

Rapport de Projet

Solveur de Systèmes d'Argumentation

Aidoudi Aaron

Année Universitaire 2025 - 2026

Table des matières

I.	Introduction	3
	A. Contexte Scientifique	3
	B. Objectifs du Projet	3
II.	Choix Techniques et Architecture	3
	A. Choix du Langage de Programmation	3
	B. Modules	4
III.	Structures de Données	5
	A. Méthodologie	5
	B. Implémentation	5
IV.	Parsing des Fichiers d'Argumentation	6
	A. Méthodologie	6
	B. Implémentation	7
V.	Fonctions Utilitaires	8
	A. Méthodologie	8
	B. Implémentation	8
VI.	Sémantiques	10
	A. Méthodologie	10
	B. Implémentation	11
VII.	Interface	13
	A. Solveur	13
	B. Main	13
VIII.	Tests et Résultats	13
	A. Protocole de Test	13
	B. Résultats Expérimentaux	14
IX.	Conclusion	15
X.	Sources	15

I. Introduction

A. Contexte Scientifique

L'argumentation est un processus fondamental du raisonnement humain, omniprésent dans nos interactions quotidiennes, nos débats et nos prises de décision. Pour choisir un restaurant, défendre une opinion ou résoudre un conflit, nous construisons et évaluons constamment des arguments. La formalisation mathématique de ces processus argumentatifs constitue un domaine de recherche majeur en Intelligence Artificielle, avec des applications variées allant des systèmes de recommandation aux assistants virtuels, en passant par l'aide à la décision médicale ou juridique.

En 1995, Pham Minh Dung a introduit un nouveau formalisme : les systèmes d'argumentation abstraits. Cette approche représente les arguments comme des entités abstraites liées par une relation binaire d'attaque, sans se préoccuper de leur structure interne. Un **système d'argumentation** est ainsi défini comme un couple $F = \langle A, R \rangle$ où A est un ensemble d'arguments et $R \subseteq A \times A$ une relation d'attaque entre ces arguments. La question centrale devient alors : étant donné un tel système, quels arguments peut-on rationnellement accepter ? Pour y répondre, Dung a proposé plusieurs **sémantiques d'acceptabilité** basées sur le concept d'extension : un ensemble d'arguments pouvant être acceptés conjointement. Les sémantiques préférée et stable figurent parmi les plus utilisées et étudiées de ces sémantiques.

B. Objectifs du Projet

Ce projet s'inscrit dans le cadre du cours de Représentation des Connaissances et Raisonnement du Master 1 d'Intelligence Artificielle Distribuée. Il vise à développer un outil logiciel permettant de résoudre automatiquement six problèmes computationnels en argumentation abstraite :

Pour la **sémantique préférée** :

- **VE-PR** : Vérifier si un ensemble donné constitue une extension préférée.
- **DC-PR** : Déterminer l'acceptabilité crédule d'un argument (appartenance à au moins une extension).
- **DS-PR** : Déterminer l'acceptabilité sceptique d'un argument (appartenance à toutes les extensions).

Pour la **sémantique stable** :

- **VE-ST** : Vérifier si un ensemble donné constitue une extension stable.
- **DC-ST** : Déterminer l'acceptabilité crédule d'un argument.
- **DS-ST** : Déterminer l'acceptabilité sceptique d'un argument.

Ces problèmes, bien que théoriquement simples à énoncer, présentent une complexité computationnelle significative. Les problèmes de vérification sont généralement polynomiaux, tandis que les problèmes de décision crédule sont NP-complet et les problèmes de décision sceptique atteignent coNP-complet pour la sémantique stable et Π_2^P -complet pour la sémantique préférée.

Ce rapport présente dans un premier temps les choix techniques et architecturaux effectués pour la conception du solveur. Nous détaillons ensuite les structures de données implémentées pour représenter les systèmes d'argumentation. Le cœur du rapport expose les algorithmes développés pour le calcul des extensions selon les deux sémantiques, ainsi que leur analyse de complexité. Enfin, nous présentons les résultats des tests effectués et discutons des performances obtenues, avant de conclure sur les limites de notre approche et les perspectives d'amélioration.

II. Choix Techniques et Architecture

A. Choix du Langage de Programmation

Le choix du langage **C++** et de son standard C++17 pour ce projet repose sur plusieurs considérations techniques et pédagogiques.

Premièrement, la **performance** : les problèmes d'argumentation étant complexes, l'efficacité computationnelle est cruciale. C++ compile vers du code machine natif, offrant des performances bien supérieures aux langages

interprétés comme Python. Cette différence est d'autant plus visible lors du traitement de graphes d'argumentation contenant plusieurs dizaines d'arguments.

Deuxièmement, la **bibliothèque standard** C++ (STL) fournit des structures de données optimisées (*std::vector*, *std::unordered_map*) dont la gestion mémoire est automatique mais prédictible. Cette caractéristique est essentielle pour les algorithmes d'énumération qui peuvent générer un nombre exponentiel d'ensembles intermédiaires.

B. Modules

Notre projet repose sur trois couches séparées :

1. Données

- **SystemeArgumentation** : Ce module est la fondation du projet. Il encapsule le système d'argumentation et porte la responsabilité du mapping entre les noms d'arguments et leurs identifiants entiers. Il maintient aussi la cohérence des structures du graphe.
- **Parseur** : Module autonome qui lit et écrit les fichiers. Il convertit les fichiers *.apx* en instances de *SystemeArgumentation* en appliquant une validation syntaxique.

2. Moteur de Raisonnement

C'est le cœur algorithmique du solveur. Cette couche ignore totalement l'existence des chaînes de caractères de sorte à améliorer la complexité. Elle ne manipule que des entiers et des vecteurs.

- **Utilitaires** : Ce module sans état fournit des propriétés telles que la vérification de sans-conflit, la défense, l'admissibilité, et les opérations ensemblistes optimisées via des masques booléens.
- **Semantiques** : Implémente les stratégies de résolution pour les sémantiques préférées et stables. Il s'appuie sur *Utilitaires* pour valider les étapes de sa recherche en s'inspirant d'algorithmes avancés (voir les références dans la partie Sources).

