Bipolarité en argumentation

C. Cayrol
M.C. Lagasquie-Schiex

Février 2004

Rapport IRIT/2004-07-R

Résumé

L'argumentation est basée sur l'échange et l'évaluation d'arguments interagissant. Dans [CLS01, CLS02, CLS03a, CLS03c], en partant du cadre de travail proposé par [Dun95], nous avions proposé diverses évaluations graduelles qui utilisaient toutes l'interaction entre les arguments telle que l'a définie [Dun95] : une relation de contrariété entre arguments.

En nous inspirant des travaux de [KP01, Ver02], nous allons proposer la prise en compte d'un nouveau type d'interaction : une relation d'appui entre arguments. Ainsi, nous introduirons une nouvelle approche de l'argumentation dans un cadre *bipolaire* nous conduisant tout naturellement à de nouvelles sémantiques pour l'acceptabilité et à de nouvelles évaluations graduelles.

Table des matières

Chapitre 1

Introduction

[Dun95] a montré que le cadre de l'argumentation constitue un outil puissant permettant aussi bien l'étude de nombreux systèmes formels de raisonnement de sens commun que la définition d'une sémantique pour les programmes logiques. L'argumentation est basée sur l'échange et l'évaluation d'arguments supportant des opinions, des assertions. On trouve des applications notamment dans le domaine juridique, dans les systèmes d'aide à la prise de décision collective ou d'aide à la négociation. La caractéristique fondamentale d'un système d'argumentation est la présence d'interactions et notamment de *relations de contrariété* entre les arguments avancés. Si l'argument prend par exemple la forme d'une preuve logique, on peut avancer des arguments pour une proposition et des arguments contre cette proposition, *i.e.* pour la proposition contraire.
Si on suppose que les arguments et les interactions sont donnés, le processus d'argumentation comporte alors une étape d'*évaluation de la force relative des arguments* en présence, suivie d'une étape de *sélection des arguments les plus acceptables*, éventuellement en fonction de l'évaluation choisie.

En ce qui concerne l'étape d'évaluation, on distingue au moins 2 cas :

- une évaluation dite *intrinsèque* qui évalue un argument indépendamment des interactions avec les autres arguments. On peut ainsi exprimer à quel point l'argument augmente la confiance en l'assertion qu'il supporte. Cette évaluation peut prendre différentes formes (voir [KAEF95, Par97, PS97, AC98]).
- une évaluation *des contrariétés* selon laquelle un argument est évalué en fonction de ses "contrariants", des contrariants de ses contrariants (ses "défenseurs"), ... Plusieurs approches ont été proposées (voir [Dun95, AC98, JV99, BH01, CLS01, CLS02, CLS03a]) qui se distinguent par la richesse de l'ensemble des valeurs disponibles pour évaluer un argument. Par exemple, dans [CLS01, CLS02, CLS03a, CLS03c], nous avons proposé quelques formalismes permettant une évaluation des interactions graduelle pour les arguments (soit locale et générique recouvrant la plupart des approches existantes [BH01, JV99][1], soit globale et totalement nouvelle dans laquelle on développe un étiquetage sous la forme d'un tuple permettant de mémoriser la structure du graphe représentant les contrariétés[2]).

Évaluation intrinsèque et prise en compte des contrariétés ont très souvent été utilisées séparément, selon les applications envisagées. On trouve cependant quelques travaux qui proposent une combinaison de ces deux critères (voir par exemple [AC98] et dans une moindre mesure [CLS01]).

Quant au processus de sélection des arguments les plus acceptables, il peut prendre de nombreuses formes. Cela peut être une *acceptabilité individuelle* : par exemple, un argument est accepté car il n'a pas de contrariant. Mais, cela peut être aussi une *acceptabilité conjointe* : on détermine l'acceptabilité d'un ensemble d'arguments par l'utilisation d'une sémantique particulière (c'est-à-dire le respect par l'ensemble d'arguments de certaines règles, certaines contraintes) associé éventuellement à la prise en compte des résultats d'une évaluation spécifique. Ces différentes approches ont été étudiées par exemple dans [Dun95, Amg99, Dou02, CLS03b] et bien d'autres.

La principale critique de tous ces travaux, quelle que soit l'étape du processus d'argumentation abordée,

[1] Cette approche est essentiellement "locale" puisqu'elle consiste à trouver la valeur d'un argument uniquement en fonction de la valeur de ses contrariants.

[2] Cela se fait en associant à chaque branche sa longueur (nombre d'arcs de la feuille jusqu'au nœud courant) dans le graphe des contrariétés, sachant qu'une branche de longueur impaire est une branche d'attaque pour le nœud courant, alors qu'une branche de longueur paire est une branche de défense pour ce même nœud.

est le fait qu'il n'existe souvent qu'un seul type d'interaction possible entre arguments. Or, il est évident que de nombreux exemples réels nécessitent de pouvoir représenter au moins 2 types d'interaction : des arguments peuvent en attaquer d'autres et des arguments peuvent en aider d'autres.
Par exemple, dans les travaux portant sur le système HERMES (*cf.* [KP01]), l'interaction entre des positions (ou points de vue) permet de compléter l'information disponible en vue de prendre une décision. Une nouvelle position est avancée soit pour appuyer, soit pour apporter une objection sur une position précédemment avancée. De même, dans les travaux plus récents sur le système DEFLOG (*cf.* [Ver02]), un appui ou une attaque entre assertions peut s'exprimer dans le langage, au moyen d'une nouvelle assertion.

Ces 2 types d'interactions évoquent la notion de *bipolarité* dans la représentation et l'utilisation des interactions entre arguments. C'est-à-dire le fait qu'il faut prendre en compte deux éléments indépendants, de nature opposée et représentant des forces qui se repoussent.
Notons que ce n'est pas le seul aspect de la bipolarité en argumentation puisqu'on retrouve une notion de bipolarité lors de la construction des arguments et aussi lors de la phase de sélection (voir [ACLS04]) même si on ne s'appuie pas sur deux types d'interactions différents.

Dans ce document, nous nous proposons donc d'introduire dans un cadre formel (celui de [Dun95]) un nouveau type d'interaction correspondant à un appui entre arguments. Ainsi, nous définirons un *système d'argumentation bipolaire* par opposition au système d'argumentation unipolaire proposé par [Dun95], la bipolarité étant due ici à la prise en compte simultanée de deux types d'interactions.
Ce faisant, nous proposerons un cadre plus général que celui de [Dun95] pouvant aussi englober celui consistant à définir des arguments "pour" et des arguments "contre" et à raisonner sur eux.
Puis à l'aide de ce modèle formel, nous définirons de nouvelles sémantiques pour l'acceptabilité et de nouvelles évaluations si possible graduelles prenant en compte les deux interactions.
Le plan de ce document est le suivant :
- en section 2 page suivante, nous rappellerons le cadre pour l'argumentation défini par [Dun95] ;
- en section 3 page 7, nous montrerons sur quelques exemples réels l'intérêt d'avoir deux types d'interaction ;
- en section 4 page 11, nous décrirons les quelques rares systèmes d'argumentation bipolaires existants ;
- en section 5 page 15, nous proposerons notre propre définition d'un système d'argumentation bipolaire ;
- en section 6 page 29, nous montrerons comment étendre la notion d'acceptabilité à un tel système ;
- en section 7 page 35, nous définirons une évaluation locale générique dans un système d'argumentation bipolaire et nous proposerons différentes instances de cette évaluation ;
- en section 8 page 49, nous définirons une évaluation globale dans un système d'argumentation bipolaire ;
- en section 9 page 67, nous conclurons.

Notons que de manière systématique les preuves des propriétés, théorèmes, ... seront données en annexe C page 83.

Chapitre 2

Un système d'argumentation sans bipolarité proposé par [Dun95]

Un des cadres formels proposés pour l'argumentation est le cadre abstrait défini par [Dun95].

2.1 Cadre abstrait de [Dun95] pour l'argumentation

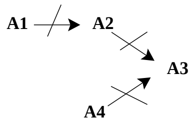
[Dun95] propose une représentation graphique facilement utilisable et de nombreuses propriétés mais dans le contexte d'une argumentation n'utilisant qu'un seul type d'interaction entre les arguments : la contrariété. Dans cette section, nous rappelons rapidement les éléments de base de ce travail.

Définition 1 *Un* système d'argumentation *est un couple* $<\mathcal{A},\mathcal{R}>$ *avec* $\mathcal{A}$ *un ensemble d'arguments et* $\mathcal{R}$ *une relation binaire sur* $\mathcal{A}$ *appelée* relation de contrariété.
Soit A_i *et* $A_j \in \mathcal{A}$, $A_i\mathcal{R}A_j$ *signifiera que* A_j *est contrarié par* A_i, *ou que* A_i *contrarie* A_j *(aussi noté* $(A_i, A_j) \in \mathcal{R}$*).*
Un système d'argumentation sera dit bien-fondé *si et seulement s'il n'existe pas de séquence infinie* A_0, A_1, $\ldots$, A_n, $\ldots$ *telle que* $\forall i, A_i \in \mathcal{A}$ *et* $A_{i+1}\mathcal{R}A_i$.

Nous ne préciserons pas davantage le format des arguments, ni la relation de contrariété.

Notations : $<\mathcal{A},\mathcal{R}>$ définit un graphe orienté $\mathcal{G}$ (dit *graphe des contrariétés*) dans lequel $A_i\mathcal{R}A_j$ sera noté par $A_i \not\rightarrow A_j$.
Soit $A \in \mathcal{A}$, l'ensemble $\{A_i \in \mathcal{A} | A_i\mathcal{R}A\}$ est noté $\mathcal{R}^-(A)$ et l'ensemble $\{A_i \in \mathcal{A} | A\mathcal{R}A_i\}$ est noté $\mathcal{R}^+(A)$.

Exemple Le système $<\mathcal{A} = \{A_1, A_2, A_3, A_4\}, \mathcal{R} = \{(A_2, A_3), (A_4, A_3), (A_1, A_2)\}>$ définit le graphe $\mathcal{G}$ suivant ayant A_3 pour racine :



Définition 2 (Représentation graphique du système d'argumentation) *Soit* $\mathcal{G}$ *le graphe des contrariétés associé à un système d'argumentation* $<\mathcal{A},\mathcal{R}>$, *on définit :*

Feuille du graphe des contrariétés *Un argument* $A \in \mathcal{A}$ *tel que* $\mathcal{R}^-(A) = \varnothing$ *sera une* feuille de $\mathcal{G}$.

Chemin dans le graphe des contrariétés *Un* chemin de A vers B *est une suite d'arguments* $\mathcal{C} = A_1 - \ldots - A_n$ *telle que :*
- $A = A_1$,
- $A_1\mathcal{R}A_2$,
- $\ldots$,
- $A_{n-1}\mathcal{R}A_n$,
- $A_n = B$.

La longueur de ce chemin *est alors $n-1$ (le nombre d'arcs constituant ce chemin) et sera notée $l_{\mathcal{C}}$.*

L'ensemble des chemins de A vers B sera noté $\mathcal{C}(A, B)$.

Dépendance, indépendance, racine-dépendance d'un chemin *Soit 2 chemins $\mathcal{C}_A \in \mathcal{C}(A_1, A_n)$ et $\mathcal{C}_B \in \mathcal{C}(B_1, B_m)$.*

Ces deux chemins seront dits dépendants *ssi $\exists A_i \in \mathcal{C}_A$, $\exists B_j \in \mathcal{C}_B$ tel que $A_i = B_j$.* Indépendants *sinon.*

Ces deux chemins seront dits racine-dépendants en A_n *ssi $A_n = B_m$ et $\forall A_i \neq A_n \in \mathcal{C}_A$, $\nexists B_j \in \mathcal{C}_B$ tel que $A_i = B_j$.*

Circuits dans le graphe des contrariétés *Un* circuit[1] *est un chemin $\mathcal{C} = A_1 - \ldots - A_n - A_1$ tel que $\forall i, j \in [1, n], i \neq j$, $\nexists A_i, A_j \in \mathcal{C}$ tels que $A_i = A_j$.*

Un circuit est isolé *quand aucun des arguments le composant n'a d'attaquant en dehors du circuit.*

Deux circuits $\mathcal{C}_A = A_1 - \ldots - A_n - A_1$ et $\mathcal{C}_B = B_1 - \ldots - B_m - B_1$ sont interconnectés *ssi $\exists i \in [1, n], \exists j \in [1, m]$ tels que $A_i = B_j$.*

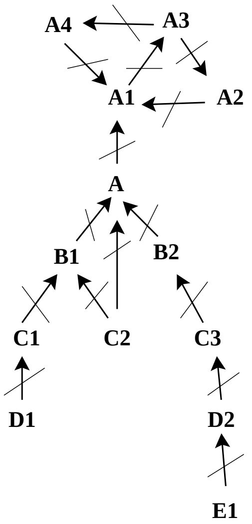
Nous allons utiliser les notions d'attaque et de défense directe ou indirecte inspirées par [Dun95] mais légèrement modifiées par nos soins[2]. On a donc :

Définition 3 (Attaquants/Défenseurs directs/indirects d'un argument) *Soit $A \in \mathcal{A}$:*
- *Les* attaquants directs *de A sont les éléments de $\mathcal{R}^-(A)$.*
- *Les* défenseurs directs *de A sont les attaquants directs des éléments de $\mathcal{R}^-(A)$.*
- *Les* attaquants indirects *de A sont les éléments A_i définis par :*
$$\exists \mathcal{C} \in \mathcal{C}(A_i, A) \text{ tel que } l_{\mathcal{C}} = 2k+1, \text{ avec } k \geq 1.$$
- *Les* défenseurs indirects *de A sont les éléments A_i définis par :*
$$\exists \mathcal{C} \in \mathcal{C}(A_i, A) \text{ tel que } l_{\mathcal{C}} = 2k, \text{ avec } k \geq 2.$$

On dira plus généralement que, si l'argument A est un attaquant (direct ou indirect) de l'argument B, alors *A attaque B* (ou *B est attaqué par A*). De même, si l'argument A est un défenseur (direct ou indirect) de l'argument B, alors *A défend B* (ou *B est défendu par A*).
Attention, bien qu'il y ait ici 2 notions qui paraissent opposées (l'attaque et la défense), il ne s'agit pas de bipolarité puisque la notion de défense est construite à partir de celle d'attaque (ce n'est pas une notion indépendante).

Définition 4 (Branches d'attaque et de défense d'un argument) *Soit $A \in \mathcal{A}$, une* branche d'attaque *(resp.* de défense*) pour A est un chemin dans $\mathcal{G}$ d'une feuille vers A de longueur impaire (resp. paire). On dira alors que A est* racine d'une branche d'attaque *(resp.* de défense*).*

Toutes ces notions sont illustrées sur l'exemple suivant :



Sur ce graphe $\mathcal{G}$, on a donc (entre autres) :
- un chemin de C_2 vers A de longueur 2 ($C_2 - B_1 - A$),
- 2 circuits $A_1 - A_3 - A_2 - A_1$ et $A_1 - A_3 - A_4 - A_1$, chacun de longueur 3 qui ne sont pas isolés (remarquons que $A_1 - A_3 - A_2 - A_1 - A_3 - A_4 - A_1$ n'est pas un circuit d'après notre définition),
- les deux circuits cités précédemment sont interconnectés (en A_1 et A_3),
- les chemins $D_1 - C_1 - B_1$ et $C_3 - B_2 - A$ sont indépendants, alors que $D_1 - C_1 - B_1 - A$ et $C_3 - B_2 - A$ sont racine-dépendants et que $D_1 - C_1 - B_1 - A$ et $C_2 - B_1 - A$ sont dépendants,
- D_1, C_2, E_1 sont les feuilles de $\mathcal{G}$,
- $D_1 - C_1 - B_1 - A$ est une branche d'attaque pour A, alors que $C_2 - B_1 - A$ est une branche de défense pour A,
- B_1 et B_2 sont les deux attaquants directs de A,
- C_1, C_2 et C_3 sont les trois défenseurs directs de A,
- D_1 et D_2 sont les deux attaquants indirects de A,
- E_1 est le seul défenseur indirect de A.

[1] Cette définition d'un circuit correspond à la définition d'un circuit élémentaire en théorie des graphes (ne contient pas 2 arcs ayant la même origine, ou le même but).

[2] Les notions définies dans [Dun95] sont celles d'attaque et de défense indirecte, sachant que pour [Dun95] un attaquant (resp. défenseur) direct est aussi un attaquant (resp. défenseur) indirect, ce qui n'est pas le cas ici.

2.2 Acceptabilité collective de [Dun95]

[Dun95] propose diverses sémantiques pour l'acceptabilité des arguments. L'idée exploitée ici est que l'acceptabilité d'un argument dépend de son appartenance à certains ensembles (dits *ensembles acceptables* ou *extensions*) caractérisés par des propriétés telles que :

Définition 5 (Propriétés de base des extensions d'après [Dun95]) *Soit* $<\mathcal{A}, \mathcal{R}>$ *un système d'argumentation, on a :*

Ensemble sans conflit *Un ensemble* $E \subseteq \mathcal{A}$ *est* sans conflit *si et seulement si* $\nexists A, B \in E$ *tel que* $A\mathcal{R}B$.

Défense collective *Soit* $E \subseteq \mathcal{A}$, $A \in \mathcal{A}$. E défend (collectivement) A *si et seulement si* $\forall B \in \mathcal{A}$, *si* $B\mathcal{R}A, \exists C \in E$ *tel que* $C\mathcal{R}B$. E défend tous ses éléments *si et seulement si* $\forall A \in E$, *si* $\exists B \in \mathcal{A}$ *tel que* $B\mathcal{R}A$ *alors* $\exists C \in E$ *tel que* $C\mathcal{R}B$.

[Dun95] définit plusieurs sémantiques pour l'acceptabilité collective dont les *sémantiques admissible*, *préférée* et *stable* (avec, pour extensions respectives, les ensembles admissibles, les extensions préférées et les extensions stables) :

Définition 6 (Quelques extensions d'après [Dun95]) *Soit* $<\mathcal{A}, \mathcal{R}>$ *un système d'argumentation.*

Ensemble admissible *Un ensemble* $E \subseteq \mathcal{A}$ *est* admissible *si et seulement si* E *est sans conflit et* E *défend tous ses éléments.*

Extension préférée *Un ensemble* $E \subseteq \mathcal{A}$ *est une* extension préférée *si et seulement si* E *est maximal pour l'inclusion parmi les ensembles admissibles.*

Extension stable *Un ensemble* $E \subseteq \mathcal{A}$ *est une* extension stable *si et seulement si* E *est sans conflit et* E *contrarie tout argument n'appartenant pas à* E *(c'est-à-dire* $\forall A \in \mathcal{A} \setminus E$, $\exists B \in E$ *tel que* $B\mathcal{R}A$*).*

Remarquons que l'acceptabilité d'un argument au sens de Dung est basée sur la prise en compte de chaque contrariant de cet argument pris séparément (il n'y a pas de notion de contrariété globale sur un argument). Nous rappelons aussi quelques propriétés essentielles :

Propriété 1 (Existence d'extension [Dun95]) *Soit* $<\mathcal{A}, \mathcal{R}>$ *un système d'argumentation, on a :*

1. *Tout ensemble admissible de* $<\mathcal{A}, \mathcal{R}>$ *est contenu dans une extension préférée de* $<\mathcal{A}, \mathcal{R}>$.
2. $<\mathcal{A}, \mathcal{R}>$ *possède au moins une extension préférée.*
3. *Si* $<\mathcal{A}, \mathcal{R}>$ *est bien-fondé alors il possède une et une seule extension préférée qui est aussi la seule extension stable.*
4. *Toute extension stable est aussi une extension préférée (et non vice-versa).*
5. *Il n'existe pas toujours d'extension stable.*

Chapitre 3

Des exemples bipolaires dans la réalité

Dans ce chapitre, nous allons présenter trois exemples différents issus de situations réelles. Ces trois exemples mettent tous en œuvre un processus d'argumentation bipolaire :
- il s'agit chaque fois d'une "discussion" entre plusieurs agents,
- cette "discussion" a un but (une conclusion à déduire, une décision à prendre),
- au cours de cette "discussion", il y a échange de points de vue, certains étant en accord et d'autres étant en opposition.

3.1 L'exemple de l'orientation

Le contexte utilisé ici est celui d'une discussion entre 2 conseillers d'orientation (O_1 et O_2), et l'enjeu est l'orientation ou pas en IUT d'un individu X. Cet exemple va être présenté sous la forme d'un dialogue.

O_1 **:** Il faut orienter X vers un IUT.

O_2 **:** Attention, un IUT est une formation très courte.

O_1 **:** Pas forcément ! Avec les nouvelles licences professionnelles, cela devient une formation en trois ans.

O_2 **:** Que ce soit en deux ou en trois ans, cela reste une formation qui manque d'aspects théoriques solides. (Cela risque de gêner X plus tard, dans sa vie professionnelle.)

O_1 **:** Ce n'est pas bien grave, car X pourra éventuellement compléter ses connaissances par un retour dans un cycle universitaire long plus pointu au niveau théorique.

O_2 **:** Elle manque aussi de connaissances générales. (Et là, l'impact futur concernera aussi bien la vie professionnelle que la vie de tous les jours.)

O_1 **:** Là-aussi, ce n'est pas très gênant. Les meilleurs de l'IUT peuvent ensuite rejoindre une école d'ingénieurs généraliste.

O_2 **:** Oui, mais sera-t-il un des meilleurs ? Son dossier actuel n'est pas en sa faveur.

O_1 **:** Bon, bon. Mais, il s'agit quand même d'une formation professionnelle. X aura donc plus de facilités pour trouver ensuite du travail.

O_2 **:** C'est vrai ! Il y a beaucoup d'heures de stages.

O_1 **:** Et aussi des contacts importants entre les entreprises et l'équipe pédagogique.

O_2 **:** Du coup, les débouchés professionnels en sont facilités.

O_1 **:** Tu vois bien que ce serait un bon choix !

O_2 **:** Il y a quand même un problème important dont il nous faut tenir compte : X n'est pas vraiment fixé sur la thématique qu'il veut suivre. Cela va donc être difficile de lui imposer une formation spécialisée dès le départ.

O_1 **:** Flûte ! J'avais oublié ce détail !

O_2 **:** Et pourtant, c'est vrai que cette formation conviendrait bien à son dossier scolaire qui n'est pas brillant.

O_1 **:** Oui, c'est un individu qui a besoin d'être encadré.

O_2 : Dans une formation générale de masse comme les premières années d'université, il va à l'échec.

...

La discussion peut ainsi continuer longtemps.

Sur cet exemple, on repère facilement les deux intervenants.
On constate aussi que les positions avancées sont beaucoup moins manichéennes que dans un contexte d'argumentation classique où seule l'attaque serait possible. En effet, bien que l'opinion de O_1 soit en permanence en faveur de l'orientation en IUT, celle de O_2 varie au cours du temps : d'abord hostile à ce choix, puis reconnaissant l'intérêt de l'aspect professionnel de la formation et enfin revenant à son opposition initiale. On a donc ici des "arguments" (au sens informel du terme : des pièces d'information) qui s'opposent ou qui s'appuient.
On voit aussi que les deux intervenants s'appuient parfois sur des connaissances qui ne sont pas toujours exprimées explicitement (nous en avons mis certaines entre parenthèses) :
- une formation courte peut être problématique,
- il est mieux pour une formation de fournir des débouchés professionnels,
- le manque de théorie et de connaissances générales est un problème pour le futur,
- on ne peut pas s'engager dans une spécialisation si on ne sait pas ce sur quoi on veut travailler.

Cet aspect sera à prendre en compte lors de la formalisation du problème (par exemple sous la forme de connaissances communes).

3.2 L'exemple de la prothèse du genou

Ici, le contexte est celui d'une discussion entre deux médecins (M_1 et M_2) au sujet d'un patient X souffrant d'un grave problème au genou.

M_1 : Il faut poser une prothèse du genou à X.

M_2 : Attention, pour cela il faut une anesthésie et il y a actuellement un gros risque de complications dues à cette anesthésie.

M_1 : Oui, si on fait une anesthésie générale. Mais on pourrait se contenter d'une anesthésie locale.

M_2 : Il reste quand même les risques d'infections post-opératoires.

M_1 : C'est vrai. Il y a actuellement une recrudescence des risques d'infections nosocomiales dans l'hôpital.

M_2 : Et en plus, le patient est bien trop jeune pour avoir déjà une prothèse.

M_1 : Oui X est très jeune, mais c'est aussi un sportif de haut niveau pour lequel son genou est un outil indispensable.

M_2 : Et alors ?

M_1 : La prothèse lui permettrait de récupérer très vite sa motricité.

M_2 : Mais pourquoi ne pas utiliser un autre traitement ?

M_1 : Tous les traitements classiques sont en échec. On a tout essayé et rien ne marche.

M_2 : Non, on n'a pas encore tout essayé. Il existe un nouveau traitement expérimental.

M_1 : Les expériences, les expériences, on ne sait pas où cela risque de nous conduire !

...

On retrouve sur cet exemple toutes les remarques faites sur l'exemple précédent :
- plusieurs intervenants,
- qui échangent des "arguments" sous la forme de pièces d'information,
- qui peuvent être tantôt en accord, tantôt en conflit.

Dans ce cas aussi, il faudra être prudent lors de la traduction de cet exemple en langage formel, pour éviter d'oublier des connaissances génériques que les intervenants ne pensent pas forcément à expliciter.

3.3 L'exemple de l'enquête policière

Sherlock Holmes (SH) et le Dr Watson (DW) enquêtent sur le meurtre de Marmaduke Brandford, un riche vendeur d'objets d'arts. Son corps a été trouvé dans la bibliothèque. Il a été frappé dans le dos avec un couteau de cuisine. Il y a au moins trois suspects possibles : Mme Benton, la cuisinière, Partridge, le maître d'hôtel, et Fenimore Brandford, le neveu de la victime.

DW : Un couteau de cuisine ... Je soupçonne la cuisinière, Mme Benton. Qu'en pensez-vous Holmes ?

SH : Ne trouvez-vous pas cela un peu trop simpliste ? Bien que Mme Benton ait eu de nombreux motifs de tuer son patron (c'était un homme extrêmement violent), elle était absente hier soir. Par contre, Partridge, le maître d'hôtel, était bien là !

DW : Vous devez avoir raison, Holmes. Mais comment expliquez-vous alors le gras qui est sur le manche du couteau ? Une cuisinière a toujours les mains grasses, n'est-ce pas ?

SH : Au contraire, mon cher Watson ! Si Mme Benton avait tué Brandford, elle n'aurait sûrement pas laissé ces marques de gras sur le couteau.

DW : Oui, je suppose ... Mais je ne peux imaginer Partridge assassinant son maître après l'avoir servi pendant trente ans.

SH : Moi non plus ! Par contre, n'oublions pas que Fenimore, le neveu de Brandford, héritera de la fortune de son oncle, alors que Mme Benton et Partridge n'auront rien.

DW : Fenimore Brandford ... pourquoi pas après tout ? Il fait un suspect parfait. Par contre, cela n'explique pas le gras sur le couteau.

SH : Si, Watson, si ! Je suppose que vous savez que Fenimore Brandford a perdu son bras droit dans le Transvaal durant la guerre avec les Zoulous ...

DW : Oui et je me souviens même qu'il avait été décoré de la "Victoria Cross" pour bravoure. Toutefois, je ne vois pas le rapport ...

SH : Fenimore Brandford a une prothèse, Watson, un bras mécanique qu'il doit graisser méticuleusement chaque jour afin qu'il ne grince pas et ...

DW : Le gras sur le couteau ! Bien-sûr ! Vous êtes un génie, Holmes !

SH : Élémentaire, mon cher Watson ...

Ici encore, on retrouve les principaux éléments correspondant à une argumentation bipolaire.

Chapitre 4

Bipolarité d'interactions en argumentation : l'existant

Il existe à notre connaissance deux exemples de travaux utilisant une notion de bipolarité d'interactions en argumentation : [KP01, Ver02].
Le premier se situe au niveau de l'étape d'évaluation et utilise deux types d'interactions pour calculer les valeurs des arguments.
Le second se situe au niveau de l'étape de sélection et utilise deux types d'interactions pour déterminer si un argument appartient ou pas à une extension (un ensemble d'arguments acceptables).

4.1 Le système HERMES de [KP01]

Il s'agit d'un système d'argumentation permettant des discussions sur le Web, entre groupes d'agents, en vue de prendre une décision. Ce système autorise l'expression et la pondération d'arguments et de préférences entre arguments, et offre des mécanismes de conduite d'une discussion, de vérification de cohérence des préférences et de valuation des arguments.
Les éléments de base sont les issues, les solutions ("alternative"), les positions et les contraintes.
- Une *issue* est une question dont la réponse est ouverte à la discussion. Par exemple, en présence de la pathologie X chez le patient Y, quel est le traitement approprié ? Une issue rassemble un ensemble de *solutions*. Exemples de solutions : intervention chirurgicale, traitement par médicaments.
- Une *position* exprime l'appui, ou l'objection pour une solution, ou pour une autre position, ou encore pour une contrainte. Une position apporte une information supplémentaire dans la discussion.
- Une *contrainte* exprime une préférence entre deux positions. Cela permettra de classer des positions.

L'objectif du système est d'étiqueter les solutions et les positions par le statut "actif" ou "inactif". Une position "active" (resp. "inactive") est considérée acceptée (resp. rejetée) d'après la discussion qui s'y rapporte. Une solution "active" est un choix recommandé parmi les autres solutions d'une même issue.
Différentes procédures d'étiquetage ont été proposées dans HERMES. Ce sont des procédures récursives qui calculent l'étiquette d'un élément à partir des étiquettes des éléments qui s'y rapportent dans le graphe de la discussion. Voici les caractéristiques communes à ces procédures :
- les cycles sont exclus d'un graphe de discussion,
- l'évaluation d'une position ne prend en compte que les positions actives qui s'y rapportent,
- l'évaluation d'une position est toujours binaire, même lorsque les contraintes de préférence sont prises en compte.

Procédure 1 : Une position est active si et seulement si elle ne reçoit aucun appui ni aucune attaque, ou bien si elle est appuyée par une position active.

Procédure 2 : Une position est active si et seulement si elle n'est attaquée par aucune position active.

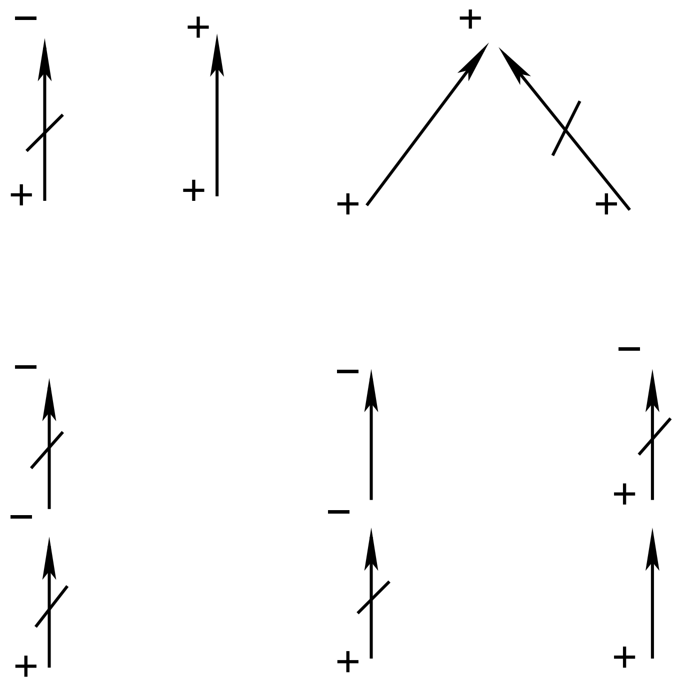
On peut remarquer que les procédures 1 et 2 ne permettent pas de prendre en compte à la fois les appuis et les attaques sur une position.

Procédure 3 : Une position est active si et seulement si elle ne reçoit aucun appui ni aucune attaque, ou bien si son score est strictement positif. Le score s'obtient par la différence entre la somme des poids des positions actives qui appuient et la somme des poids des positions actives qui attaquent. Chaque position reçoit au départ de la discussion un poids intrinsèque identique, c'est la prise en compte de contraintes de préférence entre les positions qui modifie les poids relatifs des différentes positions.

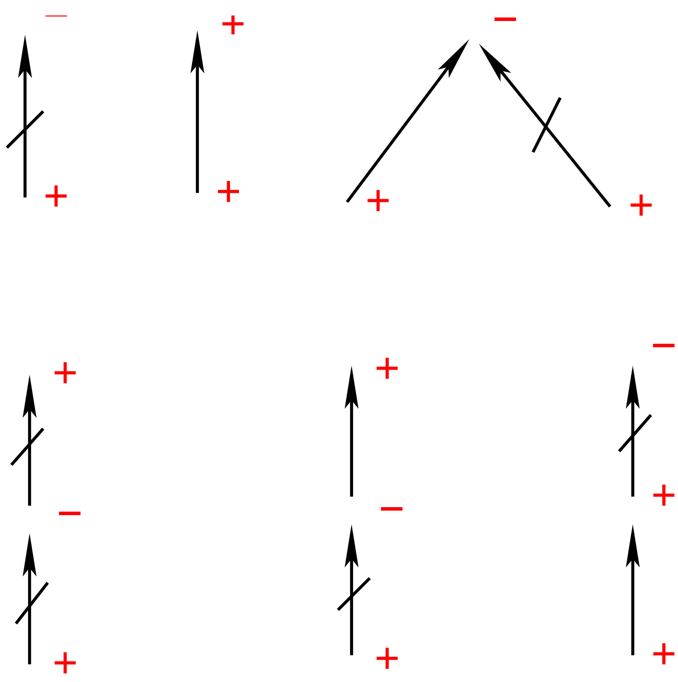
Dans la procédure 3, on peut remarquer que le score d'une position ne dépend pas du score des positions actives qui s'y rapportent mais uniquement de leurs poids relatifs.

Exemples :
Dans les exemples ci-dessous, nous noterons + (resp. -) une position "active" (resp. "inactive").

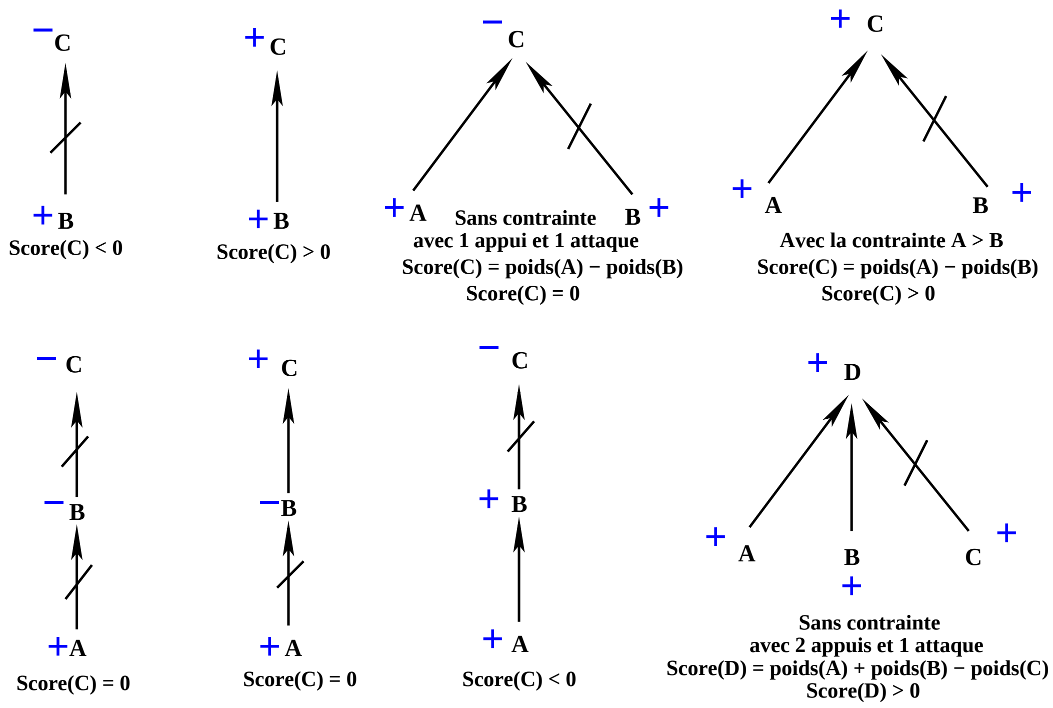
Procédure 1 :



Procédure 2 :



Procédure 3 :



4.2 Le système DEFLOG de [Ver02]

Le système DEFLOG manipule des assertions, c'est-à-dire des formules d'un langage. Un appui ou une attaque entre assertions peut ainsi s'exprimer dans le langage, au moyen d'une nouvelle assertion utilisant un connecteur spécifique du type d'interaction. Les deux relations d'appui et de contrariété ne sont donc pas figées. Exemples d'assertions (avec $\not\rightarrow$ symbolisant la relation d'attaque et $\rightarrow$ celle d'appui) : A, B, $(A \rightarrow B)$, $(A \not\rightarrow B)$, $(C \rightarrow (A \rightarrow B))$, $(D \not\rightarrow (A \rightarrow B))$
Les notions fondamentales sont inspirées des travaux de Dung.

Définition 7 *Soit S un ensemble d'assertions.*
– S supporte *une assertion H ssi H appartient à S ou H se déduit de S en suivant une séquence d'appuis.*
– S attaque *une assertion H ssi il existe une assertion G telle que S supporte G et $(G \not\rightarrow H)$.*
– *S est* sans conflit *ssi il n'existe pas d'assertion H telle que S à la fois supporte et attaque H.*

Le principal objectif du système DEFLOG est de définir les interprétations dialectiques d'un ensemble donné d'assertions.

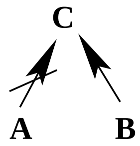
Définition 8 *Soit S un ensemble d'assertions, et (J, D) une partition de S.*
(J, D) interprète *S ssi J est sans-conflit et attaque les assertions de D.*
Si (J, D) interprète S, $(Supp(J), Att(J))$ est une interprétation dialectique *(encore appelée* extension*) de S. $Supp(J)$ (resp. $Att(J)$) dénote l'ensemble des assertions supportées (resp. attaquées) par J.*
Les assertions de $Supp(J)$ sont les assertions justifiées *et celles de $Att(J)$ les* assertions défaites.

On peut remarquer que contrairement au système d'argumentation de Dung, la relation de contrariété n'est pas figée dans le système, mais figure explicitement dans les assertions. Il en est de même pour la relation d'appui. Comme conséquence, on peut avoir une extension $(Supp(J), Att(J))$ d'une théorie S telle que certaines assertions supportées par J n'appartiennent pas à S. Si l'on traduit le système d'argumentation de Dung $<\mathcal{A}, \mathcal{R}>$ dans DEFLOG, on obtient comme assertions les éléments de $\mathcal{A}$ et les assertions du type $(A \not\rightarrow B)$ où $A\mathcal{R}_{att}B$. Dans ce cas particulier, les extensions calculées par DEFLOG correspondent aux extensions stables dans le système de Dung.

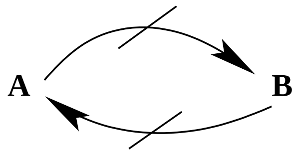
Exemples :

Soit $S = \{A, B, C, B \rightarrow C, A \not\rightarrow C\}$. Dans ce cas-là, il n'y a pas d'extension.

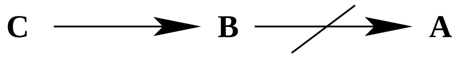
Le graphe correspondant est :



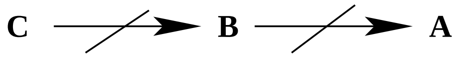
Soit $S = \{A, B, A \not\rightarrow B, B \not\rightarrow A\}$. Dans ce cas-là, il y a deux extensions :
– $(\{A, A \not\rightarrow B, B \not\rightarrow A\}, \{B\})$,
– $(\{B, A \not\rightarrow B, B \not\rightarrow A\}, \{A\})$.
Le graphe correspondant est :



Soit $S = \{A, B, C, B \not\rightarrow A, C \rightarrow B\}$. Dans ce cas-là, il y a une extension $(\{B, C, C \rightarrow B, B \not\rightarrow A\}, \{A\})$.
Le graphe correspondant est :

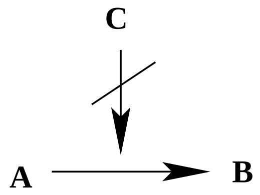


Soit $S = \{A, B, C, B \not\rightarrow A, C \not\rightarrow B\}$. Dans ce cas-là, il y a une extension $(\{A, C, C \not\rightarrow B, B \not\rightarrow A\}, \{B\})$.
Le graphe correspondant est :



Soit $S = \{B, C, B \rightarrow A, C \not\rightarrow B\}$. On peut remarquer que A n'appartient pas à S. Cela correspond à une partition de l'ensemble des assertions en ensemble d'hypothèses (éléments de S) et ensemble de questions (ou d'issues). Il s'agit de savoir si un étiquetage des hypothèses (justifiées, rejetées) permet de répondre à la question A. Dans ce cas-là, il y a une extension $(\{C, C \not\rightarrow B, B \rightarrow A\}, \{B\})$ (attention, A n'est pas supporté par $J = \{C, C \not\rightarrow B, B \rightarrow A\}$).
Le graphe correspondant est :

Soit $S = \{A, C, A \rightarrow B, C \not\rightarrow (A \rightarrow B)\}$. On remarque que $B \notin S$. B est donc la question. Dans ce cas-là, il y a une extension $(\{A, C, C \not\rightarrow (A \rightarrow B)\}, \{A \rightarrow B\})$ (on ne peut rien conclure sur B, B n'est ni justifié, ni défait).
Le graphe correspondant est :



Chapitre 5

Un nouveau système d'argumentation : le système d'argumentation bipolaire

Le cadre de travail proposé par [Dun95] dans le cas où il n'existe qu'un seul type d'interaction est abstrait dans le sens où il n'impose rien du point de vue du format des arguments, ni du point de vue de la relation de contrariété entre les arguments.
L'introduction d'un nouveau type d'interaction entre arguments pourrait se faire sous des hypothèses toutes aussi abstraites (un ensemble d'arguments et deux relations d'interaction entre ces arguments), ce qui permettrait de prendre en compte n'importe quel graphe d'interactions.
Malheureusement, cette approche nous a paru poser un problème de réalisme sur de nombreux exemples simples. Par exemple, un argument peut-il être à la fois un appui et un contrariant d'un autre argument ? Autre exemple, si l'argument A est un contrariant de l'argument B, B peut-il être un appui pour A ?
Toutes ces questions nous ont amenées à clarifier plus précisément ce qu'on choisit d'appeler "argument", et comment on définit nos deux relations d'interaction entre arguments.
Ainsi, bien que nous restions dans le cadre formel proposé par [Dun95] pour rajouter une seconde relation d'interaction, nous choisissons de perdre un peu en abstraction afin de gagner en réalisme.

