in R\right)$$

Une autre fonction permet de vérifier si un argument est défendu par un ensemble d'arguments. Un argument est considéré comme défendu lorsque :

- Tous ses attaquants sont identifiés.
- Chacun de ces attaquants est attaqué par au moins un argument de l'ensemble.

Cette notion est essentielle pour déterminer l'acceptabilité des arguments.

Admissibilité d'un ensemble

$$S \text{ est admissible} \iff S \text{ est sans conflit et } \forall a \in S,\ S \text{ défend } a$$

À partir de ces deux notions, il est possible de vérifier si un ensemble d'arguments est admissible, en s'assurant qu'il est **sans conflit** et qu'il **défend tous ses arguments**.

4.2 Extensions admissibles

Entrée : $F = \langle A, R \rangle$
Idée : tester tous les sous-ensembles $S \subseteq A$
Sortie : $\{S \mid S \text{ est admissible}\}$

Chaque sous-ensemble est ensuite analysé afin de vérifier s'il respecte les conditions d'admissibilité. Un ensemble est conservé comme extension admissible s'il respecte les deux conditions suivantes :

- L'ensemble est **sans conflit**.
- L'ensemble **défend tous ses arguments**.

Cet algorithme repose sur une approche exhaustive. Il permet de s'assurer que toutes les extensions admissibles du système sont bien prises en compte.

Extrait de code Le code suivant illustre le principe de calcul des extensions admissibles, basé sur une exploration complète des sous-ensembles d'arguments.

```
def admissible_extensions(systeme_argumentation: AS) -> list[set[str]]:
    """
    Donne toutes les extensions admissibles.
    Args:
        - systeme_argumentation: systeme d'argumentation <A, R>.
    Returns:
        - Liste des ensembles admissibles.
    """
    res = []
    sous_ensembles = all_subsets(systeme_argumentation.A)
    for se in sous_ensembles: # On boucle sur chaque sous-ensemble.
        if is_admissible(systeme_argumentation, se): # On teste si le sous-ensemble est
            admissible.
            res.append(se) # Si tel est le cas on le rajoute a la liste des extensions
                admissibles.
    return res
```

4.3 Extensions préférées

Entrée : $F = \langle A, R \rangle$
Idée : calculer les ensembles admissibles, puis garder ceux maximaux pour l'inclusion
Sortie : $\{ S \mid S$ est une extension préférée de $F \}$

Pour calculer les extensions préférées, le programme commence par calculer toutes les extensions admissibles du système d'argumentation. Ces extensions servent ensuite de base pour trouver les extensions préférées.

Chaque extension admissible est comparée aux autres pour vérifier si elle est déjà incluse dans une autre extension admissible ou pas. Une extension est retenue comme extension préférée si :

- Elle est admissible,
- Elle n'est pas incluse dans une autre extension admissible.

Cet algorithme permet d'identifier toutes les extensions préférées du système.

4.4 Extensions stables

Entrée : $F = \langle A, R \rangle$
Idée : tester tous les sous-ensembles $S \subseteq A$ et conserver ceux qui sont sans conflit et qui attaquent tous les arguments n'appartenant pas à S
Sortie : $\{ S \mid S$ est une extension stable de $F \}$

Pour calculer les extensions stables, le programme génère l'ensemble des sous-ensembles possibles d'arguments du système d'argumentation. Chaque sous-ensemble est ensuite analysé.

Un ensemble est conservé comme extension stable s'il respecte les deux conditions suivantes :

- L'ensemble est sans conflit,
- L'ensemble attaque tous les arguments qui ne lui appartiennent pas.

Cet algorithme exhaustif permet de déterminer toutes les extensions stables du système, lorsqu'elles existent.

4.5 Résolution des requêtes VE / DC / DS

Une fois les extensions calculées, le programme est capable de répondre aux requêtes définies dans le sujet. Ces requêtes permettent d'analyser la validité des arguments selon différentes sémantiques.

Requête	Sens	Condition testée
VE	Vérifier une extension	$S \in \text{Ext}$
DC	Acceptation crédulée	$\exists S \in \text{Ext},\ a \in S$
DS	Acceptation sceptique	$\forall S \in \text{Ext},\ a \in S$

Ici Ext est la liste des extensions préférées ou stables selon PR/ST.

