Rapport
Abstract Argumentation Solver

Master 1: IAD

Fait par : Bouadi Sabrina

Année universitaire 2024/2025

Table des Matières

Objectif du Projet

Le projet vise à développer une solution logicielle pour traiter des problèmes liés aux cadres argumentatifs (Frameworks d'Argumentation ou AF). Les tâches à réaliser sont désignées comme suit :

- **VE-CO** : Vérification d'une extension par rapport à une sémantique.
- **DC-CO** : Décision de l'acceptabilité crédule d'un argument par rapport à une sémantique.
- **DS-CO** : Décision de l'acceptabilité sceptique d'un argument par rapport à une sémantique.
- **VE-ST**, **DC-ST**, **DS-ST** : Tâches similaires mais relatives aux extensions stables.

Le cadre argumentatif (AF) est fourni sous forme de fichier contenant les arguments et les relations d'attaque. Le programme, implémenté en Python, doit permettre de répondre **OUI** ou **NON** à chaque tâche, selon un format spécifique passé en ligne de commande.

Organisation du Code

Module `utils.py`

Ce fichier contient des fonctions utilitaires, notamment :

- `is_valid_argument` : Validation des arguments.
- `powerset` : Génération des sous-ensembles.
- `is_attacked` et `get_arg_attackers` : Gestion des relations d'attaque.
- `compute_acceptability` : Calcul de l'acceptabilité d'un argument.

Module `main.py`

Ce fichier constitue le point d'entrée principal du programme. Il gère :

1. La lecture du fichier d'entrée et l'extraction des arguments et relations.
2. L'interprétation des commandes en ligne (VE, DC, DS).

3. L'appel aux fonctions pertinentes pour répondre aux requêtes.
4. L'affichage des résultats sous forme de **OUI** ou **NON**.

Analyse du Code

Modules Utilisés

Nous avons utilisé les modules suivants :

- **os** et **sys** : pour interagir avec le système d'exploitation et traiter les arguments en ligne de commande.
- **itertools** : pour créer des itérateurs, notamment pour générer des permutations et des sous-ensembles (power sets).

Variables Globales

Deux variables globales principales sont définies :

- **arguments** : un ensemble stockant les noms des arguments.
- **attacks** : un ensemble contenant les relations d'attaque entre arguments.

L'utilisation des ensembles (*sets*) offre plusieurs avantages :

- **Unicité** : Chaque argument et chaque relation est unique, ce qui évite les doublons.
- **Efficacité** : Tester l'appartenance à un ensemble a une complexité moyenne de **O(1)**, ce qui est crucial pour les fonctions telles que `is_attacked(arg, attacks)`.

Principales Fonctions

Voici une liste non exhaustive des fonctions clés implémentées dans le projet :

- **`is_valid_argument(arg)`** : Vérifie si un nom d'argument est valide (alphanumérique et non réservé comme "arg" ou "att").

- **`process_line(line, line_number)`** : Traite chaque ligne du fichier d'entrée pour extraire les arguments et les relations d'attaque.
- **`is_attacked(arg, attacks)`** : Vérifie si un argument est attaqué par un autre.
- **`get_arg_attackers(arg, attacks)`** : Retourne un ensemble des attaquants d'un argument donné.
- **`get_attacked_args(set_of_args, attacks)`** : Retourne un ensemble des arguments attaqués par un ensemble donné d'arguments.
- **`powerset(iterable)`** : Génère le *power set* d'un itérable.
- **`compute_acceptability(arg, E, relations)`** : Calcule l'acceptabilité d'un argument par rapport à un ensemble donné.
- **`decide(elem, arg, set1)`** : Décide si un élément appartient à un ensemble et affiche "OUI" ou "NON".

Algorithmes Implémentés

Ensembles sans conflit (*Conflict-Free Sets*)

L'algorithme `compute_cfs` :

1. Génère tous les sous-ensembles possibles des arguments à l'aide de `powerset`.
2. Filtre les sous-ensembles contenant des arguments qui s'attaquent mutuellement.