3. Interface

- **Solveur** : Reçoit les requêtes, convertit les paramètres en identifiants numériques, délègue le calcul au module *Semantiques*, et retraduit le résultat pour l'utilisateur.
- **Main** : Point d'entrée gérant l'interaction en ligne de commande avec les drapeaux (*-p*, *-f*, *-a*) et l'affichage final.

III. Structures de Données

*La représentation interne d'un système d'argumentation constitue le socle de toutes les opérations algorithmiques ultérieures. Cette section présente la classe **SystemeArgumentation** qui encapsule un système d'argumentation abstrait $F = \langle A, R \rangle$.*

A. Méthodologie

i. Indexation numérique

Dans une première phase de développement, nous avons opté pour une approche naïve reposant sur la bibliothèque standard C++ et la manipulation directe des chaînes de caractères. Nous utilisons des `std::set<std::string>` pour stocker les arguments et des `std::set<std::pair<string, string>>` pour les attaques. Bien que fonctionnelle, cette approche a rapidement montré ses limites lors de tests avec des systèmes comptant plusieurs dizaines d'arguments.

Pour pallier ce problème, nous avons opéré une refonte de ces structures de données en adoptant une stratégie de **mapping**. Le principe est de dissocier la représentation via les noms des arguments de la représentation machine via des identifiants numériques. Dès la phase de parsing, chaque argument se voit attribuer un **identifiant unique entier** allant de 0 à N-1 (où N est le nombre d'arguments).

Nous maintenons deux structures de traduction pour faire le pont :

- Une table de hachage via `std::unordered_map<string, int>` permet de passer du nom à l'identifiant avec une complexité $O(1)$.
- Un tableau dynamique via `std::vector<string>` permet l'opération inverse, de l'identifiant au nom, avec une complexité $O(1)$.

Grâce à cette transformation, toutes les opérations ultérieures du solveur manipulent exclusivement des entiers.

ii. Représentation du Graphe

Une fois les arguments réduits à des entiers, la question de la représentation des attaques R s'est posée. Plutôt que d'utiliser une matrice d'adjacence, nous avons opté pour des **listes d'adjacence** stockées sous forme de vecteurs d'entiers via `std::vector<std::vector<int>>`.

La liste d'adjacence est plus économe en mémoire et permet d'itérer sur les voisins d'un nœud sans parcourir inutilement des cases vides, ce qui est crucial pour les algorithmes de propagation de contraintes.

Enfin, une particularité de notre implémentation est la maintenance explicite du **graphe inverse**. Dans notre classe, nous stockons non seulement la liste des cibles pour chaque argument via `adjacence_`, mais également la liste de ses attaquants `parents_`.

En théorie des graphes classique, on se contenterait des successeurs, mais en argumentation, la notion de défense est centrale, un argument pouvant être défendu si ses attaquants sont eux-mêmes attaqués. Pour vérifier la défense d'un argument a , il faut savoir qui attaque a . Sans le graphe inverse, cette opération nécessiterait un parcours complet de toutes les attaques du système, soit une complexité $O(|R|)$. Avec la structure `parents_`, l'accès aux attaquants est instantané (avec une complexité $O(1)$ pour l'accès au vecteur), ce qui accélère les algorithmes de vérification d'admissibilité.

B. Implémentation

i. Insertion et Mapping

La méthode `ajouterArgument()` est le point d'entrée de la construction du système. Plutôt que de gérer un compteur d'identifiants manuel, nous exploitons les propriétés du `std::vector`. L'identifiant d'un nouvel argument est implicitement défini par la taille actuelle du vecteur `idVersNom_`.

Concrètement, si le système contient déjà k arguments indexés de 0 à $k-1$, le prochain argument recevra l'identifiant k . Cette approche garantit qu'il n'y a aucun "trou" dans la numérotation.

Lors de l'ajout, une vérification préalable dans la table de hachage `nomVersId_` empêche la création de doublons. Si l'argument est nouveau, nous redimensionnons immédiatement les vecteurs d'adjacence `adjacence_` et `parents_`. Cette allocation de mémoire est importante car elle évite les réallocations coûteuses lors de l'ajout ultérieur d'attaques et prévient tout risque de débordement d'index.

ii. Gestion des Relations

L'ajout d'une attaque via *ajouterAttaque()* suit une logique transactionnelle pour maintenir l'invariant du graphe inverse. L'opération nécessite d'abord une phase de traduction : les noms *source* et *cible* sont convertis en entiers via la table de hachage. Si l'un des deux est introuvable, l'opération est annulée pour préserver l'intégrité du système. Une fois les identifiants validés, nous procédons à une double insertion. L'arc est ajouté dans le graphe principal *adjacence_* mais aussi dans le graphe des parents *parents_*. Il est alors impossible en théorie qu'une attaque existe dans un sens sans être référencée dans l'autre, garantissant la cohérence.

Nous avons également implémenté une vérification d'unicité avant l'insertion. Bien que les vecteurs *std::vector* autorisent les doublons (contrairement aux *std::set*), nous parcourons la liste d'adjacence existante pour éviter de stocker deux fois la même attaque.

iii. Séparation des Responsabilités

L'implémentation distingue deux catégories de méthodes d'accès, répondant à des besoins différents :

- Les **Getters** servant au **solveur**, avec des méthodes comme *getId()*, *getAdjacence()* ou *getParents()* sont conçues pour être appelées un grand nombre de fois par le solveur. Elles sont implémentées pour être aussi légères que possible, retournant souvent des références constantes (via *const &*) vers les structures internes pour éviter toute copie inutile de données. *getNom()* se résume par exemple à un simple accès tableau, offrant une complexité $O(1)$.
- Les **Getters** servant à l'**utilisateur**, avec des méthodes comme *getAttaques()* retournant une liste de paires de chaînes, ou *getArguments()* étant destinées à l'affichage ou au debug. Contrairement à notre première version qui maintenait une liste permanente des attaques sous forme de chaînes, cette version les reconstruit à la volée en parcourant le graphe d'entiers. Ce choix libère de la mémoire vive, nous ne stockons pas ce qui peut être déduit.

