I Problématique

Les débats sont omniprésents à l'heure des réseaux sociaux, que ce soit au sujet des programmes des candidats aux élections, des forces et faiblesses des derniers blockbusters hollywoodiens, ou à peu près n'importe quel sujet imaginable. Dans la vie de tous les jours, on débat pour faire des choix même sur des choses simples (Où manger ce soir ? Quel film aller voir au cinéma ?). Le but du projet est de proposer une modélisation de débats simples, et une méthode de résolution de tels débats.
Le but du projet est donc de développer un logiciel qui permet :

1) de représenter les arguments et les contradictions entre eux ;

2) de construire (manuellement, ou automatiquement) une solution admissible ou préférée ;

3) de vérifier si un ensemble d'arguments est une solution admissible ou préférée.

II Modélisation

On rappelle qu'un *graphe orienté* est composé d'un ensemble de *noeuds*, qui sont reliés par des *arcs*. Formellement, un tel graphe est noté $G = (N, A)$, où N est l'ensemble des noeuds, et $A \subseteq N \times N$ est l'ensemble des arcs, c'est-à-dire que chaque arc d'un noeud x vers un noeud y est représenté comme un couple de noeuds (x,y). On peut représenter un graphe de manière visuelle, comme par exemple la Figure 1.

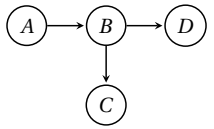

FIGURE 1 – Un exemple de graphe orienté

On peut représenter un débat par un graphe orienté, dont les noeuds sont les arguments et les arcs sont les contradictions. Le graphe de la Figure 1 peut, par exemple, représenter le débat suivant :

C : « On pourrait aller faire un tour au jardin du Luxembourg demain matin. »

D : « Ce serait sympa si on faisait un pique-nique demain midi. »

B : « J'ai cru comprendre qu'il allait pleuvoir demain, donc on ne pourra pas faire d'activités en extérieur. »

A : « Selon le site de Météo France, la pluie est uniquement prévue en fin d'après-midi, donc on n'aura pas de problème. »

Étant donné un graphe de débat, un ensemble d'arguments $E \subseteq N$ est **une solution admissible** du débat si et seulement si

- il n'y a pas deux arguments dans E qui se contredisent ;
- et pour tout argument a qui contredit un élément de E, il existe un élément de E qui contredit a ; on dit alors que E *se défend* contre a.

Formellement, on doit respecter les deux conditions :

- $\forall a, b \in E$, $(a,b) \notin A$ (absence de contradiction) ;
- $\forall a \in E$, $\forall b \in N$, si $(b,a) \in A$ alors $\exists c \in E$ tel que $(c,b) \in A$ (défense).

Par exemple, si on considère le débat représenté par la Figure 2, les solutions admissibles sont :

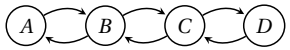

FIGURE 2 – Un exemple de graphe de débat

- $\emptyset$, (l'ensemble vide est toujours une solution admissible, car trivialement il n'a pas de contradiction interne, et il n'a pas besoin de se défendre puisqu'il n'est pas contredit par quoi que ce soit),
- $\{A\}$, $\{B\}$, $\{C\}$, $\{D\}$, (chaque argument est en mesure de se défendre lui même contre toutes les contradictions qu'il reçoit),
- $\{A,C\}$, $\{A,D\}$, $\{B,D\}$, (ce sont les seuls ensembles de deux arguments sans contradiction interne).

Ensuite, **une solution préférée** E est une solution admissible qui est dite *maximale*, c'est-à-dire qu'il n'existe pas de solution admissible E' telle que $E \subset E'$.
Si on continue l'exemple précédent, les solutions préférées sont les ensembles $\{A,C\}$, $\{A,D\}$ et $\{B,D\}$, car ce sont les seules solutions admissibles qui ne sont pas incluses dans d'autres solutions admissibles (par exemple, $\{A\}$ est admissible mais n'est pas préférée, car $\{A\} \subset \{A,C\}$).
Des conseils d'implémentation concernant les graphes vous seront donnés dans un document à part, disponible sur Moodle prochainement.

III Instructions

1. Tâches à réaliser

1.a. Première phase

Dans un premier temps, vous devez développer un programme qui permet à l'utilisateur de configurer « manuellement » un débat, et de tester si un ensemble d'arguments est une solution admissible. Au démarrage, le programme doit demander à l'utilisateur le nombre d'arguments n. Pour cette première phase, nous supposerons que les arguments sont nommés automatiquement $A1$, $A2$, …, An.