Acceptabilité d'un Argument

L'algorithme vérifie si l'ensemble **E** défend un argument **arg** contre ses attaquants. Si, pour chaque attaquant, il existe un argument dans **E** qui attaque cet attaquant, alors **arg** est accepté.

Ensembles Admissibles

Un ensemble admissible est sans conflit et chaque argument qu'il contient est acceptable par rapport à l'ensemble. L'algorithme :

1. Vérifie la validité des arguments et relations.
2. Génère les ensembles sans conflit.
3. Identifie les ensembles pour lesquels chaque argument est acceptable.

Extensions Stables

Une extension stable est un ensemble sans conflit qui attaque tous les arguments qui ne lui appartiennent pas. L'algorithme :

1. Identifie les ensembles sans conflit.
2. Retient les ensembles attaquant tous les arguments externes.

Extensions Complètes

Une extension complète est un ensemble admissible dans lequel chaque argument acceptable appartient à l'ensemble. L'algorithme :

1. Génère les ensembles admissibles.
2. Retient ceux où tous les arguments acceptables sont inclus.

Acceptabilité Sceptique

Un argument est sceptiquement accepté s'il est accepté dans toutes les extensions possibles. L'algorithme :

1. Génère toutes les extensions (stables ou complètes).
2. Retient les arguments présents dans toutes les extensions.

Acceptabilité Crédule

Un argument est crédulement accepté s'il est accepté dans au moins une extension. L'algorithme :

1. Génère toutes les extensions.
2. Retient les arguments présents dans au moins une extension.

Structures de Données

Le projet utilise plusieurs structures de données pour gérer efficacement les arguments, les relations d'attaque, et les extensions dans le cadre argumentatif (AF). Voici une explication des structures de données principales utilisées dans le code :

1. **Ensembles (Sets)**
    - **arguments** : Un ensemble est utilisé pour stocker les arguments dans le cadre argumentatif. L'utilisation des ensembles garantit l'unicité des arguments et permet une recherche rapide pour vérifier si un argument existe ou non.
    - **attacks** : Un autre ensemble est utilisé pour stocker les relations d'attaque entre les arguments. Chaque élément de cet ensemble est un couple (tuple) représentant une relation d'attaque, où le premier élément attaque le second.
2. **Listes**
    - Certaines fonctions, comme **get_arg_attackers** ou **get_attacked_args**, utilisent des listes pour itérer sur les attaques et collecter les attaquants ou les arguments attaqués. Les listes sont utilisées ici car l'ordre des éléments peut être important dans certains calculs.
3. **Tuples**
    - Les relations d'attaque sont représentées par des tuples de deux éléments (arg1, arg2), où arg1 attaque arg2. Les tuples sont immuables, ce qui les rend adaptés pour stocker des relations fixes qui ne doivent pas être modifiées.
4. **Itérateurs (via itertools)**
    - **permutations** : Cette fonction est utilisée pour générer toutes les permutations d'un ensemble d'éléments, notamment dans la fonction **is_combination_in_list**. Cela permet de vérifier si une combinaison d'arguments attaqués parvient à être attaquée par d'autres.
    - **combinations** : Utilisée dans la fonction **powerset**, cette fonction génère tous les sous-ensembles (ou power sets) d'un ensemble donné, ce qui est crucial pour les algorithmes de calcul des extensions.
5. **Classes**
    - **Extensions** : Cette classe représente les extensions dans le cadre argumentatif. Elle contient des ensembles d'arguments et fournit des méthodes pour obtenir les arguments acceptés de manière sceptique ou crédule.
    - **Dung** : La classe principale représentant un cadre argumentatif (AF) dans le projet. Elle gère les arguments, les relations d'attaque et les

algorithmes associés pour calculer des extensions stables, complètes, et admissibles. Elle contient également une classe interne **Semantics** pour implémenter différentes sémantiques d'extensions.

Conclusion

Le projet a permis de développer un outil robuste pour traiter les cadres argumentatifs, en mettant en œuvre des algorithmes optimisés pour calculer des ensembles et des extensions selon diverses sémantiques. Les structures de données comme les ensembles (*sets*) et les modules Python standard ont contribué à la simplicité et à l'efficacité du code.