Enfin, la robustesse de l'implémentation s'appuie sur une gestion des erreurs ; les méthodes de modification retournent des booléens pour permettre à l'appelant *Parseur* de gérer les erreurs de format sans interruption brutale, tandis que les méthodes d'accès comme la conversion de nom vers identifiant lèvent des exceptions explicites en cas d'incohérence, facilitant la détection de bugs lors du développement.

IV. Parsing des Fichiers d'Argumentation

*Le chargement des données d'entrée, fournies au format .apx, est confié à la classe statique **Parseur**. Ce module a la responsabilité de valider l'intégrité syntaxique du fichier avant de construire le système d'argumentation.*

A. Méthodologie

Le format **.apx** définit une syntaxe simple pour représenter des systèmes d'argumentation. Chaque ligne déclare soit un argument avec le préfixe **arg**, soit une attaque préfixe **att**. Les noms d'arguments peuvent contenir des lettres, chiffres et underscores, à l'exception des préfixes précédents. Enfin, tous les arguments doivent être déclarés avant d'être utilisés dans une attaque.

i. Analyse Manuelle

Pour l'analyse lexicale, nous avons évalué deux approches : l'utilisation du moteur d'**expressions régulières** via *std::regex* de la bibliothèque standard C++ et une approche de parsing manuel par manipulation de chaînes. Notre choix s'est porté sur l'analyse manuelle pour des raisons de performance et de précision. Les lectures de différentes sources et tests ont montré que l'overhead de compilation des *regex* était pénalisant sur de grands fichiers, rendant l'approche manuelle plus rapide pour ce format spécifique.

De plus, l'approche manuelle permet un diagnostic plus précis. Là où une *regex* indique simplement qu'une ligne ne correspond pas au motif attendu, notre parseur peut identifier l'erreur exacte (parenthèse manquante, virgule absente, nom invalide) et fournir un message contextuel précis à l'utilisateur, facilitant le débogage des instances.

ii. Flux de Traitement

Le processus de parsing suit un flux linéaire. Le fichier est lu ligne par ligne, et chaque ligne subit une série de transformations :

1. La **normalisation** via la suppression des espaces blancs superflus en début et fin de ligne pour tolérer les variations d'indentation.
2. Le **filtrage**, avec l'ignorance silencieuse des lignes vides et des commentaires.
3. La **classification** par l'identification du type d'instruction par analyse des quatre premiers caractères.
4. **L'extraction** via la récupération des données, la vérification des caractères autorisés, et l'insertion dans le *SystemeArgumentation*.

Cette approche garantit qu'aucune donnée invalide ne pollue le système, si une erreur est détectée à une certaine ligne, le processus s'arrête immédiatement avant d'avoir tenté de traiter la ligne suivante.

B. Implémentation

i. Extraction

L'extraction des données repose sur des opérations de découpage de chaînes à faible coût. Pour un argument, la méthode *parserLigneArgument()* isole la sous-chaîne située entre les balises "arg(" et ")." via *std::string::substr*. Pour une attaque, *parserLigneAttaque()* localise d'abord le séparateur (la virgule) à l'aide de *std::string::find*, puis scinde le contenu en deux parties distinctes, la source et la cible.

Un point important de l'implémentation est la fonction utilitaire *trim()*. Plutôt que d'écrire des boucles manuelles propices aux erreurs d'indexation, nous avons implémenté cette fonction en utilisant les algorithmes *std::find_if* combinés à des expressions lambda. Cette approche délègue la logique de recherche à la STL, garantissant une exécution optimisée.

ii. Gestion des Erreurs

La robustesse du parseur est assurée par un mécanisme d'exceptions. Nous avons défini une classe ***ErreurParsing*** héritant de *std::runtime_error*. Chaque anomalie détectée, telle qu'un fichier introuvable, une syntaxe incorrecte, ou une référence à un argument inconnu, déclenche le lancer d'une exception comportant le numéro de ligne et une description explicite du problème. Le choix des exceptions, plutôt que des codes de retour, se justifie par la nature de l'opération de parsing : un fichier est soit valide, soit invalide. Il n'existe pas d'état intermédiaire exploitable par le solveur. L'exception force l'arrêt du traitement et garantit que le code appelant ne peut pas ignorer l'erreur.

Enfin, l'intégrité sémantique est vérifiée à la volée. Lors de l'analyse d'une ligne d'attaque *att(a,b)*, le parseur interroge le *SystemeArgumentation* pour vérifier l'existence préalable des arguments *a* et *b*. Si l'un d'eux est manquant, une exception spécifique est levée, empêchant la création de liens n'existant pas dans le graphe.

V. Fonctions Utilitaires

Le module **Utilitaires** constitue la couche entre la structure de données *SystemeArgumentation* et les algorithmes de résolution *Semantiques*. Ce module regroupe les propriétés nécessaires à la vérification des propriétés fondamentales de l'argumentation abstraite (conflit, défense, admissibilité, ...).

A. Méthodologie

La transition vers une représentation numérique des arguments a modifié notre approche des opérations ensemblistes. Là où notre première version manipulait des conteneurs lourds de type `std::set<std::string>`, cette nouvelle implémentation travaille exclusivement sur des vecteurs d'entiers `std::vector<int>`, désignés sous le type *EnsembleIds*.

i. Ensembles & Recherche

Pour garantir la cohérence des comparaisons et des opérations ensemblistes, nous avons imposé une contrainte : tout *EnsembleIds* manipulé par le solveur doit être **trié**, ce qui offre deux avantages majeurs. Tout d'abord, **l'unicité** de représentation, car deux ensembles contenant les mêmes arguments sont garantis d'être identiques, ce qui permet des comparaisons d'égalité en temps linéaire. Aussi, **l'optimisation** algorithmique, étant le tri permettant l'utilisation ultérieure de recherches dichotomiques via `std::binary_search` ou d'algorithmes d'intersection optimisés, bien que nous privilégions souvent l'accès direct par index.

De plus, de sorte à améliorer une fois de plus la complexité, nous utilisons des **masques booléens** via `std::vector<bool>` pour les tests d'appartenance fréquents. Dans les algorithmes de sémantique stable, nous devons vérifier répétitivement si un argument appartient à l'ensemble S. Plutôt que d'effectuer une recherche en $O(|S|)$ ou $O(\log|S|)$ dans le vecteur, nous pré-calculons un tableau de booléens de taille N où l'index i est vrai si l'argument i est dans S. Cela réduit la complexité du test d'appartenance à un temps constant $O(1)$, offrant un gain de performance significatif sur les graphes denses.

