

HMIN321: Scociété Virtuelle

Rendu : Argumentation Abstraite

Tianning MA

M2 IMAGINA

12/11/2020

Table de matière

1. Introduction	2
2. Rendu et explication des exercices	2
2.1 Arguments sur le sujet « Covid lockdown - Université »	2
2.2 Implémentation « complete extention algorithm »	4

1. Introduction

Ce compte rendu est dédié à la partie « Argumentation » pour UE Société Virtuelle. L'objectif de ce rendu est pour montrer ma compréhension, ma réflexion ainsi que mon implémentation pour l'argumentation et dialogue au sein du système multi-agents.

Vous pouvez trouver le code source de mon implémentation à l'adresse suivante :

<https://github.com/matianning/SocieteVirtuelle/tree/main/AbstractArgumentation>

2. Rendu et explication des exercices

2.1 Arguments sur le sujet « Covid lockdown - Université »

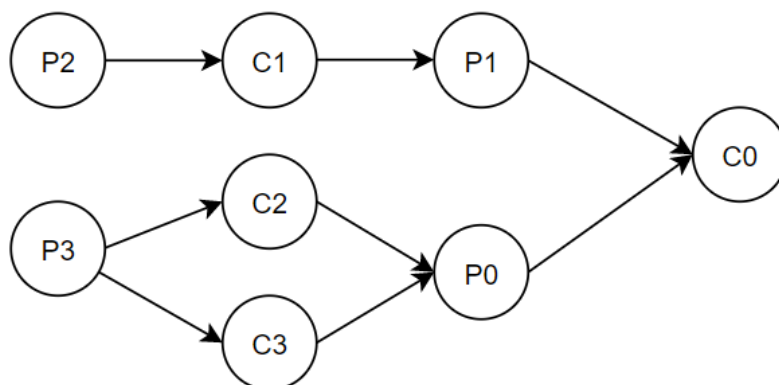
Après votre cours et des lectures des tutoriels ou articles en ligne, selon ma compréhension, un système argumentation abstraite est une collection d'arguments avec une relation : qui attaque qui.

Remarque : les arguments cités ici ne sont pas exhaustives. Ce sont les arguments selon mon point de vue.

Mon argument framework : $\{X, A\}$

$X = \{P1, P2, P3, C1, C2, C3\}$

$A = \{(P1, C0), (P0, C0), (C1, P1), (P2, C1), (C2, P0), (C3, P0), (P3, C2), (P3, C3)\}$



C0 : Le covid n'est qu'un rhyume, ce n'est pas important, il n'est pas nécessaire de fermer l'université

P0 : Le covid est très dangereux, il faut fermer l'université pour éviter la propagation.

P1 : Le Covid est dangereux et qui pourrait se propager vite entre les étudiants à l'université, car les étudiants sont nombreux dans une pièce fermée pendant les cours.

C1 : L'université a préparé les gels désinfectants et les masques pour les étudiants, le covid ne pourrait pas se propager vite dans le campus.

C2 : La fermeture de l'université ne contribue pas à l'arrêt de propagation de virus, car les étudiants pourraient se réunir en dehors de campus.

C3 : La fermeture de l'université n'est pas nécessaire, car la faculté a déjà en semi-présentiel et diminuer le nombre d'effectifs dans une salle

P2 : Les experts montrent que les gels désinfectants et les masques ne sont pas à 100% efficace contre le virus, donc il y a toujours la chance d'attraper le virus quand t'as eu un contact avec un porteur.

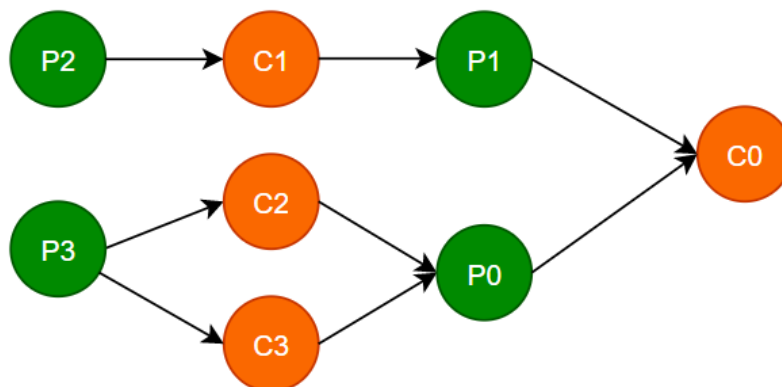
P3 : la fermeture de l'université pourrait aider à arrêter la propagation de virus, car les étudiants ne sont plus en contact avec les autres pour les cours.