Ces requêtes sont résolues en parcourant les extensions calculées et en appliquant directement les algorithmes définis plus haut.

5 Tests réalisés et résultats

Afin de vérifier le bon fonctionnement du programme, plusieurs tests ont été réalisés à l'aide de l'ensemble des systèmes d'argumentation fournis dans les fichiers `.apx` et de différentes requêtes. Les tests ont été effectués en utilisant la ligne de commande, en comparant les résultats obtenus avec les résultats théoriques attendus.

Les figures suivantes représentent les cinq systèmes d'argumentation utilisés lors des tests.

5.1 Systèmes d'argumentation utilisés

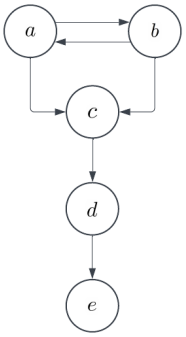

Figure 2: AF1

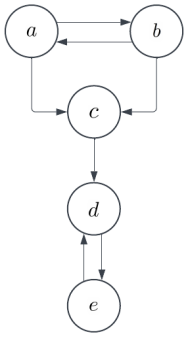

Figure 3: AF2

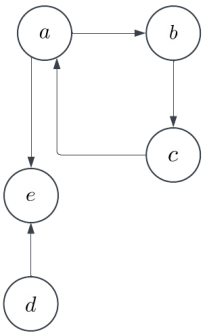

Figure 4: AF3

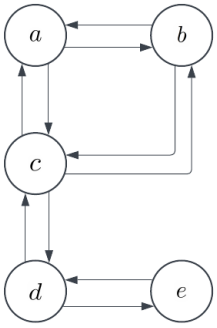

Figure 5: AF4

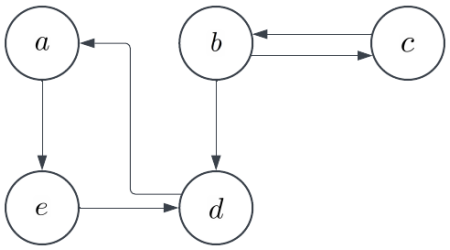

Figure 6: AF5

Figure 7: Systèmes d'argumentation utilisés lors des tests

Tous ces systèmes d'argumentation décrits, dans les fichiers `.apx` présents dans le dossier de tests ont été testés. Ils ont été testés avec différentes requêtes afin de valider le bon fonctionnement du programme.

Afin de ne pas alourdir cette section, un focus est réalisé sur trois systèmes représentatifs : AF1, AF3 et AF5. Ces systèmes permettent de présenter des cas différents.

5.2 Focus sur certains systèmes

5.2.1 Système AF1

Le système AF1 possède plusieurs extensions préférées et stables. Il permet de tester les requêtes VE, DC et DS dans des situations où plusieurs extensions existent.

Le système AF1 possède deux extensions préférées, qui sont également des extensions stables :

$$\{a, d\} \quad \text{et} \quad \{b, d\}$$

Vérification d'extension préférée (VE-PR) :

```
python3 program.py -p VE-PR -f Fichiers-tests/test_af1.apx -a A,d
```

Résultat : **YES** (la casse n'a pas d'impact)

```
python3 program.py -p VE-PR -f Fichiers-tests/test_af1.apx -a a,b
```

Résultat : **NO** (ensemble non préféré)

Acceptation crédulée (DC-PR) :

```
python3 program.py -p DC-PR -f Fichiers-tests/test_af1.apx -a a
```

Résultat : **YES** (l'argument a appartient à au moins une extension préférée)

Acceptation sceptique (DS-PR) :

```
python3 program.py -p DS-PR -f Fichiers-tests/test_af1.apx -a d
```

Résultat : **YES** (l'argument d appartient à toutes les extensions préférées)

```
python3 program.py -p DS-PR -f Fichiers-tests/test_af1.apx -a a
```

Résultat : **NO** (l'argument a n'appartient pas à toutes les extensions préférées)

Côté stable (ST) :

```
python3 program.py -p VE-ST -f Fichiers-tests/test_af1.apx -a b,d
```

Résultat : **YES**

```
python3 program.py -p DC-ST -f Fichiers-tests/test_af1.apx -a c
```

Résultat : **NO** (l'argument c n'appartient à aucune extension stable)

5.2.2 Système AF3

Le système AF3 est intéressant car il ne possède aucune extension stable. Il permet donc de vérifier que le programme gère correctement ce cas particulier.