B. Implémentation

L'implémentation se divise en deux parties : la conversion des données et la vérification des propriétés sémantiques. Conversion et Interface

Les fonctions `convertirNomsEnIds()` et `convertirIdsEnNoms()` assurent la cohérence entre l'interface utilisateur avec les chaînes de caractères et la partie assurant le calcul. La conversion vers les identifiants inclut systématiquement une étape de tri via `std::sort`. Nous utilisons également `reserve()` sur les vecteurs pour éviter les réallocations mémoire multiples lors de la transformation.

Les fonctions de vérification de propriétés exploitent directement la structure du *SystemeArgumentation* :

i. Sans-Conflit via `estSansConflit()`

Cette fonction constitue le premier filtre de validité pour tout ensemble candidat à devenir une extension. Elle vérifie l'absence **d'attaques internes** en testant toutes les paires d'arguments de l'ensemble S.

L'implémentation utilise une double boucle imbriquée avec une complexité de $O(|S|^2)$ en termes de nombre de tests. Cependant, grâce à la représentation par liste d'adjacence du *SystemeArgumentation*, chaque appel à `attaqueExiste()` s'effectue en temps (quasiment) constant. Concrètement, nous parcourons le vecteur `adjacence_` pour vérifier une présence. Cette recherche linéaire dans un petit vecteur est beaucoup plus rapide qu'un accès dans une structure de type `std::set`.

ii. Défense via *defend()*

L'implémentation de la fonction de défense illustre l'intérêt du graphe inverse *parents_* dans notre architecture. Cette fonction vérifie si un ensemble **S défend** un argument *a* contre tous ses **attaquants**, une propriété centrale.

Sans le graphe inverse, nous aurions été contraints de parcourir l'intégralité de la relation d'attaque **R** pour identifier les attaquants d'un argument *a*. Grâce à *getParents()*, nous obtenons directement la liste des attaquants en temps constant $O(1)$. Cette liste est généralement très courte : dans les graphes d'argumentation que nous testons, un argument subit rarement plus de 3 à 5 attaques.

Pour chaque attaquant *b* de l'argument *a*, nous recherchons un défenseur dans **S** qui contre-attaque *b*. Cette recherche s'arrête dès qu'un défenseur est trouvé, évitant ainsi de parcourir inutilement le reste de **S**. La complexité résultante est $O(|S| \times |Attaquants(a)|)$.

Un argument sans attaquants est défendu par tout ensemble, y compris l'ensemble vide. Ce cas est géré naturellement par la boucle qui, ne trouvant aucun attaquant à neutraliser, valide immédiatement la défense. Cette propriété est importante pour le calcul de l'extension basique qui commence par inclure tous les arguments non attaqués.

iii. Admissibilité via *estAdmissible()*

La composition des deux propriétés précédentes forme la notion **d'admissibilité** : un ensemble **S** est admissible s'il est sans conflit interne et si **S** défend chacun de ses éléments.

L'ordre d'exécution des vérifications n'est pas anodin. Nous testons d'abord la propriété de sans-conflit avant de vérifier la défense, la plupart des ensembles non admissibles échouant au test de sans-conflit. Invalider rapidement ces candidats via un test $O(|S|^2)$ évite de lancer le test de défense plus coûteux $O(|S| \times \sum_{a \in S} |Attaquants(a)|)$.

La boucle sur les éléments de **S** pour tester leur défense s'interrompt dès qu'un argument non défendu est détecté. Dans les algorithmes de backtracking utilisés pour les sémantiques préférée et stable, cette interruption précoce permet d'élaguer des branches entières de l'arbre de recherche, évitant ainsi d'explorer un grand nombre de combinaisons invalides.

iv. Sémantique Stable et *attaqueToutExterieur()*

La fonction *attaqueToutExterieur()* est directement liée à la sémantique stable. Elle doit vérifier que tout argument hors de **S** est **attaqué** par au moins un élément de **S**. Cette propriété combinée au sans-conflit caractérise les extensions stables.

C'est ici que le masque booléen *estDansS* est utilisé. L'algorithme se déroule en deux temps :

1. **Construction du masque** en $O(|S|)$: nous initialisons un tableau de **N** booléens (où **N** est le nombre total d'arguments) à false, puis nous marquons à true les positions correspondant aux arguments de **S**. Cette phase permet ensuite de tester l'appartenance à **S** en temps constant.
2. **Vérification** : pour chaque argument *a* de l'univers, nous vérifions d'abord s'il est hors de **S** (test en $O(1)$ grâce au masque). Si c'est le cas, nous parcourons ses attaquants via *getParents()*. Grâce au masque, vérifier si un attaquant appartient à **S** est instantané.

v. Fonction Caractéristique

La fonction caractéristique de **Dung**, définie par $F(S) = \{a \in A \mid S \text{ défend } a\}$, est un outil central pour la caractérisation des extensions complètes (qui sont exactement les points fixes de **F**) et pour le calcul de l'extension basique obtenue par itération de **F** depuis l'ensemble vide.

Notre approche consiste à tester la défense pour chaque argument de l'univers. Bien que cette méthode soit de complexité $O(N \times |Attaquants_{max}| \times |S|)$, elle présente l'avantage de la clarté et de la robustesse.

Le résultat retourné est un *EnsembleIds* trié, prêt à être comparé à d'autres ensembles ou réutilisé dans les calculs de points fixes itératifs. Cette propriété est essentielle pour l'algorithme de calcul de l'extension basique qui itère **F** jusqu'à convergence : la comparaison d'égalité $F^{k+1}(S) = F^k(S)$ se fait alors en temps linéaire.

VI. Sémantiques

Le module **Semantiques** constitue le cœur algorithmique du solveur. Ce module aborde la dynamique du raisonnement : la recherche d'ensembles d'arguments satisfaisant des critères logiques.