5.1 Description d'une instance possible d'argumentation bipolaire

Nous allons décrire ici un exemple d'arguments et de relations d'interaction entre ces arguments. Le but poursuivi est de préciser à la fois nos idées et nos contraintes pour un cadre d'argumentation bipolaire.
Il s'agit d'arguments de type "explicatif" couramment utilisées dans le domaine du traitement de l'inconsistance dans les bases de connaissances. D'autres types d'arguments peuvent être construits, notamment dans le cadre de l'aide à la décision.

5.1.1 Une définition d'argument

Soit $\mathcal{L}$ un langage logique (par exemple, la logique des prédicats). Soit $\vdash$ la relation d'inférence associée à $\mathcal{L}$.

Définition 9 (Argument) *Un argument est un couple (S, C) avec :*

- *S un support (ensemble de formules de $\mathcal{L}$ consistant),*
- *C une formule consistante de $\mathcal{L}$,*

respectant les contraintes suivantes :
- *$S \vdash C$[1] ,*
- *S minimal pour obtenir C ($\forall \Psi_i \in S$, $S \setminus \psi_i \not\vdash C$).*

[1]Le support d'un argument peut être vu soit comme un ensemble de formules, soit comme la conjonction de ces formules (donc comme étant lui-même une formule).

Notons que cette définition est uniquement destinée à nous aider lors des sections suivantes (section 5.1.2, section 5.1.3 page 18, section 5.2 page 19) à distinguer entre les cas que nous nous autoriserons et ceux que nous nous interdirons.
Remarquons aussi que ce formalisme n'est pas aussi restreint qu'il semble l'être. Par exemple, il permet aussi d'exprimer de simples propositions ou de simples faits : soit p un fait, on peut avoir l'argument $(\{p\}, p)$.

5.1.2 Les différents types de contrariété

Avec la définition des arguments que nous nous sommes donnée, nous avons plusieurs relations de contrariété possibles. Celles qui nous semblent les plus significatives sont les suivantes (voir [Amg99] pour une étude plus approfondie des relations de contrariété).

Définition 10 (Attaque d'une partie du support de B à l'aide du support de A) *Soit A et B deux arguments. A `attaque`$_1$ B (noté $A \stackrel{1}{\not\rightarrow} B$) si et seulement si il existe X une formule telle que $S_A \vdash \neg X$ et $X \in S_B$.*

Définition 11 (Attaque d'une partie du support de B à l'aide de la conclusion de A) *Soit A et B deux arguments. A `attaque`$_2$ B (noté $A \stackrel{2}{\not\rightarrow} B$) si et seulement si $\neg C_A \in S_B$.*

Définition 12 (Attaque de la conclusion de B à l'aide de la conclusion de A) *Soit A et B deux arguments. A `attaque`$_3$ B (noté $A \stackrel{3}{\not\rightarrow} B$) si et seulement si $\neg C_A \equiv C_B$[2].*

Définition 13 (Attaque de la conclusion de B à l'aide du support de A) *Soit A et B deux arguments. A `attaque`$_4$ B (noté $A \stackrel{4}{\not\rightarrow} B$) si et seulement si il existe X une formule telle que $S_A \vdash \neg X$ et $X \equiv C_B$.*

Propriété 2 *Soit A et B deux arguments, on a :*

1. *La relation `attaque`$_1$ n'est pas symétrique.*
2. *$A \stackrel{3}{\not\rightarrow} B \Rightarrow B \stackrel{3}{\not\rightarrow} A$ (la relation `attaque`$_3$ est symétrique).*
3. *La relation `attaque`$_2$ n'est pas symétrique.*
4. *La relation `attaque`$_4$ n'est pas symétrique.*
5. *$A \stackrel{1}{\not\rightarrow} B \Rightarrow \exists C$ un argument tel que $C \stackrel{2}{\not\rightarrow} B$.*
6. *$A \stackrel{4}{\not\rightarrow} B \Rightarrow \exists C$ un argument tel que $C \stackrel{3}{\not\rightarrow} B$.*
7. *$A \stackrel{2}{\not\rightarrow} B \Rightarrow A \stackrel{1}{\not\rightarrow} B$.*
8. *$A \stackrel{3}{\not\rightarrow} B \Rightarrow A \stackrel{4}{\not\rightarrow} B$.*

Remarques :
– La relation d'`attaque`$_2$ correspond à la relation de contrariété proposée dans [EGFK93].
– De même, la relation d'`attaque`$_3$ correspond à la relation de contrariété proposée dans [Pol92, EGFK93, SL92, PS97], et connue en général sous le terme de "réfutation".
– Toute relation symétrique induit des circuits de longueur 2 entre *chaque couple d'arguments* liés par la dite relation.
– La relation d'`attaque`$_1$ (resp. d'`attaque`$_4$) induit une partie de la relation d'`attaque`$_2$ (resp. d'`attaque`$_3$).
– La relation d'`attaque`$_2$ (resp. d'`attaque`$_3$) implique la relation d'`attaque`$_1$ (resp. d'`attaque`$_4$).
Les trois dernières remarques vont orienter nos choix parmi toutes les relations de contrariété possibles. En effet, elles indiquent que les relations d'`attaque`$_1$ et d'`attaque`$_4$ sont inutiles. Cette réflexion nous mène à la série de questions/réponses suivantes :

[2] $X \equiv Y$ signifie que X est logiquement équivalent à Y.

Question Peut-on toujours remplacer $A \overset{1}{\not\rightarrow} B$ (resp. $A \overset{4}{\not\rightarrow} B$) par l'introduction de l'argument C et de l'arc $C \overset{2}{\not\rightarrow} B$ (resp. $C \overset{3}{\not\rightarrow} B$), dans le graphe des contrariétés, sans perdre de l'information ?

Réponse Oui, à condition de garder A s'il sert pour un autre arc que celui qui le liait à B.

Question Dans le contexte de la question précédente, peut-on dire que C est indépendant (non redondant avec A) ?

Réponse Oui, bien que son support soit inclus dans celui de A, sa conclusion est différente de celle de A.

Question Si on se limite aux relations d'$\texttt{attaque}_2$ et d'$\texttt{attaque}_3$ quel est l'impact sur les circuits ?

Réponse Il restera les circuits de longueur 2 induits par $\texttt{attaque}_3$ (puisque c'est une relation symétrique), mais aussi des circuits de longueur 2 induits par $\texttt{attaque}_2$:

- $A = (\{a, a \rightarrow b\}, b)$
- $B = (\{\neg b, \neg b \rightarrow \neg a\}, \neg a)$

$A \overset{2}{\not\rightarrow} B$ et $B \overset{2}{\not\rightarrow} A$. Donc, même si on se limite à $\texttt{attaque}_2$ et $\texttt{attaque}_3$, il y aura encore des circuits (et même de longueur 2).

Question Soit A et B deux arguments tels que $A \overset{2}{\not\rightarrow} B$. Est-il possible de construire systématiquement un argument A' tel que $A \overset{2}{\not\rightarrow} A'$ et $A' \overset{2}{\not\rightarrow} A$?

Réponse Cela paraît simple sur certains exemples :

- Posons $A = (\{a, a \rightarrow b\}, b)$.
- On sait qu'il existe un argument B tel que $\neg b \in S_B$, donc la formule $\neg b$ peut apparaître dans un support.
- Construisons $A' = (\{\neg b, a \rightarrow b\}, \neg a)$.
- On a bien $A \overset{2}{\not\rightarrow} A'$ et $A' \overset{2}{\not\rightarrow} A$.

Et un peu moins simple sur d'autres exemples :

- Posons $A = (\{a, b, (a \wedge b) \rightarrow c\}, c)$.
- On sait qu'il existe un argument B tel que $\neg c \in S_B$, donc la formule $\neg c$ peut apparaître dans un support.
- Construisons $A' = (\{\neg c, (a \wedge b) \rightarrow c\}, \neg(a \wedge b))$.
- Ici, on a $A \overset{2}{\not\rightarrow} A'$ mais pas l'inverse car la formule $a \wedge b$ *n'appartient pas* à S_A ! La relation d'$\texttt{attaque}_2$ présente donc une sensibilité forte à la syntaxe.
- Toutefois, on peut trouver $A'' = (\{\neg c, b, (a \wedge b) \rightarrow c\}, \neg a)$. pour lequel on a bien $A \overset{2}{\not\rightarrow} A''$ et $A'' \overset{2}{\not\rightarrow} A$.

Nous n'approfondirons pas cet aspect dans ce document, mais il faut le garder à l'esprit pour la suite sous la forme d'une contrainte très forte :

> Dans le graphe des contrariétés, n'apparaissent que les arguments et les relations $\texttt{attaque}_2$ et $\texttt{attaque}_3$ entre arguments avec lesquels on souhaite travailler.
> On supposera donc que le graphe proposé est "significatif pour le problème représenté".
> Ce qui veut dire que tous les arguments sont *significatifs* (pas de redondance) et que tous les arcs présents dans le graphe ont un sens alors que ceux qui n'y sont pas, n'y sont pas de manière *volontaire* (pas question de déduire de nouveaux arcs en cours de raisonnement).

Remarquons aussi qu'il existe des arcs "obligatoires" du fait des choix effectués lors de la définition des relations d'attaque. Par exemple, le graphe $A \overset{3}{\not\rightarrow} B$ est impossible, on ne peut avoir que $A \underset{\overset{3}{\not\rightarrow}}{\overset{3}{\not\leftarrow}} B$.

De même, le graphe $A \begin{array}{c} \overset{3}{\not\rightarrow} \\ \overset{2}{\not\rightarrow} \end{array} B$ est impossible, on ne peut avoir que[3] $A \begin{array}{c} \overset{3}{\not\leftarrow} \\ \overset{2}{\not\rightarrow} \\ \overset{3}{\not\rightarrow} \end{array} B$.

Propriété 3 *Si* $A \overset{i}{\not\rightarrow} B$*, pour* $i = 2, 3$ *alors* $S_A \cup S_B \vdash \perp$.

Un exemple Soit les 4 arguments suivants :
- $A = (\{demission, demission \rightarrow \neg ministre\}, \neg ministre)$
- $B = (\{ministre, ministre \rightarrow \neg personneprivee\}, \neg personneprivee)$
- $C = (\{personneprivee, \neg accord, (personneprivee \wedge publication) \rightarrow accord\}, \neg publication)$
- $D = (\{infoimportante, infoimportante \rightarrow publication\}, publication)$

On a le graphe des contrariétés suivant :

$$A \overset{2}{\not\rightarrow} B \overset{2}{\not\rightarrow} C \begin{array}{c} \overset{3}{\not\leftarrow} \\ \overset{3}{\not\rightarrow} \end{array} D$$

Dernière remarque : Si on interdit les circuits dans le graphe alors la relation `attaque`$_3$ est elle-aussi interdite.

5.1.3 Les différents types d'appui

Il s'agit d'introduire ici une nouvelle relation entre arguments bâtie sur l'idée inverse de la contrariété : un argument peut en aider un autre. Ici aussi, comme pour la relation de contrariété, il faut passer en revue les différents cas pour voir plus précisément à quoi peut correspondre cette nouvelle relation.
Avant toute chose, posons une hypothèse de départ : pour qu'un argument puisse en appuyer un autre, il faut que les supports soient en accord (donc que l'union des supports soit consistante[4]). Cette hypothèse sera reprise dans chaque définition de la relation d'appui.
Si on s'inspire des différents cas vus pour l'attaque, on a ici aussi 4 possibilités :

Définition 14 (Appui d'une partie du support de B à l'aide du support de A) *Soit A et B deux arguments. A* `appuie`$_1$ *B (noté $A \xrightarrow{1} B$) si et seulement si $S_A \cup S_B \not\vdash \perp$ et il existe X une formule telle que $S_A \vdash X$ et $X \in S_B$.*

Définition 15 (Appui d'une partie du support de B à l'aide de la conclusion de A) *Soit A et B deux arguments. A* `appuie`$_2$ *B (noté $A \xrightarrow{2} B$) si et seulement si $S_A \cup S_B \not\vdash \perp$ et $C_A \in S_B$.*

Définition 16 (Appui de la conclusion de B à l'aide de la conclusion de A) *Soit A et B deux arguments. A* `appuie`$_3$ *B (noté $A \xrightarrow{3} B$) si et seulement si $S_A \cup S_B \not\vdash \perp$ et $C_A \equiv C_B$.*

Définition 17 (Appui de la conclusion de B à l'aide du support de A) *Soit A et B deux arguments. A* `appuie`$_4$ *B (noté $A \xrightarrow{4} B$) si et seulement si $S_A \cup S_B \not\vdash \perp$ et il existe X une formule telle que $S_A \vdash X$ et $X \equiv C_B$.*

Propriété 4 *Soit A et B deux arguments, on a :*
1. *La relation* `appuie`$_1$ *n'est pas symétrique.*
2. *La relation* `appuie`$_2$ *n'est pas symétrique.*
3. *La relation* `appuie`$_3$ *est symétrique.*
4. *La relation* `appuie`$_4$ *n'est pas symétrique.*
5. *Si $A \xrightarrow{4} B$ alors $\exists C$ un argument tel que $C \xrightarrow{3} B$.*
6. *Si $A \xrightarrow{1} B$ alors $\exists C$ un argument tel que $C \xrightarrow{2} B$.*
7. $A \xrightarrow{2} B \Rightarrow A \xrightarrow{1} B$.
8. $A \xrightarrow{3} B \Rightarrow A \xrightarrow{4} B$.

[3]Cela se produira si le support de B est aussi sa conclusion.
[4]En effet, il semble irréaliste qu'un argument A appuie un argument B si les raisons d'utiliser A sont en contradiction avec celles d'utiliser B.

Remarques :
- La relation $\texttt{appuie}_3$ étant symétrique, elle induit un circuit de longueur 2 entre *chaque couple d'arguments* liés par cette relation.
- La relation d'$\texttt{appuie}_1$ (resp. d'$\texttt{appuie}_4$) induit une partie de la relation d'$\texttt{appuie}_2$ (resp. d'$\texttt{appuie}_3$).
- La relation $\texttt{appuie}_3$ ne paraît pas très saine. En effet, quand deux arguments arrivent à la même conclusion, peut-on dire qu'ils s'appuient mutuellement (ils pourraient arriver à la même conclusion pour des raisons totalement différentes !).

Ces remarques vont orienter nos choix parmi toutes les relations de contrariété possibles. En effet, elles indiquent que les relations d'$\texttt{appuie}_1$ et d'$\texttt{appuie}_4$ sont inutiles. D'autre part, on choisit de considérer que la relation d'$\texttt{appuie}_3$ n'est pas utilisable dans le cadre de cette étude, car elle ne représente pas un appui *pour un argument*, mais plutôt un appui *pour une conclusion*.
Et enfin, la contrainte de non redondance donnée en fin de section 5.1.2 page 16 reste valable dans le cadre d'un graphe d'appui. On obtient donc :

> Dans le graphe des appuis, n'apparaissent que les arguments et la relation $\texttt{appuie}_2$ entre arguments avec lesquels on souhaite travailler.
> On supposera donc que le graphe proposé est "significatif pour le problème représenté".
> Ce qui veut dire que tous les arguments sont *significatifs* (pas de redondance) et que tous les arcs présents dans le graphe ont un sens alors que ceux qui n'y sont pas, n'y sont pas de manière *volontaire* (pas question de déduire de nouveaux arcs en cours de raisonnement).

Il reste quand même quelques questions :

Question L'argument $A = (\{a, a \to b\}, b)$ appuie l'argument $B = (\{b, b \to c\}, c)$. Mais peut-on dire que l'argument $A' = (\{a\}, a)$ appuie l'argument $A = (\{a, a \to b\}, b)$?

Réponse Pourquoi pas ! Ceci étant, l'argument A' est très particulier : il s'auto-appuie ! Il crée donc un circuit impair de longueur 1 dans le graphe de la relation $\mathcal{R}_{app}$.

Un exemple Soit les 3 arguments suivants :
- $A = (\{lettredemission, demissionacceptee, (lettredemission \land demissionacceptee) \to demission\}, demission)$
- $B = (\{demission, demission \to \neg ministre\}, \neg ministre)$
- $C = (\{\neg ministre, \neg ministre \to \neg accesMatignon\}, \neg accesMatignon)$

On a le graphe des appuis suivant :

$$A \xrightarrow{2} B \xrightarrow{2} C$$

5.2 Un système d'argumentation bipolaire

Maintenant que nous avons su décrire une instance possible d'argumentation bipolaire, nous allons poser le cadre formel.
En utilisant le cadre de [Dun95] et en rajoutant une seconde relation d'interaction entre arguments, on obtient :

Définition 18 (SABP : système d'argumentation bipolaire) *Un système d'argumentation bipolaire (noté SABP) est un triplet* $<\mathcal{A}, \mathcal{R}_{att}, \mathcal{R}_{app}>$ *avec :*
- $\mathcal{A}$ *: un ensemble d'arguments*[5]*,*
- $\mathcal{R}_{att}$ *: une relation de contrariété*[6]*,*
- $\mathcal{R}_{app}$ *: une relation d'appui*[7]*.*

[5] Certaines instances de cet ensemble peuvent respecter la définition 9 page 15 et, dans ce cas, il s'agira d'une restriction du cadre général et abstrait posé par [Dun95].

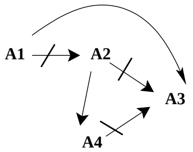
[6] Certaines instances de cette relation peuvent respecter les définitions 11 page 16 ou 12 page 16 et, dans ce cas, il s'agira d'une restriction du cadre général et abstrait posé par [Dun95].

[7] Certaines instances de cette relation peuvent respecter la définition 15 page précédente et, dans ce cas, il s'agira d'une restriction du cadre général et abstrait posé par [Dun95].

Notations : Soit $A \in \mathcal{A}$, $B \in \mathcal{A}$.
$A\mathcal{R}_{att}B$ sera représenté graphiquement par : $A \not\rightarrow B$.
$A\mathcal{R}_{app}B$ sera représenté graphiquement par : $A \rightarrow B$.
L'ensemble $\{A_i \in \mathcal{A} | A_i\mathcal{R}_{att}A\}$ est noté ${\mathcal{R}_{att}}^{-}(A)$ et l'ensemble $\{A_i \in \mathcal{A} | A\mathcal{R}_{att}A_i\}$ est noté ${\mathcal{R}_{att}}^{+}(A)$.
L'ensemble $\{A_i \in \mathcal{A} | A_i\mathcal{R}_{app}A\}$ est noté ${\mathcal{R}_{app}}^{-}(A)$ et l'ensemble $\{A_i \in \mathcal{A} | A\mathcal{R}_{app}A_i\}$ est noté ${\mathcal{R}_{app}}^{+}(A)$.
$<\mathcal{A}, \mathcal{R}_{att}, \mathcal{R}_{app}>$ définit un graphe orienté $\mathcal{G}$ (dit *graphe des interactions bipolaire*).

Exemple :

Le système $<\mathcal{A} = \{A_1, A_2, A_3, A_4\}, \mathcal{R}_{att} = \{(A_2, A_3), (A_4, A_3), (A_1, A_2)\}, \mathcal{R}_{app} = \{(A_2, A_4), (A_3, A_1)\}>$ définit le graphe $\mathcal{G}$ suivant :



Définition 19 (Représentation graphique du système d'argumentation bipolaire)
Soit $\mathcal{G}$ le graphe des interactions bipolaire associé à un système d'argumentation bipolaire $<\mathcal{A}, \mathcal{R}_{att}, \mathcal{R}_{app}>$, on définit :

Feuille du graphe des interactions bipolaire

Un argument $A \in \mathcal{A}$ tel que ${\mathcal{R}_{att}}^{-}(A) = \varnothing$ et ${\mathcal{R}_{app}}^{-}(A) = \varnothing$ sera une feuille de $\mathcal{G}$.

Séquences dans le graphe des interactions bipolaire

- *Une* séquence *est une suite d'arguments $A_1 - \ldots - A_n$, $n \geq 2$, telle que :*
  - $A_1\mathcal{R}_1A_2$,
  - $\ldots$,
  - $A_{n-1}\mathcal{R}_{n-1}A_n$,
  - *et* $\forall i = 1 \ldots n-1$, $\mathcal{R}_i = \mathcal{R}_{att}$ *ou* $\mathcal{R}_i = \mathcal{R}_{app}$.

  Remarquons qu'une séquence ne peut être vide : elle contiendra au moins deux arguments, donc un arc. L'argument A_n sera appelé racine *de la séquence.*
- *Une* séquence d'attaques *est une séquence $A_1 - \ldots - A_n$, $n \geq 2$, telle que $\forall i = 1 \ldots n-1$, $\mathcal{R}_i = \mathcal{R}_{att}$ et $n-1$ est impair. C'est donc une séquence qui ne contient que des arcs d'attaque et en nombre impair.*
- *Une* séquence de défense *est une séquence $A_1 - \ldots - A_n$, $n \geq 2$, telle que $\forall i = 1 \ldots n-1$, $\mathcal{R}_i = \mathcal{R}_{att}$ et $n-1$ est pair. C'est donc une séquence qui ne contient que des arcs d'attaque et en nombre pair.*
- *Une* séquence d'appuis *est une séquence $A_1 - \ldots - A_n$, $n \geq 2$, telle que $\forall i = 1 \ldots n-1$, $\mathcal{R}_i = \mathcal{R}_{app}$. C'est donc une séquence qui ne contient que des arcs d'appui.*
- *Une* attaque appuyée *est une séquence $A_1 - \ldots - A_n$, $n \geq 3$, telle que $\forall i = 1 \ldots n-2$, $\mathcal{R}_i = \mathcal{R}_{app}$ et $\mathcal{R}_{n-1} = \mathcal{R}_{att}$. C'est donc une séquence qui ne contient qu'un seul arc d'attaque, celui menant à la racine, et des arcs d'appui.*

  Par extension, une séquence d'attaques réduite à un seul arc d'attaque pourra aussi être appelée attaque appuyée.

Chemins dans le graphe des interactions bipolaire

- *Un* chemin de A vers B, *noté $\mathcal{C}$, est une séquence $A_1 - \ldots - A_n$ avec $A = A_1$ et $A_n = B$.*
- *La* longueur de ce chemin *est alors $n-1$ (le nombre d'arcs constituant ce chemin) et sera notée $l_{\mathcal{C}}$.*
- *Le* nombre d'attaques de ce chemin *est alors le nombre de $\mathcal{R}_i = \mathcal{R}_{att}$, pour $i = 1 \ldots n$, et sera noté $natt_{\mathcal{C}}$.*
- *Le* nombre d'appuis de ce chemin *est alors le nombre de $\mathcal{R}_i = \mathcal{R}_{app}$, pour $i = 1 \ldots n$, et sera noté $napp_{\mathcal{C}}$.*
- *L'ensemble des chemins de A vers B sera noté $\mathcal{C}(A, B)$.*

Sous-chemin d'un chemin *Soit 2 chemins $\mathcal{C}_A = A_1 - \ldots - A_n$ et $\mathcal{C}_B = B_1 - \ldots - B_m$. $\mathcal{C}_B$ est un* sous-chemin du chemin $\mathcal{C}_A$ *ssi* $\mathcal{C}_B \subseteq \mathcal{C}_A$[8].

[8] Une suite $X_1 - \ldots - X_n$ correspondant à $X_1\mathcal{R}_1^X X_2 \ldots X_{n-1}\mathcal{R}_{n-1}^X X_n$ est incluse dans une autre suite $Y_1 - \ldots - Y_m$ correspondant à $Y_1\mathcal{R}_1^Y Y_2 \ldots Y_{m-1}\mathcal{R}_{m-1}^Y Y_m$ ssi il existe Y_i et Y_j tels que $X_1 = Y_i$, $X_n = Y_j$ et $\forall \mathcal{R}_k^X$, $k = 1 \ldots n-1$, $\mathcal{R}_k^X = \mathcal{R}_{i+k-1}^Y$. On notera $X_1 - \ldots - X_n \subseteq Y_1 - \ldots - Y_m$.

Dépendance, indépendance, racine-dépendance d'un chemin *Soit 2 chemins* $\mathcal{C}_A \in \mathcal{C}(A_1, A_n)$ *et* $\mathcal{C}_B \in \mathcal{C}(B_1, B_m)$.

- *Ces deux chemins seront dits* dépendants *ssi* $\exists A_i \in \mathcal{C}_A$, $\exists B_j \in \mathcal{C}_B$ *tel que* $A_i = B_j$. Indépendants *sinon.*
- *Ces deux chemins seront dits* racine-dépendants en A_n *ssi* $A_n = B_m$ *et* $\forall A_i \neq A_n \in \mathcal{C}_A$, $\nexists B_j \in \mathcal{C}_B$ *tel que* $A_i = B_j$.

Circuits dans le graphe des interactions bipolaire

- *Un* circuit[9] *est un chemin* $\mathcal{C} = A_1 - \ldots - A_n - A_1$ *tel que* $\forall i, j \in [1, n], i \neq j$, $\nexists A_i, A_j \in \mathcal{C}$ *tels que* $A_i = A_j$.
- *Un circuit est* isolé *quand aucun des arguments le composant n'a d'attaquant, ni d'appui en dehors du circuit.*
- *Deux circuits* $\mathcal{C}_A = A_1 - \ldots - A_n - A_1$ *et* $\mathcal{C}_B = B_1 - \ldots - B_m - B_1$ *sont* interconnectés *ssi* $\exists i \in [1, n], \exists j \in [1, m]$ *tels que* $A_i = B_j$.

Remarquons que les deux dernières définitions de la définition 19 page ci-contre sont identiques à celles données dans le cas d'un graphe des contrariétés (voir la définition 2 page 3).

Nous allons utiliser les notions d'attaque et de défense directe ou indirecte inspirées par [Dun95] mais modifiées par nos soins pour prendre en compte la relation d'appui. On a donc :

Définition 20 (Attaquants/Défenseurs directs/indirects d'un argument) *Soit* $A \in \mathcal{A}$ *:*

- *Les* attaquants directs *de* A *sont les éléments de* ${\mathcal{R}_{att}}^{-}(A)$.
- *Les* défenseurs directs *de* A *sont les attaquants directs des éléments de* ${\mathcal{R}_{att}}^{-}(A)$.
- *Les* attaquants indirects *de* A *sont les éléments* A_i *définis par :*
  $\exists \mathcal{C} \in \mathcal{C}(A_i, A)$ *tel que* $natt_{\mathcal{C}} = 2k + 1$, *avec* $k \geq 0$ *et* A_i *n'est pas attaquant direct.*
- *Les* défenseurs indirects *de* A *sont les éléments* A_i *définis par :*
  $\exists \mathcal{C} \in \mathcal{C}(A_i, A)$ *tel que* $natt_{\mathcal{C}} = 2k$, *avec* $k \geq 1$ *et* A_i *n'est pas défenseur direct.*

L'introduction de la relation d'appui implique donc une modification de la notion d'attaquants indirects et de défenseurs indirects. En effet, on conserve l'idée de compter les nombres d'arcs d'attaque du chemin en testant la parité mais on ajoute le fait qu'il puisse y avoir des arcs d'appui n'importe où sur ce même chemin.

Définition 21 (Appuis directs/indirects d'un argument) *Soit* $A \in \mathcal{A}$ *:*

- *Les* appuis directs *de* A *sont les éléments de* ${\mathcal{R}_{app}}^{-}(A)$.
- *Les* appuis indirects *de* A *sont les éléments* A_i *définis par :*

$$\exists \mathcal{C} \in \mathcal{C}(A_i, A) \text{ tel que } napp_{\mathcal{C}} = l_{\mathcal{C}} \geq 2.$$

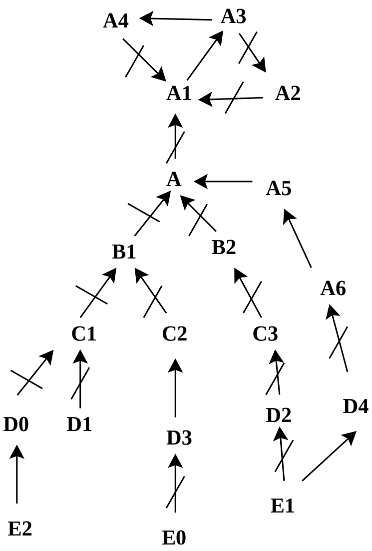
C'est-à-dire qu'on n'autorise aucun arc d'attaque entre deux arguments pour que l'un soit appui indirect ou direct de l'autre.

On dira plus généralement que :

- Si l'argument A est un attaquant (direct ou indirect) de l'argument B, alors A *attaque* B (ou B *est attaqué par* A).
- Si l'argument A est un défenseur (direct ou indirect) de l'argument B, alors A *défend* B (ou B *est défendu par* A).
- Si l'argument A est un appui (direct ou indirect) de l'argument B, alors A *appuie* B (ou B *est appuyé par* A).

Toutes ces notions sont illustrées sur l'exemple suivant :

[9]Cette définition d'un circuit correspond à la définition d'un circuit élémentaire en théorie des graphes (ne contient pas 2 arcs ayant la même origine, ou le même but).



Sur ce graphe $\mathcal{G}$, on a donc (entre autres) :
- un chemin de C_2 vers A de longueur 2 ($C_2 - B_1 - A$),
- 2 circuits $A_1 - A_3 - A_2 - A_1$ et $A_1 - A_3 - A_4 - A_1$, chacun de longueur 3 qui ne sont pas isolés (remarquons que $A_1 - A_3 - A_2 - A_1 - A_3 - A_4 - A_1$ n'est pas un circuit d'après notre définition),
- les deux circuits cités précédemment sont interconnectés (en A_1 et A_3),
- les chemins $D_1 - C_1 - B_1$ et $C_3 - B_2 - A$ sont indépendants, alors que $D_1 - C_1 - B_1 - A$ et $C_3 - B_2 - A$ sont racine-dépendants et que $D_1 - C_1 - B_1 - A$ et $C_2 - B_1 - A$ sont dépendants,
- D_1, E_2, E_1 et E_0 sont les feuilles de $\mathcal{G}$,
- B_1 et B_2 sont les deux attaquants directs de A,
- C_1, C_2 et C_3 sont les trois défenseurs directs de A,
- E_0, D_0, D_1, D_2, D_4, E_1 et E_2 sont les attaquants indirects de A,
- E_1 et D_3 sont les défenseurs indirects de A,
- A_5 est le seul appui direct de A,
- A_6 est le seul appui indirect de A.

Remarque : nous ne donnerons pas ici de définition pour la notion de branche (d'appui, d'attaque ou de défense), contrairement à ce qui avait été fait en section 2 page 3. En effet, cette notion est intimement liée aux choix effectués lors de la définition des évaluations globales (voir section 8 page 49).

Quelques cas particuliers
Si on veut respecter les définitions choisies pour les arguments et les relations d'interaction données en section 5.1 page 15, il existe quelques graphes d'interaction impossibles à obtenir :
- Le graphe $A \underset{\rightarrow}{\overset{\not\rightarrow}{}} B$ n'a pas de sens car il contredit l'hypothèse de la consistance de l'union des supports en cas d'appui.
- De même, le graphe $A \underset{\leftarrow}{\overset{\not\rightarrow}{}} B$ n'a pas de sens (il contredit lui-aussi l'hypothèse de la consistance de l'union des supports en cas d'appui).

Un exemple : Soit les 6 arguments suivants :
- $A = (\{demission, demission \rightarrow \neg ministre\}, \neg ministre)$
- $B = (\{ministre, ministre \rightarrow \neg personneprivee\}, \neg personneprivee)$
- $C = (\{personneprivee, \neg accord, (personneprivee \wedge publication) \rightarrow accord\}, \neg publication)$
- $D = (\{infoimportante, infoimportante \rightarrow publication\}, publication)$
- $E = (\{lettredemission, demissionacceptee, (lettredemission \wedge demissionacceptee) \rightarrow demission\}, demission)$
- $F = (\{infoconcernesantepublique, infoconcernesantepublique \rightarrow infoimportante\}, infoimportante)$

En n'utilisant pour relations de contrariété et d'appui que les relations `attaque`$_2$ et `appuie`$_2$, on a le graphe des interactions bipolaire suivant :

$$E \rightarrow A \not\rightarrow B \not\rightarrow C \underset{\not\rightarrow}{\overset{\not\leftarrow}{}} D \leftarrow F$$

5.3 Retour sur les exemples

Nous allons ici reprendre les exemples donnés au chapitre 3 page 7 et les implémenter sous la forme d'un SABP.

5.3.1 L'exemple de l'orientation

Rappelons qu'il s'agit d'une discussion entre 2 conseillers d'orientation (O_1 et O_2) dont l'enjeu est l'orientation ou pas en IUT d'un individu X. Nous allons donner sous la forme d'un tableau le nom de l'intervenant, ce qu'il dit et l'argument utilisé :

Nom	Élément du dialogue	Argument
O_1 :	Il faut orienter X vers un IUT.	A_0
O_2 :	Attention, un IUT est une formation très courte.	A_1
O_1 :	Pas forcément ! Avec les nouvelles licences professionnelles, cela devient une formation en trois ans.	A_2
O_2 :	Que ce soit en deux ou en trois ans, cela reste une formation qui manque d'aspects théoriques solides. (Cela risque de gêner X plus tard, dans sa vie professionnelle.)	A_3
O_1 :	Ce n'est pas bien grave, car X pourra éventuellement compléter ses connaissances par un retour dans un cycle universitaire long plus pointu au niveau théorique.	A_4
O_2 :	Elle manque aussi de connaissances générales. (Et là, l'impact futur concernera aussi bien la vie professionnelle que la vie de tous les jours.)	A_5
O_1 :	Là-aussi, ce n'est pas très gênant. Les meilleurs de l'IUT peuvent ensuite rejoindre une école d'ingénieurs généraliste.	A_6
O_2 :	Oui, mais sera-t-il un des meilleurs ? Son dossier actuel n'est pas en sa faveur.	A_7
O_1 :	Bon, bon. Mais, il s'agit quand même d'une formation professionnelle. X aura donc plus de facilités pour trouver ensuite du travail.	A_8
O_2 :	C'est vrai ! Il y a beaucoup d'heures de stages.	A_9
O_1 :	Et aussi des contacts importants entre les entreprises et l'équipe pédagogique.	A_{10}
O_2 :	Du coup, les débouchés professionnels en sont facilités.	A_{11}
O_1 :	Tu vois bien que ce serait un bon choix !	
O_2 :	Il y a quand même un problème important dont il nous faut tenir compte : X n'est pas vraiment fixé sur la thématique qu'il veut suivre. Cela va donc être difficile de lui imposer une formation spécialisée dès le départ.	A_{12}
O_1 :	Flûte ! J'avais oublié ce détail !	
O_2 :	Et pourtant, c'est vrai que cette formation conviendrait bien à son dossier scolaire qui n'est pas brillant.	A_{13}
O_1 :	Oui, c'est un individu qui a besoin d'être encadré.	A_{14}
O_2 :	Dans une formation générale de masse comme les premières années d'université, il va à l'échec.	A_{15}

On peut traduire cet exemple sous différentes formes. L'une d'entre elles est la suivante en logique propositionnelle :

Les connaissances générales :

k_1 **:** IUT → formation_courte

k_2 **:** licence_pro → ¬(IUT → formation_courte)

k_3 **:** IUT → manque_théorie

k_4 **:** possibilité_retour_université → ¬(IUT → manque_théorie)

k_5 **:** IUT → manque_connaissances_générales

k_6 **:** possibilité_intégrer_ENSI → ¬(IUT → manque_connaissances_générales)

k_7 **:** dossier_médiocre → ¬possibilité_intégrer_ENSI

k_8 **:** ¬IUT → ¬formation_professionnelle

k_9 **:** stage → formation_professionnelle

k_{10} **:** contact_avec_entreprises → formation_professionnelle

k_{11} **:** ¬formation_professionnelle → ¬débouchés

k_{12} **:** IUT → formation_spécialisée

k_{13} **:** formation_spécialisée → choix_domaine

k_{14} **:** dossier_médiocre $\rightarrow$ IUT
k_{15} **:** $\neg$IUT $\rightarrow$ $\neg$étudiant_encadré
k_{16} **:** Possibilité_faire_DEUG $\rightarrow$ $\neg$étudiant_encadré

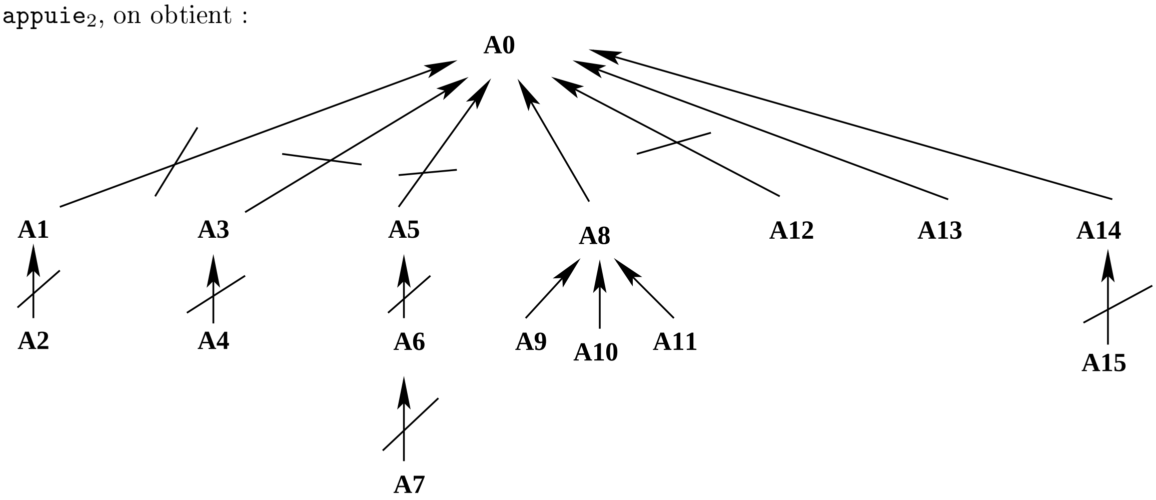
Les connaissances factuelles et les contraintes :

f_1 **:** $\neg$formation_courte (contrainte)
f_2 **:** licence_pro
f_3 **:** $\neg$manque_théorie (contrainte)
f_4 **:** possibilité_retour_université
f_5 **:** $\neg$manque_connaissances_générales (contrainte)
f_6 **:** possibilité_intégrer_ENSI
f_7 **:** dossier_médiocre
f_8 **:** formation_professionnelle (contrainte)
f_9 **:** stage
f_{10} **:** contact_avec_entreprises
f_{11} **:** débouchés (contrainte)
f_{12} **:** $\neg$choix_domaine
f_{13} **:** étudiant_encadré (contrainte)
f_{14} **:** Possibilité_faire_DEUG

Les arguments :

A_0 **:** $(\{\text{IUT}\}, \text{IUT})$
A_1 **:** $(\{k_1, f_1\}, \neg\text{IUT})$
A_2 **:** $(\{k_2, f_2\}, \neg k_1)$
A_3 **:** $(\{k_3, f_3\}, \neg\text{IUT})$
A_4 **:** $(\{k_4, f_4\}, \neg k_3)$
A_5 **:** $(\{k_5, f_5\}, \neg\text{IUT})$
A_6 **:** $(\{k_6, f_6\}, \neg k_5)$
A_7 **:** $(\{k_7, f_7\}, \neg f_6)$
A_8 **:** $(\{k_8, f_8\}, \text{IUT})$
A_9 **:** $(\{k_9, f_9\}, f_8)$
A_{10} **:** $(\{k_{10}, f_{10}\}, f_8)$
A_{11} **:** $(\{k_{11}, f_{11}\}, f_8)$
A_{12} **:** $(\{k_{12}, k_{13}, f_{12}\}, \neg\text{IUT})$
A_{13} **:** $(\{k_{14}, f_7\}, \text{IUT})$
A_{14} **:** $(\{k_{15}, f_{13}\}, \text{IUT})$
A_{15} **:** $(\{k_{16}, f_{14}\}, \neg f_{13})$

Et sous forme graphique, en n'utilisant pour relations de contrariété et d'appui que les relations `attaque`$_2$ et `appuie`$_2$, on obtient :



Remarquons que l'argument "racine" A_0 joue un rôle très particulier puisqu'il représente le choix à effectuer. Étant donné qu'il n'y a que 2 choix possibles (IUT ou ¬IUT), l'argument A_0 correspond au choix IUT et il ne semble pas utile d'introduire un second argument "racine" $A_{0'}$ pour ¬IUT. En effet, le graphe ayant $A_{0'}$ pour racine serait l'image de celui donné pour A_0 avec les attaques menant à A_0 transformées en appui pour $A_{0'}$ et les appuis menant à A_0 transformés en attaque pour $A_{0'}$ (les autres arcs restant tous inchangés).

5.3.2 L'exemple de la prothèse du genou

Ici, le contexte est celui d'une discussion entre deux médecins (M_1 et M_2) au sujet d'un patient X souffrant d'un grave problème au genou.