Une fois que le nombre d'arguments n est fixé, un menu s'affiche avec deux options :

1) ajouter une contradiction ;

2) fin.

Dans le cas où l'option 1 est retenue, on demande à l'utilisateur les arguments entre lesquels il faut ajouter une contradiction, puis on revient au menu précédent. Naturellement, on tiendra compte de l'ordre dans lequel les arguments sont indiqués par l'utilisateur (par exemple, s'il tape d'abord $A3$, puis $A5$, cela veut dire que l'argument $A3$ contredit l'argument $A5$).
Dans le cas où l'option 2 est retenue, l'utilisateur a terminé de représenter le graphe qui décrit les arguments et les contradictions. L'utilisateur peut maintenant essayer de trouver une solution au problème. Une solution potentielle va être mémorisée pendant le reste de l'exécution du programme. Cette solution est initialisée avec l'ensemble vide, et à chaque étape l'utilisateur a le choix entre ajouter un argument dans la solution potentielle, retirer un argument de cette solution potentielle, ou vérifier si cette solution potentielle est bel et bien une solution admissible. [1]
Concrètement, l'utilisateur fait face à un menu avec quatre options :

1) ajouter un argument ;

2) retirer un argument ;

3) vérifier la solution ;

4) fin.

Quand l'option 1 est sélectionnée, on demande à l'utilisateur d'indiquer le nom d'un argument qui doit être ajouté. S'il est déjà dans la solution

1. Rappelons que la solution $E = \emptyset$ est toujours admissible, pour n'importe quel graphe de débat. Cependant, il est plus intéressant de trouver des solutions admissibles non vides, et même (idéalement) maximales (c'est-à-dire préférées).

potentielle E, alors on l'indique à l'utilisateur et on ne modifie pas E. Sinon, on ajoute cet argument dans E.
Quand l'option 2 est sélectionnée, le programme demande à l'utilisateur le nom d'un argument qui doit être retiré. Si cet argument n'est pas dans E, alors on le signale à l'utilisateur et on ne modifie pas E. Sinon on retire cet argument de E.
Quand l'option 3 est sélectionnée, le programme vérifie si la valeur actuelle de E correspond à une solution admissible, et l'indique à l'utilisateur. Si ce n'est pas le cas, le programme doit indiquer pourquoi ce n'est pas admissible (par exemple, en indiquant deux arguments dans l'ensemble qui se contredisent, ou en identifiant un argument dans l'ensemble qui n'est pas défendu contre tous ses contradicteurs).
Après avoir effectué chacune des options 1, 2 ou 3, le programme rappelle la valeur actuelle de E à l'utilisateur avant de ré-afficher le menu.
Enfin, si l'option 4 est sélectionnée, le programme s'arrête, en rappelant à l'utilisateur la valeur de E, et si c'est une solution admissible ou non.
Il est nécessaire de gérer certaines erreurs. Par exemple, quand le menu propose trois options, il faut indiquer à l'utilisateur que sa réponse est incorrecte s'il essaye de taper 4. Par contre, il n'est pas demandé (pour cette première phase) de gérer les erreurs dues à l'entrée d'une information de mauvais type (par exemple si l'utilisateur entre le nombre 1,5 au lieu d'un nombre entier). Ce type d'erreur sera à gérer dans la seconde partie, quand nous intègrerons la gestion des exceptions.

1.b. Seconde phase
En seconde phase, le but est d'automatiser le plus possible les traitements nécessaires à la résolution du problème. En particulier, deux choses sont nécessaires :

- l'utilisation de fichiers pour représenter un graphe de débat, il suffira alors de lire le contenu du fichier au lieu d'entrer manuellement le contenu du graphe,
- la recherche d'une solution admissible, ou préférée.