A. Méthodologie

Les problèmes traités par ce module appartiennent à des classes de complexité élevées. Les problèmes de vérification sont généralement polynomiaux, tandis que les problèmes de décision crédule sont NP-complet et les problèmes de décision sceptique atteignent coNP-complet pour la sémantique stable et Π_2^P -complet pour la sémantique préférée. Ainsi, une approche par énumération exhaustive est inenvisageable pour des graphes dépassant la vingtaine d'arguments.

Nous avons donc implémenté des algorithmes de recherche arborescente avec **backtracking** ainsi qu'une méthode fondée sur le **labelling**. Notre démarche s'inspire des solveurs SAT modernes (voir les références dans la partie Sources) et repose sur cette approche par labelling : elle consiste à reformuler la recherche d'extensions comme un problème de satisfaction de contraintes.

i. Sémantiques à base de Label

Nos algorithmes attribuent à chaque argument l'un des trois états suivants :

- **IN** : l'argument est accepté dans l'extension courante.
- **OUT** : l'argument est rejeté (attaqué par un argument **IN**).
- **UNDEC** : l'argument n'a pas encore été traité.

Ces états permettent de définir des règles de propagation strictes. Par exemple, dès qu'un argument est étiqueté **IN**, toutes ses cibles doivent devenir **OUT**. Inversement, pour qu'un argument soit légitimement **OUT**, il doit avoir au moins un parent **IN**.

ii. Backtracking et Élagage

La résolution s'effectue via une exploration récursive de l'arbre de recherche par *trouverAdmissibleRecursive()* pour la sémantique préférée et *trouverStableRecursive()* pour la sémantique stable. À chaque étape :

- **Phase 1 : Détection de conflit**

L'algorithme vérifie si l'état partiel actuel ne respecte pas une contrainte de cohérence. Par exemple : deux arguments **IN** qui s'attaquent mutuellement, un argument **IN** avec un attaquant également **IN**. Si un conflit est détecté, la branche est immédiatement coupée sans explorer davantage, évitant ainsi des combinaisons invalides.

- **Phase 2 : Détection et résolution des besoins de défense**

Si un argument a est **IN** mais possède un attaquant b qui est **UNDEC**, l'algorithme identifie un problème de défense. Pour résoudre ce problème, il doit trouver un argument c qui attaque b et faire passer c à **IN**, ce qui propagera automatiquement b à **OUT**.

Cette résolution s'effectue par backtracking : l'algorithme essaie successivement tous les défenseurs potentiels de b . Pour chaque défenseur candidat, on sauvegarde l'état actuel, on émet l'hypothèse que le défenseur devient **IN**, puis on propage les conséquences (ses cibles deviennent **OUT**). Ensuite, l'appel récursif traite les nouveaux problèmes générés. Si la récursion échoue, on restaure l'état (on backtrack) et on essaie le défenseur suivant.

Cette méthode garantit que toutes les solutions sont explorées tout en bénéficiant d'un élagage grâce aux propagations de contraintes.

- **Phase 3 : Convergence et validation**

Lorsque tous les arguments ont un état stable c'est à dire qu'on a aucun conflit, tous les **IN** sont défendus, l'algorithme a trouvé une configuration valide. Pour les extensions stables, une vérification finale s'assure que tous les arguments **OUT** sont effectivement attaqués par **S**.

B. Implémentation

i. Vérification d'Extension (VE)

Les problèmes de vérification consistent à valider si un ensemble S respecte les critères d'une sémantique donnée.

- **Stable (VE-ST)**

La vérification se fait en deux étapes indépendantes, chacune correspondant à une condition de la définition.

Tout d'abord, la condition **sans-conflit** : nous appelons *estSansConflit()* qui vérifie en $O(|S|^2)$ l'absence d'attaques internes. C'est le test le plus rapide à invalider, la plupart des ensembles non-stables échouent dès cette étape.

Ensuite, l'**attaque** de tout les autres arguments : nous appelons *attaqueToutExterieur()* qui utilise le masque booléen pour vérifier en $O(N \times \text{degré}_{in})$ que chaque argument hors de S est attaqué. Cette fonction exploite le graphe inverse *parents_* pour accéder directement aux attaquants de chaque argument.

La complexité totale est donc $O(|S|^2 + N \times \text{degré}_{in})$ où degré_{in} est le degré entrant moyen du graphe.

- **Préférée (VE-PR)**

Ce problème est coNP-complet dans le cas général, car prouver la maximalité requiert théoriquement de montrer qu'aucune extension de S n'est admissible. Ainsi, plutôt que d'énumérer toutes les extensions possibles, nous procédons en deux étapes. Premièrement, l'**admissibilité** via *estAdmissible()*. Cette vérification polynomiale filtre immédiatement les ensembles invalides.

Deuxièmement, la **maximalité** que nous prouvons sans énumérer toutes les extensions mais en testant si S peut être étendu localement. Nous créons un masque booléen *estDansS* pour tester en $O(1)$ l'appartenance, puis nous itérons sur tous les arguments x n'étant pas dans S : pour chaque x , nous testons si S avec x est admissible via *estExtensibleAvec()* testant l'admissibilité de S' telle que l'union de S et x .

Si nous trouvons un x tel que S' est admissible, alors S n'est pas maximal, donc pas une extension préférée.

La complexité est alors en pratique $O(N \times |S|^2)$ dans le pire cas, mais avec une interruption dès qu'une extension est trouvée.

ii. Acceptabilité Crédule (DC)

Le problème d'acceptabilité crédule vise à déterminer s'il existe au moins une extension contenant un argument a .

- **Stable (DC-ST)**

La stratégie adoptée est de **forcer** a à être IN et tenter de construire une extension stable complète autour de cette contrainte. Ainsi après l'initialisation de tous les arguments à $UNDEC$ et a à IN , nous propageons la contrainte en mettant toutes les cibles de a dans OUT .

S'en suit l'exploration récursive, nous lançons *trouverStableRecursive()* qui parcourt séquentiellement les arguments en essayant pour chacun les deux branches IN ou OUT . Notons une optimisation dans cette fonction : l'algorithme teste d'abord la branche IN , puis la branche OUT , privilégiant la construction d'ensembles plus grands, ce qui accélère la satisfaction de la condition "*attaque tous les autres*", un ensemble plus grand attaquant potentiellement plus d'arguments externes.