Nom	Élément du dialogue	Argument
M_1 :	Il faut poser une prothèse du genou à X.	A_0
M_2 :	Attention, pour cela il faut une anesthésie et il y a actuellement un gros risque de complications dues à cette anesthésie.	A_1
M_1 :	Oui, si on fait une anesthésie générale. Mais on pourrait se contenter d'une anesthésie locale.	A_2
M_2 :	Il reste quand même les risques d'infections post-opératoires.	A_3
M_1 :	C'est vrai. Il y a actuellement une recrudescence des risques d'infections nosocomiales dans l'hôpital.	A_4
M_2 :	Et en plus, le patient est bien trop jeune pour avoir déjà une prothèse.	A_5
M_1 :	Oui X est très jeune, mais c'est aussi un sportif de haut niveau pour lequel son genou est un outil indispensable.	A_6
M_2 :	Et alors ?	
M_1 :	La prothèse lui permettrait de récupérer très vite sa motricité.	A_7
M_2 :	Mais pourquoi ne pas utiliser un autre traitement ?	
M_1 :	Tous les traitements classiques sont en échec. On a tout essayé et rien ne marche.	A_8
M_2 :	Non, on n'a pas encore tout essayé. Il existe un nouveau traitement expérimental.	A_9
M_1 :	Les expériences, les expériences, on ne sait pas où cela risque de nous conduire !	A_{10}

On peut traduire cet exemple sous différentes formes. L'une d'entre elles est la suivante en logique propositionnelle :

Les connaissances générales :

k_1 **:** prothèse → opération

k_2 **:** opération → risque_anesthésie

k_3 **:** opération → risque_infection

k_4 **:** anesthésie_locale_possible → ¬(opération → risque_anesthésie)

k_5 **:** infections_nosocomiales_dans_hôpital → (opération → risque_infection)

k_6 **:** réserve_sur_âge → ¬prothèse

k_7 **:** patient_jeune → réserve_sur_âge

k_8 **:** sportif_haut_niveau → ¬(patient_jeune → réserve_sur_âge)

k_9 **:** ¬prothèse → récupération_longue

k_{10} **:** sportif_haut_niveau → ¬récupération_longue

k_{11} **:** échec_tous_autres_traitements → prothèse

k_{12} **:** nouveau_traitement → ¬échec_tous_autres_traitements

k_{13} **:** nouveau_traitement → risque_nouveau_traitement

Les connaissances factuelles et les contraintes :

f_1 **:** patient_jeune

f_2 **:** sportif_haut_niveau

f_3 **:** $\neg$risque_anesthésie (contrainte)

$f_{3'}$ **:** $\neg$risque_infection (contrainte)

$f_{3''}$ **:** $\neg$risque_nouveau_traitement (contrainte)

f_4 **:** échec_tous_autres_traitements

f_5 **:** infections_nosocomiales_dans_hôpital

f_6 **:** anesthésie_locale_possible

f_7 **:** nouveau_traitement

Les arguments :

A_0 **:** $(\{\text{prothèse}\}, \text{prothèse})$

A_1 **:** $(\{f_3, k_2, k_1\}, \neg\text{prothèse})$

A_2 **:** $(\{f_6, k_4\}, \neg k_2)$

A_3 **:** $(\{f_{3'}, k_3, k_1\}, \neg\text{prothèse})$

A_4 **:** $(\{f_5, k_5\}, k_3)$

A_5 **:** $(\{f_1, k_6, k_7\}, \neg\text{prothèse})$

A_6 **:** $(\{f_2, k_8\}, \neg k_7)$

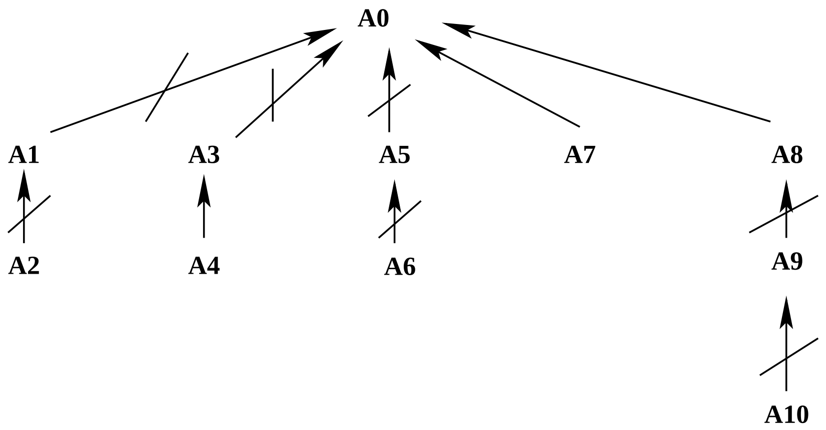
A_7 **:** $(\{f_2, k_9, k_{10}\}, \text{prothèse})$

A_8 **:** $(\{f_4, k_{11}\}, \text{prothèse})$

A_9 **:** $(\{f_7, k_{12}\}, \neg f_4)$

A_{10} **:** $(\{f_{3''}, k_{13}\}, \neg f_7)$

Et sous forme graphique, en n'utilisant pour relations de contrariété et d'appui que les relations `attaque`$_2$ et `appuie`$_2$, on obtient :



Ici aussi, comme pour l'exemple de l'orientation, il n'y a que 2 choix possibles (prothèse ou $\neg$prothèse) et l'argument A_0 représente le choix prothèse.

5.3.3 L'exemple de l'enquête policière

Sherlock Holmes (SH) et le Dr Watson (DW) enquêtent sur le meurtre de Marmaduke Brandford, un riche vendeur d'objets d'arts. Son corps a été trouvé dans la bibliothèque. Il a été frappé dans le dos avec un couteau de cuisine. Il y a au moins trois suspects possibles : Mme Benton, la cuisinière, Partridge, le maître d'hôtel, et Fenimore Brandford, le neveu de la victime.

Nom	Élément du dialogue	Argument
DW :	Un couteau de cuisine ... Je soupçonne la cuisinière, Mme Benton. Qu'en pensez-vous Holmes ?	A_1
SH :	Ne trouvez-vous pas cela un peu trop simpliste ? Bien que Mme Benton ait eu de nombreux motifs de tuer son patron (c'était un homme extrêmement violent), elle était absente hier soir. Par contre, Partridge, le maître d'hôtel, était bien là !	A_2
DW :	Vous devez avoir raison, Holmes. Mais comment expliquez-vous alors le gras qui est sur le manche du couteau ? Une cuisinière a toujours les mains grasses, n'est-ce pas ?	A_3
SH :	Au contraire, mon cher Watson ! Si Mme Benton avait tué Brandford, elle n'aurait sûrement pas laissé ces marques de gras sur le couteau.	A_4
DW :	Oui, je suppose ... Mais je ne peux imaginer Partridge assassinant son maître après l'avoir servi pendant trente ans.	A_5
SH :	Moi non plus ! Par contre, n'oublions pas que Fenimore, le neveu de Brandford, héritera de la fortune de son oncle, alors que Mme Benton et Partridge n'auront rien.	A_6
DW :	Fenimore Brandford ... pourquoi pas après tout ? Il fait un suspect parfait. Par contre, cela n'explique pas le gras sur le couteau.	A_7
SH :	Si, Watson, si ! Je suppose que vous savez que Fenimore Brandford a perdu son bras droit dans le Transvaal durant la guerre avec les Zoulous ...	A_8 (début)
DW :	Oui et je me souviens même qu'il avait été décoré de la "Victoria Cross" pour bravoure. Toutefois, je ne vois pas le rapport ...	
SH :	Fenimore Brandford a une prothèse, Watson, un bras mécanique qu'il doit graisser méticuleusement chaque jour afin qu'il ne grince pas et ...	A_8 (fin)
DW :	Le gras sur le couteau ! Bien-sûr ! Vous êtes un génie, Holmes !	
SH :	Élémentaire, mon cher Watson ...	

On peut traduire cet exemple sous différentes formes. L'une d'entre elles est la suivante en logique propositionnelle :

Les connaissances générales :

k_1 **:** arme_est_couteau_cuisine → cuisinière_coupable

k_2 **:** cuisinière_absente ↔ ¬ cuisinière_coupable

k_3 **:** maître_hôtel_absent ↔ ¬ maître_hôtel_coupable

k_4 **:** (gras_sur_couteau ∧ mains_cuisinière_grasses) → cuisinière_coupable

k_5 **:** cuisinière_coupable → se_serait_essuyé_mains

k_6 **:** se_serait_essuyé_mains → ¬ gras_sur_couteau

k_7 **:** maître_hôtel_sert_depuis_30_ans → ¬ maître_hôtel_coupable

k_8 **:** ¬ cuisinière_hérite → ¬ cuisinière_coupable

$k_{8'}$ **:** ¬ maître_hôtel_hérite → ¬ maître_hôtel_coupable

$k_{8''}$ **:** neveu_hérite → neveu_coupable

k_9 **:** neveu_coupable → ¬ gras_sur_couteau

k_{10} **:** neveu_a_perdu_bras → neveu_a_bras_mécanique

k_{11} **:** neveu_a_bras_mécanique → neveu_utilise_graisse

k_{12} **:** neveu_utilise_graisse → (gras_sur_couteau → neveu_coupable)

Les connaissances factuelles et les contraintes :

f_1 **:** arme_est_couteau_cuisine

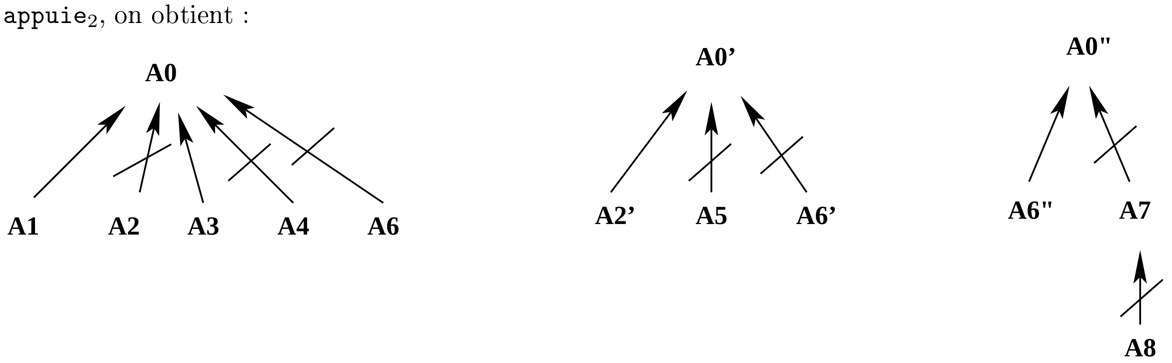
f_2 **:** cuisinière_absente

f_3 **:** ¬ maître_hôtel_absent
f_4 **:** gras_sur_couteau
f_5 **:** mains_cuisinière_grasses
f_6 **:** maître_hôtel_sert_depuis_30_ans
f_7 **:** ¬ cuisinière_hérite
f_8 **:** ¬ maître_hôtel_hérite
f_9 **:** neveu_hérite
f_{10} **:** neveu_a_perdu_bras

Les arguments :

A_0 **:** ({cuisinière_coupable}, cuisinière_coupable)
$A_{0'}$ **:** ({maître_hôtel_coupable}, maître_hôtel_coupable)
$A_{0''}$ **:** ({neveu_coupable}, neveu_coupable)
A_1 **:** $(\{f_1, k_1\}$, cuisinière_coupable)
A_2 **:** $(\{f_2, k_2\}$, ¬cuisinière_coupable)
$A_{2'}$ **:** $(\{f_3, k_3\}$, maître_hôtel_coupable)
A_3 **:** $(\{f_4, f_5, k_4\}$, cuisinière_coupable)
A_4 **:** $(\{f_4, k_5, k_6\}$, ¬cuisinière_coupable)
A_5 **:** $(\{f_6, k_7\}$, ¬maître_hôtel_coupable)
A_6 **:** $(\{f_7, k_8\}$, ¬cuisinière_coupable)
$A_{6'}$ **:** $(\{f_8, k_{8'}\}$, ¬maître_hôtel_coupable)
$A_{6''}$ **:** $(\{f_9, k_{8''}\}$, neveu_coupable)
A_7 **:** $(\{f_4, k_9\}$, ¬neveu_coupable)
A_8 **:** $(\{f_{10}, f_4, k_{10}, k_{11}, k_{12}\}, \neg k_9)$

Et sous forme graphique, en n'utilisant pour relations de contrariété et d'appui que les relations `attaque`$_2$ et `appuie`$_2$, on obtient :



Contrairement aux exemples précédents, on a ici 3 conclusions possibles (donc 3 arguments "racine") :
– soit le coupable est Mme Benton, la cuisinière (argument A_0),
– soit le coupable est Partridge, le maître d'hôtel (argument $A_{0'}$),
– soit le coupable est Fenimore Brandford, le neveu de la victime (argument $A_{0''}$).

Chapitre 6

L'acceptabilité dans un SABP

En nous appuyant sur le cadre formel donné en section 5.2 page 19, nous allons définir de nouvelles sémantiques pour l'acceptabilité dans un SABP.

6.1 Notion d'attaque et d'appui d'un argument par un ensemble dans un SABP

Soit $<\mathcal{A}, \mathcal{R}_{att}, \mathcal{R}_{app}>$ un SABP.

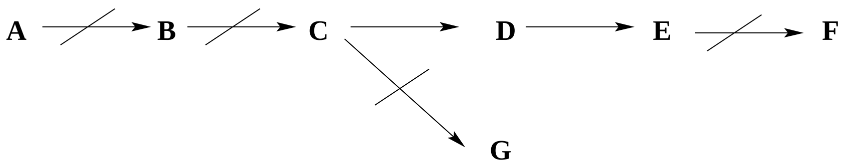
Définition 22 (Attaque/Appui directs par un ensemble dans un SABP) *Soit $S \subseteq \mathcal{A}$, soit $A \in \mathcal{A}$. On dira que :*
– S attaque directement A *ssi* $\exists B \in S$ *tel que* $B\mathcal{R}_{att}A$ *;*
– S appuie directement A *ssi* $\exists B \in S$ *tel que* $B\mathcal{R}_{app}A$*.*

Plus généralement :

Définition 23 (Attaque/Appui par un ensemble dans un SABP) *Soit $S \subseteq \mathcal{A}$, soit $A \in \mathcal{A}$. On dira que :*
– S attaque A *ssi* $\exists B \in S$ *tel que :*
  – *soit* $B\mathcal{R}_{att}A$*,*
  – *soit un des chemins de B vers A est une attaque appuyée.*
– S appuie A *ssi* $\exists B \in S$ *tel que un des chemins de B vers A est une séquence d'appuis.*

Remarquons qu'il existe d'autres définitions possibles pour cette notion d'attaque par un ensemble dans un SABP (par exemple, S attaque A ssi $\exists B \in S$ tel que : soit il existe une séquence d'attaques de B vers A, soit un des chemins de B vers A est une attaque appuyée). Toutefois, on aurait obtenu moins de propriétés intéressantes par la suite.

Remarquons aussi que les éléments B de S qui permettent à S d'attaquer A sont donc des attaquants directs ou indirects de A mais que les attaquants indirects de A ne forment pas toujours un ensemble susceptible d'attaquer A :



Dans cet exemple :
– les attaquants indirects de F sont D, C, A et celui de G est A,
– l'ensemble $\{C\}$ attaque G, F et appuie D, E,
– l'ensemble $\{A\}$ attaque B, mais pas G, ni F.

6.2 Notion d'ensemble sans conflit dans un SABP

On a ici deux types de conflits possibles :
- ceux dus à une attaque entre deux arguments,
- ceux dus au fait que l'on peut en même temps attaquer et appuyer un argument.

Nous allons d'abord nous intéresser au premier type de conflit.

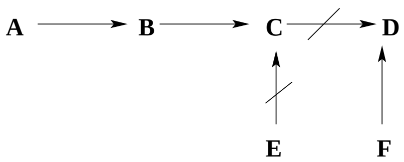
6.2.1 Définition d'ensemble sans conflit dans un SABP

Cette définition est la transposition de la définition donnée par [Dun95] au cas bipolaire en exploitant la notion d'“attaque” proposée en définition 23 page précédente.

Définition 24 (Ensemble sans-conflit dans un SABP) *Soit $S \subseteq \mathcal{A}$. S est* sans conflit *ssi $\nexists A, B \in S$ tels que $\{A\}$ attaque B.*

Ainsi, on empêche la présence simultanée de 2 arguments dont l'un attaque l'autre.

Un exemple :



Dans cet exemple, les ensembles suivants sont sans conflit et maximaux pour l'inclusion :
- $S_1 = \{A, B, C, F\}$,
- $S_2 = \{A, B, E, F\}$,
- $S_3 = \{D, E, F\}$.

Remarquons qu'il existe d'autres ensembles sans conflit (par exemple : $\{D\}$, $\{C\}$).
On constate toutefois qu'avec cette définition on obtient des ensembles qui peuvent présenter un aspect un peu gênant correspondant au second type de conflit évoqué en début de section 6.2 :
- S_1 attaque et appuie D,
- S_2 attaque et appuie C.

Nous allons donc proposer une autre définition qui prendra en compte cet aspect.

6.2.2 Définition d'ensemble sûr dans un SABP

On cherche ici à identifier des ensembles d'arguments qui ne pourront pas à la fois attaquer et appuyer un argument. Pour cela, nous nous inspirons des travaux de [Ver02] et de la notion d'*argument controversé* proposé par [Dun95] :

Définition 25 (Argument controversé [Dun95]) *Soit $<\mathcal{A}, \mathcal{R}>$ un système d'argumentation au sens de Dung. Un argument B est* controversé *ssi il existe un argument $A \in \mathcal{A}$ tel que B attaque indirectement A et B défend indirectement A*[1]*.*

En général, les systèmes d'argumentation $<\mathcal{A}, \mathcal{R}>$ ne contenant pas d'argument controversé possèdent de bonnes propriétés (voir [Dun95]).
La définition 24 correspond à l'absence de conflit interne. Celle que nous allons proposer maintenant correspond à une cohérence vis à vis de l'extérieur de l'ensemble. Elle exploite elle-aussi la notion d'“attaque” proposée en définition 23 page précédente.

Définition 26 (Ensemble sûr dans un SABP) *Soit $S \subseteq \mathcal{A}$. S est* sûr[2] *ssi $\nexists B \in \mathcal{A}$ tel que S attaque B et, soit S appuie B, soit $B \in S$.*

[1] Attention, ces notions d'attaque indirecte et de défense indirecte sont celles données par Dung et ne correspondent pas exactement aux nôtres !

[2] *safe* en anglais.

Reprenons l'exemple de la section 6.2.1 page précédente : S_1 et S_2 ne sont pas sûrs alors que S_3 l'est. Remarquons aussi que, si on s'intéresse aux ensembles sûrs maximaux pour l'inclusion ensembliste, on ne trouve que $\{A, B, C\}$ et $\{D, E, F\}$.

Nous avons alors la propriété suivante :

Propriété 5 *Si S est sûr alors S est sans conflit.*

La preuve est évidente d'après les définitions 23 page 29, 24 page précédente et 26 page ci-contre.

6.3 Notion d'extension stable dans un SABP

Maintenant que nous avons défini la notion d'ensemble sans conflit et celle d'ensemble sûr, nous allons proposer une première sémantique d'acceptabilité en adaptant la définition de la sémantique stable proposée par [Dun95]. Nous utiliserons bien-sûr la notion d'"attaque" proposée en définition 23 page 29.

Définition 27 (Extension stable dans un SABP) *Soit $S \subseteq \mathcal{A}$. S est une* extension stable *ssi S est sans conflit et $\forall A \notin S$, S attaque A.*

Reprenons l'exemple de la section 6.2.1 page ci-contre : S_2 est une extension stable (mais n'est pas sûr).

Nous avons alors quelques propriétés :

Propriété 6 *Soit $S \subseteq \mathcal{A}$ un ensemble sûr.*
Si S est stable alors S contient tous les arguments qu'il appuie.
On dira que S est fermé pour la relation $\mathcal{R}_{app}$.

Propriété 7 *Soit $S \subseteq \mathcal{A}$ un ensemble fermé pour la relation $\mathcal{R}_{app}$.*
S attaque A ssi S attaque directement A.

Conséquence 1 *Si S est un ensemble sûr et une extension stable, alors S attaque directement tout argument qui n'appartient pas à S.*

Remarquons que, dans l'exemple donné en section 6.2.1 page précédente, il n'y a pas d'ensemble à la fois sûr et stable.

Propriété 8 *Soit $S \subseteq \mathcal{A}$ sans-conflit et fermé pour la relation $\mathcal{R}_{app}$. S est sûr.*

On constate donc que :
- sûr et stable $\Rightarrow$ fermé pour $\mathcal{R}_{app}$,
- fermé pour $\mathcal{R}_{app}$ et sans conflit $\Rightarrow$ sûr.

Reprenons encore une fois l'exemple donné en section 6.2.1 page ci-contre et récapitulons les propriétés des différents ensembles :

Propriétés	S_1	S_2	S_3	$S'_1 \subseteq S_1$
sans conflit	X	X	X	X
sûr	—	—	X	X
fermé pour $\mathcal{R}_{app}$	—	—	X	X
stable	—	X	—	—

avec X signifiant propriété possédée, — propriété non possédée et :
- $S_1 = \{A, B, C, F\}$,
- $S_2 = \{A, B, E, F\}$,
- $S_3 = \{D, E, F\}$,
- $S'_1 = \{A, B, C\}$.

6.4 Notion d'admissibilité dans un SABP

Nous proposons maintenant une notion d'admissibilité dans le cadre d'un SABP en utilisant une nouvelle notion de défense collective qui est directement issue de celle proposée par [Dun95] mais utilisant la notion d'“attaque” proposée en définition 23 page 29 :

Définition 28 (Défense par un ensemble dans un SABP) *Soit $S \subseteq \mathcal{A}$. Soit $A \in \mathcal{A}$. S* défend *A ssi $\forall B \in \mathcal{A}$, si B attaque A alors $\exists C \in S$ tel que C attaque B.*

Nous avons quelques propriétés concernant cette notion de défense collective :

Propriété 9 *Si S est un ensemble fermé pour $\mathcal{R}_{app}$ et défend A alors $\forall B \in \mathcal{A}$, si B attaque A alors S attaque directement B .*

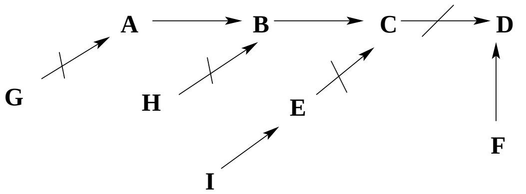
Cette propriété est une conséquence de la propriété 7 page précédente.

Propriété 10 *Toute extension stable défend ses éléments.*

Nous allons définir la nouvelle notion d'admissibilité en combinant les notions d'ensemble sans conflit, de fermeture pour $\mathcal{R}_{app}$ et la nouvelle défense collective.

Définition 29 (Ensemble admissible dans un SABP) *Soit $S \subseteq \mathcal{A}$. S est* admissible *ssi S est sans-conflit, fermé pour $\mathcal{R}_{app}$ et défend tous ses éléments.*

Dans l'exemple donné en section 6.2.1 page 30 : aucun des ensembles S_1, S_2 et S_3 ne sont admissibles (par exemple, pour S_3, on a A et B attaquants de D au même titre que C et S_3 ne défend D ni contre A ni contre B). Par contre, sur l'exemple suivant :



on a l'ensemble $S_4 = \{G, H, E, F, D\}$ qui est admissible.

Propriété 11 *Tout ensemble admissible est sûr.*

C'est une conséquence de la propriété 8 page précédente et de la définition 29.

Conséquence 2 *Un ensemble stable et fermé pour $\mathcal{R}_{app}$ est admissible.*
Un ensemble sûr et stable est admissible.

La réciproque est fausse (voir l'ensemble S_4 de l'exemple précédent : S_4 est admissible, $I \notin S_4$ et S n'attaque pas I).

Propriété 12 *Soit $<\mathcal{A}, \mathcal{R}_{att}, \mathcal{R}_{app}>$ un système d'argumentation bipolaire. Il existe toujours au moins un ensemble admissible.*

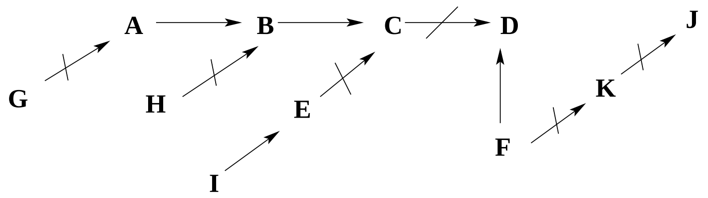
La preuve est évidente : il suffit de constater que $\varnothing$ est un ensemble admissible.

Remarques :
- Si on relâche dans la définition 29, la contrainte sur la fermeture pour $\mathcal{R}_{app}$, on perd la propriété d'avoir un ensemble sûr.
- Parmi les ensembles admissibles, on aura les extensions stables.

Bien-sûr, on peut ensuite retrouver la notion d'extension préférée comme dans [Dun95] :

Définition 30 (Extension préférée dans un SABP) *Un ensemble $E \subseteq \mathcal{A}$ est une extension préférée si et seulement si E est maximal pour l'inclusion parmi les ensembles admissibles.*

Exemple :



Seul l'ensemble $E_2 = \{G, H, I, E, F, D, J\}$ est une extension préférée du SABP.

Remarque : La propriété "toute extension stable est aussi préférée", qui est vraie dans le cas des systèmes d'argumentation de [Dun95], ne s'applique plus dans le cas des SABP (il suffit de constater sur l'exemple donné en section 6.2.1 page 30 que l'extension stable S_2 n'est pas sûre et donc qu'elle ne pourra pas être un ensemble admissible).

6.5 Application aux exemples de la section 3 page 7

Dans le cas de l'exemple d'orientation on obtient :
– l'extension stable est $\{A_2, A_4, A_7, A_5, A_8, A_9, A_{10}, A_{11}, A_{12}, A_{13}, A_{15}\}$, mais elle n'est pas sûre ;
– l'extension préférée est $\{A_2, A_4, A_7, A_5, A_{12}, A_{15}\}$ et elle n'est pas stable.
Dans le cas de l'exemple de la prothèse on obtient :
– l'extension stable est $\{A_2, A_4, A_3, A_6, A_7, A_{10}, A_8\}$, mais elle n'est pas sûre ;
– l'extension préférée est $\{A_2, A_4, A_3, A_6, A_{10}\}$ et elle n'est pas stable.
Dans le cas de l'exemple de l'enquête policière on obtient :
– l'extension stable est $\{A_1, A_2, A_3, A_4, A_6, A_{2'}, A_5, A_{6'}, A_{6''}, A_8, A_{0''}\}$, mais elle n'est pas sûre ;
– l'extension préférée est $\{A_2, A_3, A_4, A_6, A_5, A_{6'}, A_{6''}, A_8, A_{0''}\}$ et elle n'est pas stable.
On constate ici que $A_{0''}$ appartient aux extensions ce qui n'est pas le cas de A_0, ni de $A_{0'}$.

Chapitre 7

Une évaluation graduelle locale dans un SABP

Nous nous plaçons donc dans le cadre d'un SABP $<\mathcal{A}, \mathcal{R}_{att}, \mathcal{R}_{app}>$ et nous supposerons que les relations $\mathcal{R}_{att}$ et $\mathcal{R}_{app}$ sont indépendantes.
Nous allons chercher à généraliser l'approche locale de l'évaluation graduelle des contrariétés proposée dans [CLS01] pour un système d'argumentation $<\mathcal{A}, \mathcal{R}_{att}>$ en rajoutant la relation d'appui $\mathcal{R}_{app}$.

7.1 Les principes à respecter

Soit v une évaluation graduelle locale dans un système d'argumentation bipolaire, v devrait respecter les principes suivants :

P1 L'évaluation d'un argument est fonction de l'évaluation de tous ses attaquants directs (la contrariété directe) et de tous ses appuis directs.

P2 Si la qualité de l'appui augmente alors la valeur de l'argument ainsi appuyé augmente et si la qualité de l'attaque augmente alors la valeur de l'argument ainsi attaqué diminue.

P3 Si on ajoute des appuis alors la qualité de l'appui augmente et si on ajoute des attaques alors la qualité de l'attaque augmente.

7.2 Définition d'une évaluation graduelle locale dans un SABP

Soit le système d'argumentation bipolaire $<\mathcal{A}, \mathcal{R}_{att}, \mathcal{R}_{app}>$.
On dispose d'un ensemble V totalement ordonné admettant un plus petit élément (V_{Min}) et un plus grand élément (V_{Max}).

Définition 31 *Soit $A \in \mathcal{A}$ avec ${\mathcal{R}_{att}}^-(A) = \{B_1, \dots, B_n\}$ et ${\mathcal{R}_{app}}^-(A) = \{C_1, \dots, C_p\}$. Une évaluation est une application $v\ : \mathcal{A} \to V$ telle que*

$$v(A) = g(h_{app}(v(C_1), \dots, v(C_p)), h_{att}(v(B_1), \dots, v(B_n)))$$

avec :

la fonction h_{att} **:** $V^* \to \mathcal{H}_{att}$ *(V^* dénote l'ensemble des suites finies d'éléments de V et $\mathcal{H}_{att}$ est un ensemble ordonné) évaluant la qualité de l'attaque sur A et vérifiant les contraintes suivantes :*
- *pour toute permutation $(x_{i_1}, \dots, x_{i_n})$ de $(x_1, \dots, x_n)$, $h_{att}(x_{i_1}, \dots, x_{i_n}) = h_{att}(x_1, \dots, x_n)$,*
- *si $x_i \geq x'_i$ alors $h_{att}(x_1, \dots, x_i \dots, x_n) \geq h_{att}(x_1, \dots, x'_i \dots, x_n)$,*
- *$h_{att}(x_1, \dots, x_n, x_{n+1}) \geq h_{att}(x_1, \dots, x_n)$,*
- *$h_{att}() = \alpha \leq h_{att}(x_1, \dots, x_n)$, pour tous $x_1, \dots, x_n$ (donc α représentera la "valeur minimale" d'une attaque – c'est-à-dire quand il n'y a pas d'attaque !),*
- *$h_{att}(x_1, \dots, x_n) \leq \beta$, pour tous $x_1, \dots, x_n$ (donc β représentera la "valeur maximale" d'une attaque – par exemple, quand il y a une infinité d'attaques[1]).*

[1] Un tel argument n'existe pas forcément !

la fonction h_{app} **:** $V^* \to \mathcal{H}_{app}$ *(V^* dénote l'ensemble des suites finies d'éléments de V et $\mathcal{H}_{app}$ est un ensemble ordonné) évaluant la qualité de l'appui sur A et vérifiant les contraintes suivantes :*
- *pour toute permutation $(x_{i_1}, \ldots, x_{i_n})$ de $(x_1, \ldots, x_n)$, $h_{app}(x_{i_1}, \ldots, x_{i_n}) = h_{app}(x_1, \ldots, x_n)$,*
- *si $x_i \geq x'_i$ alors $h_{app}(x_1, \ldots, x_i \ldots, x_n) \geq h_{app}(x_1, \ldots, x'_i \ldots, x_n)$,*
- *$h_{app}(x_1, \ldots, x_n, x_{n+1}) \geq h_{app}(x_1, \ldots, x_n)$,*
- *$h_{app}() = \alpha \leq h_{app}(x_1, \ldots, x_n)$, pour tous $x_1, \ldots, x_n$ (donc α représentera la "valeur minimale" d'un appui – c'est-à-dire quand il n'y a pas d'appui !),*
- *$h_{app}(x_1, \ldots, x_n) \leq \beta$, pour tous $x_1, \ldots, x_n$ (donc β représentera la "valeur maximale" d'un appui – par exemple, quand il y a une infinité d'appuis[2]).*

la fonction g **:** $\mathcal{H}_{app} \times \mathcal{H}_{att} \to V$ *vérifiant la contrainte suivante :*
- *$g(x, y)$ est croissante en x et décroissante en y.*

Avec cette définition, nous avons les propriétés suivantes :

Propriété 13
- *$\forall x, y$, $g(x, \alpha) \geq g(\alpha, y)$.*
- *$g(\beta, \alpha) = V_{Max}$ et $g(\alpha, \beta) = V_{Min}$.*
- *Si ${\mathcal{R}_{att}}^-(A) = {\mathcal{R}_{app}}^-(A) = \varnothing$ alors $v(A) = g(\alpha, \alpha)$.*
- *Si ${\mathcal{R}_{att}}^-(A) \neq \varnothing$ et ${\mathcal{R}_{app}}^-(A) = \varnothing$ alors $v(A) = g(\alpha, y) \leq g(\alpha, \alpha)$ pour $y \geq \alpha$.*
- *Si ${\mathcal{R}_{att}}^-(A) = \varnothing$ et ${\mathcal{R}_{app}}^-(A) \neq \varnothing$ alors $v(A) = g(x, \alpha) \geq g(\alpha, \alpha)$ pour $x \geq \alpha$.*

Cela nous conduit donc à l'échelle suivante :

$$\begin{array}{ccccccccc} V_{\text{Min}} & \leq & g(\alpha, y) & \leq & g(\alpha, \alpha) & \leq & g(x, \alpha) & \leq & V_{\text{Max}} \\ & & (\text{pour } y \geq \alpha) & & & & (\text{pour } x \geq \alpha) & & \end{array}$$

Propriété 14 *Soit v une évaluation au sens de la définition 31 page précédente, v respecte les principes* **P1** *à* **P3***.*

Définition 32 *Soit A et B deux arguments. Si $v(A) < v(B)$ alors $A \prec B$.*

7.3 Trois instances d'évaluation graduelle locale dans un SABP

7.3.1 Première instance

Posons $V = \{m, I, M\}$ avec $V_{\text{Min}} = m$ et $V_{\text{Max}} = M$. On va donc avoir un modèle d'évaluation à 3 valeurs. Posons aussi que les échelles de valeur sont les mêmes pour toutes les fonctions h_{app}, h_{att} et v (donc $\mathcal{H}_{att} = \mathcal{H}_{app} = V$).
On peut donc poser que $m = \alpha$, $M = \beta$ et on en conclut que $I = g(m, m)$ et que l'on a $m < I = g(m, m) < M$.
D'après les contraintes sur g vues en section 7.2 page précédente, on a :
- $g(M, m) = M$ (appui maximal et pas d'attaque),
- $g(m, M) = m$ (pas d'appui et attaque maximale).

Donc :

$$m = g(m, M) \leq g(m, I) \leq g(m, m) = I \leq g(I, m) \leq g(M, m) = M$$

mais aussi (puisque $m < I < M$ et que $g(x, y)$ est croissante en x et décroissante en y) :

$$m = g(m, M) \leq g(I, M) \leq g(M, M) \leq g(M, I) \leq g(M, m) = M$$

À ce stade, on a divers choix. Si on impose aussi que $g(M, M) = I$ et que $g(I, I) = I$. On obtient alors les contraintes supplémentaires suivantes pour la fonction g :

$$\begin{array}{c} g(m, I) \in \{m, I\} \\ g(I, m) \in \{I, M\} \\ g(I, M) \in \{m, I\} \\ g(M, I) \in \{I, M\} \end{array}$$

[2]Même remarque que pour l'attaque.

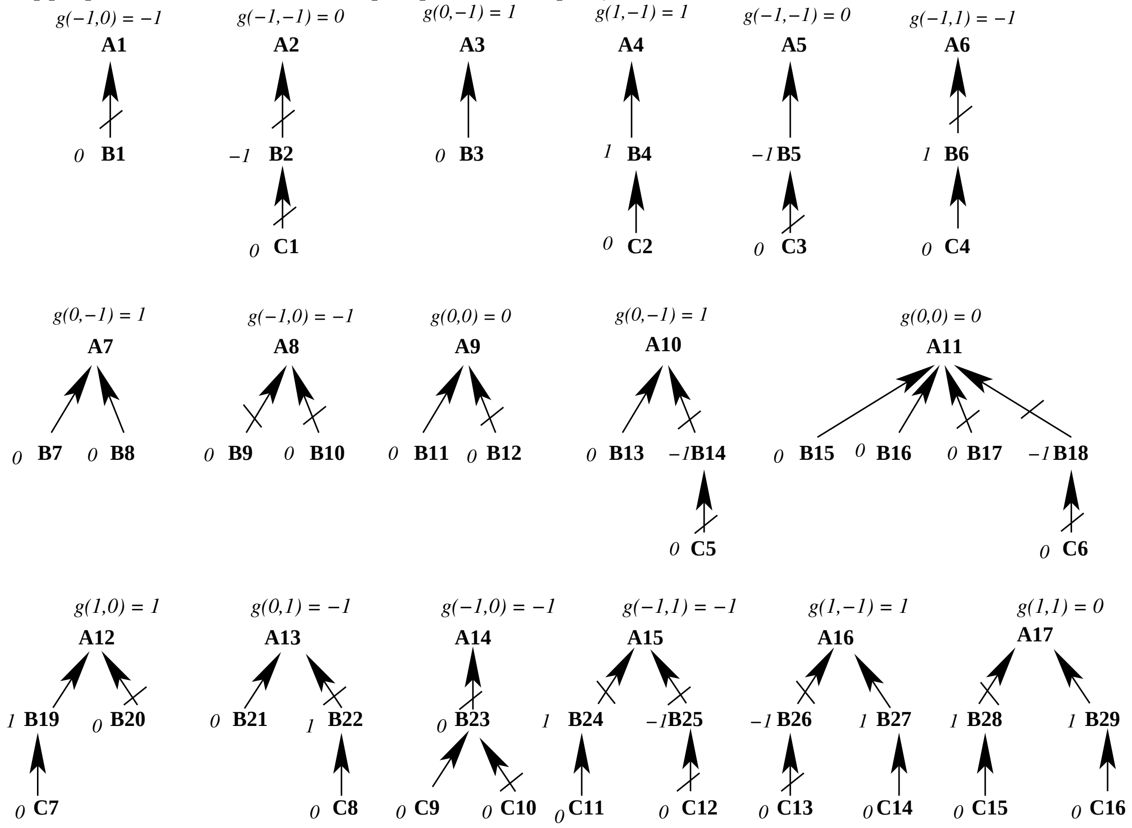
Et donc la table suivante pour la fonction g :

$g(x,y)$	m	I	M
m	I	m ou I	m
I	I ou M	I	m ou I
M	M	I ou M	I

Un exemple : $V = \{-1, 0, 1\}$ avec $h_{att} = h_{app} = \max$ et

$g(x,y)$	-1	0	1
-1	0	-1	-1
0	1	0	-1
1	1	1	0

Si on applique cette évaluation sur quelques cas simples, on obtient :



En étudiant plus précisément les exemples proposés, on constate que l'on peut identifier au moins 7 classes différentes d'arguments que l'on peut classer d'après l'"ordre idéal"[3] attendu sur les valeurs dans le tableau suivant (de la plus mauvaise valeur à la meilleure) :

<table>
<tr><td>de la plus mauvaise valeur</td><td colspan="3">arguments racines que de séquences d'attaque</td></tr>
<tr><td rowspan="3">vers</td><td rowspan="3">arguments racines de séquences d'attaque, de défense et d'appui</td><td>arguments racines que de séquences d'attaque et de défense</td><td rowspan="3">arguments racines que de séquences d'attaque et d'appui</td></tr>
<tr><td>arguments racines que de séquences de défense</td></tr>
<tr><td>arguments racines que de séquences d'appui et de défense</td></tr>
<tr><td>la meilleure valeur</td><td colspan="3">arguments racines que de séquences d'appui</td></tr>
</table>

[3]Cet ordre peut ne pas être respecté sur certains cas particuliers, mais il représente la tendance vers laquelle les principes choisis en section 7.1 page 35 nous amènent !

En effet, d'après les principes donnés en section 7.1 page 35, les arguments ne possédant que des appuis devraient être meilleurs que ceux possédant au moins une attaque ou une défense. D'autre part, la décroissance de $g(x,y)$ sur y représentant les attaques implique la prise en compte de la notion de défense et induit que les arguments possédant des défenses sont meilleurs que ceux possédant des attaques. Il existe par contre des cas où l'on ne peut trancher ; par exemple, il est impossible *a priori* de dire si un argument possédant des attaques et des appuis est meilleur qu'un autre argument possédant des défenses et des appuis.
En utilisant cette idée, on partitionne les arguments A_i de l'exemple donné précédemment :

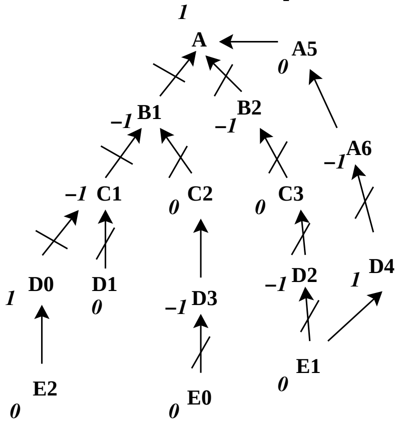
<table>
<tr><td colspan="3">A_1, A_6, A_8</td></tr>
<tr><td rowspan="3">A_{11}</td><td>A_{14}, A_{15}</td><td rowspan="3">A_9, A_{12}, A_{13}, A_{17}</td></tr>
<tr><td>A_2</td></tr>
<tr><td>A_{10}, A_{16}</td></tr>
<tr><td colspan="3">A_3, A_4, A_5, A_7</td></tr>
</table>

Puis, on compare avec l'ordre obtenu entre les A_i :

$$A_1, A_6, A_8, A_{13}, A_{14}, A_{15} \prec A_2, A_5, A_9, A_{11}, A_{17} \prec A_3, A_4, A_7, A_{10}, A_{12}, A_{16}$$

On peut constater que :
- les arguments uniquement racines de séquences d'attaque sont moins bons que ceux qui sont uniquement racines de séquences d'appui ;
- les arguments uniquement racines de séquences de défense sont moins bons que ceux qui sont racines de séquences d'appui et de séquences de défense ;
- par contre, il arrive que certains arguments racines à la fois de séquences d'attaque et de séquences d'appui soient meilleurs que des arguments uniquement racines de séquences d'appui (voir A_5 et A_{12}) ;
- on peut aussi constater que les appuis peuvent être affaiblis (voir par exemple l'appui de B_5 sur A_5 qui est affaibli par l'attaque sur B_5 : la valeur de A_5 est moins bonne que celle de A_3 !).