Après la lecture d'un article, selon AA framework (Dung, 1995), tous les sémantiques pour « AA agree » avec l'exemple précédent est dialectiquement justifié par considérer {(Les termes en Vert)} comme arguments « gagnants » (**IN**) et les autres **OUT**.

Donc, selon les règles générales (vu en cours) pour les argumentations basiques (récursive sur le graphe) :

An argument is **IN** if all its attackers are **OUT**
An argument is **OUT** if at least one of its attackers is **IN**
Otherwise is UNDEC

J'ai obtenu le graphe comme celui-ci :



Donc, on voit les arguments {P0, P1, P2, P3} sont les arguments gagnants, et les {C0, C1, C2, C3} sont les arguments perdants.

2.2 Implémentation « complete extention algorithm »

Pour l'implémentation de cet algorithme. J'ai crée une classe « **Argument** » qui peut être identifié par un string nom.

```
class Argument{
public :
    Argument() = default;
    Argument(const std::string & a);
    Argument(const Argument & argument){name = argument.getName();}
    std::string getName() const{return name;}

private :
    std::string name;
};
```

Ensuite pour représenter le graphe d'argumentation, j'ai crée aussi une classe pour cela .

```
class ArgumentationGraph{
public :
    ArgumentationGraph() = default;

    void addArgument(const Argument & argument);
    void addAttaque(const Argument & a, const Argument & b);
    void resolve();
    void showGraph();

private :
    std::vector<std::vector<Argument>> arguments;
    /*
    Argument0      |Argument1      |Argument2 ...
    ArgumentAttaquant0      ArgumentAttaquant0      ArgumentAttaquant0
    ArgumentAttaquant1      ArgumentAttaquant1      ArgumentAttaquant1
    ArgumentAttaquant2      ArgumentAttaquant2      ArgumentAttaquant2
    ...                  ...                  ...
    */
    std::vector<Argument> In;
    std::vector<Argument> Out;
    int maxIter = 5;
};
```

Comme l'image montrée ci-dessus, dans la classe « **ArgumentationGraph** », on a un vector « arguments » pour stocker le graph, un vector « In » pour stocker les arguments gagnants (IN), et « Out » pour stocker les arguments perdants (OUT)

Nombre de itération pour résoudre ce graphe pourrait être changé, pour le moment le nombre d'itération est initialisé à 5. (Vu que le graphe pour le problème précédent n'est pas si grand)

Dans le main :

```

int main()
{
    /*
    Exemple utilisé :
    X = {P1, P2, P3, C1, C2, C3}
    A = {(P1, C0), (P0, C0), (C1, P1), (P2, C1), (C2, P0), (C3, P0), (P3, C2), (P3, C3)}
    */

    ArgumentationGraph initialFramework;
    Argument C0("C0"), C1("C1"), C2("C2"), C3("C3"),
        P0("P0"), P1("P1"), P2("P2"), P3("P3");

    initialFramework.addArgument(C0);
    initialFramework.addArgument(C1);
    initialFramework.addArgument(C2);
    initialFramework.addArgument(C3);
    initialFramework.addArgument(P0);
    initialFramework.addArgument(P1);
    initialFramework.addArgument(P2);
    initialFramework.addArgument(P3);

    initialFramework.addAttaque(P1, C0);
    initialFramework.addAttaque(P0, C0);
    initialFramework.addAttaque(C1, P1);
    initialFramework.addAttaque(P2, C1);
    initialFramework.addAttaque(C2, P0);
    initialFramework.addAttaque(C3, P0);
    initialFramework.addAttaque(P3, C2);
    initialFramework.addAttaque(P3, C3);

    initialFramework.showGraph(); //Affichage des relations
    initialFramework.resolve(); //Affichage des résultats
}

```

On entre simplement les termes et les relations, puis on lance le framework, cela pourrait sortir dans le console le résultat.