- **Préférée (DC-PR)**

Nous exploitons ici la propriété qu'un argument appartient à une extension préférée si et seulement si il appartient à une extension **admissible**. Notre algorithme se contente alors de trouver un seul ensemble admissible contenant a .

Ici, nous ne pouvons pas forcer immédiatement tous les attaquants de a à OUT , cela créerait des configurations incohérentes : forcer un argument b à OUT signifie qu'il doit être attaqué par un argument IN . Mais si b attaque a et qu'aucun autre argument n'attaque b , alors il est impossible de justifier son statut OUT . Ainsi, nous laissons les attaquants de a à l'état $UNDEC$, et *trouverAdmissibleRecursive()* détectera automatiquement que a est IN mais possède des attaquants non- OUT , identifiant ainsi un besoin de défense (la fonction énumérera alors tous les arguments c qui attaquent b et pour chaque défenseur potentiel c : elle fera passer c à IN , puis propagera).

La complexité est ici exponentielle dans le pire cas ($O(2^N)$), mais l'élagage grâce aux propagations diminue celle-ci.

iii. Acceptabilité Sceptique (DS)

Le problème d'acceptabilité sceptique cherche à établir si un argument a appartient à l'ensemble de toutes les extensions d'une sémantique donnée.

C'est la classe de problèmes la plus complexe, coNP-complet pour stable et Π_2^P -complet pour préférée, justifié par le fait que prouver qu'un argument est dans toutes les extensions nécessite de quantifier universellement sur un ensemble potentiellement exponentiel d'extensions. Nous adoptons ainsi pour cette partie une approche par preuve **par l'absurde**, en cherchant une extension valide ne contenant pas a .

- **Stable (DS-ST)**

Nous implémentons le problème par négation en **forçant** explicitement a à *OUT* dans le labelling initial, puis nous lançons le solveur stable *trouverStableRecursive()*. Si le solveur trouve une extension stable avec cette contrainte, alors nous avons un contre-exemple : il existe une extension stable ne contenant pas a . Sinon, le solveur ne trouve aucune extension, alors a doit obligatoirement être présent dans toutes les extensions stables.

Un cas particulier existe dans lequel le graphe ne possède aucune extension stable (par exemple un cycle impair), alors a est sceptique car l'ensemble vide d'extensions satisfait la condition universelle.

La complexité est identique à DC-ST, mais la réponse est inversée.

- **Préférée (DS-PR)**

Avant de chercher un contre-exemple, nous vérifions d'abord que a est au moins **crédule**, vérification rapide évitant des calculs inutiles. Ensuite, plutôt que d'énumérer exhaustivement toutes les extensions préférées, ce qui serait exponentiel, nous adoptons une approche constructive par points de départ multiples :

Phase 1 : Test de l'extension depuis l'ensemble vide

Nous testons d'abord le cas où l'extension maximale obtenue en partant de l'ensemble vide ne contient pas a . Ce test est rapide et détecte les cas où a n'est pas nécessaire pour défendre d'autres arguments.

Phase 2 : Exploration depuis chaque argument comme graine

Pour chaque argument x différent de a du graphe, nous tentons de construire une extension préférée ne contenant pas a en partant de x :

1. Initialisation du labelling avec a en *OUT*, x en *IN*, et les attaques de x sont propagées.
2. Recherche d'admissible avec l'appel à *trouverAdmissibleRecursive()* pour compléter un ensemble admissible S respectant les contraintes de labelling.
3. Extension gloutonne : si un ensemble admissible S est trouvé, nous l'étendons de manière gloutonne via *etendreEnMaximal()*, produisant ainsi une extension maximale (donc préférée).
4. Vérification : après extension, nous vérifions si a n'est pas dans S_{max} . Il est possible qu'en étendant S , l'algorithme glouton ait été forcé d'inclure a pour défendre d'autres arguments de S . Dans ce cas, S_{max} ne constitue pas un contre-exemple valide, et nous continuons l'exploration. Sinon, nous avons trouvé une extension préférée ne contenant pas a , c'est donc un contre-exemple valide.

Si après avoir exploré tous les points de départ possibles (ensemble vide et chaque argument), nous n'avons trouvé aucun contre-exemple, nous concluons que a appartient à toutes les extensions préférées.

Cette approche est tout de même limitée de par sa nature **non exhaustive**. Dans des graphes très complexes, il pourrait exister des extensions préférées cachées non découvrables par construction gloutonne depuis les points de départ testés. Cependant, sur l'ensemble de nos tests, cette méthode s'est révélée correcte et efficace, trouvant systématiquement les contre-exemples lorsqu'ils existent.

La complexité est en pratique $O(N \times \text{coût-admissible} \times \text{coût-extension})$.

VII. Interface

*La couche supérieure de l'application est constituée des modules **Solveur** et **Main**. Elle assure le rôle d'interface entre le monde extérieur (utilisateur, système de fichiers, ligne de commande) et le moteur de calcul.*

A. Solveur

Le rôle de la classe **Solveur** est d'isoler la complexité de la gestion des identifiants numériques de l'interface utilisateur qui travaille avec des chaînes de caractères. Cette couche garantit que *Semantiques* reste indépendant de la provenance des requêtes.

Lorsqu'une requête est reçue, par exemple via `verifierExtensionPreferee()`, le solveur commence par vérifier que tous les arguments fournis existent dans le système chargé. Si un argument inconnu est détecté, la réponse est immédiatement négative. Il traduit ensuite les chaînes de caractères en identifiants grâce aux utilitaires de mapping, puis délègue enfin ces identifiants au module *Semantiques*.

Cette conception rend le moteur de calcul indépendant de la provenance des requêtes, facilitant les tests et la maintenance.

B. Main

Le module **Main** a la responsabilité de respecter le protocole de communication donné par le sujet. Nous avons alors implémenté un parseur d'arguments prenant en compte les différents drapeaux et leurs arguments respectifs. Le programme suit le flux d'exécution suivant :

1. Analyse des arguments de la ligne de commande
2. Chargement du fichier *.apx*.
3. Exécution de la résolution
4. Affichage du résultat (*OUI* ou *NON*) sur le flux de sortie standard. Toute erreur, telle qu'un fichier introuvable, ou une syntaxe invalide est redirigée vers le flux d'erreur et retourne un code de sortie non nul.