Un autre exemple Appliquons cette évaluation à l'exemple suivant :

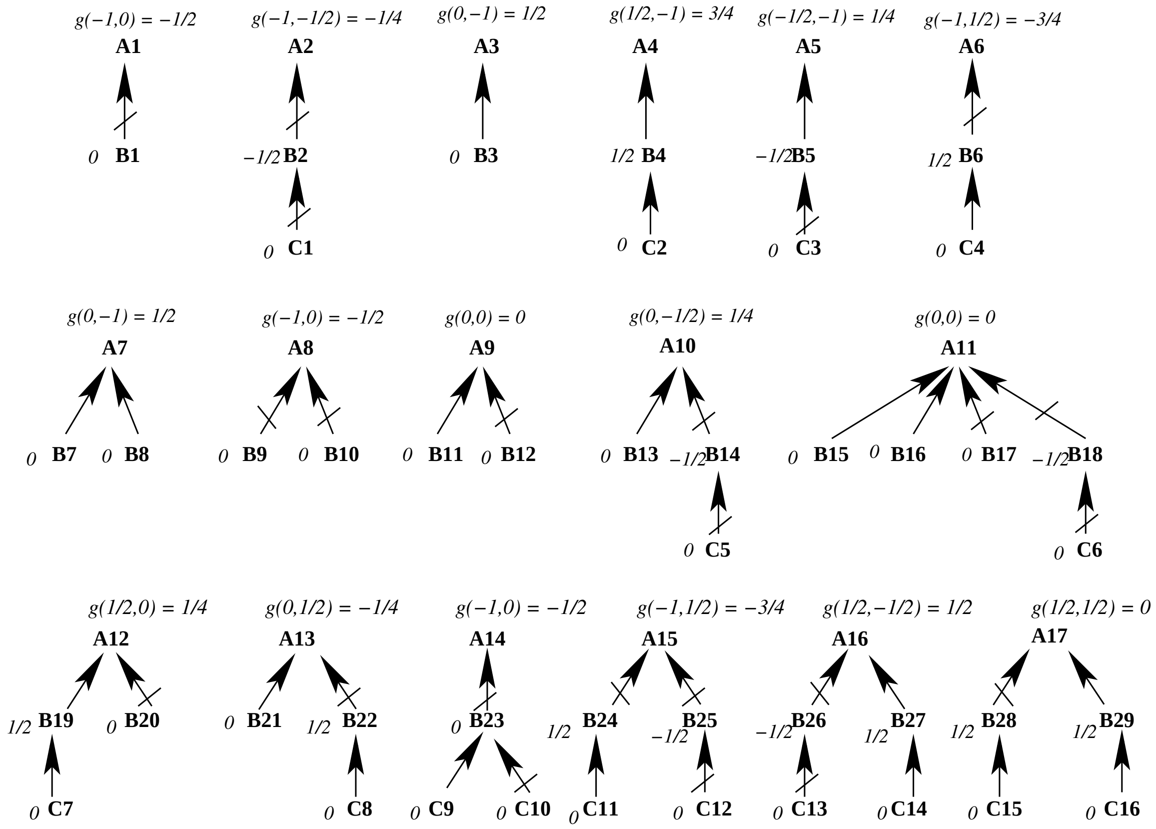


7.3.2 Seconde instance

Posons que :
- $\mathcal{H}_{att} = \mathcal{H}_{app} = V = [-1, 1]$ intervalle de réels,
- $h_{att} = h_{app} = \text{Max}$,
- $g(x,y) = \frac{x-y}{2}$,
- $\alpha = -1$ et $\beta = 1$ et $g(\alpha, \alpha) = 0$.

On a bien $h_{att}(x_1, \ldots, x_n, x_{n+1}) \geq h_{att}(x_1, \ldots, x_n)$ puisque $\text{Max}(x_1, \ldots, x_n, x_{n+1}) \geq \text{Max}(x_1, \ldots, x_n)$ (et idem pour h_{app}).

Quelques exemples :



Cela donne l'ordre suivant entre les arguments A_i :

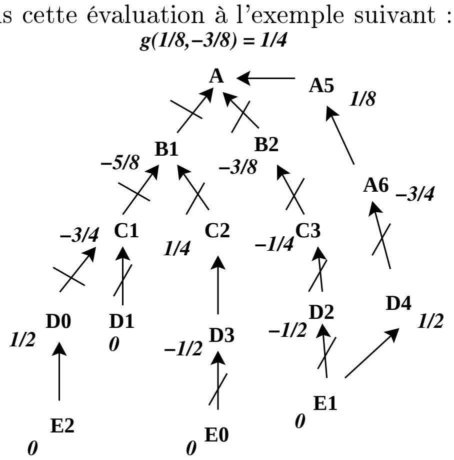
$$A_6, A_{15} \prec A_1, A_8, A_{14} \prec A_{13}, A_2 \prec A_9, A_{11}, A_{17} \prec A_5, A_{10}, A_{12} \prec A_3, A_7, A_{16} \prec A_4$$

Remarquons que l'ordre obtenu raffine en général l'ordre proposé en section 7.3.1 page 36 excepté pour A_{13} par rapport à A_2 (idem entre A_5 et A_{10}) : dans la section 7.3.1 page 36, on a $A_{13} < A_2$, alors qu'ici A_{13} et A_2 ont la même valeur.

Puis, en utilisant la partition des arguments A_i (voir section 7.3.1 page 36), on retrouve les mêmes remarques que celles données en section 7.3.1 page 36 :

- les arguments uniquement racines de séquences d'attaque (A_1, A_6, A_8) sont moins bons que ceux qui sont uniquement racines de séquences d'appui (A_3, A_4, A_5, A_7) ;
- les arguments uniquement racines de séquences de défense (A_2) sont moins bons que ceux qui sont racines de séquences d'appui et de séquences de défense (A_{10}, A_{16}) ;
- il arrive que certains arguments racines à la fois de séquences d'attaque et de séquences d'appui soient aussi bons que des arguments uniquement racines de séquences d'appui (ici aussi, voir A_5 et A_{12}).

Un autre exemple Appliquons cette évaluation à l'exemple suivant :

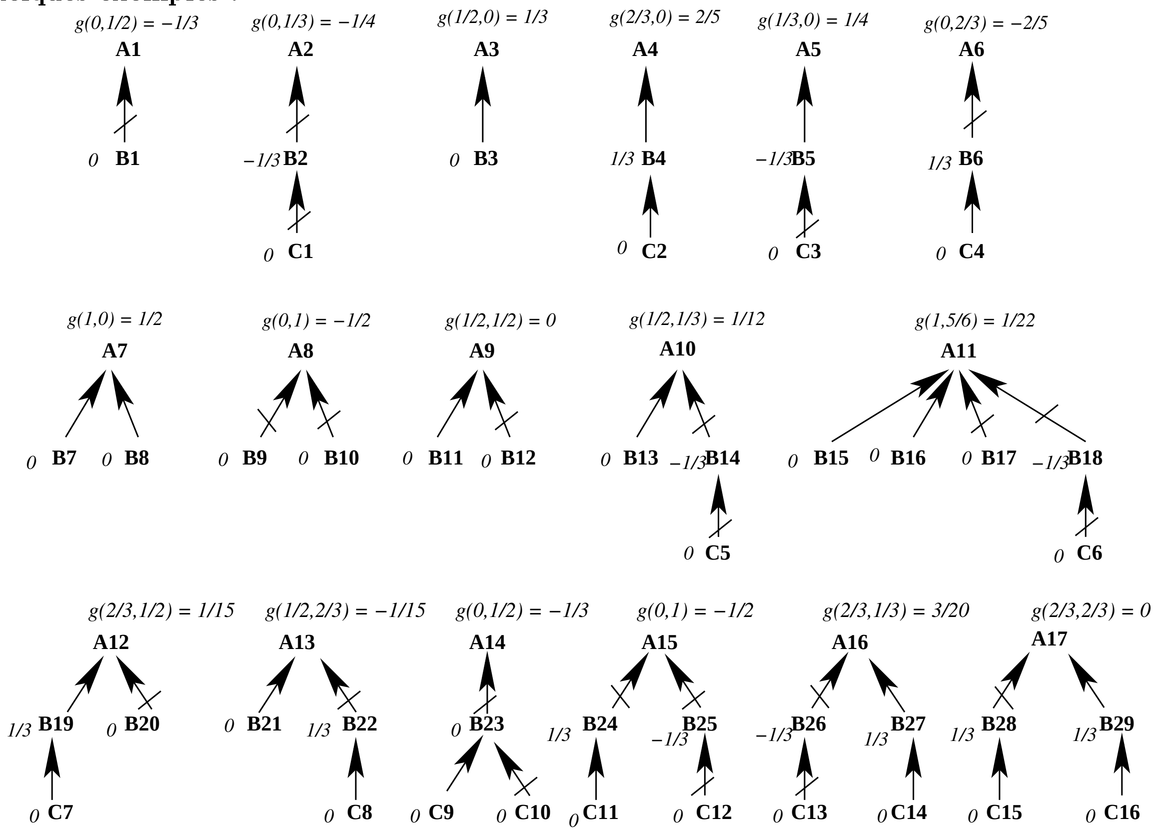


7.3.3 Troisième instance

Posons que :
- $V = [-1, 1]$ intervalle de réels,
- $\mathcal{H}_{att} = \mathcal{H}_{app} = [0, \infty[$ intervalle de réels,
- $h_{att}(x_1, \ldots, x_n) = h_{app}(x_1, \ldots, x_n) = \Sigma_{i=1}^{n} \frac{x_i+1}{2}$,
- $g(x, y) = \frac{1}{1+y} - \frac{1}{1+x}$.

On a alors $\alpha = 0$ et $\beta = \infty$ et $g(\alpha, \alpha) = 0$.
On a bien $h_{att}(x_1, \ldots, x_n, x_{n+1}) \geq h_{att}(x_1, \ldots, x_n)$ puisque $\frac{x_{n+1}+1}{2} \geq 0$ quand $x_{n+1} \in [-1, 1]$ (et idem pour h_{app}).
On a aussi $h_{att}() = h_{app}() = \alpha$ et α valeur minimale de $[0, \infty[$, alors que β est la valeur maximale de $[0, \infty[$.
On vérifie aussi que $g(\alpha, \beta) = g(0, \infty) = -1$ et que $g(\beta, \alpha) = g(\infty, 0) = 1$ (1 et -1 étant respectivement V_{Min} et V_{Max}).

Quelques exemples :

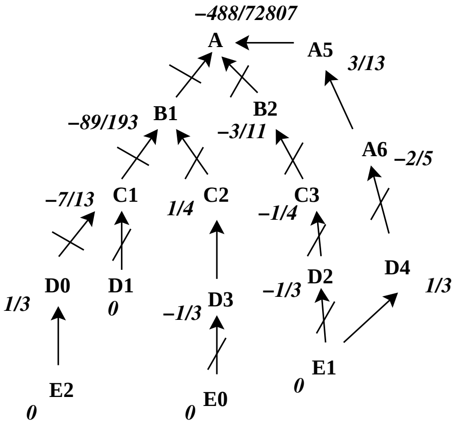


Cela donne l'ordre suivant entre les arguments A_i :

$$A_8, A_{15} \prec A_6 \prec A_1, A_{14} \prec A_2 \prec A_{13} \prec A_9, A_{17} \prec A_{11} \prec A_{12} \prec A_{10} \prec A_{16} \prec A_5 \prec A_3 \prec A_7 \prec A_4$$

Remarquons que l'ordre obtenu raffine en général l'ordre proposé en section 7.3.1 page 36 (excepté pour A_5-A_{12} et A_2-A_{13}) tout en étant légèrement différent de celui proposé en section 7.3.2 page 38 (voir l'emplacement de A_5).
Puis, en utilisant la partition des arguments A_i (voir section 7.3.1 page 36), on retrouve presque les mêmes remarques que celles données en section 7.3.1 page 36 et en section 7.3.2 page 38 :
- les arguments uniquement racines de séquences d'attaque (A_1, A_6, A_8) sont moins bons que ceux qui sont uniquement racines de séquences d'appui (A_3, A_4, A_5, A_7) ;
- les arguments uniquement racines de séquences de défense (A_2) sont moins bons que ceux qui sont racines de séquences d'appui et de séquences de défense (A_{10}, A_{16}).

Un autre exemple Appliquons cette évaluation à l'exemple suivant :



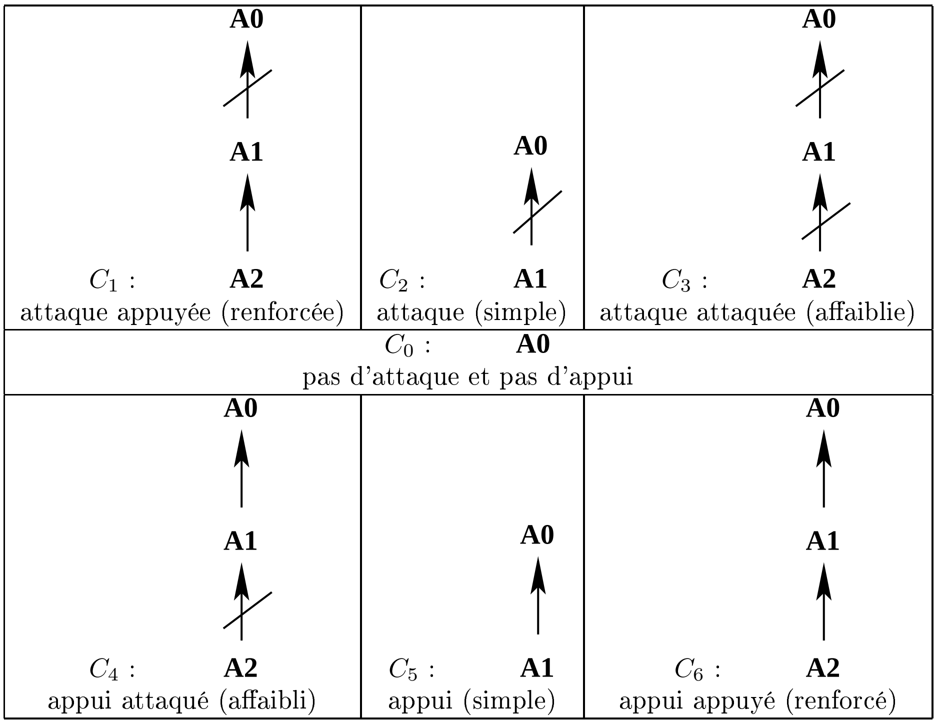
7.3.4 Analyse comparative

Ce travail a été réalisé en collaboration avec Dominique Gay, lors de son stage de DEA (voir [Gay04]). Il s'agit d'analyser le comportement des différentes instances d'évaluation locale bipolaire proposées ici. Cette analyse va se faire suivant plusieurs axes :

- d'abord sur des configurations de base (nous en avons identifié 7 – voir section 7.3.4.1),
- puis sur des configurations de saturation en appui ou en attaque (voir section 7.3.4.2 page suivante),
- et enfin sur des cas d'équilibre, c'est-à-dire qu'on va rechercher les configurations dans lesquelles la valeur de l'argument se trouve exactement à l'équilibre entre les attaques et les appuis (voir section 7.3.4.3 page 44). Un cas particulier sera alors la recherche des mécanismes permettant à une valeur de "basculer" autour de ce point d'équilibre.

7.3.4.1 Les configurations de base

Nous avons identifié 7 configurations de base notées C_0 à C_6 qui représentent les cas les plus simples d'interaction (depuis celui où il n'y a pas d'interaction, à ceux où l'on a deux interactions consécutives) :



Remarquons que C_3 correspond à la notion de défense.
Remarquons aussi que, d'après les principes donnés en section 7.1 page 35, on doit avoir :

$$C_1 \leq C_2 \leq C_3 \leq C_0 \leq C_4 \leq C_5 \leq C_6$$

(c'est-à-dire que la valeur de A_0 dans C_1 doit être plus faible que celle de A_0 dans C_2 ... qui doit être plus faible que celle de A_0 dans C_6)
On donne dans les tableaux suivants la valeur obtenue par A_0 dans chacune des 7 configurations de base.

Résultats avec l'instance 1 :

C_1	C_2	C_3	C_0	C_4	C_5	C_6
-1	-1	0	0	0	1	1

Résultats avec l'instance 2 :

C_1	C_2	C_3	C_0	C_4	C_5	C_6
$-\frac{3}{4}$	$-\frac{1}{2}$	$-\frac{1}{4}$	0	$\frac{1}{4}$	$\frac{1}{2}$	$\frac{3}{4}$

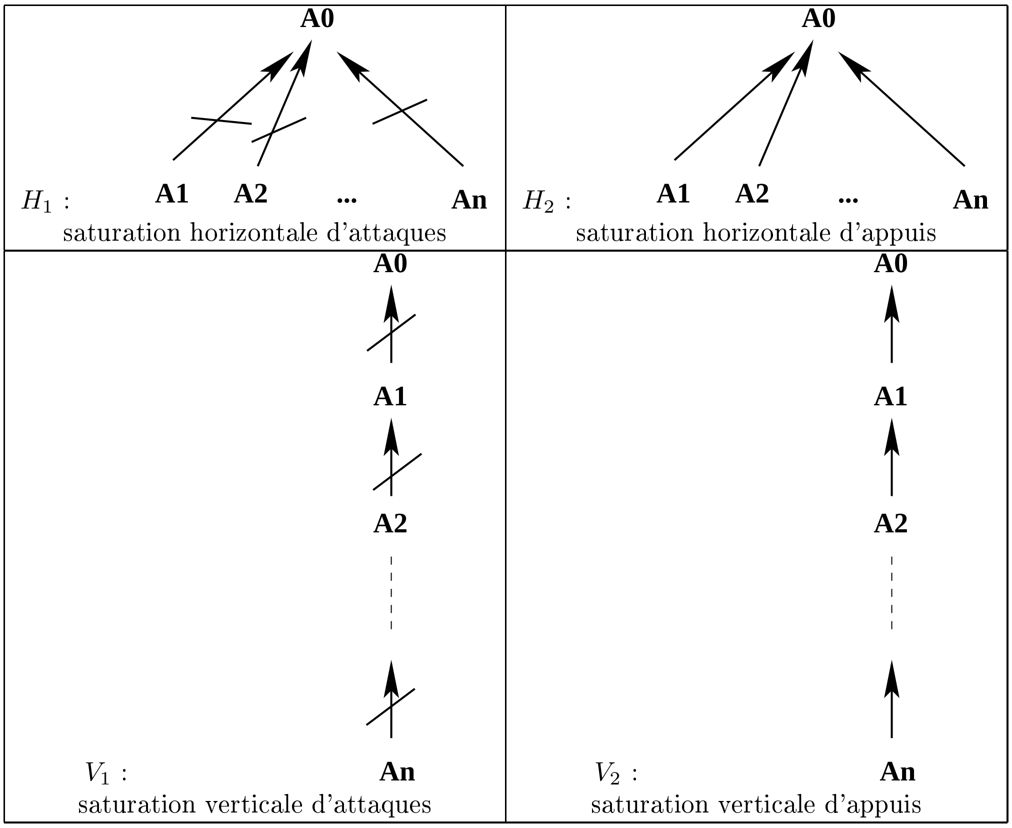
Résultats avec l'instance 3 :

C_1	C_2	C_3	C_0	C_4	C_5	C_6
$-\frac{2}{5}$	$-\frac{1}{3}$	$-\frac{1}{4}$	0	$\frac{1}{4}$	$\frac{1}{3}$	$\frac{2}{5}$

Pour toutes les instances, les principes sont bien respectés et on a $C_1 \leq C_2 \leq C_3 \leq C_0 \leq C_4 \leq C_5 \leq C_6$.
D'autre part, les résultats permettent d'identifier une certaine "symétrie" entre les configurations que nous exploiterons lorsque que nous chercherons les configurations d'équilibre (C_1 "symétrique"[4] avec C_6, idem entre C_2 et C_5 et entre C_3 et C_4).

7.3.4.2 Les configurations de saturation

Ici, on va étudier l'effet d'une saturation d'attaques ou d'appuis sur la valeur des arguments. Pour cela, on propose deux types de configuration, celles correspondant à des saturations horizontales et celles correspondant à des saturations verticales :



[4] C'est-à-dire que la valeur de A_0 dans C_1 est le symétrique de celle de A_0 dans C_6.

Les principes énoncés en section 7.1 page 35 permettent de dire que $H_1 \leq H_2$, $V_1 \leq V_2$, $H_1 \leq V_2$ et $V_1 \leq H_2$. Par contre, ils ne permettent pas de comparer V_1 avec H_1 et V_2 avec H_2.
On donne dans les tableaux suivants la valeur obtenue par A_0 dans chacune des 4 configurations de saturation.

Résultats avec l'instance 1 :

H_1	H_2	V_1	V_2
-1 ($\forall n$)	1 ($\forall n$)	0 (si n pair), -1 sinon	1 ($\forall n$)

Les preuves pour H_1, H_2 et V_2 sont évidentes et celle pour V_1 est très simple aussi : les valeurs des arguments le long de la branche $A_n - \ldots - A_1 - A_2$ respectent la séquence 0 (pour A_n), -1 (pour A_{n-1}), 0 (pour A_{n-2}), -1, 0, ... donc si n est pair la valeur de A_0 sera 0 et sinon elle sera -1. Notons que si n est pair alors la branche $A_n - \ldots - A_1 - A_2$ est une branche de défense pour A_0 et sinon c'est une branche d'attaque.

Résultats avec l'instance 2 :

H_1	H_2	V_1	V_2
$-\frac{1}{2}$ ($\forall n$)	$\frac{1}{2}$ ($\forall n$)	$-\frac{1}{3}$ quand $n \to \infty$	1 quand $n \to \infty$

Les preuves pour H_1 et H_2 sont évidentes et celles pour V_1 et V_2 reposent sur le calcul du point fixe d'une fonction :
- pour V_1 : posons $U_0 = v(A_n) = 0$ et $U_i = v(A_{n-i})$; on sait que $U_n = g(\alpha, U_{n-1}) = \frac{-1-U_{n-1}}{2}$; on a donc une suite qui converge vers le point fixe de la fonction $x \to -\frac{1}{2}(1+x)$ que l'on obtient en résolvant l'équation $x = -\frac{1}{2}(1+x)$; cette équation du premier degré n'a qu'une racine appartenant à l'intervalle $[-1, 1]$: $-\frac{1}{3}$.
- pour V_2 : posons $U_0 = v(A_n) = 0$ et $U_i = v(A_{n-i})$; on sait que $U_n = g(U_{n-1}, \alpha) = \frac{1+U_{n-1}}{2}$; on a donc une suite qui converge vers le point fixe de la fonction $x \to \frac{1}{2}(1+x)$ que l'on obtient en résolvant l'équation $x = \frac{1}{2}(1+x)$; cette équation du premier degré n'a qu'une racine appartenant à l'intervalle $[-1, 1]$: 1.

Résultats avec l'instance 3 :

H_1	H_2	V_1	V_2
-1 quand $n \to \infty$	1 quand $n \to \infty$	$\sqrt{3} - 2$ quand $n \to \infty$	$\sqrt{2} - 1$ quand $n \to \infty$

Les preuves pour H_1 et H_2 sont les suivantes :
- pour H_1 : $v(A_0) = -\frac{n}{n+2}$, donc $v(A_0) = -1$ quand n tend vers l'infini.
- pour H_2 : $v(A_0) = \frac{n}{n+2}$, donc $v(A_0) = 1$ quand n tend vers l'infini.

Et celles pour V_1 et V_2 reposent sur le calcul du point fixe d'une fonction :
- pour V_1 : posons $U_0 = v(A_n) = 0$ et $U_i = v(A_{n-i})$; on sait que $U_n = g(\alpha, \frac{U_{n-1}+1}{2}) = \frac{-1-U_{n-1}}{3+U_{n-1}}$; on a donc une suite qui converge vers le point fixe de la fonction $x \to \frac{-1-x}{3+x}$ que l'on obtient en résolvant l'équation $x = \frac{-1-x}{3+x}$; cette équation du second degré n'a qu'une racine appartenant à l'intervalle $[-1, 1]$: $\sqrt{3} - 2$.
- pour V_2 : posons $U_0 = v(A_n) = 0$ et $U_i = v(A_{n-i})$; on sait que $U_n = g(\frac{U_{n-1}+1}{2}, \alpha) = \frac{1+U_{n-1}}{3+U_{n-1}}$; on a donc une suite qui converge vers le point fixe de la fonction $x \to \frac{1+x}{3+x}$ que l'on obtient en résolvant l'équation $x = \frac{1+x}{3+x}$; cette équation du second degré n'a qu'une racine appartenant à l'intervalle $[-1, 1]$: $\sqrt{2} - 1$.

Remarquons que, pour toutes les instances, on obtient :

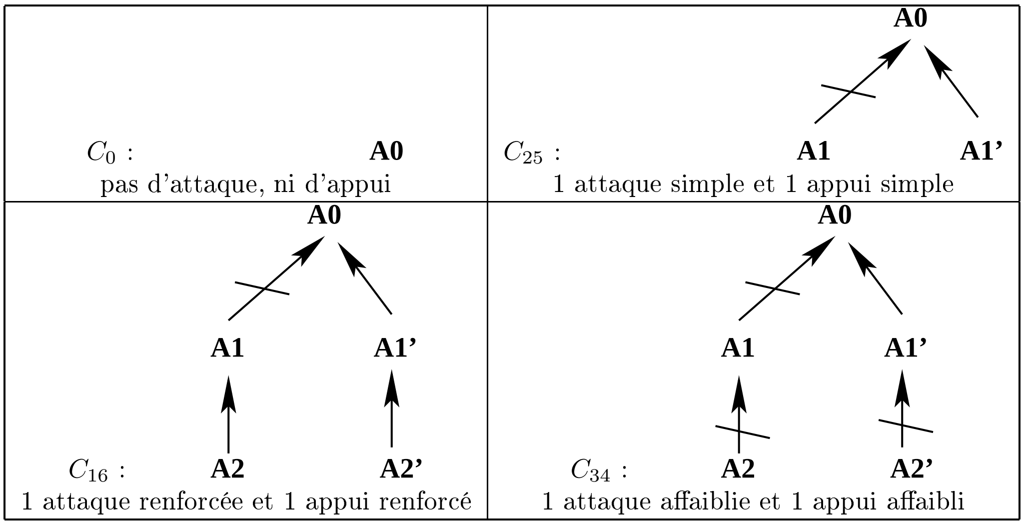
$$H_1 \leq V_1 \leq H_2 \text{ et } H_1 \leq V_1 \leq V_2$$

En revanche,

$$H_2 = V_2 \text{ pour l'instance 1, } H_2 \leq V_2 \text{ pour l'instance 2 et } V_2 \leq H_2 \text{ pour l'instance 3}$$

7.3.4.3 Les configurations d'équilibre

Dans cette section, on étudie les configurations menant à l'équilibre entre les attaques et les appuis, c'est-à-dire les configurations dans lesquelles la valeur de A_0 sera la même que celle d'un argument ne subissant ni attaque, ni appui. En utilisant les résultats présentés en section 7.3.4.1 page 41, on peut déjà identifier au moins 4 configurations d'équilibre :



On donne dans les tableaux suivants la valeur obtenue par A_0 dans chacune des 4 configurations d'équilibre.

Résultats avec l'instance 1 :

C_0	C_{25}	C_{16}	C_{34}
0	0	0	0

Résultats avec l'instance 2 :

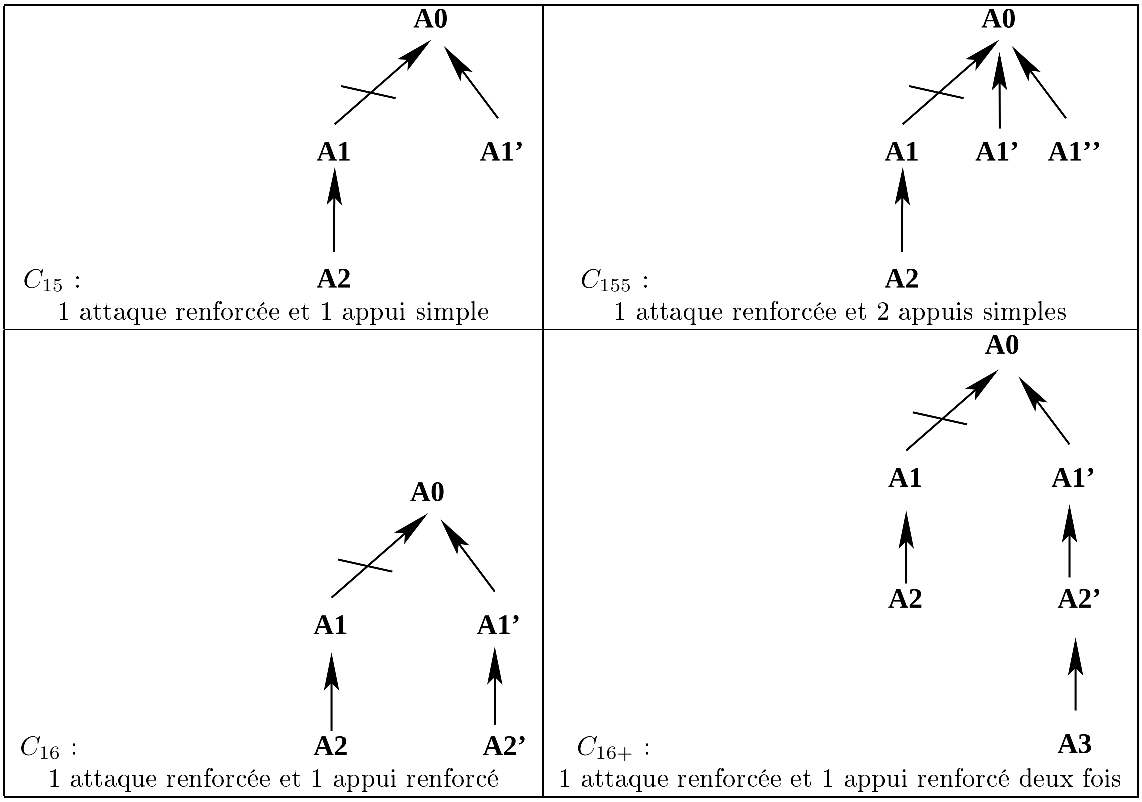
C_0	C_{25}	C_{16}	C_{34}
0	0	0	0

Résultats avec l'instance 3 :

C_0	C_{25}	C_{16}	C_{34}
0	0	0	0

Ces configurations d'équilibre étaient tout à fait évidentes à trouver de part leur "symétrie". La question qui se pose maintenant est la suivante : existe-t-il d'autres configurations menant à l'équilibre ?
Pour résoudre ce problème, il nous faut trouver les configurations de "basculement", c'est-à-dire celles dans lesquelles la valeur de A_0 passe de l'autre côté du point d'équilibre (de négative elle devient positive ou vice-versa). Pour cela, nous allons partir d'une configuration d'équilibre, la déséquilibrer en renforçant la partie attaque puis essayer de la rééquilibrer en renforçant la partie appui mais sans utiliser la symétrie exhibée dans les configurations d'équilibre déjà connues. La même méthode pourrait bien-sûr être proposée en partant des appuis puis en agissant sur les attaques, mais nous ne le ferons pas, car les résultats obtenus n'apporteraient rien de plus.
On va donc partir de la configuration C_{25} ($v(A_0) = 0$) et on va renforcer l'attaque, donc $v(A_0)$ va devenir négative (configuration C_{15}). Puis, on a deux possibilités :
- soit on rajoute des appuis simples en cherchant le moment où la valeur de A_0 va redevenir positive,
- soit on renforce l'appui en cherchant le moment où la valeur de A_0 va redevenir positive.

Le second cas est déjà en partie connu (c'est la configuration C_{16}) et on sait qu'il mène à l'équilibre. Par contre, il sera intéressant de voir l'impact de l'augmentation du renforcement suivant les différentes instances. Cela nous donne les 4 configurations suivantes :



On donne dans les tableaux suivants la valeur obtenue par A_0 dans chacune de ces configurations.

Résultats avec l'instance 1 :

C_{15}	C_{155}	C_{16}	C_{16+}
-1	-1	0	0

On constate ici que, quel que soit le nombre d'appuis simples que l'on rajoute, la valeur de A_0 reste inchangée. Cela est dû à l'utilisation de la fonction Max pour les fonctions h_{att} et h_{app}. Nous n'obtiendrons jamais l'équilibre par cette méthode.
D'autre part, quelle que soit l'augmentation du renforcement que nous ferons ici (donc quelle que soit la longueur de la branche d'appui), la valeur de A_0 reste inchangée aussi. Cela est dû à la valeur obtenue lors de la saturation verticale d'une branche d'appui (1, $\forall n$ la longueur de la branche). Ainsi dans tous les cas, dès qu'un appui est renforcé, il suffit à contrebalancer une attaque renforcée.

Résultats avec l'instance 2 :

C_{15}	C_{155}	C_{16}	C_{16+}
$-\frac{1}{4}$	$-\frac{1}{4}$	0	$\frac{1}{8}$

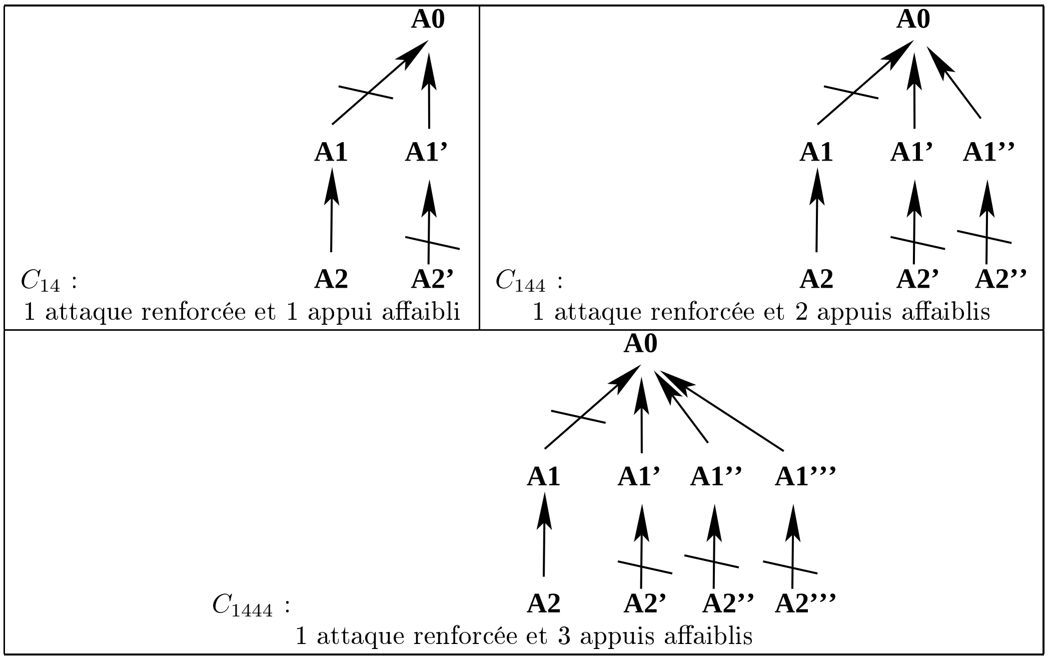
Nous avons ici la même constatation que pour l'instance 1. L'utilisation de la fonction Max pour les fonctions h_{att} et h_{app} fait que la valeur de A_0 reste inchangée quel que soit le nombre d'appuis simples que l'on rajoute. Nous n'obtiendrons jamais l'équilibre par cette méthode.
En ce qui concerne le renforcement des appuis, la valeur de A_0 va croître au fur et à mesure que ce renforcement augmente jusqu'à atteindre une limite imposée par la valeur de saturation verticale des appuis. Donc $v(A_0)$ tend vers $\frac{1}{4}$ quand la longueur de la branche d'appui tend vers l'infini.

Résultats avec l'instance 3 :

C_{15}	C_{155}	C_{16}	C_{16+}
$-\frac{1}{15}$	$\frac{1}{10}$	0	$\frac{1}{85}$

Ici, le basculement a eu lieu dès l'ajout du second appui. Et désormais plus on rajoutera d'appuis et plus la valeur de A_0 sera grande (jusqu'à atteindre une limite imposée par la valeur de saturation horizontale). Ainsi $v(A_0)$ tend vers $\frac{3}{5}$ quand le nombre d'appuis tend vers l'infini.
En ce qui concerne le renforcement des appuis, la valeur de A_0 va croître au fur et à mesure que ce renforcement augmente jusqu'à atteindre une limite imposée par la valeur de saturation verticale des appuis. Donc $v(A_0)$ tend vers $\frac{3\times\sqrt{2}-4}{5\times\sqrt{2}+5}$ ($\approx 0,02$) quand la longueur de la branche d'appui tend vers l'infini.

On obtient le même type de résultat en partant d'une configuration encore plus déséquilibrée :



C_{14} : 1 attaque renforcée et 1 appui affaibli

C_{144} : 1 attaque renforcée et 2 appuis affaiblis

C_{1444} : 1 attaque renforcée et 3 appuis affaiblis

On donne dans les tableaux suivants la valeur obtenue par A_0 dans chacune de ces 3 configurations.

Résultats avec l'instance 1 :

C_{14}	C_{144}	C_{1444}
-1	-1	-1

Toujours pour la même raison, la valeur de A_0 reste inchangée quel que soit le nombre d'appuis affaiblis que l'on rajoute. Nous n'obtiendrons jamais l'équilibre par cette méthode.

Résultats avec l'instance 2 :

C_{14}	C_{144}	C_{1444}
$-\frac{1}{2}$	$-\frac{1}{2}$	$-\frac{1}{2}$

Toujours pour la même raison, la valeur de A_0 reste inchangée quel que soit le nombre d'appuis affaiblis que l'on rajoute. Nous n'obtiendrons jamais l'équilibre par cette méthode.

Résultats avec l'instance 3 :

C_{14}	C_{144}	C_{1444}
$-\frac{3}{20}$	0	$\frac{1}{10}$

Ici, le point d'équilibre est atteint avec la configuration C_{144} puis la valeur de A_0 continue à augmenter au fur et à mesure que l'on rajoute des appuis affaiblis. Ainsi $v(A_0)$ tend vers $\frac{3}{5}$ quand le nombre d'appuis affaiblis tend vers l'infini[5]. Par contre, il a fallu un plus grand nombre d'appuis affaiblis pour provoquer le basculement que dans le cas précédent (configurations C_{15} et C_{155}), ce qui n'est pas surprenant puisque la valeur de chaque appui était plus faible.

[5] Remarquons que ce résultat est le même que les appuis rajoutés soient affaiblis ou pas. En effet, la valeur obtenue lors de la saturation horizontale d'appuis affaiblis pour l'instance 3 est exactement la même que celle obtenue lors d'une saturation horizontale d'appuis simples.

7.4 Application aux exemples de la section 3 page 7

Dans le cas de l'exemple d'orientation, on obtient $v(A_0) = 0.0137$.
Dans le cas de l'exemple de la prothèse, on obtient $v(A_0) = -0.1429$.
Dans le cas de l'exemple de l'enquête policière, on obtient :
- $v(A_0) = -0.3333$,
- $v(A_{0'}) = -0.1666$,
- $v(A_{0''}) = 0.0833$.

Dans ce dernier cas, on constate que $v(A_o) \leq v(A_{0'}) \leq v(A_{0''})$.

Chapitre 8

Une évaluation graduelle globale dans un SABP

8.1 Les principes à respecter

Soit v une évaluation graduelle globale dans un système d'argumentation bipolaire, v devrait respecter les principes suivants :

P0 Les valeurs proposées sont réparties sur un plan délimité par deux échelles (l'échelle "positive", l'échelle "négative") et le point de jonction de ces deux échelles est appelé *valeur neutre*.

P1 L'évaluation n'est neutre que pour un argument sans attaquant et sans appui[1].

P2 L'évaluation d'un argument prend en compte tous les chemins dont cet argument est racine.

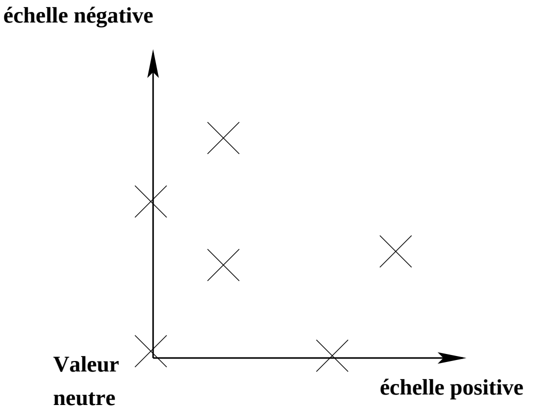
P3 L'amélioration de la défense, de l'appui ou la détérioration de l'attaque conduisent à augmenter la valeur d'un argument.

P4 L'amélioration de l'attaque ou la détérioration de la défense ou de l'appui conduisent à diminuer la valeur d'un argument.

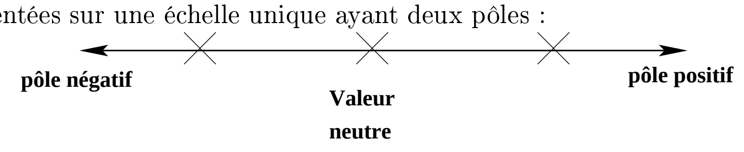
P5 La valeur d'un argument n'ayant que des appuis sera sur l'échelle "positive".

P6 La valeur d'un argument n'ayant que des attaques ou des défenses sera sur l'échelle "négative".

Le principe **P0** montre l'impact de la bipolarité du système d'argumentation (deux types d'interaction) sur l'ensemble des valeurs qui, par conséquent, sera lui-aussi bipolaire (deux "composantes" pour chaque valeur). On pourra représenter ces valeurs de la manière suivante (deux échelles unipolaires) :



À titre de comparaison, notons que toutes les instances d'évaluation graduelle locale vues au chapitre 7 page 35 peuvent être représentées sur une échelle unique ayant deux pôles :



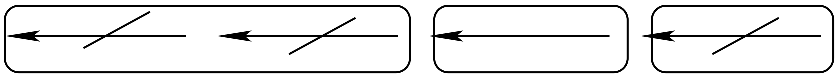
[1] Ce qui n'est pas toujours vrai dans le cas d'une évaluation graduelle locale (voir section 7 page 35).