Les extensions attendues pour AF3 sont les suivantes :

Extensions préférées : $\{d\}$ Extensions stables : aucune

Vérification d'extension préférée (VE-PR) :

```
python3 program.py -p VE-PR -f Fichiers-tests/test_af3.apx -a d
```

Résultat : **YES**

Vérification d'extension stable (VE-ST) :

```
python3 program.py -p VE-ST -f Fichiers-tests/test_af3.apx -a d
```

Résultat : **NO** (il n'y a pas d'extension stable)

Acceptation crédulée stable (DC-ST) :

```
python3 program.py -p DC-ST -f Fichiers-tests/test_af3.apx -a d
```

Résultat : **NO** (aucune extension stable n'existe)

Acceptation sceptique stable (DS-ST) :

```
python3 program.py -p DS-ST -f Fichiers-tests/test_af3.apx -a a
```

Résultat : **YES** (convention : si aucune extension stable n'existe, la requête DS est vraie)

5.2.3 Système AF5

Le système AF5 permet d'illustrer la différence entre les sémantiques préférées et stables. Il possède plusieurs extensions préférées, mais une seule extension stable.

Les extensions attendues pour AF5 sont les suivantes :

Extensions préférées : $\{c\}$ et $\{a, b\}$ Extensions stables : $\{a, b\}$

Vérification d'extension préférée (VE-PR) :

```
python3 program.py -p VE-PR -f Fichiers-tests/test_af5.apx -a c
```

Résultat : **YES** (l'ensemble $\{c\}$ est une extension préférée)

Vérification d'extension stable (VE-ST) :

```
python3 program.py -p VE-ST -f Fichiers-tests/test_af5.apx -a c
```

Résultat : **NO** (l'ensemble $\{c\}$ n'est pas stable)

```
python3 program.py -p VE-ST -f Fichiers-tests/test_af5.apx -a a,b
```

Résultat : **YES** (l'ensemble $\{a, b\}$ est l'unique extension stable)

Acceptation sceptique préférée (DS-PR) :

```
python3 program.py -p DS-PR -f Fichiers-tests/test_af5.apx -a c
```

Résultat : **NO** (l'argument c n'appartient pas à toutes les extensions préférées)

Acceptation sceptique stable (DS-ST) :

```
python3 program.py -p DS-ST -f Fichiers-tests/test_af5.apx -a a
```

Résultat : **YES** (l'argument a appartient à toutes les extensions stables)

L'ensemble des tests réalisés produit des résultats conformes aux attentes. Ces résultats permettent de valider le bon fonctionnement du programme ainsi que sa conformité aux consignes du projet.

6 Conclusion

Dans ce projet, nous avons développé un programme permettant d'analyser des systèmes d'argumentation à partir de fichiers `.apx`. Le programme permet de calculer différentes extensions et de répondre aux requêtes du sujet.

Le système est capable de :

- Parser correctement un système d'argumentation.
- Calculer les extensions admissibles, préférées et stables.
- Répondre aux requêtes VE, DC et DS pour les sémantiques préférée et stable.

Les tests réalisés sur plusieurs systèmes d'argumentation ont montré que le programme fonctionne correctement et produit des résultats conformes aux attentes du projet. Les différents cas vus précédemment permettent de valider la gestion de différentes situations, comme la présence de plusieurs extensions ou l'absence d'extension stable.

Ce projet nous a permis de mieux comprendre le fonctionnement des systèmes d'argumentation ainsi que la mise en œuvre concrète des systèmes d'argumentation. Il a également permis de travailler sur la structuration d'un programme et la gestion des entrées utilisateur.

Aide à la rédaction

Une intelligence artificielle (ChatGPT) a été utilisée lors de la rédaction de ce rapport. Elle a servi à reformuler certaines phrases et à vérifier l'orthographe.

L'intelligence artificielle a également été utilisée pour aider à écrire certaines formules en LaTeX. Toutefois, l'ensemble des droits de propriété intellectuelle de ce document demeure exclusivement attribué à ses auteurs.