VIII. Tests et Résultats

La correction d'un solveur d'argumentation ne pouvant se déduire à sa compilation, nous avons alors mis en place une méthodologie de test de sorte à détecter les anomalies et les corriger.

A. Protocole de Test

Nous avons mis en place une double stratégie de test comprenant, d'une part, des **tests unitaires** ciblés basés sur de petits graphes (les fichiers *.apx* donnés) afin de valider manuellement chaque sémantique, et d'autre part, une **validation automatisée** via un script Python externe (*tests/verifier_tout.py*) capable de générer exhaustivement tous les sous-ensembles d'arguments, d'en vérifier le statut à l'aide du solveur et de confronter ces résultats aux attentes théoriques.

Cette approche nous a permis de valider le comportement du solveur.

Notons que le script Python n'a pas pour but d'évaluer les performances du solveur sur des instances de grande taille, mais uniquement à garantir sa **correction logique** sur des cas ayant un faible nombre d'arguments. La validation par énumération exhaustive implique de tester 2^N sous-ensembles (pour un système de 20 arguments, cela représente plus d'un million de combinaisons).

Le facteur limitant ici n'est pas la vitesse de notre solveur C++ qui traite une requête très rapidement, mais le coût système induit par le script de test. Pour chaque combinaison testée, le script Python doit demander à l'OS d'instancier un nouveau processus, charger l'exécutable, parser le fichier *.apx* et allouer la mémoire.

Par conséquent, cet outil de validation est restreint aux instances de petite taille ($N < 15$), la performance sur les grandes instances étant validée par des requêtes unitaires.

B. Résultats Expérimentaux

Les tests ont été menés en deux phases distinctes : la vérification sur les systèmes d'argumentations imposés par le sujet, et l'évaluation de la performance sur des instances de taille supérieure.

Conformément aux spécifications du projet, nous avons validé le comportement du solveur sur les fichiers *.apx* fournis en exemple. Les tests ont été exécutés via l'interface en ligne de commande standardisée (ex : `./solveur -p DC-PR -f tests/cas_test/test_af1.apx -a A`), ainsi qu'à travers notre script d'automatisation Python (*verifier_tout.py*).

Pour l'ensemble des six problèmes, le solveur produit les sorties *YES* ou *NO* attendues. Le script Python a permis de vérifier que les réponses du solveur concordent parfaitement avec une résolution par force brute sur ces petites instances, validant alors nos algorithmes de backtracking.

Ci-dessous des captures de résultats d'exécution du fichier *verifier_tout.py* :

```
python3 tests/verifier_tout.py
Analyse de : tests/cas_test/test_af5.apx
Extensions :
- VE-PR : [{'C'}, {'A', 'B'}]
- VE-ST : [{'A', 'B'}]
Acceptabilité crédule :
- DC-PR : ['A', 'B', 'C']
- DC-ST : ['A', 'B']
Acceptabilité sceptique :
- DS-PR : []
- DS-ST : ['A', 'B']
```

```
python3 tests/verifier_tout.py tests/cas_test/test_af2.apx
Analyse de : tests/cas_test/test_af2.apx
Extensions :
- VE-PR : [{'A', 'D'}, {'A', 'E'}, {'B', 'D'}, {'B', 'E'}]
- VE-ST : [{'A', 'D'}, {'A', 'E'}, {'B', 'D'}, {'B', 'E'}]
Acceptabilité crédule :
- DC-PR : ['A', 'B', 'D', 'E']
- DC-ST : ['A', 'B', 'D', 'E']
Acceptabilité sceptique :
- DS-PR : []
- DS-ST : []
```

Au-delà de ces exemples, nous avons soumis le solveur à des systèmes de taille plus conséquente, comportant de 20 à plus de 60 arguments avec une densité d'attaques variable, de sorte à évaluer l'efficacité de nos choix architecturaux. Ces tests s'appuient sur les fichiers *.apx* issus du jeu de tests du solveur ASPARTIX-V de l'ICCMA 2021 (par Wolfgang Dvořák, Matthias König, Johannes P. Wallner and Stefan Woltran). Leurs solutions étant également disponibles, cela nous a permis de valider nos résultats. La conclusion de ces tests est le bon fonctionnement des algorithmes, mais l'augmentation de la taille des graphes entraîne des temps de calcul plus longs pour la résolution du problème DS-PR.

Ci-dessous des captures de résultats d'exécution de différentes commandes depuis le fichier *traffic1.apx*, contenant 61 arguments et 68 attaques :

```
aaronaidoudi@MacBookPro Projet %
./solveur -p VE-PR -f tests/cas_test/traffic1.apx -a a3,a6,a7,a11,a12,a14,a16,a19,a20,a21,a22,a24,a28,a31,a32,a34,a35,a3]
6,a39,a41,a42,a44,a45,a48,a49,a52,a57,a59
YES
aaronaidoudi@MacBookPro Projet %
./solveur -p VE-ST -f tests/cas_test/traffic1.apx -a a3,a7,a9,a11,a12,a14,a16,a19,a20,a21,a22,a24,a28,a31,a34,a35,a36,a3]
9,a41,a42,a44,a45,a48,a49,a52,a54
NO
aaronaidoudi@MacBookPro Projet %
./solveur -p VE-ST -f tests/cas_test/traffic1.apx -a a3,a7,a9,a11,a12,a14,a16,a19,a20,a21,a22,a24,a28,a31,a34,a35,a36,a3]
9,a41,a42,a44,a45,a48,a49,a52,a54,a57
YES
aaronaidoudi@MacBookPro Projet %
./solveur -p DC-PR -f tests/cas_test/traffic1.apx -a a3
YES
aaronaidoudi@MacBookPro Projet %
./solveur -p DC-PR -f tests/cas_test/traffic1.apx -a a1
NO
```

IX. Conclusion

Ce projet avait pour objectif de **concevoir** et **d'implémenter** un solveur d'argumentation abstraite capable de traiter les sémantiques stable et préférée et des requêtes de vérification d'extension, d'acceptabilité crédule et sceptique. Au terme de ce travail, le résultat est un outil modulaire, qui respecte le protocole d'entrée/sortie donné et s'appuie sur des choix algorithmiques et structurels justifiés.