Les principes **P1** à **P4** sont issus de ceux proposés dans [CLS01, CLS02, CLS03a] et modifiés pour prendre en compte la bipolarité[2].
Les principes **P5** à **P6** donnent la signification de la bipolarité en terme d'évaluation.
Suivant ces principes, il existe au moins 2 méthodes possibles pour définir une évaluation globale :

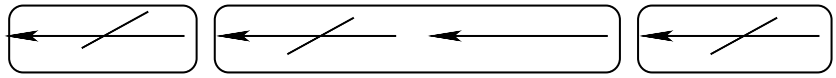
– La première méthode proposée en section 8.2 respecte les idées suivantes : chaque chemin entre une feuille et l'argument A va être découpé en partant de A en séquences d'attaque et séquences d'appui de longueur maximale ; chaque séquence est caractérisée par sa longueur et la première séquence (celle d'extrémité A) fixe la nature du chemin (séquence d'appui, d'attaque ou de défense). On privilégie ainsi la séquence la plus proche de l'argument au détriment des autres séquences qui ne servent qu'à qualifier la première. On obtient ainsi une évaluation qui pourrait être intermédiaire entre globale et locale.
– La seconde méthode proposée en annexe B page 79 respecte les idées suivantes : chaque chemin entre une feuille et l'argument A va être découpé en partant de A en attaques appuyées et séquences d'appui de longueur maximale. Dans ce cas, tout appui précédant une attaque dans le chemin est "intégré" à cette attaque (il augmente la qualité de l'attaque et "disparaît").

Un exemple pour fixer les idées :

Découpage selon la méthode 1 :



Découpage selon la méthode 2 :



Dans le premier cas, on a donc une attaque de longueur 2 (donc une défense au sens de [Dun95]) appuyée faiblement (puisque l'appui est attaqué).
Dans le second cas, on a une attaque attaquée par une attaque appuyée, cette attaque appuyée étant à son tour attaquée.
On voit bien que l'interprétation que l'on fait d'un chemin est essentielle dans l'évaluation de sa racine ; dans le premier cas, la racine subit une défense alors que dans le second on a un enchaînement d'un nombre impair d'attaques (donc la racine subit une attaque).
Dans ce chapitre, nous ne développerons que la première méthode d'évaluation globale (voir section 8.2), la seconde n'étant abordée que succinctement à titre de perspective (voir sections 8.4 page 65 et B page 79).

8.2 Une évaluation graduelle globale dans un SABP

8.2.1 Notion de branches

On va ici définir précisément le découpage que l'on effectue pour déterminer la nature de chacune des branches menant à un argument A donné.

Découpage Soit $\mathcal{C} = A_1 - \ldots - A_n$ un chemin d'une feuille vers A. On découpe $\mathcal{C}$ de la manière suivante :

1. on cherche A_{k_1} tel que $\forall i = k_1 \ldots n-1$, $A_i \mathcal{R}_{k_1} A_{i+1}$ et $\mathcal{R}_{k_1} = \mathcal{R}_{n-1}$ et soit $A_1 = A_{k_1}$, soit $A_{k_1-1} \mathcal{R}_x A_{k_1}$ avec $\mathcal{R}_x \neq \mathcal{R}_{n-1}$;
2. si $A_1 \neq A_{k_1}$, on cherche A_{k_2} tel que $\forall i = k_2 \ldots k_1 - 1$, $A_i \mathcal{R}_{k_2} A_{i+1}$ et $\mathcal{R}_{k_2} \neq \mathcal{R}_{n-1}$ et soit $A_1 = A_{k_2}$, soit $A_{k_2-1} \mathcal{R}_x A_{k_2}$ avec $\mathcal{R}_x = \mathcal{R}_{n-1}$;
3. si $A_1 \neq A_{k_2}$, on réitère le processus 1-2-3 jusqu'à atteindre un $A_{k_j} = A_1$.

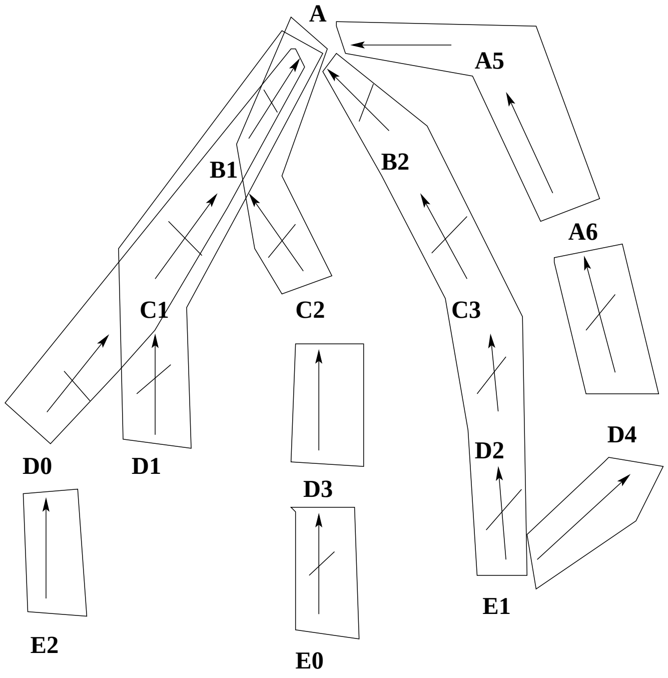
Autre formalisation de ce découpage Soit $\mathcal{C} = A_1 - \ldots - A_n$ un chemin d'une feuille vers A. On découpe $\mathcal{C}$ de la manière suivante :

1. On pose $k_1 = \mathrm{Min}\{1 \leq i \leq n-2 / \forall j = 1 \ldots n-1, \mathcal{R}_i = \mathcal{R}_j\}$ si cet ensemble est non vide, et $n-1$ sinon,

[2]Les notions d'amélioration/détérioration sont à définir en fonction des notions d'attaque, défense et appui utilisées.

2. ...
3. On pose $k_{i+1} = \text{Min}\{1 \leq i \leq k_i - 2/\forall j = 1 \ldots k_i - 1, \mathcal{R}_i = \mathcal{R}_j\}$ si cet ensemble est non vide, et $k_i - 1$ sinon,
4. on s'arrête lorsque $k_i = 2$.

Exemple de découpage



Définition 33 (Branches d'appui, d'attaque et de défense d'un argument) *Soit $A \in \mathcal{A}$.*

Branche d'appui *Un chemin $\mathcal{C}$ dans $\mathcal{G}$ d'une feuille vers A est une* branche d'appui *pour A ssi il existe au moins un sous-chemin de $\mathcal{C}$ qui soit une séquence d'appui pour A*[3]*.*

On dira alors que A est racine d'une branche d'appui.

La longueur du plus long de ces sous-chemins sera appelée la longueur utile *de la branche d'appui et notée l_{app}.*

Branche d'attaque *Un chemin $\mathcal{C}$ dans $\mathcal{G}$ d'une feuille vers A est une* branche d'attaque *pour A ssi il existe au moins un sous-chemin de $\mathcal{C}$ qui soit une séquence d'attaque pour A et le plus long de ces sous-chemins est de longueur impaire*[4]*.*

On dira alors que A est racine d'une branche d'attaque.

La longueur de ce sous-chemin sera appelée la longueur utile *de la branche d'attaque et notée l_{att}.*

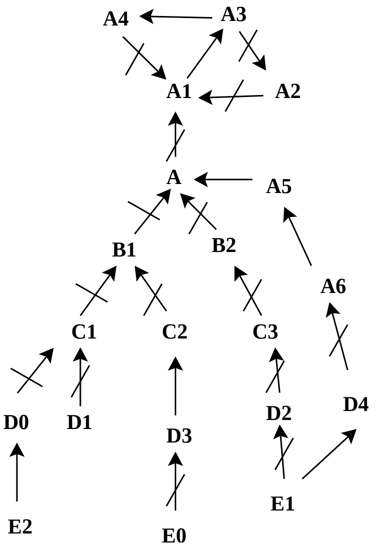
Branche de défense *Un chemin $\mathcal{C}$ dans $\mathcal{G}$ d'une feuille vers A est une* branche de défense *pour A ssi il existe au moins un sous-chemin de $\mathcal{C}$ qui soit une séquence d'attaque pour A et le plus long de ces sous-chemins est de longueur paire*[5]*.*

On dira alors que A est racine d'une branche de défense.

La longueur de ce sous-chemin sera appelée la longueur utile *de la branche de défense et notée l_{def}.*

Toutes ces notions sont illustrées sur l'exemple suivant :

[3] La première séquence obtenue par le découpage est une séquence d'appuis.
[4] La première séquence obtenue par le découpage est une séquence d'attaques de longueur impaire.
[5] La première séquence obtenue par le découpage est une séquence de défense.



Sur ce graphe $\mathcal{G}$, on a donc :
- 2 branches d'attaque pour A :
  $D_1 - C_1 - B_1 - A$ de longueur utile 3 et
  $E_2 - D_0 - C_1 - B_1 - A$ aussi de longueur utile 3,
- 2 branches de défense pour A :
  $E_0 - D_3 - C_2 - B_1 - A$ de longueur utile 2 et
  $E_1 - D_2 - C_3 - B_2 - A$ de longueur utile 4,
- 1 branche d'appui pour A :
  $E_1 - D_4 - A_6 - A_5 - A$ de longueur utile 2.

8.2.2 Définition d'une évaluation graduelle globale dans un SABP

Soit le système d'argumentation bipolaire $<\mathcal{A}, \mathcal{R}_{att}, \mathcal{R}_{app}>$ *sans cycle*[6]. Nous commençons par préciser la notion de suite de bits pour définir ce qu'est la valeur d'une branche.

8.2.2.1 Valeur d'une branche et suites de bits

Définition 34 (Suite de bits) *La suite $x_1 \ldots x_n$ est une suite de bits ssi $\forall i = 1 \ldots n$, soit $x_i = 0$, soit $x_i = 1$.*
La suite de bits vide est notée $-$.

Notons $\mathcal{S}$ l'ensemble des suites de bits et $\mathcal{T}$ l'ensemble des tuples de suites de bits.

Définition 35 (Valeur d'une branche dans un SABP) *Soit une branche $\mathcal{C} = A_1 - \ldots - A_n$ dans le SABP $<\mathcal{A}, \mathcal{R}_{att}, \mathcal{R}_{app}>$. La* valeur de la branche *(notée $v_\mathcal{C}$) est la suite de bits $x_1 \ldots x_{n-1}$ vérifiant la contrainte : $\forall i = 1 \ldots n-1$,*
- $x_i = 0$ si $A_i \mathcal{R}_{app} A_{i+1}$,
- $x_i = 1$ si $A_i \mathcal{R}_{att} A_{i+1}$.

La valeur d'une branche est donc un élément de $\mathcal{S}$.

Point de vue algorithmique Pour implémenter les éléments de $\mathcal{S}$, nous avons plusieurs choix. Le premier consiste à utiliser des listes de bits et le second à considérer qu'un élément de $\mathcal{S}$ est le codage en base 2 d'un entier (par exemple la valeur de la branche $A_1 \not\to A_2 \to A_3 \to A_4 \not\to A_5 \to A_6$ est 10010 ce qui correspond à l'entier 18).

La première possibilité est intéressante car très proche de la définition 35 mais présente l'inconvénient de la manipulation de listes de taille variable (donc des structures de données complexes à manipuler du point de vue algorithmique). Nous avons implémenté cette solution avec le langage CAMLLIGHT en annexe A page 69. Dans ce cas, nous avons les conséquences suivantes :

Conséquence 3 (Lien entre le type de la branche et sa valeur) *S'il s'agit d'une branche d'appui, le bit le plus à droite dans la liste est* 0.
S'il s'agit d'une branche d'attaque ou de défense, le bit le plus à droite dans la liste est 1.

Conséquence 4 (Calcul de la longueur utile d'une branche)
Soit $\mathcal{C}$ une branche de valeur $x_1 \ldots x_p$, la longueur utile de $\mathcal{C}$ est n le nombre de x_i, pour $1 \leq i \leq p$, vérifiant : $x_i = x_p$ et $\forall j, i \leq j \leq p$, $x_j = x_p$.
Si $\mathcal{C}$ est une branche d'appui alors $l_{app} = n$.
Si $\mathcal{C}$ est une branche d'attaque alors n est impair et $l_{att} = n$.
Si $\mathcal{C}$ est une branche de défense alors n est pair et $l_{def} = n$.

[6] L'absence de cycle facilitera l'identification des branches menant à l'argument que l'on souhaite évaluer.

Des exemples :

– La branche de valeur 10010 est une branche d'appui et sa longueur utile est 1 ($1001\underbrace{0}$).
– La branche de valeur 11000 est une branche d'appui et sa longueur utile est 3 ($11\underbrace{000}$).
– La branche de valeur 0000 est une branche d'appui et sa longueur utile est 4 ($\underbrace{0000}$).
– La branche de valeur 10011 est une branche d'attaque ou de défense. Sa longueur utile est paire et égale à 2 ($100\underbrace{11}$), donc il s'agit d'une branche de défense.
– La branche de valeur 0001 est une branche d'attaque ou de défense. Sa longueur utile est impaire et égale à 1 ($000\underbrace{1}$), donc il s'agit d'une branche d'attaque.
– La branche de valeur 11111 est une branche d'attaque ou de défense. Sa longueur utile est impaire et égale à 5 ($\underbrace{11111}$), donc il s'agit d'une branche d'attaque.

La seconde possibilité, plus compacte du point de vue des données enregistrées et manipulées, présente un inconvénient majeur : ce codage (suite de bits $\rightarrow$ entier) n'est pas une bijection (un entier peut correspondre à plusieurs suites : 12 peut être codé par 1100 mais aussi par 01100, ou par 001100, ...). Il faut donc se donner un moyen pour traiter les 0 qui sont en début de suite (par exemple en rajoutant artificiellement un 1 au début de la suite de bits – voir conséquence 6).

Conséquence 5 (Lien entre le type de la branche et sa valeur) *S'il s'agit d'une branche d'appui, la valeur de la branche correspond à un entier pair.*
S'il s'agit d'une branche d'attaque ou de défense, la valeur de la branche correspond à un entier impair.

Conséquence 6 (Calcul de la longueur utile d'une branche d'appui) *Soit $\mathcal{C}$ une branche d'appui ayant pour valeur entière $v_\mathcal{C}$, $l_{app} = n$, n étant le plus grand entier tel que $\exists X$ un entier et $X \times 2^n = v_\mathcal{C}$.*
Cas particulier : celui d'une branche d'appui donc la valeur correspond à l'entier 0[7]. Dans ce cas, avant de calculer la longueur utile, on rajoutera artificiellement un 1 au début de la suite de bits et on fera le calcul de la longueur utile sur la nouvelle suite.

Donc, la valeur d'une branche d'appui est un multiple d'une puissance de 2 (la plus grande possible) et cette puissance est la longueur utile de cette branche.

Conséquence 7 (Calcul de la longueur utile d'une branche d'attaque) *Soit $\mathcal{C}$ une branche d'attaque ayant pour valeur entière $v_\mathcal{C}$, $l_{att} = 2k+1$, k étant le plus grand entier ≥ 0 tel que $\exists X$ un entier et $X \times 2^{2k+1} = v_\mathcal{C} + 1$.*

Conséquence 8 (Calcul de la longueur utile d'une branche de défense) *Soit $\mathcal{C}$ une branche de défense ayant pour valeur entière $v_\mathcal{C}$, $l_{def} = 2k$, k étant le plus grand entier ≥ 1 tel que $\exists X$ un entier et $X \times 2^{2k} = v_\mathcal{C} + 1$.*

Donc, la valeur d'une branche d'attaque ou de défense à laquelle on a ajouté 1 est un multiple d'une puissance de 2 (la plus grande possible) et cette puissance est la longueur utile de cette branche qui est donc une branche d'attaque quand cette puissance est impaire ou une branche de défense quand cette puissance est paire.

Des exemples :

– La branche de valeur 10010 correspondant à l'entier 18 est une branche d'appui. La longueur utile est alors 1 car 18 est multiple de $2 = 2^1$ mais pas de 4, ni de 8, ni de 16.
– La branche de valeur 11000 correspondant à l'entier 24 est une branche d'appui. La longueur utile est alors 3 car 24 est multiple de 2, de 4 et de $8 = 2^3$, mais pas de 16.
– La branche de valeur 0000 correspondant à l'entier 0 est une branche d'appui. On rajoute artificiellement un 1 en début de suite et on obtient 10000 correspondant à l'entier 16. La longueur utile est 4 car 16 est multiple de 2, de 4, de 8 et de $16 = 2^4$.
– La branche de valeur 10011 correspondant à l'entier 19 est une branche d'attaque ou de défense. Rajoutons 1 à 19, cela donne 20 et cherchons de quelle puissance de 2 est multiple 20 : 20 est multiple de 2, de $4 = 2^2$, mais ni de 8, ni de 16. Donc, la plus grande puissance étant paire, la branche est donc une branche de défense avec une longueur utile égale à 2.

[7]Cet entier 0 n'est pas suffisant pour distinguer par exemple entre les suites 00000 et 00.

- La branche de valeur 0001 correspondant à l'entier 1 est une branche d'attaque ou de défense. Rajoutons 1 à 1, cela donne 2 et cherchons de quelle puissance de 2 est multiple 2 : 2 est multiple de $2 = 2^1$. Donc, la plus grande puissance étant impaire, la branche est donc une branche d'attaque avec une longueur utile égale à 1.
- La branche de valeur 11111 correspondant à l'entier 31 est une branche d'attaque ou de défense. Rajoutons 1 à 31, cela donne 32 et cherchons de quelle puissance de 2 est multiple 32 : 32 est multiple de 2, de 4, de 8, de 16 et de $32 = 2^5$. Donc, la plus grande puissance étant impaire, la branche est donc une branche d'attaque avec une longueur utile égale à 5.

8.2.2.2 Valeur d'un argument et valeurs tuplées bipolaires

Nous allons utiliser les valeurs des branches pour définir celles des arguments.

Définition 36 (Valeur d'un argument dans un SABP) *Soit un argument $A \in \mathcal{A}$. La valeur de A, notée $v(A)$, est un triplet de tuples, chaque tuple représentant respectivement la valeur de l'appui, la valeur de l'attaque et la valeur de la défense que subit l'argument :*

$$v(A) = [v_{app}, v_{att}, v_{def}]$$

avec :

$$\begin{aligned} v_{app} &= (v_{App_1}, \ldots, v_{App_n}) \textit{ si } A \textit{ est racine des branches d'appui } App_1, \ldots, App_n \\ &\qquad \textit{ayant pour valeurs respectives } v_{App_i}, \forall i = 1 \ldots n \\ &= (-) \textit{ sinon} \end{aligned}$$

$$\begin{aligned} v_{att} &= (v_{Att_1}, \ldots, v_{Att_m}) \textit{ si } A \textit{ est racine des branches d'attaque } Att_1, \ldots, Att_m \\ &\qquad \textit{ayant pour valeurs respectives } v_{Att_i}, \forall i = 1 \ldots n \\ &= (-) \textit{ sinon} \end{aligned}$$

$$\begin{aligned} v_{def} &= (v_{Def_1}, \ldots, v_{Def_p}) \textit{ si } A \textit{ est racine des branches de défense } Def_1, \ldots, Def_p \\ &\qquad \textit{ayant pour valeurs respectives } v_{Def_i}, \forall i = 1 \ldots n \\ &= (-) \textit{ sinon} \end{aligned}$$

On dira que $v(A)$ est une valeur tuplée bipolaire[8]*.*

Remarquons que, contrairement à ce qui avait été fait dans [CLS01, CLS02, CLS03a], il n'y a pas ici de tuples vides mais des suites vides représentant l'absence de branche.
Notons $\mathcal{V}$ la partie de $\mathcal{T} \times \mathcal{T} \times \mathcal{T}$ représentant l'ensemble des valeurs tuplées bipolaires (donc $\forall v \in \mathcal{V}$, v est une valeur tuplée bipolaire respectant la définition 36).
La définition précédente permet de calculer directement la valeur d'un argument en fonction de la partie du graphe dont cet argument est racine. Nous allons proposer aussi une méthode de calcul de cette valeur en fonction de la valeur des contrariants directs et des appuis directs de l'argument. Pour cela, nous nous inspirons de ce que nous avions fait dans [CLS03a] en définissant quelques opérations particulières (une sur les éléments de $\mathcal{S}$ et deux sur ceux de $\mathcal{T}$) :

Définition 37 *L'ajout d'un bit à une suite de bits est définie par l'application $\oplus_s : \mathcal{S} \times \{0,1\} \to \mathcal{S}$ telle que*

$$\begin{aligned} - \oplus_s k &= k \\ x_1 \ldots x_n \oplus_s k &= x_1 \ldots x_n k \end{aligned}$$

Définition 38 *Il y a deux opérations sur les tuples :*

[8] Bien qu'il y ait en fait 3 tuples !

– *concaténation entre deux tuples définie par l'application* $\star\ : \mathcal{T} \times \mathcal{T} \rightarrow \mathcal{T}$ *telle que*

$$\begin{array}{rcl} t \star (-) & = & t \\ (s_1, \ldots, s_n) \star (s'_1, \ldots, s'_n) & = & (s_1, \ldots, s_n, s'_1, \ldots, s'_n) \end{array}$$

– *addition entre un tuple et un bit définie par l'application* $\oplus_t\ : \mathcal{T} \times \{0, 1\} \rightarrow \mathcal{T}$ *telle que*

$$\begin{array}{rcl} (-) \oplus_t k & = & (k) \\ (s_1, \ldots, s_n) \oplus_t k & = & (s_1 \oplus_s k, \ldots, s_n \oplus_s k) \end{array}$$

Conséquence 9 (Calcul de la valeur d'un argument)
Soit un argument A apparaissant dans le graphe des interactions bipolaire, donc :

– *pouvant avoir pour attaquants directs B_1, …, B_n, chaque B_i ayant pour valeur $v(B_i) = [v_{app}(B_i)$, $v_{att}(B_i)$, $v_{def}(B_i)]$*
– *et pouvant avoir pour appuis directs C_1, …, C_m, chaque C_i ayant pour valeur $v(C_i) = [v_{app}(C_i)$, $v_{att}(C_i)$, $v_{def}(C_i)]$*

La valeur de A est le triplet de tuples suivant :

$$v(A) = [v_{app}, v_{att}, v_{def}]$$

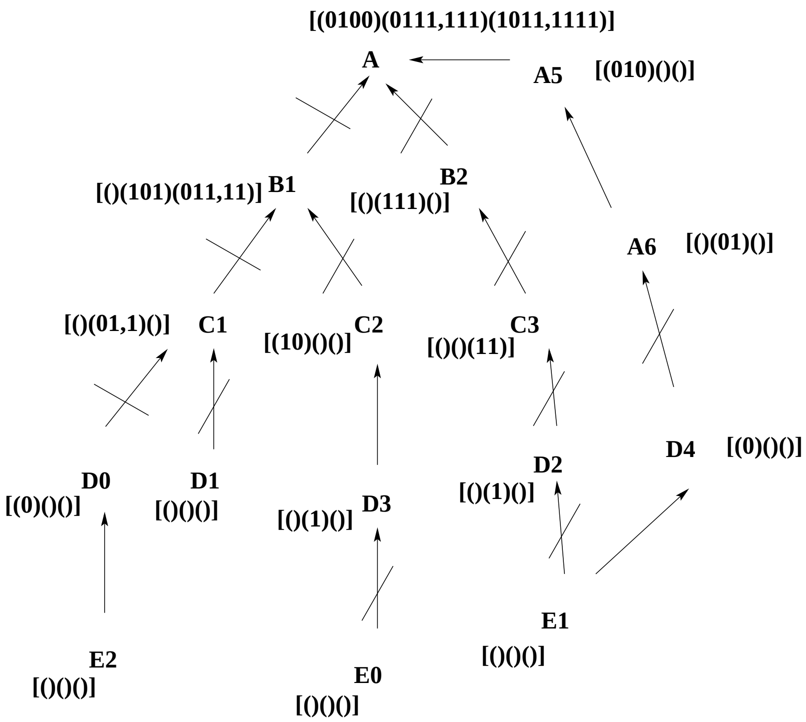
avec :

$$\begin{array}{rcll} v_{app} & = & \left[\begin{array}{l} (v_{app}(C_1) \oplus_t 0) \star \ldots \star (v_{app}(C_m) \oplus_t 0) \star \\ (v_{att}(C_1) \oplus_t 0) \star \ldots \star (v_{att}(C_m) \oplus_t 0) \star \\ (v_{def}(C_1) \oplus_t 0) \star \ldots \star (v_{def}(C_m) \oplus_t 0) \end{array} \right. & \textit{si les } C_i \textit{ existent} \\ & = & (-) \qquad\qquad \textit{sinon} & \end{array}$$

$$\begin{array}{rcll} v_{att} & = & \left[\begin{array}{l} (v_{def}(B_1) \oplus_t 1) \star \ldots \star (v_{def}(B_n) \oplus_t 1) \star \\ (v_{app}(B_1) \oplus_t 1) \star \ldots \star (v_{app}(B_n) \oplus_t 1) \end{array} \right. & \textit{si les } B_i \textit{ existent} \\ & = & (-) \qquad\qquad \textit{sinon} & \end{array}$$

$$\begin{array}{rcll} v_{def} & = & (v_{att}(B_1) \oplus_t 1) \star \ldots \star (v_{att}(B_n) \oplus_t 1) & \textit{si les } B_i \textit{ existent} \\ & = & (-) \qquad\qquad \textit{sinon} & \end{array}$$

Un exemple Appliquons cette évaluation globale à l'exemple suivant :



8.2.3 Comparaison d'arguments

Pour comparer des arguments (donc des valeurs tuplées bipolaires), nous souhaitons exploiter le même type d'algorithme que celui proposé dans [CLS01, CLS02, CLS03a], mais en l'étendant au cas de valeurs tuplées bipolaires.
Dans un premier temps, nous comparons donc les nombres de branches ayant la même nature. Si cette comparaison d'ordre quantitatif n'est pas suffisamment discriminante (car les arguments ont autant de branches d'appui, autant de branches d'attaque et autant de branches de défense), nous étudions alors les tuples 2 à 2 suivant leur nature (tuple d'appui avec tuple d'appui, et idem pour ceux représentant les attaques et pour ceux représentant les défenses). Cette comparaison est alors d'ordre qualitatif et implique que l'on dispose d'un ordre de type lexicographique issu d'un ordre sur des valeurs de branches quelle que soit la nature des dites branches.
C'est pourquoi dans la section 8.2.3.1, nous introduisons cet ordre portant sur des valeurs de branches, donc des suites de bits.
Puis en section 8.2.3.2 page 59, nous donnerons l'ordre lexicographique issu de l'ordre sur les suites de bits.
Et enfin en section 8.2.3.3 page 59, nous donnerons l'algorithme de comparaison d'arguments.

8.2.3.1 Comparaison de suites de bits

Pour comparer deux suites de bits, nous allons comparer leur longueur utile et donc utiliser les conséquences 6 page 53, 7 page 53 et 8 page 53 qui montrent comment calculer ces longueurs.
Remarquons tout de suite qu'on ne compare deux à deux que des suites ayant la même signification (deux suites d'appui, ou deux suites d'attaque, ou deux suites de défense).
L'idée est ici de conclure en fonction de la longueur utile des suites. Il existe donc des cas où, les longueurs étant égales, nous ne pourrons pas conclure immédiatement. Il faudra alors réitérer le processus mais sur des suites privées de leur partie utile. Cela revient à considérer que la comparaison va se faire de la droite vers la gauche en travaillant chaque fois sur les parties des suites ayant la même signification. Par exemple, pour comparer les suites d'appui 1001100 et 0000100, on identifie les différentes parties constituant les suites ($\underbrace{1}\underbrace{00}\underbrace{11}\underbrace{00}$ et $\underbrace{0000}\underbrace{1}\underbrace{00}$) ; ensuite, on va d'abord comparer 00 avec 00, puis 11 avec 1, et ainsi de suite jusqu'à pouvoir conclure.

Comparaison de suites de bits représentant des appuis L'algorithme 1 page ci-contre permet de comparer deux suites de bits quand ces suites sont des suites d'appui. S'il ne peut conclure en étudiant les parties “appui” les plus à droite de chaque suite, alors il appellera l'algorithme 2 pour continuer à étudier les parties suivantes de chaque suite, qui sont des parties “attaque-défense”.

Algorithme 1: COMPARER-APPUI(s_1, s_2)

```
% s1 et s2 deux suites de bits représentant des appuis
% l1 = lapp(s1) et l2 = lapp(s2) leurs longueurs utiles
% renvoie :
%        0 si s1 = s2
%        1 si les appuis représentés par s1 sont meilleurs que ceux représentés par s2
%                (donc la suite s1 “appuie” mieux un argument que la suite s2),
%        -1 sinon
début
    si s1 = s2 alors retourner 0
    si l1 > l2 alors retourner 1
    si l1 < l2 alors retourner-1
    sinon
        s'1 = décaler s1 de l1 bits sur la droite
        s'2 = décaler s2 de l1 bits sur la droite
        res = COMPARER-ATTAQUE-DÉFENSE(s'1, s'2)
        retourner res
fin
```

Comparaison de suites de bits représentant des attaques ou des défenses L'algorithme 2 permet de comparer deux suites de bits quand ces suites sont des suites d'attaque ou de défense. Et, suivant le cas (attaque ou défense), il appellera les algorithmes 3 page suivante ou 4 page suivante. Si ces algorithmes ne peuvent conclure en étudiant les parties “attaque” ou “défense” les plus à droite de chaque suite, alors ils appelleront l'algorithme 1 pour continuer à étudier les parties suivantes de chaque suite, qui sont des parties “appui”.

Algorithme 2: COMPARER-ATTAQUE-DÉFENSE(s_1, s_2)

```
% s1 et s2 deux suites de bits représentant des attaques ou des défenses
% l1 = lapp(s1) et l2 = lapp(s2) leurs longueurs utiles
% renvoie :
%        0 si s1 = s2 ,
%        1 si s1 représente une défense et s2 représente une attaque
%          OU (si s1 et s2 sont de même nature
%                ET (si les défenses représentées par s1 sont meilleures que celles représentées par s2
%                      (donc la suite s1 “défend” mieux un argument que la suite s2),
%                   OU les attaques représentées par s1 sont moins bonnes que celles représentées par s2
%                      (donc la suite s1 “attaque” moins bien un argument que la suite s2))),
%        -1 sinon
début
    si s1 = s2 alors retourner 0
    si l1 pair et l2 pair alors
        res = COMPARER-DÉFENSE(s1, s2)
        retourner res
    sinon
        si l1 impair et l2 impair alors
            res = COMPARER-ATTAQUE(s1, s2)
            retourner res
        sinon
            si l1 impair et l2 pair alors retourner-1
            si l1 pair et l2 impair alors retourner 1
fin
```

Algorithme 3: COMPARER-ATTAQUE(s_1, s_2)

% s_1 et s_2 deux suites de bits représentant des attaques
% $l_1 = l_{app}(s_1)$ et $l_2 = l_{app}(s_2)$ leurs longueurs utiles
% renvoie :
% 0 si $s_1 = s_2$,
% 1 si les attaques représentées par s_1 sont moins bonnes que celles représentées par s_2,
% (donc la suite s_1 "attaque" moins bien un argument que la suite s_2),
% -1 sinon

début
- **si** $l_1 > l_2$ **alors retourner** 1
- **si** $l_1 < l_2$ **alors retourner**-1
- **sinon**
  - s'_1 = décaler s_1 de l_1 bits sur la droite
  - s'_2 = décaler s_2 de l_1 bits sur la droite
  - res = COMPARER-APPUI(s'_1, s'_2)
  - **retourner** (-res)

fin

Algorithme 4: COMPARER-DÉFENSE(s_1, s_2)

% s_1 et s_2 deux suites de bits représentant des défenses
% $l_1 = l_{app}(s_1)$ et $l_2 = l_{app}(s_2)$ leurs longueurs utiles
% renvoie :
% 0 si $s_1 = s_2$,
% 1 si les défenses représentées par s_1 sont meilleures que celles représentées par s_2,
% (donc la suite s_1 "défend" mieux un argument que la suite s_2),
% -1 sinon

début
- **si** $l_1 > l_2$ **alors retourner**-1
- **si** $l_1 < l_2$ **alors retourner** 1
- **sinon**
  - s'_1 = décaler s_1 de l_1 bits sur la droite
  - s'_2 = décaler s_2 de l_1 bits sur la droite
  - res = COMPARER-APPUI(s'_1, s'_2)
  - **retourner** res

fin

Pré-ordre sur $\mathcal{S}$ On définit les relations $\prec$ et $\preceq$ sur $\mathcal{S}$ de la manière suivante :

Définition 39 (Relations $\prec$, $=$ et $\preceq$ sur $\mathcal{S}$) *Soit s_1 et s_2 deux suites de bits, les relations $\prec$, $=$ sur $\mathcal{S}$ se définissent par :*
- *si s_1 et s_2 ne sont pas de même nature avec s_1 représentant un appui et s_2 représentant une attaque ou une défense alors $s_2 \prec s_1$,*
- *si s_1 et s_2 ne sont pas de même nature avec s_1 représentant une défense et s_2 représentant une attaque alors $s_2 \prec s_1$,*
- *si s_1 et s_2 sont des valeurs représentant un appui, et alors*
  - *$s_2 \prec s_1$ ssi* COMPARER-APPUI*(s_1, s_2) = 1*
  - *$s_1 \prec s_2$ ssi* COMPARER-APPUI*(s_1, s_2) = -1*
  - *$s_1 = s_2$ ssi* COMPARER-APPUI*(s_1, s_2) = 0*
- *si s_1 et s_2 sont des valeurs représentant une défense, et alors*
  - *$s_2 \prec s_1$ ssi* COMPARER-ATTAQUE-DÉFENSE*(s_1, s_2) = 1*
  - *$s_1 \prec s_2$ ssi* COMPARER-ATTAQUE-DÉFENSE*(s_1, s_2) = -1*
  - *$s_1 = s_2$ ssi* COMPARER-ATTAQUE-DÉFENSE*(s_1, s_2) = 0*
- *si s_1 et s_2 sont des valeurs représentant une attaque, et alors*
  - *$s_1 \prec s_2$ ssi* COMPARER-ATTAQUE-DÉFENSE*(s_1, s_2) = 1*
  - *$s_2 \prec s_1$ ssi* COMPARER-ATTAQUE-DÉFENSE*(s_1, s_2) = -1*
  - *$s_1 = s_2$ ssi* COMPARER-ATTAQUE-DÉFENSE*(s_1, s_2) = 0*

Soit s_1 et s_2 deux suites de bits, la relation $\preceq$ sur $\mathcal{S}$ se définit par : $s_1 \preceq s_2$ ssi $s_1 \prec s_2$ ou $s_1 = s_2$.

Propriété 15 *La relation $\preceq$ sur $\mathcal{S}$ donnée par la définition 39 page ci-contre est un pré-ordre total.*

8.2.3.2 Comparaison de tuples de suites de bits

Les relations $\prec$ et $=$ sur $\mathcal{S}$ données par la définition 39 page précédente vont servir à définir un ordre lexicographique sur $\mathcal{T}$:

Définition 40 (Relation lexicographique sur les tuples de suites de bits)
Soit $(x_1, \ldots, x_n)$ et $(y_1, \ldots, y_m)$ 2 tuples finis de suites de bits $\in \mathcal{T}$. Soit $\prec$ et $=$ les relations sur $\mathcal{S}$ données par la définition 39 page ci-contre.
On dira que $(x_1, \ldots, x_n) <_{lex} (y_1, \ldots, y_m)$ ssi $\exists m \geq i \geq 1$ tel que :
- *$\forall j < i$, $x_j = y_j$ et*
- *soit $n = i - 1$[9], soit $x_i \prec y_i$.*

De même, on dira que $(x_1, \ldots, x_n) =_{lex} (y_1, \ldots, y_m)$ ssi $n = m$ et $\forall i$, $1 \leq i \leq n$, $x_i = y_i$.
Donc $(x_1, \ldots, x_n) \leq_{lex} (y_1, \ldots, y_m)$ ssi
$$(x_1, \ldots, x_n) =_{lex} (y_1, \ldots, y_m) \text{ ou } (x_1, \ldots, x_n) <_{lex} (y_1, \ldots, y_m).$$

Propriété 16 *La relation $\leq_{lex}$ sur $\mathcal{T}$ donnée par la définition 40 est un pré-ordre total.*

8.2.3.3 Comparaison de valeurs tuplées bipolaires

La comparaison entre valeurs tuplées bipolaires va suivre les mêmes principes que ceux utilisés pour comparer deux valeurs tuplées dans [CLS01, CLS02, CLS03a]. Dans l'algorithme 5, on va donc avoir une première phase de comparaison quantitative durant laquelle on compare le nombre de branches de chaque nature (appui, attaque et défense) menant aux arguments. Puis, si la première phase ne permet pas de conclure, on appelle l'algorithme 6 page suivante pour une seconde phase de comparaison qualitative durant laquelle on comparera la qualité des branches de chaque nature menant aux arguments.

Algorithme 5: COMPARER-VALEURS-TUPLÉES-BIPOLAIRES(v_1, v_2)

```
% v1 = [vapp1, vatt1, vdef1] et v2 = [vapp2, vatt2, vdef2] deux valeurs tuplées bipolaires différentes %
% renvoie 1 si v1 ≻ v2, -1 si v1 ≺ v2, 0 si v1 et v2 sont incomparables %
début
    nbapp1 = |vapp1|
    nbapp2 = |vapp2|
    nbatt1 = |vatt1|
    nbatt2 = |vatt2|
    nbdef1 = |vdef1|
    nbdef2 = |vdef2|
    si nbapp1 = nbapp2 ET nbatt1 = nbatt2 ET nbdef1 = nbdef2 alors
        res = COMPARER-VALEURS-TUPLÉES-BIPOLAIRES-QUALITATIVEMENT(v1,v2)      /* Cas 1 */
        retourner res
    si nbapp1 ≥ nbapp2 ET nbatt1 ≤ nbatt2 ET nbdef1 ≥ nbdef2 alors retourner 1      /* Cas 2 */
    si nbapp1 ≤ nbapp2 ET nbatt1 ≥ nbatt2 ET nbdef1 ≤ nbdef2 alors retourner-1      /* Cas 3 */
    sinon retourner 0                                                              /* Cas 4 */
fin
```

Lors de la phase de comparaison qualitative, nous utiliserons l'ordre lexicographique sur $\mathcal{T}$ donné par la définition 40.

[9] Donc x_i n'existe pas, alors que y_i existe et on a donc $n < m$.

Algorithme 6: COMPARER-VALEURS-TUPLÉES-BIPOLAIRES-QUALITATIVEMENT(v_1, v_2)

% $v_1 = [v_{app}1, v_{att}1, v_{def}1]$ et $v_2 = [v_{app}2, v_{att}2, v_{def}2]$ deux valeurs tuplées bipolaires différentes %
% renvoie 1 si $v_1 \succ v_2$, -1 si $v_1 \prec v_2$, 0 si v_1 et v_2 sont incomparables %
% la fonction $fc(t)$ sert à classer par ordre croissant les valeurs données dans le tuple t %
% en utilisant l'ordre $\prec$ défini sur $\mathcal{S}$ (voir section 8.2.3.1 page 56) %

début
$v_{app}1' = fc(v_{app}1)$
$v_{app}2' = fc(v_{app}2)$
$v_{att}1' = fc(v_{att}1)$
$v_{att}2' = fc(v_{att}2)$
$v_{def}1' = fc(v_{def}1)$
$v_{def}2' = fc(v_{def}2)$
si $v_{app}1' \geq_{lex} v_{app}2'$ *ET* $v_{def}1' \geq_{lex} v_{def}2'$ *ET* $v_{att}1' \leq_{lex} v_{att}2'$ **alors retourner** 1 /* Cas 1.1 */
si $v_{app}1' \leq_{lex} v_{app}2'$ *ET* $v_{def}1' \leq_{lex} v_{def}2'$ *ET* $v_{att}1' \geq_{lex} v_{att}2'$ **alors retourner**-1 /* Cas 1.2 */
sinon retourner 0 /* Cas 1.3 */
fin

8.2.3.4 Pré-ordre sur $\mathcal{V}$

À l'aide de ce qui précède, on définit :

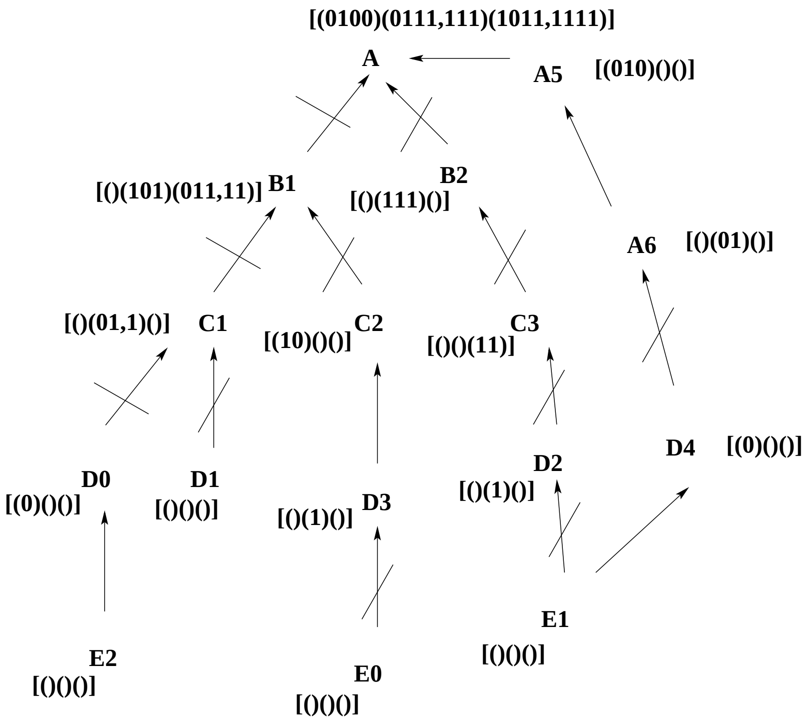
Définition 41 (Relations $\prec$, $\preceq$, $=$ sur $\mathcal{V}$) *Soit v_1 et v_2 deux valeurs tuplées bipolaires, les relations $\prec$ et $=$ sont définies par :*
- *si v_1 et v_2 sont identiques alors $v_1 = v_2$,*
- *si v_1 et v_2 sont différentes alors :*
  - *$v_2 \prec v_1$ ssi* COMPARER-VALEURS-TUPLÉES-BIPOLAIRES*(v_1, v_2) = 1,*
  - *$v_1 \prec v_2$ ssi* COMPARER-VALEURS-TUPLÉES-BIPOLAIRES*(v_1, v_2) = -1,*
  - *v_1 et v_2 sont incomparables ssi* COMPARER-VALEURS-TUPLÉES-BIPOLAIRES*(v_1, v_2) = 0,*

Soit v_1 et v_2 deux valeurs tuplées bipolaires, la relation $\preceq$ est définie par : $v_1 \preceq v_2$ ssi $v_1 \prec v_2$ ou $v_1 = v_2$.