Dans le console : (résultat)

```

20:00:12: Starting D:\DeductifAgent\AbstractArgument
*****Le Graphe d'Argumentation*****
C0 : P1 P0
C1 : P2
C2 : P3
C3 : P3
P0 : C2 C3
P1 : C1
P2 :
P3 :
*****
Resolve Arguments ...
OUT Arguments : C1 C2 C3 C0
IN Arguments : P2 P3 P0 P1

```

Comparer avec le résultat précédant (dans la partie 1), on voit que les résultats finaux par le program est le même. Donc, cela nous montre que le program pour résoudre ce graphe est bien fonctionnel. Donc on a bien { P0, P1, P2, P3 } sont les arguments gagnants et {C0, C1, C2, C3} sont les arguments perdants.

Le coeur de cet algorithme est dans la fonction « resolve » :

```
void ArgumentationGraph::resolve(){
    std::cout<<"Resolve Arguments ..."<<std::endl;
    //Pour exemple : Selon AA agreee, P2 et P3 n'ont pas d'arguments attaquants, ils sont IN.
    //On va résoudre ce graphe à partir de cela.
    In.push_back(Argument("P2"));
    In.push_back(Argument("P3"));
```

```
for(int iter = 0; iter < 5; iter++){
    //*****Chercher les arguments OUT*****
    //An argument is Out if at least one of its attackers is IN
    for(size_t i = 0; i < arguments.size(); i++){
        bool out = false;
        for(size_t j = 1; j < arguments[i].size(); j++){
            for(size_t k = 0; k < In.size(); k++){
                if(arguments[i][j].getName() == In[k].getName()){
                    out = true;
                    break;
                }
            }
            if(out) break;
        }
        if(out){
            bool doublon = false;
            for(size_t w = 0; w < Out.size(); w++){
                if(Out[w].getName() == arguments[i][0].getName()){
                    doublon = true;
                    break;
                }
            }
            if(!doublon){
                Out.push_back(Argument(arguments[i][0]));
            }
        }
    }
}
```

Selon les arguments courants
(le résultat temporaire dans le
vector de IN et OUT), chercher
les arguments OUT

Selon :

An argument is OUT if at least
one of its attackers is IN

Pour « maxlter »
itération

(ici maxlter = 5)

```
//*****Chercher les arguments IN*****
for(size_t i = 0; i < arguments.size(); i++){
    bool undec = true;
    for(size_t j = 0; j < Out.size(); j++){
        if(arguments[i][0].getName() == Out[j].getName()) undec = false;
    }
    if(undec){
        bool in = true;
        for(size_t k = 1; k < arguments[i].size(); k++){
            if(in){
                for(size_t w = 0; w < Out.size(); w++){
                    if(arguments[i][k].getName() == Out[w].getName()) break;
                }
                else{
                    if(w!=Out.size()-1) continue;
                    else{
                        in = false;
                        break;
                    }
                }
            }
        }
        else break;
    }
    if(in){
        bool doublon = false;
        for(size_t ii = 0; ii < In.size(); ii++){
            if(arguments[i][0].getName() == In[ii].getName()) doublon = true;
        }
        if(!doublon)
            In.push_back(Argument(arguments[i][0]));
    }
}
}
```

Selon les arguments
courants, chercher les
arguments IN

Selon :

An argument is IN if all its
attackers are OUT

```
std::cout<<"IN Arguments : ";
for(size_t i = 0; i < In.size(); i++){
    std::cout << In[i].getName()<<" ";
}
std::cout<<std::endl;

std::cout<<"OUT Arguments : ";
for(size_t i = 0; i < Out.size(); i++){
    std::cout << Out[i].getName()<<" ";
}
std::cout<<std::endl;
```

Affichage