Le format utilisé pour décrire les graphes dans des fichiers sera le suivant : chaque argument sera défini via une ligne avec la syntaxe suivante :

```
argument(nom_argument).
```

et chaque contradiction sera représentée par une ligne de ce type :

```
contradiction(nom_argument_1,nom_argument_2).
```

Pour chaque ligne, nous supposerons que des espaces peuvent apparaître (par exemple, entre la parenthèse ouvrante et le nom de l'argument, ou autour de la virgule). Cela ne doit pas poser de problème lors de la lecture du fichier. Par contre, une contrainte que nous imposons pour chaque fichier : tous les arguments doivent être définis avant leur utilisation dans une contradiction. Cela signifie que si un argument n'est pas défini, ou est défini après avoir été utilisé dans une contradiction, on peut avertir l'utilisateur que le fichier est mal formé.
Un exemple de fichier complet, correspondant au graphe de débat de la Figure 1, est donné ci-dessous :

```
argument(A).
argument(B).
argument(C).
argument(D).
contradiction(A,B).
contradiction(B,C).
contradiction(B,D).
```

Les noms des arguments peuvent être des chaînes de caractères quelconques, sauf le mot `argument` et le mot `contradiction`, qui sont réservés pour la spécification des différentes lignes du fichier, et les caractères virgule (`,`) et parenthèses (`(` et `)`) sont également interdits.
Ainsi, votre programme devra prendre **en paramètre sur la ligne de commande** le nom d'un fichier. La première chose à faire sera donc de lire ce fichier pour construire le graphe de débat correspondant, ou informer

l'utilisateur que le fichier est mal formé. Dans ce cas là (ou si le fichier n'existe pas, ou pour tout autre problème lié à la lecture du fichier), le programme s'arrête après que l'utilisateur ait été informé du problème. Il est essentiel dans ce cas que le programme s'arrête « proprement », avec un message clair pour l'utilisateur, et que ça ne ressemble pas à un plantage brutal du programme. Sinon, on affiche un menu à l'utilisateur :

1) chercher une solution admissible ;

2) chercher une solution préférée ;

3) sauvegarder la solution ;

4) fin.

Si l'utilisateur choisit l'option 1 ou l'option 2, le programme calcule une solution admissible ou une solution préférée et l'affiche avec un format précis, par exemple si la solution est $E = \{A1, A3, A7\}$ l'affichage doit être :

```
A1,A3,A7
```

en une seule ligne, sans autres caractères que les virgules pour séparer les noms des arguments.
Si l'utilisateur choisit l'option 3, le programme doit sauvegarder la dernière solution dans un fichier texte dont le chemin est demandé à l'utilisateur, en utilisant le même format que pour l'affichage. Pour cela, il faut que l'option 1 ou l'option 2 ait été choisie au moins une fois avant, dans le cas contraire.
Après le choix de l'option 1, 2 ou 3, le programme revient au menu. Si l'utilisateur tape à nouveau 1 ou 2, il est préférable d'afficher une nouvelle solution admissible ou préférée, au lieu d'en afficher une qui a déjà été retournée. Bien entendu, si toutes les solutions admissibles ou préférées ont déjà été calculées et affichées, il n'est pas gênant de les répéter.
Enfin, si l'utilisateur choisit l'option 4, le programme s'arrête.

2. Remise du projet

Ce projet est à réaliser par groupes de **deux ou trois étudiants**, issus du **même groupe de TD**. Des conseils sur l'implémentation seront fournis prochainement sur Moodle.

2.a. Première phase

Votre code source, correctement documenté, sera à remettre sur Moodle au plus tard le **11 Novembre 2022** (23h59, heure de Paris), sous forme d'une archive jar ou zip (un seul dépôt par binôme/trinôme). Il n'y aura pas de correction détaillée de ce code, ni de note associée, mais il sera utilisé durant une séance de TD pour faire un point avec vous sur l'avancement du projet, et vous donner d'éventuels conseils sur des points particuliers qui vous bloquent. Vous aurez alors à faire une petite démonstration du fonctionnement de votre programme, et nous tiendrons compte dans la note finale de la réalisation d'une démonstration correcte des fonctionnalités de la première phase. Même si cette première phase n'est pas notée, il est recommandée de la faire sérieusement afin de faciliter la réalisation du travail sur la second phase.

2.b. Seconde phase

Votre code source, correctement documenté, sera à remettre sur Moodle au plus tard le **16 Décembre 2022** (23h59, heure de Paris), sous forme d'une archive jar ou zip (un seul dépôt par binôme/trinôme). En plus de l'implémentation correcte des fonctionnalités demandées, une attention particulière sera accordée à :

- la gestion des erreurs (notamment, les exceptions liées aux entrées/-sorties),
- la qualité du code et de la conception de l'application (respect des conventions de nommage, encapsulation, découpage du code en packages et classes pertinents,...),
- la qualité de la documentation du code.

Le nom de votre projet, dans Eclipse, doit être `Debat_NomsEtudiants`, par exemple `Debat_Delobelle_Mailly_Martinez` pour un projet qui serait réalisé par les enseignants responsables de l'UE.