Propriété 17 *La relation $\preceq$ sur $\mathcal{V}$ donnée par la définition 41 est un pré-ordre.*

8.2.3.5 Un exemple

Reprenons l'exemple suivant et étudions l'ordre entre A et ses prédécesseurs directs :



On obtient les résultats suivants :

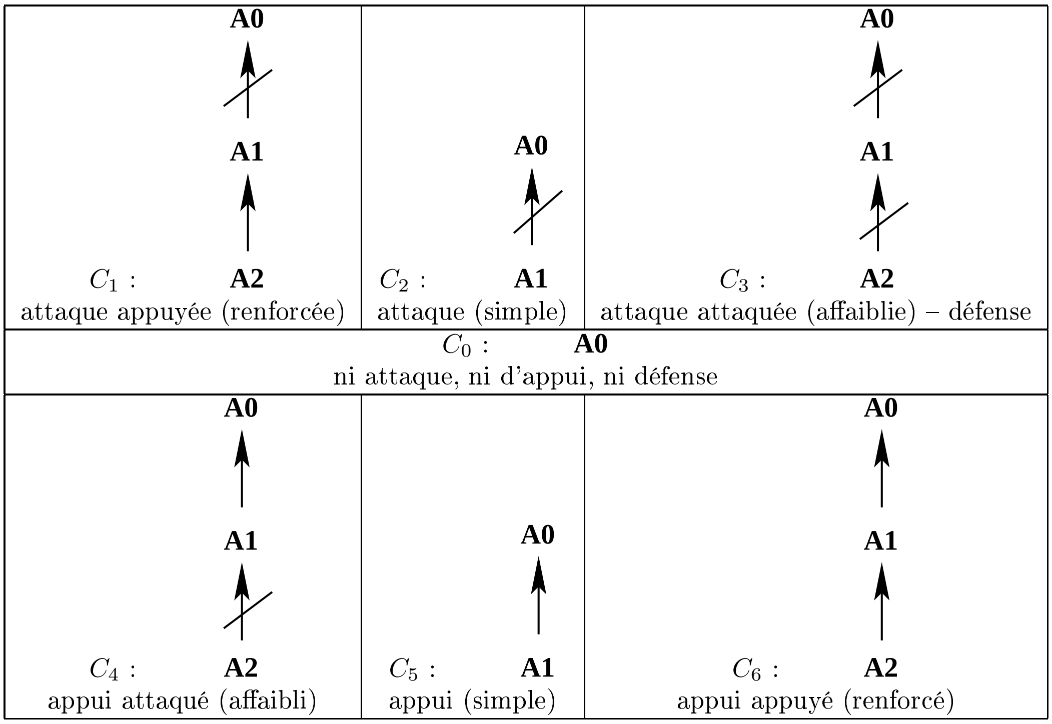
$$\begin{array}{rcl} B_1 & \succ & B_2 \\ A_5 & \succ & B_2 \\ B_1 \text{ et } A_5 & \text{sont} & \text{incomparables} \\ A \text{ et } B_1 & \text{sont} & \text{incomparables} \\ A \text{ et } B_2 & \text{sont} & \text{incomparables} \\ A \text{ et } A_5 & \text{sont} & \text{incomparables} \end{array}$$

8.2.4 Analyse comparative

Ce travail a été réalisé en collaboration avec Dominique GAY, lors de son stage de DEA (voir [Gay04][10]). Il s'agit d'analyser le comportement de l'évaluation globale bipolaire proposée ici.
Comme pour l'évaluation locale, cette analyse va se faire suivant plusieurs axes en utilisant les mêmes configurations de test que celles utilisées en section 7.3.4 page 41 :
- d'abord sur des configurations de base (voir section 8.2.4.1),
- puis sur des configurations de saturation en appui ou en attaque (voir section 8.2.4.2 page suivante),
- et enfin sur des cas d'équilibre, c'est-à-dire qu'on va rechercher les configurations dans lesquelles la valeur de l'argument se trouve exactement à l'équilibre entre les attaques et les appuis (voir section 8.2.4.3 page 63). Un cas particulier sera alors la recherche des mécanismes permettant à une valeur de "basculer" autour de ce point d'équilibre. Attention, nous aurons ici une notion de point d'équilibre totalement différente de celle utilisée en section 7.3.4 page 41.

8.2.4.1 Les configurations de base

Rappelons les 7 configurations de base notées C_0 à C_6 représentant les cas les plus simples d'interaction :



D'après les principes donnés en section 8.1 page 49, on doit avoir :

$$C_1 \leq C_2 \leq C_0 \leq C_3 \text{ et } C_1 \leq C_2 \leq C_0 \leq C_4 \leq C_5 \leq C_6$$

[10]Toutefois, dans son rapport, Dominique GAY n'étudie pas exactement l'évaluation globale définie ici. Il en définit une autre qui ne prend en compte que les attaques et les appuis (les défenses étant considérées comme des attaques affaiblies).

(c'est-à-dire que la valeur de A_0 dans C_1 doit être plus faible que celle de A_0 dans C_2 ... qui doit être plus faible que celle de A_0 dans C_6)
Remarquons que les principes proposés ne permettent pas de comparer C_3 (1 branche de défense) avec C_4, C_5 et C_6 (1 branche d'appui). Remarquons aussi que la configuration C_0 (ni attaque, ni appui, ni défense) se trouve par les principes choisis moins bonne que la configuration C_3.
On donne dans le tableau suivant la valeur obtenue par A_0 dans chacune des 7 configurations de base.

C_1	C_2	C_3	C_0	C_4	C_5	C_6
[()(01)()]	[()(1)()]	[()()(11)]	[()()()]	[(10)()()]	[(0)()()]	[(00)()()]

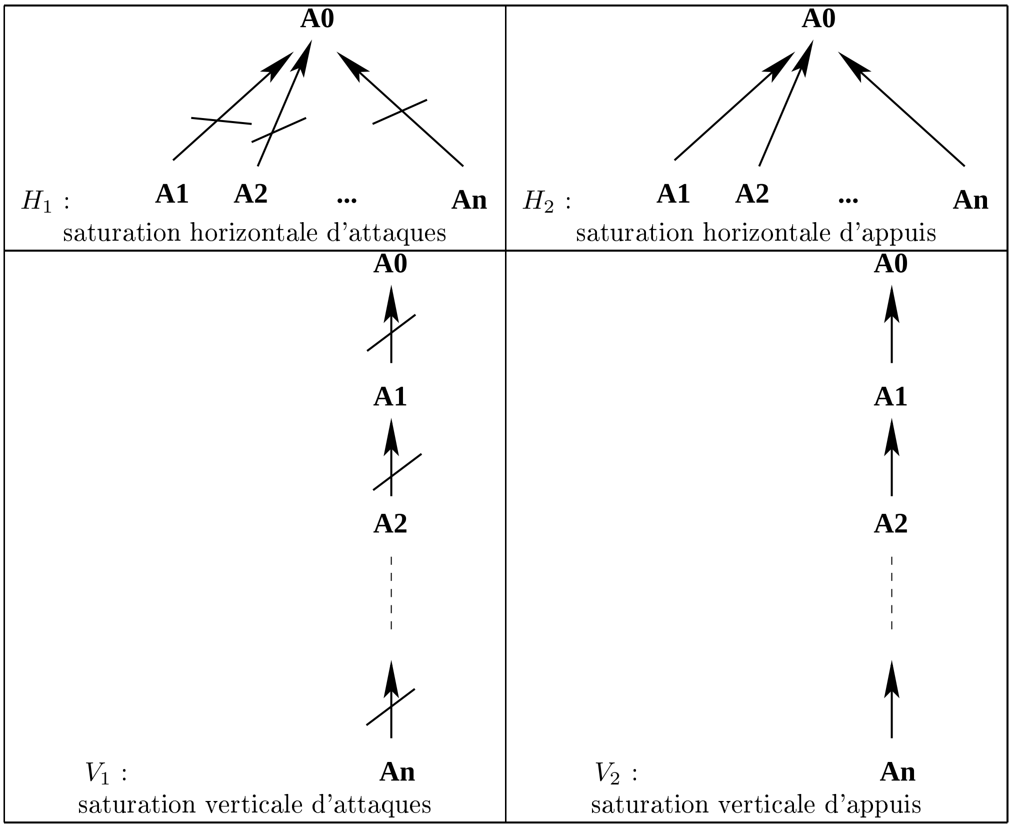
Les principes sont bien respectés puisqu'on obtient :

$$C_1 \leq C_2 \leq C_0 \leq C_3 \text{ et } C_1 \leq C_2 \leq C_0 \leq C_4 \leq C_5 \leq C_6$$

D'autre part, contrairement au cas de l'évaluation locale, les résultats ne permettent pas d'identifier ici de "symétrie" entre les configurations.

8.2.4.2 Les configurations de saturation

Ici, on va étudier l'effet d'une saturation d'attaques ou d'appuis sur la valeur des arguments. Rappelons donc les configurations correspondant à des saturations horizontales et verticales :



Les principes énoncés en section 8.1 page 49 permettent de dire que $H_1 \leq H_2$, $H_1 \leq V_2$. D'autre part, suivant la parité de n dans le cas de la saturation verticale d'attaque, on a :
- si n est impair (donc la branche est une branche d'attaque), $V_1 \leq H_2$ et $V_1 \leq V_2$; par contre, les principes proposés ne permettent pas alors de comparer V_1 avec H_1 et V_2 avec H_2.
- si n est pair (donc la branche est une branche de défense), $H_1 \leq V_1$; par contre, les principes proposés ne permettent pas alors de comparer V_1 avec H_2, V_1 avec V_2 et V_2 avec H_2.

On donne dans le tableau suivant la valeur obtenue par A_0 dans chacune des 4 configurations de saturation.

H_1	H_2	V_1	V_2
$[()(\underbrace{1,\ldots,1}_{n \text{ fois}})()]$	$[(\underbrace{0,\ldots,0}_{n \text{ fois}})()()]$	$[()(\underbrace{1\ldots1}_{n \text{ fois}})()]$ si n est impair et $[()()(\underbrace{1\ldots1}_{n \text{ fois}})]$ si n est pair	$[(\underbrace{0\ldots0}_{n \text{ fois}})()()]$

Remarquons qu'on obtient dans tous les cas :

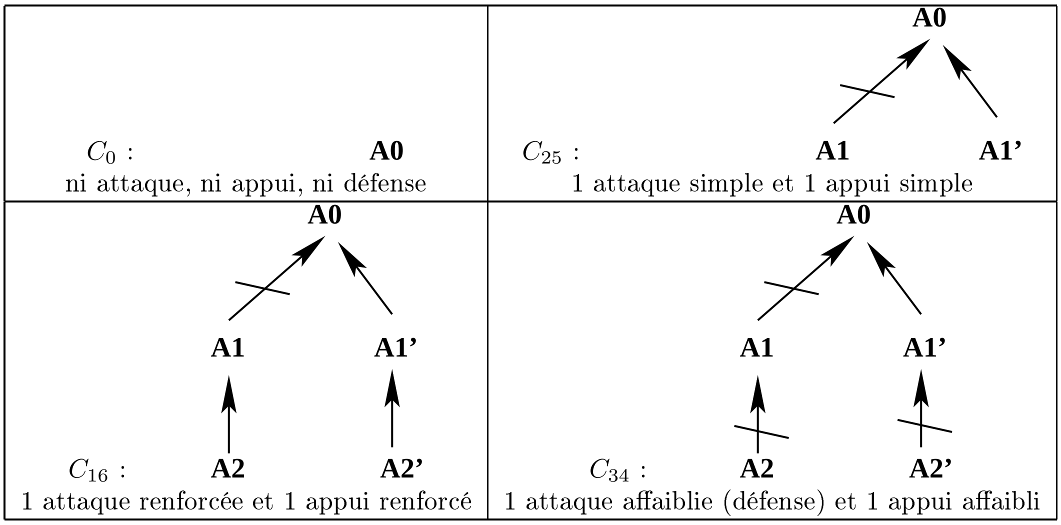
$$H_1 \leq V_1 \text{ et } H_1 \leq V_2 \leq H_2$$

D'autre part, si n est impair dans la saturation d'attaque, on a un ordre total :

$$H_1 \leq V_1 \leq V_2 \leq H_2$$

8.2.4.3 Les configurations d'équilibre

Dans cette section, on étudie les configurations menant à l'équilibre entre les attaques et les appuis. Cette notion d'équilibre est ici différente de celle utilisée en section 7.3.4 page 41.
En effet, si on suppose que la notion d'équilibre se rattache à la valeur "neutre" évoquée dans les principes donnés en section 8.1 page 49, alors les seuls arguments pouvant avoir cette valeur "neutre" sont ceux qui ne sont ni attaqués, ni appuyés. Cela correspond uniquement à la configuration C_0.
Toutefois, afin d'obtenir un maximum d'éléments de comparaison entre les évaluations locale et globale, nous allons quand même étudier les 4 configurations proposées en section 7.3.4 page 41 :



On donne dans le tableau suivant la valeur obtenue par A_0 dans chacune des 4 configurations précédentes.

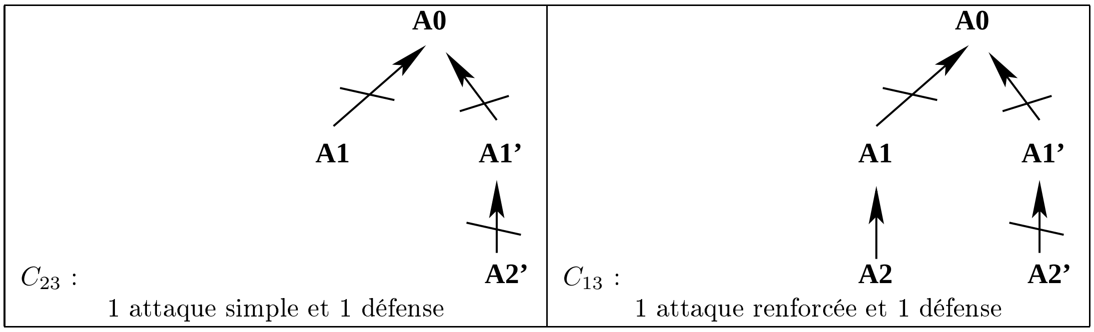
C_0	C_{25}	C_{16}	C_{34}
[()()()]	[(0)(1)()]	[(00)(01)()]	[(10)()(11)]

on obtient alors :

$$C_0 \leq C_{34} \text{ et } C_{25} \leq C_{34} \text{ et } C_{16} \leq C_{34}$$

C_{34} est la meilleur configuration, mais toutes les autres sont incomparables deux à deux.

Une autre idée pour déterminer cette notion d'équilibre serait de définir une configuration d'équilibre comme une configuration dans laquelle le nombre de branches d'attaque = le nombre de branches de défense + le nombre de branches d'appui. Ainsi, les configurations C_0, C_{25} et C_{16} seraient des configurations d'équilibre mais plus C_{34}, et on aurait en plus, au moins deux autres configurations d'équilibre à prendre en compte[11] :



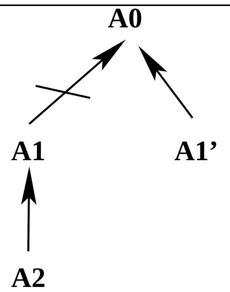
[11] Notons que ces configurations ne sont pas des configurations d'équilibre pour les évaluations locales. Par exemple, avec l'instance 3, on aurait $v(A_0) = -\frac{13}{25}$ pour C_{23} et $v(A_0) = -\frac{1}{2}$ pour C_{13}.

On donne dans le tableau suivant la valeur obtenue par A_0 dans chacune de 2 nouvelles configurations d'équilibre.

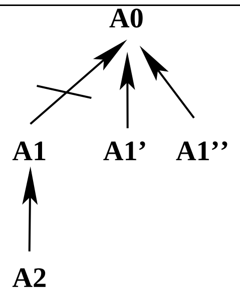
C_{23}	C_{13}
$[()(1)(11)]$	$[()(01)(11)]$

On obtient $C_{13} \leq C_{23}$.
Ici aussi, comme lors de l'étude de l'évaluation locale, on peut trouver des configurations de "basculement" en partant de la configuration C_{25}, en renforçant l'attaque par un appui (configuration C_{15}), puis :
- soit en rajoutant des appuis simples,
- soit en renforçant l'appui (de la configuration C_{16}, qui peut être vue comme une configuration d'équilibre, à des configurations dans lesquelles on renforce encore plus l'appui),
- soit en rajoutant des défenses.

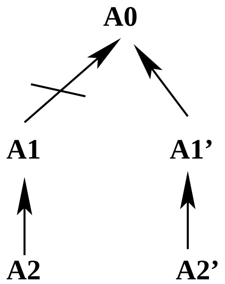
Les configurations utilisées sont les suivantes :



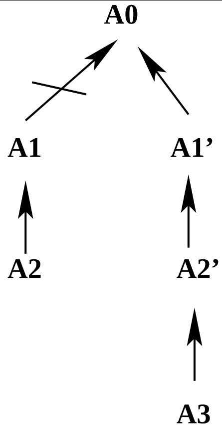
C_{15} :
1 attaque renforcée et 1 appui simple



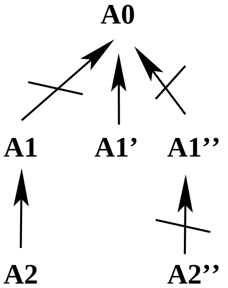
C_{155} :
1 attaque renforcée et 2 appuis simples



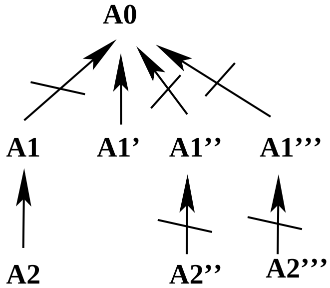
C_{16} :
1 attaque renforcée et 1 appui renforcé



C_{16+} :
1 attaque renforcée et 1 appui renforcé deux fois



C_{153} :
1 attaque renforcée, 1 appui, 1 défense



C_{1533} :
1 attaque renforcée, 1 appui, 2 défenses

On donne dans le tableau suivant la valeur obtenue par A_0 dans chacune de ces configurations.

C_{15}	C_{155}	C_{16}	C_{16+}	C_{153}	C_{1533}
$[(0)(01)()]$	$[(0,0)(01)()]$	$[(00)(01)()]$	$[(000)(01)()]$	$[(0)(01)(11)]$	$[(0)(01)(11,11)]$

On constate ici que :

$$C_{15} \leq C_{16} \leq C_{16+} \leq C_{155} \text{ et } C_{15} \leq C_{16} \leq C_{16+} \leq C_{153} \leq C_{1533}$$

(C_{155} restant incomparable à C_{153} et C_{1533})
Et si on s'en tient à la notion d'équilibre construite à partir du nombre de branches, alors seules les configurations C_{15}, C_{16}, C_{16+} sont des configurations d'équilibre.
D'autre part, du point de vue du "basculement", on constate que :

$$C_{15} \leq C_{25} \leq C_{155} \text{ et } C_{25} \leq C_{153} \leq C_{1533}$$

Il semblerait donc bien que l'on ait détérioré la valeur de A_0 avec la configuration C_{15}, puis qu'on l'ait ensuite améliorée suffisamment pour passer au delà du point d'équilibre C_{25}, sauf dans le cas des configurations C_{16} et C_{16+} qui sont incomparables avec C_{25}.

8.3 Application aux exemples de la section 3 page 7

Dans le cas de l'exemple d'orientation, on obtient $v(A_0) = [(00, 00, 00, 10)(111, 1)(11, 11)]$ (4 appuis, 2 attaques et 2 défenses).
Dans le cas de l'exemple de la prothèse, on obtient $v(A_0) = [(110, 0)(01)(11, 11)]$ (2 appuis, 1 attaque et 2 défenses).
Dans le cas de l'exemple de l'enquête policière, on obtient :
- $v(A_0) = [(0)(1, 1, 1, 1)()]$ (1 appui et 4 attaques),
- $v(A_{0'}) = [(0)(1, 1)()]$ (1 appui et 2 attaques),
- $v(A_{0''}) = [(0)()(11)]$ (1 appui et 1 défense).

Dans ce dernier cas, on constate que $v(A_o) \leq v(A_{0'}) \leq v(A_{0''})$ (ce qui correspond aussi au résultat fourni par l'évaluation locale).

8.4 Perspectives pour les évaluations graduelles globales dans un SABP

Il existe plusieurs possibilités d'évolution de l'évaluation graduelle globale.
La première consiste tout simplement à étudier la seconde méthode évoquée en section 8.1 page 49 et développée très succinctement en annexe B page 79.
Il y a aussi beaucoup d'autres idées :

Idée 1 : la première idée vient du fait que bien qu'il y ait *deux* relations entre arguments, il y a *trois* types d'informations prises en compte dans l'évaluation globale proposée en section 8.2 page 50 : l'attaque, la défense et l'appui. Cela crée un déséquilibre entre les deux relations. Il y a au moins deux moyens de résoudre ce problème :

1. supprimer la notion de défense. C'est ce qu'a proposé Dominique GAY dans son rapport de stage (voir [Gay04]). Toutefois, il est difficile de mener cette idée à son terme car il y a une différence sémantique importante entre une séquence de deux attaques ($C \not\to B \not\to A$) et une séquence de trois attaques ($D \not\to C \not\to B \not\to A$). Dans le premier cas, A doit avoir une valeur plus forte que son attaquant direct B, et dans le second cas A doit avoir une valeur plus faible que son attaquant direct B. Cette manière de traiter les séquences d'attaques est justifiée par de très nombreux exemples qui s'appuient tous sur la notion de "réinstallation" : A est "réinstallé" par l'attaque par C de son attaquant B si C est lui-même justifié (soit parce qu'il n'est pas attaqué, soit parce qu'il est aussi réinstallé).
2. introduire une notion supplémentaire au niveau de l'appui pour faire le "pendant" de la défense par rapport à l'attaque (par exemple une notion d'"appui affaibli"). Ce qui permettrait de ramener la notion de défense à une notion d'"attaque affaiblie". On aurait alors les 4 notions suivantes :
   - appui affaibli ($\not\to\to$),
   - appui renforcé ($\to\to$),
   - attaque affaiblie ($\not\to\not\to$) (anciennement défense),
   - attaque renforcée ($\to\not\to$).

Idée 2 : travailler avec un méta-graphe dans lequel il ne restera plus que des méta-arguments et des arcs d'attaque. On obtiendrait ce méta-graphe de la manière suivante :

$$A \rightarrow B \not\rightarrow C \not\rightarrow D \rightarrow E \rightarrow F$$

deviendrait

$$B' \not\rightarrow C \not\rightarrow F'$$

avec $B' = A \oplus B$ et $F' = D \oplus E \oplus F$, l'opération $\oplus$ restant bien-sûr à définir.

Dans ce cas, les appuis sont vus comme générateurs d'une valeur intrinsèque du méta-argument (donc, on prend en compte le nombre d'appuis comme un "poids" du méta-argument).

Par contre, les arcs d'attaque sont inchangés.

On peut donc réutiliser tout ce qui a été fait dans le cadre uni-polaire mais avec des poids sur les arguments.

Intérêt immédiat : cette idée aide à préciser la nature des branches (et tend d'ailleurs vers l'approche globale décrite en section B page 79).

Idée 3 : lien avec les ATMS (les ATMS ne représenteraient-ils pas un système ne contenant que les appuis ?).

Chapitre 9

Conclusion et perspectives

Nous inspirant des travaux de [KP01, Ver02], nous nous sommes attachées à introduire la notion de *bipolarité* dans le cadre de l'argumentation sous la forme de la prise en compte de *deux types d'interactions entre arguments* :
- une relation d'attaque,
- une relation d'appui.

Pour cela, nous avons étendu le modèle proposé par [Dun95] et défini les *Systèmes d'Argumentation BiPolaires* (SABP).
Puis, dans ce nouveau cadre, nous avons étendu nos travaux précédents (voir par exemple [CLS03c]) et proposé :
- de nouvelles sémantiques pour l'acceptabilité,
- de nouvelles évaluations graduelles d'arguments (une évaluation locale générique pour laquelle nous avons identifié et étudié 3 trois instances, et une évaluation globale).

Il reste encore beaucoup de choses à faire :
- étudier les nouvelles sémantiques pour l'acceptabilité du point de vue de leurs propriétés, de leurs capacités calculatoires, et les comparer avec les sémantiques déjà existantes.
- étudier, comparer et calculer de manière efficace les différentes évaluations graduelles.
- prendre en compte les cycles.