Sur le plan de **l'architecture**, la séparation entre la représentation du graphe via *SystemeArgumentation*, les propriétés via *Utilitaires*, les algorithmes de résolution via *Semantiques* et l'interface avec *Solveur* et *Main* a permis d'isoler les préoccupations. Le recours systématique à des identifiants numériques et à des structures de vecteurs et de masques booléens a offert un bon compromis entre lisibilité du code et performances, en particulier pour les opérations ensemblistes répétées.

Les **algorithmes** de résolution reposent sur une approche par labelling et backtracking, inspirée des travaux sur les sémantiques de Dung et sur les solveurs SAT. Cette stratégie a permis de traiter des problèmes difficiles tout en conservant une implémentation maîtrisable et extensible. Les variantes du problème ne sont pas gérées comme des cas particuliers, mais comme des versions d'un même modèle de recherche, ajusté selon les contraintes de départ et quelques tests supplémentaires.

La phase de **test** et de validation a joué un rôle central. L'utilisation des systèmes donnés et d'un script Python explorant exhaustivement l'espace des sous-ensembles pour de petits graphes a permis de confronter systématiquement le comportement du solveur aux propriétés théoriques attendues. Cette démarche a permis d'identifier certains bugs, puis de les corriger.

Le solveur obtenu constitue une base solide pour des expériences supplémentaires (nouvelles sémantiques, nouveaux formats). S'il ne prétend pas rivaliser avec les solveurs spécialisés de la littérature, il offre néanmoins un compromis satisfaisant.

X. Sources

Les principales sources consultées et utilisées pour le développement et la compréhension du projet sont listées ci-dessous via des liens hyper-textes.

Dans ***SystemeArgumentation*** :

- Le choix de *std::vector* plutôt que *std::set* est justifié par le fait que *std::vector* est souvent plus rapide que *std::list* ou *std::set* même pour des insertions, grâce à la localité du cache :
[*Why You Should AVOID Linked Lists*](#) - Par ThePrimeTime et reprenant la conférence de Bjarne Stroustrup.
[*C++ benchmark – std::vector VS std::list VS std::deque*](#) - Par Baptiste Wicht.
[*Advantages of vector vs. linked list & The common std::vector layout*](#) - Posts Stackoverflow.
- L'approche par indexation numérique s'aligne sur les pratiques courantes, notamment observées dans les solveurs de la compétition ICCMA, où la manipulation d'entiers est un pré-requis pour l'efficacité des algorithmes de résolution.
[*Site de l'International Competition on Computational Models of Argumentation*](#) - Regroupant divers papiers justifiant les choix des participants.
[*CoQuiAAS : Applications de la programmation par contraintes à l'argumentation abstraite*](#) - Détail sur le fonctionnement de CoQuiAAS, par Jean-Marie Lagniez, Emmanuel Lonca et Jean-Guy Mailly.
- Le recours aux listes d'adjacence plutôt qu'aux matrices s'explique par les recommandations de la littérature algorithmique : elles permettent une utilisation mémoire bien plus économe que la complexité quadratique associée aux matrices d'adjacence.
[*Representation of Graph*](#) - Par geeksforgeeks.org.

Dans *Utilitaires* :

- L'utilisation de tableaux de booléens pour vérifier l'appartenance à un ensemble S permet des tests d'appartenance à un ensemble en temps constant, bien plus efficace que la recherche par dichotomie.
[Direct Address Table](#) - Par [geeksforgeeks.org](#).
[Learn Bits and How To Use std::bitset In C++](#) - Par [Learn C++](#).
- Les algorithmes de la STL sur des vecteurs triés (*std::includes*, *std::set_intersection*, ...) assurent une complexité optimale, et bénéficient également des optimisations bas niveau du compilateur, souvent supérieures aux boucles manuelles.
[Vue d'ensemble de la bibliothèque standard C++ \(STL\)](#) - Par [Microsoft](#)
[Standard Containers](#) - [Documentation C++](#)

Dans *Semantiques* :

- Pour la résolution des problèmes complexes, nous avons opté pour une approche de backtracking. Cette technique est le standard pour l'exploration d'arbres de recherche dans les problèmes NP-complet ou les problèmes de satisfaction de contraintes, car permettant d'élaguer rapidement les branches invalides.
[Backtracking Algorithms](#) - Par [geeksforgeeks.org](#)
[Constraint Satisfaction Problems \(CSP\) in Artificial Intelligence](#) - Par [geeksforgeeks.org](#)
[Pruning decision trees](#) - Par [geeksforgeeks.org](#)
- Les références justifiant les méthodologies et algorithmes employés dans les différentes sémantiques sont détaillées ci-dessous. Une même source peut donner des méthodologies à propos de différents problèmes d'extensions.
[ASPARTIX: Implementing Argumentation Frameworks Using Answer-Set Programming](#) - Par Uwe Egly, Sarah Alice Gaggl, et Stefan Woltran
[A Four-Label-Based Algorithm for Solving Stable Extension Enumeration in Abstract Argumentation Frameworks](#) - Par Hubei University of Technology
[Minisat \(open-source SAT solver\)](#) - par Niklas Eén et Niklas Sörensson
[Algorithms for decision problems in argument systems under preferred semantics](#) - Par Par Samer Nofal, Katie Atkinson et Paul Dunne
[Complexity-sensitive decision procedures for abstract argumentation](#) - Par Wolfgang Dvořák, Matti Järvisalo, Johannes Peter Wallner et Stefan Woltran
[CoQuiAAS: Applications de la programmation par contraintes à l'argumentation abstraite](#) - Détail sur le fonctionnement de CoQuiAAS, par Jean-Marie Lagniez, Emmanuel Lonca et Jean-Guy Mailly
[Site de l'International Competition on Computational Models of Argumentation](#) - Regroupant divers papiers justifiant les choix des participants
[A Randomized Approach to Reasoning in Argumentation Frameworks Based on Random Walks](#) - Par Dominik Hillmann
[ICCMA Benchmarks Repository](#) - Sur GitHub