Annexe A

Implémentation d'une évaluation globale bipolaire en CAMLLIGHT

Il s'agit de l'implémentation des algorithmes de comparaison utilisant pour représenter les éléments de $\mathcal{S}$ des listes de bits. Nous avons toutefois prévu des fonctions de conversion pour passer de la représentation des éléments de $\mathcal{S}$ par liste de bits à la représentation par des entiers, et vice-versa.

```
(*****************************************
 **** FICHIER compar-att-app.ml
 **** de fonctions pour comparer des
 **** valeurs tuplees bipolaires
 ******************************************)
(*****************************************
 ** Notations :
 **  b : bit
 **  s : suite de bits
 **  n : entier representant une suite de bits
 **  si : suite de bits inversee (pour travailler les
 **       suites de bits plus facilement avec le type liste)
 **  t : tuple de suites de bits
 **  tn : tuple d'entiers
 **  tapp : tuple d'appui
 **  tatt : tuple d'attaque
 **  tdef : tuple de defense
 **  v : valeur tuplee bipolaire (un triplet de tuples) utilisant les suites de bits
 **  vn : valeur tuplee bipolaire (un triplet de tuples) utilisant les entiers
 ******************************************)

set_print_length 10000;;
set_print_depth 10000;;


(*****************************************
****** TYPES INTERNES utilises
******************************************)
type Bit == int;;
type SuiteBits == Bit list;;
type SuiteBitsInv == SuiteBits;; (* pour faciliter la manipulation des listes
                                    les bits les + a droite d'une SuiteBits
                                    sont les + a gauche d'une SuiteBitsInv *)
type Tuple == SuiteBits list;;
type TupleInt == int list;;

(*****************************************
****** TYPES EXTERNES utilises
******************************************)
type ValTupleBipo == Tuple * Tuple * Tuple;;
type ValTupleIntBipo == TupleInt * TupleInt * TupleInt;;
```

```
(******************************************
****** FONCTIONS INTERNES utilisees
****** diverses
*******************************************)
(*===================== fonction =============================
 * NOM : champ1 (resp. champ2, champ3)
 * BUT : acces au 1er (resp. 2eme, 3eme) element d'un triplet
 * TYPE DE RETOUR : type de l'element du triplet accede
 * PARAMETRES : un triplet
 *========================================================*)
let champ1 ((x1 : 'a), (x2 : 'b), (x3 : 'c)) = x1;;
let champ2 ((x1 : 'a), (x2 : 'b), (x3 : 'c)) = x2;;
let champ3 ((x1 : 'a), (x2 : 'b), (x3 : 'c)) = x3;;

(champ1 (1,2,3) : 'a) = 1;;
(champ2 (1,2,3) : 'b) = 2;;
(champ3 (1,2,3) : 'c) = 3;;


(******************************************
****** FONCTIONS INTERNES utilisees
****** pour le type SUITEBITS
*******************************************)
(*===================== fonction =============================
 * NOM : Conversion_SuiteBits_Int
 * BUT : conversion d'une suite de bits en un entier
 * TYPE DE RETOUR : int
 * PARAMETRES : s SuiteBits
 *========================================================*)
let rec Conversion_SuiteBits_Int_Aux ((s : SuiteBits), (acc : int)) =
    if s = [] then acc
    else Conversion_SuiteBits_Int_Aux ((tl s), acc * 2 + (hd s));;

let Conversion_SuiteBits_Int ((s : SuiteBits)) =
    Conversion_SuiteBits_Int_Aux( s, 0);;

(Conversion_SuiteBits_Int [1;0;0;1;0] : int) = 18;;
(Conversion_SuiteBits_Int [] : int) = 0;;
(Conversion_SuiteBits_Int [1;0;0;1] : int) = 9;;
(Conversion_SuiteBits_Int [1] : int) = 1;;
(Conversion_SuiteBits_Int [0] : int) = 0;;


(*===================== fonction =============================
 * NOM : Conversion_Int_SuiteBits
 * BUT : conversion d'un entier en une suite de bits
 * TYPE DE RETOUR : SuiteBits
 * PARAMETRES : n int
 *========================================================*)
let rec Conversion_Int_SuiteBits ((n : int))=
    if n = 0 then []
    else Conversion_Int_SuiteBits (n/2) @ [n mod 2];;

(Conversion_Int_SuiteBits (18) : SuiteBits) = [1;0;0;1;0];;
(Conversion_Int_SuiteBits (0) : SuiteBits) = [];;
(Conversion_Int_SuiteBits (9) : SuiteBits) = [1;0;0;1];;
(Conversion_Int_SuiteBits (1) : SuiteBits) = [1];;


(*===================== fonction =============================
 * NOM : Inv_SuiteBits
 * BUT : inverser une suite de bits
 * TYPE DE RETOUR : SuiteBitsInv
 * PARAMETRES : s SuiteBits
 *========================================================*)
let Inv_SuiteBits ((s : SuiteBits)) =
    rev s;;

(Inv_SuiteBits [0;1;1;0;0;1] : SuiteBitsInv) = [1;0;0;1;1;0];;
(Inv_SuiteBits [] : SuiteBitsInv) = [];;
(Inv_SuiteBits [1] : SuiteBitsInv) = [1];;
```

```
(******************************************
****** FONCTIONS INTERNES utilisees
****** pour le type SUITEBITSINV
******************************************)
(*===================== fonction ==============================
 * NOM : Conversion_SuiteBitsInv_Int
 * BUT : conversion d'une suite de bits inversee en un entier
 * TYPE DE RETOUR : int
 * PARAMETRES : si SuiteBitsInv
 *==============================================================*)
let rec Conversion_SuiteBitsInv_Int ((si : SuiteBitsInv)) =
    if si = [] then 0
    else (hd si) + (2 * Conversion_SuiteBitsInv_Int(tl si));;

(Conversion_SuiteBitsInv_Int (Inv_SuiteBits [1;0;0;1;0]) : int) = 18;;
(Conversion_SuiteBitsInv_Int (Inv_SuiteBits []) : int) = 0;;
(Conversion_SuiteBitsInv_Int (Inv_SuiteBits [1;0;0;1]) : int) = 9;;
(Conversion_SuiteBitsInv_Int (Inv_SuiteBits [1]) : int) = 1;;
(Conversion_SuiteBitsInv_Int (Inv_SuiteBits [0]) : int) = 0;;


(*===================== fonction ==============================
 * NOM : Conversion_Int_SuiteBitsInv
 * BUT : conversion d'un entier en une suite de bits inversee
 * TYPE DE RETOUR : SuiteBitsInv
 * PARAMETRES : n int
 *==============================================================*)
let rec Conversion_Int_SuiteBitsInv ((n : int))=
    if n = 0 then []
    else (n mod 2) :: Conversion_Int_SuiteBitsInv (n/2);;

(Conversion_Int_SuiteBitsInv (18) : SuiteBitsInv) = Inv_SuiteBits [1;0;0;1;0];;
(Conversion_Int_SuiteBitsInv (0) : SuiteBitsInv) = Inv_SuiteBits [];;
(Conversion_Int_SuiteBitsInv (9) : SuiteBitsInv) = Inv_SuiteBits [1;0;0;1];;
(Conversion_Int_SuiteBitsInv (1) : SuiteBitsInv) = Inv_SuiteBits [1];;


(*===================== fonction ==============================
 * NOM : Calcul_Lg_Utile
 * BUT : calculer la longueur utile d'une suite de bits inversee
 * TYPE DE RETOUR : int
 * PARAMETRES : si : SuiteBitsInv
 *==============================================================*)
let rec Calcul_Lg_Utile_Aux ((si : SuiteBitsInv), (b : Bit), (lg : int)) =
   if si = [] then lg
   else if (hd si) = b then
         Calcul_Lg_Utile_Aux ((tl si), b, lg+1)
      else lg;;

let Calcul_Lg_Utile ((si : SuiteBitsInv)) =
   if si = [] then 0
   else Calcul_Lg_Utile_Aux ((tl si), (hd si), 1);;

(Calcul_Lg_Utile (Inv_SuiteBits [1;0;0;1;0]) : int) = 1;;
(Calcul_Lg_Utile (Inv_SuiteBits [1;1;0;0;0]) : int) = 3;;
(Calcul_Lg_Utile (Inv_SuiteBits [0;0;0;0]) : int) = 4;;
(Calcul_Lg_Utile (Inv_SuiteBits [1;0;0;1;1]) : int) = 2;;
(Calcul_Lg_Utile (Inv_SuiteBits [0;0;0;1]) : int) = 1;;
(Calcul_Lg_Utile (Inv_SuiteBits [1;1;1;1;1]) : int) = 5;;


(*===================== fonction ==============================
 * NOM : Type_SuiteBits
 * BUT : tester si la suite de bits est une suite d'appui ou d'attaque-defense
 * TYPE DE RETOUR : bool (vrai si appui, faux sinon)
 * PARAMETRES : si : SuiteBitsInv
 * ATTENTION : n'utiliser cette fonction qu'avec une suite de bits NON VIDE!
 *==============================================================*)
let Type_SuiteBits ((si : SuiteBitsInv)) =
```

```
    (hd si) = 0;;

(Type_SuiteBits (Inv_SuiteBits [1;0;0;1;0]) : bool) = true;;
(Type_SuiteBits (Inv_SuiteBits [1;1;0;0;0]) : bool) = true;;
(Type_SuiteBits (Inv_SuiteBits [0;0;0;0]) : bool) = true;;
(Type_SuiteBits (Inv_SuiteBits [1;0;0;1;1]) : bool) = false;;
(Type_SuiteBits (Inv_SuiteBits [0;0;0;1]) : bool) = false;;
(Type_SuiteBits (Inv_SuiteBits [1;1;1;1;1]) : bool) = false;;


(*===================== fonction ==============================
 * NOM : Decaler_SuiteBits
 * BUT : supprimer les nb bits en début de suite
 * TYPE DE RETOUR : si : SuiteBitsInv
 * PARAMETRES : si : SuiteBitsInv et nb : int
 *==========================================================*)
let rec Decaler_SuiteBits((si : SuiteBitsInv), (nb : int)) =
    if nb = 0 then si
    else if si =[] then si
         else Decaler_SuiteBits ((tl si), nb -1);;

(Decaler_SuiteBits (Inv_SuiteBits [1;0;0;1;0], 1) : SuiteBitsInv) = [1;0;0;1];;
(Decaler_SuiteBits (Inv_SuiteBits [1;1;0;0;0],3) : SuiteBitsInv) = [1;1];;
(Decaler_SuiteBits (Inv_SuiteBits [0;0;0;0],4) : SuiteBitsInv) = [];;
(Decaler_SuiteBits (Inv_SuiteBits [1;0;0;1;1],2) : SuiteBitsInv) = [0;0;1];;
(Decaler_SuiteBits (Inv_SuiteBits [0;0;0;1],1) : SuiteBitsInv) = [0;0;0];;
(Decaler_SuiteBits (Inv_SuiteBits [1;1;1;1;1],5) : SuiteBitsInv) = [];;


(*===================== fonction ==============================
 * NOM : Compare_Appui_SuiteBits
 * BUT : comparer deux suites de bits sachant qu'il s'agit de suites d'appui
 * TYPE DE RETOUR : int (-1 si si1 < si2, 0 si si1 = si2, 1 si si1 > si2)
 * PARAMETRES : si1 : SuiteBitsInv et si2 : SuiteBitsInv
 *==========================================================*)
let rec Compare_Appui_SuiteBits((si1 : SuiteBitsInv),(si2 : SuiteBitsInv)) =
    if si1 = si2 then 0
    else let (lgu1 : int) = Calcul_Lg_Utile (si1)
         and (lgu2 : int) = Calcul_Lg_Utile (si2)
         in if lgu1 > lgu2 then 1
            else if lgu1 < lgu2 then -1
                 else let (si1bis : SuiteBitsInv) = Decaler_SuiteBits(si1, lgu1)
                      and (si2bis : SuiteBitsInv) = Decaler_SuiteBits(si2, lgu2)
                      in Compare_Attaque_Defense_SuiteBits (si1bis, si2bis)


and

(*===================== fonction ==============================
 * NOM : Compare_Attaque_Defense_SuiteBits
 * BUT : comparer deux suites de bits sachant qu'il s'agit de suites
 *        d'attaque ou de défense
 * TYPE DE RETOUR : int (-1 si si1 < si2, 0 si si1 = si2, 1 si si1 > si2)
 * PARAMETRES : si1 : SuiteBitsInv et si2 : SuiteBitsInv
 *==========================================================*)
    Compare_Attaque_Defense_SuiteBits ((si1 : SuiteBitsInv),(si2 : SuiteBitsInv)) =
    if si1 = si2 then 0
    else let (lgu1 : int) = Calcul_Lg_Utile (si1)
         and (lgu2 : int) = Calcul_Lg_Utile (si2)
         in if (lgu1 mod 2 = 0) && (lgu2 mod 2 = 0) then
               Compare_Defense_SuiteBits (si1, lgu1, si2, lgu2)
            else if (lgu1 mod 2 <> 0) && (lgu2 mod 2 <> 0) then
                    Compare_Attaque_SuiteBits (si1, lgu1, si2, lgu2)
                 else if (lgu1 mod 2 = 0) && (lgu2 mod 2 <> 0) then 1
                      else -1

and

(*===================== fonction ==============================
 * NOM : Compare_Attaque_SuiteBits
 * BUT : comparer deux suites de bits sachant qu'il s'agit de suites
```

```
 *        d'attaque
 * TYPE DE RETOUR : int (-1 si si1 < si2, 0 si si1 = si2, 1 si si1 > si2)
 * PARAMETRES : si1 : SuiteBitsInv et si2 : SuiteBitsInv
 *==============================================================*)
    Compare_Attaque_SuiteBits ((si1 : SuiteBitsInv),(lgu1 : int),
                               (si2 : SuiteBitsInv),(lgu2 : int)) =
    if lgu1 > lgu2 then 1
    else if lgu1 < lgu2 then −1
         else let (si1bis : SuiteBitsInv) = Decaler_SuiteBits(si1, lgu1)
              and (si2bis : SuiteBitsInv) = Decaler_SuiteBits(si2, lgu2)
              in −Compare_Appui_SuiteBits (si1bis, si2bis)

and

(*====================== fonction ==============================
 * NOM : Compare_Defense_SuiteBits
 * BUT : comparer deux suites de bits sachant qu'il s'agit de suites
 *        de defense
 * TYPE DE RETOUR : int (-1 si si1 < si2, 0 si si1 = si2, 1 si si1 > si2)
 * PARAMETRES : si1 : SuiteBitsInv et si2 : SuiteBitsInv
 *==============================================================*)
    Compare_Defense_SuiteBits ((si1 : SuiteBitsInv),(lgu1 : int),
                               (si2 : SuiteBitsInv),(lgu2 : int)) =
    if lgu1 > lgu2 then −1
    else if lgu1 < lgu2 then 1
         else let (si1bis : SuiteBitsInv) = Decaler_SuiteBits(si1, lgu1)
              and (si2bis : SuiteBitsInv) = Decaler_SuiteBits(si2, lgu2)
              in Compare_Appui_SuiteBits (si1bis, si2bis) ; ;



(Compare_Appui_SuiteBits (Inv_SuiteBits [1 ;0 ;0 ;1 ;0],Inv_SuiteBits [1 ;0 ;0 ;1 ;0]) : int) = 0 ; ;
(Compare_Appui_SuiteBits (Inv_SuiteBits [0 ;1 ;0],Inv_SuiteBits [1 ;1 ;0 ;0 ;0]) : int) = −1 ; ;
(Compare_Appui_SuiteBits (Inv_SuiteBits [1 ;1 ;0 ;0 ;0],Inv_SuiteBits [0 ;1 ;0]) : int) = 1 ; ;
(Compare_Appui_SuiteBits (Inv_SuiteBits [1 ;1 ;1 ;1 ;0],Inv_SuiteBits [0 ;1 ;0]) : int) = 1 ; ;
(Compare_Appui_SuiteBits (Inv_SuiteBits [0 ;0 ;1 ;0],Inv_SuiteBits [1 ;1 ;1 ;1 ;0]) : int) = −1 ; ;
(Compare_Appui_SuiteBits (Inv_SuiteBits [1 ;0 ;0 ;1 ;0],Inv_SuiteBits [1 ;0 ;1 ;0]) : int) = −1 ; ;
(Compare_Appui_SuiteBits (Inv_SuiteBits [0 ;1 ;0 ;1 ;0],Inv_SuiteBits [1 ;1 ;0 ;1 ;0]) : int) = 1 ; ;


(Compare_Attaque_Defense_SuiteBits (Inv_SuiteBits [1 ;0 ;0 ;1 ;0 ;1],
                                   Inv_SuiteBits [1 ;0 ;0 ;1 ;0 ;1]) : int) = 0 ; ;
(Compare_Attaque_Defense_SuiteBits (Inv_SuiteBits [1 ;0 ;0 ;1 ;1],
                                   Inv_SuiteBits [1 ;1 ;1]) : int) = 1 ; ;
(Compare_Attaque_Defense_SuiteBits (Inv_SuiteBits [1 ;1 ;1],
                                   Inv_SuiteBits [1 ;0 ;0 ;1 ;1]) : int) = −1 ; ;
(Compare_Attaque_Defense_SuiteBits (Inv_SuiteBits [1 ;1 ;1 ;1 ;0 ;1],
                                   Inv_SuiteBits [1 ;0 ;0 ;1 ;0 ;0 ;1]) : int) = 1 ; ;
(Compare_Attaque_Defense_SuiteBits (Inv_SuiteBits [1 ;0 ;0 ;1 ;0 ;0 ;1],
                                   Inv_SuiteBits [1 ;1 ;1 ;1 ;0 ;1]) : int) = −1 ; ;
(Compare_Attaque_Defense_SuiteBits (Inv_SuiteBits [0 ;1 ;0 ;1 ;1],
                                   Inv_SuiteBits [1 ;1 ;0 ;1 ;1]) : int) = −1 ; ;
(Compare_Attaque_Defense_SuiteBits (Inv_SuiteBits [0 ;1 ;0 ;1 ;0 ;1 ;1],
                                   Inv_SuiteBits  [1 ;0 ;1 ;0 ;1 ;1]) : int) = 1 ; ;
(Compare_Attaque_Defense_SuiteBits (Inv_SuiteBits [1 ;0 ;0 ;1 ;0 ;1],
                                   Inv_SuiteBits [1 ;1 ;1]) : int) = −1 ; ;
(Compare_Attaque_Defense_SuiteBits (Inv_SuiteBits [1 ;0 ;0 ;1 ;0 ;1],
                                   Inv_SuiteBits [1 ;0 ;0 ;1 ;1]) : int) = −1 ; ;

(******************************************
****** FONCTIONS INTERNES utilisees
****** pour le type TUPLE
*******************************************)
(*====================== fonction ==============================
 * NOM : Conversion_Tuple_SuiteBits_Tuple_Int
 * BUT : conversion d'une suite de bits en un entier
 * TYPE DE RETOUR : TupleInt
 * PARAMETRES : t Tuple
 *==============================================================*)
let rec Conversion_Tuple_SuiteBits_Tuple_Int ((t : Tuple)) =
    if t = [] then []
```

```
    else Conversion_SuiteBits_Int(hd t) : :
        Conversion_Tuple_SuiteBits_Tuple_Int(tl t) ; ;

(Conversion_Tuple_SuiteBits_Tuple_Int [[1 ;0 ;0 ;1 ;0] ;[ ] ;[1 ;0 ;0 ;1] ;[1] ;[0]] : TupleInt) =
                                        [18 ;0 ;9 ;1 ;0] ; ;
(Conversion_Tuple_SuiteBits_Tuple_Int [ ] : TupleInt) = [ ] ; ;


(*====================== fonction ===============================
 * NOM : Conversion_Tuple_Int_Tuple_SuiteBits
 * BUT : conversion d'un entier en une suite de bits
 * TYPE DE RETOUR : Tuple
 * PARAMETRES : tn TupleInt
 *==============================================================*)
let rec Conversion_Tuple_Int_Tuple_SuiteBits ((tn : TupleInt))=
    if tn = [ ] then [ ]
    else Conversion_Int_SuiteBits(hd tn) : :
        Conversion_Tuple_Int_Tuple_SuiteBits(tl tn) ; ;

(Conversion_Tuple_Int_Tuple_SuiteBits [18 ;0 ;9 ;1 ;0] : Tuple) =
                                        [[1 ;0 ;0 ;1 ;0] ;[ ] ;[1 ;0 ;0 ;1] ;[1] ;[ ]] ; ;
(Conversion_Tuple_Int_Tuple_SuiteBits [ ] : Tuple) = [ ] ; ;


(*====================== fonction ===============================
 * NOM : Partition
 * BUT : partitionner un tuple en 3 tuples
 *      - le premier contient les elements < ou = au pivot
 *      - le second contient le pivot
 *      - le troisieme contient les elements > au pivot
 * TYPE DE RETOUR : un triplet de tuples
 * PARAMETRES :
 *      - le tuple a partitionner (t : Tuple)
 *      - le pivot (s : SuiteBits)
 *      - le tuple en cours de construction pour les elements < ou = au pivot ([] au depart)
 *      - le tuple en cours de construction pour les elements > au pivot ([] au depart)
 *      - la fonction de comparaison a utiliser entre les elements de t
 *==============================================================*)
let rec partition ((t : Tuple), (s : SuiteBits), (t1 : Tuple), (t2 : Tuple),
            (fc_compar : SuiteBitsInv * SuiteBitsInv -> int)) =
    if t = [ ] then (t1, [s], t2)
    else if (fc_compar(Inv_SuiteBits(s),Inv_SuiteBits(hd t)) = 1) or
            (fc_compar(Inv_SuiteBits(s),Inv_SuiteBits(hd t)) = 0) then
            partition((tl t), s, (hd t) : :t1, t2, fc_compar)
        else
            partition((tl t), s, t1, (hd t) : :t2, fc_compar) ; ;

(partition ([[1 ;0 ;0 ;1 ;0 ;1] ;[1 ;0 ;0 ;1 ;1] ;[1 ;1 ;1] ;[1 ;1 ;1 ;1 ;0 ;1] ;[1 ;0 ;0 ;1 ;0 ;0 ;1] ;
                [0 ;1 ;0 ;1 ;1] ; [1 ;1 ;0 ;1 ;1] ; [0 ;1 ;0 ;1 ;0 ;1 ;1]],
            [1 ;0 ;0 ;1 ;0 ;1],
            [ ],
            [ ],
            Compare_Attaque_Defense_SuiteBits) : Tuple * Tuple * Tuple) =
([[1 ;0 ;0 ;1 ;0 ;0 ;1] ;[1 ;1 ;1 ;1 ;0 ;1] ;[1 ;0 ;0 ;1 ;0 ;1]],
[[1 ;0 ;0 ;1 ;0 ;1]],
[[0 ;1 ;0 ;1 ;0 ;1 ;1] ;[1 ;1 ;0 ;1 ;1] ; [0 ;1 ;0 ;1 ;1] ; [1 ;1 ;1] ;[1 ;0 ;0 ;1 ;1]]) ; ;

(partition ([[1 ;0 ;0 ;1 ;0] ;[0 ;1 ;0] ;[1 ;1 ;0 ;0 ;0] ;[0 ;0 ;1 ;0] ;[1 ;0 ;0 ;1 ;0] ;
                [0 ;1 ;0 ;1 ;0] ;[1 ;1 ;1 ;1 ;0] ;[1 ;0 ;1 ;0] ;[1 ;1 ;0 ;1 ;0]],
            [1 ;0 ;0 ;1 ;0],
            [ ],
            [ ],
            Compare_Appui_SuiteBits) : Tuple * Tuple * Tuple) =
([[1 ;0 ;0 ;1 ;0] ;[0 ;0 ;1 ;0] ;[1 ;0 ;0 ;1 ;0]],
[[1 ;0 ;0 ;1 ;0]],
[[1 ;1 ;0 ;1 ;0] ;[1 ;0 ;1 ;0] ;[1 ;1 ;1 ;1 ;0] ;[0 ;1 ;0 ;1 ;0] ;[1 ;1 ;0 ;0 ;0] ;[0 ;1 ;0]]) ; ;


(*====================== fonction ===============================
 * NOM : Tri_Tuple_Aux
 * BUT : effectue le tri rapide d'un tuple sachant que l'on connait
```

```
 *        la fonction de comparaison entre les elements du tuple
 * TYPE DE RETOUR : tuple trie par ordre croissant
 * PARAMETRES :
 *       - tuple a trier (t : Tuple) (ne contient pas de suite de bits vide)
 *       - fonction de comparaison entre les elements de t (fc_compar)
 *==================================================================*)
let rec Tri_Tuple_Aux ((t : Tuple),
              (fc_compar : SuiteBitsInv * SuiteBitsInv -> int)) =
    if t = [] then []
    else let triplet = partition((tl t), (hd t), [], [], fc_compar)
         in Tri_Tuple_Aux (champ1(triplet), fc_compar)
            @ champ2(triplet)
            @ Tri_Tuple_Aux (champ3(triplet), fc_compar);;

(Tri_Tuple_Aux ([[1;0;0;1;0;1];[1;0;0;1;1];[1;1;1];[1;1;1;1;0;1];[1;0;0;1;0;0;1];
                    [0;1;0;1;1]; [1;1;0;1;1]; [0;1;0;1;0;1;1]],
              Compare_Attaque_Defense_SuiteBits) : Tuple) =
[[1;0;0;1;0;0;1];[1;1;1;1;0;1];[1;0;0;1;0;1]; [1;1;1];
 [0;1;0;1;1];[0;1;0;1;0;1;1];[1;1;0;1;1]; [1;0;0;1;1]];;

(Tri_Tuple_Aux ([[1;0;0;1;0];[0;1;0];[1;1;0;0;0];[0;0;1;0];[1;0;0;1;0];
                   [0;1;0;1;0];[1;1;1;1;0];[1;0;1;0];[1;1;0;1;0]],
              Compare_Appui_SuiteBits) : Tuple) =
[[0;0;1;0];[1;0;0;1;0];[1;0;0;1;0];[0;1;0];[1;1;0;1;0];
 [1;0;1;0];[0;1;0;1;0];[1;1;1;1;0];[1;1;0;0;0]];;


(*====================== fonction ==============================
 * NOM : Tri_Tuple
 * BUT : effectue le tri rapide d'un tuple
 * TYPE DE RETOUR : tuple trie par ordre croissant
 * PARAMETRES : tuple a trier (t : Tuple)
 *==================================================================*)
let Tri_Tuple ((t : Tuple)) =
    if t = [] then []
    else if (hd t) <> [] then
            if Type_SuiteBits(Inv_SuiteBits (hd t)) then (* Appui *)
              Tri_Tuple_Aux (t, Compare_Appui_SuiteBits)
            else (* Attaque-Defense *)
              Tri_Tuple_Aux (t, Compare_Attaque_Defense_SuiteBits)
         else t;;

(Tri_Tuple ([[1;0;0;1;0;1];[1;0;0;1;1];[1;1;1];[1;1;1;1;0;1];[1;0;0;1;0;0;1];
                    [0;1;0;1;1]; [1;1;0;1;1]; [0;1;0;1;0;1;1]]) : Tuple) =
[[1;0;0;1;0;0;1];[1;1;1;1;0;1];[1;0;0;1;0;1]; [1;1;1];
 [0;1;0;1;1];[0;1;0;1;0;1;1];[1;1;0;1;1]; [1;0;0;1;1]];;

(Tri_Tuple ([[1;0;0;1;0];[0;1;0];[1;1;0;0;0];[0;0;1;0];[1;0;0;1;0];
                   [0;1;0;1;0];[1;1;1;1;0];[1;0;1;0];[1;1;0;1;0]]) : Tuple) =
[[0;0;1;0];[1;0;0;1;0];[1;0;0;1;0];[0;1;0];[1;1;0;1;0];
 [1;0;1;0];[0;1;0;1;0];[1;1;1;1;0];[1;1;0;0;0]];;


(*====================== fonction ==============================
 * NOM : Compare_Tuples_Aux
 * BUT : compare deux tuples differents de meme type (appui ou attaque-defense)
 *        en connaissant la fonction de comparaison de ce type
 * TYPE DE RETOUR : int (-1 si t1 < t2, 0 si t1 = t2, 1 si t1 > t2)
 * PARAMETRES :
 *       - t1 : Tuple
 *       - t2 : Tuple
 *       - fc_compar : fonction de comparaison des elements de ce type de tuple
 *==================================================================*)
let rec Compare_Tuples_Aux ((t1 : Tuple), (t2 : Tuple),
                         (fc_compar : SuiteBitsInv * SuiteBitsInv -> int)) =
    if t1 = [] then -1
    else if t2 = [] then 1
         else let s1 = (hd t1)
              and s2 = (hd t2)
              in if s1 = s2 then
                   Compare_Tuples_Aux ((tl t1), (tl t2), fc_compar)
```

```
        else
          fc_compar(Inv_SuiteBits(s1),Inv_SuiteBits(s2)) ;;

(Compare_Tuples_Aux ([[1 ;0 ;0] ;[0 ;0 ;0]],[[1 ;0 ;0] ;[1 ;0 ;0]],
                     Compare_Appui_SuiteBits) : int) = 1 ;;
(Compare_Tuples_Aux ([[0 ;0 ;0] ;[1 ;0 ;0]],[[1 ;0 ;0] ;[1 ;0 ;0]],
                     Compare_Appui_SuiteBits) : int) = 1 ;;
(Compare_Tuples_Aux ([[1 ;0 ;0] ;[0 ;0 ;0]],[[1 ;0 ;0]],
                     Compare_Appui_SuiteBits) : int) = 1 ;;
(Compare_Tuples_Aux ([[1 ;0 ;0] ;[1 ;0 ;0]],[[1 ;0 ;0] ;[0 ;0 ;0]],
                     Compare_Appui_SuiteBits) : int) = −1 ;;
(Compare_Tuples_Aux ([[1 ;0 ;0] ;[1 ;0 ;0]],[[0 ;0 ;0] ;[1 ;1 ;0]],
                     Compare_Appui_SuiteBits) : int) = −1 ;;
(Compare_Tuples_Aux ([[1 ;0 ;0] ;[1 ;0 ;0]],[[0 ;0 ;0] ;[1 ;1 ;0] ;[1 ;1 ;0]],
                     Compare_Appui_SuiteBits) : int) = −1 ;;
(Compare_Tuples_Aux ([[1 ;0 ;0] ;[1 ;0 ;0] ;[1 ;0 ;0] ;[1 ;0 ;0]],[[0 ;0 ;0] ;[1 ;1 ;0] ;[1 ;1 ;0]],
                     Compare_Appui_SuiteBits) : int) = −1 ;;

(Compare_Tuples_Aux ([[1 ;0 ;0 ;1] ;[0 ;0 ;0 ;1]],[[1 ;0 ;0 ;1] ;[1 ;0 ;0 ;1]],
                     Compare_Attaque_Defense_SuiteBits) : int) = −1 ;;
(Compare_Tuples_Aux ([[1] ;[1 ;0 ;0 ;1]],[[1 ;0 ;0 ;1] ;[1 ;0 ;0 ;1]],
                     Compare_Attaque_Defense_SuiteBits) : int) = 1 ;;
(Compare_Tuples_Aux ([[1] ;[0 ;0 ;1]],[[1 ;0 ;1]],
                     Compare_Attaque_Defense_SuiteBits) : int) = 1 ;;
(Compare_Tuples_Aux ([[1 ;0 ;0 ;1] ;[1 ;0 ;0 ;1]],[[1 ;0 ;0 ;1] ;[0 ;0 ;0 ;1]],
                     Compare_Attaque_Defense_SuiteBits) : int) = 1 ;;
(Compare_Tuples_Aux ([[1 ;0 ;0 ;1] ;[1 ;0 ;0 ;1]],[[0 ;0 ;0 ;1] ;[1 ;1]],
                     Compare_Attaque_Defense_SuiteBits) : int) = 1 ;;
(Compare_Tuples_Aux ([[1 ;0 ;0 ;1] ;[1 ;0 ;0 ;1] ;[1 ;1]],[[0 ;0 ;0 ;1] ;[1 ;1]],
                     Compare_Attaque_Defense_SuiteBits) : int) = 1 ;;
(Compare_Tuples_Aux ([[1 ;0 ;0 ;1] ;[1 ;0 ;0 ;1]],[[0 ;0 ;0 ;1] ;[1 ;1] ;[1 ;1]],
                     Compare_Attaque_Defense_SuiteBits) : int) = 1 ;;


(*====================== fonction ==============================
 * NOM : Compare_Tuples
 * BUT : compare deux tuples de meme type (appui ou attaque-defense)
 * TYPE DE RETOUR : int (-1 si t1 < t2, 0 si t1 = t2, 1 si t1 > t2)
 * PARAMETRES : t1 : Tuple et t2 : Tuple
 *============================================================*)
let Compare_Tuples((t1 : Tuple), (t2 : Tuple)) =
   if t1 = t2 then 0
   else if t1 = [] then −1
      else if t2 = [] then 1
         else if Type_SuiteBits (Inv_SuiteBits (hd t1)) then (* Appui *)
               Compare_Tuples_Aux (t1,t2, Compare_Appui_SuiteBits)
             else
               Compare_Tuples_Aux (t1,t2, Compare_Attaque_Defense_SuiteBits) ;;

(Compare_Tuples ([[1 ;0 ;0] ;[0 ;0 ;0]],[[1 ;0 ;0] ;[0 ;0 ;0]]) : int) = 0 ;;
(Compare_Tuples ([[1 ;0 ;0] ;[0 ;0 ;0]],[[1 ;0 ;0] ;[1 ;0 ;0]]) : int) = 1 ;;
(Compare_Tuples ([[0 ;0 ;0] ;[1 ;0 ;0]],[[1 ;0 ;0] ;[1 ;0 ;0]]) : int) = 1 ;;
(Compare_Tuples ([[1 ;0 ;0] ;[0 ;0 ;0]],[[1 ;0 ;0]]) : int) = 1 ;;
(Compare_Tuples ([[1 ;0 ;0] ;[1 ;0 ;0]],[[1 ;0 ;0] ;[0 ;0 ;0]]) : int) = −1 ;;
(Compare_Tuples ([[1 ;0 ;0] ;[1 ;0 ;0]],[[0 ;0 ;0] ;[1 ;1 ;0]]) : int) = −1 ;;
(Compare_Tuples ([[1 ;0 ;0] ;[1 ;0 ;0]],[[0 ;0 ;0] ;[1 ;1 ;0] ;[1 ;1 ;0]]) : int) = −1 ;;
(Compare_Tuples ([[1 ;0 ;0] ;[1 ;0 ;0] ;[1 ;0 ;0] ;[1 ;0 ;0]],[[0 ;0 ;0] ;[1 ;1 ;0] ;[1 ;1 ;0]]) : int) = −1 ;;

(Compare_Tuples ([[1 ;0 ;0 ;1] ;[0 ;0 ;0 ;1]],[[1 ;0 ;0 ;1] ;[0 ;0 ;0 ;1]]) : int) = 0 ;;
(Compare_Tuples ([[1 ;0 ;0 ;1] ;[0 ;0 ;0 ;1]],[[1 ;0 ;0 ;1] ;[1 ;0 ;0 ;1]]) : int) = −1 ;;
(Compare_Tuples ([[1] ;[1 ;0 ;0 ;1]],[[1 ;0 ;0 ;1] ;[1 ;0 ;0 ;1]]) : int) = 1 ;;
(Compare_Tuples ([[1] ;[0 ;0 ;1]],[[1 ;0 ;1]]) : int) = 1 ;;
(Compare_Tuples ([[1 ;0 ;0 ;1] ;[1 ;0 ;0 ;1]],[[1 ;0 ;0 ;1] ;[0 ;0 ;0 ;1]]) : int) = 1 ;;
(Compare_Tuples ([[1 ;0 ;0 ;1] ;[1 ;0 ;0 ;1]],[[0 ;0 ;0 ;1] ;[1 ;1]]) : int) = 1 ;;
(Compare_Tuples ([[1 ;0 ;0 ;1] ;[1 ;0 ;0 ;1] ;[1 ;1]],[[0 ;0 ;0 ;1] ;[1 ;1]]) : int) = 1 ;;
(Compare_Tuples ([[1 ;0 ;0 ;1] ;[1 ;0 ;0 ;1]],[[0 ;0 ;0 ;1] ;[1 ;1] ;[1 ;1]]) : int) = 1 ;;



(*******************************************
****** FONCTIONS INTERNES utilisees
```

```
****** pour le type VALTUPLEBIPO
*******************************************)
(*====================== fonction ==============================
 * NOM : Compare_ValTupleBipo_Qualitatif
 * BUT : comparer qualitativement 2 valeurs tuplees bipolaires sachant que :
 *       - elles sont differentes
 *       - elles ont le meme nombre d'appuis, le meme nombre d'attaques
 *       et le meme nombre de defenses
 * TYPE DE RETOUR : int (-1 si v1 < v2, 1 si v1 > v2 et 0 si v1 et v2
 *       sont incomparables)
 * PARAMETRES : v1 : ValTupleBipo et v2 : ValTupleBipo
 *==============================================================*)
let Compare_ValTupleBipo_Qualitatif((v1 : ValTupleBipo), (v2 : ValTupleBipo)) =
    let t1app = Tri_Tuple(champ1(v1))
    and t1att = Tri_Tuple(champ2(v1))
    and t1def = Tri_Tuple(champ3(v1))
    and t2app = Tri_Tuple(champ1(v2))
    and t2att = Tri_Tuple(champ2(v2))
    and t2def = Tri_Tuple(champ3(v2))
    in if (Compare_Tuples(t1app, t2app) >= 0) &&
          (Compare_Tuples(t1att, t2att) >= 0) &&
          (Compare_Tuples(t1def, t2def) >= 0) then 1
       else if (Compare_Tuples(t1app, t2app) <= 0) &&
               (Compare_Tuples(t1att, t2att) <= 0) &&
               (Compare_Tuples(t1def, t2def) <= 0) then -1
            else 0;;

(Compare_ValTupleBipo_Qualitatif(([[0;0]],[],[]),
                                 ([[0]],[],[])) : int) = 1;;
(Compare_ValTupleBipo_Qualitatif(([],[[1]],[]),
                                 ([],[[1;1;1]],[])) : int) = -1;;
(Compare_ValTupleBipo_Qualitatif(([],[],[[1;1]]),
                                 ([],[],[[1;1;1;1]])) : int) = 1;;
(Compare_ValTupleBipo_Qualitatif(([[0;0]],[[1;1;1]],[[1;1]]),
                                 ([[0]],[[1]],[[1;1;1;1]])) : int) = 1;;
(Compare_ValTupleBipo_Qualitatif(([[0]],[[1]],[[1;1;1;1]]),
                                 ([[0;0]],[[1;1;1]],[[1;1]])) : int) = -1;;
(Compare_ValTupleBipo_Qualitatif(([[0]],[[1;1;1]],[[1;1;1;1]]),
                                 ([[0;0]],[[1]],[[1;1]])) : int) = 0;;



(*******************************************
****** FONCTIONS EXTERNES utilisees
****** pour le type VALTUPLEBIPO
*******************************************)
(*====================== fonction ==============================
 * NOM : Conversion_ValTupleBipo_ValTupleIntBipo
 * BUT : convertir une valeur tuplee bipolaire composee de 3
 *        tuples de suites de bits en 3 tuples d'entiers
 * TYPE DE RETOUR : ValTupleInt
 * PARAMETRES : v : ValTupleBipo
 *==============================================================*)
let Conversion_ValTupleBipo_ValTupleIntBipo ((v : ValTupleBipo)) =
    (Conversion_Tuple_SuiteBits_Tuple_Int(champ1(v)),
     Conversion_Tuple_SuiteBits_Tuple_Int(champ2(v)),
     Conversion_Tuple_SuiteBits_Tuple_Int(champ3(v)));;

(Conversion_ValTupleBipo_ValTupleIntBipo ([[0;0];[1;0]],
                             [[1];[1;0;1];[1;0;1;1;1]],
                             [[0;1;1]]) : ValTupleIntBipo) = ([0;2],[1;5;23],[3]);;

(*====================== fonction ==============================
 * NOM : Conversion_ValTupleIntBipo_ValTupleBipo
 * BUT : convertir un triplet de tuples d'entier en une valeur
 *        tuplee bipolaire composee de 3 tuples de suites de bits
 * TYPE DE RETOUR : ValTupleBipo
 * PARAMETRES : vn : ValTupleIntBipo
 *==============================================================*)
let Conversion_ValTupleIntBipo_ValTupleBipo ((vn : ValTupleIntBipo)) =
    (Conversion_Tuple_Int_Tuple_SuiteBits(champ1(vn)),
```

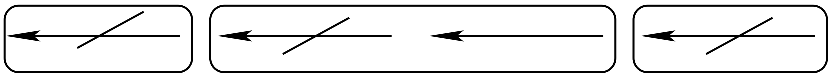
```
        Conversion_Tuple_Int_Tuple_SuiteBits(champ2(vn)),
        Conversion_Tuple_Int_Tuple_SuiteBits(champ3(vn))) ;;

(Conversion_ValTupleIntBipo_ValTupleBipo ([0 ;2],[1 ;5 ;23],[3]) : ValTupleBipo) =
        ([[] ;[1 ;0]],
         [[1] ;[1 ;0 ;1] ;[1 ;0 ;1 ;1 ;1]],
         [[1 ;1]]) ;;


(*===================== fonction =============================
 * NOM : Compare_ValTupleBipo
 * BUT : comparer 2 valeurs tuplees bipolaires sachant qu'elles sont differentes
 * TYPE DE RETOUR : int (-1 si v1 < v2, 1 si v1 > v2 et 0 si v1 et v2
 *       sont incomparables)
 * PARAMETRES : v1 : ValTupleBipo et v2 : ValTupleBipo
 *============================================================*)
let Compare_ValTupleBipo ((v1 : ValTupleBipo), (v2 : ValTupleBipo)) =
    let n1app = list_length(champ1(v1))
    and n1att = list_length(champ2(v1))
    and n1def = list_length(champ3(v1))
    and n2app = list_length(champ1(v2))
    and n2att = list_length(champ2(v2))
    and n2def = list_length(champ3(v2))
    in if (n1app = n2app) && (n1att = n2att) && (n1def = n2def) then
            Compare_ValTupleBipo_Qualitatif(v1,v2)
       else if (n1app >= n2app) && (n1att <= n2att) && (n1def >= n2def) then 1
            else if (n1app <= n2app) && (n1att >= n2att) && (n1def <= n2def) then -1
                 else 0 ;;

(Compare_ValTupleBipo (([[0]],[[1]],[[1 ;1]]),([],[[1]],[])) : int) = 1 ;;
(Compare_ValTupleBipo (([[0]],[[1]],[[1 ;1]]),([[0] ;[0]],[[1]],[])) : int) = 0 ;;
(Compare_ValTupleBipo (([[0]],[[1] ;[1]],[[1 ;1]]),([[0]],[[1]],[])) : int) = 0 ;;
(Compare_ValTupleBipo (([[0]],[[1]],[[1 ;1]]),([],[[1]],[[1 ;1] ;[1 ;1 ;1 ;1]])) : int) = 0 ;;
(Compare_ValTupleBipo (([[0]],[[1]],[]),([],[[1]],[])) : int) = 1 ;;
(Compare_ValTupleBipo (([[0]],[],[[1 ;1]]),([],[],[[1 ;1]])) : int) = 1 ;;
(Compare_ValTupleBipo (([[0] ;[0 ;0]],[[1]],[[1 ;1]]),([],[[1] ;[1 ;1 ;1]],[])) : int) = 1 ;;
(Compare_ValTupleBipo (([[0] ;[0 ;0]],[[1]],[]),([[0 ;0] ;[0 ;0]],[[1]],[])) : int) = -1 ;;

(Compare_ValTupleBipo (([[0 ;0]],[],[]),
                       ([[0]],[],[])) : int) = 1 ;;
(Compare_ValTupleBipo (([],[[1]],[]),
                       ([],[[1 ;1 ;1]],[])) : int) = -1 ;;
(Compare_ValTupleBipo (([],[],[[1 ;1]]),
                       ([],[],[[1 ;1 ;1 ;1]])) : int) = 1 ;;
(Compare_ValTupleBipo (([[0 ;0]],[[1 ;1 ;1]],[[1 ;1]]),
                       ([[0]],[[1]],[[1 ;1 ;1 ;1]])) : int) = 1 ;;
(Compare_ValTupleBipo (([[0]],[[1]],[[1 ;1 ;1 ;1]]),
                       ([[0 ;0]],[[1 ;1 ;1]],[[1 ;1]])) : int) = -1 ;;
(Compare_ValTupleBipo (([[0]],[[1 ;1 ;1]],[[1 ;1 ;1 ;1]]),
                       ([[0 ;0]],[[1]],[[1 ;1]])) : int) = 0 ;;
```

Annexe B

Une autre évaluation graduelle globale dans un SABP

Cette seconde évaluation graduelle globale n'est citée ici qu'à titre de perspective et est donc évoquée de manière très succincte. Il s'agit ici d'exploiter une autre idée déjà évoquée en section 8.1 page 49 dans l'identification de la nature des branches pour définir une autre évaluation : chaque chemin entre une feuille et l'argument A va être découpé en partant de A en attaques appuyées et séquences d'appui de longueur maximale. Dans ce cas, tout appui précédant une attaque dans le chemin est "intégré" à cette attaque (il augmente la qualité de l'attaque et "disparaît").

Donc sur l'exemple suivant, on a une attaque attaquée par une attaque appuyée, cette attaque appuyée étant à son tour attaquée :



B.1 Notion de branches

On va ici définir précisément le découpage que l'on effectue pour déterminer la nature de chacune des branches menant à un argument A donné.

Découpage Soit $\mathcal{C} = A_1 - \ldots - A_n$ un chemin d'une feuille vers A. On découpe $\mathcal{C}$ de la manière suivante :

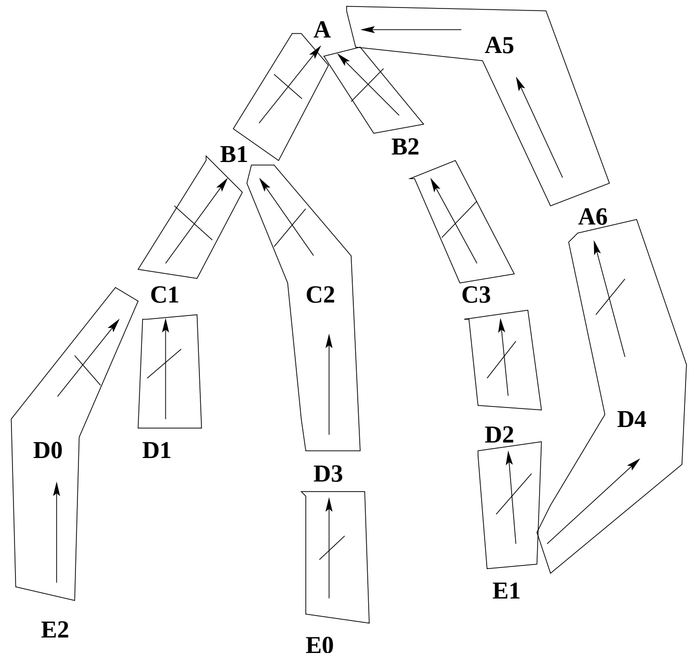
– si $\mathcal{R}_{n-1} = \mathcal{R}_{app}$ alors :

1. on cherche A_{k_1} tel que $A_{k_1} - \ldots - A_n$ soit une séquence d'appuis de longueur maximale (donc soit $A_1 = A_{k_1}$, soit $A_{k_1-1} \mathcal{R}_x A_{k_1}$ avec $\mathcal{R}_x = \mathcal{R}_{att}$) ;
2. si $A_1 \neq A_{k_1}$, on cherche A_{k_2} tel que $A_{k_2} - \ldots - A_{k_1-1}$ soit une attaque appuyée de longueur maximale (donc soit $A_1 = A_{k_2}$, soit $A_{k_2-1} \mathcal{R}_x A_{k_2}$ avec $\mathcal{R}_x = \mathcal{R}_{att}$) ;
3. si $A_1 \neq A_{k_2}$, on réitère le processus 2-3 jusqu'à atteindre un $A_{k_j} = A_1$.

– si $\mathcal{R}_{n-1} = \mathcal{R}_{att}$ alors :

1. on cherche A_{k_1} tel que $A_{k_1} - \ldots - A_n$ soit une attaque appuyée de longueur maximale (donc soit $A_1 = A_{k_1}$, soit $A_{k_1-1} \mathcal{R}_x A_{k_1}$ avec $\mathcal{R}_x = \mathcal{R}_{att}$) ;
2. si $A_1 \neq A_{k_1}$, on réitère le processus 1-2 jusqu'à atteindre un $A_{k_j} = A_1$.

Exemple de découpage



Propriété 18 *Soit $\mathcal{C}$ un chemin d'une feuille vers A. Dans le découpage défini précédemment, il y a au plus une séquence d'appui et si elle existe,c'est la première séquence rencontrée (celle qui part de A).*

Preuve : Soit $\mathcal{C} = A_1 - \ldots - A_n$ un chemin d'une feuille vers A. Dans le découpage défini ci-dessus, si $\mathcal{R}_{n-1} = \mathcal{R}_{att}$, on n'obtient que des attaques appuyées. Donc le seul cas où l'on peut obtenir des séquences d'appuis est le cas où $\mathcal{R}_{n-1} = \mathcal{R}_{app}$.

Si $\mathcal{R}_{n-1} = \mathcal{R}_{app}$, la première séquence rencontrée est une séquence d'appuis (la séquence $A_{k_1} - \ldots - A_n$) ; comme elle est de longueur maximale :

- soit $A_{k_1} = A_1$ et alors il y a une seule séquence d'appuis dans le découpage ;
- soit $A_{k_1} \neq A_1$ et alors $A_{k_1-1}\mathcal{R}_{att}A_{k_1}$. Donc, à partir de là, on n'obtient plus que des attaques appuyées.

Donc, il y a toujours au plus une seule séquence d'appuis dans le découpage d'une branche. □

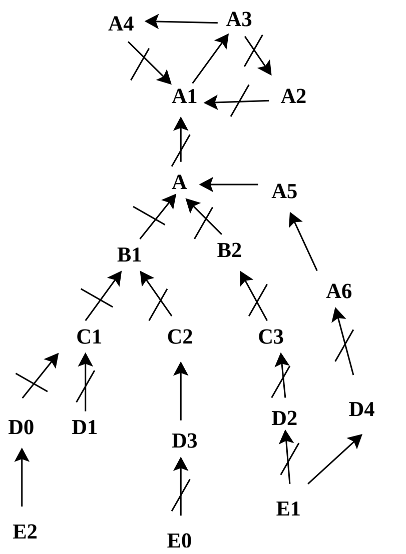
Définition 42 (Branches d'appui, d'attaque et de défense d'un argument) *Soit $A \in \mathcal{A}$.*

Branche d'appui *Un chemin $\mathcal{C}$ dans $\mathcal{G}$ d'une feuille vers A est une* branche d'appui *pour A ssi la séquence d'extrémité A est une séquence d'appui.*

Branche d'attaque *Un chemin $\mathcal{C}$ dans $\mathcal{G}$ d'une feuille vers A est une* branche d'attaque *pour A ssi la séquence d'extrémité A est une attaque appuyée et qu'il y a un nombre impair d'attaques appuyées dans $\mathcal{C}$.*

Branche de défense *Un chemin $\mathcal{C}$ dans $\mathcal{G}$ d'une feuille vers A est une* branche de défense *pour A ssi la séquence d'extrémité A est une attaque appuyée et qu'il y a un nombre pair d'attaques appuyées dans $\mathcal{C}$.*

Toutes ces notions sont illustrées sur l'exemple suivant :



Sur ce graphe $\mathcal{G}$, on a donc :
- 3 branches d'attaque pour A :
  $D_1 - C_1 - B_1 - A$,
  $E_2 - D_0 - C_1 - B_1 - A$ et
  $E_0 - D_3 - C_2 - B_1 - A$,
- 1 branche de défense pour A :
  $E_1 - D_2 - C_3 - B_2 - A$,
- 1 branche d'appui pour A :
  $E_1 - D_4 - A_6 - A_5 - A$.

B.2 Valeur d'une branche

Ici, tout reste à faire et ce qui suit n'est que l'énumération de quelques idées :
- 1 étiquette pour chaque séquence obtenue lors du découpage de la branche en distinguant séquence d'appuis et attaque appuyée (par exemple 1 pour les appuis et -1 pour les attaques appuyées).
- On peut aussi tenir compte du nombre d'appuis (dans la séquence d'appuis et dans l'attaque appuyée).
- Dans tous les cas, la valeur d'une branche = tuple de valeurs.

B.3 Comparaison des deux approches globales

Si on compare cette approche globale avec celle proposée en section 8.2 page 50, on constate rapidement que les différences entre ces 2 approches portent sur la définition des branches d'attaque et de défense.
Sur l'exemple fourni en sections 8.2.1 page 50 et B.1 page 79, on constate qu'il y a 5 branches distinctes et qu'elles n'ont pas toujours la même nature suivant le découpage choisi :

Branche	Nature de la branche avec	
	l'approche de la section 8.2 page 50	l'approche de la section 8.2 page 50
$E_2 - D_0 - C_1 - B_1 - A$	attaque	attaque
$D_1 - C_1 - B_1 - A$	attaque	attaque
$E_0 - D_3 - C_2 - B_1 - A$	***défense***	***attaque***
$E_1 - D_2 - C_3 - B_2 - A$	défense	défense
$E_1 - D_4 - A_6 - A_5 - A$	appui	appui

B.4 Conclusion sur cette seconde approche globale

Cette annexe ne donne que quelques idées pour définir une nouvelle évaluation graduelle globale. Il reste beaucoup de travail à faire pour la finaliser et cela fait partie (entre autres) des perspectives (voir section 8.4 page 65).

Annexe C

Preuves

Preuve :(de la propriété 2 page 16)
Soit A et B deux arguments, on a :

1. Montrons que la relation $\texttt{attaque}_1$ n'est pas symétrique. Prenons les arguments suivants :
   - $A = (\{p, \neg r, \neg q\}, p \wedge \neg r \wedge \neg q)$
   - $B = (\{p \to q, q \to r\}, p \to r)$

   Ici, $A \overset{1}{\nrightarrow} B$ (car $S_A \vdash \neg(p \to q)$) et l'inverse est faux.
2. $A \overset{3}{\nrightarrow} B$
   $\Rightarrow \neg C_A \equiv C_B$
   $\Rightarrow \neg C_B \equiv C_A$
   $\Rightarrow B \overset{3}{\nrightarrow} A$.
3. Montrons que la relation $\texttt{attaque}_2$ n'est pas symétrique. Prenons les arguments suivants :
   - $A = (\{p, p \to o, o \to v\}, v)$
   - $B = (\{p, p \to o, p \to \neg v\}, \neg(o \to v))$

   Ici, $B \overset{2}{\nrightarrow} A$ et l'inverse est faux.
4. Montrons que la relation $\texttt{attaque}_4$ n'est pas symétrique. Prenons les arguments suivants :
   - $A = (\{a, a \to b, b \to c\}, c)$
   - $B = (\{d, d \to \neg b\}, \neg b)$

   Ici, $A \overset{4}{\nrightarrow} B$ et l'inverse est faux.
5. $A \overset{1}{\nrightarrow} B$
   $\Rightarrow \exists X$ une formule telle que $S_A \vdash \neg X$ et $X \in S_B$.
   Construisons l'argument C tel que $C_C = \neg X$ et $S_C \subseteq S_A$ avec $S_C \vdash \neg X$ et S_C minimal pour l'obtention de $\neg X$.

   $\Rightarrow \exists C$ un argument tel que $C \overset{2}{\nrightarrow} B$.
6. $A \overset{4}{\nrightarrow} B$
   $\Rightarrow \exists X$ une formule telle que $S_A \vdash \neg X$ et $X \equiv C_B$.
   Construisons l'argument C tel que $C_C = \neg X$ et $S_C \subseteq S_A$ avec $S_C \vdash \neg X$ et S_C minimal pour l'obtention de $\neg X$.

   $\Rightarrow \exists C$ un argument tel que $C \overset{3}{\nrightarrow} B$.
7. $A \overset{2}{\nrightarrow} B$
   $\Rightarrow \neg C_A \in S_B$
   $\Rightarrow \exists X / S_A \vdash \neg X$ et $X \in S_B$ (c'est $X = \neg C_A$)
   $\Rightarrow A \overset{1}{\nrightarrow} B$.

8. $A \overset{3}{\not\to} B$
$\Rightarrow \neg C_A \equiv C_B$
$\Rightarrow \exists X / S_A \vdash \neg X$ et $X \equiv C_B$ (c'est $X = \neg C_A$)
$\Rightarrow A \overset{4}{\not\to} B.$

$\square$

Preuve :(de la propriété 3 page 18)
$A \overset{2}{\not\to} B \Rightarrow S_A \vdash C_A$ et $\neg C_A \in S_B \Rightarrow S_A \vdash C_A$ et $S_B \vdash \neg C_A \Rightarrow S_A \cup S_B \vdash \bot.$
De même, $A \overset{3}{\not\to} B \Rightarrow S_A \vdash C_A$ et $\neg C_A \equiv C_B \Rightarrow S_A \vdash C_A$ et $S_B \vdash \neg C_A \Rightarrow S_A \cup S_B \vdash \bot.$ $\square$

Preuve :(de la propriété 4 page 18)
Soit A et B deux arguments, on a :

1. Montrons que la relation `appuie`$_1$ n'est pas symétrique. Posons $A = (\{a, a \to b, b \to c\}, c)$ et $B = (\{b, b \to d\}, d)$. On a $A \xrightarrow{1} B$ mais pas $B \xrightarrow{1} A$.
2. Montrons que la relation `appuie`$_2$ n'est pas symétrique. Posons $A = (\{a, a \to b\}, b)$ et $B = (\{b, b \to d\}, d)$. On a $A \xrightarrow{2} B$ mais pas $B \xrightarrow{2} A$.
3. Montrons que la relation `appuie`$_3$ est symétrique :
   $A \xrightarrow{3} B$
   $\Rightarrow C_A \equiv C_B$
   $\Rightarrow C_B \equiv C_A$
   $\Rightarrow B \xrightarrow{3} A.$
4. Montrons que la relation `appuie`$_4$ n'est pas symétrique. Posons $A = (\{a, a \to b, b \to c\}, c)$ et $B = (\{d, d \to b\}, b)$. On a $A \xrightarrow{4} B$ mais pas $B \xrightarrow{4} A$.
5. $A \xrightarrow{4} B$
   $\Rightarrow \exists X$ tel que $S_A \vdash X$ et $X \equiv C_B$
   Posons C tel que $C_C = X$ et $S_C \subseteq S_A$ avec $S_C \vdash X$ et S_C minimal pour l'obtention de X
   $\Rightarrow \exists C$ un argument tel que $C \xrightarrow{3} B$.
6. $A \xrightarrow{1} B$
   $\Rightarrow \exists X$ tel que $S_A \vdash X$ et $X \in S_B$
   Posons C tel que $C_C = X$ et $S_C \subseteq S_A$ avec $S_C \vdash X$ et S_C minimal pour l'obtention de X
   $\Rightarrow \exists C$ un argument tel que $C \xrightarrow{2} B$.
7. $A \xrightarrow{2} B$
   $\Rightarrow C_A \in S_B$
   $\Rightarrow \exists X / S_A \vdash X$ et $X \in S_B$ (c'est $X = C_A$)
   $\Rightarrow A \xrightarrow{1} B.$
8. $A \xrightarrow{3} B$
   $\Rightarrow C_A \equiv C_B$
   $\Rightarrow \exists X / S_A \vdash X$ et $X \equiv C_B$ (c'est $X = C_A$)
   $\Rightarrow A \xrightarrow{4} B.$

$\square$

Preuve :(de la propriété 6 page 31)
Soit $B \in \mathcal{A}$ tel que S appuie B. Si $B \notin S$, comme S est stable alors S attaque B. On aurait donc S qui appuie et qui attaque B, ce qui est en contradiction avec l'hypothèse "S sûr". Donc $B \in S$. $\square$

Preuve :(de la propriété 7 page 31)
On sait déjà que si S attaque directement A alors S attaque A (voir la définition 23 page 29). Montrons maintenant que si S attaque A alors S attaque directement A. Supposons que S attaque A mais pas directement. Alors $\exists B \in S$ tel que un des chemins entre B et A est une attaque appuyée ne se réduisant pas à un seul arc d'attaque. On a donc :
$B_1 \rightarrow B_2 \rightarrow \ldots \rightarrow B_n \not\rightarrow A$ avec $B_1 = B$.
Or, B appuie B_n donc $B_n \in S$ puisque S est fermé pour la relation $\mathcal{R}_{app}$. Donc S attaque directement A. □

Preuve :(de la propriété 8 page 31)
Supposons que S ne soit pas sûr. Il existe donc $B \in \mathcal{A}$ tel que S attaque B et, soit appuie B, soit $B \in S$. Or, S étant fermé pour la relation $\mathcal{R}_{app}$, si S appuie B alors $B \in S$. Donc, il existe $B \in S$ tel que S attaque B. Ce qui est en contradiction avec "S sans conflit". Donc, S est sûr.
□

Preuve :(de la propriété 10 page 32)
Soit S une extension stable, donc sans conflit. Soit $A \in S$ attaqué par B, alors $B \notin S$ puisque S sans conflit. Donc S attaque B puisque S est stable. □

Preuve :(de la propriété 13 page 36)
- $\forall x, y,\ g(x, \alpha) \geq g(\alpha, y)$:
  on sait que $\forall x, y,\ x \geq \alpha$ et $y \geq \alpha$; donc, $g(x, y) \geq g(\alpha, y)$ (croissance de g pour x) et $g(x, y) \leq g(x, \alpha)$ (décroissance de g pour y) ; donc $\forall x, y,\ g(x, \alpha) \geq g(\alpha, y)$.
- $g(\beta, \alpha) = V_{\mathrm{Max}}$ et $g(\alpha, \beta) = V_{\mathrm{Min}}$:
  on sait que $\forall x, y,\ \beta \geq x \geq \alpha$ et $\beta \geq y \geq \alpha$; donc $g(x, y) \leq g(\beta, y)$ et $g(x, y) \geq g(x, \beta)$; de même, $g(\beta, y) \leq g(\beta, \alpha)$ (cas où $x = \beta$ et $y \geq \alpha$) et $g(x, \beta) \geq g(\alpha, \beta)$ (cas où $y = \beta$ et $x \geq \alpha$) ; on a donc $\forall x, y,\ g(\alpha, \beta) \leq g(x, y) \leq g(\beta, \alpha)$; donc $g(\beta, \alpha) = V_{\mathrm{Max}}$ et $g(\alpha, \beta) = V_{\mathrm{Min}}$.
- Si ${\mathcal{R}_{att}}^{-}(A) = {\mathcal{R}_{app}}^{-}(A) = \varnothing$ alors $v(A) = g(\alpha, \alpha)$:
  puisque ${\mathcal{R}_{att}}^{-}(A) = {\mathcal{R}_{app}}^{-}(A) = \varnothing$, on a donc $v(A) = g(h_{app}(), h_{att}()) = g(\alpha, \alpha)$.
- Si ${\mathcal{R}_{att}}^{-}(A) \neq \varnothing$ et ${\mathcal{R}_{app}}^{-}(A) = \varnothing$ alors $v(A) = g(\alpha, y) \leq g(\alpha, \alpha)$ pour $y \geq \alpha$:
  puisque ${\mathcal{R}_{app}}^{-}(A) = \varnothing$ et que ${\mathcal{R}_{att}}^{-}(A) \neq \varnothing$, posons ${\mathcal{R}_{att}}^{-}(A) = \{B_1, \ldots, B_n\}$; on a alors $v(A) = g(h_{app}(), h_{att}(v(B_1), \ldots, v(B_n)))$; posons $y = h_{att}(v(B_1), \ldots, v(B_n))$; on a alors $v(A) = g(\alpha, y)$; d'autre part, on sait que, α étant la valeur minimale de $\mathcal{H}_{att}$, $y \geq \alpha$ et $g(x, y)$ est décroissante en y ; on obtient alors $v(A) = g(\alpha, y) \leq g(\alpha, \alpha)$.
- Si ${\mathcal{R}_{att}}^{-}(A) = \varnothing$ et ${\mathcal{R}_{app}}^{-}(A) \neq \varnothing$ alors $v(A) = g(x, \alpha) \geq g(\alpha, \alpha)$ pour $x \geq \alpha$:
  puisque ${\mathcal{R}_{att}}^{-}(A) = \varnothing$ et que ${\mathcal{R}_{app}}^{-}(A) \neq \varnothing$, posons ${\mathcal{R}_{app}}^{-}(A) = \{C_1, \ldots, C_p\}$; on a alors $v(A) = g(h_{app}(v(C_1), \ldots, v(C_p)), h_{att}())$; posons $x = h_{app}(v(C_1), \ldots, v(C_p))$; on a alors $v(A) = g(x, \alpha)$; d'autre part, on sait que, α étant la valeur minimale de $\mathcal{H}_{app}$, $x \geq \alpha$ et $g(x, y)$ est croissante en x ; on obtient alors $v(A) = g(x, \alpha) \geq g(\alpha, \alpha)$.

□

Preuve :(de la propriété 14 page 36)
Soit A tel que ${\mathcal{R}_{att}}^{-}(A) = \{B_1, \ldots, B_n\}$ et ${\mathcal{R}_{app}}^{-}(A) = \{C_1, \ldots, C_p\}$.

Respect de P1 : $h_{app}(v(C_1), \ldots, v(C_p))$ évalue l'appui direct de A et $h_{att}(v(B_1), \ldots, v(B_n))$ évalue la contrariété directe de A. Donc, $v(A)$ est bien fonction (*via* g) de l'évaluation de l'appui direct et de la contrariété directe sur A.

Respect de P2 : La qualité de l'appui est évaluée par h_{app}. D'après les propriétés de la fonction g si $h_{app}(v(C_1), \ldots, v(C_p))$ augmente alors $v(A)$ augmente. De plus, si la qualité d'un des appuis augmente (par exemple $v(C_i)$ augmente) alors d'après la propriété 1 imposée à h_{app}, $h_{app}(v(C_1), \ldots, v(C_p))$ augmente aussi, et donc $v(A)$.
Le raisonnement est le même pour l'attaque.

Respect de P3 : Il repose sur la deuxième propriété imposée aux fonctions h_{app} et h_{att}.

□

Preuve : (de la propriété 15 page 59)
Montrons que la relation est totale. Il suffit de constater que les suites de $\mathcal{S}$ sont soit des suites représentant des appuis, soit des suites représentant des attaques-défenses et que toutes les

valeurs de retour des algorithmes (voir algorithmes 1 page 57, 2 page 57, 3 page 58 et 4 page 58) sont prises en compte dans la définition 39 page 58. Donc, $\forall s_1, s_2$, soit on a $s_1 \preceq s_2$, soit on a $s_2 \preceq s_1$.

Montrons que $\preceq$ est réflexive. Soit une suite de bits s, si s est une valeur d'appui, appliquons la fonction COMPARER-APPUI(s, s). D'après l'algorithme 1 page 57, le résultat de la fonction est 0. Donc $s = s$, donc $s \preceq s$. De même, si s est une valeur d'attaque ou de défense, appliquons la fonction COMPARER-ATTAQUE-DÉFENSE(s, s) qui renverra aussi, d'après l'algorithme 2 page 57, le résultat 0. Donc $s = s$, donc $s \preceq s$.

Montrons que $\preceq$ est transitive. Soit trois suites de bits s_1, s_2 et s_3 telles que $s_3 \preceq s_2$ et $s_2 \preceq s_1$.
- **Cas 1 :** soit $s_1 = s_2$, soit $s_2 = s_3$, soit $s_1 = s_3$, on a alors $s_3 \preceq s_1$.
- **Cas 2 :** on n'a ni $s_1 = s_2$, ni $s_2 = s_3$, ni $s_1 = s_3$. Donc, $s_3 \prec s_2$ et $s_2 \prec s_1$ avec s_1, s_2 et s_3 différentes 2 à 2. On a alors plusieurs sous-cas :
  - **Cas 2.1 :** s_1, s_2 et s_3 ne sont pas de même nature (certaines sont des suites de bits représentant des appuis et d'autres des attaques-défenses). On a alors :
    - soit s_1 et s_2 représentent des appuis et s_3 une attaque-défense, alors d'après la définition 39 page 58, on a bien $s_3 \prec s_1$;
    - soit s_1 représente un appui et s_2, s_3 des attaques-défenses, alors d'après la définition 39 page 58, on a bien $s_3 \prec s_1$;
    - soit s_1 représente une attaque-défense et s_2, s_3 des appuis, alors d'après la définition 39 page 58, on ne peut avoir $s_2 \prec s_1$; donc cas impossible !
    - soit s_1 et s_2 représentent des attaques-défenses et s_3 un appui, alors d'après la définition 39 page 58, on ne peut avoir $s_3 \prec s_2$; donc cas impossible !
  - **Cas 2.2 :** s_1, s_2 et s_3 sont de même nature (toutes sont soit des suites de bits représentant des appuis et soit des suites de bits représentant des attaques-défenses). On considère que chaque s_i est constituée de la droite vers la gauche de n_i paquets des bits (dans un même paquet tous les bits sont égaux[1]), chacun étant noté X_j^i avec $j = 1 \dots n_i$ et dont la longueur est notée $l(X_j^i)$. Remarquons que si s_i représente un appui alors X_1^i ne contient que des 0 et $l(X_1^i) = l_{app}$ de la branche de valeur s_i. De même, si s_i représente une attaque-défense alors X_1^i ne contient que des 1 et $l(X_1^i) = l_{att}$ de la branche de valeur s_i.
    Nous allons faire un raisonnement par récurrence sur le nombre de paquets constituant s_1 en considérant que s_2 et s_3 ont un nombre de paquets quelconque.
    - **Cas 2.2.1 : Cas de base de la démonstration par récurrence** s_1 ne contient qu'un seul paquet. Pour des besoins techniques liés à la preuve dans le cas où l'on a $n+1$ paquets dans s_1, nous allons montrer que $s_3 \prec s_2$ et $s_2 \prec s_1 \Rightarrow s_3 \prec s_1$ et que $s_1 \prec s_2$ et $s_2 \prec s_3 \Rightarrow s_1 \prec s_3$. Examinons chaque possibilité :
      - d'abord $s_3 \prec s_2$ et $s_2 \prec s_1 \Rightarrow s_3 \prec s_1$:
        s_1, s_2 et s_3 représentent des appuis :
        puisque $s_2 \prec s_1$, on a soit $l_1 > l_2$, soit $l_1 = l_2$ et ...
        puisque $s_3 \prec s_2$, on a soit $l_2 > l_3$, soit $l_2 = l_3$ et ...
        Si $l_1 > l_2$ et ($l_2 > l_3$ ou $l_2 = l_3$) alors $l_1 > l_3$, donc $s_3 \prec s_1$ (algo 1 page 57).
        Si $l_1 = l_2$ et $l_2 > l_3$ alors $l_1 > l_3$, donc $s_3 \prec s_1$ (algo 1 page 57).
        Si $l_1 = l_2$ et $l_2 = l_3$ alors on décale et on obtient s'_1 vide, s'_2 non vide, s'_3 non vide (puisque les 3 suites sont différentes 2 à 2) et appel de l'algo 2 page 57. On a alors $l'_1 = 0$ et $l'_2 \neq 0$, $l'_3 \neq 0$. Si l'_3 est pair, comme $l'_1 = 0$, on a $l'_1 < l'_3$, donc $s'_3 \prec s'_1$, donc $s_3 \prec s_1$ (algo 4 page 58). Et si l'_3 est impair, comme l'_1 est pair, on a $s'_3 \prec s'_1$, donc $s_3 \prec s_1$ (algo 2 page 57).
        s_1, s_2 et s_3 représentent des attaques :
        puisque $s_2 \prec s_1$, on a soit $l_1 > l_2$, soit $l_1 = l_2$ et ...
        puisque $s_3 \prec s_2$, on a soit $l_2 > l_3$, soit $l_2 = l_3$ et ...
        Si $l_1 > l_2$ et ($l_2 > l_3$ ou $l_2 = l_3$) alors $l_1 > l_3$, donc $s_3 \prec s_1$ (algo 3 page 58).
        Si $l_1 = l_2$ et $l_2 > l_3$ alors $l_1 > l_3$, donc $s_3 \prec s_1$ (algo 3 page 58).
        Si $l_1 = l_2$ et $l_2 = l_3$ alors on décale et on obtient s'_1 vide, s'_2 non vide, s'_3 non vide

[1]Donc un paquet ne contient que des 1 ou (exclusif) que des 0.

(puisque les 3 suites sont différentes 2 à 2) et appel de l'algo 1 page 57. On a alors $l'_1 = 0$ et $l'_2 \neq 0$, $l'_3 \neq 0$. Donc, on a $l'_1 < l'_3$, donc $s'_3 \succ s'_1$, donc $s_3 \prec s_1$ (algos 1 page 57 et 3 page 58).

s_1, s_2 et s_3 représentent des défenses :
puisque $s_2 \prec s_1$, on a soit $l_1 < l_2$, soit $l_1 = l_2$ et ...
puisque $s_3 \prec s_2$, on a soit $l_2 < l_3$, soit $l_2 = l_3$ et ...
Si $l_1 < l_2$ et ($l_2 < l_3$ ou $l_2 = l_3$) alors $l_1 < l_3$, donc $s_3 \prec s_1$ (algo 4 page 58).
Si $l_1 = l_2$ et $l_2 < l_3$ alors $l_1 < l_3$, donc $s_3 \prec s_1$ (algo 4 page 58).
Si $l_1 = l_2$ et $l_2 = l_3$ alors on décale et on obtient s'_1 vide, s'_2 non vide, s'_3 non vide (puisque les 3 suites sont différentes 2 à 2) et appel de l'algo 1 page 57. On a alors $l'_1 = 0$ et $l'_2 \neq 0$, $l'_3 \neq 0$. Donc, on a $l'_1 < l'_2$, donc $s'_1 \prec s'_2$ (algo 1 page 57), donc $s_1 \prec s_2$, ce qui est en contradiction avec l'hypothèse de départ ($s_2 \prec s_1$). Donc cas impossible.

– puis $s_1 \prec s_2$ et $s_2 \prec s_3 \Rightarrow s_1 \prec s_3$:

s_1, s_2 et s_3 représentent des appuis :
puisque $s_1 \prec s_2$, on a soit $l_1 < l_2$, soit $l_1 = l_2$ et ...
puisque $s_2 \prec s_3$, on a soit $l_2 < l_3$, soit $l_2 = l_3$ et ...
Si $l_1 < l_2$ et ($l_2 < l_3$ ou $l_2 = l_3$) alors $l_1 < l_3$, donc $s_1 \prec s_3$ (algo 1 page 57).
Si $l_1 = l_2$ et $l_2 < l_3$ alors $l_1 < l_3$, donc $s_1 \prec s_3$ (algo 1 page 57).
Si $l_1 = l_2$ et $l_2 = l_3$ alors on décale et on obtient s'_1 vide, s'_2 non vide, s'_3 non vide (puisque les 3 suites sont différentes 2 à 2) et appel de l'algo 2 page 57. On a alors $l'_1 = 0$ et $l'_2 \neq 0$, $l'_3 \neq 0$. Pour que $s_1 \prec s_2$, il faut que $s'_1 \prec s'_2$; donc, sachant que $l'_1 = 0$, il faut que l'_2 soit pair et que l'algo 4 page 58 renvoie -1. Or, avec $l'_1 = 0$ et $l'_2 \neq 0$, on a $l'_1 < l'2$ donc $s'_2 \prec s'_1$, donc $s_2 \prec s_1$, ce qui est en contradiction avec l'hypothèse prise. Donc cas impossible.

s_1, s_2 et s_3 représentent des attaques :
puisque $s_1 \prec s_2$, on a soit $l_1 < l_2$, soit $l_1 = l_2$ et ...
puisque $s_2 \prec s_3$, on a soit $l_2 < l_3$, soit $l_2 = l_3$ et ...
Si $l_1 < l_2$ et ($l_2 < l_3$ ou $l_2 = l_3$) alors $l_1 < l_3$, donc $s_1 \prec s_3$ (algo 3 page 58).
Si $l_1 = l_2$ et $l_2 < l_3$ alors $l_1 < l_3$, donc $s_1 \prec s_3$ (algo 3 page 58).
Si $l_1 = l_2$ et $l_2 = l_3$ alors on décale et on obtient s'_1 vide, s'_2 non vide, s'_3 non vide (puisque les 3 suites sont différentes 2 à 2) et appel de l'algo 1 page 57. On a alors $l'_1 = 0$ et $l'_2 \neq 0$, $l'_3 \neq 0$. Or, cela signifie que l'on a $l'_1 < l'_2$, donc $s'_2 \succ s'_1$, donc $s_2 \prec s_1$ (algos 1 page 57 et 3 page 58) ce qui est en contradiction avec l'hypothèse prise. Donc cas impossible.

s_1, s_2 et s_3 représentent des défenses :
puisque $s_1 \prec s_2$, on a soit $l_1 > l_2$, soit $l_1 = l_2$ et ...
puisque $s_2 \prec s_3$, on a soit $l_2 > l_3$, soit $l_2 = l_3$ et ...
Si $l_1 > l_2$ et ($l_2 > l_3$ ou $l_2 = l_3$) alors $l_1 > l_3$, donc $s_1 \prec s_3$ (algo 4 page 58).
Si $l_1 = l_2$ et $l_2 > l_3$ alors $l_1 > l_3$, donc $s_1 \prec s_3$ (algo 4 page 58).
Si $l_1 = l_2$ et $l_2 = l_3$ alors on décale et on obtient s'_1 vide, s'_2 non vide, s'_3 non vide (puisque les 3 suites sont différentes 2 à 2) et appel de l'algo 1 page 57. On a alors $l'_1 = 0$ et $l'_2 \neq 0$, $l'_3 \neq 0$. Donc, on a $l'_1 < l'_3$, donc $s'_1 \prec s'_3$ (algo 1 page 57), donc $s_1 \prec s_3$.

– **Cas 2.2.1 : Cas général de la démonstration par récurrence** Posons comme hypothèse de récurrence

$$\text{si } s_1 \text{ contient } n_1 \leq n \text{ paquets, alors}$$
$$(s_2 \prec s_1 \text{ et } s_3 \prec s_2 \Rightarrow s_3 \prec s_1) \text{ ET } (s_1 \prec s_2 \text{ et } s_2 \prec s_3 \Rightarrow s_1 \prec s_3).$$

Étudions le cas d'un s_1 ayant $n + 1$ paquets, sachant que s_2 a n_2 paquets et s_3 a n_3 paquets (n_2 et n_3 étant quelconques). Examinons chaque possibilité :

– d'abord $s_2 \prec s_1$ et $s_3 \prec s_2 \Rightarrow s_3 \prec s_1$:

s_1, s_2 et s_3 représentent des appuis :
puisque $s_2 \prec s_1$, on a soit $l_1 > l_2$, soit $l_1 = l_2$ et ...
puisque $s_3 \prec s_2$, on a soit $l_2 > l_3$, soit $l_2 = l_3$ et ...
Si $l_1 > l_2$ et ($l_2 > l_3$ ou $l_2 = l_3$) alors $l_1 > l_3$, donc $s_3 \prec s_1$ (algo 1 page 57).
Si $l_1 = l_2$ et $l_2 > l_3$ alors $l_1 > l_3$, donc $s_3 \prec s_1$ (algo 1 page 57).

Si $l_1 = l_2$ et $l_2 = l_3$ alors on décale et on obtient s'_1 avec n paquets, s'_2 avec $n_2 - 1$ paquets, s'_3 avec $n_3 - 1$ paquets (puisque les 3 suites sont différentes 2 à 2) et appel de l'algo 2 page 57. Comme $s_2 \prec s_1$, alors $s'_2 \prec s'_1$ et, de même, comme $s_3 \prec s_2$, alors $s'_3 \prec s'_2$. On a donc $s'_2 \prec s'_1$ et $s'_3 \prec s'_2$ avec s'_1 de n paquets et s'_2, s'_3 ayant un nombre quelconque de paquets. La partie 1 de l'hypothèse de récurrence s'applique et on obtient que $s'_3 \prec s'_1$, donc $s_3 \prec s_1$ (algos 2 page 57 et 1 page 57).

s_1, s_2 et s_3 représentent des attaques :
puisque $s_2 \prec s_1$, on a soit $l_1 > l_2$, soit $l_1 = l_2$ et ...
puisque $s_3 \prec s_2$, on a soit $l_2 > l_3$, soit $l_2 = l_3$ et ...
Si $l_1 > l_2$ et ($l_2 > l_3$ ou $l_2 = l_3$) alors $l_1 > l_3$, donc $s_3 \prec s_1$ (algo 3 page 58).
Si $l_1 = l_2$ et $l_2 > l_3$ alors $l_1 > l_3$, donc $s_3 \prec s_1$ (algo 3 page 58).
Si $l_1 = l_2$ et $l_2 = l_3$ alors on décale et on obtient s'_1 avec n paquets, s'_2 avec $n_2 - 1$ paquets, s'_3 avec $n_3 - 1$ paquets (puisque les 3 suites sont différentes 2 à 2) et appel de l'algo 1 page 57. Comme $s_2 \prec s_1$, alors $s'_2 \succ s'_1$ et, de même, comme $s_3 \prec s_2$, alors $s'_3 \succ s'_2$. On a donc $s'_1 \prec s'_2$ et $s'_2 \prec s'_3$ avec s'_1 de n paquets et s'_2, s'_3 ayant un nombre quelconque de paquets. La partie 2 de l'hypothèse de récurrence s'applique et on obtient que $s'_1 \prec s'_3$, donc $s_3 \prec s_1$ (algos 1 page 57 et 3 page 58).

s_1, s_2 et s_3 représentent des défenses :
puisque $s_2 \prec s_1$, on a soit $l_1 < l_2$, soit $l_1 = l_2$ et ...
puisque $s_3 \prec s_2$, on a soit $l_2 < l_3$, soit $l_2 = l_3$ et ...
Si $l_1 < l_2$ et ($l_2 < l_3$ ou $l_2 = l_3$) alors $l_1 < l_3$, donc $s_3 \prec s_1$ (algo 3 page 58).
Si $l_1 = l_2$ et $l_2 < l_3$ alors $l_1 < l_3$, donc $s_3 \prec s_1$ (algo 3 page 58).
Si $l_1 = l_2$ et $l_2 = l_3$ alors on décale et on obtient s'_1 avec n paquets, s'_2 avec $n_2 - 1$ paquets, s'_3 avec $n_3 - 1$ paquets (puisque les 3 suites sont différentes 2 à 2) et appel de l'algo 1 page 57. Comme $s_2 \prec s_1$, alors $s'_2 \prec s'_1$ et, de même, comme $s_3 \prec s_2$, alors $s'_3 \prec s'_2$. On a donc $s'_2 \prec s'_1$ et $s'_3 \prec s'_2$ avec s'_1 de n paquets et s'_2, s'_3 ayant un nombre quelconque de paquets. La partie 1 de l'hypothèse de récurrence s'applique et on obtient que $s'_3 \prec s'_1$, donc $s_3 \prec s_1$ (algos 1 page 57 et 4 page 58).

- d'abord $s_1 \prec s_2$ et $s_2 \prec s_3 \Rightarrow s_1 \prec s_3$:

  s_1, s_2 et s_3 représentent des appuis :
  puisque $s_1 \prec s_2$, on a soit $l_1 < l_2$, soit $l_1 = l_2$ et ...
  puisque $s_2 \prec s_3$, on a soit $l_2 < l_3$, soit $l_2 = l_3$ et ...
  Si $l_1 < l_2$ et ($l_2 < l_3$ ou $l_2 = l_3$) alors $l_1 < l_3$, donc $s_1 \prec s_3$ (algo 1 page 57).
  Si $l_1 = l_2$ et $l_2 < l_3$ alors $l_1 < l_3$, donc $s_1 \prec s_3$ (algo 1 page 57).
  Si $l_1 = l_2$ et $l_2 = l_3$ alors on décale et on obtient s'_1 avec n paquets, s'_2 avec $n_2 - 1$ paquets, s'_3 avec $n_3 - 1$ paquets (puisque les 3 suites sont différentes 2 à 2) et appel de l'algo 2 page 57. Comme $s_1 \prec s_2$, alors $s'_1 \prec s'_2$ et, de même, comme $s_2 \prec s_3$, alors $s'_2 \prec s'_3$. On a donc $s'_1 \prec s'_2$ et $s'_2 \prec s'_3$ avec s'_1 de n paquets et s'_2, s'_3 ayant un nombre quelconque de paquets. La partie 2 de l'hypothèse de récurrence s'applique et on obtient que $s'_1 \prec s'_3$, donc $s_1 \prec s_3$ (algos 2 page 57 et 1 page 57).

  s_1, s_2 et s_3 représentent des attaques :
  puisque $s_1 \prec s_2$, on a soit $l_1 < l_2$, soit $l_1 = l_2$ et ...
  puisque $s_2 \prec s_3$, on a soit $l_2 < l_3$, soit $l_2 = l_3$ et ...
  Si $l_1 < l_2$ et ($l_2 < l_3$ ou $l_2 = l_3$) alors $l_1 < l_3$, donc $s_1 \prec s_3$ (algo 3 page 58).
  Si $l_1 = l_2$ et $l_2 < l_3$ alors $l_1 < l_3$, donc $s_1 \prec s_3$ (algo 3 page 58).
  Si $l_1 = l_2$ et $l_2 = l_3$ alors on décale et on obtient s'_1 avec n paquets, s'_2 avec $n_2 - 1$ paquets, s'_3 avec $n_3 - 1$ paquets (puisque les 3 suites sont différentes 2 à 2) et appel de l'algo 1 page 57. Comme $s_1 \prec s_2$, alors $s'_2 \succ s'_1$ et, de même, comme $s_2 \prec s_3$, alors $s'_2 \succ s'_3$. On a donc $s'_2 \prec s'_1$ et $s'_3 \prec s'_2$ avec s'_1 de n paquets et s'_2, s'_3 ayant un nombre quelconque de paquets. La partie 1 de l'hypothèse de récurrence s'applique et on obtient que $s'_3 \prec s'_1$, donc $s_1 \prec s_3$ (algos 1 page 57 et 3 page 58).

  s_1, s_2 et s_3 représentent des défenses :
  puisque $s_1 \prec s_2$, on a soit $l_1 > l_2$, soit $l_1 = l_2$ et ...
  puisque $s_2 \prec s_3$, on a soit $l_2 > l_3$, soit $l_2 = l_3$ et ...
  Si $l_1 > l_2$ et ($l_2 > l_3$ ou $l_2 = l_3$) alors $l_1 > l_3$, donc $s_1 \prec s_3$ (algo 3 page 58).
  Si $l_1 = l_2$ et $l_2 > l_3$ alors $l_1 > l_3$, donc $s_1 \prec s_3$ (algo 3 page 58).

Si $l_1 = l_2$ et $l_2 = l_3$ alors on décale et on obtient s'_1 avec n paquets, s'_2 avec $n_2 - 1$ paquets, s'_3 avec $n_3 - 1$ paquets (puisque les 3 suites sont différentes 2 à 2) et appel de l'algo 1 page 57. Comme $s_1 \prec s_2$, alors $s'_1 \prec s'_2$ et, de même, comme $s_2 \prec s_3$, alors $s'_2 \prec s'_3$. On a donc $s'_1 \prec s'_2$ et $s'_2 \prec s'_3$ avec s'_1 de n paquets et s'_2, s'_3 ayant un nombre quelconque de paquets. La partie 2 de l'hypothèse de récurrence s'applique et on obtient que $s'_1 \prec s'_3$, donc $s_1 \prec s_3$ (algos 1 page 57 et 4 page 58).

□

Preuve : (de la propriété 16 page 59)
Montrons que la relation est totale. Il suffit de constater que les relations $=$ et $\prec$ servant à construire $\leq_{lex}$ sont des relations totales. Donc, $\forall t_1, t_2$, soit on a $t_1 \leq_{lex} t_2$, soit on a $t_2 \leq_{lex} t_1$.

Montrons que $\leq_{lex}$ est réflexive. Soit $t = (x_1, \ldots, x_n) \in \mathcal{T}$. $t = t$, donc $t =_{lex} t$ (d'après la définition 40 page 59 de $=_{lex}$), donc $t \leq_{lex} t$.

Montrons que $\leq_{lex}$ est transitive. Posons $t_1 = (x_1, \ldots, x_m)$, $t_2 = (y_1, \ldots, y_n)$ et $t_3 = (z_1, \ldots, z_p)$. On a $t_1 \leq_{lex} t_2$ et $t_2 \leq_{lex} t_3$. Remarquons tout de suite que si on a soit $t_1 =_{lex} t_2$, soit $t_2 =_{lex} t_3$, la preuve de la transitivité est immédiate. Étudions donc le cas où $t_1 \neq_{lex} t_2$ et $t_2 \neq_{lex} t_3$. On a donc $t_1 <_{lex} t_2$ et $t_2 <_{lex} t_3$. On a alors 4 cas possibles :
- Soit $\exists n \geq i \geq 1$ tel que $\forall j < i$, $x_j = y_j$ et $m = i - 1$ et $\exists p \geq i' \geq 1$ tel que $\forall j < i'$, $y_j = z_j$ et $n = i' - 1$, alors on a $m = i - 1 < n = i' - 1$, et donc $\exists p \geq i'' \geq 1$ $(i'' = i)$ tel que $\forall j < i''$, $x_j = z_j$ et $m = i'' - 1$, donc $t_1 <_{lex} t_3$.
- Soit $\exists n \geq i \geq 1$ tel que $\forall j < i$, $x_j = y_j$ et $m = i - 1$ et $\exists p \geq i' \geq 1$ tel que $\forall j < i'$, $y_j = z_j$ et $y_{i'} \prec z_{i'}$, alors on a deux cas possibles :
  - soit $i > i'$ et alors, en posant $i'' = i'$, on obtient $\exists p \geq i'' \geq 1$ tel que $\forall j < i''$, $x_j = z_j$ et $x_{i''} = y_{i''} \prec z_{i''}$ donc $t_1 <_{lex} t_3$;
  - soit $i' > i$ et alors, en posant $i'' = i$, on obtient $\exists p \geq i'' \geq 1$ tel que $\forall j < i''$, $x_j = z_j$ et $m = i'' - 1$ donc $t_1 <_{lex} t_3$.
- Soit $\exists n \geq i \geq 1$ tel que $\forall j < i$, $x_j = y_j$ et $x_i \prec y_i$ et $\exists p \geq i' \geq 1$ tel que $\forall j < i'$, $y_j = z_j$ et $n = i' - 1$, alors on a deux cas possibles :
  - soit $i > i'$ et alors cela signifie qu'on a en même temps $y_{i'}$ qui n'existe pas et y_i qui existe alors que $i > i'$! cela est impossible, ce cas ne peut pas se produire ;
  - soit $i' > i$ et alors, en posant $i'' = i$, on obtient $\exists p \geq i'' \geq 1$ tel que $\forall j < i''$, $x_j = z_j$ et $x_{i''} \prec y_{i''} = z_{i''}$ donc $t_1 <_{lex} t_3$.
- Soit $\exists n \geq i \geq 1$ tel que $\forall j < i$, $x_j = y_j$ et $x_i \prec y_i$ et $\exists p \geq i' \geq 1$ tel que $\forall j < i'$, $y_j = z_j$ et $y_{i'} \prec z_{i'}$, alors on a deux cas possibles :
  - soit $i > i'$ et alors, en posant $i'' = i'$, on obtient $\exists p \geq i'' \geq 1$ tel que $\forall j < i''$, $x_j = z_j$ et $x_{i''} = y_{i''} \prec z_{i''}$ donc $t_1 <_{lex} t_3$;
  - soit $i' > i$ et alors, en posant $i'' = i$, on obtient $\exists p \geq i'' \geq 1$ tel que $\forall j < i''$, $x_j = z_j$ et $x_{i''} \prec y_{i''} = z_{i''}$ donc $t_1 <_{lex} t_3$.

□

Preuve :(de la propriété 17 page 60)
Montrons que $\preceq$ est réflexive. Soit une valeur tuplée bipolaire v, on a $v = v$, donc $v \preceq v$.

Montrons que $\preceq$ est transitive. Soit trois valeurs tuplées bipolaires v_1, v_2 et v_3 telles que $v_2 \preceq v_1$ et $v_3 \preceq v_2$. Remarquons tout d'abord que si $v_1 = v_2$ ou $v_2 = v_3$ alors la transitivité est immédiate. Étudions donc le cas où $v_1 \neq v_2$ et $v_2 \neq v_3$. On pose donc que $v_2 \prec v_1$ et $v_3 \prec v_2$. On a alors plusieurs cas possibles correspondant aux cas identifiés dans les algorithmes 5 page 59 et 6 page 60 :
- soit (cas 1 de l'algorithme 5 page 59)
  $nbapp_1 = nbapp_2$ et $nbatt_1 = nbatt_2$ et $nbdef_1 = nbdef_2$ et
  $nbapp_2 = nbapp_3$ et $nbatt_2 = nbatt_3$ et $nbdef_2 = nbdef_3$ et alors on a (cas 1.1 de l'algorithme 6 page 60) :
  $v_{app}1 \geq_{lex} v_{app}2$ et $v_{att}1 \leq_{lex} v_{att}2$ et $v_{def}1 \geq_{lex} v_{def}2$ et
  $v_{app}2 \geq_{lex} v_{app}3$ et $v_{att}2 \leq_{lex} v_{att}3$ et $v_{def}2 \geq_{lex} v_{def}3$;

comme la relation $\leq_{lex}$ est un pré-ordre total, on a alors :
$v_{app}1 \geq_{lex} v_{app}3$ et $v_{att}1 \leq_{lex} v_{att}3$ et $v_{def}1 \geq_{lex} v_{def}3$; donc $v_3 \prec v_1$.

– soit (cas 2 de l'algorithme 5 page 59)
$nbapp_1 \geq nbapp_2$ et $nbatt_1 \leq nbatt_2$ et $nbdef_1 \geq nbdef_2$ (sachant qu'on ne peut pas avoir en même temps $nbapp_1 = nbapp_2$ et $nbatt_1 = nbatt_2$ et $nbdef_1 = nbdef_2$) et
$nbapp_2 \geq nbapp_3$ et $nbatt_2 \leq nbatt_3$ et $nbdef_2 \geq nbdef_3$ (sachant qu'on ne peut pas avoir en même temps $nbapp_2 = nbapp_3$ et $nbatt_2 = nbatt_3$ et $nbdef_2 = nbdef_3$) et on a alors :
$nbapp_1 \geq nbapp_3$ et $nbatt_1 \leq nbatt_3$ et $nbdef_1 \geq nbdef_3$ (sachant qu'on ne peut pas avoir en même temps $nbapp_1 = nbapp_3$ et $nbatt_1 = nbatt_3$ et $nbdef_1 = nbdef_3$) ; donc $v_3 \prec v_1$.

$\square$

Bibliographie

[AC98] Amgoud (Leila) et Cayrol (Claudette). – On the acceptability of arguments in preference-based argumentation. *In : Proc. of the 14^{th} Uncertainty in Artificial Intelligence*, éd. par Cooper (G. F.) et Moral (S.). pp. 1–7. – Madison, Wisconsin, 1998.

[ACLS04] Amgoud (Leila), Cayrol (Claudette) et Lagasquie-Schiex (Marie-Christine). – On the bipolarity in argumentation frameworks. *In : Proc. of the 10^{th} NMR workshop (Non Monotonic Reasoning), Uncertainty Framework subworkshop*, éd. par Delgrande (J.) et Schaub (T.), pp. 1–9. – Whistler, BC, Canada, 2004.

[Amg99] Amgoud (Leila). – *Contribution à l'intégration des préférences dans le raisonnement argumentatif.* – Toulouse, Thèse de PhD, Université Paul Sabatier, juillet 1999.

[BH01] Besnard (Philippe) et Hunter (Anthony). – A logic-based theory of deductive arguments. *Artificial Intelligence*, vol. 128 (1-2), 2001, pp. 203–235.

[CLS01] Cayrol (Claudette) et Lagasquie-Schiex (Marie-Christine). – *Évaluation graduelle d'arguments.* – Rapport de recherche n2001-18-R, France, Institut de Recherche en Informatique de Toulouse (I.R.I.T.), novembre 2001.

[CLS02] Cayrol (Claudette) et Lagasquie-Schiex (Marie-Christine). – Gradual handling of contradiction in argumentation frameworks. *In : Proc. of the 9^{th} IPMU (Information Processing and Management of Uncertainty in Knowledge-Based Systems)*, pp. 83–90. – Annecy, France, 2002.

[CLS03a] Cayrol (Claudette) et Lagasquie-Schiex (Marie-Christine). – *Critique et amélioration de l'évaluation graduelle par tuples pour le traitement des circuits.* – Rapport de recherche n2003-13-R, France, Institut de Recherche en Informatique de Toulouse (I.R.I.T.), septembre 2003.

[CLS03b] Cayrol (Claudette) et Lagasquie-Schiex (Marie-Christine). – Gradual acceptability in argumentation systems. *In : Proc. of the 3^{rd} CMNA (International workshop on computational models of natural argument)*, pp. 55–58. – Acapulco, Mexique, 2003.

[CLS03c] Cayrol (Claudette) et Lagasquie-Schiex (Marie-Christine). – Gradual handling of contradiction in argumentation frameworks. *In : Intelligent Systems for Information Processing : From representation to Applications*, éd. par Bouchon-Meunier (B.), L.Foulloy et Yager (R.R.), chap. Reasoning, pp. 179–190. – Elsevier, 2003.

[Dou02] Doutre (Sylvie). – *Autour de la sémantique préférée des systèmes d'argumentation.* – Thèse, Université Paul Sabatier, IRIT, 2002.

[Dun95] Dung (Phan Minh). – On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artificial Intelligence*, vol. 77, 1995, pp. 321–357.

[EGFK93] Elvang-Goransson (Morten), Fox (John) et Krause (Paul). – Dialectic reasoning with inconsistent information. *In : Proc. of the 9^{th} UAI*, éd. par Heckerman (David) et Mamdani (Abe). pp. 114–121. – Washington, DC, 1993.

[Gay04] Gay (Dominique). – Représentation et évaluation de la bipolarité en argumentation. – Rapport de DEA. DEA RCFR, Université Paul Sabatier Toulouse, 2004.

[JV99] Jakobovits (Hadassa) et Vermeir (Dick). – Robust semantics for argumentation frameworks. *Journal of logic and computation*, vol. 9(2), 1999, pp. 215–261.

[KAEF95] Krause (Paul), Ambler (Simon), Elvang (Morten) et Fox (John). – A logic of argumentation for reasoning under uncertainty. *Computational Intelligence*, vol. 11 (1), 1995, pp. 113–131.

[KP01] Karacapilidis (Nikos) et Papadias (Dimitris). – Computer supported argumentation and collaborative decision making : the HERMES system. *Information systems*, vol. 26, n4, 2001, pp. 259–277.

[Par97] Parsons (Simon). – Normative argumentation and qualitative probability. *In : Proc. of the first International Joint Conference on Qualitative and quantitative practical reasoning, ECSQARU-FAPR, LNAI 1244*, pp. 466–480. – Germany, 1997.

[Pol92] Pollock (J. L.). – How to reason defeasibly. *Artificial Intelligence*, vol. 57, 1992, pp. 1–42.

[PS97] Prakken (Henry) et Sartor (Giovanni). – Argument-based extended logic programming with defeasible priorities. *Journal of Applied Non-Classical Logics*, vol. 7, 1997, pp. 25–75.

[SL92] Simari (G.R.) et Loui (R.P.). – A mathematical treatment of defeasible reasoning and its implementation. *Artificial Intelligence*, vol. 53, 1992, pp. 125–157.

[Ver02] Verheij (Bart). – On the existence and multiplicity of extension in dialectical argumentation. *In : Proceedings of the 9th International Workshop on Non-Monotonic Reasoning (NMR'2002)*, éd. par Benferhat (Salem) et Giunchiglia (Enrico), pp. 416–425.